Procesadores del Lenguaje

Curso 2020-2021

Práctica – Tercera Aproximación.

Traductor de Expresiones a un Lenguaje en Notación Prefija

Esta tercera parte de la práctica consiste en transformar el traductor de las sesiones previas. El primer traductor seguía un esquema conocido como traducción directa, es decir, la conversión e impresión de la entrada analizada se hace sobre la marcha al mismo tiempo que se explora el árbol de derivación. En la segunda aproximación se transformaba en un esquema de traducción más elaborado imprescindible para traducir a expresiones en notación prefija.

En la nueva propuesta ya nos centramos en la traducción a un lenguaje de programación que usa notación prefija. En este caso nos interesa recuperar el comportamiento de nuestras primeras calculadoras. En aquellas, cada expresión introducida se evaluaba y se imprimía su resultado. Ahora también nos interesa ser capaces de evaluar las expresiones, aunque lo haremos a través de un lenguaje objeto. Con ello podremos usar un intérprete que procese la salida de nuestro traductor, con lo que tendríamos una cadena de evaluación automática.

Trabajo a realizar:

Para esta aproximación se pide la traducción de expresiones aritméticas y de asignaciones a variables de una notación infija como la que usamos habitualmente en el Lenguaje C a notación prefija que serán evaluadas por un intérprete de Lisp.

1. Leed el enunciado completo antes de comenzar.
2. Probad los ejemplos incluidos en la Introducción a Lisp con el intérprete disponible en: https://rextester.com/l/common_lisp_online_compiler.
3. Utilizad el vuestro fichero de la segunda aproximación y renombradlo a trad1.y
4. Adaptad la gramática y la semántica para que traduzca las expresiones introducidas a notación prefija de forma que pueda ser evaluado por el intérprete online. Seguid la secuencia de desarrollo propuesta en las especificaciones (al final del enunciado).
5. Evaluad los resultados usando el intérprete online. Para ello podéis:
   a. editar un fichero (por ejemplo, **`prueba.txt`**) con una serie de expresiones
   b. ejecutar **`cat prueba.txt | ./trad1`**
   c. copiar la salida y pegarla en la ventana del intérprete
   d. con F8 se compila y ejecuta.

Breve Introducción al Lenguaje Lisp

Lisp es uno de los primeros Lenguajes de Programación desarrollados, un poco posterior a Fortran, que además sigue perviviendo en nuestros días. Estuvo además muy vinculado al desarrollo de la Inteligencia Artificial a partir de los años 60. Sus principales características son su enorme simplicidad de base y que su principal tipo de datos son las listas. Dado que un programa en Lisp se representa como una lista de listas de elementos, resulta factible procesar programas mediante otros programas. Es además el primer representante del paradigma de programación funcional y recursiva. Y también uno de los primeros lenguajes interpretados. Aunque ha caído en desuso, hay otras variantes como Scheme y Clojure que tienen bastante uso hoy en día.

Para esta práctica vamos a proponer una serie de variaciones iniciales. Disponéis de un intérprete online en `https://rextester.com/l/common_lisp_online_compiler` que os permitirá ir probando los ejemplos incluidos aquí. Con F8 se compila y ejecuta lo introducido en la ventana.

El tipo de datos genérico será una *Expresion*. Los casos elementales son:

1. *Número* (un literal de valor de tipo entero)
2. *Variable* (en principio para almacenar valores enteros)

Pero también tendremos elementos compuestos:

3. *Expresión* (Compuesta). Los elementos de una expresión serán *Operadores/Funciones*, y otras *Expresiones* (*Números*, *Variables* y *Expresiones* Compuestas). La notación para representar una *Expresión* Compuesta se basa en el uso de paréntesis:

`(* 2 3)`	para representar la multiplicación `2*3`
`(+ 1 a)`	para sumar el valor de una variable con un número
`(+ 1 (* 2 3))`	para representar una expresión más compleja `1+2*3`

Hay otros elementos que intervienen en el Lenguaje. Destacamos:

4. *Operador* o *Función*, en una expresión serán el primer elemento. Los demás elementos de la Expresión serán los parámetros de dicho Operador o Función.

