Practica Final

1. Frontend - Traductor de código en C a código en Lisp

En esta primera parte del documento pdf se va a hablar de las diferentes modificaciones que se le han realizado al código dado inicialmente ***'trad1.y'*** para que el código entregado cumpla con todas las especificaciones. Para ello vamos a ir enumerando cada especificación en el orden del pdf de la práctica indicando qué cambios se han realizado para el cumplimiento de la misma.

1.1 Especificación 1:

En esta especificación se nos pide eliminar la producción que deriva una **Sentencia → Expresion ;** para ello simplemente se ha ido al nodo no terminal de sentencia y se ha eliminado dicha producción, quedando así el nodo de sentencia de la siguiente manera:

```
sentencia:   IDENTIF '=' expresion     { sprintf (temp, "(setq %s %s)", $1.code, $3.code) ;
                                         $$.code = gen_code (temp) ; }
           | '@' expresion               { sprintf (temp, "(print %s)", $2.code) ;
                                           $$.code = gen_code (temp) ; }
       ;
```

1.2 Especificación 2:

Para esta especificación, se ha realizado un cambio en el nodo '***Sentencia'***, además, se ha creado un nuevo nodo denominado imprimir, con el cual de forma recursiva se pueden imprimir varias expresiones tal y como se pide en el enunciado. Ambos nodos no terminales se ven de la siguiente forma:

```
sentencia:   IDENTIF '=' expresion     { sprintf (temp, "(setq %s %s)", $1.code, $3.code) ;
                                           $$.code = gen_code (temp) ; }
           | '@' '(' imprimir  ')'       { $$.code = gen_code (temp) ; }
         ;
imprimir:    expresion          { sprintf (temp, "(print %s)", $1.code);
                                  $$.code = gen_code (temp) ;     }

           | expresion ',' imprimir   { sprintf (temp, "(print %s) ", $1.code);
                                        strcat (temp,$3.code);
                                        $$.code = gen_code (temp) ; }
           ;
```

1.3 Especificación 3:

1.3 Especificación 3 a:

Para esta primera especificación del apartado 3, se le ha añadido al nodo '***Sentencia'*** una nueva derivación: ' ***INTEGER IDENTIF '*** quedando dicha derivación de la siguiente forma:

```
| INTEGER IDENTIF        {sprintf (temp, "(setq %s 0)", $2.code) ;
                          $$.code = gen_code (temp) ;}
```

1.3 Especificación 3 b:

Al igual que en el caso anterior, se ha añadido una nueva derivación en el nodo de ***'Sentencia'*** la cual permite declarar una variable con un valor. Dicha derivación toma la siguiente forma:

```
| INTEGER IDENTIF '=' termino  {sprintf (temp, "(setq %s %s)", $2.code,  $4.code);
                                  $$.code = gen_code (temp) ;}
```

1.3 Especificación 3 c:

Al igual que se ha realizado en la especificación 2, se ha creado un nuevo nodo no terminal denominado **'setq'**, debido a la creación de de dicho nodo, se han tenido que realizar cambios en el nodo **'Sentencia'** de nuevo para que no hubiera fallos. De esta forma, ambos nodos juntos toman la siguiente forma:

```
sentencia:   IDENTIF '=' expresion     { sprintf (temp, "(setq %s %s)", $1.code, $3.code) ;
                                          $$.code = gen_code (temp) ; }
           | '@' '(' imprimir  ')'     {  $$.code = gen_code (temp) ; }
           | INTEGER setq              {  $$.code = gen_code (temp) ;}
         ;
setq:    IDENTIF                     {sprintf (temp, "(setq %s 0)", $1.code) ;
                                      $$.code = gen_code (temp) ;}
       |  IDENTIF '=' expresion      {sprintf (temp, "(setq %s %s)", $1.code,  $3.code) ;
                                      $$.code = gen_code (temp) ;}

       |   IDENTIF ',' setq          {sprintf (temp, "(setq %s 0)", $1.code) ;
                                      strcat (temp,$3.code);
                                      $$.code = gen_code (temp) ;}

       |  IDENTIF '=' expresion ',' setq   {sprintf (temp, "(setq %s %s)", $1.code,  $3.code) ;
                                            strcat (temp,$5.code);
                                            $$.code = gen_code (temp) ;}
     ;
```

