Universidad Carlos III

Procesadores del Lenguaje
Curso 2024-25

Práctica Final - Traductor C-Lisp-Forth

Grupo 208

07/04/2025

Salvador Ayala Iglesias 100495832@alumnos.uc3m.es

Inés Fuai Guillén Peña 100495752@alumnos.uc3m.es

Índice

Introducción

Esta práctica tiene como objetivo desarrollar un traductor de C a Lisp, al que nos referiremos como “frontend”. Posteriormente, se desarrollará un traductor que traducirá dicho código en Lisp a Forth, al que nos referiremos como “backend”.

Los traductores desarrollados cumplen con todos los requisitos especificados en el enunciado. Además, en el caso del backend, se incluye un fichero adicional (back-withFunctions.y) que implementa una gramática para reconocer y traducir parcialmente la definición de funciones (ver Backend con funciones).

En esta memoria, por tanto, se cubren las explicaciones pertinentes sobre el frontend y el backend. Además, se detallan las pruebas realizadas a ambos traductores, tanto las aportadas por el profesor como las desarrolladas por nosotros.

Todo el código ha sido desarrollado por los estudiantes que conforman este grupo. El trabajo realizado ha sido dividido de manera equitativa entre ambos. Se ha hecho uso de IA generativa para facilitar la comprensión de los dos lenguajes nuevos con los que hemos trabajado (Lisp y Forth), así como generar casos de ejemplo para estudiar su comportamiento de cara a desarrollar pruebas posteriormente.

Para evitar extendernos más de lo necesario, se omitirá las explicaciones de los tokens, ya que los nombres de estos son autodescriptivos. Todos los tokens reconocen la palabra que describen, la cual se ha añadido a la tabla *key_words* para poder procesarlos sin modificar el *yy_lex()*.

Frontend

El frontend tiene como objetivo tomar un fichero de código en C y traducirlo a Lisp, asegurando que su comportamiento no se vea modificado al cambiar de lenguaje. El formato de impresión difiere enormemente entre ambos lenguajes. Exceptuando esto, la salida debe ser idéntica.

Axioma

```
C/C++
axioma:       declVars                  { printf("%s", $1.code); }
              defFunc                   { ; }
              ;
```

El axioma, como se ha especificado, sólo puede tener la declaración de variables globales antes de las definiciones de las funciones. Las declaraciones de variables, en caso de haber alguna, se imprimen en el axioma, mientras que las definiciones de funciones se imprimen en su propia producción.

Variables globales y locales

```
C/C++
declVars:
              INTEGER IDENTIF asignacion cadenaDecl ';' declVars
{ if (strlen(current_func) > 0) { add_localVar($2.code); sprintf(temp,
"(setq %s_%s%s)%s\n%s",current_func, $2.code, $3.code, $4.code, $6.code);
} else { sprintf(temp, "(setq %s %s)%s\n%s", $2.code, $3.code, $4.code, $6.code); }
$$.code = gen_code(temp) ; }

              | INTEGER IDENTIF '[' operando ']' ';' declVars
{ if (strlen(current_func) > 0) { add_localVar($2.code); sprintf (temp, "(setq %s_%s
(make-array %s))\n%s", current_func, $2.code, $4.code, $7.code);} else {sprintf (temp,
"(setq %s (make-array %s))\n%s", $2.code, $4.code, $7.code); }
$$.code = gen_code(temp) ; }

              | /* lamda */  { sprintf (temp, ""); $$.code = gen_code (temp) ; }
              ;

cadenaDecl:
              ',' IDENTIF asignacion cadenaDecl     { if (strlen(current_func) > 0)
{add_localVar($2.code); sprintf(temp, "(setq %s_%s %s)%s", current_func, $2.code,
$3.code, $4.code); } else { sprintf(temp, "(setq %s %s)%s", $2.code, $3.code,
$4.code); }$$.code = gen_code (temp) ; }

              | /* lambda */ { sprintf (temp, ""); $$.code = gen_code (temp) ;}
              ;
```

```
asignacion: /* lamda */ { sprintf (temp, "0") ; $$.code = gen_code (temp) ; }
| '=' llamadaOExpresion { sprintf (temp, "%s", $2.code); $$.code = gen_code (temp) ; }
;
```

En este punto, trataremos conjuntamente las declaraciones de variables globales y locales, ya que la producción es la misma para ambas.

