Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 8

Gramática para expresiones aritméticas

Factor de ponderación: 10

1. Objetivos

El objetivo de la práctica es consolidar los conocimientos adquiridos sobre Gramáticas Independientes del Contexto (GIC o *Context-Free Grammar*, CFG) [1] al mismo tiempo que se continúan desarrollando capacidades para diseñar y desarrollar programas orientados a objetos en C++. El programa leerá desde un fichero la especificación de una gramática y a partir de las reglas de producción y de una secuencia de derivaciones, se deberá obtener la cadena de símbolos gramaticales (terminales y/o no terminales) obtenida.

Al igual que en las prácticas anteriores, también se propone que el alumnado utilice este ejercicio para poner en práctica aspectos generales relacionados con el desarrollo de programas en C++. Algunos de estos principios que enfatizaremos en esta práctica serán los siguientes:

- **Programación orientada a objetos**: es fundamental identificar y definir clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea.
- **Diseño modular**: el programa debiera escribirse de modo que las diferentes funcionalidades que se precisen sean encapsuladas en métodos concretos cuya extensión textual se mantenga acotada.
- **Pautas de estilo**: es imprescindible ceñirse al formato para la escritura de programas en C++ que se propone en esta asignatura [11]. Algunos de los principales criterios de estilo ya se han descrito reiteradamente en las prácticas anteriores de esta asignatura.

- **Documentación**: se requiere que los comentarios del código fuente sigan el formato especificado por Doxygen [6]. Además, se sugiere seguir esta referencia [7] para mejorar la calidad de la documentación de su código fuente.
- Compilación de programas utilizando `make` [9, 10].

2. Gramáticas independientes del contexto

Desde la antigüedad, los lingüistas han descrito las gramáticas de los idiomas en términos de su estructura de bloques, y han descrito cómo las oraciones se construyen recursivamente a partir de frases más pequeñas y, finalmente, de palabras o elementos de palabras individuales. Una propiedad esencial de estas estructuras de bloques es que las unidades lógicas nunca se solapan. Por ejemplo, la frase en inglés:

John, whose blue car was in the garage, walked to the grocery store

usando corchetes como meta-símbolos puede agruparse como:

[John [, [whose [blue car]] [was [in [the garage]]],]] [walked [to [the [grocery store]]]]

Una gramática independiente del contexto (*Context Free Grammar, CFG*) [2] proporciona un mecanismo simple y matemáticamente preciso para describir los métodos por los cuales las cadenas de algún lenguaje formal se construyen a partir de bloques más pequeños, capturando la "estructura de bloques" de las frases de manera natural. Su simplicidad hace que este formalismo sea adecuado para un estudio matemático riguroso. En cualquier caso, las características de la sintaxis de los lenguajes naturales no se pueden modelar mediante gramáticas independientes del contexto.

Este formalismo y también su clasificación como un tipo especial de gramática formal fue desarrollado a mediados de los años 50 del siglo pasado por Noam Chomsky [3]. Basándose en esta notación de gramáticas basada en reglas, Chomsky agrupó los lenguajes formales en una serie de cuatro subconjuntos anidados conocidos como la *Clasificación de Chomsky* [4]. Esta clasificación fue y sigue siendo fundamental para la teoría de lenguajes formales, especialmente para la teoría de los lenguajes de programación y el desarrollo de compiladores.

La estructura de bloques fue introducida en los lenguajes de programación por el proyecto Algol (1957-1960), el cual, como consecuencia, también incluía una gramática independiente del contexto para describir la sintaxis del lenguaje. Esto se convirtió en una característica estándar de los lenguajes de programación, aunque no solo de éstos. En [5], a modo de ejemplo, se utiliza una gramática independiente del contexto para especificar la sintaxis de JavaScript.