1.4 Especificación 4:

Para este punto de la práctica se decidió borrar casi por completo la gramática que se tenía hasta este momento, de esta forma se pudo realizar una nueva gramática con el enfoque de traducir códigos en C a Lisp. Lo primero que se hizo fue crear un nuevo nodo no terminal ***'Axioma'***, el cual deriva en 'decl_var' y en ***'decl_fun'***, posteriormente en el nodo no terminal ***'decl_var'*** se implementaron todas las derivaciones creadas en la especificación anterior, pero añadiendo algún que otro no terminal que permitiese la declaración de múltiples variables globales en diferentes líneas de código. Para ***'decl_fun'*** actualmente solo funciona para main, pero esto cambiará en el futuro. Dicho esto, la gramática queda:

```
axioma:      decl_var decl_fun     { strcpy(temp,"");
                                     strcat(temp, $1.code) ;
                                     strcat(temp, $2.code) ;
                                     printf("%s\n",temp);}
        ;
decl_var:  /*lambda*/              { strcpy(temp,"");
                                     $$.code = gen_code (temp) ; }
          |  global ';' decl_var {  sprintf (temp, "%s%s", $1.code, $3.code) ;
                                    $$.code = gen_code (temp) ;}
        ;
global:    INTEGER setq_glob           {sprintf (temp, "%s", $2.code);
                                        strcat(temp,"\n");
                                        $$.code = gen_code (temp) ; }
       ;
setq_glob: IDENTIF              {  strcpy(temp,"");
                                   sprintf (temp, "(setq %s 0)", $1.code) ;
                                   $$.code = gen_code (temp) ;}
        | IDENTIF '=' termino    {sprintf (temp, "(setq %s %s)", $1.code,  $3.code) ;
                                  $$.code = gen_code (temp) ;}
        |  IDENTIF ',' setq_glob              {sprintf (temp, "(setq %s 0)", $1.code) ;
                                               strcat (temp,$3.code);
                                               $$.code = gen_code (temp) ;
        | IDENTIF '=' termino ',' setq_glob   { sprintf (temp, "(setq %s %s)", $1.code,  $3.code) ;
                                                strcat (temp,$5.code);
                                                $$.code = gen_code (temp) ;}
         ;
decl_fun :      MAIN                  { sprintf(name_func,"%s",$1.code); } ← Especificación 11
                    '(' argumentos_func')' '{' cuerpo '}'    {   strcpy(temp,"");
                                                                 sprintf(temp,"(defun main (%s)\n",$4.code);
                                                                 strcat(temp, $7.code);
                                                                 strcat(temp,"\n");    strcat(temp,")");
                                                                 $$.code = gen_code (temp) ;}
          ;
```

1.5 Especificación 5:

Como se ha visto en la gramática de la especificación anterior, se ha creado un nodo no terminal denominado cuerpo, en este nodo se tendrán derivaran todas las posibles acciones que pueden ocurrir dentro de una función en C. Debido a esto, en el nodo no terminal de **'cuerpo'** se ha añadido lo siguiente, ***'PUTS'*** y ***'STRING'*** son 2 palabras reservadas:

```
cuerpo:     PUTS '(' STRING ')' ';'        { strcpy(temp,"");
                                              sprintf(temp, "(print \"%s\")",$3.code);
                                              $$.code = gen_code (temp) ;}
          | PUTS '(' STRING ')' ';'  cuerpo      {  strcpy(temp,"");
                                                    sprintf(temp, "(print \"%s\")",$3.code);
                                                    strcat(temp,"\n");
                                                    strcat(temp, $6.code);
                                                    $$.code = gen_code (temp) ;}
          ;
```

De esta forma nos aseguramos de que se puedan añadir uno o varios **puts** dentro de una función.

1.6 Especificación 6:

Lo que se ha realizado para esta especificación es similar a lo mostrado anteriormente con el comando **puts**, pero en vez de usar la palabra reservada **PUTS**, se usa **PRINTF**, al mismo tiempo que lo que se encuentra entre paréntesis es el nuevo nodo no terminal denominado ***'imprimir'***,se ha asumido que al igual que puts, los string van con las " " que lo cierran ,vamos a mostrar como se ve esto en código:

```
cuerpo:    PRINTF '(' imprimir  ')'  ';'     {   strcpy(temp,"");
                                                 strcat(temp,$3.code);
                                                 $$.code = gen_code (temp) ; }
           | PRINTF '(' imprimir  ')'  ';'  cuerpo    {   strcpy(temp,"");
                                                          strcat(temp, $3.code);
                                                          strcat(temp,"\n");
                                                          strcat(temp, $6.code);
                                                         $$.code = gen_code (temp) ; }

imprimir:   STRING ',' imprimir2     {   sprintf (temp, "%s", $3.code);
                                         $$.code = gen_code (temp) ; }
         ;
imprimir2:    expresion          { sprintf (temp, "(prin1 %s)", $1.code);
                                   $$.code = gen_code (temp) ;     }
            | STRING             { sprintf (temp, "(prin1 \"%s\") ", $1.code);
                                   $$.code = gen_code (temp) ; }
```

```
  | expresion ',' imprimir2 {  sprintf (temp, "(prin1 %s) ", $1.code);
                               strcat (temp,$3.code);
                               $$.code = gen_code (temp) ; }
| STRING ',' imprimir2     {   sprintf (temp, "(prin1 \"%s\") ", $1.code);
                               strcat (temp,$3.code);
                               $$.code = gen_code (temp) ; }
;
```