Como se puede observar en el código, se distingue entre variables simples y vectores, habiendo recursividad por la derecha para permitir N número de variables. También se permite la posibilidad de que no haya ninguna variable, incluyendo lambda en la producción.

En caso de asignarle un valor en la propia declaración, se le pondrá ese valor. En caso de no tratarse de ningún valor, se establecerá a cero por defecto.

Se permite la declaración de variables en una sola línea, separadas por comas. En cambio, para declaración de vectores, solo se permite la declaración de estos de uno en uno, ya que en el enunciado no se especifica que sea necesario, aunque sería trivial añadirlo a la producción. Por simplicidad, no hemos contemplado este caso.

La distinción entre variables locales y globales se hace en función de dos parámetros: la función actual y la tabla de variables:

```C/C++
typedef struct Node {
	char *var_name;
	struct Node *next;
} Node;
Node *localVars = NULL;

int search_localVar(char *var_name);
int add_localVar(char *var_name);
void free_localVars();

char current_func[64] = "";
```

La variable *current_func* empieza vacía, por lo que nos permite conocer si las declaraciones son globales (al incio) o dentro de una función. En caso de ser dentro de una función, la variable contendrá el nombre de esta. En este caso, se añade el nombre de la variable a la tabla para poder buscarla más tarde. En la tabla solo se almacena el nombre de la variable, independientemente de si es un vector o una variable simple.

Las variables locales tomarán el nombre de la función a la que pertenecen seguido de un guión bajo y el nombre de la variable, como se especifica en el enunciado. Los parámetros de la función también seguirán este formato, como se verá más adelante. La tabla se vacía al terminar la producción que define una función, como veremos a continuación.

Definición de funciones

```C/C++
defFunc:      main_func           { printCode($1.code) ; free_localVars(); }
              | name_func         { printCode($1.code) ; free_localVars(); }
              defFunc             { ; }
              | /* lambda */      { ; }
              ;
```

La definición de las funciones permite que no exista ninguna función, así como que existan N definiciones de funciones genéricas. Siempre habrá una única definición de la función main, en caso de haber alguna. Además, esta debe estar al final.

Las impresiones de estas definiciones se hacen al acabar la definición, haciendo uso de la función que hemos creado, llamada printCode. Esta se encarga de imprimir las tabulaciones necesarias en función del número de paréntesis abiertos (sentencias de lisp) delante de la traducción generada. Al acabar la impresión, se vacía la tabla de variables locales para permitir una nueva declaración de funciones con sus respectivas variables.

Definición del main

```C/C++
set_main: /* vacío */ { strcpy(current_func, "main"); }

main_func:  MAIN '(' ')' set_main '{' declVars sentencias '}'   { sprintf
(temp, "(defun main()\n%s%s)", $6.code, $7.code) ; $$.code = gen_code (temp)
; }
          ;
```

El main sigue la misma estructura que las definiciones de funciones genéricas, con la particularidad de que no recibe parámetros. La producción auxiliar set_main permite que se guarde el nombre de la función para usarla posteriormente en la impresión de variables locales.

Definición de funciones genéricas:

```C/C++
set_func: IDENTIF { strcpy(current_func, $1.code); $$ = $1; }

name_func: set_func '(' argumentos ')' '{' declVars sentencias '}'
{ sprintf(temp, "(defun %s (%s)\n%s%s)", $1.code, $3.code, $6.code,
$7.code); $$.code = gen_code(temp); } ;
```

```
argumentos:   /* lamda */ { sprintf (temp, ""); $$.code = gen_code (temp); }
            | INTEGER IDENTIF masArgs
{ if (strlen(current_func) > 0) {add_localVar($2.code); sprintf(temp,
"%s_%s%s", current_func, $2.code, $3.code); } else { sprintf(temp, "%s%s",
$2.code, $3.code); } $$.code = gen_code(temp); }
            ;

masArgs:     /* lamda */ { sprintf (temp, ""); $$.code = gen_code (temp) ; }
      | ',' INTEGER IDENTIF masArgs
{ if (strlen(current_func) > 0) { add_localVar($3.code); sprintf(temp, "
%s_%s%s", current_func, $3.code, $4.code); } else { sprintf(temp, " %s%s",
$3.code, $4.code); } $$.code = gen_code(temp); }
      ;
```

