Universidad Carlos III

Procesadores del Lenguaje

2023-24

Práctica final

Ingeniería Informática, Tercer Curso

César López Mantecón (NIA: 100472092, e-mail: 100472092@alumnos.uc3m.es)

Paula Subías Serrano (NIA: 100472119, e-mail: 100472119@alumnos.uc3m.es)

Prof. Juan Manuel Alonso Weber, Jorge Martínez Gil

Grupo: 81

Índice

Frontend: traducción de C a Lisp

Eliminación producción sentencia expresión

Se ha eliminado de las producciones del no terminal *sentencia* la producción:

`Sentencia → expresion`

Así, la gramática ya **no acepta sentencias que no sean asignaciones o impresiones,** como 3 + 3;.

Impresión de varias expresiones seguidas

Se ha modificado la gramática cambiando la producción del no terminal *sentencia* encargada de la traducción de las impresiones, añadiendo **un nuevo no terminal *rest_print* que permite concatenar varios prints en la misma línea**. Además, se han añadido paréntesis a la producción para solo permitir sentencias con la estructura *@(expresion)*, ya que se considera impresión como una función. Así, la producción resultante es:

`Sentencia → ‘@’ ‘(’ expresion rest_print ‘)’`

Para poder imprimir los valores de *rest_print* en la acción semántica la llamada a la función *sprintf* queda de la siguiente forma

```C/C++
sprintf (temp, "(print %s) %s", $3.code, $4.code)

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

Rest_print cuenta con dos producciones:

`Rest_print → λ | ‘,’ expresion rest_print`

En la primera producción, la acción semántica asociada es asignar a la cadena del no terminal una cadena vacía (i.e. “”). Esto es necesario para que no imprima nada en la acción semántica asociada a la producción asociada a la regla de *Sentencia*.
En la segunda producción, contamos con una acción semántica idéntica a la regla de *Sentencia* explicada en este mismo apartado.

Variables globales

Se ha ampliado la gramática para reconocer dos tipos de sentencias de asignación, con el objetivo de posibilitar la definición de variables globales. Con este cambio, ***sentencia* ahora**

tiene la posibilidad de ser una declaración (esto se elimina más adelante al completar puntos posteriores), además de las dos funciones antes mencionadas.

```
sentencia → INTEGER IDENTIF ‘=’ NUMBER

sentencia → INTEGER IDENTIF
```

La primera sentencia corresponde a la declaración de una variable con el valor de NUMBER asignado. La segunda, en cambio, solo reconoce la declaración de variable sin valor, y por lo tanto se le asigna por defecto un 0.
La traducción se hará a través de las acciones semánticas:

```C/C++
//sentencia -> INTEGER IDENTIF ‘=’ NUMBER

sprintf (temp, "(setq %s %d)", $2.code, $4.value)

$$.code = code_gen(temp) ;

//sentencia -> INTEGER IDENTIF

sprintf (temp, "(setq %s 0)", $2.code)

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

Una vez realizada la estructura base, se modifica la gramática para contemplar la definición de varias variables seguidas. Para esta tarea se añade un símbolo no terminal *rest_declar*, que permite reconocer un número ilimitado de declaraciones después de la primera. *rest_declar* cuenta con dos producciones recursivas y una producción de la cadena vacía usada para finalizar las sentencias. La gramática resultante es:

```
sentencia → INTEGER IDENTIF ‘=’ NUMBER rest_declar
sentencia → INTEGER IDENTIF rest_declar
rest_declar → ‘,’ IDENTIF rest_declar
rest_declar → ‘,’ IDENTIF ‘=’ NUMBER rest_declar
rest_declar → λ
```

Las acciones semánticas se modifican para añadir los datos provenientes de *rest_declar*, concatenando así a la cadena las declaraciones posteriores:

```C/C++
//sentencia -> INTEGER IDENTIF ‘=’ NUMBER rest_declar

sprintf (temp, "(setq %s %d) %s", $2.code, $4.value, $5.code)
```

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

//sentencia -> INTEGER IDENTIF rest_declar

sprintf (temp, "(setq %s 0) %s", $2.code, $6)

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

Las acciones semánticas de las producciones de *rest_declar* son idénticas a su equivalente en las producciones de declaración de sentencia. En el caso de la producción de la cadena vacía, se le asigna a $$.code, cadena del no terminal, un *string* vacío “”.

Función main y redefinición estructura

Para seguir la estructura descrita en el enunciado, ha sido necesario modificar varias reglas para lograr una **gramática estructurada**.

El axioma da lugar a la estructura de programa, siendo necesario añadir 2 símbolos no terminales más: *Declaraciones* y *codigo*.

`axioma → declaraciones MAIN ‘(’ ‘)’ ‘{‘ codigo ‘}’`

Las reglas antes contenidas en el axioma han sido incluídas en *codigo*. De esta forma, Además, las reglas para la declaración de variables globales se han movido desde *Sentencia* a *Declaraciones*.

Con estas modificaciones el lenguaje descrito por la gramática es distinto al reconocido inicialmente. Las consecuencias más destacables son las siguientes:

- Una sentencia sólo puede ser una asignación o una impresión. No puede ser una declaración de variable global. Además, una sentencia sólo se podrá dar en el cuerpo de la función *main*.
- La declaración de variables globales deberá ser antes de la función *main*.
- El programa más corto reconocido por la gramática será una función *main* con una única sentencia.

```
C/C++
main(){setencia; /*lambda*/}
```

Imprimir cadenas literales: puts

Para implementar esta especificación se ha añadido una **nueva regla a *Sentencia*** y la palabra ***puts*** **al conjunto de palabras reservadas** del lenguaje. La regla añadida ha sido la siguiente.

`Sentencia → PUTS ‘(‘ STRING ‘)’`

A esta regla se le asocia la siguiente acción semántica:

```C/C++
sprintf (temp, "(print \"%s\")", $3.code) ;
$$.code = gen_code (temp) ;
```

De esta forma se traducirá la *Sentencia* con la estructura que describe el enunciado.

Implementación printf

Se ha añadido la palabra *printf* al conjunto de palabras reservadas; y se ha sustituído la regla `Sentencia → ‘@’ ‘(’ expresion rest_print ‘)’` por la siguiente:
`Sentencia → PRINTF '(' STRING ',' expresion rest_print ')'`

Las acciones semánticas asociadas a esta regla permiten construir la traducción de la sentencia. Son iguales a las de la regla que se ha sustituído, con una pequeña variación en los índices debido a la inclusión del token STRING. Tal y como indica el enunciado, el token STRING se reconoce pero se ignora en la traducción.

