Procesadores del Lenguaje

Curso 2024-2025

Práctica Final – Cuarta Parte

Traductor frontend: subconjunto de lenguaje C a LISP (código intermedio)

Traductor backend: LISP a notación Postfix (código final)

En esta cuarta parte los objetivos son

- terminar el traductor del frontend
- iniciar la parte de backend

Trabajo a realizar (frontend):

1. Leed el enunciado pasado completo antes de comenzar.
2. Revisad las especificaciones de este documento que tendrá algunas anotaciones añadidas resaltadas. Comprobad que vuestro trabajo ya realizado sigue dichas especificaciones.
3. Renombrad vuestro fichero de código a **trad4.y** .
4. Continuad con el trabajo en el punto que corresponda, procurad guardar una versión de vuestro código cada vez que resolváis un apartado.
5. **Descargad el archivo tests.zip que contiene una serie de programas en C para probar el traductor frontend.**
6. Instalad el intérprete de `clisp` de forma local usando

   ```
   sudo apt-get install clisp
   ```

   Probad vuestro programa usando:

   Opción 1:

   ```
   ./trad4 <prueba.c | clisp
   ```

   Opción 2:

   ```
   ./trad4 <prueba.c >prueba.l
   clisp prueba.l
   ```

   o

   ```
   ./trad4 <prueba.c >prueba.l ;  clisp prueba.l
   ```
7. EL INTÉRPRETE WEB NO SE UTILIZARÁ EN LA EVALUACIÓN. Procurad usarlo sólo para pruebas puntuales.

COMPROBAD TAMBIÉN LA PARTE II CORRESPONDIENTE AL BACKEND (más adelante en el documento)

CUESTIONES IMPORTANTES:

Los errores de entrega influirán en la evaluación de la práctica. Esto incluye las prácticas no entregadas.

No se valorarán aportaciones que no hayan sido acordadas previamente con los profesores.

De cara a la evaluación de ésta práctica tened en cuenta que la dificultad será incremental conforme avancen los puntos planteados. Los primeros se pueden resolver en clase por lo que también contarán menos. En las últimas sesiones se valorará vuestra autonomía a la hora de completar los puntos propuestos.

No se recomienda en ningún caso dejar la práctica para última hora. Las sesiones de prácticas están pensadas para dar soporte con los errores que surgen al usar *bison*.

#1

Si se desea generar una salida Lisp formateada, no se debe intentar desde la semántica del parser, puesto que eso la complica y dificulta su comprensión.

Una buena solución consiste en programar una función backend específica que recibe la cadena traducida para imprimir y que la vaya recorriendo y generando el formato deseado a la vez que la imprime.

#2

Revisad en el código **`yylex()`** la utilidad de la directiva de código embebido **`//@`**. Tendrá un papel fundamental en las pruebas de evaluación.

Para lanzarlas es necesario recurrir a una instrucción **`//@ (main)`** que estará incluida al final del programa de prueba.

Ejemplo:

```
int a = 10 ;
int b = 23 ;

f1 (int a, int b) {
  return a+b ;
}
//@ (print (f1 2 3))      ; permite probar f1 desde el código fuente

main () {
  printf ("%d", f1 (a, b)) ;
}
//@ (main)                ; permite lanzar la ejecución del programa
```

#3

Debido a lo anterior se prohíbe generar de forma automática la salida **`(main)`** en la traducción ya que esto provoca dobles ejecuciones y otros problemas.

Tampoco debéis realizar llamadas a **`exit()`** cuando consideréis terminado el programa. Esto sucede de forma automática cuando **`yylex`** lee un *EOF* y devuelve **`return(0)`** a **`yyparse`**.

#4

Debéis formatear el código bison de forma que sea legible. Se trata siempre de una buena praxis, igual que cuando programamos. En AG encontraréis un ejemplo de buenas prácticas programando con bison.

#5

La forma de probar el funcionamiento de vuestro traductor no va a ser usando el intérprete web. En la evaluación se usará el Common Lisp de gnu en un entorno Linux.

Existen otros intérpretes, pero todos ellos pueden alterar las condiciones de ejecución. Por ello no se podrán admitir prácticas que sólo funcionen en otros intérpretes.

Debería estar disponible en guernika. Pero lo podéis instalar en vuestro sistema Linux. En Ubuntu con el comando:

```
sudo apt-get install clisp
```

La forma de probar sería con los siguientes comandos:

Opción 1:

```
./trad <prueba.c | clisp
```

Opción 2: (evita salidas duplicadas de la función eval)

```
./trad <prueba.c >prueba.l
clisp prueba.l
```

Opción 3: (evita salidas duplicadas de la función eval)

```
./trad <prueba.c >prueba.l ;  clisp prueba.l
```

Comprobad que tenéis accesos definidos mediante PATH o usad el camino de forma explícita.

La salida obtenida con todas las opciones debería ser igual a la que obtendremos compilando prueba.c con gcc y ejecutando el resultado. La excepción serán los saltos de línea que introduce la producción **`print`** de Lisp. Y en algún caso adicional cómo los múltiples valores, que en C estándar no existen.

#6

Es importante intentar que el volcado (impresión) de la traducción NO se haga en la producción del axioma. Como se explicó en magistral, esto presenta varios problemas, entre ellos el consumo exponencial de memoria dinámica. Se debe intentar hacer por ejemplo, cada vez que se termine de definir una función, etc.

También es importante que vuestro traductor imprima la salida mucho antes de llegar al axioma, porque de lo contrario el código embebido quedará desfasado.

