Lenguaje de programación CAT*

Procesadores del lenguaje

María Goretti Suárez Rivero y Salvador Guedes Santana
Curso 2016/2017

CONTENIDO

1 INTRODUCCIÓN

En este informe se definirá la creación de un nuevo lenguaje muy especial llamado CAT*. Este lenguaje está pensado para los amantes de los felinos, y las diferentes palabras claves a lo largo de su definición harán recordar al usuario cuán maravilloso es el mundo gatuno. El lenguaje, de tamaño más bien mediano, podrá implementar funcionalidades como operaciones matemáticas y manejo de vectores y ristras de caracteres, abarcando un notable abanico de posibilidades. Podrá notarse la inspiración en lenguajes como Java y C++, pero las diferencias de notación son notables.

Se empezará definiendo el lenguaje al completo, tanto los tipos de variable como su declaración, pasando por los vectores y operadores, hasta llegar a las distintas estructuras de control que el lenguaje posee. Se concluirá con una batería de pruebas que el lenguaje debe superar, así como un pequeño programa de demostración de uso del lenguaje.

2 DESARROLLO

2.1 DEFINICIÓN DEL LENGUAJE

2.1.1 Tipos de datos

En esta sección hablaremos de los distintos tipos de datos que podemos encontrar en CAT*. Este lenguaje tendrá varios tipos de datos que se definen en la siguiente tabla:

Tipos	Definición	Ejemplos
cat	Se utilizará para declarar números enteros	5,0,-9
kitty	Se utilizará para declarar números reales de precisión simple con 32 bits	5.2,1.00047
kitten	Se utilizará para declarar números reales en coma flotante de 64 bit	0.333333333333333314829 61625624739099293947 21

La mayoría de operaciones que se realizarán con este lenguaje serán fundamentalmente numéricas, pero pueden crearse ristras de caracteres para poder mostrar mensajes por pantalla al usuario. Se podrán concatenar ristras explícitas a la hora de imprimir, con el operador ".".

2.1.2 Declaraciones de variables

La declaración de estas variables se hará siguiendo la siguiente estructura concreta:

<tipo> <nombre_de_variable> ~ <valor> ;

El operador "**~**" es el operador de asignación dentro del contexto de este lenguaje. Se usará siempre que se quiera adjudicar un valor a una variable. Una vez definido el tipo antes del nombre, solo se le podrán asignar valores de ese tipo a la variable. Si quisiéramos, por ejemplo, asignar el valor 5.2 a una variable cuyo tipo es "cat", se produciría un error.

Además, todas las declaraciones, así como todas las instrucciones simples dentro del lenguaje, acabarán con el símbolo ";".

Cada variable se declarará de manera independiente, en una línea aparte, terminando la declaración con ";". Algo como "cat a,b;" no estaría permitido, debería ser:

cat a;

cat b;

Otro punto a destacar es que al declarar las variables sin ningún tipo de asignación de valor a ella, no existe inicialización, es decir, no hay inicialización implícita. Al declarar una variable, se reservará espacio en memoria para ella, pero ese espacio estará vacío.

Por último, una vez una variable ha sido declarada con un nombre específico, no se podrá volver a crear otra variable con ese nombre, independientemente de que sea de un tipo de dato distinto.

2.1.2.1 Variables numéricas y de ristras

Si nos ceñimos a los tipos declarados en el apartado anterior, las declaraciones posibles serán las siguientes:

- **cat** nombre_variable_entera **~** 5
- **kitty** nombre_variable_32bits ~ 5.25
- **kitten** nombre_variable_64bits ~ 0.33333333333333331482961625624739

Los números decimales se tendrán que especificar siempre con ".", nunca con ",". Por ejemplo, es válido poner "5.3", y no es válido poner "5,3".

2.1.2.2 Vectores

En nuestro lenguaje no existirán vectores de ningún tipo, por lo que no se pueden manejar este tipo de estructuras.

