Teoría de Lenguajes - Trabajo Práctico
SLS: Un simple lenguaje de scripting

Versión 1.0

1er cuatrimestre 2016

Fecha de entrega: jueves 30 de junio

1. Introducción

Se desea incorporar un lenguaje de scripting, denominado Simple Lenguaje de Scripting (SLS), a un sistema de software ya existente. Para ello se requiere desarrollar un analizador léxico y sintáctico para este lenguaje.

Así, se recibirá como entrada un código fuente, el cual se deberá chequear por si cumple la sintaxis y restricciones de tipado del lenguaje, para, finalmente, formatear el código con la 'indentación' adecuada para SLS. En caso de haberse detectado algún error, se deberá informar claramente cuáles son las características del mismo.

2. Descripción del lenguaje de entrada

El lenguaje sigue los lineamientos generales de sintaxis y convenciones básicas del lenguaje ISO C++, y, en particular, cuenta con las siguientes características:

- **Código de entrada:** Se considera inválido el ingreso de un código vacío o sólo lleno con espacios, tabulados, saltos de línea u otros caracteres no imprimibles.
- **Sentencias:** Todas las sentencias del lenguaje terminarán con punto y coma.
- **Comentarios:** Existe un solo tipo de comentario, que requiere marcar sólo su comienzo con el símbolo numeral ("#"). Toda cadena que se encuentre luego del marcador de comentario hasta el próximo salto de línea deberá ser ignorada.
- **Bloques:** Todos los bloques de código definidos, encerrados entre llaves, deben contener al menos una sentencia.
- **Condicional (if else):** Los condicionales reciben una condición encerrada entre paréntesis, y luego un bloque con las sentencias en caso de que se cumpla la condición, y un bloque `else`, opcional. En caso de que alguno de estos bloques tenga sólo una sentencia, no será necesario utilizar las llaves.
- **Bucles:** Este lenguaje permitirá el uso de tres tipos de bucles: for, while y do-while. En todos los casos, si sólo abarcan una sentencia se puede obviar la utilización de la apertura y cierres de bloque (i.e., las llaves).
  - *for*: Como es usual, tiene tres componentes entre paréntesis, de los cuales sólo el segundo es obligatorio, separados por punto y coma: la inicialización, la condición de corte y, finalmente, la acción a ejecutar luego de cada paso (e.g., incremento o decremento de una variable). Luego, está seguido por un bloque de instrucciones.
  - *while* : Recibe la condición de ejecución entre paréntesis, y luego tiene el bloque de sentencias a ejecutar.
  - *do while*: Comienza con el keyword `do`, luego contiene el bloque de sentencias a ejecutar, y por último el `while`, con la condición de ejecución entre paréntesis, y el cierre con ";".