#7

Evitad imprimir mensajes o textos ajenos a la traducción. Si queréis imprimir algún mensaje de error, hacédlo por el canal de error stderr. Pero tened en cuenta que al mezclarse con la salida estándar (sdout) puede interferir en las correcciones.

#8

Cuestiones que se tendrán en cuenta en la evaluación:

- Errores en las pruebas iniciales y avanzadas (-½ por subapartado de las especificaciones del enunciado).
- Errores gramaticales que permiten procesar estructuras no solicitadas, es decir, diseños gramaticales que permitan reconocer entradas no indicadas en las especificaciones o que no tengan sentido.
- El uso de recursividad por la izquierda salvo en aquellos casos que sea necesario para utilizar precedencia y asociatividad de operador con doble recursividad.
- Fallos no detectados por las pruebas proporcionadas por el alumno.
- Pruebas que produzcan bloqueos del programa trad o del intérprete de Lisp.
- Conflictos shift-reduce o reduce-reduce.
- Código difícil de entender por formateo inadecuado o por añadidos no solicitados (formateo de los paréntesis de Lisp).
- Retoques no autorizados en las funciones proporcionadas (yylex, etc.).

Especificaciones (Frontend)

Las especificaciones previas del fichero **`trad1.y`** proporcionado son:

- Hay una estructura jerárquica en la gramática que empieza con:
  - **`Axioma`**, se encarga de la recursividad (**`r_axioma`**) y deriva:
  - **`Sentencia`**, que a su vez deriva la asignaciones y las impresión de expresiones.
- La sentencia que contiene únicamente una expresión se ha eliminado. Aunque en el lenguaje C se permiten sentencias como **`1+2;`** no les sacaremos partido, y además pueden ser una fuente de conflictos.
- Podéis añadir vuestra solución para asignaciones encadenadas si disponéis de ella.
- El código de **`trad1.y`** utiliza código diferido. Dejaremos el posible uso de AST para más adelante.
- Estudiad las soluciones proporcionadas para:

```
a. termino ::= operando
b. termino ::= + operand
c. termino ::= - operand
d. operando ::= IDENTIF
e. operando ::= ( expresion )
```

Proponemos una secuencia de pasos para desarrollar esta práctica. Se recomienda abordar cada uno de los pasos de forma secuencial.

1. **Variables Globales**. Incluid la definición de variables globales en la gramática.
   a. La definición en C será **`int <id>;`**[2] que debe traducirse a **`(setq <id> 0)`**. Es necesario incluir un segundo parámetro en Lisp para inicializar las variables. En el caso de la inicialización más simple en C, este valor será 0 por omisión. Puede haber 0, 1 o más líneas con definiciones de variables cada una comenzando por **`int`**.
   b. Ampliad la gramática para contemplar la definición de una variable con inicialización incluida: **`int <id> = <cte> ;`** que debe ser traducido a **`(setq <id> <cte>)`**. Utilizamos aquí **`<cte>`** para representar un valor numérico constante. No consideraremos por ahora expresiones en las declaraciones.
   c. Ampliad la gramática para contemplar la definición múltiple de variables con asignaciones opcionales: **`int <id1> = 3, <id2>, ..., <idk> = 1 ;`** que debe ser traducido a una secuencia de definiciones individuales **`(setq <id1> 3) (setq <id2> 0) ... (setq <idk> 1)`** . El orden de impresión de las variables debe corresponder al de la declaración. Prestad atención a que se asignan valores constantes (numéricos), no expresiones evaluables. Esto es así porque para las variables globales en C no es posible evaluar dichas expresiones en tiempo de compilación.

   > Esto corresponde a la definición de variables globales. Recordamos que en C no está permitido inicializar variables con expresiones (con variables y funciones) en la instrucción en que es declarada puesto que el proceso se realiza en tiempo de compilación. La evaluación de expresiones se hace en tiempo de ejecución.
   > Algunos compiladores sí disponen de una fase previa capaz de evaluar expresiones simples del tipo **`int ab32 = 3+7 ;`** y de sustituirlas por **`int ab32=10;`** pero sigue siendo una tarea realizada en tiempo de compilación.

[2] A partir de este punto comenzamos a traducir de C a Lisp

2. **Función MAIN**. En C `main` es el procedimiento/función principal, Debe tener una palabra reservada en la tabla correspondiente, y debe enlazar con el *Token* `Main`. Ampliad la gramática para reconocer la función `main`. Debe permitir la inclusión de sentencias. Prestad atención al siguiente punto. Incluimos un ejemplo de traducción:

C	Lisp

`int a ;`
`main () {`
`  @ (a + 1) ;`
`}`
	`(setq a 0)`
	`(defun main ()`
	`  (print (+ a 1))`
	`)`

`//@ (main)`	`(main) ; Para ejecutar el programa`

A través de la gramática es posible obligar a que exista una (única) función `main`.

Considerad que la estructura de un programa en C debe ser:
`<Decl Variables> <Def Funciones>.` En teoría debería poder intercalarse ambos tipos de definiciones, pero eso puede producir conflictos en el *parser*. Por ello seguiremos una estructura fija. Las sentencias que define la gramática sólo deben aparecer dentro del cuerpo de una función. Revisad la estructura de vuestra gramática. Debe estar diseñada de forma muy cuidadosa y estructurada, intentando que sea lo más jerárquica posible.

Aquí se indican varias cuestiones.

1) Hay que diseñar una gramática lo más jerárquica o estructurada posible. Los nombres de No Terminales deben escogerse con cuidado y deben ser representativos y significativos.