2.1.3 Operadores

Los operados nos permitirán realizar diversas operaciones con nuestras variables. La estructura que seguirán las expresiones que utilizan operadores será la siguiente:

nombre_variable_1 <operador> nombre_variable_2 <operador> …

2.1.3.1 Operadores aritméticos

Los operadores aritméticos trabajan con variables únicamente numéricas y realizarán operaciones aritméticas, como su nombre indica. Los distintos operadores que existen están definidos en la siguiente tabla:

Operador	Descripción	Ejemplo
+	Suma dos variables	5+1, 5.2+4.1
-	Resta dos variables	9-1, 2.5-1.2
*	Multiplica dos variables	4*5, 4.1*0.5
/	Divide dos variables	4/2, 4.1/4.2

Los operandos a los lados de cada operador serán necesariamente del mismo tipo, y la operación que realice cada operando devolverá un resultado del mismo tipo que sus operandos. Para el caso de divisiones cuyo numerador sea menor que su denominador, el resultado seguirá siendo del mismo tipo que los operandos. Si quisiéramos dividir 1 entre 5, el resultado trabajando con números enteros sería 0. Si quisiéramos el resultado decimal, tendríamos que trabajar con 1.0 y 5.0, es decir, operandos decimales.

Si se quiere trabajar con números negativos, será necesario restarle a 0 el valor deseado. Si se quiere un -5, tendría que hacerse 0 – 5, y este valor puede almacenarse en una variable.

2.1.3.2 Operadores relacionales

Los operadores relacionales servirán para hacer distintas comparaciones entre las diferentes variables que creemos con nuestro lenguaje. Los dos primeros operandos que encontramos en la tabla inferior determinan la igualdad o desigualdad de dos variables. Dos variables serán iguales sin comparten el mismo tipo y el mismo valor, en cualquier otro caso serán distintas. En el caso concreto de los caracteres, se hará distinción entre mayúsculas y minúsculas.

Los siguientes operadores en la tabla determinarán si una variable tiene un valor mayor o menor que otra, y para ello necesitarán ser del mismo tipo.

El resultado de aplicar estos operadores será uno de los dos siguientes:

- 1 → Se cumple la comparación. La evaluación ha dado un valor positivo: es igual, es mayor, es menor …
- 0 → No se cumple la comparación. La evaluación da un valor negativo: no es mayor, no es menor…

Operador	Descripción	Ejemplo
=	Igualdad de dos variables	5 = 5, ‘a’ = ‘a’
!	Desigualdad de dos variables	5 ! 2, ‘a’ ! ‘b’, ‘a’ ! 5
>	Si una variable es mayor a otra	5 > 2
<	Si una variable es menor que otra	2 < 5

>=	Si una variable es mayor o igual a otra	2 >= 2, 5 >= 5
<=	Si una variable es menor o igual a otra	2 <= 5, 2<=2

En los operadores “>=” y “<=” importará el orden, debe especificarse de esa manera, no es admisible escribir “=>” ni “=<”.

2.1.3.3 Operadores lógicos

Los operadores lógicos se utilizan para poder usar varias expresiones relacionales en una misma expresión global, de manera que podamos hacer varias evaluaciones en la condición de un bucle, por ejemplo, o simplemente varias condiciones en una sola expresión. Los operadores lógicos que existen son los siguientes

Operador	Descripción	Ejemplo
&	AND lógico (Ambas expresiones se deben cumplir)	5=5 & a=b
|	OR lógico (Se cumple uno u otro)	5=5 | 5>n
¬	NOT (Negación)	¬a=5

Las expresiones en su conjunto no necesitarán paréntesis en sus extremos. El resultado de aplicar estos operadores será uno de los dos siguientes:

- 1 → La evaluación es positiva para todo el conjunto de expresiones. Se cumplen las restricciones.
- 0 → La evaluación es negativa para todo el conjunto de expresiones. No se cumplen las restricciones.