Operadores lógicos

Para añadir los operadores lógicos, ha sido necesario añadir **nuevos tokens** para los compuestos e incluirlos en la lista de palabras reservadas. Además, al símbolo *Sentencia* se le han añadido las reglas para reconocer operaciones lógicas entre dos expresiones siguiendo el mismo formato que las operaciones aritméticas incluídas en el código de apoyo.

Para establecer la prioridad de operadores y su asociatividad se ha utilizado el siguiente código de bison:

```Unset
%right '='                      // mínima preferencia
%left OR                        // muy mucho menor
%left AND                       // muy menor
%left EQUAL NOTEQ               // aun aun menor
%left '<' LEQ '>' GEQ           // aun menor
```

```
%left '+' '-'                    // menor orden de precedencia
%left '*' '/'                    // orden intermedio
%left UNARY_SIGN                 // máxima preferencia
```

While

Se ha añadido al no terminal *Codigo* una **nueva regla de producción** para derivar un bloque *while*.

`Codigo → WHILE ‘(‘ Expresion ‘)’ ‘{‘ Codigo ‘}’ R_expr`

Para la acción semántica, construímos la cadena siguiendo la estructura indicada en el enunciado.

```C/C++
sprintf(temp, "(loop while %s do\n%s)\n%s", $3.code, $6.code, $8.code);
$$.code = gen_code(temp);
```

If

Para un bloque de código condicional seguimos la misma aproximación que en el apartado anterior. La regla añadida a *Codigo* es la siguiente:

`Codigo → WHILE ‘(‘ Expresion ‘)’ ‘{‘ Codigo ‘}’ Est_else R_expr`

Donde *Est_else* es un no terminal anulable que recoge una posible sentencia *else* consecutiva al *if*.

`Est_else → λ`

`Est_else → ELSE ‘{‘ Codigo ‘}’`

En las acciones semánticas traducimos este bloque de esta forma:

`(progn <sentencia1> <sentencia2>... <sentenciaN>)`

Así, incluímos la posibilidad de tener bloques con más de una sentencia.

For

El *bucle for* se debe traducir a una instrucción ***while*** donde la **condición de salida coincida con la expresión incluída en la cabecera del *for***, y exista una variable índice que incrementa o decrementa en cada iteración. Añadimos entonces la impresión de la

inicialización de la variable en la línea superior al bucle *while*, y se incluye al final del bucle antes de la condición de salida una sentencia de incremento o decremento, en la que se asume que la variable utilizada será la del índice. El código se añadirá dentro del bucle *while* de la misma manera que para los bucles comunes. Esto se ha traducido en la siguiente regla en nuestra gramática:

`Codigo → FOR ‘(‘Inicializar;Expresion;IDENTIF‘=’Incr_decr‘)’‘{‘Codigo’}’`

Hemos añadido **dos nuevos no terminales** para satisfacer las restricciones de este tipo de sentencia:

- Inicializar: deriva en una asignación.
  `Inicializar → IDENTIF = Expresion`
- Incrr_decr: deriva en una expresión de tipo suma o resta
  `Incr_decr → Expresion + Expresion`
  `Incr_decr → Expresion - Expresion`

De esta forma, sin embargo, podrían surgir algunos problemas. La expresión, aunque suma o resta, no tiene por qué ser necesariamente un incremento. Además, la variable (*IDENTIF*) afectada por el incremento/decremento puede ser distinta a la que se inicializa.

Para la traducción se emplean las siguientes acciones semánticas:

```
C/C++
{sprintf(temp, "%s(loop while %s do\n%s(setf %s-%s%s)) \n%s", $3.code,
$5.code, $12.code,nombre_funcion, $7.code,$9.code, $14.code);
$$.code = gen_code(temp);}

{ sprintf (temp, "(setf %s-%s %s)", nombre_funcion,$1.code, $3.code) ;
$$.code = gen_code (temp) ; }

{ sprintf (temp, "(+ %s %s)", $1.code, $3.code) ;
$$.code = gen_code (temp) ; }

{ sprintf (temp, "(- %s %s)", $1.code, $3.code) ;
$$.code = gen_code (temp) ; }
```

Un bucle *for* se traduce como un bucle *while* de LISP. El incremento se traduce como última expresión del bucle.

Variables locales

La declaración de variables locales debe ser al principio de la función. **Se han generalizado las declaraciones de variables** en el no terminal *declaracion_variables*.

`dec_var → lambda`

```
dec_var[1] → INTEGER IDENTIF '=' NUMBER rest_declar ';' dec_var

dec_var → INTEGER IDENTIF rest_declar ';' dec_var

dec_var → INTEGER IDENTIF '[' NUMBER']' rest_declar_vector ';' dec_var
```

Este tipo de variables no existen en LISP. No obstante, para simularlas **concatenamos el nombre de la función a la variable** de forma que esta pueda distinguirse de otras variables globales o parámetros homónimos. Para poder distinguirlas hacemos uso de una estructura *t_list* y 3 funciones que nos permiten la búsqueda, inserción y vaciado de dicha lista. Toda variable local será almacenada en la estructura de forma que, ante el uso de una variable dentro de la función, si existe dicha variable en la estructura se concatena el nombre de la función. Para asegurarnos de guardar el nombre de la función nada más reconocerlo, insertamos una acción semántica en mitad de la regla descrita en bison de la siguiente forma:

```C/C++
declaracion_func: {$$.code = "";}
   | IDENTIF '(' args ')' '{' {nombre_funcion = gen_code($1.code);}
declaracion_variables codigo '}'  {printf("(defun %s (%s)\n%s%s)\n",
$1.code, $3.code, $7.code, $8.code); remove_all(&var_list);}
declaracion_func
             ;
```

El nombre de la función se añade a la variable a través de acciones semánticas. Se añade una como ejemplo. Las demás son similares, corrigiendo los índices de *$* para la regla.

```C/C++
{
       char aux[1026] = "";
       if (strcmp(nombre_funcion, "")) {
             sprintf(aux, "%s_", nombre_funcion); insert(&var_list,
       $2.code);
       }
       sprintf(temp, "(setq %s%s 0) %s \n%s", aux, $2.code, $3.code,
       $5.code);
       $$.code = gen_code(temp);
}
```

De la misma forma, ha sido necesario modificar las producciones de *Sentencia* y *Operando* que incluyen variables para concatenar el nombre de la función a la que pertenecen.