2) Esto evitará la aparición de errores típicos de diseños "enmarañados" o excesivamente complicados.

3) Se sugiere emplear una estructura de programa que empiece con la declaración de variables y luego con la definición de funciones.

4) La recomendación es definir las funciones en orden inverso de jerarquía, empezando con las funciones más sencillas y terminando por la principal, el `main`. Esto permitirá que más adelante el traductor tenga siempre referencia de las funciones que se usan porque ya se han definido previamente. Si se definen primero las funciones principales y luego las de inferior jerarquía, un compilador necesitará hacer averiguaciones sobre el tipo de las funciones que se llaman y que aún no se han definido, o emplear dos pasadas para realizar traducciones parciales y condicionadas.

El compilador de C requiere en estos casos de una declaración previa (prototipo) de las funciones. En Lisp deben definirse las funciones antes de ser usadas. Para evitarnos complicaciones con los prototipos usaremos programas en C con las funciones en orden jerárquico inverso.

3. **Impresión de cadenas**. Para imprimir cadenas literales proponemos: **`puts(<string>);`** que se convierte en **`(print <string>)`**. Ejemplo: **`puts("Hola mundo");`** que se convierte en **`(print "Hola mundo")`**. Comprobad que **`yylex`** detecta las secuencias de entrada entre dobles comillas y las envía como un *token* **`String`**.

4. **Impresión de expresiones y cadenas**. Sustituid la sentencia para imprimir usando **`@`** por una nueva que comience con la palabra reservada **`printf`**. El formato de la impresión es **`printf(<string>, <elem1>, ... , <elemN>);`**

   a. Comenzad por la versión simplificada **`printf(<string>, <elem1>);`** traduciéndola a **`(princ <elem1>)`**. Tomad nota que el contenido de la primera cadena, la cadena de formato, es complejo y requiere de un procesamiento muy elaborado para interpretarla que por ahora excede nuestros objetivos. Por ello debe ser reconocida por la gramática, pero no se traducirá de ninguna forma. Se omitirá en la traducción. El segundo parámetro **`<elem1>`** puede ser o bien una **`<expresion>`** o bien un **`<string>`**, y se traducirá con **`(princ <elem1>)`**

   b. Ampliad la gramática para reconocer:
   `printf(<string>, <elem1>, ... , <elemN>);` El segundo y los siguientes parámetros **`<elemX>`** pueden ser o bien una **`<expresion>`** o bien un **`<string>`**, y se traducirán con **`(princ <elemX>)`** preservando el orden original. Es decir: **`(princ <elem1>)...(princ <elemN>)`**

   > La función **`print`** genera un salto de línea al final. La función **`princ`** omite dicho salto de línea y las dobles comillas de los strings..

5. **Operadores, precedencia y asociatividad**. Los operadores aritméticos, lógicos y de comparación en C se pueden combinar sin que el programador pueda apreciar matices particulares. En realidad se trata de operadores de diferente naturaleza. Los primeros devuelven valores numéricos, mientras que los lógicos y de comparación devuelven valores booleanos. Tendría sentido tratarlos por separado en la gramática, pero esto debe hacerse con cuidado dado que puede producir algunos conflictos o perderse las precedencias definidas por las directivas de bison. Los operadores lógicos y de comparación en Lisp se relacionan con los equivalentes en C en la siguiente tabla. Prestad atención a la asociatividad y precedencia definida para cada operador nuevo que incluyáis.

   https://en.cppreference.com/w/c/language/operator_precedence

C	&&	||	!	!=	==	<	<=	>	>=	%
Lisp	and	or	not	/=	=	<	<=	>	>=	mod

6. **Estructura de control WHILE**: **`while ( <expr> ) { <codigo> }`** . La traduciremos con la estructura **`(loop while <expr> do <codigo>)`** . Prestad atención a que esta sentencia de control en C no requiere de un separador ; al final.

   Y, ¿qué sucede si la entrada contiene algo como **`while(10-a){ a=a+1 ; }`** ? Más adelante explicaremos cómo tratar estos casos.

7. **Estructura de control IF**. Traducir la estructura **`if (<expr>) { <codigo> }`** con **`(if <expr> <codigo>)`** . En Lisp la estructura if sin else acaba de forma temprana con el último paréntesis. Ampliar la estructura de control if con el bloque else: **`if (<expr>) { <codigo1> } else { <codigo2> }`** se traduce como **`(if <expr> <codigo1> <codigo2>)`**. Esta ampliación puede provocar algún tipo de conflicto en el código generado por bison. Estudiad en las transparencias de teoría el origen de dicho conflicto, e intentad resolverlo. Se recomienda el uso de los marcadores de bloque { y } obligatorios. Recordad que la estructura de control **`if`** no termina en ; cuando se usan llaves.

   Planteamos un ejemplo de estructura mínima para el if-then-else:

C	Lisp

?	`(defun prueba-if (flag)`
	`  (if flag`
	`  123`
	`  456)`
	`)`

	`(print (prueba-if T))` ⇒ 123 `(print (prueba-if NIL))` ⇒ 456

   Esto muestra el carácter funcional de Lisp, que no tiene correspondencia en C. En Lisp las expresiones devuelven un valor. En C dichos valores deben asignarse a una variable. No se podrá usar directamente la notación funcional de Lisp al traducir desde C.