Las gramáticas independientes del contexto son lo suficientemente simples como para permitir la construcción de algoritmos de análisis eficientes que, para una cadena dada, determinan si y cómo se puede generar esa cadena a partir de la gramática. El *algoritmo de Cocke-Younger-Kasami* es un ejemplo de ello, mientras que los ampliamente usados analizadores LR y LL son algoritmos más simples que tratan sólo con subconjuntos más restrictivos de gramáticas independientes del contexto.

Formalmente una *Gramática Independiente del Contexto* G (por simplicidad nos referiremos a "una gramática") viene definida por una tupla $G \equiv (\Sigma,\ V,\ S,\ P)$ cada uno de cuyos componentes se explican a continuación:

- Σ: Conjunto de símbolos terminales (o alfabeto de la gramática), $\Sigma \cap V = \emptyset$
- V: Conjunto de símbolos no terminales ($V \neq \emptyset$)
- S: Símbolo de arranque (***S**tart*) o axioma de la gramática ($S \in V$)
- P: Conjunto de reglas de producción:

  $P = \{A \to \alpha \mid A \in V,\ \alpha \in (V \cup \Sigma)^*\}$

  $P \subset V \times (V \cup \Sigma)^*,\ (P \neq \emptyset)$

Habitualmente, para especificar una gramática, se especifican todas sus reglas de producción (P), y se sigue un convenio de notación que permite determinar todos los elementos. El símbolo de arranque es aquel cuyas producciones aparecen en primer lugar en la lista de producciones. Por ejemplo, las siguientes producciones:

$S \to U \mid W$

$U \to TaU \mid TaT$

$W \to TbW \mid TbT$

$T \to aTbT \mid bTaT \mid \epsilon$

definen una gramática independiente del contexto para el lenguaje de cadenas de letras a y b en las que hay un número diferente de unas que de otras. Como se puede apreciar, a partir de las producciones anteriores podremos deducir la especificación completa de la gramática:

- $\Sigma = \{a,\ b\}$
- $V = \{S,\ U,\ W,\ T\}$
- El símbolo de arranque es S
- Las reglas de producción son las de la relación anterior

3. Ejercicio práctico

Para profundizar en el funcionamiento de las gramáticas independientes del contexto, vamos a definir una gramática muy sencilla para trabajar con un subconjunto reducido de expresiones aritméticas. Vamos a definir esta gramática tal y como se muestra a continuación:

- $V = \{E,\ N,\ D\}$
- $\Sigma = \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9\ (,\ ),\ +,\ -\}$
- El símbolo de arranque es E.
- El conjunto de reglas de producción P se define como:

  $E \to (E) \mid E + E \mid E - E \mid N$

  $N \to ND \mid D$

  $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

A partir de la gramática anterior podríamos obtener la expresión aritmética siguiente:

$$(4 + (10 - 7))$$

Para entender cuál es el proceso a partir del cual se obtendría esta expresión, vamos a describir las sucesivas derivaciones que realizaremos a partir de la gramática. En este proceso utilizaremos exclusivamente derivaciones más a la izquierda, esto es, en cada paso sustituiremos siempre el símbolo no terminal que se encuentre más a la izquierda en nuestra cadena de símbolos gramaticales.

