Procesadores del Lenguaje

Curso 2020-2021

Práctica Final – v3

Traductor de Expresiones a un Lenguaje en Notación Prefija

En sucesivas sesiones se irán detallando las especificaciones del lenguaje que se debe traducir, con algunas explicaciones adicionales, e información técnica. Más adelante se aportarán pruebas de evaluación. La ampliación se hará de forma gradual para intentar evitar problemas técnicos como conflictos del parser.

Trabajo a realizar:

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 en sucesivas sesiones a trad2.y, trad3.y, ...
4. Continuad con el trabajo en el punto que corresponda, desarrollando los puntos indicados en las especificaciones hasta donde os de tiempo. En futuras sesiones continuaremos con ellas.
5. Es posible intentar abordar puntos que no se han especificado, pero con la reserva de que más adelante se indiquen restricciones no contempladas.
6. Podéis evaluar los resultados de traducción usando el intérprete online https://rextester.com/l/common_lisp_online_compiler. Para ello podéis:
    a. editar un fichero (por ejemplo, **`prueba.c`**) con una serie de expresiones
    b. ejecutar **`cat prueba.c | ./trad >prueba.l`**
    c. copiar la salida y pegarla en la ventana del intérprete
    d. con F8 se compila y ejecuta.

Entrega de Sesión:

Seguid las instrucciones del entregador.

Subid el fichero **`trad2.y`** (**`trad3.y`** …) con el trabajo realizado hasta el momento.

Incluid en la cabecera del fichero dos líneas de comentario iniciales. La primera con vuestros nombres y número de grupo. Y la segunda con los correos electrónicos (separados con un espacio).

Entregad también un documento llamado **`trad2.pdf, trad3.pdf,...`** con una breve explicación del trabajo realizado.

Después de hacer la entrega, descargad vuestro fichero y comprobad que funciona correctamente y que cumple con las indicaciones.

Especificaciones

Proponemos una secuencia de pasos para desarrollar esta práctica. Se recomienda abordar cada uno de los pasos de forma secuencial. Pero en caso de complicarse alguno, se puede pasar al siguiente.

Con un recuadro encontraréis anotaciones para comentar preguntas frecuentes.

1. Retomando la sentencia simplificada para imprimir, el scanner actual procesa el símbolo **#** de forma especial, así que puede que no funcione. Cambiad el símbolo de impresión **#** por el símbolo **$**. Incluid también los paréntesis para englobar la función **$ (<exp>) ;** debe traducirse a **(print <exp>)** .

2. Adaptad la gramática para que traduzca la impresión con múltiples parámetros. La entrada **$ (<exp1>, <exp2>, <exp3>) ;** debe traducirse a **(print <exp1>) (print <exp2>) (print <exp3>)** , en el mismo orden original. Las dos indicaciones previas se modificarán más adelante para emplear ya la palabra printf con los parámetros que corresponden.

3. Incluid la definición de variables en la gramática. La definición en C será **int <id> ;** 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. A partir de este punto comienza la traducción de C a Lisp.

4. Ampliad la gramática para contemplar la definición de una variable con asignación 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.

Esto corresponde a la definición de variables globales. Recordamos que en C no está permitido asignar expresiones (con variables y funciones) a una variable global en la instrucción en que es declarada puesto que el proceso tiene que hacerse en tiempo de compilación. La evaluación de expresiones se hace en tiempo de ejecución.

5. 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) ; Para ejecutar el programa`

A través de la gramática es posible obligar a que exista una función **main**.

6. 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. Por ello seguiremos una estructura fija. Las sentencias que genera 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. 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. Se sugiere no permitir la mezcla de declaraciones de variables y de definición de funciones para simplificar la gramática y evitar conflictos. No se prohíbe esta opción. Simplemente, no se recomienda recurrir a ello en el inicio de la práctica.

7. 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:

C	*Lisp*

`int a ;`
`main () {`
`  int a = 4 ;`
`  $ (a + 1) ;`
`}`
	`(setq a 0)`
	`(defun main ()`
	`  (setq a 4)`
	`  (print (+ a 1))`
	`)`

	`(main) ; Para ejecutar el programa`

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. Esta traducción la ampliaremos en puntos posteriores.

8. 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)` . Intentad que el orden de impresión de las variables corresponda al de la declaración. Tened en cuenta que las variables no inicializadas en C se han de inicializar a 0 en Lisp.

   Las variables locales se pueden declarar en C alternando con sentencias. También es posible declararlas con una inicialización en la que se asigna el valor de una expresión, al contrario que en el caso de las variables globales. Para comenzar proponemos declararlas únicamente al comienzo del bloque de código y restringir la asignación en la declaración a valores numéricos. Esto no es obligatorio, es una recomendación para simplificar la gramática y evitar problemas. Más adelante veremos la posibilidad de intercalar sentencias y declaración de variables.

9. Eliminad la producción que deriva una **`Sentencia → Expresion ;`** En su momento tenían su utilidad porque las expresiones se evaluaban de forma interactiva como en una calculadora. En C están permitidas las sentencias que constan sólo de una expresión, pero no les sacaremos partido ahora mismo.

