LENGUAJE Ñ_ MATEMÁTICO

LAURA DEL PINO DÍAZ– GRADO EN INGENIERÍA INFORMÁTICA – MENCIÓN EN COMPUTACIÓN

Procesadores de Lenguaje

ÍNDICE

Tabla de contenido

INTRODUCCIÓN

En este documento definiremos la creación de un nuevo lenguaje de programación denominado Ñ matemático, derivado del lenguaje Ñ presentado anteriormente con el compañero. Este lenguaje estará escrito en español y tendrá una base la cual estará formada por funcionalidades de diferentes lenguajes de programación, entre los que destacan MATLAB, Java y Korn Shell.

Por ello en la siguiente sección modelaremos como va hacer nuestro lenguaje de programación con el objetivo de poder crear nuestro compilador.

DEFINICIÓN DEL LENGUAJE Ñ_MATEMÁTICO

TIPOS DE DATOS

En esta sección definiremos el modelo de nuestro lenguaje. El lenguaje Ñ estará formado por tres tipos de datos:

- **REAL**. Este tipo lo utilizaremos para codificar cualquier número.
- **VECTOR**. Este tipo lo utilizaremos para almacenar un conjunto de números en un espacio de memoria contigua y que podremos referenciar mediante un nombre. El índice de la primera posición es 0.

Tipos	Ejemplos
REAL	5, -1 , 3.05
VECTOR	[1,-2,3.05,-50.10],[H,o,l,a]

También es posible utilizar las ristras de caracteres, pero su uso está limitado a mostrar por pantalla mensajes más amigables al usuario. Véase ENTRADA-SALIDA

DECLARACIÓN DE VARIABLES REALES

Para declarar una variable de tipo “REAL” se puede utilizar las siguientes gramáticas:

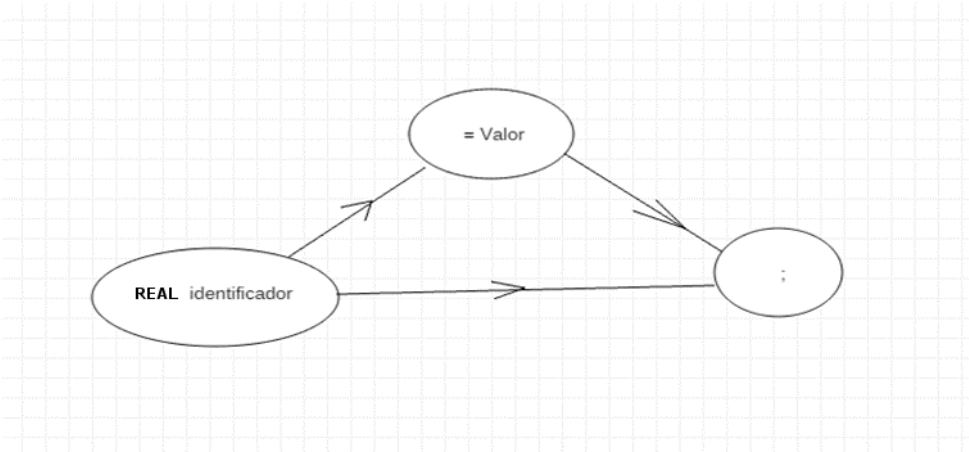

Ilustración 1: declaración de una variable del tipo REAL

Para declarar una variable de tipo "VECTOR" se puede utilizar las siguientes gramáticas:

