UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

INFORME DE TRABAJO PROFESIONAL

INTÉRPRETE Y LENGUAJE PARA LA CREACIÓN DE PRUEBAS AUTOMATIZADAS DE PROGRAMAS DE LÍNEA DE COMANDOS

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

PRESENTA:

GILBERTO CORONEL RUÍZ

ASESOR: LAURA SANDOVAL MONTAÑO

CIUDAD UNIVERSITARIA 2015

ÍNDICE

Introducción 3
Capítulo 1. Descripción de la empresa 5
1.1 Historia 5
1.2 Organigrama 6
Capítulo 2. Descripción de puesto de trabajo 7
2.1 Puesto 7
2.2 Responsabilidades 7
Capítulo 3. Descripción del proyecto principal 9
3.1 Objetivo 9
3.2 Requerimientos 9
3.3 Análisis y Diseño 15
Sintaxis 18
3.4 Implementación 34
Análisis léxico 34
Implementación en Perl del Lexer 35
Análisis sintáctico 39
Implementación del parser 43
Creación del árbol de sintaxis abstracto 53
Evaluador 67
Conclusiones 71
Glosario 72
Referencias 73
Anexos 74
Anexo 1: Plataformas soportadas por TimesTen 74
Anexo 2: Expresiones regulares de Perl 75
Anexo 3: Documentación del formato largo de ls 76

Introducción

El *software* es parte primordial en esta era tecnológica, se encuentra en todas partes de la vida diaria; desde los semáforos de cualquier cruce vehicular hasta las bolsas de valores que controlan las distintas economías del mundo. El planeta se ha vuelto tan dependiente del *software* que es vital poder garantizar su calidad a los clientes que lo usan directa o indirectamente. Un error de *software* puede ser devastador pudiendo llegar a costar millones de dólares por hora a una compañía o incluso pérdidas humanas. Debido a la importancia del *software* en la vida diaria, se requieren herramientas que ayuden a facilitar la creación de pruebas de *software*, para que de esta manera se puedan detectar de manera temprana errores que de no ser encontrados podrían causar pérdidas inesperadas.

Este documento describe la definición de un lenguaje de programación de pruebas para programas de línea de comandos y la implementación del intérprete que puede reconocer el lenguaje y ejecutarlo. Este lenguaje será usado principalmente para la creación de pruebas automatizadas para las utilerías de la base de datos *TimesTen*.

Capítulo 1. Descripción de la empresa.

1.1 Historia

Oracle Corporation es una de las principales empresas de tecnología a nivel mundial, fue fundada el 16 de Junio de 1977 en Santa Clara, California, Estados Unidos.
A pesar de ofrecer productos empresariales en distintas áreas tanto de software como de *hardware*, el principal producto de Oracle es su manejador de bases de datos relacional (***RDBMS***) el cual es líder de ventas a nivel mundial.

TimesTen es una base de datos relacional en memoria creada en 1996 en Palo Alto, California. Su característica principal es que los datos son almacenados en memoria principal y toda su arquitectura está basada en ese principio. Debido a esta característica, el tiempo de respuesta de la base de datos se ve incrementado drásticamente ya que la velocidad de acceso a la memoria principal es mucho mayor al compararla con la memoria secundaria.

TimesTen es usada principalmente en áreas donde se requiere gran procesamiento de datos en poco tiempo y en aplicaciones críticas que suelen ser en tiempo real. Entre sus principales clientes se encuentran compañías de telecomunicaciones, bancos y casas de bolsa.

En 2005 *TimesTen* fue adquirida por *Oracle* para hacerla parte de sus soluciones empresariales. *TimesTen* fue evolucionando para incluir características propias de *Oracle* mejorando así su compatibilidad con productos de oracle como lo es el lenguaje ***PL/SQL***, también se le añadió la capacidad de trabajar en conjunto con bases de datos Oracle como cache para acelerar las consultas.

1.2 Organigrama

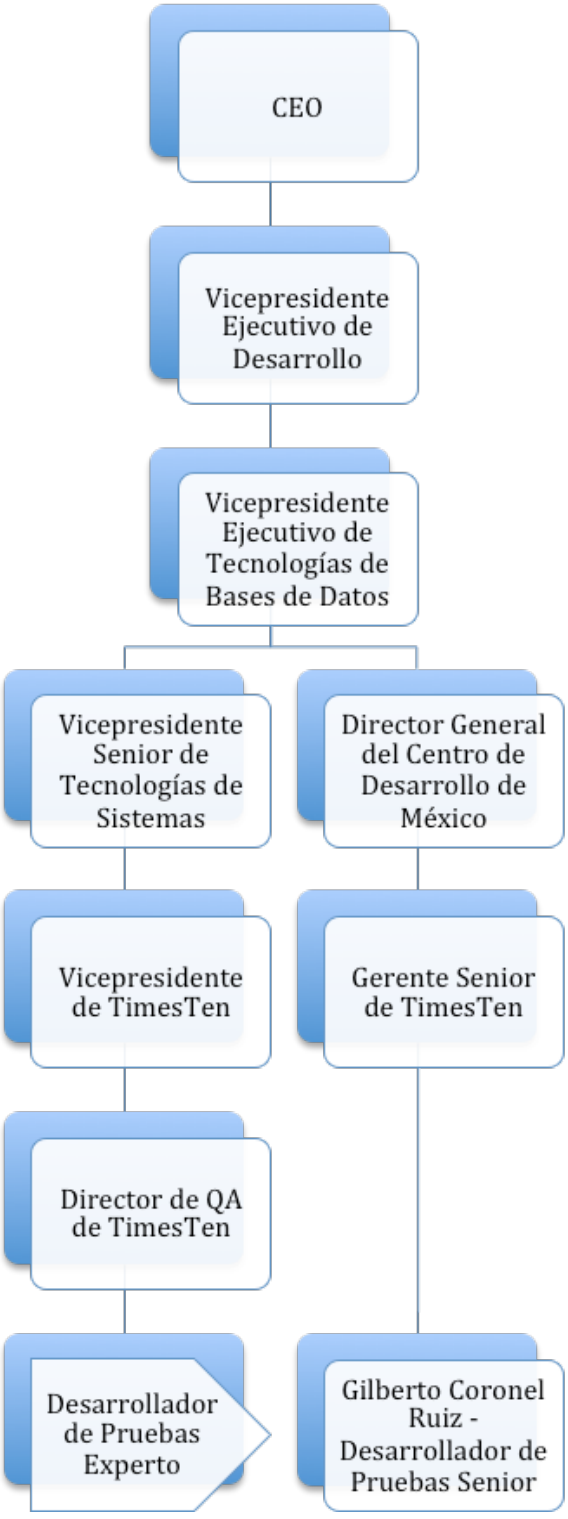

Diagrama 1. Organigrama

Capítulo 2. Descripción de puesto de trabajo

2.1 Puesto

El nombre del puesto dentro de la empresa es *Software Developer 3*, puestos de este tipo son usados principalmente para programadores y están divididos por niveles desde 1 a 5 cada vez requiriendo más años de experiencia y habilidad para subir de puesto.

Puesto	Título
Software Developer 1	Junior Member of Technical Staff.
Software Developer 2	Member of Technical Staff.
Software Developer 3	Senior Member of Technical Staff.
Software Developer 4	Principal Member of Technical Staff.
Software Developer 5	Consulting Member of Technical Staff.

Tabla 1. Puestos

El área en la que trabajo es en el área de *Quality Assurance (**QA**)* y trabajo en la sección de programación de pruebas automatizadas. Las pruebas automatizadas se utilizan principalmente para garantizar la continua calidad del producto ayudando a detectar de manera temprana las regresiones de funcionalidad introducidas por nuevos componentes.

2.2 Responsabilidades

Mis responsabilidades son el diseño, implementación y mantenimiento de pruebas automatizadas para distintos componentes de la base de datos en memoria *TimesTen*. Las pruebas son desarrolladas en varios lenguajes utilizando las distintas interfaces de programación de aplicaciones (***API)*** que *TimesTen* soporta y deben de ser capaces de correr en todas la plataformas soportadas por *TimesTen*.

Siendo miembro del equipo de pruebas he desarrollado pruebas para distintos componentes entre los que se encuentran: Structured Query Language (***SQL)***, Transaction Log API (***XLA)***, Open DataBase Connectivity (***ODBC)***, Cache, Client/Server y diversas utilerías. Las pruebas que he realizado han sido implementadas utilizando principalmente los lenguajes de programación C y Perl.

El proceso principal para la creación de pruebas es el siguiente:
Leer la especificación del programa o componente a probar, ésta deberá tener toda la información del programa, esto incluye al menos la siguiente información:

- ¿Qué hace?
- ¿Para qué?
- ¿Cómo usarlo?
- Impacto en el resto del producto
- Características
    - Descripción detallada de las diferentes opciones del programa
    - Opciones que requieren ser especificadas simultáneamente

        - Opciones que son mutuamente excluyentes
        - Parámetros mandatorios
        - Opciones no documentadas al público general
- Ejemplos
    - Básicos (para la elaboración de ***sanity tests***)
    - Complejos
    - Negativos
        - Condiciones de falla esperadas
        - Códigos de error.

Una vez analizado y entendido el componente a probar, se procede a diseñar un plan de pruebas en el cual se deben describir los siguientes campos:

- ¿En qué lenguaje se van a programar las pruebas?
- ¿Qué componentes se van a probar?
- ¿Qué componentes no se van a probar?
- Descripción detallada de los casos que se probarán
- Clasificación de las pruebas por importancia
- Pruebas de estrés
- Casos negativos

Una vez revisado y aprobado el plan de pruebas se procede con la implementación.

Además de la programación de pruebas, también soy uno de los responsables de la creación y mantenimiento de la infraestructura del arnés de pruebas así como de la creación y mantenimiento de utilerías auxiliares para la creación de pruebas.

Otra parte fundamental del puesto es el reporte de las distintas fallas del producto encontradas a la hora de la creación de las pruebas para que sean corregidas antes de que el producto salga al mercado garantizando así la satisfacción del cliente.

Capítulo 3. Descripción del proyecto principal

3.1 Objetivo

Crear un lenguaje de programación y su intérprete para la automatización de pruebas de línea de comandos que se usará principalmente para la creación de pruebas para las distintas utilerías que ofrece *TimesTen*. El nombre propuesto tanto para el intérprete como para el lenguaje es "*utiltest*" y se usará a lo largo del documento.