   Ejemplo de traducción de C a Lisp:

C	Lisp

`int a = 1 ;`
`int b ;`
 	
`main () {`
`  if (a == 0) {`
`  b = 123 ;`
`  } else {`
`  b = 456 ;`
`  }`
`  printf (¨%d¨, b) ;`
`}`
`//@ (main)`
	`(setq a 1)`
	`(setq b 0)`
	
	`(defun main ()`
	`  (if (= 0 a)`
	`  (setq b 123)`
	`  (setq b 456)`
	`  )`
	
	`  (princ b)`
	`)`
	`(main)`

Para incluir en Lisp varias sentencias en las ramas then o else del if es necesario insertar una función **progn** que reciba dichas sentencias como parámetros. Esto se ilustra en el ejemplo de función **es_par**.

```
int ep ;

es_par (int v) {
  printf ("%d", v) ;
  if (v % 2 == 0) {
      puts (" es par") ;
      ep = 1 ;
  } else {
     puts (" es impar") ;
     ep = 0 ;
  }
  return ep ;
}
```

```
(setq ep 0)

(defun es_par (v)
   (princ v)
   (if (= (mod v 2) 0)
     (progn (print "es par")
            (setq ep 1))
     (progn (print "es impar")
            (setq ep 0)) )


    ep
}
```

En C las condicionales se definen como expresiones genéricas, sin hacer distinción de si se trata de expresiones aritméticas, booleanas o de comparación. Un motivo es que en C no existe el tipo booleano, se simula mediante valores enteros. El booleano False se representa con 0, y el True con cualquier otro valor. Un problema es que en Lisp no se admiten condicionales de tipo aritmético, provocan un error. La solución inicial que vamos a adoptar **inicialmente** es suponer que los programas de prueba usarán un C de tipo canónico, sin extravagancias como **while (10-a)** …

Ejemplos:

C	Lisp

`int a = 1 ;`
`main () {`
`  a = a && 0 ; // a = 0 ;`
`  while (10-a) { // 10-a != 0`
`  printf (¨%d¨, a) ;`
`  a = a + 1 ;`
`}`
	`(setq a 1)`
	`(defun main ()`
	`  (setq a (and a 0))`
	`  (loop while (- 10 a) do ; error`
	`  (princ a)`
	`  (setq a (- a 1))`
	`))`

`int a = 1 ;`
`main () {`
`  a = 0 ;`
`  while (a != 10) {`
`  printf (¨%d¨, a) ;`
`  a = a + 1 ;`
`  }`
`}`
	`(setq a 1)`
	`(defun main ()`
	`  (setq a 0)`
	`  (loop while (/= a 10) do`
	`  (princ a)`
	`  (setq a (+ a 1))`
	`  )`
	`)`

	`(main) ; Para ejecutar el programa`
Imprimirá la secuencia:	`0 1 2 3 4 5 6 7 8 9`

Se entiende que a priori no usaremos programas en C que contengan sentencias como las marcadas en Amarillo, deberán ser como las marcadas en Cyan.

8. **Estructura de Control FOR**. Implementación de la estructura de control `for` de C. Proponemos limitar el uso de este bucle a su versión más canónica. Esto es:

   `for ( <inicializ> ; <expr> ; <inc/dec>) { <codigo> }` donde debemos interpretar **`<inicializ>`** como una **sentencia de asignación de un valor a la variable índice**, **`<expr>`** es una expresión condicional cuyo valor lógico determina si se prosigue el bucle, y **`<inc/dec>`** es una única sentencia para modificar el valor de la variable índice. Utilizad la estructura **`(loop while <expr> do <codigo>)`** para traducir. Recordamos que en el caso de la estructura **`for`** la modificación de la variable índice se realiza en último lugar.

   Es posible usar esta estructura de control de forma menos estricta, incluyendo declaraciones y más de una sentencia en la inicialización, más operaciones simultáneas en la expresión condicional e incluso el cuerpo entero de código dentro del tercer campo de la cabecera. No obstante, desaconsejamos abordar estas opciones.

> **No implementéis operaciones del tipo `i++, ++i, i+=1`.**
> Utilizad incrementos en asignación del tipo: **`i=i+1`** o **`a=a-2`**

Ejemplo de bucle for en C con estilo canónico:

```
for (a = 0  ; a < n ; a = a + 2) {
    . . .
}
```

9. **Variables Locales**. Dentro del cuerpo de las funciones pueden declararse variables, que tanto en C como en Lisp serán variables locales. Se emplea la misma traducción de las globales, pero en este caso deben generarse dentro del cuerpo de la función. Hay que tener cuidado a la hora de incluir la definición de variables locales en la gramática, para evitar conflictos con la definición de las globales.

   Revisad detenidamente el código del siguiente ejemplo para entender lo que se requiere en los siguientes apartados:

   a. Las variables globales se definirán como siempre usando **`setq`**
   b. Las variables locales se definirán usando **`setq`** con un cambio, concatenando su nombre con el nombre de la función. Esto se debe a que las variables definidas con **`setq`** son siempre variables globales en Lisp. El uso de la concatenación sirve para evitar colisiones entre variables locales y globales. Hasta el siguiente punto considerad sólo el caso de main
   c. Cambiad el código de salida de las sentencias de **asignación**: utilizad a partir de ahora **`setf`** en lugar de **`setq`**. Ambas funciones **`setq`** y **`setf`** son muy similares en Lisp, pero las utilizaremos para diferenciar entre declaraciones de variables con inicialización y sentencias de asignación.