2.1.3.4 Operadores de entrada y salida

En CAT* existen unos operadores de entrada y salida de información para permitir al usuario introducir valores mediante la lectura del teclado y también para mostrárselos por la pantalla mediante la escritura en esta. Contamos con dos palabras reservadas:

1. **kin** → Con ella podemos leer del teclado, y que así el usuario pueda introducir el valor una variable de tipo numérica. Si se introduce algo que no sea numérico, se almacenará 0.

```
cat a;

kin a;
```

2. **kout** → Con ella podemos mostrar por pantalla tanto ristras como valores de variables numéricas para que el usuario pueda verlo, simplemente llamando a los identificadores de las variables.

```
cat a ~ 5;

kout a;
```

Cuando se quiera especificar una ristra explícita que se quiere imprimir, esta irá entrecomillada. Se podrán concatenar valores a la hora de imprimir por pantalla con el Kout utilizando el ".". Por ejemplo:

kout "El valor de la variable entero es: " . nombre_variable

kout "El valor de la variable float es: " . nombre_otra_variable

Para realizar un salto de línea tras cada impresión, bastará con añadir el elemento /CR a lo que se desea imprimir como una concatenación. Por ejemplo:

kout "El valor de la variable entero es: " . nombre_variable . /CR

A la hora de realizar el kin no realizamos comprobaciones de tipos puesto que no podemos comprobar que el usuario introduzca lo que queramos.

2.1.3.5 Orden de evaluación

Los operadores aritméticos se evaluarán de izquierda a derecha, y tendrán prioridad los paréntesis en caso de haberlos. Se seguirá el orden de prioridad entre operadores que existe actualmente en la matemática normal: las divisiones y multiplicaciones irán primero, y luego las sumas y las restas, y ante igualdad en nivel de prioridad, se evaluarán de izquierda a derecha.

- Primer Nivel : ()
- Segundo Nivel: *, /
- Tercer Nivel: +, -

Si nos ceñimos a los operadores lógicos, tienen todos el mismo nivel de prioridad, y se evalúan según su orden de aparición.

Si existiesen operadores de todos los tipos en una sola expresión, se evaluarían primero los paréntesis que existiesen, lo que hay dentro de ellos según los órdenes establecidos, luego se evaluarían los operadores relacionales en el orden especificado y, por último, se evaluarían los operadores lógicos.

2.1.4 Estructuras de control

En este apartado se definirán las distintas estructuras de control que posee el lenguaje, tanto sentencias condicionales como bucles para diferentes usos.

2.1.4.1 Estructuras condicionales

En este lenguaje se permite el uso de estructuras condicionales del tipo:

```
if (condición){
        acciones para el caso favorable
} else {
```

acciones para el caso no favorable

}

En nuestro lenguaje se realizará esta estructura con otras palabras reservadas:

claw (condición){

acciones para el caso favorable

} scratch {

acciones para el caso no favorable

}

Esta condición debe hacer uso de un operador relacional que compara dos expresiones. Si queremos hacer una comparación entre más de dos expresiones se deben hacer uso de los operadores lógicos. Además, en las expresiones se pueden hacer uso de operadores aritméticos. Ejemplos de condiciones:

Condición simple → <variables> <operador aritmético> <variable>

Condición compuesta → <condición simple> <operador lógico> <condición simple>

También se pueden realizar concatenaciones de condiciones compuestas por medio de otro operador lógico.

2.1.4.2 Estructuras de repetición

Con este tipo de estructuras podemos realizar repeticiones de una sección de código, para poder realizar esto podemos usar dos estructuras como pueden ser:

while (condición) {

acciones

}

for (variable; condición; avance){

acciones

}

En nuestro lenguaje y siguiendo los patrones anteriormente establecidos encontraremos estas estructuras de repetición.

meow (condición){

acciones

}

fishy (variable; condición; avance){

acciones

```
}
```

En esta estructura, ante la evaluación positiva de la condición, se ejecutará el contenido de la estructura. Cuando se ejecute, se volverá a evaluar la condición. En el caso del *meow (while)* la condición se corresponde a una condición simple o compuestas de las explicadas en el apartado anterior y no se debe olvidar realizar las modificaciones necesarias en las variables controladoras del bucle en el apartado de acciones para impedir una ejecución continua del bucle.