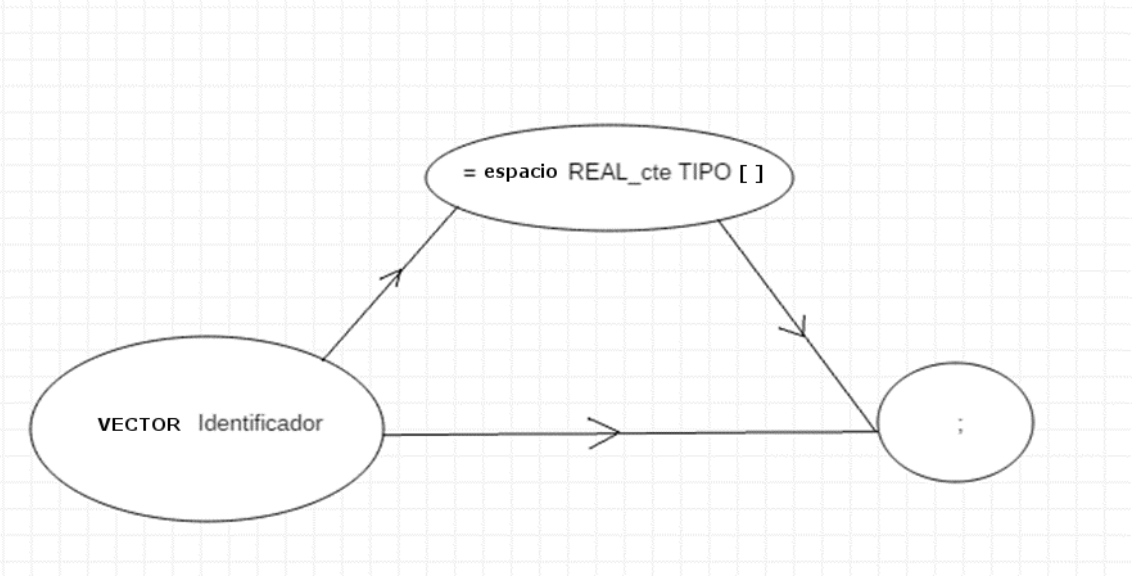

Ilustración 3: Declaración de una variable del tipo VECTOR.

DECLARACIÓN DE VECTORES Y MATRICES

En este lenguaje podremos declarar vector de reales y matrices de dos dimensiones. Todos los tipos de vectores nombrados los podremos declarar y posteriormente "rellenar"

con datos. También podremos declarar un vector y la vez indicar que conjunto de datos contiene ese vector. Ambas formas se realizan de la siguiente manera:

- Vector de reales:
  1. **VECTOR** numeros = **espacio** 5 ;
  2. **VECTOR** numeros = **[**1,2,3,4,5**]**;

Los vectores carecen de crecimiento dinámico, es decir, el espacio es reservado estáticamente en la declaración.

Para acceder al valor de una posición de un vector de "REAL" hay que tener en cuenta que va desde la posición 0 hasta el tamaño del vector -1, por tanto el primer valor estará almacenado en la posición 0 y así sucesivamente. Por ejemplo de la siguiente manera accederíamos a la posición dos del vector, por tanto, en la variable numero se almacenaría un 3.

```
VECTOR numeros = [1,2,3];

REAL numero = numeros [2]
```

OPERADORES ARITMÉTICOS

En este lenguaje dispondremos de un conjunto de operadores bastante grandes, lo que nos permitirá realizar prácticamente cualquier operación de manera directa. Dispondremos de operadores aritméticos, relacionales y lógicos.

Los operadores aritméticos los utilizaremos para realizar operaciones de tipo numérico por tanto serán utilizados por el tipo "REAL" o un "VECTOR con un conjunto de números.

Operador	Descripción	Ejemplo
+	Suma	5 + 2 = 7
-	Resta	5-3 = 2

*	Producto	3 * 5 = 15
/	División	5/2.5 = 2

Para cambios de signo de variables se recomienda el uso de una construcción del tipo ‘0-variable’.

OPERADORES RELACIONALES

También tendremos operadores relaciones los cuales utilizaremos para realizar una comparación numérico o de caracteres entre dos operandos. Exactamente los caracteres utilizarán únicamente los operadores de igual o distinto y solo podrán ser dos caracteres los que se comparen por tanto no se podrán comparar números y caracteres al mismo tiempo. Además no son iguales dos caracteres iguales pero uno en mayúscula y otro en minúscula. Tampoco se podrá utilizar el vector completo como operando pero si los elementos que componen dicho vector. El resultado de la evaluación de cualquier operador relacional será 1 o 0, 1 si se cumple esa comparación y 0 sino se cumple.

Operador	Descripción	Ejemplo/Resultado
==	Igual	5 == 5 /1
!=	Distinto	‘a’ != ‘b’ /0
>	Mayor	5 > 0 /1
<	Menor	5 < 4 /0
>=	Mayor o igual	5 >= 2 / 1
<=	Menor o igual	10 <= 5 / 0

OPERADORES LÓGICOS

Los operadores lógicos se aplican en operaciones relacionales con el objetivo de poder evaluar varias operaciones relacionales simultáneamente. En el caso de que se cumpla la condición el resultado será un 1, en caso contrario será un 0.