[1] *Se han reducido algunos nombres de símbolos terminales para mejorar la legibilidad del documento. En el código se corresponde al terminal declaracion_variables.*

Código embebido: traducción en el reconocimiento de funciones

Nuestro traductor debe ser capaz de reconocer expresiones de *Lisp* dentro del código a traducir. Estas vienen precedidas por la cadena "//@". **Este tipo de expresiones se deben transcribir directamente a la salida del traductor.**
En el código de apoyo esta casuística ya está recogida en el lexer. Para evitar que se rezaguen las escrituras de código embebido (y evitar el uso exponencial de memoria) hemos modificado el lugar donde se realiza una traducción. **Las traducciones se realizan cada vez que se cierra una función**, en lugar de únicamente en el axioma.

```C/C++
// al reconocer una función
{printf("(defun %s (%s)\n%s%s)\n", $1.code, $3.code, $7.code,
$8.code);}
```

Además, las variables globales podrían derivar en el mismo problema. Es por esto que se traducen directamente al terminar de reconocerse. Para contener esta acción semántica manteniendo la generalización de las declaraciones, se ha añadido un nuevo no terminal de forma que la gramática queda de la siguiente forma:

```
axioma → var_locales declaracion_func MAIN '('')' '{'
           declaracion_variables codigo '}'

var_globales → declaracion_variables
```

La impresión del código traducido se hace en los símbolos *var_globales* y *declaracion_func*. En el axioma sólo se mantiene la impresión de la función *main* para mantener su obligatoreidad.

Declaración funciones

La estructura de los códigos reconocidos por el traductor es la siguiente:

```Unset
[declaración variables globales]
[declaración funciones]
main () {
      sentencia;
      [sentencia;...]
}
```

Por esto, reconoceremos el bloque de *declaración funciones* en el axioma. Se encapsula dicho bloque en un no terminal *Declaracion_func*. Esto se traduce en la **modificación de la producción del axioma y la adición de las producciones para el nuevo símbolo.**

```
Declaracion_func → λ
Declaracion_func → IDENTIF ‘(‘ Args ‘)’ ‘{‘ var_locales codigo ‘}’
Args → INTEGER IDENTIF Rest_args
Args → λ
Rest_args → ‘,’ INTEGER IDENTIF Rest_args
Rest_args → λ
```

Para reconocer una sentencia *return* en cualquier punto del código se ha añadido a sentencia una producción:

```
Sentencia → RETURN expresion
```

En consecuencia, se ha añadido un nuevo *token* a la gramática (*Return*).
Este tipo de sentencias se traducirán de la siguiente manera:

```
C/C++
{sprintf(temp, "(return-from %s %s)", nombre_funcion,$2.code);
$$.code = gen_code (temp) ;}
```

Hemos decidido **traducir cualquier return al operador *return-from*** de Lisp debido a las limitaciones semánticas de la práctica. De esta forma, independientemente del lugar donde se coloque el return se traducirá a una sentencia con el comportamiento adecuado.
Tras terminar la declaración de funciones, se añade la posibilidad de llamar a una función con parámetros desde cualquier otra función. Para esto, se crea una nueva producción en el no terminal *sentencia* que acepta un identificador con una serie de parámetros, para lo que se ha creado otro no terminal. La gramática resultante es:

```
Sentencia → IDENTIF ‘(‘ expresion rest_params ‘)’
Sentencia → IDENTIF ‘(‘ ‘)’
rest_params → λ | ‘,’ expresion rest_params
```

Vectores

Los vectores son un tipo de variable con una estructura concreta distinta a la del resto de variables. Es por esto que se han **tratado por separado**.
Para reconocerlos y traducirlos ha sido necesario incluir nuevas reglas para tratar su estructura por separado en distintos no terminales: *sentencia*, para asignaciones; *expresion*, para su uso en operaciones; y *declaración*, para declararlos adecuadamente.

```
dec_var→ INTEGER IDENTIF ‘[‘ NUMBER ‘]’ Rest_declar_vector; dec_var
operando → IDENTIF ‘[’ expresion ‘]’
sentencia → vector ‘=’ expresion
```

Con esto, nuestro traductor reconoce una estructura de *array* de C y la traduce a su equivalente en Lisp haciendo uso del operador *mak-array*. En cuanto a la recuperación de un valor (e.g. arr[3]), hacemos uso del operador *aref*.

Backend: traducción de Lisp a Forth

Este programa **no necesita validar el lenguaje de entrada** ya que esa tarea la habrá desempeñado el *frontend*. Con esto en mente, podemos definir una gramática que acepte un lenguaje más amplio que el lenguaje de entrada, lo que facilitará el diseño.
La gramática comenzará reconociendo un bloque de código entre paréntesis.

```
axioma → '(' bloque ')' codigo
codigo → '(' bloque ')' codigo | λ
```

Mediante estas dos producciones aceptamos una sucesión de uno o más bloques de código. Un bloque representará cada una de las posibles estructuras de código que podemos leer en el lenguaje intermedio.

```
bloque → sentencia
       | declaracion
       | definicion_funcion
       | while_loop
       | if_expr
```

Se han resumido las estructuras de cada bloque en un único símbolo no terminal con el fin de mejorar la legibilidad del documento. En el código se puede ver como algunos de estos símbolos se expanden a las estructuras concretas de cada bloque.

Declaración variables globales

Se inicia el traductor con la creación de variables, que incluye su declaración, su asignación y su uso como operando. Estas se separan gracias al uso **diferenciado de setq y setf** en la traducción a Lisp, utilizando la primera sólo en declaraciones y la segunda solo en asignaciones.
Cada variable global **se debe traducir a dos líneas**, una **declaración** y una **asignación** de un valor.

`(setq a 0)` se traduce a `variable a` y `0 a !`

Sentencia

Una sentencia encapsula las siguientes instrucciones: *setf, print, prin1, return-from* y una *llamada a una función*. Esto se hace mediante 5 producciones:

```
sentencia → SETF IDENTIF expresion
          | PRINT STRING
          | PRIN1 prin1_arg
          | RETURN '-' FROM IDENTIF expresion
          | funcion
```

Asignación

La asignación se traduce a *forth* pasando la notación prefija a notación postfija y sustituyendo el operador *setf* por el operador *!* de forth. Además, si nos encontramos en el interior del cuerpo de una función comprobamos si la variable afectada es un argumento de la función

(que en nuestro programa se tratan como variables siguiendo una aproximación similar al *frontend*, más adelante en el documento indagaremos en ello), en cuyo caso corregimos el nombre del mismo concatenando el la cadena "*arg_<nombre_funcion>_*" al comienzo del nombre del argumento.