Por otro lado, en el uso del *fishy (for)* poseemos tres parámetros:

- Variable → declaramos e inicializamos la variable que controla el bucle
  - Restricción del lenguaje → solo se permite que la variable sea *cat*
- Condición → condición de parada del bucle del mismo estilo que el explicado en la sección anterior.
- Avance → realizamos la modificación que deseamos sobre la variable que controla el bucle:

<variable> = <variable> <operador aritmético> <número>

Se dispondrá de una instrucción de salida del bucle, llamada *purr*. Al utilizarla, se saldrá del bucle actual y se irá al anterior ambiente, es decir, a la función o al anterior bucle. En este ejemplo, se saldrá del segundo *meow* y se irá al primero:

```
meow (condición){
        acciones
        meow(condición) {
                acciones
                purr;
        }
}
```

2.1.4.3 Estructuras recursivas

En el lenguaje CAT* se permite el uso de llamadas recursivas, es decir, podemos realizar llamadas a nuestro propio método.

```
define nombre (tipo param1, tipo param2…) in (tipo variable_salida) {
        acciones
        nombre (param1, param2…) in variable_salida
}
```

Además, se podrá llevar a cabo recursividad indirecta:

Función A → Función B → Función C → Función A

Las estructuras recursivas deben poseer dentro de las acciones uno o más casos base que nos permitan finalizar la recursividad. Además, tenemos que tener en cuenta que el problema debe ir reduciéndose en cada llamada.

Los parámetros de salida de una función, en caso de realizar una llamada dentro de otra función, pueden usarse como parámetro de entrada para la llamada. Por ejemplo:

define <nombre función 1>(tipo <nombre parámetro entrada>) in (tipo <nombre parámetro salida>) {

cat a;

call <nombre función 2>(<nombre parámetro salida>) in a;

}

Cabe destacar que la función principal, llamada *main()* no admite recursividad, solo las funciones restantes en el fichero.

Si la función a la que se intenta acceder no se ha declarado en ninguna parte del fichero, se generará un error.

2.1.5 Estructura de un programa

Para realizar un programa en nuestro lenguaje definiremos en un fichero de texto con extensión ".ct" todo lo necesario para obtener la solución del mismo. En este se pueden tener el número de funciones que deseemos, pero siempre teniendo una de estas declarada como principal, usando la palabra clave *main* como nombre, y que debe estar declarada al inicio del fichero:

define main() → función principal

define nombre() → función auxiliar

Estas funciones irán delimitadas por llaves como se puede ver en el ejemplo de las estructuras recursivas. Al declarar una variable en una función determinada del fichero, ésta será considerada como variable local y solo podrá ser usada en la función que esté definida. Además, podrá servir de parámetro en una llamada a una función.

Este lenguaje no se ve obligado a ningún tipo de identación, aunque se recomienda el uso de una identación escalonada para un mayor entendimiento del programa.

En las funciones de nuestro programa se podrá devolver cualquier tipo de dato de las aceptadas por el lenguaje, así como no devolver nada.

A la hora de realizar las llamadas a las funciones podemos realizar:

call <nombre función>( );

ó

<tipo variable> <nombre_var_salida> ;

call <nombre función>(parámetros) in <nombre_var_salida>

El primer ejemplo representa a una función que no devuelve ningún resultado, mientras que la segunda si puede dar un resultado. Para que puedas coger un resultado, dentro de la función se tendrá que hacer uso de la variable de salida para que se almacene en ella lo que queremos. Por ejemplo:

```
define main () {
        cat param1;
        cat z;
        call <nombre función> (param1) in z;
}
```

```
define <nombre función>(tipo <nombre parámetro entrada>) in (tipo <nombre parámetro salida>) {
        cat a ~ 5 + 1;
        <nombre parámetro salida> ~ a;
}
```

Cuando se acabe de ejecutar la llamada, en el parámetro de salida estará el resultado, y cuando en la función principal usemos z después de la llamada, esta tendrá el valor dado en la llamada.