Operador	Descripción	Ejemplo/Resultado
&&	AND	(5 == 5 && 'A' == 'a') / 0
||	OR	(2 !=1 || 10.5 >= 2) / 1
!	NOT	!(2 == 1)/1

ESTRUCTURA DE LAS OPERACIONES

Las operaciones presentadas en los apartados anteriores solamente se pueden emplear usando el siguiente esquema:

<variable> = <termino> <operador> <termino>

OPERADORES Y OPERACIONES

El orden en el que se evaluarán los operadores aritméticos es el siguiente de izquierda a derecha (todos aquellos que están en la misma columna pertenecen al mismo nivel, si hay que evaluar varios de la misma columna se sigue el orden de aparición de izquierda a derecha). Por tanto la prioridad de los operadores aritméticos es primero los paréntesis y posteriormente los operadores que se encuentran en el primer nivel, segundo nivel y tercer nivel.

Primer Nivel	Segundo Nivel
/	+
*	-

El orden en el que se evaluarán los operadores relacionales es el siguiente (siguiendo las mismas reglas que en el caso anterior):

Primer nivel	Segundo nivel	Tercer nivel
>=	>	==
<=	<	!=

En el caso de los operadores lógicos se sigue su orden de aparición.

Ante el caso de que haya operadores de todos los tipos mencionados anteriormente se evaluarán los operadores aritméticos en primer lugar, seguidos por los relacionales y en último lugar los lógicos.

Todas las operaciones anteriores se pueden realizar con las siguientes combinaciones de tipos en el orden en el que aparecen a continuación:

- Escalar <operación> Escalar
- Escalar <operación> Vector
- Vector <operación> Vector

Para aquellas operaciones en las que interviene un vector y un escalar, la operación se realizará elemento a elemento del vector.

SENTENCIAS CONDICIONALES

En el lenguaje Ñ dispondremos de una sentencia condicional, la sentencia si para condiciones simples.

La sentencia “si” nos permite decidir entre dos posibles opciones excluyentes. La expresión que acompaña al “si”, la cual tiene que ser una expresión matemática y si el resultado de ésta es distinto de 0 ejecuta el conjunto de sentencias que está dentro del bloque.

```
si ( expresión){
    sentencias;
}
```

BUCLE MIENTRAS

Este tipo de bucle nos permite ejecutar un conjunto de sentencias repetitivamente mientras se cumpla una determinada condición. Una característica del bucle "mientras" es que se ejecuta 0 o N veces ya que si la condición del bucle no se cumple no se entra a ejecutar las sentencias.

Por ello su funcionamiento es muy sencillo, se evalúa la expresión que está en "mientras" y si como resultado se obtiene un número distinto de 0, se ejecutan el conjunto de sentencias que se encuentran en el interior del "mientras". Esto se repetirá hasta que la evaluación de la expresión de un 0, por tanto, se deje de cumplir la condición.

La sintaxis de "mientras" es la siguiente:

```
mientras (expresión) {
    sentencias;
}
```

Para salir del bucle se puede usar la cláusula "parar". Se recomienda que esta cláusula esté dentro de una estructura del tipo "si" para que realmente se pueda iterar en el bucle.

ESTRUCTURA DE UN PROGRAMA

Para elaborar un programa en Ñ, todo el código se debe definir en el mismo fichero de texto. Este fichero de texto podrá contener tantas funciones y variables globales como se desee.

Las variables globales se definen añadiendo la cláusula "global" delante de la declaración de la variable. Se deben definir fuera de una función y sólo se les puede dar valor dentro de las funciones. Para inicializarlo hay que escribir otra vez la definición de la variable, es decir <tipoDeVariable> <identificador>, donde el identificador es el mismo que tiene en la definición como variable global.(Véase el programa de conteo de coincidencias como ejemplo) No es necesario que todas las variables globales sean declaradas una detrás de otra, pueden declararse después del cierre de una función.

