Práctica 2
Analizador Sintáctico TCL

Tomado: LP_01 Prof: Felipe Restrepo Calle.

Grupos: Máximo 2 personas por grupo, ambas personas deberán estar presentes el día de la entrega y estar en capacidad de alterar el programa dado que surjan nuevos requerimientos durante la sustentación).

Nota: Tener en cuenta que el lenguaje Tcl debe ser adaptado para que su gramática sea LL(1). Por lo tanto, deberán seguir únicamente las reglas declaradas en este documento.

Dado el código fuente de un programa en TCL, su tarea consiste en realizar el análisis sintáctico sobre dicho código. Nos enfocaremos en los errores sintácticos generados. En este documento se especifica la manera correcta de mostrar las salidas.

En el caso de que el programa se encuentre bien formado respecto a las reglas especificadas de TCL (para este curso), se debe escribir la siguiente salida:

El analisis sintactico ha finalizado correctamente.

Note que no se permiten tildes en el mensaje de salida.

En caso contrario, es decir, si se encontró un error sintáctico, se debe **abortar** el análisis y escribir únicamente el primer error encontrado.

Consideraciones gramaticales

A continuación se presentarán las reglas gramaticales que debe tener en cuenta a la hora de realizar el análisis sintáctico (tenga en cuenta que estas reglas han sido adaptadas para el curso, por lo que en algunos casos difieren con la documentación original de TCL).

- Un programa escrito en TCL debe seguir la siguiente estructura

```
# Declaracion de cero o mas subrutinas
proc sum {{a} {b} {c}}{
 return [expr {$a + $b + $c}];
}

# Modulo Principal
# Consiste de cero o mas instrucciones

set var 10;

for{set i 1}{$i <= 10}{incr i}{
```

```
 puts $i;
}
```

Si un programa no respeta esta estructura deberá considerarse como un error sintáctico. El Modulo principal puede no contener ninguna instrucción.

Agrupación con corchetes o Ejecuciones

La agrupación con corchetes permite ejecutar comandos antes de las demás instrucciones, dentro de ellos se ejecutan **expresiones** e **instrucciones**, su estructura gramatical está marcada por corchetes **[ ],** dentro de ellas se permiten las siguientes estructuras.

```
[expr{expresion}];
[array size id];
[array exists id];
[identificador {argumento1} {argumento2} .... {argumento n} ]
[gets stdin]
```

En el caso de la llamada de un identificador de una subrutina, el número de argumentos llamados puede ser cero.

Declaración y uso de variables

Como se había especificado en el manual de referencia la declaración y el uso de las variables puede seguir cualquiera de las siguientes estructuras:

```
set variable valor(entero, double o string);
set variable $identificador;
set variable3 [expr{expresion}];
set arreglo( valor(entero, double o string) )valor(entero, double o string);
set arreglo("string") $identificador;
set arreglo([expr{expresion}]) "string";
```

Como se puede observar, la estructura de declaración de variables se conforma por la palabra reservada **set** seguida por el nombre de la variable, opcionalmente se pueden incluir índices para arreglos (los cuales pueden ser valores numéricos, strings o agrupación con corchetes), dichos índices deberán ser escritos dentro de paréntesis. Finalmente se requiere un valor para ser asignado

Get, puts

Los comandos para ingresar e imprimir datos por consola están dados por la siguiente forma gramatical.

```
gets stdin;
puts (ejecucion_de_instruccion);
puts valor(string, double o entero);
puts $identificador;
#ejemplo

puts [expr{2 + 2}];
puts [sum {2} {2} {2}];
puts [gets stdin];
puts 2;
puts "string";
puts $variable;
puts 2.2;

#ejemplo
set variable [gets stdin];
```

Note que la palabra gets siempre debe ser seguida por la palabra stdin, adicionalmente **puts** siempre debe tener un valor para imprimir, este valor puede ser un valor entero, double o string; además se puede imprimir el llamado al resultado de una expresión o de un sub proceso, en ambos casos deberán ser encerrados entre corchetes **[ ]**.