Siempre que se llame a una función que devuelve un valor, como en el segundo ejemplo, el usuario debe declarar la variable donde se va a almacenar el resultado de salida antes de hacer la llamada. Se recomienda el uso de comentarios antes de la definición de cada función expresando el tipo de valor que devuelve. El paso de parámetros es por copia, es decir, tu primero declararas los parámetros con un valor, y se manda una copia de este valor a los parámetros de la función llamada, y la función trabajará con una copia de los valores originales. No está permitido el casteo. Todas las variables usadas como parámetros en la llamada a una función deben estar previamente definidas.

Además, todas las instrucciones simples dentro del lenguaje, acabarán con el símbolo ";". y el símbolo "@" será empleado para los comentarios en este lenguaje. Este carácter se puede emplear en cualquier posición del fichero, pero esto ocasionará que lo posterior a él sea un comentario.

Existirá una palabra reservada llamada "halt" para abortar la ejecución del programa. Si se encuentra esa palabra en el código, se parará la ejecución.

2.2 PRUEBAS

2.2.1.1 Prueba 1: Declaraciones y Asignaciones. Impresión de variables.

```
@Funcion principal del programa. No retorna nada.
define main (){
        cat entero;
```

```
        kitty float;
        kitten double;

        entero ~ 1;
        float ~ 1.32;
        double ~ 0.3333333333;

        kout “El valor de la variable entero es: ” . entero . /CR;
        kout “El valor de la variable float es: ” . float . /CR;
        kout “El valor de la variable double es: ” . double . /CR;
}
```

2.2.1.2 Prueba 2: Entrada y Salida

```
define main(){
        cat dato;
        kout “Escriba un numero entero: ” . /CR;
        kin dato;
        kout dato;
}
```

2.2.1.3 Prueba 3: Operadores aritméticos

```
define main(){
        cat variable1 ~ 6;
        cat variable2 ~ 2;
        kitty solucion ~ variable1 + variable2;
        kout “La solucion de la suma de las variables es: ” . solucion . /CR;
        solucion ~ variable1 - variable2;
        kout “La solucion de la resta de las variables es: ” . solucion . /CR;
        solucion ~ variable1 * variable2;
        kout “La solucion de la multiplicacion de las variables es: ” . solucion . /CR;
        solucion ~ variable1 / variable2;
        kout “La solucion de la division de las variables es: ” . solucion . /CR;
```

```
        solucion ~ variable1 % variable2;
        kout “El resto de la division de las variables es: ” . solucion . /CR;
}
```

2.2.1.4 Prueba 4: Operadores relacionales y estructuras

```
define main(){
        cat dato1;
        cat dato2;
        kout “Escriba un numero entero: ” . /CR;
        kin dato1;
        kout “Escriba otro numero entero: ” . /CR;
        kin dato2;
        claw ( dato1 = dato2){
                kout “Las variables son iguales” . /CR;
        } scratch {
                claw (dato1 > dato2){
                        kout “La variable dato1 es mayor” . /CR;
                } scratch {
                        kout “La variable dato2 es mayor” . /CR;
                }
        }
}
```