Print y prin1

Las instrucciones print y prin1 permiten imprimir por pantalla información. En el caso de ***print*** vendrá seguida de **una cadena de caracteres** a la que llamaremos *string*; en este caso se traduce a la sintaxis de forth sustituyendo la instrucción por la sentencia *." string"*. En el caso de ***prin1***, vendrán **una o varias expresiones o cadenas de caracteres**. Las cadenas de caracteres se traducen igual que en el caso anterior, el resto de expresiones se imprimen por pantalla añadiendo detrás de su traducción un caracter '.'.

Return-from

Esta sentencia **se traduce a dos líneas de forth**. La primera, traduce la expresión a retornar para que su valor se almacene en la cima de la pila. La segunda línea será un exit de forma que se detenga la ejecución de la función en el punto deseado. De esta forma podemos tener una **instrucción *return* en cualquier punto de la función**.

`(return-from <f1> <expr>)` se traduce a `<epxr> exit`

Funcion

Este bloque se encarga de traducir las llamadas a funciones ignorando su valor de retorno. En el no terminal *funcion* reconocemos una llamada a una función y la traducimos de la siguiente forma:

`(funcion_name arg1 arg2 …)` se traduce a `arg1 arg2 … funcion_name`

Definición de funciones genéricas

Un bloque para la declaración de funciones viene encabezado por la instrucción *defun*, seguido de una lista de parámetros y un bloque de código. Este bloque de código se puede dividir en dos partes: la **declaración de variables locales** y una **serie de instrucciones**.

A continuación describimos cómo hemos enfrentado cada una de las problemáticas que presenta la declaración de funciones.

```
fucnion → defun IDENTIF '(' fun_arg ')' codigo
fun_arg → IDENTIF fun_arg | λ
```

Además, hemos incluído una acción semántica para guardar el nombre de la función en una variable global.

Funciones con valor de retorno

En *forth* no existe una instrucción de retorno. *Retornar un valor* se debe traducir a un código que deje dicho valor en la *cima de la pila*. Para esto, en el conjunto de producciones sentencia

ya contempla el reconocimiento y traducción de la instrucción (véase sentencia > return-from).

Funciones con parámetros

Para poder traducir funciones con parámetros hemos seguido una aproximación similar a la forma de traducir las variables locales a código intermedio. **Un argumento se traducirá como una variable declarada antes de declarar la función** con el siguiente nombre: *arg_<nombre_funcion>_<nombre_argumento>*. En el cuerpo de la función se añaden unas líneas al principio que asignan a estas variables los valores en la cima de la pila en orden. Para esto se ha usado una tabla de símbolos igual a la empleada en *trad4.y*. Esto ha hecho de las acciones semánticas algo más complejo, ya que cada vez que se reconozca un *IDENTIF* es necesario comprobar si nos encontramos en el cuerpo de una función y si dicho identif es un argumento para poder darle el nombre adecuado. También, para poder imprimir las líneas de asignación de los argumentos al inicio de la función ha sido necesario añadir un bucle en las acciones semánticas.
Reconocimiento de IDENTIFs como argumentos de función:

```C/C++
char asign_args[2048];
strcpy(asign_args, "");
int i;
for (i = 0; i < argumentos.i ; ++i) {
      sprintf(asign_args, "%sarg_%s_%s !\n", asign_args,nombre_funcion,
argumentos.lista[i]);
} // asignacion de argumentos
...
sprintf(temp, "%s%s: %s\n%s%s%s;", $5.code, variables_locales, $2.code,
asign_args, asign_local, $7.code);
$$.code = gen_code(temp);
strcpy(nombre_funcion, "");
remove_all(&argumentos);
```

De esta forma un ejemplo de traducción será el siguiente:

funcion.c	**funcion.gforth**
``C/C++ suma (int a, int b) { return a + b; } ```	```Unset variable arg_suma_a variable arg_suma_b : suma arg_suma_b ! arg_suma_a ! arg_suma_a @ arg_suma_b @ + ```

	exit;

Si bien es cierto que existe una forma de incluir parámetros a las funciones en forth incluyéndolos entre llaves en la declaración de las funciones, debido a las limitaciones impuestas en clase y la poca versatilidad de esta solución nos hemos decantado por su traducción como variables.

Funciones con variables locales

Para las variables locales, hemos aprovechado que la traducción a lenguaje intermedio les asigna un nombre especial para poder declararlas *antes* de especificar el nombre de la función en *gforth*. El único detalle relevante es la necesidad de almacenar los valores de inicialización para poder realizar las asignaciones pertinentes en el cuerpo de la función, para lo que usamos una estructura similar a la tabla de símbolos implementada para *trad.y*. Para esto hemos necesitado un nuevo bucle en la acción semántica (encargado de asignar los valores a las variables) y la declaración de variables locales previo a la declaración de la función.

```C/C++
char asign_local[2048];
strcpy(asign_local, "");
for (i = 0; i < var_local.i ; ++i) {
      sprintf(asign_local, "%s%i %s !\n", asign_local, var_local.values[i],
var_local.lista[i]);
} // asignacion de var locales
```

Con todo lo anterior, una función con variables locales se traduce de la siguiente forma:

funcion.c	**funcion.gforth**
C/C++ `division (int a, int b) {` `int resto;` `if (b == 0) {` `puts("division entre 0");` `return 1;` `}` `resto = a % b;` `return a/b;`	Unset `variable arg_division_a` `variable arg_division_b` `variable division_resto` `: division` `arg_division_b !` `arg_division_a !` `0 division_resto !`

`}`	`arg_division_b @ 0 = IF ."` `division entre 0"` `1` ` exit` `  THEN` `arg_division_a @ arg_division_b @` `mod division_resto !` `." resto: "` `division_resto @ .` `arg_division_a @ arg_division_b @` `/` ` exit` `;`

Funciones recursivas

Para poder ejecutar funciones recursivas es necesario utilizar la palabra *RECURSE* de forth. Para esto ha sido necesario añadir una comprobación al realizar una llamada a una función de forma que **si se llama a la misma función que se está definiendo** (cuyo nombre está almacenado en la variable *nombre_funcion*) **será sustituído por la llamada *RECURSE***.

```C/C++
{
       if (strcmp(nombre_funcion, $1.code) == 0) {
           sprintf(temp, "%s %s", $2.code, "RECURSE");
       } else {
           sprintf(temp, "%s %s", $2.code, $1.code);
           $$.code = gen_code(temp);
       }
       $$.code = gen_code(temp);
}
```