3.2 Requerimientos

Debido a que utiltest es un proyecto personal, los requerimientos serán enfocados en satisfacer las tareas más comunes que se me han presentado en mi trabajo cotidiano como miembro del equipo de automatización de pruebas.

Soporte multiplataforma

TimesTen es una base de datos ampliamente usada por distintas empresas, soporta varios sistemas operativos basados en UNIX así como plataformas Windows. Entre los sistemas operativos soportados se encuentran: Linux, Windows, Solaris y AIX, entre otros.

Debido a que el propósito principal de *utiltest* es crear pruebas para utilerías de *TimesTen*, *utiltest* debe ser portable entre todas las plataformas soportadas por *TimesTen*.

El anexo 1 contiene la lista completa de las plataformas soportadas por *TimesTen*

Lenguaje

Utiltest tendrá una sintaxis propia y deberá ser diseñada para facilitar las distintas tareas necesarias para probar programas de línea de comandos, entre estas tareas se encuentran:

- La ejecución y terminación de programas
- El manejo de entrada, salida y error estándar del programa
- La lectura, creación y borrado de archivos
- Comparación de archivos
- Conexión a la base de datos *TimesTen*
- Acceso completo al *SQL* soportado en *TimesTen*.

Es conveniente incluir la funcionalidad de generar casos de prueba dinámicamente de manera aleatoria al especificar una lista de opciones o rango de valores, de este modo los casos dejan de ser estáticos y pueden probar distintas combinaciones en cada ejecución expandiendo el espacio de búsqueda de errores en los programas.

Para agregar aún más funcionalidad al lenguaje de programación, *utiltest* tendrá el soporte de herramientas comunes de los lenguajes de programación de alto nivel.

La sintaxis del *utiltest* soportará lo siguiente:

- Variables
- Bloques condicionales
- Ciclos
- Funciones.

Dentro de la sintaxis se deberá incluir la posibilidad de agregar fácilmente funciones auxiliares en el intérprete para poder ser usadas dentro de las pruebas, como lo son funciones para generar números e identificadores aleatorios.

Ejecución de programas de línea de comandos

La capacidad de ejecutar programas es la base de la funcionalidad del intérprete debido a que no es posible analizar la salida de un programa sin previamente haberlo ejecutado.

El enfoque de *utiltest* es el ser capaz de analizar únicamente programas que funcionen en línea de comandos, el simular a un usuario manejando una interfaz gráfica está fuera de los alcances de esta implementación.

Acceso a información del entorno

El intérprete deberá tener algún método con el cual tenga acceso a datos importantes para la prueba, principalmente información del ambiente.
Entre la información importante a la cual se requiere acceso se encuentra la siguiente:

- Nombre del usuario actual
- Nombre de la máquina actual
- Plataforma
- Versión de la instalación de *TimesTen*
- Directorio original de la prueba
- Directorio actual
- Directorio destinado para archivos de salida de la prueba
- Nombre de bases de datos disponibles
- Credenciales necesarias para conectarse a la base de datos
- Acceso a las variables de ambiente

Capacidad de capturar la salida de un programa

Para poder analizar la salida de un programa es necesario tener acceso a ella y poder capturarla ya sea total o parcialmente para su uso posterior. *Utiltest* será capaz de leer y capturar tanto salida estándar como error estándar.

Comparación de archivos

La capacidad de comparar archivos es fundamental para *utiltest* debido a que su labor principal es utilizar la salida en texto plano de un programa. Cuando la salida esperada es fija, lo más sencillo es compararla con un "*archivo maestro*" también conocido como ***golden file***.

La comparación de archivos es una de las maneras tradicionales para probar que un programa se está comportando de manera esperada.

Existen diversas utilerías para comparar archivos, siendo la más usada la utilería de UNIX llamada ***diff***.

El Ejemplo 1 ilustra el uso común del comando diff para comparar dos archivos.

hola1.c	hola2.c

`#include <stdio.h>`
 	
`void main() {`
`  printf("Hola mundo");`
`}`
	`#include <stdio.h>`
	`main() {`
	`  printf("Hola mundo");`
	`}`

```
$ diff hola1.c hola2.c
2,3c2
<
< void main() {
---
> main() {
```

Ejemplo 1. Uso del comando diff

Redirección de salida de un programa

Para el uso del comando diff previamente mencionado, es necesario tener dos archivos a comparar. En el caso de pruebas, uno de los archivos es el *golden file* y el otro la salida del programa. Para lograr obtener el segundo archivo es necesario soportar la redirección de salida estándar (stdout) y error estándar (stderr) a un archivo.

La sintaxis tradicional para la redirección de archivos desde línea de comandos es:

$ programa > archivo.out

El comando anterior crea un nuevo archivo llamado archivo.out el cual tiene como contenido la salida estándar del programa ejecutado.

También es posible añadir información a un archivo existente usando la sintaxis “>>”

$ programa >> archivo_existente.txt

El comportamiento de “>” es únicamente redirigir la salida (stdout), los errores en pantalla se manejan en otro flujo que sigue dirigido a la consola, sin embargo, es posible redirigir individualmente cada uno de los descriptores de archivo estándar. La tabla 2 contiene los descriptores estándar.

Descriptor	Nombre Común	Descripción
0	stdin	Entrada estándar
1	stdout	Salida estándar
2	stderr	Error estándar

Tabla 2. Descriptores estándar

Ejemplos de redirección desde la consola:

$ programa < entrada.txt /* el archivo entrada.txt se pasará como entrada estándar para el programa */

$ programa 1> salida.txt /* la salida estándar del programa se redirigirá a salida.txt, el error estándar se sigue desplegando en la consola */

$ programa 2> error.txt /* el error estándar del programa se redirigirá a error.txt, la salida estándar se sigue desplegando en la consola */

$ programa > salida_error.txt 2&>1 /* tanto salida estándar como error estándar serán redirigidos al archivo salida_error.txt */

Manejo de entrada estándar del programa

La idea fundamental del intérprete es crear una interfaz para la creación rápida de pruebas basadas en el análisis de la salida de programas por línea de comandos. Sin embargo, varios de estos programas requieren de datos de entrada por parte del usuario y no siempre pueden ser especificados mediante argumentos al momento de ejecutar el programa. Por esta razón es necesario el soporte del uso entrada estándar una vez iniciado el programa.

Soporte de expresiones regulares

Una de las herramientas más potentes para el análisis y procesamiento de textos son las ***expresiones regulares***; incluir el soporte de expresiones regulares otorgará a *utiltest* el poder

necesario para realizar adecuadamente varias de sus funciones. Con ellas, el intérprete podrá usarlas en la evaluación de expresiones condicionales, esperar salida dinámica, poder extraer y asignar información específica del texto a variables, filtrar texto que no sea de interés para la prueba, etc.

Filtrado y limpieza de textos

En la mayoría de las ocasiones en las que se utiliza el método de *golden file*, es posible que se requiera filtrar y/o limpiar secciones de la salida del programa, lo más común es que estas secciones sean datos variables que, de no ser filtradas, ocasionarían diferencias al momento de comparar los archivos utilizando la utilería diff.

Estos son algunos ejemplos:

- Tiempo
    - fechas
    - tiempo de ejecución
    - ***timestamps*** (ej. 2014-10-10 09:26:50.12)

- Identificadores
    - nombres de máquina
    - nombres de usuario
    - identficadores generados aleatoriamente

- Diferencias entre plataformas
    - precisión de punto flotante
    - estilos de rutas de archivos
    - rutas y extensiones de los binarios ejecutables

Lectura de archivos

A pesar de que la idea fundamental del intérprete es analizar salida por salida estándar, también es útil extender esta funcionalidad para poder utilizarla para leer y analizar archivos estáticos. Esto nos ayuda en diversas ocasiones como por ejemplo para limpiar un archivo maestro antes de utilizarlo para un *diff* o bien simplemente verificar que los archivos creados por el programa a probar contengan la información esperada.

Consultas a TimesTen

Dado que *utiltest* será utilizado principalmente para probar las distintas utilerías de la base de datos *TimesTen*, es necesario soportar conexiones a esa base de datos para poder realizar consultas.

El intérprete deberá proveer una interfaz mediante la cual sea sencillo ejecutar consultas sobre la base de datos así como proveer una manera sencilla de capturar los resultados de las consultas para su futura utilización dentro de la prueba.

Esta funcionalidad permite en varios casos comprobar que los resultados que el programa está regresando son correctos mediante una consulta a los datos de la base o a los metadatos que *TimesTen* guarda en sus tablas y vistas de sistema.

Reporte de resultados

Una vez ejecutada cada prueba se requiere reportar los resultados, cada prueba debe de tener algún fin específico y se deben determinar las condiciones necesarias para considerarlas ya sea una prueba satisfactoria o un fallo.

Con el fin de tener un buen reporte de resultados se debe determinar previamente cuales son las salidas esperadas, los rangos de error para seguir considerándolas satisfactorias, revisar que los códigos de salida de los programas sean los requeridos. Toda esta información puede ser extraída del documento de la especificación del programa.

Utiltest debe ser capaz de generar un resumen conciso de los resultados al terminar las diversas pruebas. Este resumen debe contener la siguiente información:

- Nombres de las pruebas corridas
- Número de pruebas corridas
- Tiempos de ejecución de cada prueba y en conjunto
- Reportar individualmente si la prueba pasó o falló
- En caso de fallas dar información para diagnóstico
    - Razón de la falla
    - Ruta del archivo de salida de la prueba
    - Estatus de retorno del programa
    - Ruta de los archivos generados por la prueba que pudieran ser útiles

Es posible que las pruebas se corran repetitivamente cada noche mediante algún otro programa que programe su ejecución, esto suele hacerse para asegurarse que nuevos cambios a los programas probados no introduzcan ninguna ***regresión*** en el producto.

Capacidad de terminar procesos

Debido a que las pruebas suelen ser corridas automáticamente en tiempos programados que son usualmente una vez al día, se requiere tener la capacidad de forzar el cierre de los programas a probar en caso de exceder un límite de tiempo. Es común que, bajo ciertas

condiciones, los programas a probar se queden esperando indefinidamente por algo, ya sea una entrada por parte del usuario, o algún recurso de la red, entre otros.

Un caso como los mencionados resultaría en que la prueba se "*cuelgue*" y que el conjunto subsecuente de pruebas no se ejecute en esa máquina. Por esta razón, se requiere añadir la capacidad de establecer tiempos máximos para la ejecución de los programas en las pruebas que puedan generar los problemas mencionados anteriormente.

