Universidad Carlos III
Grado en ingeniería informática
Curso Procesadores del Lenguaje 2024-25
Práctica Final - Entrega Final
Curso 2024-25

GRUPO: **403**

Alumnos: **Mario Ramos Salsón (100495849)**

Miguel Yubero Espinosa (100495984)

ÍNDICE CONTENIDOS

DECLARACIÓN USO DE IA GENERATIVA

En la realización de esta práctica se ha hecho uso de la IA Generativa de forma limitada únicamente en casos muy excepcionales para preguntar formas de solucionar algún error muy puntual, o ejemplos de programas en C para pruebas. Todo el diseño, desarrollo y estructura del programa han sido realizados íntegramente por los integrantes del grupo, sin depender de la inteligencia artificial. El uso de la IA no ha influido en las decisiones clave ni en la implementación general del proyecto.

La práctica ha sido desarrollada de forma equitativa por los integrantes del grupo.

DESARROLLO DE LA GRAMÁTICA DEL FRONTEND / TRAD.Y (C → Lisp)

Primero de todo, vamos a comenzar explicando el código y desarrollo de la gramática del trad.y para traducir una entrada en C a lenguaje Lisp.

- **Axioma**

Es la producción principal, y establece que la traducción ha finalizado cuando se han impreso las variables globales, las funciones y el código main.

```
C/C++
axioma: printGlobales printFunciones main { ; }
```

- **Main**

Esta producción define la estructura de la función principal del programa. Para el funcionamiento correcto de esta función, hemos tenido que guardar en una variable auxiliar “*current_function”* el nombre *“main”* ya que, toda variable que se declare dentro de esta función tiene que ir precedida por *“main_*” para indicar que es local. Más adelante veremos exactamente el uso de “*current_function”*.

```C/C++
main: MAIN '(' ')' { strcpy(current_function, "main"); }
    '{' cuerpo '}' {
        limpiar_tabla_local();
        sprintf(temp, "(defun main ()\n\n%s\n)\n\n", $6.code);
        $$.code = gen_code(temp);
        strcpy(current_function, "");  // Borramos funcion actual
        printf("%s", $$.code);};
```

- **Variables globales**

A continuación, explicamos la producción completa de las variables globales. Estas pueden estar o no, de ahí que tengamos una producción para dar Lambda y otra para imprimir las variables. Cabe destacar que se imprimen en este NT, ya que su impresión en *“main”* es una mala práctica. Por ello mismo nos hemos visto obligados a crear la regla “*printGlobales”* una regla únicamente para imprimirlas.

```C/C++
printGlobales: { $$.code = NULL; }
    | globales {
        printf("%s\n\n", $$.code);};
```

- **globales**

Esta producción, nos permite manejar la recursividad de la creación de variables globales, ya que puede haber una detrás de otra.

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

- **creacionVarGlobales**

Esta regla nos permite generar cualquier tipo de creación de variables globales que se haga en una misma línea, es decir cualquier línea de código que

empiece por *“int creación1, creación2, etc… ;”* Para ello permitimos dos maneras de comenzar, la primera de ellas con un *“int variable”* y la segunda con un “*int array[N]”*.

```
C/C++
creacionVarGlobales: INTEGER IDENTIF asignacionVar concatenacionVarGloables ';' {
if ($3.code == NULL && $4.code == NULL) {
            sprintf(temp, "(setq %s 0)", $2.code);  // Valor por defecto 0
        } else if ($4.code == NULL) {
            sprintf(temp, "(setq %s %s)", $2.code, $3.code);
        } else if ($3.code == NULL) {
            sprintf(temp, "(setq %s 0) %s", $2.code, $4.code);
        } else {
            sprintf(temp, "(setq %s %s) %s", $2.code, $3.code, $4.code);}
        $$.code = gen_code(temp);}
    | INTEGER IDENTIF '[' NUMBER ']' concatenacionVarGloables ';' {
        if ($6.code == NULL) {
            sprintf(temp, "(setq %s (make-array %d))", $2.code, $4.value);
        } else {
                    sprintf(temp, "(setq %s (make-array %d)) %s", $2.code, $4.value,
$6.code);}
        $$.code = gen_code(temp);};
```