Con esto, una función recursiva será traducido de la siguiente manera:

funcion.c	**funcion.gforth**
`C/C++` `fibonacci (int a, int b, int n)` `{`	`Unset` `variable arg_fibonacci_a` `variable arg_fibonacci_b`

```
    if (n < 2) {
        return b ;
    }

    return fibonacci (b, a+b ,
n-1) ;
}
```

```
variable arg_fibonacci_n
: fibonacci
arg_fibonacci_n !
arg_fibonacci_b !
arg_fibonacci_a !
arg_fibonacci_n @ 2 < IF
 arg_fibonacci_b @
 exit
  THEN

arg_fibonacci_b @ arg_fibonacci_a
@ arg_fibonacci_b @ +
arg_fibonacci_n @ 1 -  RECURSE
 exit
;
```

Estructura While

Al reconocer una estructura *while* se traduce a la propia de forth. Dado que tanto un bucle *for* como un bucle *while* se traducen de forma similar al lenguaje intermedio, ambos están contemplados en esta estructura.
Para reconocer esta estructura usamos la siguiente producción:

`bloque → LOOP WHILE expresion DO codigo`

Un bucle *while* se traduce de la siguiente forma:

`LOOP WHILE <expr> DO <codigo>` se traduce en `BEGIN <expr> WHILE <codigo> REPEAT`

Estructura If Else

Para reconocer una estructura *if* debemos tener en cuenta que **puede contener una parte opcional encabezada por un *else***. Por esto, para el reconocimiento de esta estructura de control necesitamos un nuevo no terminal anulable.
Con esto, las producciones necesarias para reconocer esta estructura son las siguientes:

`bloque → IF expresion '(' PROGN codigo ')' else`
`else → '(' PROGN codigo ')' | λ`

Una vez reconocida la estructura se traduce a su equivalente en forth.

`IF <expr> <codigo_true> [<codigo_else>]` se traduce en <expr> `IF <codigo_true> ELSE [<codigo_else>] THEN`
Donde las partes entre corchetes son opcionales.

Operaciones aritméticas y lógicas

Dado que estas operaciones deben estar entre **paréntesis**, las reconocemos a través de una recursividad en dos pasos con las siguientes producciones:

`expresion → '(' operacion ')'`

```
operacion → ‘+’ expresion expresion
          | ‘-’ expresion expresion
          | ‘*’ expresion expresion
          | ‘/’ expresion expresion
          | MOD expresion expresion
          | ‘<’ expresion expresion
          | ‘>’ expresion expresion
          | ‘=’ expresion expresion
          | AND expresion expresion
          | OR expresion expresion
          | GEQ expresion expresion
          | LEQ expresion expresion
          | NOTEQ expresion expresion
          | funcion
```

De esta forma una expresión es combinación de operaciones aritméticas, lógicas y funciones. La traducción se limita a convertir la notación prefija en postfija con los operadores de *gforth*.
En cuanto a los operadores unarios, para que se puedan aplicar a una expresión completa se contemplan en el no terminal de *expresion*.

```
expresion → ‘(‘ NOT expresion ‘)’
          | ‘(‘ ‘+’ expresion ‘)’
          | ‘(‘ ‘-’ expresion ‘)’
```

Estos operadores se traducen a su equivalente en forth. Lo más destacable es la traducción del operador ‘-’ a una expresión de la forma 0 <expresion> -.

Vectores

Se ha creado un nuevo no terminal *vector* para los vectores, así como una nueva producción en el terminal *declaracion* y *expresion*. Los vectores se comportan de manera similar a las variables, teniendo también una estructura con la **lista de vectores locales,** aunque en su caso estos no pueden ser parámetros. Se insertarán en la lista de igual manera a las variables, en la producción de declaración:

```
declaración → SETQ IDENTIF '(' MAKE'-'ARRAY NUMBER ')'
```

En la acción semántica se comprobará si las variables son locales o globales, y en caso de la primera el proceso seguido será idéntico a las variables locales, se insertan en una lista que luego la función correspondiente se encargará de imprimir. En caso de ser globales se traducirán en esta producción a postfija utilizando la estructura *"variable identif expresion cells allot"*, en la que identif y expresión contendrán el valor obtenido de la producción. Esta traducción será la misma para las variables locales.
Para el uso de vectores en cualquier sentencia se añade una única producción con la estructura:

```
vector → '(' AREF IDENTIF expresion ')'
```

En la que *identif* es el nombre del vector y *expresion* el **elemento a recuperar del vector**. La acción semántica traduce esta línea a la estructura *"identif expresion cells +",* en la que se multiplica expresión, que tendrá un valor numérico, por el tamaño de celda, y se suma a la

dirección del vector para conseguir la **dirección del elemento** a utilizar. En el no terminal *expresion* se añade la regla:

```
expresion → vector
```

Que añade el símbolo *'@'* para poder recuperar el **valor del elemento** que se había conseguido en el no terminal *vector*. Así, estos se pueden utilizar de manera equivalente a cualquier otra variable.

ANEXO: Descripción de pruebas

Se ha desarrollado una batería de pruebas de forma que contemplen de manera aislada una casuística lo más amplia posible de los lenguajes de entrada del traductor. Con esto pretendemos validar el funcionamiento del programa a través de la comparación del comportamiento del ejecutable generado al compilar el código C y la ejecución del código generado en *forth*. Además, se han ejecutado y pasado exitosamente todas las pruebas proporcionadas por los profesores.
Todas las pruebas han sido traducidas a *lisp* for **trad**, comprobado el funcionamiento correcto en CLisp, traducida esta salida a *forth* con **back**, y comprobado que el resultado era correcto en Gforth. En todos los casos la traducción y funcionalidad es correcta. En todas las pruebas se añade al final la línea //@ main, que se traducirá a *(main)* en trad y *main* en back. Esta línea se puede sustituir por //@ //@ main, pero para poder realizar ejecuciones en ambos Clisp y Gforth se utiliza la otra opción, aunque ambas sean funcionales.

Var_globales.c

Declaración de variables globales y asignación de valores a estas. Prueba todas las combinaciones posibles para declaraciones de varias variables seguidas, así como la asignación a la variable de números, expresiones, otras variables y la combinación de estas tres. Todas las asignaciones ocurren dentro de una función main.

