Universidad de La Laguna

Grado en Ingeniería Informática

Computabilidad y Algoritmia

PRÁCTICA 3

Calculadora de lenguajes formales

Factor de ponderación: 8

1. Objetivos

El objetivo de la práctica es diseñar una calculadora para lenguajes formales. El usuario podrá especificar lenguajes de entrada (operandos) así como diferentes operaciones (y combinaciones de operaciones) a realizar con dichos lenguajes. Además de profundizar en los conceptos y operaciones relacionadas con cadenas y lenguajes, también se pretende poner en práctica algunas cuestiones relacionadas con la programación en C++.

Siguiendo la dinámica de prácticas anteriores, algunos de los aspectos generales relacionados con el desarrollo de programas en C++ 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. Algunos de los principales criterios de estilo se describirán brevemente en la sección 3 de esta práctica. Estos criterios son los mismos que los ya introducidos en la práctica anterior.
- **Sets de la STL**: para la definición de conjuntos de elementos [9, 10] se deberá utilizar el contenedor `set` [8] de la STL [11] (librería estándar de C++).
- **Sobrecarga de operadores**: para la comparación entre elementos, así como para las realización de algunas de las operaciones con cadenas o con lenguajes, se deberán

aplicar los principios de la sobrecarga de operadores en C++ [12, 13].

- **Operadores de entrada/salida**: para leer o escribir símbolos, alfabetos, cadenas o lenguajes, se deberán definir los correspondientes operadores de entrada y salida estándar [14].
- Compilación de programas utilizando `make` [3, 4].

2. Ejercicio práctico

El ejercicio práctico a desarrollar se plantea como una continuación del programa desarrollado en las prácticas anteriores. Teniendo en cuenta las nociones básicas sobre *símbolos*, *alfabetos*, *cadenas* y *lenguajes*, así como las correspondientes operaciones con cadenas o lenguajes [1], en este caso nos centraremos en utilizar una notación específica para introducir combinaciones de operaciones con lenguajes. La idea general es diseñar una calculadora para lenguajes formales.

El objetivo de esta práctica será desarrollar un programa cliente que, dados unos lenguajes de entrada y una especificación de operaciones con dichos lenguajes, ejecute las operaciones correspondientes y escriba por pantalla los resultados obtenidos. La ejecucion del programa en línea de comandos debería ser similar a lo siguiente:

```
./p03_calculator filein.txt
```

En el fichero de entrada encontraremos la definición de los lenguajes así como la especificación de las operaciones a realizar con ellos. Un ejemplo de fichero de entrada podría ser el siguiente:

```
L1 = {0, 1, 01, 10}
L2 = {&, 10, 11}
L3 = {&}
L4 = {}
L1 L2 | L3 +
L2 L3 ^ L1 L2 ^ +
```

Para cada especificación de un lenguaje, tendremos una línea en la que se incluye, en primer lugar, el identificador del lenguaje, seguido de un igual y luego entre llaves, y separadas por comas, las diferentes cadenas que conforman el lenguaje. En este caso, utilizaremos los espacios para separar la variable (donde almacenar el lenguaje) y el símbolo igual (=) así como para separar el propio símbolo = de la especificación de las cadenas que conforman el lenguaje. El conjunto de cadenas del lenguaje se especificarán entre llaves. Se utilizarán comas para separar las cadenas y se incluirá siempre un espacio tras cada coma.

Teniendo en cuenta este formato para los ficheros de entrada, las llaves, el símbolo igual, las comas y los espacios no deberían usarse como símbolos de los alfabetos sobre los que se forman los lenguajes. Además, cabe destacar que para especificar la cadena vacía vamos a utilizar el símbolo &. Por lo tanto, para evitar confusiones, el símbolo & tampoco podrá utilizarse como elemento de los alfabetos que se utilizarán en esta práctica. A modo de ejemplo, la tercera línea del fichero de entrada anterior definiría al lenguaje que contiene únicamente a la cadena vacía. Por último, para especificar el lenguaje vacío se propone utilizar las llaves sin definir ninguna cadena dentro, tal y como se ha hecho en la cuarta línea del ejemplo anterior.

Tal y como se puede apreciar, en el fichero de entrada no hay una especificación explícita de los alfabetos, con lo cual para cada lenguaje habrá que generar, de forma autómatica, el alfabeto sobre el cual se han definido sus cadenas.

Por otro lado, para la especificación de las operaciones entre lenguajes utilizaremos la notación polaca inversa [2]. La notación polaca inversa o notación posfija (en inglés, *Reverse Polish Notation* o *RPN*) es un método algebraico alternativo de introducción de datos. En la notación polaca inversa primero están los operandos y después viene el operador que va a realizar los cálculos sobre ellos. En la notación polaca inversa no es necesario usar paréntesis para indicar el orden de las operaciones pues dicho orden va implícito en la propia notación.