C	Lisp

`int a ;`
`main () {`
`  int a = 4 ;`
`  a = a + 1 ;`
`  printf (“%d”, a+1) ;`
`}`
`//@ (main)`
	`(setq a 0)`
	`(defun main ()`
	`  (setq main_a 4)`
	`  (setf main_a (+ main_a 1))`
	`  (princ (+ main_a 1))`
	`)`
	`(main) ; Para ejecutar el programa`

   Utilizad el guión bajo _ para concatenar nombres en lugar del guión - . El guión es demasiado ambiguo con respecto al signo unario o al operador de resta.

   Para decidir qué variables utilizadas dentro de una función deben concatenarse con el nombre de la función, es necesario utilizar una tabla local en la que se insertan todas las variables declaradas localmente. Si una variable no se encuentra en esta tabla, puede deducirse que es una variable global y no debe concatenarse. Como punto de partida estudiad la función de búsqueda en la tabla de palabras reservadas.

10. **Funciones**. En este apartado vamos a dar una serie de indicaciones para evitar problemas. Inicialmente proponemos el uso de las funciones sin tener en cuenta:
    a. Cómo opción inicial se propone la definición de funciones:
        i. **sin el valor de retorno**. En C podemos escribir funciones que no retornan nada, incluso aunque tengan definido un tipo (podemos considerarlo una mala praxis). Incluso aunque tengan un retorno, en la función que las llama se puede ignorar dicho valor. Eso sería el uso como procedimientos.
        ii. **sin tipos explícitos para las funcione**s. Si definimos en C una función sin tipo de retorno explícito se asigna de forma implícita el tipo **`int`**. En nuestro caso, esta opción es útil, puesto que de otra forma la definición de funciones y variables tendría la misma secuencia inicial de símbolos: <tipo> <identificador>. Esta circunstancia puede provocar conflictos, que con pueden solucionar con una reescritura adecuada de la gramática, pero supone una complicación.
        iii. **sin parámetros**. Esto servirá para hacer pruebas iniciales.
    b. Posteriormente podremos ampliar para que la función contenga:
        i. **un único argumento**. Se entiende que debe contemplar cero o un argumento. La gramática debe ampliarse tanto en la definición de las funciones como en las llamadas.
        ii. **una lista de argumentos**, que puede incluir cero, uno o más parámetros. La gramática debe ampliarse tanto en la definición de las funciones como en las llamadas. Hay que prestar atención a que la secuencia traducida de parámetros siga el mismo orden en ambos casos.
    c. **Añadimos el Retorno**. Recordamos la cuestión de los retornos estructurados.
        i. Una función bien estructurada sólo debería tener return como la última sentencia de la función: **`mifuncion () { … return <expresion>; }`** deberá traducirse a Lisp como:
        `(defun mifuncion ()  … <expresion> )` Conviene recordar que return en C va seguido de una expresión que no necesariamente lleva paréntesis. No la consideramos como una función, por lo que no requiere que se le pase el valor de retorno como argumento con paréntesis.
        ii. Sin embargo, en C se pueden situar en cualquier otro punto de la función, aunque sea una mala praxis dentro de la programación bien estructurada. Para ello, tendremos que traducir la sentencia
        `return expresion ;` como
        `(return-from<nombre-funcion> <expresion>)`.
    d. **Llamadas a funciones como procedimientos**. Recordamos que en C las funciones se pueden usar tanto como procedimientos, ignorando el valor que devuelve, o como funciones. En el primer caso deberíamos considerarlo una mala praxis si la función devuelve algún valor. Pero lo podemos contemplar también como un apartado adicional. Es decir, podremos considerar que las funciones de usuario además de intervenir en una expresión, como parámetro a la llamada a otra función, o en una asignación, también podrá ser una sentencia en la que sólo se llame a la función.

Ejemplos de funciones con un parámetro y retorno:

```
#include <stdio.h>

square (int v) {
    return (v*v) ;
}

fact (int n) {
   int f ;
   if (n == 1) {
      f = 1 ;
   } else {
     f = n * fact (n-1) ;
   }
   return f ;
}

is_even (int v) {
   int ep  ;
   printf ("%d", v) ;
   if (v % 2 == 0) {
      puts (" is even") ;
      ep = 1 ;
   } else {
      puts (" is odd") ;
      ep = 0 ;
   }
   return ep ;
}

main () {
   printf ("%d\n", square (7)) ;
   puts (“ “) ;
   printf ("%d\n", fact (7)) ;
   puts (“ “) ;
   printf ("%d\n", is_even (7)) ;
   puts (“ “) ;
   printf ("%d\n", is_even (8)) ;
   puts (“ “) ;
   is_even (8) ;
}
//@ (main)
```

```
(defun square (v)
    (* v v)
)

(defun fact (n)
     (setq fact_f 0)
     (if (= n 1)
        (setf fact_f 1)
        (setf fact_f (* n (fact (- n 1))))
      )
      fact_f
)


(defun is_even (v)
   (setq is_even_ep 0)
   (princ v)
   (if (= (mod v 2) 0)
      (progn
        (print " is even")
        (setf is_even_ep 1))
      (progn
        (print " is odd")
        (setf is_even_ep 0)) )
   is_even_ep
)

(defun main ()
    (princ (square 7))
    (print " ")
    (princ (fact 7))
    (print " ")
    (princ (is_even 7))
    (print " ")
    (princ (is_even 8))
    (print " ")
    (is_even 8)
)
(main)
```

```
⇒ 49
⇒ 5040
⇒ 7 is odd 0
⇒ 8 is even 1
⇒ 8 is even
```