$E \Rightarrow (E) \Rightarrow$

$\Rightarrow (E\ +\ E) \Rightarrow$

$\Rightarrow (N\ +\ E) \Rightarrow$

$\Rightarrow (D\ +\ E) \Rightarrow$

$\Rightarrow (4\ +\ E) \Rightarrow$

$\Rightarrow (4\ +\ (E)) \Rightarrow$

$\Rightarrow (4\ +\ (E\ -\ E)) \Rightarrow$

$\Rightarrow (4\ +\ (N\ -\ E)) \Rightarrow$

$\Rightarrow (4\ +\ (ND\ -\ E)) \Rightarrow$

$\Rightarrow (4\ +\ (DD\ -\ E)) \Rightarrow$

$\Rightarrow (4\ +\ (1D\ -\ E)) \Rightarrow$

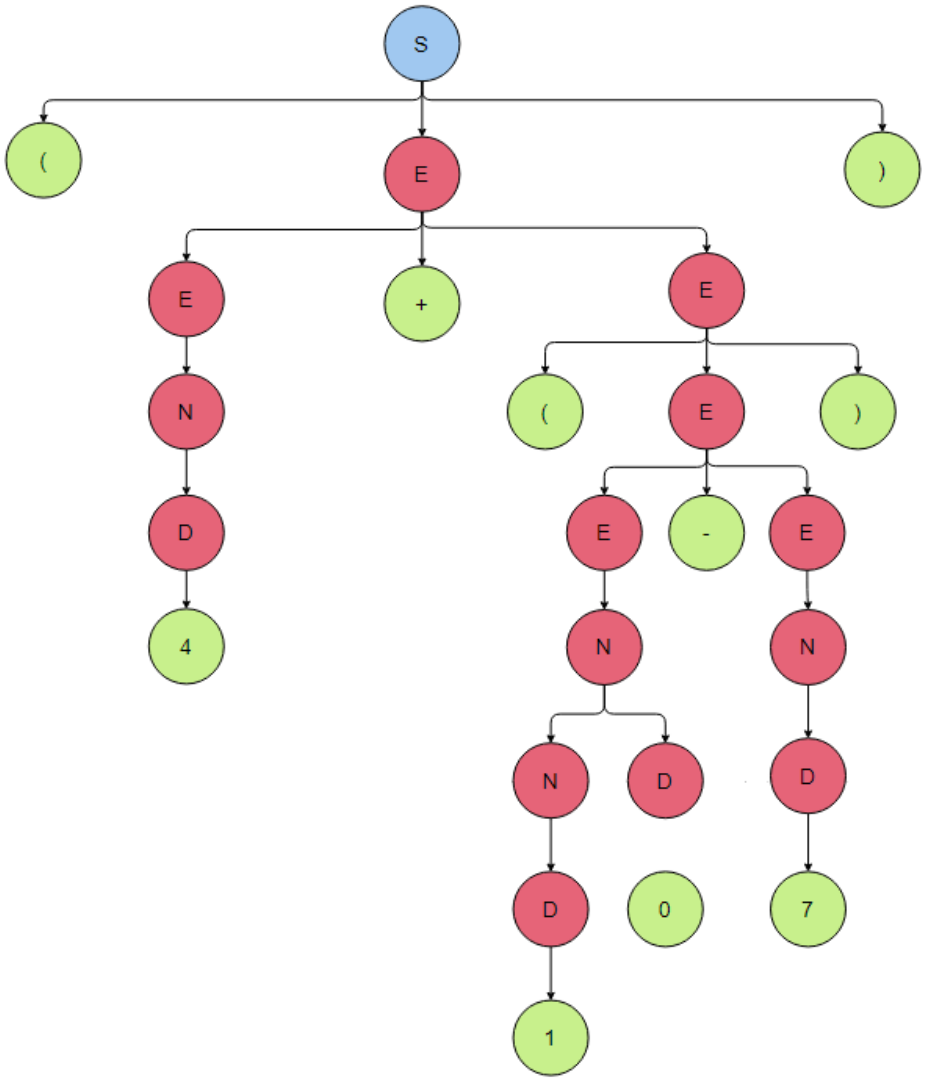

Figura 1: Árbol de análisis sintáctico para la expresión (4 + (10 - 7))

$$\Rightarrow (4 \ + \ (10 \ - \ E)) \Rightarrow$$
$$\Rightarrow (4 \ + \ (10 \ - \ N)) \Rightarrow$$
$$\Rightarrow (4 \ + \ (10 \ - \ D)) \Rightarrow$$
$$\Rightarrow (4 \ + \ (10 \ - \ 7))$$