- **asignacionVar**

Esta regla nos permite manejar el caso en el que se cree una variable de la siguiente manera, *“int a = expresion”*. Queremos destacar que debido a esta regla, nuestro código podría aceptar algo como “int a = c + d”, que estaría mal, pero si queremos restringir estos casos tendríamos que hacer una gramática mucho más amplia y consideramos que no merece la pena.

```
C/C++
asignacionVar: { $$.code = NULL; }  // Lambda
    | '=' expresion { sprintf(temp, "%s", $2.code);
                $$.code = gen_code(temp);};
```

- **concatenacionVarGloables**

Nos permite gestionar cualquier tipo de concatenación que esté permitida en la creación de una variable global. Estos casos son por ejemplo cuando cuando se concatena con un *“, variable”* o un *“, array[N]”* o con *“, variable = expresion”*.

```
C/C++
concatenacionVarGlobales: { $$.code = NULL; }  // Lambda
    | ',' IDENTIF concatenacionVarGlobales {
        if ($3.code == NULL) {
            sprintf(temp, "(setq %s 0)", $2.code);
        } else {
            sprintf(temp, "(setq %s 0) %s", $2.code, $3.code);
        }
        $$.code = gen_code(temp);
    }

    | ',' IDENTIF '[' NUMBER ']' concatenacionVarGlobales {
        if ($6.code == NULL) {
            sprintf(temp, "(setq %s (make-array %d))", $2.code, $4.value);
        } else {
            sprintf(temp, "(setq %s (make-array %d)) %s", $2.code, $4.value, $6.code);
        }
        $$.code = gen_code(temp);
    }

    | ',' IDENTIF '=' expresion concatenacionVarGlobales {
        if ($5.code == NULL) {
            sprintf(temp, "(setq %s %s)", $2.code, $4.code);
        } else {
            sprintf(temp, "(setq %s %s) %s", $2.code, $4.code, $5.code);
        }
        $$.code = gen_code(temp);};
```

- **Funciones**

La forma en la que gestionamos la impresión de las funciones es igual a la de las variables globales, una producción *“printFunciones”* que baraja la posibilidad de que no haya funciones, o en caso contrario, las imprime.

```
C/C++
printFunciones: { $$.code = NULL; }
    | funciones {printf("%s\n\n", $$.code);};
```

- **funciones**

Esta producción, nos permite manejar la creación de una o más funciones de manera recursiva.

```
C/C++
funciones: creacionFunciones {
        limpiar_tabla_local();
        sprintf(temp, "%s", $1.code);
        $$.code = gen_code(temp);
        strcpy(current_function, "");}
    | creacionFunciones funciones {
        limpiar_tabla_local();
        sprintf(temp, "%s%s", $1.code, $2.code);
        $$.code = gen_code(temp);
        strcpy(current_function, "");};
```

- **creacionFunciones**

Esta producción nos permite gestionar la forma en la que se crean las funciones. Al igual que pasaba en main, toda la variable que se cree aquí dentro será considerada como local, por lo que tendrá que ir precedida del nombre de la función. Nombre que guardamos al igual que antes, en la variable auxiliar *"current_function"*.

```
C/C++
creacionFunciones: IDENTIF {
          strcpy(current_function, $1.code); } '(' parametros ')' '{' cuerpo '}' {if
($4.code == NULL) {
            sprintf(temp, "(defun %s ()\n\n%s\n\n)", $1.code, $7.code);
        } else {
             sprintf(temp, "(defun %s (%s)\n\n%s\n\n)", $1.code, $4.code, $7.code); }
$$.code = gen_code(temp);};
```