Existen tres tipos de funciones:

1. Función de inicio: no devuelve nada no acepta ningún parámetro y es la primera en ejecutarse. Su cabecera se compone de la siguiente forma "func inicio() {…}"
2. Funciones sin devolución: no devuelven nada y aceptan parámetros de entrada. Su cabecera es la siguiente "func nombreFuncion([parámetros]){…}"
3. Funciones con devolución: devuelven únicamente reales (para devolver vectores se recomienda que se usen las variables globales) y aceptan parámetros de entrada. Su cabecera es la siguiente "func REAL nombreFuncion([parámetros]){…}

Para realizar una llamada se pueden usar las dos siguientes estructuras:

1. Llama nombreDeLaFunción ([parámetros]);
2. REAL valorDeVuelto = llama nombreDeLaFunción ([parámetros]);

ENTRADA-SALIDA

En Ñ existen las palabras reservadas “lee” y “escribe” que corresponden a artilugios del lenguaje de programación que permiten leer de teclado y escribir en pantalla.

“lee” nos permite introducir por teclado el valor de una variable numérica.Puede ser un real o un vector.

“escribe” nos permite mostrar por pantalla un texto, el valor de una variable y valores numéricos, tras mostrarlo se produce un salto de líneas. Solamente admite un tipo de los tres a la vez.

En el caso de hacer uso de vectores para la lectura de un conjunto de valores se tendrá que declarar el vector como en los otros tipos e introducir los datos separados por espacios. Por ejemplo: 1 2 3 5.4 23.5

```
func inicio (VECTOR letras_de_comando){

        VECTOR v = espacio [4];

        matriz = lee ;

}
```

Cuando vayamos a mostrar por pantalla un vector se realizará de la misma manera que los otros tipos de datos, es decir llamando al identificador del vector. En este caso nos mostrará por pantalla todos los datos que contiene el vector ya pueden ser números o caracteres.

IDENTIFICADORES DE VARIABLES

Para determinar el identificador de una variable se puede utilizar cualquier cadena de caracteres que no incluya números al comienzo del mismo y que no coincida con ninguna de las palabras reservadas del lenguaje Ñ ni ningún símbolo especial.

PROGRAMAS DE PRUEBA

En este apartado expondremos una serie de programas de prueba con el objetivo de entender el lenguaje Ñ a la perfección. Además realizaremos una micro aplicación en lenguaje Java y la "traduciremos" al lenguaje Ñ.

PROGRAMA DE DECLARACIONES DE VARIABLES Y OPERADORES

En este programa se demostrará como realizar todos los tipos de declaraciones e inicializaciones de funciones junto con un subconjunto de operadores así como se utilizan los comentarios.

```
func inicio (){
	VECTOR a = [1,2,3];
	VECTOR z = espacio [3];
	a[2] = 4 ;
	REAL genious = 0;
	a = z;
	b =  4 + 5;
	c = b > 10; // daría 0
	d = 5 && a; // 1 puesto que ambos son distintos de 0
}
```

PROGRAMA PARA LEER DE TECLADO Y ESCRIBIR EN PANTALLA

En este programa se mostrará cómo se lee un vector de teclado y como se muestra por pantalla.

```
func inicio (){
        VECTOR teclado = espacio [5]
        teclado = lee;
        escribe teclado;
```

PROGRAMAS DE SENTENCIAS CONDICIONALES

En este primer programa de sentencias condicionales veremos cuando se ejecuta el bloque se sentencias dentro del si o las del sino en el caso de que hubiera. En este caso leemos por consola una edad e indicamos si con esa edad eres mayor de edad o no.

```
func inicio(){
   # Programa que indica si una persona es mayor de edad.
   REAL edad = 18;
   si ( edad >= 18){
       escribe “Eres mayor de edad” ;
    }
    si(edad < 18){
       escribe “No eres mayor de edad” ;
    }
}
```

PROGRAMA DE BUCLES

En este programa veremos cómo se puede hacer uso del bucles mientras que existeen el lenguaje Ñ. Para ello iniciaremos un vector de números con valores del 2 al 10 e incrementando de dos en dos.

```
func inicio(){

    VECTOR v = [16,17,28,9];

    REAL r = 0;

    REAL d;

     mientras (r != 10){

         escribe “Valor de v para esta posicion” ;

         d = v[r];

         escribe d;

        r = r + 1;
```