3.3 Análisis y Diseño

Una vez concluidos los requerimientos el siguiente paso es analizarlos y diseñar el lenguaje en base a éstos, todos los requerimientos deben de ser considerados.

La idea general es tener un archivo de entrada con la sintaxis soportada por el intérprete y, una vez ejecutada la prueba, se espera un archivo de salida con el resumen de los resultados.
El diagrama 2 ilustra la idea general del uso de utiltest.

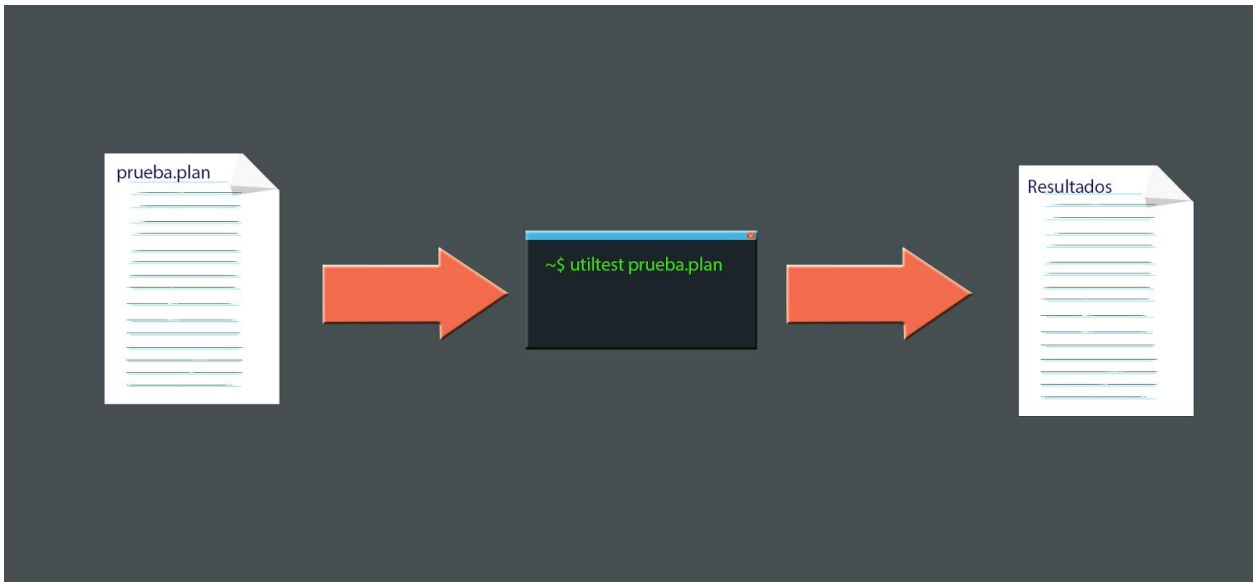

Diagrama 2. Diagrama de utiltest

El soporte multiplataforma es un requerimiento muy importante para *utiltest* ya que puede haber errores en los programas a probar que sean específicos de alguna plataforma, esto también complica la implementación del intérprete porque dependiendo de la plataforma es posible que secciones del intérprete tengan que ser programadas de manera distinta.

De implementar el intérprete en algún lenguaje fuertemente atado a la plataforma o arquitectura de la máquina como lo es C o algún otro lenguaje compilado, podría ocasionar dificultades a la hora de la implementación teniendo que agregar varios bloques condicionales para ejecutar

ciertos pedazos de código de acuerdo a la plataforma actual. Esto también hace que el esfuerzo de la implementación y del mantenimiento del intérprete se eleve. También es posible que el intérprete no funcione de la manera esperada en algunas plataformas en el caso en el que no se consideren adecuadamente estas diferencias entre las plataformas a la hora de la implementación.

En los siguientes códigos en C (1 y 2) se muestran algunos ejemplos de estas dificultades:

El uso de *APIs* específicas de Windows o UNIX

```
void crear_hilo() {
#ifdef WIN32
   // usar CreateThread para windows
#else
   // usar pthread_create para UNIX
#endif
   // Código común entre plataformas
}
```

Código en C 1. APIs específicas de plataformas

Distintos tamaño de tipos de datos entre plataformas, por ejemplo si quisiéramos un entero de 8 bytes tendríamos que hacer lo siguiente:

```
#ifdef WIN32
  typedef signed __int64 int8_bytes;
#else
  typedef signed long int8_bytes;
#endif
```

Código en C 2. Tamaño de tipos de datos

Adicionalmente, esto únicamente funcionaría en plataformas de 64-bit, en plataformas de 32-bit los tipos *long* son de 4 bytes, por lo que se tendría que utilizar *signed long long*.

Es por esto que nos conviene usar un lenguaje portable entre plataformas, por lo que usaremos un lenguaje interpretado. Los lenguajes interpretados suelen ser portables entre plataformas ya que el intérprete del lenguaje en cada plataforma se encarga de lidiar con éstas diferencias, de esta manera nos permite escribir código portable sin tomar en cuenta muy frecuentemente el comportamiento de acuerdo a la plataforma.

El problema de utilizar lenguajes interpretados es que suelen ser ligeramente más lentos que los lenguajes compilados a código máquina nativo. Esto no es problema para el caso de *utiltest*, el desempeño no es crítico para el intérprete ya que el caso común es que el programa a probar es quien genera un uso más intensivo de procesador, lo más intensivo que hará el intérprete será la lectura y manipulación del texto de salida del programa a probar.

Para decidir el lenguaje de programación a utilizar es necesario ver el resto de los requerimientos y así determinar si hay algún lenguaje que otorgue fácilmente funcionalidad que se encuentre en los requerimientos.

Al analizar los requerimientos se pueden ver dos características principales:

- *utiltest* será muy ligado a la consola
    - Esto se puede ver en requerimientos como la capacidad de ejecutar y terminar programas desde la consola, el capturar, leer y redirigir las diferentes salidas del programa así como el uso de utilerías para la comparación de archivos como el comando diff.

- Debe de ser potente para analizar texto
    - Para pruebas robustas no es suficiente comparar la salida contra una salida estática, el intérprete deberá ser capaz de capturar secciones de la salida sin tener mucha idea de los datos como tal, así como ignorar información que no sea de utilidad. Para esto el soporte de expresiones regulares es parte primordial para el análisis y filtrado de los textos.
    - La salida de un programa puede ser enorme por lo que para esto sí se necesita una manera eficiente de manejarla.

Existen varios lenguajes multiplataforma que cuentan con expresiones regulares que podrían ser usados para la creación de *utiltest* (ej. Java, Python, Perl, etc.), Entre ellos, Perl destaca por tener una sintaxis sencilla para el manejo de cadenas y el uso de su biblioteca de ***IO*** (que *utiltest* usará apliamente)***,*** sin mencionar que es uno de los lenguajes con mejor soporte para expresiones regulares. Es por esto que Perl se usará para la implementación de *utiltest*.

Perl fue introducido en el año de 1987 por el programador, autor y lingüista Larry Wall. El principal motivo de la creación de Perl fue porque el autor estaba insatisfecho con la funcionalidad que otros lenguajes de programación le ofrecían por lo que diseñó un lenguaje combinando la mejor funcionalidad de lenguajes como C, awk y Bourne shell.

Perl se ajusta perfectamente a las dos características principales mencionadas anteriormente, ya que adopta rasgos de otros lenguajes como C, *shell scripting*, *awk* y *sed*. Al ser un lenguaje de scripting con características de lenguajes de comandos puede ser utilizado de manera similar a trabajar con la línea de comandos, permitiendo fácilmente ejecutar programas externos e interactuar con ellos. Perl puede ejecutar programas y tener acceso a sus distintos *streams* (*stdin*, *stdout* y *stderr*). Las características principales de Perl son sus poderosas herramientas para el procesamiento de textos sin imponer límites arbitrarios de tamaño de los textos. Entre estas herramientas se encuentra el soporte para expresiones regulares estilo Perl, que es tan poderoso que se ha convertido en uno de los principales estándares para expresiones regulares extendiendo la funcionalidad de los estándares de ***POSIX***. Debido a su gran poder expresivo, varias utilerías y lenguajes de programación han adoptado parcialmente las expresiones regulares estilo Perl, entre estos lenguajes se encuentran Java, JavaScript, Python, Ruby y Microsoft's .NET Framework.

En el anexo 2 se encuentran tablas con información sobre las expresiones regulares que soporta Perl.

Sintaxis

Lo primero a diseñar es la sintaxis del lenguaje que el intérprete soportará. Para esto es necesario considerar nuevamente los requerimientos.

Variables

La sintaxis propuesta para la declaración de variables así como para la asignación de valores a variables nuevas o existentes será de la forma:

```
<nombre> <=  <valor>
```

```
horas <= 24
nombre   <= "Gilberto"
apellidos <= "Coronel Ruiz"
PI    <= 3.1416
```

Código en utiltest 1. Uso de variables

Para acceder al valor de la variable se usará la siguiente sintaxis:

```
:<nombre>:
```

```
nombre_completo <= "El nombre es: :nombre: :apellidos:"
```

Código en utiltest 2. Interpolación de variables

Es importante el poder tener acceso a la información del entorno donde se correrán las pruebas, esto es necesario para que la prueba pueda saber dónde puede poner archivos temporales o para tomar decisiones de acuerdo a diferencias en comportamiento de los programas de acuerdo a la plataforma donde se ejecutan, nombres de bases de datos, etc.

La manera que *utiltest* dará acceso a este tipo de información es aprovechando el soporte de variables anteriormente mencionado para la creación de variables predefinidas al momento de ejecución de la prueba. Para distinguir las variables predefinidas de las variables de usuarios la convención será que las variables predefinidas serán generadas en letras mayúsculas. En la Tabla 3 se encuentra una lista con las variables predefinidas de *utiltest*.

Variable	Descripción
TESTNAME	Nombre de la prueba actual
BINDIR	Directorio a utilizar para archivos creados por la prueba
TESTDIR	Directorio donde la prueba reside
PLATFORM	Plataforma de la máquina actual
INSTALLDIR	Carpeta raíz de la instalación de *TimesTen*.
DSN	Nombre de la base de datos a utilizar.