- **parametros**

Debido a que el cuerpo de una función puede ser exactamente igual al del "main", la última parte que nos toca gestionar son sus parámetros, ya que para que estos funcionen en Lisp, tienen que ir precedidos por el nombre de la función en su zona de paso de parámetros. Esto es para que si después se trabaja con ellos dentro de la función se identifiquen bien a esas variables. Para ello hacemos un uso auxiliar de *"param_name"*, que nos ayuda a preceder el nombre de ese parámetro con el de la función. Ya por último aceptamos la posibilidad de varios parámetros con *"r_parametros"*.

```
C/C++
parametros: { $$.code = NULL; }  // Lambda
    | INTEGER IDENTIF r_parametros {
        insertar_variable_local($2.code);
        char param_name[512];
        snprintf(param_name, sizeof(param_name), "%s_%s", current_function, $2.code);

        if ($3.code == NULL) {
            $$.code = gen_code(param_name);
        } else {
            snprintf(temp, sizeof(temp), "%s %s", param_name, $3.code);
            $$.code = gen_code(temp);}};
```

Con todo esto ya hemos acabado de explicar los 3 NT que componen la producción axioma, a continuación, pasamos a explicar el cuerpo de las funciones que es donde se ubican todas las traducciones permitidas.

- **Cuerpo**

Esta producción define las posibles estructuras que pueden aparecer dentro del cuerpo de una función o del bloque principal del programa. Se permite combinar sentencias, bucles, condicionales y declaraciones de variables locales. Además debido a que puede haber varias de estas, usamos la producción *"r_cuerpo"* para gestionar la recursividad.

```
C/C++
cuerpo: sentencia r_cuerpo {
        if ($2.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $2.code);
        }
        $$.code = gen_code(temp);
    }  // Sentencias


    | bucles r_cuerpo {
        if ($2.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $2.code);
        }
        $$.code = gen_code(temp);
    }  // Bucles
```

```
    | if r_cuerpo {
        if ($2.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $2.code);
        }
        $$.code = gen_code(temp);
    }  // If

    | locales ';' r_cuerpo {
        if ($3.code == NULL) {
            sprintf(temp, "%s", $1.code);
        } else {
            sprintf(temp, "%s%s", $1.code, $3.code);
        }
        $$.code = gen_code(temp);
    }  // Declaracion de variables;
```

- **Sentencia**

Dentro de esta producción definimos todas las sentencias permitidas para su traducción. Estas pueden ser *“printf”* de C, *“puts”* de C, una asignación de una variable *“variable = expresión”*, una asignación de un array, o el return que hay al final de una función. Esta parte del código no es muy compleja, solo decidimos que estas han de ser las normas consideradas sentencias. Cabe destacar el uso del NT “printElem” para gestionar toda la parte de la impresión que acompaña al STRING en la producción *“printf”*. Este NT será explicado más adelante.

```
C/C++
sentencia: PRINTF '(' STRING printElem ')' ';' {
        sprintf(temp, "%s", $4.code);
        $$.code = gen_code(temp);
    }

    | PUTS '(' STRING ')' ';' {
        sprintf(temp, "(print \"%s\")", $3.code);
        $$.code = gen_code(temp);
    }

    | IDENTIF '=' expresion ';' {
        if (es_variable_local($1.code)) {
            sprintf(temp, "(setf %s_%s %s)", current_function, $1.code, $3.code);  //
Variable local
        } else {
            sprintf(temp, "(setf %s %s)", $1.code, $3.code);  // Variable global
        }
```