- **Funciones disponibles:** El lenguajes sólo cuenta con las siguientes funciones, que pueden recibir variables o literales como argumento:
  - *multiplicacionEscalar:* Recibe tres argumentos, siendo el último de ellos opcional: un vector de números, un número y un booleano; y devuelve un vector numérico. El booleano indica si se debe truncar los decimales, en caso de haberlos. Por defecto, si no se especifica el tercer argumento, se muestran los decimales.
  - *capitalizar:* Recibe una cadena como argumento, y devuelve la misma cadena, pero con todas sus palabras con la primera letra en mayúscula, y el resto en minúscula.
  - *colineales:* Recibe dos vectores numéricos, y devuelve un valor lógico indicando si se trata de vectores colineales[1].
  - *print:* Recibe un argumento de cualquier tipo, y lo imprime en pantalla.
  - *length:* Recibe una cadena o vector, y retorna su longitud.
- **Operadores**
  - *Asignación:* La asignación de valores a las variables se realizará con el símbolo =. Adicionalmente, podrán utilizarse los operadores `+=`, `-=`, `*=` y `/=` con la connotación habitual.
  - *Matemáticos:* Se podrán utilizar los operadores binarios `-`, `*`, `^`, `%` y `/` sólo para numéricos, y `+` también para cadenas, con la semántica habitual; los operadores unarios de autoincremento/autodecremento (`++` y `--`) como prefijo y sufijo, además de los indicadores de signo (`+` y `-`), en todos los casos, sólo para los números.
  - *Relacionales:* Podrán utilizarse los operadores de igualdad (`==`), desigualdad (`!=`), símbolos de mayor y menor (`>` y `<`), todos con el significado habitual, y sólo con variables del mismo tipo.
  - *Lógicos:* Estarán disponibles los operadores AND, OR y NOT, sólo para booleanos.
  - *Operador ternario:* Se deberá poder utilizar el operador condicional ternario (`?:`) para la asignación de variables (por ej.: `x = (a < b)? a:0;`). Como es usual, la condición deberá ser booleana, y ambos resultados alternativos deberán ser del mismo tipo.
- **Tipos de datos**
  - El lenguaje SLS permitirá trabajar con variables —sin declaraciones ni tipos predefinidos—, las cuales podrán tener nombre alfanumérico (letras mayúsculas o minúsculas y dígitos) y guión bajo, pero con su primer caracter siendo una letra.
  - Las expresiones podrán ser de tipo cadena, booleano, numérico (con o sin decimales), o vectores, con valores todos del mismo tipo, y registros (i.e., de la forma registro.campo), donde cada campo puede ser de distinto tipo.
    - Para los vectores, se puede asignar un valor indicando la posición entre corchetes (e.g., `miVector[3] = 32;`), comenzando con el índice 0, además de asignar múltiples valores, entre corchetes y separados por comas (e.g., `miVector = [1, 3.2, 14, 44];`). Para el primer caso de asignación, si no se define el valor de una posición de un vector, esta tomará el valor por defecto de 0, para numéricos, o `""`, para cadenas.
    - Para los registros, se puede definir el valor de cada campo en forma individual (e.g., `alumno.nombre = "Juan Perez"; alumno.edad=23;`), o en un solo paso (e.g., `alumno = {nombre:"Juan Perez", edad:23};`).
- **Variables del sistema:** En caso de requerirlo, se podrá utilizar la variable predefinida `res`, en donde se podrá guardar el resultado de la ejecución del script, que será retornada al finalizar.

[1] Se denomina vectores colineales a aquellos que tienen la misma dirección.

- **Palabras reservadas:** Las siguientes palabras son reservadas y no pueden utilizarse para las variables a utilizar en el programa, ya sean en mayúsculas o minúsculas: `begin`, `end`, `while`, `for`, `if`, `else`, `do`, `res`, `return`, `true`, `false`, `AND`, `OR`, `NOT`.
- **Otras definiciones:** Este lenguaje no servirá para declarar ni implementar funciones adicionales a las ya existentes. Para otras definiciones no especificadas o ante cualquier duda, consultar con el equipo de docentes.

3. Descripción del lenguaje de salida

Como salida, el programa deberá retornar el código fuente original, manteniendo los comentarios, formateado como se describe a continuación:

- Cada sentencia del código deberá ocupar una línea.
- No deberá haber líneas en blanco.
- Cada línea deberá estar indentada adecuadamente, usando tabulados, de acuerdo a qué bloque de código pertenezca cada sentencia.
- Los comentarios deben mantener el formato que tenían originalmente, pero deberán estar indentados de acuerdo al lugar del código en el que se encuentren (por ej. si están dentro de un if, aparecerán indentados).