Este proceso de sustituciones de símbolos no terminales a través de las reglas de producción definidas en la gramática también se podría representar gráficamente mediante un *árbol de análisis sintáctico* (AAS). De hecho la derivación anterior se podría representar gráficamente mediante el AAS de la Figura 1. A partir de la figura podemos apreciar lo

siguiente:

- El nodo raíz está etiquetado con el símbolo de arranque de la gramática. Este nodo aparece en color azul en la figura.
- Los nodos internos del árbol se etiquetan con símbolos no terminales (E, N, D). Estos nodos aparecen en color rojo en la figura.
- Los nodos hoja del árbol se etiquetan con ε o con símbolos terminales (en este caso serían los dígitos del 0 al 9, los paréntesis o los signos de suma y resta). Estos nodos se han representado con fondo verde en la figura.
- Si la producción $A \rightarrow X_1X_2...X_n$ se ha utilizado en la derivación de la cadena, entonces en el AAS, el nodo A aparecerá teniendo como nodos hijos a $X_1, X_2, ..., X_n$. Por ejemplo, en la primera sustitución se aplica la regla de producción $E \rightarrow (E)$, el nodo de arranque S (que en este caso sería E) tiene como hijos a un nodo con el símbolo terminal "(" a un nodo con el símbolo no terminal E y a un nodo con el símbolo terminal ")".
- La cadena derivada puede leerse de izquierda a derecha en los nodos hoja del AAS.

Esta práctica consistirá en la implementación de un programa escrito en C++ que lea desde un fichero `.cfg` las especificaciones de una gramática independiente del contexto y, a continuación, lleve a cabo un proceso de derivaciones hasta obtener una determinada forma sentencial. Las sustituciones a realizar también serán definidas a través de un fichero de entrada con extensión `.drv`. El proceso de derivación llevado a cabo, paso a paso deberá, escribirse en un fichero de salida. Teniendo en cuenta lo anterior, el programa debe ejecutarse como:

```
$ ./p08_grammar input.cfg input.drv output.txt
```

Los ficheros de especificación de gramáticas son ficheros de texto plano con extensión `.cfg`. Estos ficheros contienen los elementos definitorios de una Gramática $G \equiv (\Sigma, V, S, P)$ en este orden: símbolos terminales, símbolos no terminales (siendo S el primero de ellos) y las producciones. El formato de cada uno de estos elementos en el fichero es el siguiente:

1. **Símbolos terminales (alfabeto)**: una línea que contiene, uno a uno y separados por espacios, cada uno de los símbolos del alfabeto.
2. **Conjunto de símbolos no terminales**: una línea que contiene uno a uno y separados por espacios, cada uno de los símbolos no terminales de la gramática. De esta enumeración de símbolos no terminales, tomaremos como **símbolo de arranque** al primero de ellos.
3. **Producciones**: una línea por cada producción de la gramática. Cada producción tendrá el formato siguiente:

   `A ->` α

siendo $\alpha \in (\Sigma \cup V)^*$, es decir una secuencia de símbolos terminales y no terminales. La cadena vacía, ϵ se representa mediante el carácter `&` (código ASCII 126).

A modo de ejemplo, la gramática para trabajar con expresiones aritméticas simples que hemos introducido en este guión de prácticas se puede especificar a través del fichero `.cfg` siguiente:

```
0 1 2 3 4 5 6 7 8 9 ( ) + -
E N D
E -> (E)
E -> E+E
E -> E-E
E -> N
N -> ND
N -> D
D -> 0
D -> 1
D -> 2
D -> 3
D -> 4
D -> 5
D -> 6
D -> 7
D -> 8
D -> 9
```

Si, por otro lado, tenemos el cuenta el proceso de derivación seguido para obtener la expresión $(4 + (10 - 7))$ podríamos definir el fichero `.drv` siguiente:

```
E: 1
E: 2
E: 4
N: 2
D: 5
E: 1
E: 3
E: 4
N: 1
N: 2
D: 2
D: 1
E: 4
N: 2
D: 8
```