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

    | IDENTIF '[' expresion ']' '=' expresion ';' {
        if (es_variable_local($1.code)) {
            sprintf(temp, "(setf (aref %s_%s %s) %s)", current_function, $1.code,
$3.code, $6.code);  // Vector local
        } else {
            sprintf(temp, "(setf (aref %s %s) %s)", $1.code, $3.code, $6.code);  //
Vector global
        }
        $$.code = gen_code(temp);    }
    | RETURN expresion ';' {
        sprintf(temp, "(return-from %s %s)", current_function, $2.code);
        $$.code = gen_code(temp);};
```

Como se puede observar, en alguna producción hacemos uso de una función que nosotros hemos creado llamada *"es_variable_local"*, que nos ayuda a comprobar si la variable con la que estamos trabajando está en la lista de variables globales de esa función y en caso de que no sea así consideramos que es una variable global. Esto nos sirve para decidir si hay que preceder la impresión de la variable con en nombre de la función delante o no. La explicación de esta función la realizamos más adelante donde explicamos todas las funciones externas que hemos creado.

- **PrintElem**

La producción *printElem* se encarga de gestionar los elementos que se imprimen en la instrucción printf. Su objetivo principal es generar el código correspondiente en formato Lisp para cada elemento que se desea imprimir. Esta producción permite manejar tanto expresiones como cadenas de texto, y se combina con la producción "*r_elem"* para soportar múltiples elementos de forma recursiva.

```
C/C++
printElem: ',' expresion r_elem {
        if ($3.code == NULL) {
            sprintf(temp, "(princ %s)", $2.code);
        } else {
            sprintf(temp, "(princ %s) %s", $2.code, $3.code);}
```

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

    | ',' STRING r_elem {
        if ($3.code == NULL) {
            sprintf(temp, "(princ \"%s\")", $2.code);
        } else {
            sprintf(temp, "(princ \"%s\") %s", $2.code, $3.code); }
        $$.code = gen_code(temp);};

r_elem: { $$.code = NULL; }  // Lambda
    | printElem { $$ = $1; };
```

- **If / else**

Esta regla nos permite gestionar la estructura condicional *"if"* tanto con o sin *"else"*. Tal y como indicaron los profesores en clase, haremos uso de *"progn"* tanto si hay más de una línea de código o solo una tanto para la parte del *"if"* como la del *"else"*. Esta producción tampoco es compleja, ya que simplemente evaluamos la estructura del *"if"* y si contiene o no *"else"* para imprimirlo o no.

```
C/C++
if: IF '(' expresion ')' '{' cuerpo '}' else {
        if ($8.code == NULL) {
            sprintf(temp, "(if %s\n\t(progn %s)\n)\n\n", $3.code, $6.code);
        } else {
                    sprintf(temp,  "(if  %s\n\t(progn  %s)\n%s\n)\n\n",  $3.code,  $6.code,
$8.code);
        }
        $$.code = gen_code(temp);
    }
else: { $$.code = NULL; }  // Lambda
    | ELSE '{' cuerpo '}' {
        sprintf(temp, "(progn %s)\n", $3.code);
        $$.code = gen_code(temp); } ;
```

- **Variables locales**

Las variables locales se manejan exactamente igual que como se ha manejado previamente la variables globales. Tenemos una producción *"locales"* que gestiona la creación de una o más variables locales.

```
C/C++
locales: creacionVarLocales r_varLocales {
    if ($2.code == NULL) {
        $$ = $1;
    } else {
        sprintf(temp, "%s %s", $1.code, $2.code);
        $$.code = gen_code(temp);
    }
}
r_varLocales: { $$.code = NULL; }  // Lambda
    | locales { $$ = $1; };
```

Cabe destacar que esta es la única producción que es un poco diferente en comparación a la variables globales, ya que las locales no se imprimen en su producción, si no cuando acaba el main, por eso no tenemos que hacer un uso auxiliar de una función para imprimir, simplemente las generamos de forma recursiva.

El resto del código es exactamente igual, a diferencia de que cuando generamos por ejemplo un *"setq"*, precedemos el nombre de la variable con el nombre actual de la función en la que esté para dejar claro que es una variable local. Todo eso es posible, ya que cuando entramos en la producción de una función guardamos su nombre en una variable auxiliar *"current_function"* que nos permite acceder al nombre cuando lo necesitemos. Esto es gracias a la función auxiliar *"insertar_variable_local",* que nos ayuda a insertar la variable local dentro de una lista para que, cuando se quiera operar con ella, se haga una buena distinción entre variables locales y globales. Esta función también se explicará en la zona de funciones auxiliares.