11. **Implementación de Vectores.**

a. **Declaración**. Para declarar un Vector podemos usar una sintaxis de Lisp extendiendo la que usamos para variables: si en C definimos:
`int myvector[32];` lo traduciremos a **`(setq myvector (make-array 32))`** creando un vector llamado **`myvector`** con 32 elementos. **No contemplaremos la opción de inicializar los elementos, cómo sí se puede hacer con variables simples.** Tanto en C como en Lisp el primer elemento se indexará con 0. El tamaño del vector será un valor entero.

b. **Operando**. Para acceder a un elemento de un vector, para operar con él dentro de una expresión utilizaremos la función **`aref`**. Por ejemplo, si queremos imprimir el quinto elemento del vector **`myvector`** previamente definido: **`printf("%s", myvector[4]);`** deberá ser traducido como **`(princ (aref myvector 4))`**. La función **`aref`** usa dos parámetros, siendo el primero el vector, y el segundo el índice para acceder y devolver el valor indexado. El índice podrá ser una expresión.

c. **Receptor**. Para modificar un elemento de un vector, en la parte izquierda de una asignación **`myvector[5]=123;`** debemos usar: **`(setf (aref mivector 5) 123)`**. El índice podrá ser una expresión.

d. Podrán declararse como variables globales o locales. Excluimos la declaración de vectores como parámetros de una función. Es decir, NO contemplamos una declaración en C como: **`int mifunc (int vect [7]) { ... }`**. A la hora de llamar a una función SI podemos pasarle un elemento indexado de un vector, se comportará como una variable individual: **`a = max (v[i], v[i+1])`**.

Mostramos ahora una secuencia de instrucciones Lisp para que se puedan valorar mejor las acciones y sus resultados.

Lisp	Resultado
`(setq b (make-array 5))`	
`(print b)`	`#(nil nil nil nil nil)`

`(setq i 0)`
`(loop while (< i 5) do`
`(setf (aref b i) i)`
`(setf i (+ i 1))`
`)`	

`(print b)`	`#(0 1 2 3 4)`
`(setf (aref b 0) 123)`	
`(print (aref b 0))`	`123`
`(print b)`	`#(123 1 2 3 4)`
`(print (aref b 10))`	`Aref Index out of range`

Traductor backend: notación LISP a Postfix (código final)

Comenzamos con la segunda parte del traductor que abordará el diseño de un pequeño backend para traducir de Lisp a un lenguaje de bajo nivel para una máquina de pila (como alternativa a las cpus más convencionales programadas con Risc V o lenguaje ensamblador x86) .

Trabajo a realizar (backend):

1. Por favor leed el enunciado completo antes de comenzar a trabajar.
2. Cambiad el nombre del archivo de código de sesiones anteriores a **`back4.y`**.
3. Continuad con el trabajo en el punto apropiado. Intentad guardar una versión de su código cada vez que resuelva una especificación.
4. Desarrollad los puntos indicados en las Especificaciones en la medida que tengas tiempo.
5. Descargad el archivo tests.zip que contiene una serie de programas en C para probar vuestros traductores (frontend y backend).
6. Instala el intérprete de **`gforth`** en el portátil siguiendo las instrucciones del apartado *Recursos para realizar la práctica*.
7. Los traductores serán evaluados utilizando el siguiente procedimiento

   `./trad4 <prueba.c | ./back4 | gforth`

   EL INTÉRPRETE WEB NO SE UTILIZARÁ EN LA EVALUACIÓN. Intentad utilizarlo sólo para pruebas específicas.
8. Comprobad que gforth proporcione los mismos resultados que usando:

   `./trad4 <prueba.c >prueba.l; clisp prueba.l`

   O

   `./prueba` después de compilar **`prueba.c`** con **`gcc`**.

Entrega:

Subid el archivo **`back4.y`** con el trabajo realizado hasta ahora junto con el archivo frontend.
Incluid dos líneas de comentarios iniciales en el encabezado del archivo. El primero con vuestros nombres y número de grupo. Y el segundo con los emails (separados por un espacio).
Incluid en el informe **`trad4.pdf`** una breve explicación del trabajo realizado.
Antes y después de realizar la entrega, comprobad que funciona correctamente y que cumple con las indicaciones.

Anexo procedente de la práctica Final - Aproximación 1

Breve Introducción a Forth

Aunque el objetivo de la práctica final es realizar un traductor de C a código intermedio, vamos a comenzar por una fase más elemental: traducir las expresiones que evaluamos con nuestra calculadora en código interpretable en Forth.

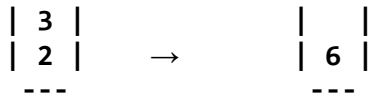
Es decir, es necesario traducir expresiones como

```
2*3<intro>
A=2*3<intro>
B=A*10+23/10<intro>  etc.
```

en un código que sea evaluable por un intérprete de Forth.

Se recomienda recurrir a un intérprete de Forth para ir probando los ejemplos que se plantean aquí. Consultad la última página para instalar **gforth**, o alternativamente al intérprete online: http://nhiro.org/learn_language/FORTH-on-browser.html

La principal particularidad de Forth es que emplea una pila de parámetros sobre la que operan todas las funciones (denominadas *palabras*). Así, la operación de multiplicación * tomará los dos valores de la cima en la pila, para sustituirlos por su producto.