PROGRAMA DE LLAMADAS A FUNCIONES

A continuación ponemos como ejemplo de realización de llamadas a funciones, este caso en concreto no hace nada sino meterse en un bucle infinito:

```
global REAL primeraVarGlobal;

func inicio {
     REAL a = 3 ;
     REAL b = 3 ;
     si ( a > 2){
     mientras (b == 3){
            b = llama calculaTerror();
     }

     }
}

global VECTOR variableGlobalExtra;

func REAL calculaTerror(){
    REAL a = 3;
    devuelve a ;
}
```

PROGRAMA: CONTEO DE COINCIDENCIAS

Para mostrar la funcionalidad completa del lenguaje Ñ, hemos desarrollado una pequeña aplicación cuya función es la del conteo de coincidencias de un real en un vector.

```
global REAL coincidencias;

func inicio {
    VECTOR v = [16,35,66,16,5,9];
    REAL elementoBuscado = 16;
    REAL pos = 0;
    REAL coincidencias = 0;

    REAL d;

    mientras (pos < 6){
        d = v[pos];
        llama esCoincidencia(REAL d, REAL elementoBuscado);
        pos = pos + 1;
    }

}

func esCoincidencia(REAL e1, REAL e2){
    si (e1 == e2){
        coincidencias = coincidencias + 1;
    }

}
```

Este programa sirve de ejemplo para mostrar también cómo se inicializa una variable global desde dentro de una función.

ÁRBOL ABSTRÁCTO SINTÁCTICO

Para la realización de la práctica del analizador sintáctico se ha ampliado la práctica con la elaboración de un *árbol abstracto sintáctico* para obtener puntuación extra tal y como se especifica en el apartado del modle de la asignatura.

El árbol consta de nodos con distinto grado a lo largo de su estructura: los nodos hoja no tienen hijos, las expresiones aritméticas tienen dos hijos y los bloques de sentencias tienen tantos como sentencias contengan.

Esta estructura es facilitada por el uso de estructuras de datos accesibles desde el lenguaje de programación C++ como es la clase *std::vector<Node>*. Esta estructura además tiene la comodidad de contar con métodos que nos facilita la inserción por la parte final del vector, su limpieza y obtener el tamaño total.

Otra de las ventajas de usar C++ con Bison es que podemos hacer uso de las clases de objetos y de la herencia y el polimorfismo. Permitiendo así, crear una clase Nodo de la que heredan los demás tipos de nodos que contendrán diferentes tipos de datos según la finalidad con la que se haya concebido el nuevo tipo de nodo.

Todos los tipos de nodos que se han creado se listan a continuación:

- *Node:* clase base
- *Asignation:* clase base para las asignaciones
- *REAL_Asignation:* permite inicializar una variable de tipo REAL
- *VECTOR_Asignation:* permite inicializar una variable de tipo VECTOR
- *ELEM_VECTOR_Asignation:* permite modificar el valor de una posición de un vector.
- *VAR2VAR_Asignation:* permite asignar una variable a otra.
- *Expression:* clase base para las operaciones matemáticas
- *Math_Term<template>:* permite almacenar un real o un vector. Se usa como nodos hojas de las expresiones matemáticas.
- *Output_Expression:* permite imprimir por pantalla variables y valores.
- *BreakNode:* simboliza la ruptura del bucle.
- *Expression2Var:* asigna una expresión a una variable ya creada.
- *Declaration:* declaración de variables.
- *FunctionCall:* llamada una función con sus parámetros.
- *AsignationFunctionCall:* asigna a una variable el retorno de una llamada a función con sus parámetros.
- *FlowControl:* Almacena la expresión que permite ejecutar el conjunto de nodos que se le pasa por parámetro. Además tiene una variable booleana que le indica si es un bucle o si la ejecución se realiza una única vez.
- *NewBlock / ResumeBlock:* tipos especiales de nodos que se insertan para separar los nodos de un bloque del bloque anidado siguiente o volver al anterior bloque.
- *FunctionDefinition:* permite definir una función sus parámetros y los nodos que lo componen.
- *GlobalVar:* contiene la definición de la variable global en cuestión.

Se ha implementado dos recorridos del árbol: *roam*, que lo recorre mostrando mensajes de por donde va y alguna información adicional propia de cada tipo de nodo y *heightSearch(string id,int max)* que permite saber si un identificador está declarado en algún ámbito superior(de menor altura) al indicado por máx.