1.7 Especificación 7:

Para este requisito, se ha tenido que modificar la precedencia de los operadores, además de añadir alguna palabra reservada para los operadores de más de un carácter. Una vez se han realizado esos cambios, se han añadido las operaciones con dichos nuevos operadores al nodo no terminal **'expresion'**. **DIFF** corresponde al operador **'!='**, **OR** es **'||'**, **AND '&&'**, **EQUAL '=='**, **SMALLER '<='** y finalmente **BIGGER '>='**.

```
%right '='
%left OR
%left AND
%left DIFF EQUAL
%left '<' SMALLER '>' BIGGER
%left '+' '-'
%left '*' '/' '%'
%left UNARY_SIGN '!'
#  Y las nuevas derivaciones de 'expresion' son:
        |   expresion OR expresion   { sprintf (temp, "(or %s %s)", $1.code, $3.code) ;
                                      $$.code = gen_code (temp) ; }
        |   expresion AND expresion   { sprintf (temp, "(and %s %s)", $1.code, $3.code) ;
                                       $$.code = gen_code (temp) ; }
        |   expresion DIFF expresion   { sprintf (temp, "(/= %s %s)", $1.code, $3.code) ;
                                       $$.code = gen_code (temp) ; }
        |   expresion EQUAL expresion   { sprintf (temp, "(= %s %s)", $1.code, $3.code) ;
                                          $$.code = gen_code (temp) ; }
        |   expresion '<' expresion   { sprintf (temp, "(< %s %s)", $1.code, $3.code) ;
                                      $$.code = gen_code (temp) ; }
        |   expresion '>' expresion   { sprintf (temp, "(> %s %s)", $1.code, $3.code) ;
                                      $$.code = gen_code (temp) ; }
        |   expresion SMALLER expresion   { sprintf (temp, "(<= %s %s)", $1.code, $3.code) ;
                                          $$.code = gen_code (temp) ; }
        |   expresion BIGGER expresion   { sprintf (temp, "(>= %s %s)", $1.code, $3.code) ;
                                          $$.code = gen_code (temp) ; }
        |   expresion '!' expresion   { sprintf (temp, "(not %s %s)", $1.code, $3.code) ;
                                      $$.code = gen_code (temp) ; }
        |   expresion '%' expresion   { sprintf (temp, "(mod %s %s)", $1.code, $3.code) ;
```

```
$$.code = gen_code (temp) ; }
```

1.8 Especificación 8:

Para esta especificación lo único que se ha tenido que realizar es añadir una nueva palabra reservada (**WHILE**) además de añadir dos nuevas derivaciones al nodo no terminal ***'cuerpo'***, quedando estas dos nuevas derivaciones de la siguiente manera:

```
| WHILE '(' expresion ')' '{' cuerpo '}'     {  strcpy(temp,"");
                                               sprintf (temp, "(loop while %s do %s)", $3.code, $6.code);
                                               $$.code = gen_code (temp) ;}

| WHILE '(' expresion ')' '{' cuerpo '}' cuerpo   { strcpy(temp,"");
                                                   sprintf (temp, "(loop while %s do %s)\n", $3.code, $6.code) ;
                                                        strcat(temp,$8.code);
                                                        $$.code = gen_code (temp) ;}
```

1.9 Especificación 9:

A diferencia de la especificación anterior, esta si ha tenido algo más de trabajo, puesto que se debe diferenciar entre un 'if' normal y un 'if-else'. Debido a esto, se han tenido que añadir en el nodo no terminal cuerpo cuatro nuevas derivaciones que lucen así, cabe destacar que al igual que ha ocurrido en especificaciones anteriores, se han tenido que añadir dos nuevas palabras reservadas, que son **IF** y **ELSE**:

```
# Los sprintf de los IF-ELSE van juntos en la misma línea, pero se han separado en 2 líneas para mayor claridad
| IF '(' expresion ')' '{' cuerpo '}'       { strcpy(temp,"");
                                              sprintf(temp,"(if %s\n(progn %s))",$3.code,$6.code);
                                              $$.code = gen_code (temp) ;}

| IF '(' expresion ')' '{' cuerpo '}' ELSE '{' cuerpo '}'    { strcpy(temp,"");
                                                              sprintf(temp,"(if %s\n(progn  %s)\n
                                                              (progn  %s))",$3.code,$6.code, $10.code);
                                                               $$.code = gen_code (temp) ;}

| IF '(' expresion ')' '{' cuerpo '}'  cuerpo    { strcpy(temp,"");
                                                   sprintf(temp,"(if %s\n(progn %s))",$3.code,$6.code);
                                                   strcat(temp,$8.code);
                                                   $$.code = gen_code (temp) ;}

| IF '(' expresion ')' '{' cuerpo '}' ELSE '{' cuerpo '}' cuerpo  { strcpy(temp,"");
                                                                    sprintf(temp,"(if %s\n(progn %s)\n
                                                                    (progn %s))",$3.code,$6.code, $10.code);
                                                                     strcat(temp,$12.code);
```

```
                                        $$.code = gen_code (temp) ;}
```