- **Bucles While y For**

Para el bucle while, hacemos una regla muy sencilla, donde se evalúa si hay una expresión como condición para el while, y si después viene un cuerpo que es donde estará todo el código a ejecutar dentro del bucle.

```
C/C++
bucles: WHILE '(' expresion ')' '{' cuerpo '}' {
        sprintf(temp, "(loop while %s do\n\t%s\n)\n\n", $3.code, $6.code);
        $$.code = gen_code(temp);};
```

Para el bucle for hemos tenido que crear dos NT auxiliares para garantizar su buena estructura.

```
C/C++
bucles: FOR '(' creacionBucle ';' expresion ';' inicializadorDec ')' '{' cuerpo '}' {
            sprintf(temp, "%s(loop while %s do\n\t%s\n\t%s\n)\n\n", $3.code, $5.code,
$10.code, $7.code);
        $$.code = gen_code(temp); };
```

- **creacionBucle**

Esta regla nos permite que en el primer argumento del for haya un *"int variable = number"* tal y como ocurre en C.

```
C/C++
creacionBucle: IDENTIF '=' NUMBER {
        if (es_variable_local($1.code)) {
            sprintf(temp, "(setq %s_%s %d)\n", current_function, $1.code, $3.value);
        } else {
            sprintf(temp, "(setq %s %d)\n", $1.code, $3.value);
        }
        $$.code = gen_code(temp);};
```

- **inicializadorDec y opBucle**

Estos dos NT auxiliares nos ayudan a comprobar que el último parámetro del bucle for tenga la estructura "variable = variable +-/* NUMBER", ya que estas son las únicas operaciones que se permiten en este tipo de bucles.

```
C/C++
inicializadorDec: IDENTIF '=' IDENTIF opBucle {
        if (es_variable_local($1.code)) {
            if (es_variable_local($3.code)) {
                sprintf(temp, "(setf %s_%s (%s %s_%s %d))", current_function, $1.code,
$4.op, current_function, $3.code, $4.value);
            } else {
                  sprintf(temp, "(setf %s_%s (%s %s %d))", current_function, $1.code,
$4.op, $3.code, $4.value);}
        } else {
            if (es_variable_local($3.code)) {
                        sprintf(temp, "(setf %s (%s %s_%s %d))", $1.code, $4.op,
current_function, $3.code, $4.value);
            } else {
```

```
                    sprintf(temp, "(setf %s (%s %s %d))", $1.code, $4.op, $3.code,
$4.value);} }
        $$.code = gen_code(temp);};

opBucle: '+' NUMBER { $$.op = "+"; $$.value = $2.value; }
        | '-' NUMBER { $$.op = "-"; $$.value = $2.value; }
        | '/' NUMBER { $$.op = "/"; $$.value = $2.value; }
        | '*' NUMBER { $$.op = "*"; $$.value = $2.value; };
```

- **Expresión**

La producción "*expresion*" define las operaciones aritméticas, lógicas y comparativas del lenguaje. Utiliza "*termino*" como base para construir expresiones más complejas. Hemos considerado no incluir más código, ya que sería repetir constantemente la misma estructura.

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

- **Término**

La producción "*termino*" define los elementos básicos de una expresión, como un operando o un número con signo.

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

- **Operando**

La producción operando define los elementos más simples de una expresión, como variables, números, llamadas a funciones o vectores.

```C/C++
operando: IDENTIF {
        if (es_variable_local($1.code)) {
            sprintf(temp, "%s_%s", current_function, $1.code);
            $$.code = gen_code(temp);
        } else {$$.code = $1.code;}}
    | NUMBER {
        sprintf(temp, "%d", $1.value);
        $$.code = gen_code(temp);}
    | '(' expresion ')' { $$ = $2; }
    | IDENTIF '(' argumentos ')' {
        if ($3.code == NULL) {
            sprintf(temp, "(%s)", $1.code);
            $$.code = gen_code(temp);
        } else {sprintf(temp, "(%s %s)", $1.code, $3.code);
            $$.code = gen_code(temp);}  }
    | IDENTIF '[' expresion ']' {
        if (es_variable_local($1.code)) {
            sprintf(temp, "(aref %s_%s %s)", current_function, $1.code, $3.code);
        } else {sprintf(temp, "(aref %s %s)", $1.code, $3.code);}
        $$.code = gen_code(temp);};
```