Estructuras de Control

Un aspecto relevante de la estructura gramatical es que pueden tener cuerpos de instrucciones vacíos. Adicionalmente para los casos de la estructuras **for** y **while**, se permite el uso de las instrucciones **continue** y **break**.

if

Para este curso la estructura gramatical que manejaremos será la siguiente:

```
if {expresion} then {
 #cuerpo
} elseif {expresion} then {
   #cuerpo
} else {
   #cuerpo
}
```

Note que las expresiones en el condicional **no** deben ser precedidas por la palabra reservada **expr**, adicionalmente el uso de **else** y **elseif** es opcional.

for

El ciclo for tiene la siguiente estructura gramatical:

```
#ciclo for
for {set identificador valor_entero} {expresion} {incr identificador (valor_entero) } {
 #cuerpo
}

for {set identificador $identificador} {expresion} {incr identificador (valor_entero) } {
 #cuerpo
}

for {set identificador $identificador(1)} {expresion} {incr identificador (valor_entero) } {
 #cuerpo
}

for {set identificador expr{$var + 2}} {expresion} {incr identificador (valor_entero) } {
 #cuerpo
}
```

Como aspectos importantes note que tampoco es permitido el uso de la palabra **expr** para la expresión de evaluación, adicionalmente en la sección de incremento es opcional indicar el valor del incremento.

Durante la declaración de la variable a iterar es posible utilizar un valor (únicamente entero) un identificador o una expresión (mediante usando el formato expr{}).

while

La estructura sintáctica de un ciclo while es muy similar a la de un condicional **if** de la siguiente manera:

```
while {expresion} {
  #cuerpo
}
```

Note que tampoco se permite el uso de la palabra reservada **expr** en la expresión de evaluación.

switch

El comando switch permite escoger una de múltiples opciones en el código. Similar al switch de C, la variable otorgada se compara a un set de valores. En caso de encontrar un valor igual, se ejecuta el cuerpo de este caso. La sintaxis es la siguiente:

```
switch $variable {
case valor_entero {
  #cuerpo
 }
case valor_entero {
  #cuerpo
 }
```

```
 default{
   #cuerpo
 }
}
```

Los valores de cada caso deben ser enteros (no se permiten expresiones ni strings). Note que para fines de esta práctica se añadió la palabra reservada **case** la cual debe ser incluida en la declaración de cada caso. Finalmente el bloque **default** es opcional.

Sub-rutinas

La creación de subrutinas en Tcl se lleva a cabo mediante el comando **proc**, el cual crea un nuevo comando, la sintaxis es la siguiente:

```
proc identificador {{argumento1} {argumento2} {argumento3}} {
   #cuerpo
   return;
}
```

Dentro de las llaves pueden escribirse cero o más argumentos, los cuales deberán ser declarados como un nombre de identificador cada uno encerrado en llaves, el uso de la palabra reservada return es opcional dentro de las sub rutinas.
Para ejecutar una subrutina se debe utilizar la siguiente estructura gramatical

```
[identificador {argumento1} {argumento2} {argumento3}];
```

Note que después del identificador de la subrutina pueden venir cero o más argumentos, cada uno dentro de llaves, cada uno de los cuales puede ser una expresión, el llamado a una ejecución mediante agrupación de corchetes **[ ]** o un identificador (posiblemente de un arreglo).

```
[identificador {$id} {expr{2 + 2}} {[obtenerIndice]} {[array size arreglo]}];
```

Nota: Tenga en cuenta que el comando **return** puede estar presente dentro de cualquier estructura de control que sea incluida dentro de la declaración de una sub rutina.

Expresiones

En Tcl existen dos escenarios en los que deben usarse expresiones, la primera es dentro de una instrucción (asignación, declaración de una variable, acceder a un índice, etc). en esta primera forma la expresión deberá seguir la siguiente estructura gramatical:

```
expr{expresion };
```

Como se puede ver esta estructura se compone de la palabra **expr** seguida de una expresión encerrada entre llaves.