1.10 Especificación 10:

La adición del bucle for es muy parecida a la adición del bucle while, la única diferencia es la creación de dos nodos no terminales denominados ***'init'*** y ***'inc_dec'***. Además de la creación de la palabra reservada FOR. Dicho esto, las nuevas derivaciones de ***'expresion'*** quedan:

```
# Los sprintf de los bucles FOR se han dividido en 2 lineas para mayor claridad
| FOR '(' init ';' expresion ';' inc_dec ')' '{' cuerpo '}'  {  strcpy(temp,"");
                                                              sprintf (temp, "%s\n(loop while %s do %s \n%s)",
                                                              $3.code,$5.code, $10.code, $7.code);
                                                              $$.code = gen_code (temp) ;}

| FOR '(' init ';' expresion ';' inc_dec ')' '{' cuerpo '}' cuerpo  { strcpy(temp,"");
                                                                      sprintf (temp, "%s\n(loop while %s do %s
                                                                    \n%s)", $3.code,$5.code, $10.code, $7.code) ;
                                                                      strcat(temp,$12.code);
                                                                      $$.code = gen_code (temp) ;}

# Los nuevos no terminales lucen así

init:      IDENTIF '=' expresion          {  strcpy(temp,"");
                                             sprintf (temp, "(setf %s_%s %s)", name_func,$1.code,$3.code) ;
                                             $$.code = gen_code (temp) ;}

         | INTEGER IDENTIF '=' expresion     {  strcpy(temp,"");
                                              sprintf (temp, "(setq %s_%s %s)", name_func,$1.code,  $3.code);
                                                    $$.code = gen_code (temp) ;}
         ;


inc_dec:    IDENTIF '=' expresion          { strcpy(temp,"");
                                             sprintf (temp, "(setf %s_%s %s)", name_func,$1.code, $3.code);
                                            $$.code = gen_code (temp) ; }
         ;
```

Cabe destacar que el no terminal init permite la creación de una variable para dicho bucle asignando un valor.

1.11 Especificación 11:

Para la correcta realización de esta especificación se han tenido que añadir bastante código en comparación con otras especificaciones anteriores. Lo primero ha sido definir tresnuevas variables, **char name_func [64], char *array_local[50], int index_local = 0.** La variable **name_func** lo que hace es guardar en ese array el nombre de la función la cual se está traduciendo en ese instante, de esta forma se podrá concatenar más adelante, **array_local** es un array el cual guarda el nombre de las variables locales para que una vez ya se han declarado, si se utilizan dentro de la función en otra parte de la misma, también aparezcan concatenadas en ese punto, finalmente **index_local** es simplemente un index para **array_local**. Para saber si un valor está dentro de nuestro array se ha creado una función **int searchArray(char*[],char*)**, y para limpiar el array se ha creado otra **void cleanArray(char*[]).** Dicho esto se ha añadido derivaciones en expresión para la declaración de variables y su posterior asignación de valores, además de la creación de algún nodo no terminal.

```
# Nuevas derivaciones del nodo expresion:
| IDENTIF '=' expresion  ';'    { strcpy(temp,"");
                                  if (searchArray(array_local, $1.code) == 0){
                                        sprintf (temp, "(setf %s_%s %s)", name_func,$1.code,  $3.code) ;
                                  }else{
                                        sprintf (temp, "(setf %s %s)", $1.code, $3.code) ;
                                  } $$.code = gen_code (temp) ; }

| INTEGER setq ';'            { strcpy(temp,"");
                                 strcat(temp,$2.code);
                                 $$.code = gen_code (temp) ;}
# Estas 2 se vuelven a repetir en expresion añadiendo cuerpo después del ';' !!
setq:       IDENTIF          {  strcpy(temp,"");
                                 array_local[index_local] =  $1.code;
                                 index_local += 1;
                                 sprintf (temp, "(setq %s_%s 0)", name_func,$1.code) ;
                                 $$.code = gen_code (temp) ;}
        |  IDENTIF '=' termino    {sprintf (temp, "(setq %s_%s %s)", name_func,$1.code,  $3.code) ;
                                   array_local[index_local] =  $1.code;
                                   index_local += 1;
                                    $$.code = gen_code (temp) ;}

# Estas 2 se vuelven a repetir en setq añadiendo ',' setq  !! ejemplo :
          |   IDENTIF ',' setq         {sprintf (temp, "(setq %s_%s 0)", name_func,$1.code) ;
                                        array_local[index_local] =  $1.code;
                                        index_local += 1;
                                        strcat (temp,$3.code);
```

```
                                    $$.code = gen_code (temp) ;}
```