Para la definición de funciones genéricas, se sigue la misma estructura mostrada en el main, haciendo uso de una producción auxiliar para guardar el nombre de esta función. El uso de esta producción fue necesario para evitar errores que no supimos resolver, por lo que buscamos esta solución que, en esencia, es lo mismo que intercalar código diferido en la producción.

La función puede recibir parámetro o no, permitiendo definir funciones sin ningún parámetro. Los parámetros están separados por comas, por lo que se necesita dividir en dos producciones (*argumentos* y *masArgs*) para distinguir entre los casos de 0, 1 y más parámetros. En cualquier caso, dichos argumentos se añaden a la tabla de variables y se imprimen con el formato de una variable local.

En caso de definir variables locales al inicio del cuerpo de la función, se usa la misma producción que para las globales, ya la estructura es la misma.

Manejo de expresiones y llamadas

```
C/C++
llamadaOExpresion:  IDENTIF '(' argsLlamada ')'
{ sprintf(temp, "(%s %s)", $1.code, $3.code);$$.code = gen_code(temp); }
                  | expresion
{ sprintf(temp, "%s", $1.code); $$.code = gen_code(temp); }
                  ;

expresion:   termino            { $$ = $1 ; }
            | llamadaOExpresion '+' llamadaOExpresion
{ sprintf (temp, "(+ %s %s)", $1.code, $3.code); $$.code = gen_code(temp) ;}
            | llamadaOExpresion '-' llamadaOExpresion
{ sprintf (temp, "(- %s %s)", $1.code, $3.code); $$.code = gen_code(temp) ;}
            | ...
```

Por ahorrar en espacio, se omiten el resto de producciones, ya que el formato es el mismo, cambiando el operador por el correspondiente. Como se puede apreciar, se distingue entre una expresión aritmético lógica y una llamada a una función.

```
C/C++
termino:     operando                                  { $$ = $1 ; }
             |   '+' operando %prec UNARY_SIGN         { $$ = $1 ; }
             |   '-' operando %prec UNARY_SIGN         { sprintf (temp,
                                                       "(-%s)", $2.code) ;
                                                $$.code = gen_code(temp) ; }
             ;

operando:    IDENTIF      { if (strlen(current_func) > 0) {
                              if (search_localVar($1.code))
                                     sprintf(temp, "%s_%s", current_func,
                              $1.code);
                              else sprintf(temp, "%s", $1.code);
                            } else {
                              sprintf(temp, "%s", $1.code); }
                            $$.code = gen_code (temp) ; }
             |   NUMBER                 { sprintf (temp, "%d", $1.value) ;
                                          $$.code = gen_code (temp) ; }
             |   '(' expresion ')'      { $$ = $2 ; }
             ;
```

Como se puede ver, el código es muy sencillo. Lo único a destacar es la distinción entre variable local y global.

Sentencias

```
C/C++
sentencias:   /* lambda */                    { sprintf (temp, "");
                                               $$.code = gen_code (temp) ; }
              | sentencia sentencias           { sprintf (temp, "%s\n%s",
                                               $1.code, $2.code);
                                               $$.code = gen_code (temp) ; }
              ;

sentencia: (...)
```

Una vez se pasa de las declaraciones de variables locales, la función puede tener N sentencias, incluso ninguna. Por tanto, la recursividad por la derecha nos permite procesar las sentencias una por una.