TABLA DE SÍMBOLOS

Para elaborar la estructura de la tabla de símbolos se opta por la construcción de una clase *SymbolTable* que contiene una estructura de datos tipo mapa en donde la clave es del tipo string y está asociado a una clase denominada *SymbolTableRecord* que almacena los siguientes tipos de datos:

- Una variable booleana para saber si es una variable del tipo global
- Un campo asociado a un tipo de datos enumerable *DataType* que puede obtener los siguientes valores: *real, DatatypeVector,initFunction,voidFunction,returnableFunction,flowControlWhile,flowControlIf.*
- Un entero que indica a que nivel del árbol AST se encuentra.
- Un vector con los nodos que cuelgan del nodo asociado a este registro.
- Un vector con la colección de nodos del tipo *Declaration* correspondientes a los parámetros de una función.

La tabla de símbolos se va construyendo a la par que el árbol sintáctico, es por ello que se aprovecha los dos últimos vectores de *SymbolTableRecord* para facilitar la construcción sin los errores que tenía en la primera versión de la construcción.

CHEQUEO DE ERRORES

Los chequeos que se realizan en esta versión son los siguientes:

- Comprobación de que las variables están declaradas antes de su utilización en expresiones matemáticas, salida por pantalla y en asignaciones.
- Tolerancia a la declaración interna de una variable global para ser utilizada dentro de una función.

GENERACIÓN DE CÓDIGO

La generación de código se realiza en el lenguaje Q que permite tener una experiencia de uso similar a la del lenguaje máquina.

Para generar la estructura de programa se recurre a un segundo recorrido del programa, haciendo uso del árbol abstracto sintáctico de las fases anteriores. En esta segunda fase se llamará a la función *generateCode* de cada nodo que devolverá una cadena con el fragmento de código que hay que añadir al fichero final. Dicha función aportará como parámetros de entrada los enteros correspondientes a la numeración de las etiquetas, de las zonas estáticas, la zona de código y la etiqueta de retorno la tabla de símbolos para que todas las llamadas posean la información necesaria para la ejecución.

Puesto que tenemos la información en la tabla de símbolos de qué entradas son globales, (bajo esta categoría se incluyen las variables globales y las declaraciones de funciones) podemos generar una cabecera dentro de las cláusulas de Q STAT(n) … CODE(n) que contenga la definición de todas las variables globales. En el mismo proceso se usa para darle una dirección de memoria en el campo correspondiente en la tabla de símbolos. Hay que destacar de esta sección que los vectores que son variables globales se les asigna 10 elementos en memoria, sin posibilidad de ampliar el espacio.

Tras la escritura de las variables globales se procede a escribir las funciones del usuario. Todas ellas comienzan por una etiqueta negativa. La primera de todas ellas está situada en -50 para evitar coincidencias con las etiquetas de la librería de Q. Las etiquetas generadas en el cuerpo de éstas son positivas.

Después de la escritura de dichas funciones se escribe la función de inicio, que comienza con la etiqueta 0 y finaliza con la llamada a GT(-2) para finalizar el programa.

Para el prototipo que se presenta junto con esta memoria del compilador se tienen programadas y probadas con confianza en su ejecución las siguientes operaciones:

- Definición de una variable del tipo REAL. Tanto como variable global como local.
- Definición de una variable del tipo VECTOR. Al igual que las variables del tipo REAL como global o como local.
- Asignaciones entre tipos de variables.
- Realización de operaciones aritméticas y relacionales entre cualquier combinación de REAL y VECTOR.
- Mostrar mensajes y valores de variables por pantalla.
- Control de flujo *si* y *mientras*.