1.12 Especificación 12:

Para este punto se han realizado cambios al nodo de ***'decl_fun'*** además, se han creado nuevos nodos no terminales como ***'argumentos_func'***, ***'parametros'*** además de añadir la palabra reservada **RETURN**. Debido a estos cambios, se han tenido que añadir algunas derivaciones en expresión y en operando para permitir el uso de funciones como parámetro en las operaciones o poder asignar el valor de una función a una variable.

```
# Nuevas derivaciones del nodo expresion:
| IDENTIF '('parametros')' ';'        { strcpy(temp,"");
                                        sprintf(temp,"(%s %s)",$1.code,$3.code);
                                        $$.code = gen_code (temp) ;}
| RETURN expresion ';'                {strcpy(temp,"");
                                       sprintf(temp,"(return-from %s %s)",name_func,$2.code);
                                       $$.code = gen_code (temp) ;}
#Se repite la derivación de IDENTIF() añadiendo cuerpo despues del ';' !!!!!!!
# Nuevas derivación del nodo operando:
| IDENTIF '('parametros')'   { strcpy(temp,"");
                               sprintf(temp,"(%s %s)",$1.code,$3.code);
                              $$.code = gen_code (temp) ;}
# Nuevo decl_fun (main es un nodo no terminal que contiene el antiguo decl_fun):
decl_fun:  funciones  main            {  strcpy(temp,"");
                                         strcat(temp, $1.code);
                                         cleanArray(array_local);
                                          strcat(temp,$2.code);
                                        $$.code = gen_code (temp) ;}
        ;
funciones:  /*lambda*/                    {  strcpy(temp,"");
                                             $$.code = gen_code (temp) ;}

            | IDENTIF                 { strcpy(name_func,$1.code);}

                    '(' argumentos_func')' '{' cuerpo '}' funciones     {   strcpy(temp,"");
                                                                            sprintf(temp,"(defun %s
                                                                            (%s)\n",$1.code,$4.code);
                                                                            strcat(temp, $7.code);
                                                                            strcat(temp,"\n");
                                                                            strcat(temp,")\n");
                                                                            cleanArray(array_local);
                                                                            strcpy(name_func,"");
                                                                           strcat(temp,$9.code);
                        ;                                                  $$.code = gen_code (temp) ;}
```

Los nodos no terminales de **'parametros'** y **'argumentos func'** son prácticamente iguales, parametros puede recibir 0 o varias expresiones mientras que argumentos 0 o varios int identif.

1.13 Especificación 13:

Esta especificación es relativamente corta, lo único que se ha realizado es añadir la posibilidad de declarar vectores dentro de los nodos de ***'setq_glob'***, y en el nodo de ***'setq'***. Además se ha añadido al nodo ***'operando'*** una derivación para acceder a los valores de un vector y operar con ellos:

```
# Nuevas derivaciones del nodo setq_glob:
 |  IDENTIF '[' operando ']'  {   strcpy(temp,"");
                                  sprintf(temp, "(setq %s (make-array %s))",$1.code,$3.code);
                                  $$.code = gen_code (temp) ;}

|  IDENTIF '[' operando ']' ',' setq_glob    {    strcpy(temp,"");
                                                  sprintf(temp, "(setq %s (make-array %s))",$1.code,$3.code);
                                                  strcat (temp,$5.code);
                                                  $$.code = gen_code (temp) ;}

# Nuevas derivaciones del nodo setq:
|  IDENTIF '=' termino     {sprintf (temp, "(setq %s_%s %s)", name_func,$1.code,  $3.code) ;
                            array_local[index_local] =  $1.code;
                            index_local += 1;
                            $$.code = gen_code (temp) ;}

|  IDENTIF '[' expresion ']' ',' setq    {   strcpy(temp,"");
                                             sprintf(temp, "(setq %s (make-array %s))",$1.code,$3.code);
                                             array_local[index_local] =  $1.code;
                                             index_local += 1;
                                             strcat (temp,$5.code);
                                             $$.code = gen_code (temp) ;}

# Nuevas derivacion del nodo operando:
| IDENTIF '['expresion']'      {  if (searchArray(array_local, $1.code) == 0){
                                      sprintf (temp, "(aref %s_%s %s)",name_func,$1.code,$3.code) ;
                                  }
                                  else{
                                      sprintf (temp, "(aref %s %s)",$1.code,$3.code);
                                  }
                                  $$.code = gen_code (temp) ; }
```

2. Backend - Traductor de código en Lisp a código en Forth

Para esta parte, se ha descargado de nuevo el código **'trad1.y'** y se ha borrado toda la gramática que contenía. Ya que para la tarea a realizar se pensó que sería mejor empezar de esta forma. Cabe destacar que a pesar de que el backend tenga menos líneas de código que el frontend, este ha sido mucho más complicado de realizar, ya que al usar en Lisp los paréntesis para todo, se generaban muchos conflictos Shift/Reduce a nada que se modificara la gramática o se añadiera algo nuevo. Sin embargo se han conseguido realizar todas las especificaciones obligatorias incluyendo además alguna especificación opcional. Igual que con el Frontend, se va a realizar una pequeña explicación del trabajo realizado para cada especificación además de mostrar algo del código referente a dicha especificación.