2.3 PROGRAMA COMPLETO

2.3.1.1 Programa: Cálculo de diferentes operaciones matemáticas

```
define main (){
   cat opcion;
   kout "Selecciona una de las operaciones permitidas: " . /CR;
   kout " 1. Factorial" . /CR;
   kout " 2. Potencia " . /CR;
   kout " 3. Fibonacci " . /CR;
```

```
kin opcion;
claw(opcion = 1) {
   @Calculo del factorial

   cat dato;
   kout "Introduzca un número entero cuyo factorial desea calcular " . /CR;

   kin dato;
   cat resultado;
   call factorial(dato) in resultado;

   kout "El resultado del calculo del factorial es: " . resultado . /CR;

} scratch {
   claw(opcion = 2) {
        @Calculo de la potencia
        cat base;
        cat exponente;

        kout "Introduzca la base de la potencia " . /CR;
        kin base;

        kout "Introduzca el exponente de la potencia " . /CR;
        kin exponente;

        cat resultado;
        call potencia(base, exponente) in resultado;
        kout "El resultado del calculo de la potencia  es: " . resultado . /CR;

   } scratch {
```

```
        claw(opcion = 3) {
            @Calculo de Fibonacci
            cat dato;

            kout "Introduzca la posicion de la rista de fibonacci que desea: " . /CR;
            kin dato;

            cat resultado;
            call fibo(dato) in resultado;
            kout "El resultado para esta posicion es: " . resultado . /CR;

        } scratch {
            @Error
            call imprimirError();
        }
    }
  }
}

@EJERCICIO DE FACTORIAL
define factorial (cat num) in (cat resultFact) {
          resultFact ~ 1;
          meow(num > 1) {
                    resultFact ~ resultFact * num;
                   num ~ num - 1;
          }
}


@EJERCICIO DE FIBONACCI -- Devuelve el valor en la serie de fibonacci para esa posición
define fibo (cat N) in (cat resultFibo){
```

```
    claw (N = 1) {
        resultFibo ~ 0;

    } scratch {
        claw(N = 2) {
            resultFibo ~ 1;

        } scratch {
            cat r1;
            cat param1 ~ N - 1;
            call fibo(param1) in r1;

            cat r2;
            cat param2 ~ N - 2;
            call fibo(param2) in r2;

            resultFibo ~ r1 + r2;
        }
    }
}

@EJERCICIO CALCULAR POTENCIA
define potencia (cat base, cat exponente) in (kitten resultPot) {
            cat expo;
            kitten resultado ~ 1;

            claw (exponente < 0) {
            expo ~ 0 - exponente;
    } scratch {
            expo ~ exponente;
```

```
            claw (base = 0) {
                claw (exponente = 0) {
                    halt;
            } scratch {}

            resultPot ~ 0;
        } scratch {}
    }
    fishy (cat i ~ 0; i < expo; i ~ i + 1) {
            resultado ~ resultado * base;
            resultPot ~ resultado;
    }
    claw (exponente < 0) {
            resultado ~ 1 / resultado;
            resultPot ~ resultado;
    } scratch {}
}

define imprimirError() {
    kout "No ha dicho una opción válida" . /CR;
}
```

2.4 ANALIZADOR LÉXICO

Para realizar el analizador léxico del programa se utilizó Flex, desarrollando así un documento donde figurarían, en una primera instancia, todas las palabras clave junto con su codificación para identificarlas en el fichero. Seguidamente, se establecieron las expresiones regulares que representarían cada elemento, y que tras ser encontradas en el fichero de código, devolviesen el código pertinente. Por ejemplo, a la hora de encontrar un entero, utilizaríamos la siguiente expresión regular *{digito}+* donde digito es cualquier número entre 0 y 9. Al encontrarlo, devolveríamos el código que representa el entero.

Con el desarrollo de la práctica, los códigos fueron quitados de aquí, y ahora serían generados por el analizador sintáctico. Ahora, el analizador léxico sigue conteniendo las expresiones regulares, y

en algunos casos devuelve el valor encontrado haciendo uso de *yyval*, mientras que en otros simplemente devuelve su código identificativo. Por ejemplo:

```
{digito}+        {yylval.ristra = strdup(yytext); return CATVALUE;}
";"              {return PUNTOYCOMA;}
```

Se tiene en cuenta el fin de fichero, así como un contador de líneas que se usará para decir dónde están los errores léxicos en caso de haberlos.

2.5 ANALIZADOR SINTÁCTICO

Para la creación del analizador sintáctico se ha utilizado la tecnología Bison. Con él hemos podido declarar todos los terminales del lenguaje como *tokens* de Bison, y Bison se encarga de generar sus códigos identificativos, que antes especificábamos en Flex. Además, se ha podido especificar la prioridad de los operadores tal y como se comentaba en el apartado correspondiente de la definición del lenguaje. Se procedió, entonces, a la definición de la estructura gramatical del programa mediante la creación de no terminales en el lenguaje que serían equivalentes a otros conjuntos de terminales/no terminales. Así, se fue definiendo la estructura que seguían las instrucciones, asignaciones, expresiones, funciones, etc. Se especificó el tipo de terminales y no terminales en los casos que era necesario, pues el valor que retornasen iba a ser usado para ciertas tareas y debía de poseer un tipo.