10. Para imprimir cadenas literales proponemos: **`puts("Hola mundo") ;`** que se convierte en **`(print "Hola mundo")`**. Estudiad que **`yylex`** detecta las secuencias de entrada entre dobles comillas las envía como un *token* **`String`**.

11. Sustituid el símbolo **`$`** empleado para imprimir por la palabra reservada **`printf`**. El formato de la impresión se aumenta para contemplar la cadena de formato **`printf(<string>, <expr1>, … , <exprN>) ;`** traduciéndola a **`(print <expr1>) (print <expr2>), …, (print <exprN>)`**. Tomad nota que el contenido de la cadena de formato es complejo y requiere de un procesamiento muy elaborado para interpretar los formatos. Por ello debe ser reconocido por la gramática, pero no se traducirá de ninguna forma. Se omitirá en la traducción. Para imprimir cadenas literales usaremos **`puts`**.

12. 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. Los operadores lógicos y de comparación en Lisp se relacionan con los equivalentes en C en la siguiente tabla:

C	`&&`	`||`	`!=`	`==`	`<`	`<=`	`>`	`>=`
Lisp	`And`	`or`	`/=`	`=`	`<`	`<=`	`>`	`>=`

   Prestad atención a la asociatividad y precedencia definida para cada operador nuevo que incluyáis.
   http://web.cse.ohio-state.edu/~babic.1/COperatorPrecedenceTable.pdf

13. Estructura de control **`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.

14. Traducir la estructura de control **`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.

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

	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

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

	`(main)`

Para incluir en Lisp varias sentencias en la rama then o else del if es necesario insertar una función superior **`progn`** que reciba dichas sentencias como parámetros. Esto se ilustra en el ejemplo de funciones más adelante (en la (función **`es_par`**).

15. 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`**; etc.

16. Implementación de Vectores.
    a. Para declarar un Vector podemos usar una sintaxis de Lisp extendiendo la que usamos para variables: si en C definimos **`int mivector [32] ;`** lo traduciremos a **`(setq mivector (make-array 32))`** creando un vector llamado **`mivector`** 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.

    b. 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 **`mivector`** previamente definido: **`printf ("%s", mivector[4]) ;`** deberá ser traducido como **`(print (aref mivector 4))`**. La función **`aref`** usa dos parámetros, siendo el primero el vector, y el segundo el índice para acceder y devolver el valor indexado.

    c. Para modificar un elemento de un vector, en la parte izquierda de una asignación **`mivector [5] = 123 ;`** debemos cambiar de función de asignación: **`(setf (aref mivector 5) 123)`**. En algunas variantes de Lisp se puede usar la función **`setq`**, pero lo habitual es usar **`setf`**. Esto obliga a realizar en este caso un tratamiento diferenciado en la semántica a los vectores de las variables.

Mostramos ahora una secuencia de instrucciones Lisp para que se pueda 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)`
`  (setq i (+ i 1))`
`)`	

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

17. 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 que carezcan de:
        i. 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. tipos explícitos para las funciones. En C se asigna de forma implícita el tipo **`int`** a las funciones que se definen de tal forma. 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 se pueden solucionar con una reescritura adecuada de la gramática.
        iii. 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 lados.
    c. Añadimos el Retorno. Recordamos la cuestión de los retornos estructurados.
        i. Una función bien estructurada sólo debería tener la sentencia return como la última 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. Recordamos que en C las funciones se pueden usar tanto como procedimientos, ignorando el valor que devuelve, o como funciones. El primer caso deberíamos considerarlo 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:

```c
#include <stdio.h>

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

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

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

main () {
   printf ("%d\n", cuadrado (7)) ;
   printf ("%d\n", factorial (7)) ;
   printf ("%d\n", es_par (7)) ;
   es_par (8) ;
}
```

```lisp
(defun cuadrado (v)
    (* v v)
    )

(defun factorial (n)
     (setq f 0)
     (if (= n 1)
        (setq f 1)
        (setq f (* n (factorial (- n 1))))
        )
    )


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


(defun main ()
    (print (cuadrado 7))
    (print (factorial 7))
    (print (es_par 7))
    (es_par 8)
    )
```

```
(main)
→ 49
→ 5040
→ 7 es impar 0
→ 8 es par 1
```

TAREAS OPCIONALES

Se especificarán más adelante en función de la demanda.

18. Ámbitos locales.
    a. Un único ámbito por función
    b. Ámbitos anidados
    c. Ámbito para la declaración en cabecera de la estructura de control for

19. Asignaciones múltiples (excluimos vectores)
    a. a,b,c=x,y,z;
    b. a, b = b, a ;
    c. Retorno de valores múltiples de funciones

Más adelante se darán únicamente algunas indicaciones técnicas para las especificaciones que no queden claras. Y se propondrá algún punto adicional para resolver como opcional.

Tened en cuenta que pueden surgir algunos operadores en las pruebas, que deberán ser añadidos a la gramática.

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

De cara a la evaluación de esta 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 números errores que surgirán al usar *bison*.

FECHA DE ENTREGA FINAL: 13-05-2021