En el proceso de derivación anterior, el símbolo no terminal E (símbolo de arranque) se sustituye por (E) haciendo uso de la primera de las producciones que la gramática tiene para el símbolo E. En el segundo paso, se sustituye el no terminal E por $E + E$ aplicando en este caso la segunda de las reglas de producción definidas para E. En el tercer paso se sustituye el primer no terminal E (no terminal de más a la izquierda) por

N (que es la cuarta de las producciones de E) de cara a poder obtener un número. Nótese que a partir del no terminal N se puede obtener un número con tantos dígitos como se desee. En cualquier caso, este proceso de sustituciones podría representarse de forma general, indicando en cada paso el no terminal que se sustituye así como el número (o identificador) de la regla de producción (de dicho no terminal) que se utilizar para hacer la sustitución. Nótese que la primera sustitución debería ser para el símbolo de arranque y que durante todo el proceso se sigue el convenio de realizar siempre la sustitución del símbolo no terminal más a la izquierda: otros tipos de derivaciones no deberían permitirse.

Teniendo en cuenta la gramática para expresiones aritméticas, así como el proceso de sustituciones anterior, el fichero de salida debería ser similar al siguiente:

```
1 E => (E) => (E+E) => (N+E) => (D+E) => (4+E) => (4 + (E)) => (4 + (E-E))
      => (4 + (N-E)) => (4 + (ND-E)) => (4 + (DD-E)) => (4 + (1D-E)) => (4
      + (10-E)) => (4 + (10-N)) => (4 + (10-D)) => (4 + (10-7))
```

4. Consideraciones de implementación

La idea central de la Programación Orientada a Objetos, OOP (*Object Oriented Programming*) es dividir los programas en piezas más pequeñas y hacer que cada pieza sea responsable de gestionar su propio estado. De este modo, el conocimiento sobre la forma en que funciona una pieza del programa puede mantenerse local a esa pieza. Alguien que trabaje en el resto del programa no tiene que recordar o incluso ser consciente de ese conocimiento. Siempre que estos detalles locales cambien, sólo el código directamente relacionado con esos detalles precisa ser actualizado.

Las diferentes piezas de un programa de este tipo interactúan entre sí a través de lo que se llama interfaces: conjuntos limitados de funciones que proporcionan una funcionalidad útil a un nivel más abstracto, ocultando su implementación precisa. Tales piezas que constituyen los programas se modelan usando objetos. Su interfaz consiste en un conjunto específico de métodos y atributos. Los atributos que forman parte de la interfaz se dicen públicos. Los que no deben ser visibles al código externo, se denominan privados. Separar la interfaz de la implementación es una buena idea. Se suele denominar encapsulamiento.

C++ es un lenguaje orientado a objetos. Si se hace programación orientada a objetos, los programas forzosamente han de contemplar objetos, y por tanto clases. Cualquier programa que se haga ha de modelar el problema que se aborda mediante la definición de clases y los correspondientes objetos. Los objetos pueden componerse: un objeto “coche” está constituido por objetos “ruedas”, “carrocería” o “motor”. La herencia es otro potente mecanismo que permite hacer que las clases (objetos) herederas de una determinada clase posean todas las funcionalidades (métodos) y datos (atributos) de la clase de la que descienden. De forma inicial es más simple la composición de objetos que la herencia, pero ambos conceptos son de enorme utilidad a la hora de desarrollar aplicaciones complejas. Cuanto más compleja es la aplicación que se desarrolla mayor es el número de clases involucradas. En los programas que desarrollará en esta asignatura será frecuente la necesidad

de componer objetos, mientras que la herencia tal vez tenga menos oportunidades de ser necesaria.