Asignaciones de variables

```
C/C++
IDENTIF llamadaOAsignacion ';'
       { if ($2.function == '0'){
              if (strlen(current_func) > 0) {
                  if (search_localVar($1.code))
                     sprintf(temp, "(setf %s_%s %s)", current_func,
                     $1.code, $2.code);
                  else sprintf(temp, "(setf %s %s)", $1.code, $2.code);
              } else {
                  sprintf(temp, "(setf %s %s)", $1.code, $2.code);
              }
       } else {
              sprintf(temp, "(%s%s)", $1.code, $2.code); }
              $$.code = gen_code (temp) ; }
```

A la hora de encontrarnos un identificador, hay que distinguir entre si se trata de una asignación de un valor a una variable o una llamada a una función sin asignación a nada. Por ello, se factorizan estas posibilidades en la producción *llamadaOAsignacion*, la cual devolverá un 0 en .función en caso de tratarse de una variable, mientras que devolverá un 1 en caso de ser una llamada a una función.

En caso de ser una variable, se imprime con el formato de asignación de Lisp. Se busca el nombre de la variable en la tabla de variables locales. En caso de encontrarse, se trata de una variable local, mientras que en caso de no encontrarla, se trata de una variable global.

```
C/C++
llamadaOAsignacion: '(' argsLlamada ')'  { sprintf(temp, "%s", $2.code);
                                           $$.code = gen_code(temp);
                                           $$.function = '1'; }
             | '=' llamadaOExpresion  { sprintf(temp, "%s", $2.code);
                                           $$.code = gen_code(temp);
                                           $$.function = '0'; }
             ;
```

En caso de ser una asignación, lo que encontraremos será un igual, seguido del valor a asignar. Dicho valor puede ser una expresión aritmética/lógica o tratarse de una llamada a una función, como hemos visto anteriormente.

En caso de ser una llamada, lo que encontraremos será un paréntesis, por lo que procesaremos los argumentos de la llamada, en caso de haber alguno. Se procesan en dos reglas separadas, para poder distinguir en el caso de las comas (varios argumentos):

```
C/C++
argsLlamada:        /* lamda */                { sprintf(temp, "");
                                                $$.code = gen_code(temp); }
             | expresion otroArgLlamada  { sprintf(temp, " %s%s", $1.code,
                                  $2.code); $$.code = gen_code(temp); }
             ;

otroArgLlamada: /* lamda */ { sprintf(temp, ""); $$.code = gen_code(temp);}
             | ',' expresion otroArgLlamada  { sprintf(temp, " %s%s",
                           $2.code, $3.code); $$.code = gen_code(temp); }
             ;
```

Asignación de vectores

```
C/C++
| IDENTIF '[' llamadaOExpresion ']' '=' llamadaOExpresion ';'
      { if (strlen(current_func) > 0) {
            if (search_localVar($1.code))
                sprintf(temp, "(setf (aref %s_%s %s) %s)", current_func,
                $1.code, $3.code, $6.code);
            else
                sprintf(temp, "(setf (aref %s %s) %s)", $1.code, $3.code,
                $6.code);
      } else {
            sprintf(temp, "(setf (aref %s %s) %s)", $1.code, $3.code,
            $6.code);
      }
      $$.code = gen_code(temp); }
```

Para asignar vectores, se sigue la misma lógica que para asignar variables normales, con la diferencia de que el formato cambia ligeramente para encajar con el manejo de vectores de Lisp.

Impresión con puts y printf

```
C/C++
| PUTS '(' STRING ')' ';'{ sprintf (temp, "(print \"%s\")", $3.code) ;
                           $$.code = gen_code (temp) ; }
| PRINTF '(' STRING printArgs ')' ';'    { sprintf(temp, "%s", $4.code);
                                           $$.code = gen_code(temp); }
```

Puts solo puede procesar strings, mientras que *printf* procesará el string de formato seguido de los argumentos oportunos. No se ha considerado el caso de un printf sin el string inicial.

```
C/C++
printArgs:    /*lamda*/                  { sprintf(temp, "");
                                         $$.code = gen_code(temp); }
              | ',' otroPrint printArgs   { sprintf(temp, "(princ %s) %s",
                                                    $2.code, $3.code);
                                          $$.code = gen_code(temp); }
              ;

otroPrint:    llamadaOExpresion    { sprintf(temp, "%s", $1.code);
                                   $$.code = gen_code(temp); }
              | STRING             { sprintf(temp, "\"%s\"", $1.code);
                                   $$.code = gen_code(temp); }
              ;
```

Los argumentos se procesan y se formatean usando *princ*, mientras que puts usaba print.