Algorithm 1 Evaluación de expresiones en notación postfija

Require: Dada una expresión en notación postfija.

```
1:  while hay elementos en la bandeja de entrada: do
2:      Leer el primer elemento de la bandeja de entrada.
3:      if el elemento es un operando then
4:          Poner el operando en la pila.
5:      else
6:          if el elemento es un operador que toma n operandos then
7:              if hay menos de n operandos en la pila then
8:                  ERROR: Operandos insuficientes en la expresión.
9:              else
10:                 Tomar los últimos n operandos de la pila.
11:                 Evaluar la operación con respecto a los operandos.
12:                 Introducir el resultado en la pila.
13:             end if
14:         end if
15:     end if
16: end while
17: if hay un solo elemento en la pila then
18:     El valor de ese elemento es el resultado del cálculo.
19: else
20:     if Si hay más de un elemento en la pila then
21:         ERROR: Expresión incorrecta.
22:     end if
```

El algoritmo que utilizan las calculadoras RPN es relativamente simple tal y como se puede apreciar en el Algoritmo 1. La bandeja de entrada (o entrada simplemente) son los elementos que vamos encontrándonos en la propia expresión cuando la evaluamos de izquierda a derecha. La "pila" es la lista de los valores que el algoritmo mantiene en su memoria después de leer operandos o de realizar una determinada operación. A modo de ejemplo, veamos la expresión (en notación postfija) siguiente y cómo se evalúa siguiendo el Algoritmo 1:

$$5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$$

Entrada	Acción	Pila
5	Introducir en la pila el 5	5
1	Introducir en la pila el 1	5, 1
2	Introducir en la pila el 2	5, 1, 2
+	Tomar los dos últimos valores de la pila (1, 2), hacer 1 + 2 y sustituirlos por el resultado (3)	5, 3
4	Introducir en la pila el 4	5, 3, 4
*	Tomar los dos últimos valores de la pila (3, 4), hacer 3 * 4 y sustituirlos por el resultado (12)	5, 12
+	Tomar los dos últimos valores de la pila (5, 12), hacer 5 + 12 y sustituirlos por el resultado (17)	17
3	Introducir en la pila el 3	17, 3
-	Tomar los dos últimos valores de la pila (17, 3), hacer 17 - 3 y sustituirlos por el resultado (14)	14

Puesto que al final del proceso queda un solo elemento, dicho valor (el 14 en nuestro caso) constituirá el resultado final de la expresión evaluada. Por otro lado, y atendiendo al proceso anterior, podemos ver que la expresión algebraica equivalente en notación infija sería la siguiente:

$$(5 + ((1 + 2) * 4)) - 3$$

Volviendo al fichero de entrada de nuestra calculadora, tendremos que aplicar el Algoritmo 1 para evaluar aquellas líneas que constituyen expresiones en notación postfija. Por ejemplo, al especificar la expresión $L1\ L2 \mid L3\ +$ debemos tener en cuenta que al evaluar deberíamos realizar las operaciones de tal forma que la expresión equivalente en notación infija sea: $(L1 \mid L2)\ +\ L3$. Por su parte, la expresión postfija $L2\ L3\ \wedge\ L1\ L2\ \wedge\ +$ tendría como equivalente la siguiente expresión infja: $(L2 \wedge L3) + (L1 \wedge L2)$.

Tal y como se ha mencionado, cada línea del fichero de entrada contendrá o bien la especificación de un lenguaje o bien la especificación de una expresión en notación postfija. Como salida del programa, se mostrará **por pantalla** el resultado de evaluar cada una de las expresiones (en notación postfija) contenidas en el fichero de entrada.

Por último, es importante tener en cuenta cuáles serán los símbolos que utilizaremos en las expresiones para denotar a cada una de las operaciones implementadas con lenguajes:

Operación	Operador	Notación infija	Notación postfija
Concatenación	$+$	$L1 + L2$	$L1\ L2\ +$
Unión	$\|$	$L1 \mid L2$	$L1\ L2\ \|$
Intersección	$\wedge$	$L1 \wedge L2$	$L1\ L2\ \wedge$
Diferencia	$-$	$L1 - L2$	$L1\ L2\ -$
Inversa	!	$!L1$	$L1\ !$
Potencia	$*$	$L1 * n$	$L1\ n\ *$

3. Estilo y formato del código

Esta relación no pretende ser exhaustiva. Contiene una serie de requisitos y/o recomendaciones cuyo cumplimiento se impulsará a la hora de evaluar TODAS las prácticas de la asignatura.