CONNSTR	Cadena de conexión a la base de datos.
VERSION	Versión de la instalación de *TimesTen*
DEVNULL	Archivo para redirigir salida descartada.
RETVAL	Valor retornado por la última ejecución de un programa
EOL	Caracter de nueva línea de la plataforma actual
NONE	Cadena vacía

Tabla 3. Variables predefinidas

Para poder acceder al valor de éstas variables predefinidas se utiliza el mismo método que el mencionado anteriormente para las variables de usuario

:<VARIABLE>:

```
salida <= ":BINDIR:/:TESTNAME:.out"
```

Código en utiltest 3. Uso de variables predefinidas

Funciones predefinidas

Utiltest contará con el soporte de funciones predefinidas para su utilización dentro de los scripts de las pruebas. Se encargarán de proveer un mecanismo simple para extender las capacidades de *utiltest* mediante una interfaz sencilla.

Función	Descripción
`&get_env(variable)`	Retorna el valor de una variable de ambiente.
`&rnd(intervalo)`	Retorna un número entero aleatorio dentro del intervalo.
`&pick_one(lista)`	Retorna un elemento de la lista aleatoriamente.
`&rnd_str(tamaño)`	Retorna una cadena aleatoria de máximo <tamaño> caracteres.
`&sleep(<tiempo>)`	Duerme y retorna en <tiempo> segundos.

Tabla 4. Funciones predefinidas.

Ejecución de programas

La sintaxis que *utiltest* usará para ejecutar los programas será la siguiente:

```
run <programa o cadena a ejecutar>
```

En caso de que la cadena a ejecutar sea una sola palabra se asumirá que es el nombre de un ejecutable y se verificará que esté en alguno de los directorios de la variable ***PATH*** o en el directorio actual.

```
run ls
```

Código en utiltest 4. Uso de run

En caso de que la cadena sea más de una palabra, se ejecutará la cadena completa sin buscar la ruta del programa, esto nos permite agregar argumentos al programa así como poder utilizar

redirecciones del *shell* o *pipes* tal como si se estuvieran ejecutando en la consola.

```
run “ls -l | wc -l”
```
Código en utiltest 5. Uso de run con una cadena

Con la cláusula *run* se podrá ejecutar cualquier programa, sin embargo, se requiere un manejo más avanzado de toda la ejecución del programa por lo que debe de ser expandido en varias subcláusulas para incluir todos los requerimientos posibles en su sintaxis a la hora de ejecutar programas.

Lo primero a añadir es el soporte para argumentos sin perder la capacidad de buscar la ruta del programa, la sintaxis es la siguiente:

```
run <programa> args “<lista de argumentos>”
```

```
run grep args “--version”
```
Código en utiltest 6. Uso de run con args

Ahora es posible ejecutar programas con los argumentos adecuados, verificando que la utilería “*grep*” esté disponible dentro del *path* o el directorio actual, sin embargo, el ejemplo anterior no hace mucho por sí solo; revisando una vez más los requerimientos se puede observar que otro de los requerimientos es la capacidad de generar casos de prueba dinámicamente con una serie de reglas, por lo que *args* es un buen lugar para generar argumentos aleatorios

La sintaxis propuesta es la siguiente:

```
run <programa> args “[<opcion1>|<opcion2>|...|<opcionN>]”
```

De esta manera en cada ejecución de la prueba se elegirá aleatoriamente alguna de las opciones mencionadas.

Ejecutar grep para que regrese la versión del programa ya sea con -V o --version

```
run grep args “[-V|--version]”
```
ó
```
run grep args “-[V|-version]”
```
Código en utiltest 7. Argumentos aleatorios

El código en utiltest 9 muestra el comando *grep* eligiendo alguna opción aleatoriamente:

```
run grep args ‘[-A 1|-B 1|-n|-b|-E] -e [“pat1”|“pat2”] archivo.txt’
```
Código en utiltest 8

Este comando en cada ejecución de la prueba seleccionará aleatoriamente un valor de cada uno de los conjuntos delimitados por corchetes.

Posibles combinaciones:

`grep -A 1 -e pat1 archivo.txt`	`grep -A 1 -e pat2 archivo.txt`
`grep -B 1 -e pat1 archivo.txt`	`grep -B 1 -e pat2 archivo.txt`
`grep -n -e pat1 archivo.txt`	`grep -n -e pat2 archivo.txt`
`grep -b -e pat1 archivo.txt`	`grep -b -e pat2 archivo.txt`
`grep -E -e pat1 archivo.txt`	`grep -E -e pat2 archivo.txt`

Tabla 5. Posibles casos generados

Podemos agregar más funcionalidad permitiendo la anidación de opciones entre corchetes.

```
run grep args '[-A [1|10]|-B [1|100 -n]] -e ["pat1"|["pat2"|"pat3"]] archivo.txt'
```

Código en utiltest 9. Anidación de argumentos

-A n lista n líneas después de la ocurrencia.
-B n lista n líneas antes de la ocurrencia.
-e especifica el patrón a buscar.
archivo.txt es el archivo a leer.

La línea anterior ejecuta el comando grep lista ya sea 1 o 10 líneas después de la ocurrencia o 1 o 100 líneas antes de la ocurrencia, en el caso donde lista 100 líneas antes también listará el número de línea, el patrón a usar será el patrón 1 el 50% de las veces que corra el programa, el patron 2 y el patron 3 25% de las veces cada uno.

Los siguientes diagramas (Diagrama 3 y Diagrama 4) muestran la expansión de las distintas opciones que pueden ser elegidas cuando se ejecuta el programa del Código en utiltest 9. Los diagramas ilustran los diferentes caminos posibles que utiltest puede recorrer al momento de construir los argumentos aleatoriamente.

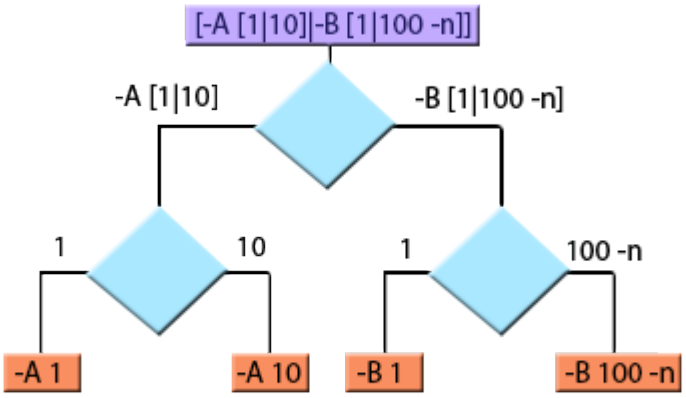

Diagrama 3

La segunda parte se puede expandir de la siguiente forma:

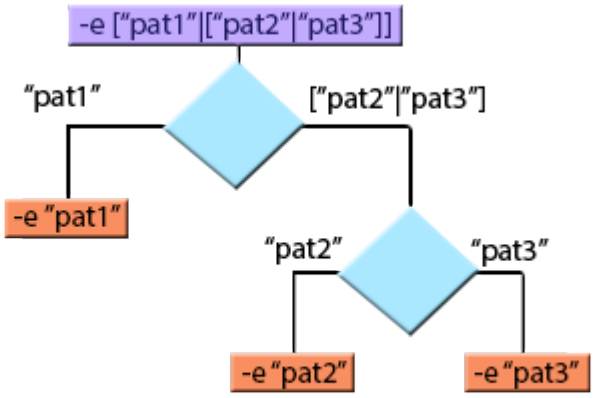

Diagrama 4

grep -A 1 -e “pat1”	grep -A 10 -e “pat1”
grep -A 1 -e “pat2”	grep -A 10 -e “pat2”
grep -A 1 -e “pat3”	grep -A 10 -e “pat3”
grep -B 1 -e “pat1”	grep -B 100 -n -e “pat1”
grep -B 1 -e “pat2”	grep -B 100 -n -e “pat2”
grep -B 1 -e “pat3”	grep -B 100 -n -e “pat3”

Tabla 6. Posibles casos generados

Con esto *utiltest* tendrá buen soporte para casos de prueba dinámicos corrida tras corrida. Ahora es posible generar argumentos aleatorios pero todavía es necesario agregar soporte para analizar la salida. Este soporte se puede incorporar en la misma cláusula *run* así como fue añadida la subcláusula *args*, la sintaxis propuesta es la siguiente:

```
run <programa> expect [not] “<salida esperada>”
```

La idea es poner el texto esperado en la subcláusula *expect*, la salida será analizada para buscar que parte del texto coincida con el texto esperado, las comparaciones de la salida esperada con la salida real se harán línea por línea, de no ser encontrada ninguna coincidencia la prueba reportará una falla y terminará su ejecución.

Expect puede ser especificado varias veces, todas las cadenas esperadas deben ser parte de la salida del programa para que la prueba se considere exitosa. De manera contraria se reportará error.

Asumiendo que el directorio actual contiene 3 archivos, archivo1.txt, archivo2.txt y archivo3.txt

```
run ls expect “archivo1.txt archivo2.txt archivo3.txt”
```

Código en utiltest 10 Uso de expect

El comando anterior ejecuta el programa ls y espera en su salida el texto “`archivo1.txt archivo2.txt archivo3.txt`”

Con esto tenemos la funcionalidad principal cubierta, ejecutar el programa y esperar que la salida coincida con un texto esperado. Ahora es posible combinar ambos subcomandos de *run* para extender el ejemplo anterior a uno un poco más elaborado.

```
run ls args “-[l|la]” expect “-rw-r--r--  1 gicorone  staff  0 Jan 10 17:32 archivo1.txt”
```

Código en utiltest 11

Al ver el ejemplo anterior se pueden observar algunas cosas interesantes.

- El comando, a pesar de ser sencillo, comienza a perder legibilidad al estar en una sola línea, por lo que es conveniente permitir que pueda ser dividido en varias líneas.
- El texto contenido en *expect* solo incluye a archivo1.txt, esto es porque las comparaciones entre la salida esperada y la real se harán línea por línea. Para solucionar este problema el subcomando *expect* aceptará varias cadenas a esperar y podrá ser especificado múltiples veces.
- Existen datos variables. El usuario y el grupo varían de acuerdo a la máquina donde se ejecute. La fecha es otro dato variable, los valores son distintos de acuerdo a la hora que se ejecute. Recordemos que la coincidencia no tiene que ser de la línea completa, podemos esperar únicamente una sección de la salida, por ejemplo el nombre del archivo, así que por el momento esperaremos únicamente que estén los 3 archivos.