- **Argumentos**

Esta producción maneja la lista de argumentos que se pasan a una función. Puede ser una lista vacía (lambda) o una lista de una o más expresiones. Todo esto se maneja con recursividad haciendo uso del NT auxiliar *“r_argumentos*”.

```C/C++
argumentos: { $$.code = NULL; }  // Lambda
    | expresion r_argumentos {
        if ($2.code == NULL) {
            $$ = $1;
        } else {
            sprintf(temp, "%s %s", $1.code, $2.code);
            $$.code = gen_code(temp); }};
r_argumentos: { $$.code = NULL; }  // Lambda
    | ',' expresion r_argumentos {
        if ($3.code == NULL) {
            $$ = $2;
        } else {
            sprintf(temp, "%s %s", $2.code, $3.code);
            $$.code = gen_code(temp); }};
```

FUNCIONES AUXILIARES PARA VARIABLES LOCALES

Para asegurarnos de que todas las variables locales funcionasen correctamente, hemos hecho uso de 3 funciones auxiliares, *“insertar_variable_local”*, *“es_variable_local”* y *“limpiar_tabla_local”*. Estas funciones como su propio nombre indica, se encargan de añadir a una lista las variables locales, de comprobar si la variable elegida es local o no, y de limpiar la tabla de variables locales una vez que la función acaba. Con todo esto, nos aseguramos de que no se confundan variables locales y globales, y además, nos ayudan a que las variables locales siempre estén precedidas del nombre de la función a la que pertenecen.

```C/C++
char *tabla_local[100];  // tabla simple
int num_locales = 0;

void insertar_variable_local(char *nombre) {
    tabla_local[num_locales++] = strdup(nombre);
}

int es_variable_local(char *nombre) {
    for (int i = 0; i < num_locales; i++) {
        if (strcmp(nombre, tabla_local[i]) == 0) return 1;
    }
    return 0;
}

void limpiar_tabla_local() {
    for (int i = 0; i < num_locales; i++) {
        free(tabla_local[i]);
    }
    num_locales = 0;
}
```

DESARROLLO DE LA GRAMÁTICA DEL BACKEND / BACK.Y (Lisp → Forth)

A continuación, vamos a explicar el código y desarrollo de la gramática del back.y para traducir una entrada en Lisp a lenguaje Forth.

- **Axioma**

El axioma define la estructura principal del programa y cómo se organiza. Todo el programa debe ajustarse a las reglas definidas en esta producción. En este caso, el axioma permite que el programa esté compuesto por tres tipos principales de bloques: declaraciones de variables (*creacionVar*), el bloque principal (*main*) y bloques de transcripción (*transcribir*). Estos bloques puede aparecer en cualquier orden y pueden repetirse, gracias a la recursividad que nos proporciona *r_axioma*.

```
C/C++
axioma: creacionVar { ; } r_axioma { ; }
    | main  { ; } r_axioma { ; }
    | transcribir { ; } r_axioma { ; }
;

r_axioma: {;} //Lambda
    | axioma { $$ = $1; } //axioma
;
```

- **Transcribir**

Esta producción se utiliza para procesar una instrucción específica en el programa. Su propósito es identificar la palabra clave MAIN dentro de un paréntesis, e imprimir main en la salida. Esto hace que podamos transcribir la instrucción “main” a Forth para que se ejecute el programa.

```
C/C++
transcribir: '(' MAIN ')' {
        printf("main\n") ;};
```

