Abstract Syntax

Foundations for Programming Languages
MASTER IN SOFTWARE AND SYSTEMS
Universidad Politécnica de Madrid/IMDEA Software

October 4, 2017

To be turned in by October 19, 2017. Send them to me by mail to `jmarino@fi.upm.es`.

Gramáticas y sintaxis concreta

Exercise 1. La siguiente gramática genera el lenguaje de las cadenas de bits con un número par de 1's.

$$\begin{array}{lcl} E & ::= & \varepsilon \mid \mathtt{0}E \mid \mathtt{1}O \\ O & ::= & \mathtt{0}O \mid \mathtt{1}E \end{array}$$

Muestra una secuencia de derivación y el árbol de derivación para la cadena `011100100`.

Exercise 2. Una gramática alternativa para el mismo lenguaje sería:

$$\begin{array}{lcl} E & ::= & \varepsilon \mid E\mathtt{0} \mid O\mathtt{1} \\ O & ::= & O\mathtt{0} \mid E\mathtt{1} \end{array}$$

Comenta diferentes opciones para demostrar (en un sentido matemático) que ambas gramáticas generan el mismo lenguaje.

Exercise 3. Da al menos dos expresiones regulares que representen el lenguaje de los ejercicios anteriores. Comenta diferentes opciones para demostrar matemáticamente que dichas expresiones representan el mismo lenguaje que las gramáticas.

Exercise 4. Define una gramática para representar secuencias de paréntesis (abiertos y cerrados) correctamente emparejadas. Proporciona un árbol de derivación para la cadena `(())()`.

Exercise 5. Alguien ha definido la siguiente gramática para definir expresiones aritméticas en un lenguaje de programación:

$$A ::= 0 \mid 1 \mid A + A \mid A - A \mid A * A$$

¿Crees que es una buena idea? ¿Por qué? (Usa algún ejemplo con árboles de derivación.)

Exercise 6. Para resolver los inconvenientes de la gramática anterior alguien decide recurrir a la *notación polaca inversa*, por medio de la siguiente gramática:

$$R ::= 0 \mid 1 \mid RR + \mid RR - \mid RR *$$

¿Se resuelven así los problemas? Justifica tu respuesta.

Exercise 7. Otra opción es usar paréntesis. Para no usarlos en exceso se ha definido la siguiente gramática:

$$\begin{array}{lll} E & ::= & T \mid E+T \mid E-T \\ T & ::= & F \mid T*F \\ F & ::= & 0 \mid 1 \mid (E) \end{array}$$

Razona hasta qué punto esta gramática sería "equivalente" a la del ejercicio anterior.

Sintaxis abstracta

Exercise 8. Define una sintaxis abstracta para expresiones aritméticas. Explica por qué esta sintaxis "comprende" a las de los ejercicios 6 y 7.

Exercise 9. El lenguaje de los "WHILE-programs" es un lenguaje de programación muy simplificado que se usa a menudo en teoría de lenguajes de programación. Las únicas construcciones permitidas son:

- el programa vacío
- asignar a una variable el resultado de evaluar una expresión aritmética (que puede involucrar variables)
- una expresión condicional "if-then-else" que toma una condición booleana y dos programas
- la composición secuencial de dos programas
- un bucle "while" que toma una condición booleana y un programa (el cuerpo del bucle).

Define una sintaxis abstracta para WHILE-programs.

Exercise 10. Un operador del cubo de Rubik se define como una secuencia de giros elementales de 90° (*quarter twists*). Los giros elementales giran cada cara en sentido horario: derecha (R), superior (U), inferior (D), izquierda (L), frontal (F) o trasera (B). Define una sintaxis abstracta para operadores del cubo de Rubik con quarter twists.

Exercise 11. Con la notación anterior un giro de 180° de la cara derecha es RR y un giro de 270° es RRR. Habitualmente estos giros se abrevian R^2 y R' o R^{-1}. Extiende la gramática del ejercicio anterior para contemplar estas abreviaturas.

Exercise 12. Otra extensión es permitir potencias genéricas de operadores. Así, si se trata de repetir tres veces el operador RUF escribiremos $(RUF)^3$. Extiende la gramática para cotemplar potencias.

Exercise 13. Una manera más sofisticada de extender el lenguaje del ejercicio 10 es permitir giros completos del cubo alrededor de los ejes X, Y o Z. Añade esta posibilidad.

Sintaxis abstracta *constructiva* en Haskell

Exercise 14. Define un tipo de datos Haskell para representar cadenas de bits con un némoro par de 1's.

Exercise 15. Define un tipo de datos Haskell que represente la sintaxis abstracta de expresiones aritméticas del ejercicio 8. Define una función Haskell que evalúe dichas expresiones aritméticas. Define *pretty printers* a las sintaxis concretas de los ejercicios 6 y 7.

Exercise 16. Define un tipo de datos Haskell para la sintaxis abstracta de WHILE-programs.

Exercise 17. Define un tipo de datos Haskell para operadores Rubik con giros de 90°. Define otro tipo de datos para operadores con giros de 90° y giros del cubo en los ejes X, Y y Z. Define una función Haskell que convierte un operador en esta última representación en el operador equivalente en la primera representación.