2.1 Especificación 1:

Tal y como se ha mencionado anteriormente, se usó el fichero **'trad1.y'** como plantilla, y posteriormente se borró toda la gramática que contenía.

2.2 Especificación 2:

Sabiendo ya de antemano la tarea que se pretendía realizar con este backend, creamos ya un nodo **'axioma'** pensando ya en el futuro para evitar la realización de modificaciones. Debido a ello en axioma ya se usan dos nodos no terminales **'decl_var'** y **'decl_fun'**, para la realización de de esta especificación se ha modificado únicamente el nodo **'decl_var'**:

```
axioma:    '('decl_var decl_fun     { strcpy(temp,"");
                                      strcpy(temp,"\n");
                                      strcat(temp, $2.code) ;
                                      strcat(temp, $3.code) ;
                                      printf("%s",temp);
                                      strcpy(temp,"");}
                 '('MAIN')'         {printf("\n\nmain\nbye");}
        ;
decl_var:  /*lambda*/               { strcpy(temp,"");
                                      $$.code = gen_code (temp) ; }
         | SETQ IDENTIF expresion ')' '(' decl_var    { strcpy(temp,"");
                                                        if (strcmp($3.code,"0")==0){
                                                         /*Si solo se declara la variable, entramos aquí*/
                                                             sprintf(temp, "variable %s\n%s"
                                                            ,$2.code,$6.code);   }
                                                        else{
                                                             sprintf(temp,"variable %s\n%s %s !\n%s"
```

```
                                        ,$2.code,$3.code,$2.code,$6.code);  }
                                                        $$.code = gen_code (temp) ;}
        ;
```

2.3 Especificación 3:

Para esta especificación se define el nodo no terminal que hemos usado anteriormente en **'axioma'**, **'decl_fun'**

```
decl_fun:   DEFUN funciones          {strcpy(temp,"");
                                      strcat(temp,$2.code);
                                     $$.code = gen_code (temp) ;}
        ;

funciones:  /*lambda*/                          {strcpy(temp,"");
                                                 $$.code = gen_code (temp) ;}

        | IDENTIF '(' argumentos ')''('decl_var cuerpo ')' '(' DEFUN funciones   { strcpy(temp,"");
                                                                                   strcpy(param_name,"");
                                                                                array_funciones[index_local] =  $1.code;
                                                                                   index_local += 1;
                                                strcmp($3.code, "( -- )") == 0 ? strcpy(drop,"") : strcpy(drop,"DROP");
                                                                                sprintf(temp,"%s: %s %s\n%s%s\n;\n%s",
                                                                            $6.code,$1.code,$3.code,$7.code, drop,$11.code);
                                                                                $$.code = gen_code (temp) ;}

        | MAIN '(' argumentos ')''(' decl_var cuerpo ')'    {strcpy(temp,"");
                                                              sprintf(temp,"%s: main
                                                              %s\n%s;",$6.code,$3.code,$7.code);
                                                             $$.code = gen_code (temp) ;}
        ;
```

Como se puede ver, se ha utiliza un array de en el nodo **'funciones'** ese array guarda el nombre de la función que se está traduciendo en ese momento, de esa forma si posteriormente en otra función se llama a esa función, el programa lo reconocerá y no lo hará la traducción como si fuera una variable. También se puede ver es uso del comando de Forth **DROP**, el programa está diseñado para añadir un Drop al final de una función si esta tiene argumentos, actualmente solo funciona con funciones que tengan un único argumento y no hagan return en ningún momento.

2.4 Especificación 4:

Para esta especificación debemos enseñar el nodo no terminal **'cuerpo'** y el nodo no terminal **'cuerpo2'**, **'cuerpo2'** es el nodo que contiene todas las derivaciones de lo que se

puede hacer o no dentro de una función, mientras que **'cuerpo'** se encarga de iterar para poder traducir todas las operaciones.

```
cuerpo:      cuerpo2 ')'          {strcpy(temp,"");
                                   sprintf(temp,"%s",$1.code);
                                   $$.code = gen_code (temp) ;}

           | cuerpo2 ')' '(' cuerpo     {strcpy(temp,"");
                                         sprintf(temp,"%s%s",$1.code,$4.code);
                                         $$.code = gen_code (temp) ;}
            ;

cuerpo2:   PRINT STRING          {strcpy(temp,"");
                                  sprintf(temp,".\" %s\"CR\n",$2.code);
                                   $$.code = gen_code (temp) ;}

           | PRIN1 STRING        {strcpy(temp,"");
                                  sprintf(temp,".\" %s\"CR\n",$2.code);
                                  $$.code = gen_code (temp) ;}
```