Condicionales

```
C/C++
| IF '(' expresion ')' '{' sentencias '}' posibleElse
       { sprintf(temp, "(if %s\n(progn %s\n)%s)", $3.code, $6.code, $8.code);
       $$.code = gen_code(temp) ; }
```

Se procesan todos los condicionales haciendo uso de progn, independientemente del número de líneas. Se podría distinguir entre ambas, pero por simplicidad se ha mantenido siempre la sentencia progn para distinguir entre ambas ramas.

```
C/C++
posibleElse:  /* lambda */                    { sprintf(temp, "");
                                               $$.code = gen_code(temp); }
              | ELSE '{' sentencias '}'        { sprintf(temp, "\n(progn
                                               %s\n)", $3.code);
                                               $$.code = gen_code(temp); }
              ;
```

En caso de tener un else, se procesa la segunda rama del condicional, también con progn.

Bucle while

```
C/C++
| WHILE '(' expresion ')' '{' sentencias '}'
       { sprintf (temp, "(loop while %s do\n%s\n)", $3.code, $6.code) ;
       $$.code = gen_code (temp) ; }
```

El bucle while, se procesa y se traduce a la estructura de lisp, procesando la expresión que determina la condición de salida del bucle y las sentencias que pueda contener este.

Bucle for

```
C/C++
| FOR '(' IDENTIF asignacion ';' expresion ';' IDENTIF asignacion ')' '{'
sentencias '}'
{ if (strlen(current_func) > 0) {
       if (search_localVar($3.code)) {
              if (search_localVar($8.code)) { sprintf (temp, "(setf %s_%s
                     %s)\n(loop while %s do\n%s\n(setf %s_%s %s)\n)",
                     current_func, $3.code, $4.code, $6.code, $12.code,
                     current_func, $8.code, $9.code) ;
              } else { sprintf (temp, "(setf %s_%s %s)\n(loop while %s
                     do\n%s\n(setf %s %s)\n)", current_func, $3.code,
                     $4.code, $6.code, $12.code, $8.code, $9.code) ;
              }
       } else {
              if (search_localVar($8.code)) { sprintf (temp, "(setf %s
                     %s)\n(loop while %s do\n%s\n(setf %s_%s %s)\n)",
                     $3.code, $4.code, $6.code, $12.code, current_func,
                     $8.code, $9.code) ;
              } else { sprintf (temp, "(setf %s %s)\n(loop while %s
                     do\n%s\n(setf %s %s)\n)", current_func, $3.code,
                     $4.code, $6.code, $12.code, $8.code, $9.code) ;
              }
       }
} else {
       sprintf (temp, "(setf %s %s)\n(loop while %s do\n%s\n(setf %s
       %s)\n)", $3.code, $4.code, $6.code, $12.code, $8.code, $9.code) ;
}
$$.code = gen_code (temp) ; }
```

El código del bucle for es tan extenso debido a las dos asignaciones de inicio y fin, que contienen una variable (que puede ser local o global). Dado que no existe el bucle for en Lisp, este se ve transformado a un bucle while que cumple con las condiciones especificadas en la declaración del bucle original.

Retorno

```
C/C++
| retorno     { sprintf(temp, "%s", $1.code); $$.code = gen_code(temp); }
;

retorno: RETURN llamadaOExpresion ';'    { sprintf(temp, "(return-from %s
             %s)", current_func, $2.code); $$.code = gen_code(temp); }
;
```

La última sentencia posible que hemos contemplado es el retorno, que puede estar en cualquier parte de la función. Por tanto, para simplificar, en caso de encontrar un retorno se traduce directamente como “return-from <funcion>”, independientemente de dónde esté localizado.