Después de los cambios anteriores el comando queda de la siguiente manera:

```
1:  run ls
2:    args "-[l|la]"
3:    expect "archivo1.txt"
4:    expect "archivo2.txt"
5:    expect "archivo3.txt"
```

Código en utiltest 12. Comandos multi-línea

En el ejemplo únicamente se encuentran los nombres de los archivos esperados, sin embargo, es probable que también se quiera verificar otra característica del archivo como por ejemplo sus permisos; con la sintaxis actual se puede escribir algo así, esperando que los 3 archivos tuvieran los mismos permisos 644.

```
1:  run ls
2:    args "-[l|la]"
3:    expect "-rw-r--r--"
4:    expect "archivo1.txt"
5:    expect "-rw-r--r--"
6:    expect "archivo2.txt"
7:    expect "-rw-r--r--"
8:    expect "archivo3.txt"
```

Código en utiltest 13

El problema con el código en utiltest 14 es que nada nos garantiza que los primeros permisos esperados corresponden con el primer archivo y así sucesivamente, cada *expect* busca una coincidencia en toda la salida por lo que la siguiente salida pasaría satisfactoriamente con el ejemplo anterior a pesar de que no todos los archivos tienen los mismos permisos:

```
total 16
-rw-r--r--  1 gicorone  staff   0 Jan 10 17:32 archivo1.txt
-rw-rw-r--  1 gicorone  staff  10 Jan 13 17:20 archivo2.txt
-rw-rw-rw-  1 gicorone  staff  20 Jan 13 17:19 archivo3.txt
```

Salida 1. Salida del comando ls -l

Los 3 permisos esperados coincidirían con los permisos del archivo archivo1.txt dado que varios *expect* pueden coincidir en la misma línea de la salida.

Ahora es momento de incluir el soporte para expresiones regulares en la salida esperada, esto nos permite definir cómo se debería de ver una línea sin necesidad de utilizar múltiples *expect* y así también podremos garantizar que los permisos del ejemplo anterior correspondan al archivo esperado.

```
1:  run ls
2:    args "-[l|la]"
3:    expect "-rw-r--r--.*archivo1\.txt"
4:           "-rw-r--r--.*archivo2\.txt"
5:           "-rw-r--r--.*archivo3\.txt"
```

Código en utiltest 14

Con esto ya es posible asociar los permisos de cada archivo a su respectivo archivo, ahora cada *expect* espera los permisos del archivo y el archivo en la misma línea ignorando la información que aparece en medio, esto también nos beneficia en que podemos ignorar la información variable y únicamente concentrarnos en la información que nos sirve.

Ahora que incluimos el soporte de expresiones regulares, ¿Por qué conformarnos con ignorar la información variable?, podríamos verificar que la información que es variable vaya de acuerdo al formato de la salida de ls -l.

```
1:  run ls
2:   args
3:     "-[l|la]"
4:   expect
5:     "-rw-r--r--  1 \w+  \w+ +\d+ \w{3} \d{1,2} \d\d:\d\d archivo1\.txt"
6:     "-rw-r--r--  1 \w+  \w+ +\d+ \w{3} \d{1,2} \d\d:\d\d archivo2\.txt"
7:     "-rw-r--r--  1 \w+  \w+ +\d+ \w{3} \d{1,2} \d\d:\d\d archivo3\.txt"
```

Código en utiltest 15

Viendo el ejemplo anterior se puede observar que las tres expresiones regulares son muy parecidas pero son necesarias para determinar que en realidad existen 3 archivos. Hay ocasiones en las que se espera exactamente un número N de ocurrencias y no es suficiente con haber encontrado el texto, por esta razón la cláusula *expect* tendrá una parte opcional para poder especificar el número de ocurrencias esperadas, de no ser encontradas exactamente el número de ocurrencias esperadas la prueba fallará a pesar de haber encontrado alguna ocurrencia.

La sintaxis propuesta es la siguiente:

```
run <programa> expect [<entero> times] “<expresión regular>”
```

Con esta nueva sintaxis opcional el ejemplo anterior se puede simplificar a:

```
1:  run ls
2:    args
3:      "-[l|la]"
4:    expect
5:      3 times "-rw-r--r--  1 \w+  \w+ +\d+ \w{3} \d{1,2} \d\d:\d\d archivo\d\.txt"
```

Código en utiltest 16

De esta manera la prueba espera que al ejecutar “`ls -l`” o “`ls -la`” en el directorio actual encuentre en la salida 3 líneas con permisos 644 y que incluyan archivo+digito+.txt como nombre. El problema de esto es que archivo0.txt, archivo4.txt,…, archivo9.txt son ocurrencias válidas, para restringir aún más lo esperado puede ser reescrito de la siguiente forma:

```
1:  run ls
2:    args
3:      "[-l|la]"
4:    expect
5:      3 times "-rw-r--r--  1 \w+  \w+ +\d+ \w{3} \d{1,2} \d\d:\d\d archivo[123]\.txt"
```

Código en utiltest 17

Con la nueva sentencia se prueba que haya 3 ocurrencias de la expresión regular y que existan los 3 archivos debido a que los nombres de archivo no se pueden repetir en un mismo directorio.

Una mejor prueba que se podría implementar es el utilizar expresiones regulares más elaboradas para probar que la estructura del comando *ls* satisfaga lo descrito en la documentación del programa, de esta manera tomaremos provecho del vasto soporte de expresiones regulares que incluye Perl.

De acuerdo a la documentación del formato largo de *ls* descrito en el anexo 3.

```
1:  run "ls -l"
2:    expect
3:      "[dsl-][rwx-]{8}[rwxtTS-]\s+\d+ \w+ \w+\s+\d+
(Jan|Feb|Ma[ry]|A(pr|ug)|Ju[nl]|Sep|Oct|Nov|Dec)\s+\d{1,2}\s+(\d\d:\d\d|\d{4})\s+[\w\
.]+(\s-> [/.\w\-]+)?"
```

Código en utiltest 18

Con esto podemos verificar que la salida de *ls* es de acuerdo a su especificación para cualquier directorio que contenga cualquier tipo de archivos.

Es posible que se requiera capturar parte de la salida de un programa para poder utilizarla más adelante en la prueba, esto suele ser útil, por ejemplo, cuando el programa a probar genera identificadores dinámicos o cuando para probarlo se requiere información que alguna otra utilería otorga.

Esta funcionalidad será agregada también como sub-cláusula de *run* y la sintaxis es la siguiente:

```
run <programa> capture <lista de variables> <= '<expresión regular>'
```

La expresión regular deberá contener la sección a capturar delimitada entre paréntesis, tal como se captura en Perl información de una expresión regular.

La lista de variables será asignada en el mismo orden de las ocurrencias de los paréntesis, si la variable no existe, será creada al momento en que se capture información. Si la variable existe, la nueva información capturada sobrescribirá la información previa de la variable.

La lista de variables debe de estar en el mismo orden de los pares de paréntesis en la expresión regular y debe ser la misma cantidad, de lo contrario se detendrá la ejecución de la prueba y ésta fallará.

Capturar el nombre del usuario actual.

```
1:  run whoami
2:    capture nombre <= '(\w+)'
```

Código en utiltest 19

En el ejemplo anterior se ejecuta la utilería *whoami* y se captura la salida en la variable nombre, la variable nombre puede ser usada más adelante en la prueba.

Otra opción para capturar el usuario actual y también capturar información de sus grupos es el comando “id”, éste despliega los id y los nombres de todos los grupos a los que pertenece un usuario, así como el id y nombre del usuario.

```
1:  run id
2:    capture nombre <= 'uid=\d+\((\w+\))'
3:    capture grupo  <= 'gid=\d+\((\w+\))'
```

Que puede ser reescrito como:

```
1:  run id
2:    capture nombre <= 'uid=\d+\((\w+\))'
3:             grupo  <= 'gid=\d+\((\w+\))'
```

O su equivalente en una sola línea:

```
1:  run id
2:    capture nombre, grupo <= 'uid=\d+\((\w+\)) gid=\d+\((w+)\)'
```

Código en utiltest 20

La expresión regular de *capture* es evaluada contra la salida completa del texto, el metacaracter punto "." de la expresión regular no incluye nuevas líneas pero éstas pueden ser especificadas explícitamente con "\n".

Con esta nueva habilidad del programa del Código en utiltest 17 se puede extender a lo siguiente:

```
1:  run id
2:    capture nombre, grupo <= 'uid=\d+\((\w+\)) gid=\d+\((w+)\)';
3:
4:  run ls
5:    args "-[l|la]"
6:    expect
7:      3 times "-rw-r--r--  1 :usuario:  :grupo: +\d+ \w{3} \d{1,2} \d\d:\d\d
archivo[123]\.txt"
```

Código en utiltest 21

Otro de los requerimientos que será incluido en *run* es la redirección de archivos, esto puede ser útil para guardar la salida para futuras referencias en caso de fallos o para pruebas de master file. Al ser redirigida la salida de un programa a un archivo puede ser comparada contra otro archivo maestro con la salida esperada.

```
run <programa> outfile <ruta de archivo>
```

```
1:  run find
2:    args
3:      '. -name "*.c"'
4:    outfile
5:      "archivos_c.out"
```

Código en utiltest 22

Con el comando anterior se genera el archivo "`archivos_c.out`" con la salida del comando ejecutado el cual puede ser utilizado más adelante en la prueba en caso de que se requiera comparar con un master file.

Un problema que se presenta comúnmente al esperar un texto específico, es que el programa a probar tenga secciones variables en su salida, lo que ocasiona que no pueda ser usada directamente para compararla contra un *master file*. Para solucionar este problema, el comando *run* contendrá la opción de filtrar secciones de la salida en tiempo de ejecución para así generar un archivo que sea determinístico.

La sintaxis propuesta es la siguiente:

```
run <programa> filter "<ocurrencia>" => "<filtro>"
```

```
$ ls -l
total 16
-rw-r--r--  1 gicorone  staff   0 Jan 10 17:32 archivo1.txt
-rw-rw-r--  1 gicorone  staff  10 Jan 13 17:20 archivo2.txt
-rw-rw-rw-  1 gicorone  staff  20 Jan 13 17:19 archivo3.txt
```

Salida 2. Salida del comando ls -l

```
1:  run id
2:    capture nombre, grupo <= 'uid=\d+\((\w+)\) gid=\d+\((w+)\)';
3:
4:  run ls
5:    args "-[l|la]"
6:    filter ':nombre:' => 'USUARIO'
7:    filter ':grupo:' => 'GRUPO'
8:    filter '\w{3} \d{1,2} \d\d:\d\d' => 'FECHA'
9:    filter 'total \d+' => 'BLOQUES'
10:   outfile "salida.out"
```

Código en utiltest 23

El comando anterior generaría el siguiente archivo:

```
$ cat salida.out
BLOQUES
-rw-r--r--  1 USUARIO  GRUPO    0 FECHA archivo1.txt
-rw-r--r--  1 USUARIO  GRUPO   10 FECHA archivo2.txt
-rw-r--rw-  1 USUARIO  GRUPO   20 FECHA archivo3.txt
```

Salida 3. Salida del comando cat salida.out

Este archivo generado ya podría usarse como archivo maestro en la prueba.