Además, Bison pasó a contener la función principal, el *main*, que antes estaba en el Flex. El Bison se encargaría ahora de leer el fichero y utilizará la función *yyparse*(). Además, tiene una función para generar los errores. Existirán métodos declarados en el fichero Bison que serán usados durante la compilación para comprobar que los tipos de las variables sean correctos, así como que las condiciones de las estructuras de control sean lógicas.

Una vez definido el analizador sintáctico y tras resolver todos los conflictos que pudieran generarse, este era capaz de leer un fichero con código en lenguaje CAT* y de identificar si había errores léxicos y sintácticos. En caso de existir un error, se llamaría a la función *yyerror()* que se encargaría de notificárselo al usuario.

2.6 TABLA DE SÍMBOLOS

Para generar la tabla de símbolos que usará el compilador del lenguaje CAT* se ha creado una clase llamada *SymbolTable* en C++. Esta clase hace uso de la estructura *Symbol*, que representa un elemento de la tabla.

La estructura *Symbol* posee los siguientes elementos:

- *name*: nombre del símbolo. Si fuese una variable, por ejemplo, el nombre de la variable.
- *type*: tipo de dato que posee el símbolo.

- *scope*: ámbito dentro del programa en el que se encuentra ese símbolo.
- *isParameter*: identifica si el símbolo se considera o no parámetro de una función.
- *direction*: dirección relativa del símbolo.
- *etiquetaFuncion*: en caso de que el símbolo sea una función, que etiqueta tiene dentro del lenguaje Q, para poder saltar a ella.
- *funcName*: nombre de la función a la que pertenece el símbolo si es declarado como parámetro.
- *next*: símbolo siguiente al símbolo declarado.
- *prev*: símbolo previo al símbolo declarado.

La clase *SymbolTable* tiene:

- Atributos
  - *actualScope*: representa el ámbito donde se encuentra el programa en todo momento. Aumenta a medida que vamos internándonos en un nuevo ámbito (Del ámbito de una función paso al ámbito de una estructura dentro de una función…). Al salir de una estructura, disminuye.
  - *relDir*: dirección relativa al registro que represente la base de la pila que tendrá un nuevo elemento que se añada a la tabla.
  - *root*: símbolo que representa el primer elemento de la tabla
  - *last*: símbolo que representa el último elemento añadido a la tabla
- Funciones
  - *search_symbol(string varName)*: busca un símbolo dentro de la tabla con el nombre que se ha pasado por parámetro. Si lo encuentra, lo devuelve, sino devuelve NULL.
  - *addSymbol(string varName, int type*): añade un símbolo a la tabla en caso de que no exista.
  - *addParameter(string name, int type)*: añade un símbolo a la tabla especificando que es un parámetro de la última función declarada.
  - *addScope()*: aumenta el ámbito del programa, el *actualScope*.
  - *quitScope()*: disminuye el ámbito del programa, el *actualScope*. Hace uso del método recursivo *metodo(Symbol* previo, Symbol* item).*
  - *addEtiquetaToFuncion(string name, int value*): asigna la etiqueta que le corresponde dentro del código Q a la función con el nombre pasado por parámetro.
  - *imprimir()*: muestra la tabla de símbolos

Tras crear la tabla de símbolos e implementar todos sus métodos, se incluyó en el fichero de Bison para poder hacerle referencia, así como en la compilación final. Ahora, en el Bison, cada vez que se encontrase una asignación o una declaración en el código, ese elemento se añadiría en la tabla de símbolos. Al encontrarnos una función, también la añadiríamos junto con sus parámetros, y al encontrar las diversas estructuras, nos encargaríamos de controlar en que ámbito del programa nos encontramos.

2.7 GENERACIÓN DE CÓDIGO

En la última parte del desarrollo generamos el código correspondiente para que fuese interpretado por la máquina Q proporcionada por el profesor. Para ello, se añadió al Bison una macro de C++, llamada *generateCode*, que nos permite escribir código Q en un fichero de texto, que abriremos en la función *main* de Bison y permitiremos escribir en él. Se cerrará una vez se haya terminado el programa, y se usará luego para el intérprete de Q.