Lisp	Gforth
Unset `(setq a 0)` `(setq b 0) (setq c 2)` `(setq d 0) (setq e 0) (setq f 0) (setq g 0)` `(defun main ()` `(setf a 1)` `(setf d (+ 2 3))` `(setf e b)` `(setf f (+ (+ a b) c))` `(setf g (+ 2 (* 3 c)))` `)` `(main)`	Unset `variable a` `0 a !` `variable b` `0 b !` `variable c` `2 c !` `variable d` `0 d !` `variable e` `0 e !` `variable f` `0 f !` `variable g` `0 g !` `: main` `1 a !` `2 3 + d !` `b @ e !` `a @ b @ + c @ + f !` `2 3 c @ * + g !` `;`

	`` main ```

Logical_op.c

Declaración de una variable para cada uno de los operadores lógicos reconocidos por la gramática y asignación de una expresión a la variable usando estos operadores. Comprueba que todos los operadores se traducen correctamente además de utilizar combinaciones de estos para confirmar que se mantiene la precedencia correcta.

Lisp	Gforth
Unset ``` (setq igual 0) (setq distinto 0) (setq mayorque 0) (setq menorque 0) (setq mayoroigual 0) (setq menoroigual 0) (setq op_and 0) (setq op_or 0) (setq op_not 0) (setq combo 0) (defun main () (setq main_op1 1) (setq main_op2 0) (setf igual (= main_op1 main_op2)) (setf distinto (/= main_op1 main_op2)) (setf mayorque (> main_op1 main_op2)) (setf mayoroigual (>= main_op1 main_op2)) (setf menorque (< main_op1 main_op2)) (setf menoroigual (<= main_op1 main_op2)) (setf op_and (and main_op1 main_op2)) (setf op_or (or main_op1 main_op2)) (setf op_not (not main_op1)) (setf combo (and (/= main_op1 (> main_op2 main_op1)) main_op2)) (prin1 igual) (prin1 distinto) (prin1 mayorque) (prin1 mayoroigual) (prin1 menorque) (prin1 menoroigual) (prin1 op_and) (prin1 op_or) (prin1 op_not)) (main) ```	Unset ``` variable igual 0 igual ! variable distinto 0 distinto ! variable mayorque 0 mayorque ! variable menorque 0 menorque ! variable mayoroigual 0 mayoroigual ! variable menoroigual 0 menoroigual ! variable op_and 0 op_and ! variable op_or 0 op_or ! variable op_not 0 op_not ! variable combo 0 combo ! variable main_op1 variable main_op2 : main 1 main_op1 ! 0 main_op2 ! main_op1 @ main_op2 @ = igual ! main_op1 @ main_op2 @ = 0= distinto ! main_op1 @ main_op2 @ > mayorque ! main_op1 @ main_op2 @ >= mayoroigual ! main_op1 @ main_op2 @ < menorque ! main_op1 @ main_op2 @ <= menoroigual ! main_op1 @ main_op2 @ and op_and ! main_op1 @ main_op2 @ or op_or ! main_op1 @ 0= op_not ! ```

	main_op1 @ main_op2 @ main_op1 @ > = 0= main_op2 @ and combo ! igual @ . distinto @ . mayorque @ . mayoroigual @ . menorque @ . menoroigual @ . op_and @ . op_or @ . op_not @ . ; main

Arithmetic.c

Uso de todos los operadores aritméticos para operaciones con variables o números, así como la combinación de estos. Comprueba la correcta traducción y que se mantenga la precedencia de operadores, incluyendo cuando se añaden paréntesis.

Lisp	Gforth
Unset (setq suma 0) (setq resta 0) (setq mult 0) (setq div 0) (setq modulo 0) (setq unario 0) (setq combo 0) (defun main () (setq main_op1 6) (setq main_op2 3) (setf suma (+ main_op1 main_op2)) (setf resta (- main_op1 main_op2)) (setf mult (* main_op1 main_op2)) (setf div (/ main_op1 main_op2)) (setf modulo (mod main_op1 main_op2)) (setf unario (- main_op1)) (setf combo (+ (- (+ main_op1 (* main_op2 main_op1)) (+ main_op2 main_op2)) (* main_op1 3))) (prin1 suma) (prin1 resta) (prin1 mult) (prin1 div) (prin1 modulo)	Unset variable suma 0 suma ! variable resta 0 resta ! variable mult 0 mult ! variable div 0 div ! variable modulo 0 modulo ! variable unario 0 unario ! variable combo 0 combo ! variable main_op1 variable main_op2 : main 6 main_op1 ! 3 main_op2 ! main_op1 @ main_op2 @ + suma ! main_op1 @ main_op2 @ - resta !

``) (main) ```	``` main_op1 @ main_op2 @ * mult ! main_op1 @ main_op2 @ / div ! main_op1 @ main_op2 @ mod modulo ! 0 main_op1 @ - unario ! main_op1 @ main_op2 @ main_op1 @ * + main_op2 @ main_op2 @ + - main_op1 @ 3 * + combo ! suma @ . resta @ . mult @ . div @ . modulo @ . ; main ```

Impresion.c

Prueba el funcionamiento de puts y su correcta traducción y la de las cadenas de caracteres que contiene. Se han creado varias combinaciones para probar el funcionamiento de printf, incluyendo la impresión de cadenas de caracteres, variables, números y expresiones.

Lisp	Gforth
Unset ``` (setq a 1) (setq b 2) (defun main () (print "Hola Mundo") (prin1 a) (prin1 b) (prin1 "Hola") (prin1 a) (prin1 "mundo") (prin1 (+ 1 1)) (print "Adios Mundo")) (main) ```	Unset ``` variable a 1 a ! variable b 2 b ! : main ." Hola Mundo" a @ . b @ . ." Hola" a @ . ." mundo" 1 1 + . ." Adios Mundo" ; main ```

If_else.c

Comprueba el funcionamiento correcto de la estructura if else, ya sea incluyendo la segunda parte else o con solo la directiva if. Se utilizan expresiones booleanas para las condiciones y funciones de impresión para la comprobación de una correcta traducción del código interior.

Lisp	Gforth
Unset `(setq menor 3)` `(defun main ()` `(setq main_mayor 5)` `(setq main_abc 1)` `(if (> main_mayor 1)` `(progn (print "TRUE")` `(prin1 main_mayor)` `)` `)` `(if (not (< menor main_mayor))` `(progn (print "TRUE")` `)` `(progn (print "FALSE")` `)` `)` `)` `(main)`	Unset `variable menor` `3 menor !` `variable main_mayor` `variable main_abc` `: main` `5 main_mayor !` `1 main_abc !` `main_mayor @ 1 > IF` `." TRUE"` `main_mayor @ .` `THEN` `menor @ main_mayor @ < 0= IF` `." TRUE"` `ELSE ." FALSE"` `THEN` `;` `main`

For.c

Se han añadido dos bucles for, uno con una variable de control ascendente y otra descendente. Esta se imprime para comprobar el correcto funcionamiento del bucle, que se traducirá a una estructura while con el frontend trad.y.