En notación de los así llamados *comentarios de pila*, para la multiplicación tendríamos:

```
(a b – a*b)
```

Para evaluar la expresión **2*3<intro>** en Forth emplearemos la siguiente secuencia:

```
2 3 * . <intro>
```

Los números 2 y 3 se reconocen como tal (*literales*) y son colocados por el intérprete en la pila de parámetros. El operador (*palabra*) * los extrae y los sustituye por el producto de ambos. El operador . es una *palabra* que imprime el valor de la cima de la pila (eliminándolo). Es fácil reconocer que la notación infija a la que estamos habituados (operador binario entre operandos) se ha transformado en notación postfija (polaca inversa: primero los operandos, luego el operador). Quién haya manejado calculadoras de HP estará habituado a esta notación. Adviértase que entre cada uno de los símbolos de entrada debe haber al menos un espacio en blanco. La línea se evalúa al pulsar ***<intro>***, y el intérprete responderá con:

```
6 ok
```

Otros operadores aritméticos son: **+, - y /**. Todos ellos operan de igual forma que la multiplicación. Es importante precisar que los números en la pila son siempre enteros, generalmente con el rango de valores -32768...+32767. La división será por tanto entera y tiene su complemento en el operador `mod` que calcula el resto de una división. Para negar un número existe la *palabra* **negate (a -- -a)**.

Algunas *palabras* habituales de gestión de pila son (entre otras):

- **dup (a -- a a)** duplica el valor de la cima de la pila
- **swap (a b -- b a)** intercambia los dos valores de la cima
- **drop (a b -- a)** elimina el valor de la cima

Cualquiera de las operaciones que se realice sobre una pila con menos operandos de los necesarios provocará un error y, en algunos intérpretes, el vaciado de la pila.

Para definir una variable usaremos la *palabra* **variable**, clasificada en Forth como *palabra constructora*, porque añade una definición al *diccionario* interno. Para crear las variables A y B:

```
variable A  variable B
```

Para realizar la operación **A=6**, usaremos el operador ! .

```
6 A !
```

El intérprete identifica la letra A como una *palabra* definida en el *diccionario* como variable. Así que deja sobre la pila la dirección de memoria asignada a la variable. El operador ! toma de la pila una dirección y un valor, y almacena dicho valor en la dirección. De forma similar podremos resolver **A=3*2** :

```
3 2 * A !
```

Para recuperar el valor de una variable, usaremos el operador **@** :

```
A @ .                          imprimirá:
6 ok
```

La expresión **B=A*10+(23/10)** traducida a Forth debe quedar:.

```
23 10 / A @ 10 * +  B !
B @ .                          imprimirá:
62 ok
```

Aunque no es necesario ahora mismo, podemos dar un pequeño atisbo de la filosofía de este lenguaje. Para construir nuevas *palabras* de tipo función, usaremos otro *constructor*, la *palabra* : .

```
: cuadrado dup * ;
```

Aquí definimos la *palabra* **cuadrado** que contiene la secuencia **dup *** y el terminador de definición **;** . Una aplicación de ésta palabra definida la tenemos en el ejemplo:

```
5 cuadrado .                   que es evaluada con
25 ok
```

En C tendríamos la definición equivalente (no la traducción):

```
int function cuadrado (int valor)
{
      return (valor*valor) ;
}
```

Forth permite el uso de las estructuras de control **`if-then-else`** y **`While`**. **Su uso sólo está permitido dentro de la definición de una palabra (función).**

El **`If-Then-Else`** en versión con la rama **`Else`** es:

```
[condition] IF
    [código a ejecutar si la condición es verdadera]
ELSE
    [opcional: código para ejecutar si la condición es falsa]
THEN
```

- **`[condition]`** es una expresión booleana que debe dejar un valor verdadero (distinto de cero) o falso (cero) en la pila.
- **`IF`** comprueba si el valor superior de la pila es cierto y ejecuta el código entre **`IF`** y **`ELSE`**. Si es falso, la ejecución salta a **`ELSE`** (si está presente) o a **`THEN`**.
- **`ELSE`** es opcional y proporciona una rama alternativa si la condición es falsa.
- **`THEN`** marca el final del bloque **`IF`**.

Ejemplo:

```
: test-condition ( n -- )
    dup 0 >= IF
        ." Numero positivo" cr
    ELSE
        ." Numero no positivo" cr
    THEN ;
```

La versión sin la rama **`ELSE`** es:

```
[condition] IF [código a ejecutar si la condición es verdadera]
THEN
```

La estructura **`WHILE`** se puede implementar con el bucle **`BEGIN ... WHILE ... REPEAT`**

```
BEGIN
    [condition]
WHILE
    [código para repetir]
REPEAT
```

- **`BEGIN`** marca el comienzo del bucle.
- **`[condition]`** es una expresión booleana que debe dejar un valor verdadero (distinto de cero) o falso (cero) en la pila.
- **`WHILE`** comprueba si el valor de la cima de la pila es cierto y ejecuta el código entre **`WHILE`** y **`REPEAT`**. Si es falso, termina después de la etiqueta **`REPEAT`**.
- **`REPEAT`** transfiere la ejecución a la etiqueta **`BEGIN`**.

Ejemplo:

```
: countdown ( n -- )     \ Espera un número en la pila
    BEGIN
        dup 0 >          \ Comprobar si el número es positivo
    WHILE
        dup . cr         \ Imprimir el número
        1 -              \ Decrementa el número
    REPEAT
    drop ;               \ Elimina el cero final de la pila
```

Información adicional:

#9

Usaremos FORTH como una alternativa de CPU / máquina de pila básica con un conjunto de instrucciones limitado. Es por eso que solo se permitirán instrucciones FORTH específicas. En concreto, todas las mencionadas anteriormente:

- Instrucciones de aritmética, lógica y comparación.
- `@` y `!` para manejar variables.
- Operadores de pila **`(drop, dup, swap, over, rot)`**.
- `:` y `;` para la definición de función
- **`IF, THEN, ELSE, DO, WHILE, REPEAT`** para estructuras de control.
- `. ." "cr` para impresión y salida de terminal.
- **`VARIABLE`** para definir variables.