- Estudien la finalidad de los especificadores de visibilidad (public, private, protected) de atributos y métodos. Una explicación breve y simple es ésta [5] aunque es fácil hallar muchas otras. Se debe restringir los atributos y métodos públicos a aquellos que son estrictamente necesarios para que un programa cliente utilice la clase en cuestión.
- El principio de responsabilidad única (*Single responsibility principle, SRP*) [6] es uno de los conocidos como principios "SOLID" y es posiblemente uno de los más fáciles de comprender. Dejando a un lado el resto de principios SOLID, que pueden ser más complejos, merece la pena que lean algo sobre el SRP. Ese principio es una de las razones por las que en sus programas no debiera haber una única clase "*que se encarga de todo*". En esta referencia [7] tienen un sencillo ejemplo (el de la clase `Person` que puede servir para ilustrar el SRP, pero no es difícil encontrar múltiples ejemplos del SRP. Tengan en cuenta que es un principio de orientación a objetos, y por lo tanto independiente del lenguaje de programación: pueden encontrar ejemplos de código en C++, Java, Pythos u otros lenguajes.
- Un programa simple en C++ debiera incluir al menos 3 ficheros:
  - `X.h` para la definición de la clase X
  - `X.cc` para la implementación de la clase X
  - `ClienteX.cc` para el programa "cliente" que utiliza la clase X
- Para evitar la doble inclusión de un mismo fichero (habitualmente un fichero de cabecera, *header*) en un código fuente hay básicamente dos posibilidades: las *include guards* [18] y el uso de la directiva *#pragma once* [19]. Es conveniente que conozcan ambas alternativas, el funcionamiento de cada una y que tengan en cuenta que la

directiva `#pragma once`, aunque soportada por la mayor parte de los compiladores de C++, no es estándar. Para más información sobre este asunto pueden leer esta discusión de StackOverflow [20].

- Ejemplo de comentario de cabecera (inicial) para (todos) los ficheros de un proyecto (práctica) de la asignatura:

```
// Universidad de La Laguna
// Escuela Superior de Ingeniería y Tecnología
// Grado en Ingeniería Informática
// Asignatura: Computabilidad y Algoritmia
// Curso: 2º
// Práctica 3: Calculadora de lenguajes formales
// Autor: Nombre y Apellidos
// Correo: aluXXXXX@ull.edu.es
// Fecha: 17/10/2022
//
// Archivo cya-P03-Calculator.cc: programa cliente.
//         Contiene la función main del proyecto que usa las clases
//         X e Y para ... (indicar brevemente el objetivo)
//         Descripción otras funcionalidades
//
// Referencias:
//          Enlaces de interés
/
// Historial de revisiones
//          17/10/2022 - Creación (primera versión) del código
```

- Además de la información anterior sobre el autor, la fecha y la asignatura, la cabecera también debería contener al menos una breve descripción sobre lo que hace el código incluyendo los objetivos del proyecto en general, las estructuras de datos utilizadas, así como un listado de modificaciones (bug fixes) que se han ido introduciendo en el código. El fichero de cabecera sería básicamente el mismo para todos los ficheros (`*.cc`, `*.h`) del proyecto, con la salvedad de que varía la descripción del contenido del fichero y su finalidad.

- El utilizar una estructura de directorios con nombres estándar (`bin`, `lib`, `src`, `config`, etc.) para los proyectos de desarrollo de software es una buena práctica cuando se trata de proyectos que involucran un elevado número de ficheros. No obstante, para la mayoría de prácticas de esta asignatura, que no contienen más de una decena de ficheros, colocar todos los ficheros de cada práctica en un mismo directorio (con nombre signficativo *CyA/practicas/p03_calculator*, por ejemplo) es posiblemente más eficiente a la hora de localizar y gestionar los ficheros de cada una de las prácticas (proyectos).

- En el directorio de cada práctica, como se ha indicado anteriormente, debería haber un fichero `Makefile` de modo que el comportamiento del programa make fuera tal

que al ejecutar:

```
make clean
```

el programa dejara en el directorio en que se ejecuta solamente los ficheros conteniendo código fuente (ficheros `*.h` y `*.cc`) y el propio `Makefile`. En estos breves tutoriales [3, 4] se explica de forma incremental cómo construir un fichero Makefile para trabajar con la compilación de proyectos simples en C++.