Lisp	Gforth
Unset `(setq b 1)` `(defun main ()` `(setq main_abc 1)` `(setq main_i 0)` `(setq main_c 0)` `(setf main_i 0)(loop while (< main_i 10) do` `(prin1 main_i)` `(setf main_c (+ b main_abc))`	Unset `variable b` `1 b !` `variable main_abc` `variable main_i` `variable main_c` `: main` `1 main_abc !` `0 main_i !`

`` (setf main_abc (+ main_i b)) (setf main_i(+ 1 main_i))) (setf main_i 10)(loop while (> main_i 0) do (prin1 main_i) (setf main_c (+ b main_abc)) (setf main_abc (+ main_i b)) (setf main_i(- 1 main_i)))) (main) ```	``` 0 main_c ! 0 main_i ! BEGIN main_i @ 10 < WHILE main_i @ . b @ main_abc @ + main_c ! main_i @ b @ + main_abc ! 1 main_i @ + main_i ! REPEAT 10 main_i ! BEGIN main_i @ 0 > WHILE main_i @ . b @ main_abc @ + main_c ! main_i @ b @ + main_abc ! 1 main_i @ - main_i ! REPEAT ; main ```

While.c

El fichero while se encarga de probar las estructuras while provenientes de whiles de c, a diferencia de los for de c que son traducidos a la misma estructura. Se plantean tres distintas pruebas dependiendo de la condición de salida, añadiendo un caso en el que no se entrará en el bucle, uno con una sola iteración, y uno que realizará varias iteraciones hasta salir. Se prueba así la correcta traducción de las distintas casuísticas del bucle while.

Lisp	Gforth
Unset ``` (setq i 0) (defun main () (setq main_control 1) (loop while (< main_control 10) do (setf main_control (* main_control 2))) (loop while (> 3 10) do (print "Esto no se va a ejecutar")) (loop while (not i) do (print "una iteracion") (setf i (= (mod 40 2) 0))) ```	Unset ``` variable i 0 i ! variable main_control : main 1 main_control ! BEGIN main_control @ 10 < WHILE main_control @ 2 * main_control ! REPEAT BEGIN 3 10 > WHILE ." Esto no se va a ejecutar" REPEAT ```

``) (main) ```	``` BEGIN i @ 0= WHILE ." una iteracion" 40 2 mod 0 = i ! REPEAT ; main ```

Funcion_0param.c

Contiene funciones sin argumentos o valores de retorno. Estas funciones serán llamadas en main y únicamente imprimirán una cadena, siendo esta prueba la más básica posible con funciones.

Lisp	Gforth
Unset ``` (defun f1 () (print "HOLA mundo")) (defun f2 () (setq f2_i 0) (prin1 "hola mundo: ") (prin1 f2_i)) (defun main () (f1) (f2)) (main) ```	Unset ``` : f1 ." HOLA mundo" ; variable f2_i : f2 0 f2_i ! ." hola mundo: " f2_i @ . ; : main f1 f2 ; main ```

Funcion_1param.c

Consiste en una función que recibe un solo parámetro y devuelve su valor al final. Este parámetro se utiliza para operaciones dentro de la función, que es llamada desde main con un valor de una variable. El valor de retorno también se utilizará en el main para asignarlo a una variable y luego imprimirlo. Se comprueba el correcto funcionamiento de un parámetro, de los retornos al final de la función, de la asignación de funciones y la llamada a estas.

Lisp	Gforth
Unset variable arg_f1_a : f1 arg_f1_a ! arg_f1_a @ 1 + arg_f1_a ! arg_f1_a @ exit ; variable main_out : main 4 main_out ! main_out @ f1 main_out ! main_out @ f1 main_out @ . 0 exit ; main	Unset (defun f1 (a) (setf a (+ a 1)) (return-from f1 a)) (defun main () (setq main_out 4) (setf main_out (f1 main_out)) (f1 main_out) (prin1 main_out) (return-from main 0)) (main)

Funcion_2params.c

Contiene dos funciones con dos parámetros que serán llamadas desde main. En la función division se añaden valores de retorno en medio de la función, probando así acciones de return fuera del final de la función. Además se utilizan variables locales en combinación con los parámetros, que serán dados valor en la llamada a las funciones mediante números.

Lisp	Gforth
Unset (defun suma (a b) (return-from suma (+ a b))) (defun division (a b) (setq division_resto 0) (if (= b 0) (progn (print "division entre 0") (return-from division 1))) (setf division_resto (mod a b)) (prin1 "resto: ") (prin1 division_resto) (return-from division (/ a b))	Unset variable arg_suma_a variable arg_suma_b : suma arg_suma_b ! arg_suma_a ! arg_suma_a @ arg_suma_b @ + exit ; variable arg_division_a variable arg_division_b variable division_resto : division arg_division_b !

``) (defun main () (setq main_r_suma 0) (setq main_r_div1 0) (setq main_r_div2 0) (setf main_r_suma (suma 1 (- 2))) (setf main_r_div1 (division 1 0)) (setf main_r_div2 (division 4 2)) (prin1 main_r_suma) (prin1 main_r_div1) (prin1 main_r_div2) (return-from main 0)) //@ main ```	``` arg_division_a ! 0 division_resto ! arg_division_b @ 0 = IF ." division entre 0" 1 exit THEN arg_division_a @ arg_division_b @ mod division_resto ! ." resto: " division_resto @ . arg_division_a @ arg_division_b @ / exit ; variable main_r_suma variable main_r_div1 variable main_r_div2 : main 0 main_r_suma ! 0 main_r_div1 ! 0 main_r_div2 ! 1 0 2 - suma main_r_suma ! 1 0 division main_r_div1 ! 4 2 division main_r_div2 ! main_r_suma @ . main_r_div1 @ . main_r_div2 @ . 0 exit ; main ```

Funcion_comoexpr.c

Comprueba que es posible utilizar funciones dentro de expresiones como operadores, en caso de que estas tengan un valor de retorno. Se comprueba que el valor resultante de la operación es el correcto.