Para asignar un valor a una variable usaremos la Función **`setq`**:

`(setq a 1)`	`a = 1`
`(setq a (+ 1 (* 2 3)))`	`a = 1+2*3`

No es necesario declarar las variables previamente. Usaremos nombres de variables de una única letra para empezar.

Para imprimir un valor usaremos la Función **`print`**

`(print a)`	imprimirá el valor de a
`(print (+ 1 (* 2 3)))`	imprimirá 7

Admite un único parámetro.

Para negar un valor podemos aplicar

`(- 0 a)`	que niega una variable
`(- 0 3)`	devuelve -3
`(- 0 (+ 1 (* 2 3)))`	devuelve -7

O más breve con un Operador menos unario.

`(- a)`	que niega una variable
`(- 3)`	devuelve -3
`(- (+ 1 (* 2 3)))`	devuelve -7

Para definir funciones usaremos la Palabra Reservada **`defun`**. Incluimos algunos ejemplos:

```
(defun prueba ()
    (print "hola"))
```

Define una función llamada **`prueba`** sin parámetros y cuyo cuerpo contiene una llamada a **`(print "hola")`**. Si a continuación evaluamos la Expresión:

`(prueba)` → Imprimirá **`"hola"`**

Podemos definir una función **`prueba2`** con un parámetro:

```
(defun prueba2 (p1)
    (print p1))
```

Al evaluar:

`(prueba2 "adios")` → Imprimirá **`"adios"`**

Para definir una función que calcule el cuadrado de un valor:

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

(print (cuadrado 4))
```

→ Imprimirá **`16`**

Lisp permite definir funciones recursivas. El factorial en versión recursiva sería:

```
(defun factorial (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))
    )

(print (factorial 5))
```

→ Imprimirá **`120`**

Esto nos servirá como introducción inicial para abordar la práctica.

Especificaciones

Proponemos una secuencia de pasos para abordar esta práctica. Incluimos muchas de la la aproximación previa cómo recordatorio:

1. Incluid las acciones semánticas adecuadas en las producciones para que la salida pase de infija a prefija. En este caso, no se debe imprimir nada en la salida estándar en los nodos inferiores. La salida debe ser añadida en el campo **`$$.cadena`** para transmitirla hacia arriba. La impresión de la traducción debe hacerse en el axioma. Dado que **`%type`** asocia un campo determinado de la **`union`** a los *No Terminales*, se podrá operar con **`$$`** en vez de con **`$$.cadena.`** La entrada **`1+2*3`** debería traducirse como **`(+ 1 (* 2 3))`**. Prestad atención a la precedencia y asociatividad definidas en la cabecera del fichero calc8.y .
2. En esta ocasión usaremos paréntesis para acotar las expresiones de salida.
3. Abordad la traducción de las *Variables*, tanto en la parte de componentes de una expresión, como receptoras de una asignación.
4. Prestad atención al diseño jerárquico de la gramática. Debería haber una *Sentencia* que deriva en *Expresion* por un lado y en *Asignacion* por otro. La traducción debería generarse volcarse a la salida a nivel de *Sentencia*. Para separar sentencias usaremos saltos de línea.
5. Incluid una nueva sentencia para imprimir valores. En Lenguaje C debería ser **`printf("%d",<expresion>)`** pero nuestro Analizador Léxicografico no está adaptado para reconocer *Palabras Reservadas* (como **`printf`**), ni tratar la cadena de formato. Por ello proponemos una simplificación. La entrada **`# <expresion>`** debe traducirse como **`(print <expresion>)`**.

Entrega:
Subid el fichero **`trad1.y`**.

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 **`trad1.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.