Comentarios relevantes

Terminadas todas posibles sentencias, no quedan más fragmentos a traducir. Todo lo que no está contemplado en el frontend dará como resultado un error de sintaxis, especificando la línea que ha dado error.

Sobre las funciones que manejan la tabla de variables locales y la función de impresión con formato tabulado, se omiten en esta memoria por simplicidad. Se pueden encontrar en el archivo listadoF, junto con el resto del código.

Todas las pruebasproporcionadas por el profesorado, así como las que hemos desarrollado nosotros, han sido ejecutadas y se han pasado con éxito.

Backend

El backend se encargará de tomar el código generado por el frontend y traducirlo a gforth. Debido a las especificaciones limitadas para evitar una complejidad elevada, el backend solo traducirá la función main, ignorando por completo las funciones genéricas.

Axioma

```
C/C++
axioma:      '(' primitivas ')'         { ; }
             r_axioma                   { ; }
             ;

r_axioma:   /* lamda */                 { ; }
             |   axioma                 { ; }
             ;
```

El axioma es simple, permite procesar sentencias “primitivas”, tantas como se quiera.

Primitivas

```
C/C++
primitivas:  SETQ IDENTIF NUMBER { printf("variable %s\n", $2.code);
                                   printf("%d %s !\n", $3.value, $2.code); }
             |   DEFUN MAIN '(' ')' sentencias
                                   {printf(": main\n%s;\n", $5.code); }
             |   MAIN              {printf("main\n"); }
             ;
```

Dado que los archivos de prueba son las salidas del frontend, estos seguirán la misma estructura que el archivo original. Por lo tanto, solo puede haber tres tipos de sentencias: la declaración de variables globales, declaración del main y la llamada a este. En caso de tratarse de una definición de una variable, se le asigna el valor numérico con el operador ! de Forth.

No se ha controlado que se redefina el main o que se llame varias veces, así como no se ha controlado el orden de estas primitivas. Consideramos que no es necesario, dado que siempre operará sobre el frontend, el cual ya asegura el orden correcto.

Manejo de expresiones

Como no tenemos funciones, no es necesario tratar las llamadas a funciones. Sin embargo, conviene comentar las producciones relativas a las expresiones aritmético lógicas.

Una vez más, omitiremos la mayoría de reglas, ya que siguen la misma estructura.

```
C/C++
expresion:    termino                                   { $$ = $1 ; }
              | '(' '+' expresion expresion ')'
{ sprintf(temp, "%s %s +", $3.code, $4.code); $$.code = gen_code(temp); }
              | '(' '-' expresion expresion ')'
{ sprintf(temp, "%s %s -", $3.code, $4.code); $$.code = gen_code(temp); }
              | (...)

termino:      operando                                  { $$ = $1 ; }
              |   '(' '-' operando ')' %prec UNARY_SIGN
{ sprintf (temp, "%s negate", $3.code) ; $$.code = gen_code (temp) ; }
              ;

operando:     IDENTIF                   { sprintf (temp, "%s @", $1.code) ;
                                          $$.code = gen_code (temp) ; }
              |   NUMBER                  { sprintf (temp, "%d", $1.value) ;
                                          $$.code = gen_code (temp) ; }
              ;
```

En este caso, las producciones del término se simplifican, ya que el operando solo puede tener un signo unario en caso de ser negativo. Dicho caso se recoge en la producción, dejando el resto de operandos como variables o números. En caso de ser una variable, se accede a su valor usando el operador @ de Forth.

Sentencias

```
C/C++
sentencias:   '(' sentencia ')' sentencias      { sprintf (temp, "%s\n%s",
                                                $2.code, $4.code);
                                                $$.code = gen_code (temp) ; }
              |   /* lamda */                   { $$.code = gen_code(""); }
              ;

sentencia: (...)
```

Como en el caso del frontend, las sentencias se procesan individualmente, permitiendo un número arbitrario de ellas gracias a la recursividad por la derecha y a la producción con lambda. Las sentencias, como pasaba con las primitivas, siempre están rodeadas por paréntesis.