Lisp	Gforth
`` Unset (setq a 0) (defun cuadrado (n) (return-from cuadrado (* n n))) (defun main () ```	``` Unset variable a 0 a ! variable arg_cuadrado_n : cuadrado arg_cuadrado_n ! ```

```
(setq main_b 0)
(setf a (cuadrado (+ 2 3) ))
(setf main_b (+ a (cuadrado 2 )))
(prin1 a)
(prin1 main_b)
(return-from main 0)
)
 (main)
```

```
arg_cuadrado_n @ arg_cuadrado_n @ *
 exit
;
variable main_b
: main
0 main_b !
2 3 +  cuadrado a !
a @ 2  cuadrado + main_b !
a @ .
main_b @ .
0
 exit
;
 main
```

Funcion_specialcalls.c

Usando las funciones de funcion_2params.c se prueba que estas pueden ser llamadas con diversas expresiones. Se utilizan operaciones, números, variables, combinaciones de estas así como otra función. En todos los casos se comprueba que la salida es la correcta mediante una impresión.

Lisp	Gforth
Unset `(defun suma (a  b )` `(return-from suma (+ a b))` `)` `(defun division (a  b )` `(setq division_resto 0)` `(if (= b 0)` `(progn (print "division entre 0")` `(return-from division 1)` `)` `)` `(setf division_resto (mod a b))` `(prin1 "resto: ") (prin1 division_resto)` `(return-from division (/ a b))` `)` `(defun main ()` `(setq main_a 4) (setq main_b 1)` `(setq main_result 0)` `(print "DIVISION      -")` `(setf main_result (division (suma main_a main_b ) 5 ))`	Unset `variable arg_suma_a` `variable arg_suma_b` `: suma` `arg_suma_b !` `arg_suma_a !` `arg_suma_a @ arg_suma_b @ +` `exit` `;` `variable arg_division_a` `variable arg_division_b` `variable division_resto` `: division` `arg_division_b !` `arg_division_a !` `0 division_resto !` `arg_division_b @ 0 = IF` `." division entre 0"` `1` `exit` `THEN`

```
(prin1 "a = ") (prin1 main_a) (prin1 " b =
") (prin1 main_b) (prin1 " result = ")
(prin1 main_result)
(setf main_result (division (+ main_a
main_b) (* 5 main_result) ))
(prin1 "a = ") (prin1 main_a) (prin1 " b =
") (prin1 main_b) (prin1 " result = ")
(prin1 main_result)
(return-from main 0)
)
 (main)
```

```
arg_division_a @ arg_division_b @ mod
division_resto !
." resto: "
division_resto @ .
arg_division_a @ arg_division_b @ /
 exit
;
variable main_a
variable main_b
variable main_result
: main
4 main_a !
1 main_b !
0 main_result !
." DIVISION    -"
main_a @ main_b @  suma 5  division
main_result !
." a = "
main_a @ .
."  b = "
main_b @ .
."  result = "
main_result @ .
main_a @ main_b @ + 5 main_result @ *
division main_result !
." a = "
main_a @ .
."  b = "
main_b @ .
."  result = "
main_result @ .
0
 exit
;
 main
```

Vectores.c

Contiene declaración, asignación, y recuperación de vectores y sus partes. Comprueba la correcta traducción de los vectores estén dentro o fuera de expresión y la funcionalidad de esta.

Lisp	Gforth
Unset `(setq vec1 (make-array 5))` `(setq vec2 (make-array 4)) (setq vec3` `(make-array 7))` `(setq var 0)` `(defun main ()` `(setf (aref vec1 2) 3)` `(setf var (+ (aref vec1 2) 3))` `(setf (aref vec2 1) var)` `(setf (aref vec3 var) (+ (aref vec1 2)` `(aref vec2 1)))` `(prin1 (aref vec1 2))` `(prin1 vec1)` `)` `(main)`	Unset `variable vec1 5 cells allot` `variable vec2 4 cells allot` `variable vec3 7 cells allot` `variable var` `0 var !` `: main` `3 vec1 2 cells + !` `vec1 2 cells + @ 3 + var !` `var @ vec2 1 cells + !` `vec1 2 cells + @ vec2 1 cells + @ + vec3` `var @ cells + !` `vec1 2 cells + @ .` `vec1 @ .` `;` `main`

prueba_combinada.c

Combinación de todas las pruebas anteriores para comprobar que los distintos elementos interaccionan correctamente entre ellos y las traducciones no colisionan unas con otras. Incluye además comentarios y código embebido, así como funciones, bucles, variables, estructuras ifs, impresiones, etc.

Lisp	Gforth
Unset `(setq abc 1)` `(setq b 0)` `(setq a 0)` `(defun fib (n )` `(setq fib_abc 1)` `(setq fib_i 0)` `(setq fib_c 0)` `(setq fib_v (make-array 40))`	Unset `variable abc` `1 abc !` `variable b` `0 b !` `variable a` `0 a !` `variable arg_fib_n` `variable fib_abc`

```
(setf b 1)
(if (< n 5)
(progn (setf (aref fib_v 31) (fib 10 ))
)
)
(setf fib_i 0)(loop while (< fib_i n) do
(setf fib_c (+ b fib_abc))
(setf fib_abc b)
(setf b fib_c)
(setf fib_i(+ 1 fib_i)))
(return-from fib fib_c)
)
 (prin1 abc)
(defun main ()
(setq main_control 1)
(if (> 3 1)
(progn (print "TRUE")
)
)
(if (not (< 1 0))
(progn (setf b 1)
)
(progn (print "FALSE")
)
)
(loop while (< main_control 10) do
(setf main_control (* main_control 2))
)
(print "FIN")
(print "---")
(if (not (< 1 0))
(progn (setf b 1)
(setf a (fib 2 ))
)
(progn (print "FALSE")
)
)
(return-from main 0)
)
 (main)
```

```
variable fib_i
variable fib_c
variable fib_v 1 cells allot
: fib
arg_fib_n !
1 fib_abc !
0 fib_i !
0 fib_c !
1 b !
arg_fib_n @ 5 < IF
 10  RECURSE fib_v 31 cells + !
  THEN
0 fib_i !
BEGIN
 fib_i @ arg_fib_n @ < WHILE
 b @ fib_abc @ + fib_c !
b @ fib_abc !
fib_c @ b !
1 fib_i @ + fib_i !
 REPEAT
fib_c @
 exit
;
variable main_control
: main
1 main_control !
3 1 > IF
 ." TRUE"
  THEN
1 0 < 0= IF
 1 b !
 ELSE ." FALSE"
 THEN
BEGIN
 main_control @ 10 < WHILE
 main_control @ 2 * main_control !
 REPEAT
." FIN"
." ---"
1 0 < 0= IF
 1 b !
2  fib a !
 ELSE ." FALSE"
 THEN
0
 exit
;
 main
```