A pesar de solucionar el problema es posible que en realidad haya líneas completas en la salida que no sean relevantes para la prueba, o que ocurran indefinidamente solo en ciertas ocasiones, como por ejemplo las ***warnings*** de un programa.
Para estos casos es mejor expandir el subcomando *filter* para poder ignorar ese tipo de líneas y excluirlas de la salida final.

```
run <programa> filter “<ocurrencia>” => “<filtro>” | skip
```

De esta manera podríamos quitar la salida del total de bloques en el ejemplo:

```
1:  run ls
2:    args "-[l|la]"
3:    filter
4:      ':nombre:' => 'USUARIO'
5:      ':grupo:' => 'GRUPO'
6:      '\w{3} \d{1,2} \d\d:\d\d' => 'FECHA'
7:    filter
8:      'total \d+' => skip
9:    outfile "salida.out"
-rw-r--r--  1 USUARIO  GRUPO    0 FECHA archivo1.txt
-rw-r--r--  1 USUARIO  GRUPO   10 FECHA archivo2.txt
-rw-r--rw-  1 USUARIO  GRUPO   20 FECHA archivo3.txt
```

Código en utiltest 24

Así como se requiere en ocasiones redirigir la salida del archivo, es también útil poder enviar al programa información a través de la entrada estándar (*stdin*).

La sintaxis será:

```
run <programa> input “entrada1”, “entrada2”, ..., “entradaN”
```

```
1:  run grep
2:    args "tiene"
3:    input
4:      'Esta linea tiene que estar en la salida',
5:      'Esta no'
6:    expect
7:          '^Esta.*salida$'
8:      not 'Esta no
```

Código en utiltest 25

La funcionalidad del sub comando input puede ser extendida con las capacidades de lectura del intérprete para manejar de manera dinámica tanto la entrada como la salida del programa.

Por último, es necesario incluir un tiempo límite de ejecución de programas para evitar programas que tarden demasiado. Esto puede ser generado por programas que generen ciclos infinitos o esperas infinitas por algún evento como la entrada de datos por parte del usuario.

La sintaxis será:

```
run <programa> kill in <entero> <unidad>
```

Donde
<entero> es cualquier número entero positivo.
<unidad> es cualquier palabra entre “sec”, “secs”, “seconds” o “second”; “min”, “mins”, “minute” o “minutes”; “hour” o “hours”.
Este subcomando puede ser especificado únicamente una vez por cada comando *run*.

Asúmase un sistema basado en UNIX, se quiere capturar el número de archivos del directorio ***home*** del usuario y se espera que la operación tarde máximo 10 segundos.

```
1: run "find ~ 2>/dev/null | wc -l"
2:   capture num_archivos <= '(\d+)'
3:   kill in 10 seconds
```

Código en utiltest 26

Es necesario en ocasiones terminar el programa con determinada señal para ciertos programas, ya sea para forzar su terminación (SIGKILL) así como para simular señales de teclado, por ejemplo, ctrl+c (SIGINT) podría ser capturada y manejada por el programa generando diferente salida.

La sintaxis para

```
run <programa> kill <señal> in <entero> <unidad>
```

Las señales únicamente son válidas para sistemas unix, en caso de windows se ignorará la señal y terminará el proceso como si se hiciera llamada a *TerminateProcess* .

```
1: run "find ~ 2>/dev/null | wc -l"
2:  capture num_archivos <= '(\d+)';
3:  kill SIGINT in 1 minute
```

Código en utiltest 27

Capacidad de leer y analizar archivos existentes

Es común encontrar casos en los que el programa a ejecutar no despliega su salida en *stdout*, si no que la redirige a algún archivo. *Utiltest* debe de ser capaz de analizar la salida aún en estos casos por lo que es necesario incluir funcionalidad para leer archivos y analizarlos de manera similar que en *run*.
Resulta fácil portar un subconjunto de la funcionalidad de *run* con el propósito de leer y analizar archivos existentes, como lo es la capacidad de esperar salida que concuerde con expresiones regulares o capturarla.

La diferencia principal con respecto a run es la necesidad de aceptar un archivo como entrada.

La sintaxis propuesta es la siguiente:

```
read_file “<ruta del archivo>”
      [<subcomando expect>]
     [<subcomando capture>]
```

La ruta del archivo deberá de ser cualquier ruta válida. *Read_file* deberá de ser capaz de manejar tanto rutas absolutas como relativas al directorio actual, así como los distintos tipos de rutas de sistemas Windows y sistemas UNIX.

Tanto *expect* como *capture* pueden ser especificadas más de una vez.

Capacidad de realizar consultas a una base de datos

La principal motivación para la creación de *utiltest* es para poder hacer pruebas de manera sencilla de la base de datos *TimesTen*, es por esto que es primordial incluir la funcionalidad de realizar consultas sobre bases de datos *TimesTen*. Es común que se requiera información que se puede encontrar dentro de tablas de la base de datos.

La sintaxis propuesta es la siguiente:

```
run_sql “<sentencia SQL>”[,“<sentencia SQL>”, ...]
                [<subcomando expect>]
               [<subcomando capture>]
```

En caso de tener más de una sentencia SQL, éstas se ejecutarán de manera secuencial en el mismo orden que fueron especificadas, sin embargo tanto *capture* como *expect* buscarán toda la salida generada por el conjunto de sentencias SQL.

Ejemplo de creación de usuarios para la prueba:

```
1:  run_sql
2:      'create user adminuser identified by adminpwd'
3:      'grant admin to adminuser'
4:      'create user regularuser identified by regularpwd'
5:      'grant create session to regularuser'
6:    expect 2 times 'user created'
```

Código en utiltest 28

Capacidad de comparar archivos

Otra funcionalidad que es necesario añadir a *utiltest* es la capacidad de comparar archivos, muchas pruebas son realizadas ejecutando algún programa y esperando una salida que concuerde con un archivo maestro. Es importante que ambos archivos a comparar sean exactamente iguales o la comparación se reportará como falla.

La sintaxis es la siguiente:

diff “<archivo1>” “<archivo2>”

Las rutas de archivo pueden ser ya sea absolutas o relativas al directorio actual donde se está ejecutando la prueba.

Ejemplos:
Verificar que la ayuda del comando grep se muestre siempre que no se especifique el archivo a utilizar sin importar que sean opciones válidas. Verificar que la ayuda se muestre completa.

```
$ cat grep_ayuda.mas
usage: grep [-abcDEFGHhIiJLlmnOoPqRSsUVvwxZ] [-A num] [-B num] [-C[num]]
      [-e pattern] [-f file] [--binary-files=value] [--color=when]
      [--context[=num]] [--directories=action] [--label] [--line-buffered]
      [--null] [pattern] [file ...]
```

Salida 4. Salida del comando cat grep_ayuda.mas

```
1: run grep
2:   args "-[a|b|c|D|E|F|G|H|h|I|i|J|L|l|m|n|O|o|P|q|R|S|s|U|V|v|w|x|Z]"
3:   outfile ":BINDIR:/grep_salida.out";
4:
5: diff ":BINDIR:/grep_salida.out" ":TESTDIR:/grep_ayuda.mas"
```

Código en utiltest 29

Comprobar que el comando *tee* escribe su salida como al archivo especificado como a la salida estándar.

```
1: run "ls -l | tee salida1.out"
2:   outfile ":BINDIR:/salida2.out";
3:
4: diff ":BINDIR:/salida2.out" "salida1.out"
```

Código en utiltest 30

La principal limitante de este comando es la necesidad que ambas salidas sean exactamente iguales para que la prueba pueda ser satisfactoria, cosa que es difícil de garantizar. Para resolver este problema se incluyó la funcionalidad *filter* del comando *run* para así poder omitir o enmascarar la información variable de la salida del programa a ejecutar.

3.4 Implementación

La tarea de *utiltest* es interpretar un archivo de entrada y evaluar las distintas acciones descritas por el lenguaje, por lo que técnicamente será un intérprete.

Para la creación de un intérprete los pasos que se deben de seguir son muy similares a los de la elaboración de un compilador, ambos requieren una fase de análisis léxico y una fase de análisis sintáctico para poder obtener una estructura que represente al programa respetando la jerarquía de dependencias entre las distintas sentencias.

La principal diferencia es que el compilador transforma esta jerarquía de sentencias a código nativo del procesador tras haberle aplicado optimizaciones mientras que el intérprete suele únicamente recorrer la estructura jerárquica y evalúa las respectivas sentencias en el orden adecuado.

Análisis léxico

La primera fase para la elaboración del intérprete es el análisis léxico, es decir, transformar la secuencia de caracteres del archivo de entrada a lexemas o ***tokens***. El propósito del análisis léxico es obtener un arreglo de *tokens* que representan al programa para luego ser usados en el parser facilitando su trabajo.

Para identificar y clasificar los distintos lexemas se hace uso de expresiones regulares por lo que lo primero que se debe de hacer es definir qué *tokens* serán válidos para el lenguaje de *utiltest* así como las expresiones regulares que los definen.