De esta forma podemos traducir las impresiones de strings de manera correcta.Como podemos observar, se hace uso de dos nuevas palabras reservadas, **PRINT** y **PRIN1**

2.5 Especificación 5:

Para esta especificación se ha tenido que añadir una nueva derivación a nuestro nodo no terminal ***'cuerpo2'*** además de la modificación del nodo ***'expresión'*** ya que Lisp traduce las operaciones de manera prefija, mientras que C lo hace de manera infija, y ahora nosotros tenemos que poner ***'expresion'*** de manera prefija para poder traducirlo a postfijo.

```
# Nueva derivación de cuerpo2:
| PRIN1 expresion       {strcpy(temp,"");
                         sprintf(temp,"%s .\n",$2.code);
                          $$.code = gen_code (temp) ;}
```

```
#Ejemplos de modificaciones en expresion, solo se muestran algunas derivaciones:

    |  '+' expresion  expresion  { sprintf (temp, "%s %s +", $2.code, $3.code) ;
                                   $$.code = gen_code (temp) ; }
    |  '-' expresion  expresion  { sprintf (temp, "%s %s -", $2.code, $3.code) ;
                                   $$.code = gen_code (temp) ; }
    |  '*' expresion  expresion  { sprintf (temp, "%s %s *", $2.code, $3.code) ;
                                   $$.code = gen_code (temp) ; }
```

2.6 Especificación 6:

Esta especificación ya se ha podido ver anteriormente cuando se declaran las variables globales, ya que si tienen un valor diferente a 0 se usa este comando para asignarlas ese valor después de su declaración. Sin embargo, para poder realizar esto dentro de nuestras funciones, es preciso añadir nuevas derivaciones al nodo no terminal ***'cuerpo2'***:

```
| SETF IDENTIF expresion   {strcpy(temp,"");
                            sprintf(temp,"%s %s !\n",$3.code,$2.code);
                             $$.code = gen_code (temp) ;}
```

Como podemos observar, se ha tenido que crear una nueva palabra reservada **SETF** para reconocer la asignación de valor a una variable.

2.7 Especificación 7:

Al igual que en el caso anterior, lo único que se ha tenido que realizar es la creación de de nuevas palabras reservadas (**WHILE**, **LOOP**, **DO**) además de añadir una nueva derivación a ***'cuerpo2'***:

```
| LOOP WHILE '(' expresion ')' DO '(' cuerpo    {strcpy(temp,"");
                                                 sprintf(temp,"BEGIN %s WHILE\n%s\nREPEAT\n"
                                                 ,$4.code,$8.code);
                                                  $$.code = gen_code (temp) ;}
```

2.8 Especificación 8:

En este caso al igual que en el Frontend, se ha tenido en cuenta el caso en el que es un if únicamente y cuando es un if-else, debido a esto se han tenido que añadir 2 derivaciones a ***'cuerpo2'*** además de añadir nuevas palabras como **PROGN** o **IF**.

```
| IF '(' expresion ')' '(' PROGN '(' cuerpo ')'  {strcpy(temp,"");
                                                  sprintf(temp,"%s IF\n%s\nTHEN\n",$3.code,$8.code);
                                                  $$.code = gen_code (temp) ;}

| IF '(' expresion ')' '(' PROGN '(' cuerpo ')' '(' PROGN '(' cuerpo ')'  {strcpy(temp,"");
                                                                           sprintf(temp,"%s
                                                                           IF\n%s\nELSE\n%s\nTHEN\n"
                                                                          ,$3.code,$8.code,$13.code);
                                                                           $$.code = gen_code (temp) ;}
```

2.9 Especificación 9:

Para esta especificación se ha puesto la misma precedencia además de los mismos operadores que en el Frontend, la única diferencia ha sido que se ha tenido que cambiar

la traducción de algunos de estos operadores.