No se permiten otras palabras, ni tampoco otros constructores de Forth.

#10

La evaluación de la práctica dependerá de si la concatenación frontend-backend genera los mismos resultados esperados del programa C compilado original o del resultado obtenido en Clisp a partir de la traducción del frontend.

```
./trad <prueba.c | ./back | gforth
./trad <prueba.c >prueba.l; clisp prueba.l
./prueba
```

después de compilar **`prueba.c`** con **`gcc`**.

Recursos para realizar la práctica:

Utilizaremos gforth para evaluar las prácticas:

Forth

El nivel de conocimiento de Forth necesario para la práctica es elemental, pero dejamos aquí los siguientes enlaces para que sirvan como referencia:

- http://en.wikipedia.org/wiki/Forth_%28programming_language%29 introducción (en inglés)
- http://www.disc.ua.es/~gil/forth.pdf introducción en castellano que más allá de dar una visión del funcionamiento básico, se extiende en los principios de la programación.
- http://www.complang.tuwien.ac.at/forth/gforth/ gforth de GNU.
- http://www.complang.tuwien.ac.at/forth/gforth/Docs-html/ su documentación (en inglés y html)
- http://www.complang.tuwien.ac.at/forth/faq/faq-general.html conjunto de FAQs sobre el lenguaje en sí (en inglés)
- Intérprete gforth online: http://nhiro.org/learn_language/FORTH-on-browser.html

Instrucciones para instalar y comenzar con gforth:

En http://www.complang.tuwien.ac.at/forth/gforth/ están disponibles los fuentes y binarios necesarios para instalar un intérprete de Forth. La instalación en Linux debería ser sencilla siguiendo estos pasos:

- Descargar el fichero **`gforth-0.7.3.tar.gz`** (la versión anterior también debería funcionar, pero da problemas en las aulas informáticas).
- Descomprimir con
  - ~~**`gzip -d gforth-0.7.3.tar.gz`**~~
  - ~~**`tar xvf gforth-0.7.3.tar`**~~
  - **`tar xvf gforth-0.7.3.tar.gz`**
- en el directorio **`gforth-0.7.3`** se ejecuta una configuración con **`./configure`**
- se compila el proyecto con **`make`** y se crean varios intérpretes/compiladores.
- Se puede usar **`gforth`** con el comando **`./gforth`**. Para salir del intérprete, hay que emplear la *palabra* **`bye`**.
- Para trasladar **`gforth`** de directorio hay que copiar también la imagen **`gforth.fi`**.

En Ubuntu podéis usar **`sudo apt-get install gforth`**

En caso de dar error, probad a primero **`sudo apt-get update`**

Si la instalación os da problemas, utilizad guernika para las pruebas y comentad el problema con los profesores.

Especificaciones del BACKEND: Traducción de LISP a FORTH

Se recomienda abordar cada uno de los pasos de forma secuencial.

1. Utilizad bison para el backend. Diseñad la gramática para un kernel LISP básico. Agregad este diseño a la plantilla de bison (tal vez **trad1.y**). Intentad hacer un traductor directo sin código diferido (sugerencia).

2. Las variables globales se declaran con **variable <var>**. Se puede asignar un valor inicial usando **<valor> <var> !**
   No intentéis crear variables o matrices locales por ahora.

3. Las funciones genéricas se definen mediante **: <fnombre> <code> ;** y en el caso del procedimiento principal **: main <code> ;**
   Tened siempre mucho cuidado al imprimir los espacios antes y después de cada operador de Forth.

4. Para imprimir cadenas, Forth utiliza la palabra **."** con otro **"** para finalizar la cadena. Ejemplo: **." Hello World"**
   El espacio después del **."** es importante, pero el **"** final no necesita un espacio previo.

5. La impresión de un valor entero se realiza con la palabra **.**
   Ejemplos:
   ```
   1 .
   A @ .
   ```

6. Asignar un valor entero a una variable requiere el uso del operador **!** (store).
   Ejemplos:
   ```
   1 A !          ( en C: A = 1 ; )
   A @ B !        ( en C: B = A ; )
   ```

7. La estructura while es la siguiente:
   begin [expr] while [code] repeat
   que es equivalente a **while (<expr>) { <code> }**
   dónde **[expr]** y **[code]** son las traducciones a postfija para **<expr>** y **<code>**

8. La estructura if es:
   [expr] if [code1] else [code2] then
   que es equivalente a **if <expr> { <code1> } else { <code2> }**

   Ojo a la sintaxis postfija que se aplica (**if-else-then**). En Forth la palabra **then** marca el fin del bloque if.

   Para el caso de un if sin rama else: **[expr] if [code1] then**

9. La correspondencia entre operadores aritméticos, lógicos y de comparación es la siguiente:

C	&&	||	!	!=	==	<	<=	>	>=	%
Lisp	and	or	not	/=	=	<	<=	>	>=	mod
Forth	and	or	0=	= 0=	=	<	<=	>	>=	mod

   En C los valores de Falso y Verdadero se suelen representar con 0 y 1. En Forth con 0 y -1.