Tenga en cuenta las siguientes consideraciones:

- No traslade a su programa la notación que se utiliza en este documento, ni en la teoría de Autómatas Finitos. Por ejemplo, el cardinal del alfabeto de su autómata no debiera almacenarse en una variable cuyo identificador sea `N`. Al menos por dos razones: porque no sigue lo especificado en la *guía de estilo* [11] respecto a la elección de identificadores y más importante aún, porque no es significativo. No utilice identificadores de un único carácter, salvo para situaciones muy concretas.
- Favorezca el uso de las clases de la STL, particularmente `std::array`, `std::vector`, `std::set`, `map` o `std::string` frente al uso de estructuras dinámicas de memoria gestionadas a través de punteros.
- Construya su programa de forma incremental y depure las diferentes funcionalidades que vaya introduciendo.
- En el programa parece ineludible la necesidad de desarrollar una clase `Grammar`. Estudie las componentes que definen a una gramática independiente del contexto y vea cómo trasladar esas componentes a su clase `Grammar`.
- Valore análogamente qué otras clases se identifican en el marco del problema que se considera en este ejercicio. Estudie esta referencia [8] para practicar la identificación de clases y objetos en su programa.

5. Criterios de evaluación

Se señalan a continuación los aspectos más relevantes (la lista no es exhaustiva) que se tendrán en cuenta a la hora de evaluar esta práctica:

- Se valorará que el alumnado haya realizado, con anterioridad a la sesión de prácticas, y de forma efectiva, todas las tareas propuestas en este guión. Esto implicará que el programa compile y ejecute correctamente. Además, el comportamiento del programa deberá ajustarse a lo solicitado en este documento.
- También se valorará que, con anterioridad a la sesión de prácticas, el alumnado haya revisado los documentos que se enlazan desde este guión.
- Paradigma de programación orientada a objetos: el programa ha de ser fiel al paradigma de programación orientada a objetos. Se valorará que el alumnado haya identificado clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se plantea.
- Paradigma de modularidad: se valorará que el programa se haya escrito de modo que las diferentes funcionalidades que se precisen hayan sido encapsuladas en métodos

concretos cuya extensión textual se mantuviera acotada.

- Documentación: se requiere que todos los atributos de las clases definidas en el proyecto tengan un comentario descriptivo de la finalidad del atributo en cuestión. Además, se requiere que los comentarios del código fuente sigan el formato especificado por Doxygen [6].
- Se valorará que el código desarrollado siga el formato propuesto en esta asignatura para la escritura de programas en C++.
- Capacidad del programador(a) de introducir cambios en el programa desarrollado.

Si el alumnado tiene dudas respecto a cualquiera de estos aspectos, debiera acudir al foro de discusiones de la asignatura para plantearlas allí. Se espera que, a través de ese foro, el alumnado intercambie experiencias y conocimientos, ayudándose mutuamente a resolver dichas dudas. También el profesorado de la asignatura intervendrá en las discusiones que pudieran suscitarse, si fuera necesario.

Referencias

[1] Transparencias del Tema 3 de la asignatura: Lenguajes y Gramáticas Independientes del Contexto, https://campusingenieriaytecnologia2122.ull.es/mod/resource/view.php?id=4237

[2] Context Free Grammar, https://en.wikipedia.org/wiki/Context-free_grammar

[3] Noam Chomsky https://en.wikipedia.org/wiki/Noam_Chomsky

[4] Chomsky hierarchy https://en.wikipedia.org/wiki/Chomsky_hierarchy

[5] JavaScript 1.4 Grammar https://www-archive.mozilla.org/js/language/grammar14.html

[6] Doxygen http://www.doxygen.nl/index.html

[7] Diez consejos para mejorar tus comentarios de código fuente https://www.genbeta.com/desarrollo/diez-consejos-para-mejorar-tus-comentarios-de-codigo-fuente

[8] Cómo identificar clases y objetos http://www.comscigate.com/uml/DeitelUML/Deitel01/Deitel02/ch03.htm

[9] Makefile Tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor

[10] C++ Makefile Tutorial: https://makefiletutorial.com

[11] Google C++ Style Guide, https://google.github.io/styleguide/cppguide.html