- **Main**

El bloque ‘main’ es esencial para estructurar el programa y garantizar que las instrucciones se ejecuten en el orden correcto.
En esta producción se utiliza como entrada la palabra clave DEFUN seguida de MAIN y unos paréntesis vacíos, que no recibe parámetros y contiene un cuerpo. La entrada se traduce a “ : main <cuerpo> ; “

```
C/C++
main: '(' DEFUN MAIN '(' ')' cuerpo ')' {
    sprintf(temp, ": main\n%s\n;\n", $6.code) ;
    $$.code = gen_code(temp) ;
    printf("%s", $$.code) ;};
```

- **Cuerpo**

El cuerpo define la estructura de un conjunto de instrucciones, como el cuerpo de una función (como hemos visto en la función ‘main), o un bucle. Permite combinar múltiples sentencias de forma recursiva, como podemos ver en el bloque de código proporcionado.

Si hay más de una sentencia, se concatenan las instrucciones generadas por la primera sentencia con las del resto del cuerpo. Si solo hay una sentencia, el código generado para el cuerpo es simplemente el código de esa sentencia.

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

- **Sentencia**

Esta producción define los diferentes tipos de instrucciones que pueden aparecer. Cada tipo de sentencia tiene su respectiva producción posteriormente, y *sentencia* simplemente actúa como un punto de unión para todas las sentencias. Los tipos de sentencias que maneja son: *imprimir* (para imprimir texto o valores en la salida); *while* (bucle que se ejecuta mientras se cumpla una condición); *if*

(condicional que se ejecuta dependiendo de si una condición es verdadera o falsa); y *variables* (declaraciones o asignaciones de valores a variables).

```
C/C++
sentencia: imprimir { $$ = $1; }
    | while { $$ = $1; }
    | if { $$ = $1; }
    | variables { $$ = $1; };
```

- **Variables globales**

Para manejar el tema de las variables, hemos utilizado dos producciones: *variables* y *creacionVar*.

- **Variables**

El bloque variables se utiliza para manejar la asignación de valores a variables. Este bloque se activa cuando se encuentra una instrucción con la palabra clave SETF, que indica una operación de asignación. Le sigue una variable (*IDENTIF*) y una *expresion*. En Forth se realiza el operador' *!* ' para la asignación como se puede ver en la traducción.

```
C/C++
variables: '(' SETF IDENTIF expresion ')' {
        sprintf(temp, "%s %s !", $4.code, $3.code) ;
        $$.code = gen_code(temp) ;};
```

- **CreacionVar**

La producción *creacionVar* se utiliza para declarar variables globales en el programa. Este bloque se activa cuando se encuentra una instrucción con la palabra clave SETQ, que indica la creación de una nueva variable, seguido de una variable y un número (*IDENTIF* y *NUMBER* respectivamente).

```
C/C++
creacionVar:  '(' SETQ IDENTIF NUMBER ')' {
        sprintf(temp, "variable %s\n%d %s !\n", $3.code, $4.value, $3.code);
        $$.code = gen_code(temp) ;
        printf("%s", $$.code) ;};
```

- **While**

Esta producción while se encarga de traducir los bucles. En el proceso de traducción, el traductor genera una estructura que comienza con BEGIN, seguida de la evaluación de la condición con WHILE, y finaliza con REPEAT para indicar que el cuerpo del bucle debe repetirse.

```
C/C++
while: '(' LOOP WHILE expresion DO cuerpo ')' {
        sprintf(temp, "BEGIN\n\t%s\nWHILE\n\t%s\nREPEAT\n", $4.code, $6.code);
        $$.code = gen_code(temp); };
```

- **If / Else**

Esta producción se encarga de manejar las sentencias condicionales. Permite representar tanto condiciones simples, donde solo se ejecuta un bloque de código, como condiciones más complejas que incluyen un bloque alternativo para el caso en que la condición sea falsa (else).