Asignaciones y declaraciones

```
C/C++
SETQ IDENTIF NUMBER        { printf("variable %s\n", $2.code);
                           sprintf(temp, "%d %s !", $3.value, $2.code);
                           $$.code = gen_code(temp); }
|   SETF IDENTIF expresion { sprintf(temp, "%s %s !", $3.code, $2.code);
                           $$.code = gen_code(temp); }
```

En caso de encontrarnos una declaración de una nueva variable (identificada con SETQ) se deberá imprimir directamente la declaración de la variable en FORTH para asegurar que se imprima antes que la definición de la función. De esta manera se asegura que la variable exista la hora de asignarle un valor.

En el caso de SETF, simplemente se le asigna el valor, ya que la variable ya existirá, como sabemos al haber traducido con el frontend.

Impresiones con princ y print

```
C/C++
|   PRINC impresion { sprintf(temp, "%s", $2.code);
                    $$.code = gen_code(temp); }
|   PRINT STRING    { sprintf(temp, ".\" %s\"", $2.code);
                    $$.code = gen_code(temp); }
```

Como sabemos que los strings con formato irán precedidos de PRINC, podemos dividirlo en dos casos. Usaremos los operadores .” <string> “ para imprimir los strings simples (precedidos de PRINT).

```
C/C++
impresion:   STRING             { sprintf(temp, ".\" %s\"", $1.code);
                                $$.code = gen_code(temp); }
             |   expresion      { sprintf(temp, "%s .", $1.code);
                                $$.code = gen_code(temp); }
             ;
```

Las impresiones de expresiones vendrán divididas en varios PRINC, cada uno conteniendo una única expresión o string. Una vez más, usaremos los operadores .” <string> “ para imprimir los strings, mientras que para las expresiones usaremos el operador . de Forth.

Loop while

Como ya reducimos los tipos de bucles a solo el loop while, no hay que considerar más que un tipo de bucle.

```
C/C++
|   LOOP WHILE expresion DO sentencias  { sprintf(temp, "BEGIN\n%s
                                        WHILE\n%sREPEAT", $3.code, $5.code);
                                        $$.code = gen_code(temp); }
```

Se transforma el código al formato de Forth, usando la estructura “BEGIN WHILE REPEAT”.

Condicionales

```
C/C++
|   IF expresion '(' PROGN sentencias ')' posibleProgn
      { if (strlen($7.code) > 0)
            sprintf(temp, "%s IF\n%sELSE\n%sTHEN",
            $2.code, $5.code, $7.code);
      else
            sprintf(temp, "%s IF\n%s\nTHEN", $2.code, $5.code);
       $$.code = gen_code(temp); }
;

posibleProgn: '(' PROGN sentencias ')'  { sprintf(temp, "%s", $3.code);
                                        $$.code = gen_code(temp); }
            |   /* lamda */             { $$.code = gen_code(""); }
            ;
```

El último tipo de sentencia son los condicionales, que siguen la estructura de Lisp. Como debemos discernir entre si tiene una sentencia de tipo ELSE o no, buscamos un token PROGN al final. En caso de encontrarlo, seguiremos la estructura de IF-ELSE-THEN.

En caso de no encontrarlo, será simplemente IF-THEN.

Comentarios relevantes

Terminadas todas posibles sentencias, no quedan más fragmentos a traducir, según las especificaciones del enunciado de la práctica.

Las pruebas ejecutadas han sido todas las del conjunto 00, todas ellas han sido pasadas con éxito.

Pruebas desarrolladas

Se han desarrollado varias pruebas para comprobar el correcto funcionamiento. Todas ellas se han pasado con éxito.