- A continuación enumeraremos algunas cuestiones relativas al formato del código:

  1. A ambos lados de un operador binario han de escribir un espacio:

     ```
     a + b
     ```

     En lugar de:

     ```
     a+b
     ```

  2. SIEMPRE después de una coma, ha de ir un espacio.
  3. Se debe indentar el código usando espacios y NO tabuladores. Cada nivel de indentación ha de hacerse con 2 espacios.
  4. TODO identificador (de clase, de método, de variable, ...) ha de ser significativo. No se pueden usar identificadores de un solo caracter salvo para casos concretos (variable auxiliar para un bucle o similar).
  5. TODO fichero de código de un proyecto ha de contener un prólogo con comentarios de cabecera donde se indique al menos: Autor, datos de contacto, Fecha, Asignatura, Práctica, Finalidad del código. Veáse el ejemplo de cabecera proporcionado en este documento.
  6. Asimismo todos los métodos y clases han de tener al menos un mínimo comentario que documente la finalidad del código.
  7. No comentar lo obvio. No se trata de comentar por comentar, sino de aclarar al lector la finalidad del código que se escribe [16].
  8. Como regla de carácter general, el código fuente de los programas no debiera contener de forma explícita constantes. En lugar de escribir

     ```
     double balance[10];
     ```

     Es preferible:

     ```
     const int SIZE_BALANCE = 10;
     ...
     double balance[SIZE_BALANCE];
     ```

Para el caso de las constantes numéricas, las únicas excepciones a esta regla son los valores 0 y 1. Tengamos en cuenta que las constantes no son solamente numéricas: puede haber constantes literales, de carácter o de otros tipos. Así en lugar de escribir:

```
myString.find(' ');
```

Es preferible:

```
const char SPACE = ' ';
...
myString.find(SPACE);
```

9. Cuando se programa un código encadenando múltiples sentencias if-else, ello suele ser síntoma de una mala práctica y, en efecto, debería buscarse alguna alternativa. En este sentido les recomendamos que revisen las diferentes sugerencias que se realizan al hilo de esta discusión [17].

10. Como regla general, se espera que todo el código fuente siga la guía de estilo de Google para C++ [15]. Dentro de esa guía, y para comenzar, prestaremos particular atención a los siguientes aspectos:

    - Formateo del código (apartado *Formatting* en [15])
    - Comentarios de código de diverso tipo (apartado *Comments* en [15])
    - Nominación de identificadores, clases, ficheros, etc. (apartado *Naming* en [15])

4. 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.
- 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: se valorará que el alumnado haya identificado clases y objetos que permitan modelar adecuadamente el escenario de trabajo que se planeta.
- 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.
- Sobrecarga de operadores: se valorará que el alumnado haya aplicado los principios básicos para la sobrecarga de operadores en C++.
- Uso de contenedores de la librería estándar: se valorará el uso de contenedores estándar ofrecidos en la STL, especialmente el uso de `sets`.
- 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 1 de la asignatura: Alfabetos, cadenas y lenguajes, https://campusingenieriaytecnologia2223.ull.es/mod/resource/view.php?id=5914

[2] Notación polaca o notación postfija: https://es.wikipedia.org/wiki/Notaci%C3%B3n_polaca_inversa

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

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

[5] Difference between private, public and protected inheritance: https://stackoverflow.com/questions/860339/difference-between-private-public-and-protected-inheritance

[6] Single-Responsability Principle (SRP): https://en.wikipedia.org/wiki/Single_responsibility_principle

[7] Example of the SRP: https://stackoverflow.com/questions/10620022/what-is-an-example-of-the-single-responsibility-principle

[8] STL sets, http://www.cplusplus.com/reference/set/set

[9] STL sets example, https://thispointer.com/stdset-tutorial-part-1-set-usage-details

[10] STL sets example, https://tinyurl.com/cyasetexample

[11] Using the C++ Standard Template Libraries, https://link.springer.com/book/10.1007%2F978-1-4842-0004-9

[12] Sobrecarga de operadores, https://es.cppreference.com/w/cpp/language/operators

[13] C++ Operator Overloading Example, https://www.programiz.com/cpp-programming/operator-overloading

[14] Input/Output Operators Overloading in C++, https://www.tutorialspoint.com/cplusplus/input_output_operators_overloading.htm

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

[16] Commenting code: https://www.cs.utah.edu/~germain/PPS/Topics/commenting.html

[17] If-else-if chains or multiple-if: https://stackoverflow.com/questions/25474906/which-is-better-if-else-if-chain-or-multiple-if

[18] Include guard: https://en.wikipedia.org/wiki/Include_guard

[19] Pragma once: https://en.wikipedia.org/wiki/Pragma_once

[20] Pragma once vs. Include guards: https://stackoverflow.com/questions/1143936/pragma-once-vs-include-guards