En el caso de una condición sin else, el traductor genera palabras clave como IF y THEN para estructurar el flujo de control en el lenguaje destino. La condición se evalúa primero, seguida del bloque de instrucciones, y finalmente se cierra con THEN para indicar el final del bloque condicional.

Cuando la condición incluye un bloque else, se ejecuta el segundo bloque de instrucciones. El traductor utiliza ELSE para separar los dos bloques y THEN para cerrar la estructura condicional.

```
C/C++
if: '(' IF expresion '(' PROGN cuerpo ')' ')' {
    sprintf(temp, "%s IF\n\t%s\nTHEN\n", $3.code, $6.code);
    $$.code = gen_code(temp);}
    | '(' IF expresion '(' PROGN cuerpo ')' '(' PROGN cuerpo ')' ')' {
        sprintf(temp, "%s IF\n\t%s\nELSE\n\t%s\nTHEN\n", $3.code, $6.code, $10.code);
        $$.code = gen_code(temp);};
```

- **Imprimir**

Esta producción traduce una instrucción que imprime una cadena de texto literal. Hay dos casos: para *PRINT* y para *PRINC*. Para la palabra clave de *PRINT*,

le sigue un *STRING*, que representa una cadena de texto. Para la traducción se traduce en el siguiente formato ‘ *." <cadena>"* ‘, donde “." es el operador de impresión en Forth.
Para la palabra clave de *PRINC*, le sigue una expresión, y se utiliza el operador ” . “ para imprimir el resultado, al igual que en el caso anterior.

```
C/C++
imprimir: '(' PRINT STRING ')' {
        sprintf(temp, ".\" %s\"", $3.code);
        $$.code = gen_code(temp);}
    | '(' PRINC expresion ')' {
        sprintf(temp, "%s .", $3.code);
        $$.code = gen_code(temp);};
```

- **Expresión**

La producción *expresion* define las operaciones aritméticas, lógicas y comparativas. Una expresión puede ser simplemente un *operando*, como un número o una variable. En este caso, no se realiza ninguna operación adicional. El resto de expresiones se realizan de la misma forma que en el *trad* pero traduciendo a su respectivo formato (en lugar de NOT se pone 0= , por ejemplo). Solo se incluye la suma, pero para el resto de expresiones es igual, pero no se incluye por tema de extensión.

```
C/C++
expresion: operando { $$ = $1; }
    |'+' expresion expresion {
        sprintf(temp, "%s %s +", $2.code, $3.code);
        $$.code = gen_code(temp);};
```

- **Término**

Al igual que en el fichero trad, esta producción representa algunos de los elementos más básicos de cualquier expresión. Este NT se encarga de generar el NT operando y de procesar la negación de un operando.

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

- **Operando**

Al igual que en trad.y, aquí se recogen las producciones más básicas del programa. Aquí se generan las variables, los números y las expresiones que vienen entre paréntesis, para permitir expresiones anidadas unas dentro de otras, respetando su prioridad.

```C/C++
operando: IDENTIF { sprintf(temp, "%s @", $1.code);
        $$.code = gen_code(temp); }
    | NUMBER { sprintf(temp, "%d", $1.value);
            $$.code = gen_code(temp); }
    | '(' expresion ')' { $$ = $2; }
;
```

ANEXO: BATERÍA DE PRUEBAS

Durante todo el transcurso de la práctica, hemos ido realizando pruebas para comprobar que cada funcionalidad implementada estaba correctamente añadida. Estas pruebas han ido desde cosas muy simples hasta algunas mucho más complejas para intentar llevar nuestro trad y back al máximo de sus capacidades. No obstante, queremos destacar que en esta parte si hemos realizado de vez en cuando uso de IA generativa para la creación de pruebas. Esto se debe a que hemos considerado que la IA podría generar código más rápido que nosotros, y que además este podría ser más complejo ya que le podíamos pedir que solo nos diese casos muy extremos y muy rebuscados. Para finalizar, hemos pensado solo en compartir las pruebas más largas y las que pudiesen ser más interesantes.