4. Ejemplos

A continuación se muestran ejemplos de entradas, a la izquierda, y salidas esperadas, a la derecha, para el analizador sintáctico solicitado.

```
# Este programa funciona bien

a=100;
for (i=1; i<a;i++) b = b+10*i;

#Aquí se devuelve el resultado esperado
res = b * 0.8;
```

```
# Este programa funciona bien
a = 100;
for (i=1; i<a; i++)
      b = b + 10 * i;
#Aquí se devuelve el resultado esperado
res = b * 0.8;
```

[VÁLIDO]

```
inicio = "Hola";
i =0;
do{ i++;
inicio += " "; # Se agrega un espacio
j = length(inicio) * i;
}while (j % 32 != 0); print ("Mundo!");
```

```
inicio = "Hola";
i = 0;
do{
      i++;
      inicio += " "; # Se agrega un espacio
      j = length(inicio) * i;
}while (j % 32 != 0);
print ("Mundo!");
```

[VÁLIDO]

```
1  do{
2        a += 2;
3  }while(true)
```

[INVÁLIDO] Error: Falta ; luego de while (línea 3).

```
1  i = 0;
2  while(true)
3  #Comentario antes del if
4  if (i<2)
5  {
6  #Algo sobre el then del if
7  valores[i]=i;
8  i++;}
9  else
10 #Algo sobre el else
11 valores[i] = valores[i-1]+valores[i-2];
12 res = valores;
```

```
1  i = 0;
2  while(true)
3        #Comentario antes del if
4        if (i < 2){
5              #Algo sobre el then del if
6              valores[i] = i;
7              i++;
8        }
9        else
10             #Algo sobre el else
11             valores[i] = valores[i-1] + valores[i-2];
12 res = valores;
```

[VÁLIDO]

```
1  r = 18;
2  pi = 3.14159;
3  a = pi * r^2;
4
5  i = a * 10;
6  if(a > i)
7        i=-0.5;
8        print("Se redujo 0.5");
9  else
10       print("No fue necesario reducir nada");
```

[INVÁLIDO] Error: Hay un `else` sin haber un `if` correspondiente (línea 9).

Este error surge porque la primera línea dentro del `if` se considera como la única sentencia del mismo, y el `print` siguiente se evalúa como fuera del `if`. Luego, al momento de procesar el `else` no hay ningún `if` abierto al cual pudiera corresponder.

```
1  unNumero = "uno";
2  nombres = [ "hola", "qué", "tal"];
3
4  segundaPalabra = nombres[unNumero];
```

[INVÁLIDO] Error: Se esperaba un número natural como índice del vector `nombres` (línea 4).

---

5. Modo de uso

El programa deberá poder ejecutarse como un comando de consola del sistema operativo, con los siguientes detalles.

Sinopsis:

./SLSParser [-o SALIDA] [-c ENTRADA | FUENTE]

Descripción:

El programa deberá recibir el código fuente a procesar, y retornar el código resultante.
Se puede especificar un nombre de archivo de entrada, y en caso de que no se especifique, se esperará recibir la cadena a procesar (FUENTE) en la misma llamada o que esta se reciba por standard input (`stdin`). En caso de no especificar un archivo de salida, el resultado se imprimirá por standard ouput (`stout`).
En caso de que hubiera algún problema en la llamada, se deberá terminar el programa con textcolorredcódigo de salida de error, y mostrar los detalles por standard error (`stderr`). En caso de que la llamada fuera correcta, se deberá devolver el código resultante, y, si no, incluir todos los detalles del error por `stderr` y no retornar nada por `stdout`.

Opciones

-o SALIDA	Se especifica un archivo de salida para el código formateado
-c ENTRADA	Se especifica el nombre del archivo de entrada con el código fuente
FUENTE	Cadena con el código fuente.

6. Implementación

Para el lenguaje descripto se solicita crear un analizador léxico, generando los tokens necesarios de acuerdo a la gramática que se haya diseñado, y un analizador sintáctico, que deberá generar el árbol sintáctico para la cadena procesada, en caso de que sea válida, o un mensaje de error adecuado, en caso de que no lo sea. Para este último caso, será deseable indicar el lugar preciso (línea y posición) en el cual se encuentra el problema detectado.