2.10 Especificación 10:

Esta especificación entra dentro de las especificaciones opcionales, pero se han realizado algunas de las peticiones que requería. Como se ha mencionado anteriormente, mi backend actualmente reconoce funciones con uno o varios argumentos, pero si se requiere la traducción a Forth de estos, solo será capaz de traducir y utilizar de manera correcta un único argumento. Al mismo tiempo se ha intentado la implementación del return-from, pero ha sido una tarea bastante complicada y al final a pesar de que lo reconoce y lo traduce de manera correcta, no he sido capaz de crear un flag que reconozca cuando ya se ha puesto un Drop en la función, por lo que se imprimen dos y da un error de underflow. También se ha creado la llamada a funciones dentro de otras funciones.
En cuerpo2:

```
| IDENTIF              {strcpy(temp,"");
                       sprintf(temp,"%s\n",$1.code);
                       $$.code = gen_code (temp) ;}

| IDENTIF param_entrada          {strcpy(temp,"");
                                 sprintf(temp,"%s %s\n",$2.code,$1.code);
                                 $$.code = gen_code (temp) ;}

| RETURN '-' FROM IDENTIF expresion        {strcpy(temp,"");
                                            if (strcmp(param_name,"")==0){
                                                  strcpy(drop,"");
                                             } else{
                                                   strcpy(drop,"DROP\n");
                                              }
                                             sprintf(temp,"%s\n%sEXIT\n",$5.code,drop);
                                              strcpy(drop,"");
                                              $$.code = gen_code (temp) ;}
          ;
param_entrada:   expresion                {strcpy(temp,"");
                                           sprintf(temp,"%s",$1.code);
                                           $$.code = gen_code (temp) ;}

                 | expresion   param_entrada2      {strcpy(temp,"");
                                                    sprintf(temp,"%s %s",$1.code,$2.code);
                                                    $$.code = gen_code (temp) ;}
          ;
```

param_entrada2 hace lo mismo que param_entrada, ahora enseñamos el nodo de argumentos:

```
argumentos:  /*lambda*/                          {strcpy(temp,"( -- )");
                                                  strcpy(param_name,"");
                                                  $$.code = gen_code (temp) ;}
        | IDENTIF                   {strcpy(temp,"");
                                     sprintf(temp,"( %s -- )",$1.code);
                                     strcpy(param_name,$1.code);
                                     $$.code = gen_code (temp) ;}
        | IDENTIF   argumentos2          {strcpy(temp,"");
                                          sprintf(temp,"( %s %s -- )",$1.code,$2.code);
                                          $$.code = gen_code (temp) ;}
        ;
argumentos2:   IDENTIF                  {strcpy(temp,"");
                                         sprintf(temp,"%s",$1.code);
                                         $$.code = gen_code (temp) ;}
              | IDENTIF   argumentos2        {strcpy(temp,"");
                                              sprintf(temp,"%s %s",$1.code,$2.code);
                                              $$.code = gen_code (temp) ;}
         ;
```

Como se puede ver, se usa una variable para guardar el nombre del argumento, para poder usar el comando **DUP** cuando este se usa, esto como se ha mencionado anteriormente solo se hace con un único argumento.

2.11 Especificación 11:

Para las variables locales se había hecho uso del comando **LOCALS** de Forth, pero al ver el problema que causaba a la hora de negar los valores, se ha optado por hacer que dichas variables locales se traduzcan como variables globales antes de utilizar una función, de esta forma podemos usarlas sin problema.

2.12 Especificación 12:

No se ha realizado.

3. Pruebas diseñadas:

Se han diseñado un total de 11 pruebas, algunas de ellas no pueden ser traducidas por nuestro backend debido a lo mencionado en las especificaciones.

- **print.c:** En esta prueba se quería demostrar el funcionamiento de la traducción del comando printf. Cabe destacar que al traducirlo a Forth, debido al salto de línea que genera el comando .” “ puede parecer que el 6 no está, pero si está.

- **puts_print.c:** Con este test se queria probar la diferencia en Lisp entre el comando prin1 y el comando print.

- **locales.c:** Con esta prueba se demuestra el funcionamiento de las variables locales y de las globales tanto en el frontend como en el backend.

- **while_loop.c:** Con este test se quiere ver la correcta traducción del bucle while tanto en el backend como en el frontend, para ello se crea una variable local la cual variará su valor dentro del bucle y se imprimirá su antes y su después.

- **for_loop.c:** El objetivo de esta prueba es ver si dada la traducción de nuestro frontend, nuestro backend es capaz de traducir y ejecutar de igual forma esta bucle.

- **if.c:** La idea de este test es comprobar 2 cosas al mismo tiempo, la primera cosa a comprobar es la precedencia de los operadores, para ver si se realizan de manera correcta, la segunda cosa a comprobar es la correcta realización de un if con 2 ramas sin else.

- **if_else.c:** Con este test se quería ver si se traduce correctamente el if-else.

- **funcion1.c:** Con este test se quería probar el funcionamiento de los traductores a la hora de enfrentarse a la llamada de una función sin parámetros.

A partir de aquí son pruebas que nuestro backend no puede ejecutar:

- **funcion2.c:** En este test se quiere probar el funcionamiento del frontend a la hora de enfrentarse a una función con un único parámetro y un return.

- **funcion3.c:** La idea del test es la misma que antes, pero esta vez la función tiene más argumentos, además de que tiene 2 return.

- **vector.c:** La idea de este test es declarar un vector global, llenarlo de valores en un bucle for y posteriormente imprimirlos uno a uno.