El segundo escenario es en una condición para una estructura de control. Dentro de estas estructuras las expresiones no deberán ser antecedidas por la palabra **expr** ni deberán ser encerradas entre llaves (a parte de las propias de cada estructura de control).

```
while {expresion} {
 ...
}
```

- Las expresiones pueden ser de cualquier tipo de dato (de momento solo hacemos un análisis sintáctico, el chequeo de tipos pertenece al análisis semántico).

En Tcl existen una serie de operadores los cuales pueden ser divididos en binarios y unarios. Los operadores binarios como su palabra lo indica deben operar dos expresiones mientras que los unarios solamente una (la derecha del operador en este caso).

a = expresion **OP_BINARIO** expresion
b = **OP_UNARIO** expresion
c = expresion **OP_BINARIO** (**OP_UNARIO** (expresion))
d = expresion **OP_BINARIO OP_UNARIO** expresion

Las expresiones pueden ser rodeadas de una cantidad arbitraria de paréntesis, siempre y cuando exista la misma cantidad de paréntesis abriendo que cerrando y la expresión esté formada correctamente; sin embargo el uso de paréntesis sin agrupar ninguna expresión **no** está permitido.

Arreglos

El acceso a los arreglos sigue la siguiente forma

```
identificador(indice (valor entero, double o string) );
identificador([ejecucion] );
#para acceder
$identificador(indice (valor entero, double o string );
```

Especificaciones de entrada y salida

Errores Sintácticos

Los errores sintácticos se producen cuando el programa no respeta la estructura gramatical mencionada. Cuando se presenta alguno **se debe reportar y abortar el análisis**.

El formato para reportar un error sintáctico es el siguiente:

<fila:columna> Error sintactico: se encontro: {lexema del token encontrado}; se esperaba: {lista de símbolos/tokens esperados}.

Donde:

- fila y columna son los números de línea y columna donde se detectó el error.
- lexema del token encontrado: corresponde al lexema encontrado que no se esperaba encerrado entre comillas simples.
- lista de símbolos/tokens esperados: corresponde a la lista de tokens esperados o los lexemas esperados cuando estos tokens representen un único símbolo del lenguaje. Los elementos de la lista deben ir separados por comas y encerrados entre comillas simples. Por ejemplo: ';', ')', ','.

Consideraciones para reporte de errores

- Cuando es necesario reportar palabras reservadas se debe hacer imprimiendo la palabra encontrada en minúscula.
- Para los lexemas de los tipo de dato primitivos se debe hacer con la palabra valor_tipodedato. Mostrado en la siguiente tabla:

token	**lo que se imprime**
token_integer	valor_entero
token_string	valor_string
token_double	valor_double

- Cuando se reportan símbolos del lenguaje se deben imprimir sin ninguna alteración. Ej. ';', '$', '!', '<', '!='.
- Cuando se reportan operadores que no son símbolos sino palabras, se debe hacer igual que con las palabras reservadas, en minúscula.
- Cuando hay más de un símbolo/token esperado la lista debe reportarse **lexicográficamente ordenada** (basado en ASCII) y separada por comas.
- Cuando se reporta un id se debe hacer con la palabra 'identificador'. Ej: "<12, 15> Error sintactico: se encontro 'identificador'; se esperaba '+'."
- Cuando se esperaba fin de archivo o se encuentra fin de archivo. Lo que se debe imprimir es 'EOF'.

Ejemplos de salidas

Entrada (in01.txt)	**Salida (out01.txt)**

set a 34; puts $a; return; #error sintactico	<3,1> Error sintactico se encontro: 'return'; se esperaba: 'EOF', '[', 'for', 'gets', 'if', 'puts', 'set', 'switch', 'while'.

Entrada (in02.txt)	Salida (out02.txt)

# Declaracion de cero o mas subrutinas 	
proc sum {{a} {b} {c}}{
return [expr {$a + $b + $c}];
}
 	
# Modulo Principal 	
# Consiste de cero o mas instrucciones 	
 	
set var 10;
 	
for{set i 1}{$i <= 10}{incr i}{
puts $i;
}
	El analisis sintactico ha finalizado correctamente.