declar_global.c

```C/C++
int x = 5;

main() {
	int a = 3, b = 4;
	a = a + b;
	puts("Hola mundo");
}
```

funcion_completa.c

```C/C++
funcion(int a) {
	int x = 5, y = 2, z;
	int v[10];
	v[0] = x + y;
	z = v[0] * 2;
	puts("Valores iniciales:");
	printf("%d", x);
	printf("%d", y);
	printf("%d", z);
	if (z > 10) {
		puts("Z es mayor que 10");
	} else {
		puts("Z es menor o igual a 10");
	}
	for (x = 0; x < 5; x = x + 1) {
		v[x] = x * 2;
		printf("%d", v[x]);
	}

	while(y < 10){
		z = z + 1;
	}
	return z;
}
```

main_completo.c

```C/C++
main() {
	int x = 5, y = 2, z;
	int v[10];
	v[0] = x + y;
	z = v[0] * 2;

	puts("Valores iniciales:");
	printf("%d", x);
	printf("%d", y);
	printf("%d", z);
	if (z > 10) {
		puts("Z es mayor que 10");
		z = z / 2;
	} else {
		puts("Z es menor o igual a 10");
		z = z * 3;
	}
	for (x = 0; x < 5; x = x + 1) {
		v[x] = x * 2;
		printf("%d", v[x]);
	}
	while(y < 10){
		z = z + 1;
	}
	return z;
}
```

main_for.c

```C/C++
main() {
	int i;
	for (i = 0; i < 3; i = i + 1) {
		puts("Hola");
	}
}
```

main_if.c

```C/C++
main() {
	int x = 5;
	if (x > 0) {
		if (x < 10) {
			return x;
		}}
}
```

main_llamfunc.c

```C/C++
suma (int a, int b) {
	return a + b; }
main() {
	int r = suma(3, 4);}
```

main_vectores.c

```C/C++
main() {
	int v[5];
	int x;
	v[0] = 10;
	x = v[0];
	return v[0] + v[1] + v[2];
}
```

main_while.c

```C/C++
main() {
	int i = 0;
	while (i < 10) {
		i = i + 1;
	}
}
```

Backend con funciones

Por curiosidad académica, se ha tratado de implementar el backend con funciones. Se adjunta el fichero *back-withFunctions.y*. No se ha llegado a implementar soluciones a las llamadas de funciones recursivas, pero las traducciones de funciones simples parecían funcionar a simple vista. En cualquier caso, la gramática se ideó para poder reconocer todas las posibles expresiones incluyendo funciones.

A pesar de no estar incluido en los contenidos de la práctica, hemos querido investigarlo. Por falta de tiempo no nos ha sido posible llegar a implementarlo, pero hemos ideado varias funciones que simularían una pila en la que almacenar parámetros y variables locales para preservarlas entre llamadas de funciones. El paso de parámetros entre funciones se haría a través de la pila de Forth, mientras que usaríamos nuestra propia pila como almacenamiento auxiliar para variables, simulando la pila de la que hacen uso los programas reales. Asimismo, se podría devolver el valor de la última sentencia, siguiendo el comportamiento de Lisp, dejándola en la pila de Forth y leyéndola al resolver la llamada a esta función.

Consideramos que esta práctica cubre aspectos muy interesantes y que, con el suficiente tiempo, nos proporcionaría un conocimiento amplio de estos lenguajes a los que estamos muy poco acostumbrados.

Conclusión

Esta práctica ha sido muy compleja y ha requerido un gran esfuerzo por nuestra parte. Sin embargo, hemos aprendido y disfrutado las sesiones de prácticas, avanzando gradualmente y resolviendo los problemas que íbamos encontrando por el camino. A pesar de esto, hemos tenido que trabajar bastante por nuestra cuenta y resolver otros tantos errores que no encontramos en las propias sesiones.

Creemos que la planificación de la asignatura está maravillosamente pensada y que se avanza a un ritmo idóneo para darnos tiempo a interiorizar los conocimientos que se van impartiendo en las clases.

Por otro lado, el dividir la traducción en dos fases ha simplificado enormemente las complejidades que podríamos haber encontrado al traducir a Forth. El dividir los tipos de asignaciones entre declaraciones y asignaciones, el poder distinguir entre tipos de impresión y el simplificar los bucles en una sola estructura ha sido crucial para poder entender la traducción a Forth. Dicha simplicidad se ve reflejada en la longitud del código del back en comparación a la del front, siendo esta última muchas veces mayor.

Consideramos que esta práctica nos ha proporcionado un mejor entendimiento de los compiladores, además de permitirnos pensar de maneras que no habíamos considerado anteriormente.