Palabras reservadas

run	run_sql	read_file	print	sql	in	diff
not	if	else	while	script	unlink	skip
capture	expect	kill	args	filter	outfile	input

Tabla 7. Palabras reservadas

Para el caso de *utiltest*, éstos serán los *tokens*:

Token	Expresión regular
‘RESERVED_WORD’	`print|in|not|run|unlink|capture|expect|kill|args|run_sql|sql|diff|times|script|if|else|while|outfile|input|filter|read_file|skip`
‘QUOTED_TEXT’	`"(.*?[^\\])?"|'(.*?[^\\])?'`
'EXPRESSION’	`` `.*?` ``
‘REGEX’	`\/.*?\/[gi]*`
'ASSIGN_OP'	`<=`

‘REPLACE_OP'	=>
‘DELIMITER’	;
‘COMMA’	,
‘FUNC_ID’	&[a-zA-Z_]\w*
‘GROUPING’	[\(\){}]
‘FLOAT’	\d*\.d+([eE]\d+)?
‘INTEGER’	\d+
‘SIGNAL’	(SIG)?(KILL|INT|TERM)
‘TIMEUNIT’	sec(ond)?s?|min(ute)?s?|hours?
‘VARIABLE’	:[a-zA-Z]\w*:
‘IDENTIFIER’	[a-zA-Z]\w*
‘COMMENT’	#.*?\n
‘EOF’	$

Tabla 8. Tokens

Ahora que ya hemos definido los distintos tipos de *tokens*, lo siguiente es evaluar el archivo con el programa a interpretar para así clasificar el texto e ir construyendo un arreglo de *tokens*. En ocasiones es importante el orden en el cual son evaluadas las distintas expresiones regulares debido a que una cadena puede ser aceptada por más de una expresión regular que definimos. Por ejemplo en este caso es necesario primero evaluar las palabras reservadas y después evaluar los identificadores ya que cualquier palabra reservada puede ser aceptada por la expresión regular para identificadores.

Implementación en Perl del Lexer

Perl es un lenguaje con gran soporte para expresiones regulares, es por esto que la implementación de un lexer en Perl es muy sencilla. Lo único necesario es tener la cadena de entrada a *tokenizar* y las expresiones regulares que definen a los distintos *tokens*.

En Perl existen distintos modificadores para las expresiones regulares así como distintos caracteres escapados dentro de las expresiones regulares. En este caso usaremos los modificadores g (global) y c (mantener la posición a pesar de falla) y el caracter escapado \G. \G acompañado de gc nos permite continuar el procesamiento de la cadena de entrada desde la posición en que la última expresión regular fue aceptada (Orwant, 1996).

```
sub lexer()
{
  # cadena de entrada
  $_ = shift;
  my $cntr = 1;
  while(1) {

    # saltar espacios en blanco
    next if /\G[ \t]+/gc;

    # saltar comentarios y nuevas líneas actualizando el contador de línea
    if (/\G(#.*?)?\n/gc) {
```

```
    $cntr++ , next
  }
  # modificador x para ignorar espacios para que sea más legible
  if (/\G(print    |  in     |  not          |
          run      |  unlink |  capture      |
          expect   |  kill   |  args         |
          run_sql  |  sql    |  diff         |
          times    |  script |  if           |
          else     |  while  |  outfile      |
          input    |  filter |  read_file    |
          skip
         )\b/gcx){
    push(@tokens,{ type => 'RESERVED_WORD',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G"(.*?[^\\])?"/gc){
    push(@tokens,{ type => 'QUOTED_TEXT',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G'(.*?[^\\])?'/gc){
    push(@tokens,{ type => 'QUOTED_TEXT',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G`(.*?)`/gc){
    push(@tokens,{ type => 'EXPRESSION',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G(\/.*?\/[gi]*)/gc){
    push(@tokens,{ type => 'REGEX',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G(<=)/gc){
    push(@tokens,{ type => 'ASSIGN_OP',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G(=>)/gc){
    push(@tokens,{ type => 'REPLACE_OP',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G(;)/gc){
    push(@tokens,{ type => 'DELIMITER',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G(,)/gc){
    push(@tokens,{ type => 'COMMA',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G&([a-zA-Z_]\w*)/gc){
    push(@tokens,{ type => 'FUNC_ID',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G([\(\){}])/gc){
    push(@tokens,{ type => 'GROUPING',
                   value => $1,
                   line => $cntr }), next;
  }
  if (/\G(\d*\.d+([eE]\d+)?)/gc){
    push(@tokens,{ type => 'FLOAT',
                   value => $1,
```

```
                    line => $cntr }), next;
    }
    if (/\G(\d+)/gc){
      push(@tokens,{ type => 'INTEGER',
                     value => $1,
                     line => $cntr }), next;
    }
    if (/\G(sec(ond)?s?|min(ute)?s?|hours?)\b/gc){
      push(@tokens,{ type => 'TIMEUNIT',
                     value => $1,
                     line => $cntr }), next;
    }
    if (/\G((SIG)?(KILL|INT|TERM))\b/gc){
      push(@tokens,{ type => 'SIGNAL',
                     value => $1,
                     line => $cntr }), next;
    }
    if (/\G:([a-zA-Z]\w*):/gc){
      push(@tokens,{ type => 'VARIABLE',
                     value => $1,
                     line => $cntr }), next;
    }
    if (/\G([a-zA-Z]\w*)\b/gc){
      push(@tokens,{ type => 'IDENTIFIER',
                     value => $1,
                     line => $cntr }), next;
    }
    # cualquier caracter no esperado es un error léxico
    die "Unexpected lexical error at \"". substr($_, pos, 40)."...\"\n" if (pos != length);
    # token para fin de cadena
    push(@tokens,{type => 'EOF', value => '$', line => $cntr});
    last;
  }
}
```

Código en Perl 1. Implementación del Lexer

La idea principal es iterar sobre la cadena de entrada en un ciclo infinito probando con cada una de las expresiones regulares. Si alguna acepta parte del inicio de la cadena se genera un *token* con la información necesaria y es introducido en el arreglo @tokens.

Los *tokens* tienen tres valores: el tipo de *token* que es su clasificación; el valor del *token* que es lo que la expresión regular que define al *token* capturó; y el número de línea en la cual el *token* fue encontrado. El número de línea actual está almacenado en la variable $cntr la cual es incrementada cada vez que se encuentra un caracter de salto de línea “\n” y será usado principalmente para el reporte de errores sintácticos.

Cualquier caracter que no sea aceptado por ninguna de las expresiones regulares provoca la salida del programa reportando un error léxico, la otra manera de terminar el ciclo infinito es consumiendo toda la cadena de entrada y cumpliendo la condición *“(pos == length)”* que significa que la última posición de la última expresión regular aceptada es igual que el tamaño de la cadena de entrada.

Si la cadena fue consumida satisfactoriamente un último *token* “EOF” es agregado a la lista de *tokens* para especificar que se llegó al fin de cadena.

Considere el siguiente programa de *utiltest*

```
1: # variable para el archivo de salida
2: salida <= ":TESTDIR:/salida.out";
3:
4: run id
5:   # capturar nombre y grupo actual
6:   capture nombre, grupo <= "uid=\d+\((\w+)\) gid=\d+\((\w+)\)";
7:
8: print "variables: nombre=:nombre:,grupo=:grupo:";
9:
10: run ls
11:   args '-l'
12:   # sustituir nombre y grupo en la salida
13:   filter ":nombre:" => "USUARIO", ":grupo:" => "GRUPO",
14:          "total \d+" => skip # quitar de la salida el total
15:   # filtrar las fechas también
16:   filter "\w{3} +\d{1,2} +\d\d:\d\d" => "FECHA"
17:   outfile ":salida:"
18:   kill SIGKILL in 10 seconds;
19: # comparar la salida contra la salida esperada
20: diff ":salida:" ":TESTDIR:/salida.mas";
```

Código en utiltest 31

Tras procesar el programa anterior con el lexer se obtienen los siguientes *tokens*.

Línea	Tipo	Valor
2	IDENTIFIER	salida
2	ASSIGN_OP	<=
2	QUOTED_TEXT	:TESTDIR:/salida.out
2	DELIMITER	;
4	RESERVED_WORD	run
4	IDENTIFIER	id
6	RESERVED_WORD	capture
6	IDENTIFIER	nombre
6	COMMA	,
6	IDENTIFIER	grupo
6	ASSIGN_OP	<=
6	QUOTED_TEXT	uid=\d+\((\w+)\) gid=\d+\((\w+)\)
6	DELIMITER	;
8	RESERVED_WORD	print
8	QUOTED_TEXT	variables: nombre=:nombre:, grupo=:grupo:
8	DELIMITER	;
10	RESERVED_WORD	run
10	IDENTIFIER	ls
11	RESERVED_WORD	args
11	QUOTED_TEXT	-l
13	RESERVED_WORD	filter
13	QUOTED_TEXT	:nombre:
13	REPLACE_OP	=>

13	QUOTED_TEXT	USUARIO
13	COMMA	,
13	QUOTED_TEXT	:grupo:
13	REPLACE_OP	=>
13	QUOTED_TEXT	GRUPO
13	COMMA	,
14	QUOTED_TEXT	total \d+
14	REPLACE_OP	=>
14	QUOTED_TEXT	BLOQUES
16	RESERVED_WORD	filter
16	QUOTED_TEXT	\w{3} +\d{1,2} +\d\d:\d\d
16	REPLACE_OP	=>
16	QUOTED_TEXT	FECHA
17	RESERVED_WORD	outfile
17	QUOTED_TEXT	:salida:
18	RESERVED_WORD	kill
18	SIGNAL	SIGKILL
18	RESERVED_WORD	in
18	INTEGER	10
18	TIMEUNIT	seconds
18	DELIMITER	;
20	RESERVED_WORD	diff
20	QUOTED_TEXT	:salida:
20	QUOTED_TEXT	:TESTDIR:/salida.mas
20	DELIMITER	;
21	EOF	$

Tabla 9. Tokens generados

Ahora que el lexer puede generar arreglos de *tokens* lo siguiente es crear un parser que pueda usarlos para la verificación de la sintaxis del lenguaje.

Análisis sintáctico

El análisis sintáctico se encarga de verificar la cadena de *tokens* que recibe como entrada y comprobar, mediante una serie de reglas, si la secuencia de *tokens* correspondiente es válida y pertenece al lenguaje que definen las reglas (Alfred V. Aho, 1986).

Para poder reconocer un lenguaje se requieren una serie de reglas de producción que sean capaces de definir al lenguaje, a esta serie de reglas de producción se les denomina *gramática*.

Una gramática está formalmente definida por un conjunto finito N de símbolos terminales, un conjunto finito Σ de símbolos no terminales, un símbolo S inicial y un conjunto finito P de reglas de producción tal que cada regla de producción es de la forma $(\Sigma \cup N)^*N(\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$. En otras palabras, las reglas de producción son de la forma "$\boldsymbol{\alpha} \rightarrow \boldsymbol{\beta}$" donde tanto **α** como **β** pueden ser una serie de símbolos terminales y no terminales con la única restricción de que **α**

debe contener cuando menos un símbolo no terminal. Como su nombre lo indica, los símbolos terminales no pueden expandirse mientras que los símbolos no terminales pueden ser sustituidos mediante alguna de las reglas de producción que los definan.