Al iniciar cualquier programa, antes de generar el código correspondiente a las funciones y su contenido, se generará una cabecera que contendrá los tipos de datos, necesarios para hacer en un futuro impresiones de datos por pantalla. Tras esto, se colocará la base de la pila, representada por el registro R6, después de la declaración de estos tipos, y a partir de ahí se generará el código correspondiente. La etiqueta L0 corresponderá a la primera asignación de la base de la pila, mientras que la etiqueta L1 siempre representará el comienzo de la función *main* del programa en lenguaje CAT*. Se tiene una variable global en el Bison que representa las etiquetas, su valor comenzará en el 2 y se usará cada vez que se desee establecer una etiqueta en el código Q. Cada vez que se use, obtendremos su valor, y luego lo aumentaremos, dejando preparada la variable para su siguiente uso.

Cada vez que se desee asignar un espacio en la pila (creación de una nueva variable, añadir un valor a utilizar, etc.), se desplazará el registro R7, que representa la cima de la pila. El registro R6 cambiará de posición siempre que cambiemos el ámbito o hagamos llamadas a otras funciones, siempre conservándose en la pila su estado anterior antes de cambiarlo. Este uso se combinará normalmente con el uso de la tabla de símbolos, ya que el desplazamiento con respecto a R6 que adquiera R7 se usará como el valor que debe tener el símbolo en la tabla en su atributo *direction*, y también irá actualizándose el *relDir* de la propia tabla. Para la creación de variables en la pila junto con su declaración en la tabla de símbolos se usarán *createVar(int type, string name)* y *varEnPilaSymbol(string name)*, métodos declarados en Bison. También se dispondrá de un método que simplemente reserve espacio en la pila, llamado *varEnPilaValue(int tipo).* Si se necesita la posición de memoria en todo el programa que posee un símbolo, se usará el método *posicionMemoria(Symbol * ítem)*, que calcula la posición del símbolo basándose en los saltos generados por el cambio de ámbito y en las posiciones relativas.

Normalmente, todas las variables, independientemente del tipo que se ha declarado en el programa al que pertenecen, se tratarán como de tipo *double* dentro del programa Q, asignándole a cada una 8 espacios de memoria.

Cabe destacar que la máquina Q ha sido modificada para implementar a nuestro modo las operaciones de impresión y lectura de datos, cambiando el método *putf_* que figura en *Qlib.c* y añadiendo un método llamado *scanf_*.

Para la implementación de la entrada y salida de datos se ha modificado la maquina Q suministrada, cambiando el putf_ y creando el scanf_.

En la modificación del putf_ se emplean los siguientes registros:

- R1 → Dirección de memoria donde está alojado el formato que deseamos imprimir.
- R2 → Dirección de memoria del dato que deseamos imprimir.
- R3 → Valor mayor o igual que 0 implica que imprimimos una String, -1 imprimimos un valor doublé, -2 imprimimos un entero, -3 imprimimos el salto de línea.
- R0 → Etiqueta de retorno

Para el uso del Scanf_ se emplean los siguientes registros:

- R1 → Dirección de memoria donde se guarda el dato suministrado por teclado.
- R0 → Etiqueta de retorno

NOTA: se emplea el registro RR1 para poder guardar el valor.

Durante esta parte se crearon, además, dos clases nuevas parecidas a *SymbolTable*: la clase *breaklist* y la clase FunctionsList. Ambas clases tienen estructuras parecidas, y usan una estructura Nodo adaptada a las necesidades de cada una.

En la clase *breaklist*, tendremos una estructura con los *purr* que se han establecido durante el programa, y tendremos los métodos necesarios para ir gestionándolos. De esta manera, podremos salir de los bucles siempre que se quiera.

La segunda clase, FuncitionList, tendremos todas las funciones que se han creado durante la revisión del código de CAT*. Con esta estructura sabremos que funciones se han llamado y cuales están declaradas en el programa, y nos ayudará a notificar errores en el caso de llamar a funciones que no se hayan definido.

Ambas clases serán incluidas en la compilación así como en los ficheros de Bison y Flex para poder ser utilizadas.