Hay dos grupos de herramientas que se pueden utilizar para generar los analizadores léxicos y sintácticos:

- uno utiliza expresiones regulares y autómatas finitos para el análisis lexicografico, y la técnica LALR para el análisis sintáctico. Ejemplos de esto son lex y yacc, que generan código C o C++, JLex y CUP, que generan código Java y ply, que genera código Python. flex y bison son implementaciones libres y gratuitas de lex y yacc.
  En particular, permitiremos utilizar PLY 3.6[2] (o superior).

- el otro grupo utiliza la técnica ELL(k), tanto para el análisis léxico como para el sintáctico, generando parsers descendentes iterativos recursivos. Ejemplos son JavaCC y ANTLR, que están escritos en Java. JavaCC puede generar código Java o C++. ANTLR puede generar Java, C#, Python y JavaScript.
  En particular, para este trabajo se podrá usar ANTLR 4.X[3], y se recomienda utilizar el plugin[4] para Eclipse.

[2] PLY: Python Lex & Yacc. Disponible en https://pypi.python.org/pypi/ply
[3] ANTLR (ANother Tool for Language Recognition): http://www.antlr.org
[4] ANTLR plugin for Eclipse: http://antlreclipse.sourceforge.net

7. Detalles de la entrega

Se deberá enviar el código a la dirección de e-mail tptleng@gmail.com, satisfaciendo lo siguiente:

- el **asunto de mail** debe ser `[TL-TP]` seguido por el nombre del grupo (e.g., "[TL-TP] La banda de Kleene").
- en el mail deberán estar copiados todos los integrantes del grupo.

La entrega debe incluir lo descripto a continuación:

- un programa que cumpla con lo solicitado,
- el código fuente del mismo, adecuadamente documentado,
- informe enviado por e-mail y entregado impreso, con los siguientes contenidos:
  - carátula con datos de integrantes del grupo y nombre del grupo,
  - breve introducción al problema a resolver,
  - la gramática (incluyendo las expresiones regulares de los tokens) obtenida a partir del enunciado y las transformaciones de la misma, en caso de haberlas, que hubieran sido necesarias para la implementacion (explicándolas y justificándolas),
  - indicación del tipo de la gramática definida, de acuerdo a los vistos en clase,
  - el código de la solución. Si se usaron herramientas generadoras de código, imprimir la fuente ingresada a la herramienta, no el código generado,
  - descripción de cómo se implementó la solución, con decisiones que hayan tenido que tomar y justificación de las mismas,
  - información y requerimientos de software para ejecutar y recompilar el tp (versiones de compiladores, herramientas, plataforma, etc). Sería como un pequeño manual del usuario, que además de los requerimientos, contenga instrucciones para compilarlo, ejectuarlo, información de parámetros y lo que consideren necesario,
  - casos de prueba con expresiones sintácticamente correctas e incorrectas (al menos tres para cada caso) y
  - un resumen de los resultados obtenidos, y conclusiones del trabajo.

Es parte de lo que se espera de la resolución del trabajo práctico la detección de puntos no especificados en el enunciado y su resolución. En cualquier caso, pueden realizar consultas al respecto.

Referencias

[1] Aho, A.V., Lam, M.S., Sethi, R., *Compilers: Principles, Techniques and Tools - Second Edition*, Pearson Education, 2007.

[2] Goyvaerts, J., Levithan, S., *Regular Expressions Cookbook*, O'Reilly, 2009.

[3] Grune, D., Jacobs, C.J.H, *Parsing Techniques: A Practical Guide - Second Edition*, Springer, 2008.

[4] Hopcroft, J.E., Motwani, R., Ullman, J.D., *Introduction to Automata Theory, Languages, and Computation - Third Edition*, Addison Wesley, 2007.

[5] Parr, T., *The Definitive ANTLR 4 Reference*, The Pragmatic Programmers, 2012.

[6] PLY (Python Lex-Yacc). Documentación. Disponible en http://www.dabeaz.com/ply/ply.html