Las variables para este prototipo se establecen también en la zona de las variables globales pero ante la escasez de tiempo no se ha podido corregir para insertar dichas variables en la pila, como sería deseable. Así pues la definición de cualquier variable sigue el esquema de:

- Calcular la próxima dirección de memoria, atendiendo al tamaño de los tipos de datos.
- Establecer en la tabla de símbolos dicha dirección.
- Guardar el valor en la zona estática de datos[mientras la versión sea un prototipo]

La generación del código asociado a las operaciones aritméticas es más complejo puesto que admitimos que se realicen operaciones real-real, vector-vector, real-vector y vector-real. Además a esto hay que sumarle que los términos matemáticos están asociados a nodos terminales mientras que la expresión en sí misma está en un nivel superior por lo que es necesario transmitir información de forma ascendente. Para resolver este problema se sigue la siguiente receta:

- Nodos terminales.
  - Abre hueco en la pila de su mismo tamaño.
  - Se guarda a sí mismo. (Si es una variable que hay que ir a buscar el valor a la zona de datos).
  - Escribe en la primera posición de la cadena a devolver el símbolo ‘v’ si es un vector y ‘n’ si se trata de un real (se omite el carácter r para no confundirlo con R de registro de la máquina Q).
- Nodos no terminales
  - Se lee los primeros caracteres de las dos cadenas que se obtienen.
  - Si ambos son reales se carga cada valor de la pila a un registro RR1 o RR2 para después realizar la operación y dejar el resultado en RR0.
  - Si ambos son vectores se realiza una comprobación de que ambos son del mismo tamaño. Si no lo son se lanza un mensaje y se aborta la ejecución. En caso contrario se realiza una operación elemento a elemento.
  - Si el real es el primero se carga desde la base de la pila (se conoce su posición por guardarse en R6 anteriormente) y el vector es todo lo demás.
  - Por último si el primero es el vector el real se carga de la cima de la pila.
  - Estos dos últimos puntos ejecutan la operación aplicando la misma operación con todos los elementos del vector manteniendo como segundo operando el real.

Los distintos tipos de asignaciones que están implementados son los siguientes:

- Asignación de un valor real a una variable, definida en ese punto.
- Asignación de un vector a una variable, definida en ese putno.
- Asignación de un valor real a una posición de un vector definido previamente.
- Asignación de una variable a otra.
- Asignación de un elemento de un vector a una variable.

- Asignación de una expresión a una variable.

Mostrar los reales por pantalla requiso de la elaboración de una función extra en Qlib.c que recogiese el valor real del registro para coma flotante RR0 y lo mostrase. Para mostrar un vector se itera llamando a esta función. También es posible mostrar cadenas de caracteres, pero este caso usa la función printf_ que ya traía Qlib.

Otra operación que se implementó en la librería Qlib.c es la función de casting o paso de real a entero. Su necesidad viene de la implementación del acceso a las posiciones de un vector cuando la posición a la que se accede viene determinada por el valor de una variable real.

El control de flujo tanto de la sentencia condicional *si* como del bucle de iteración *mientras* comparten un mismo tipo de nodo y además una receta común:

- Se genera el código asociado al nodo hijo que contiene la expresión matemática a evaluar. Este código dejará en el registro RR0 el resultado.
- Si el contenido de RR0 es 0 (en Q esto es equivalente *a IF(!RR0)GT(...)*) entonces se salta a la etiqueta calculada como fin del bloque. En caso contrario se ejecutarán las instrucciones que viene a continuación.
- En el caso de los bucles se añade un bloque de sentencias que devuelven el flujo de la ejecución otra vez al comienzo del bucle. Se recomienda al programador que codifique sentencias que varíen el valor de la variable que controla el bucle para evitar bucles infinitos.

APÉNDICE 1: MEZCLAR BISON CON C++

Para utilizar C++ con bison se puede optar por dos caminos:

1. Construir un Controlador personalizado que permita a Bison++ y a Flex++ ir intercalando sus funciones, es decir, según Flex++ lea un símbolo se pase a Bison++ para que compruebe las reglas de la gramática. Véase la Referencia 2.
2. A la hora de compilar usar el compilador g++ y añadir los archivos .cpp a la lista de ficheros a compilar. Tal y como ilustra la Referencia 1.

Para este trabajo de curso he optado por la segunda vía puesto que es la que requería menos tiempo y permitía llegar a la entrega. Los pasos de compilación están en el script adjunto a esta práctica.

BIBLIOGRAFÍA

Referencia 1 : Jumpstart Flex and Bison Bo Waggoner Universidad Hardvard - http://people.seas.harvard.edu/~bwaggoner/writeups/jumpstart/flexbison/jumpstart_flexbison.pdf

Referencia 2: Ejemplo GNU. - http://www.gnu.org/software/bison/manual/html_node/A-Complete-C_002b_002b-Example.html