Existen distintas clasificaciones de las distintas gramáticas, pero las más usadas para la creación de lenguajes de programación son las denominadas gramáticas de contexto libre, las cuales generan lenguajes que pueden ser reconocidos por un autómata con pila y sus reglas de producción son de la forma "A → β" donde A es un símbolo no terminal y β es una cadena de símbolos terminales y/o símbolos no terminales.

Así como hay distintos tipos de gramáticas también hay distintas sub-clasificaciones de las gramáticas libres de contexto. Esto es importante porque de acuerdo al tipo de gramática de libre contexto que se defina se pueden o no usar ciertas técnicas para la creación de su respectivo *parser*.

Debido a que el lenguaje de *utiltest* es relativamente sencillo, podemos hacer algunos sacrificios del poder de expresión de la gramática y así crear una gramática que sea relativamente fácil de *parsear*. En este caso se creará un *parser* llamado "*parser* predictivo recursivo descendente" que utiliza la técnica de *parseo* llamada *top-down* usando *lookahead* para evitar tener que utilizar *backtracking*. Este tipo de gramáticas son denominadas gramáticas LL(k) donde k es el número de *tokens* para el *lookahead*, que en este caso será k=1.

Para que una gramática sea considerada LL(1) es necesario que se satisfagan algunas condiciones:

1.- Es necesario que sea posible determinar qué producción se debe usar basándose únicamente en el símbolo no terminal actual y el *token* de *lookahead*, es decir evitar producciones del estilo

$$S \rightarrow a$$

$$S \rightarrow aT$$

Si existe este tipo de producciones, cuando se trate de expandir un no terminal S dado que el siguiente *token* sea una "a", no se podrá determinar cuál de las dos reglas de producción se debe de aplicar.

2.- Es necesario que la gramática no sea recursiva por la izquierda.

$$S \rightarrow Aa$$

$$A \rightarrow S$$

Si existe una regla de producción que genere recursión por la izquierda ésta debe de ser eliminada debido a que un *parser* descendente entraría en un ciclo infinito con este tipo de gramáticas.

Existen diferentes notaciones para describir gramáticas, la más común siendo la Notación de Backus-Naur (BNF por sus siglas en inglés), en este caso se utilizará la versión Extendida de la Notación de Backus-Naur (EBNF). EBNF contiene notación para representar repeticiones, secciones opcionales, entre otras, que ayudan a eliminar de la gramática las transiciones *epsilon* haciéndola más compacta y a su vez permite transformarla de manera más directa a funciones en algún lenguaje de programación.

La Tabla 10 describe la notación que se usará para escribir la gramática.

Símbolo	Descripción
cadena	símbolo no terminal.
“cadena”	símbolo terminal.
{...}*	metacaracter para cero o más repeticiones.
{...}+	metacaracter para una o más repeticiones.
[...]	metacaracter para contenido opcional.
|	metacaracter para separar las distintas reglas para el mismo no terminal.
=	metacaracter de producción

Tabla 10

La gramática LL(1) escrita en EBNF que define al lenguaje reconocido por *utiltest* es la siguiente:

```
start = {stmt}+

stmt = ifstmt
     | whilestmt
     | function
     | variable
     | cmd “;”

cmd = run
    | run_sql
    | read_file
    | print
    | unlink
    | diff

variable = “identifier” “<=” value

value = “quoted_text”
      | “expression”
      | “variable”
      | “integer”
```

```
      | function

diff = “diff” “quoted_text” “quoted_text”

run = “run” utility {run_opts}*

utility = “identifier”
        | “quoted_text”

run_opts = expect
         | args
         | kill
         | input
         | output
         | capture
         | filter

expect = “expect” expect2

expect2 = “integer” “times” to_match
        | {[“not”] to_match}+

to_match = “quoted_text”
         | “regex”

args = “args” “quoted_text”;

kill = “kill” [“signal”] “in” value “timeunit”;

input = “input” {“quoted_text”}+

output = “outfile” “quoted_text”

capture = “capture” identifiers “<=” “quoted_text”

filter = “filter” filter2

filter2 = “quoted_text” “=>” filter3 [“,” filter2]

filter3 = “quoted_text”
        | skip

identifiers = “identifer” {“,” identifiers}*

read_file = “read_file” “quoted_text” {read_file_opts}+

read_file_opts = expect
               | capture

run_sql = “run_sql” sql {run_sql_opts}*

sql = “script” “quoted_text”
    | {“quoted_text”}+

run_sql_opts = read_file_opts
             | filter

ifstmt = “if” “expression” “{“ {stmt}+ “}” [elsestmt]

elsestmt = “else” “{“ {stmt}+ “}”
```

```
whilestmt = “while” “expression” “{“ {stmt}* “}”
function = “&” “identifier” “(” params “)”
params = value {“,” value}
print = “print” “quoted_text”
diff = “diff” “quoted_text” [“,”] “quoted_text”
unlink = “unlink” “quoted_text” {“,” “quoted_text”}*
```

Tabla 11. Gramática de Utiltest

Implementación del parser

La idea principal de hacer un gramática es tomarla como base para la creación del parser, en este caso, para cada una de las reglas de producción es posible crear una función que la represente dentro del lenguaje, estas funciones serán mutuamente recursivas y serán llamadas de acuerdo al orden de la secuencia de los *tokens*.

Las reglas generales que se seguirán para la implementación del parser recursivo son las siguientes:

1. Para cada símbolo no terminal habrá una función asociada al símbolo.
2. Se tendrá acceso al siguiente *token* de la secuencia.
3. Se tendrá una función para consumir el *token* actual y actualizar el siguiente.
4. De acuerdo al siguiente *token* se decidirá qué función llamar (*lookahead*).
5. Las secciones opcionales de las reglas de producción serán implementadas por sentencias “*if*”.
6. Las secciones con repeticiones de 0 o más repeticiones {α}* serán implementadas por sentencias “*while*”.
7. Las secciones con repeticiones de 1 o más repeticiones {α}+ serán implementadas por sentencias “*do…while*”.

Además de las funciones que saldrán de la gramática se requieren algunas funciones auxiliares que se encarguen la actualización del *token* de *lookahead* y del reporte de los errores de sintaxis.

La función consume() espera como argumentos la clasificación del *token* a esperar así como un argumento opcional, que es el valor del *token*. Esta función se encarga de verificar que el siguiente *token* de entrada es de la misma clasificación que el *token* esperado. Si el valor del *token* también fue pasado a la función se verifica que también corresponda a lo esperado.
Una vez que se verifica el *token* actual, actualiza el *token* de lookahead tomando el siguiente elemento del arreglo de *tokens*. Si encuentra algún problema se termina la ejecución del programa reportando el error correspondiente.

Es conveniente tener una función auxiliar para el reporte de errores de sintaxis para evitar duplicar código en varias funciones.

```
sub syntax_error(){
  my $line = "Syntax error near \"".$next->{value};
  foreach my $tok (@tokens){
    last if($tok->{line} != $next->{line});
    $line .= " $tok->{value}";
  }
  $line = substr($line, 0, 50);
  return $line . "\" at line $next->{line}.\n";
}
```

Código en Perl 2. Implementación de syntax_error

La función syntax_error() se encarga de retornar una cadena con la información del estado actual de los *token* al momento en que fue llamada. En esta función es donde se hace el uso del número de línea que fue capturado dentro de los *tokens*, esto facilita la localización del error en el archivo fuente. La cadena que se retorna incluye la línea en la que ocurrió el error acompañada de una sección del texto posterior a donde ocurrió el error.

Se requiere una función que consuma algún *token* esperado y que a su vez actualice el *token* de lookahead.

```
sub consume($;$){
  my ($match_type, $match_val) = @_;
  my $curval = $next->{value};
  if($match_type eq $next->{type}){
    if(defined($match_val) and $match_val ne $next->{value}){
      die("Unexpected symbol $next->{type} $next->{value}");
    }
    if (@tokens){
      $next = shift(@tokens);
    } else {
      die("Empty Stack");
    }
  } else {
    $match_val = "" unless defined($match_val);
    die &syntax_error() .
        "\ngot $next->{type}:$next->{value} while expecting $match_type:$match_val\n";
  }
  return $curval;
}
```

Código en Perl 3. Implementación de consume

Como última función auxiliar se puede agregar una función que su único propósito sea el inicializar el *token* de lookahead y llamar a la función que representa la regla inicial de la gramática.

```
sub parse()
{
  $next = shift(@tokens);
  start();
}
```

Código en Perl 4. Implementación de la función parse

Una vez que se tienen las funciones auxiliares se puede proceder a implementar la gramática transformando cada no terminal en una función.

```
# start = {stmt}+
sub start(){
  do {
    stmt();
  } while( $type ne 'END');
  print "ACCEPTED!!\n";
}

# stmt = ifstmt  | whilestmt
#      | fuction | variable
#      | cmd ";"
sub stmt(){
  if($next->{type} eq 'RESERVED_WORD'){
    if ($next->{value} eq 'if'){
      ifstmt();
    }
    elsif( $next->{value} eq 'while'){
      whilestmt();
    }
    else{
      cmd();
      consume('DELIMITER');
    }
  }
  elsif($next->{type} eq 'FUNC_ID'){
    function();
    consume('DELIMITER');
  }
  elsif($next->{type} eq 'IDENTIFIER'){
    variable();
    consume('DELIMITER');
  }
  else {
    die &syntax_error;
  }
}


# cmd = run | run_sql | read_file
#     | print    | unlink  | diff
sub cmd(){
  if ($next->{value} eq 'run'){
    runutil();
```

```
  }
  elsif ($next->{value} eq 'run_sql'){
    runsql();
  }
  elsif ($next->{value} eq 'read_file'){
    readfile();
  }
  elsif ($next->{value} eq 'print'){
    prnt();
  }
  elsif ($next->{value} eq 'unlink'){
    unlnk();
  }
  elsif ($next->{value} eq 'diff'){
    diff();
  }
  else {
    die &syntax_error();
  }
  return $cmd;
}

# run = "run" utility {run_opts}*
sub runutil(){
  consume('RESERVED_WORD', 'run');
  utility();
  do {
    run_opts();
  } while ($next->{type} eq 'RESERVED_WORD');
}

# run_opts = expect | args    | kill | input
#              | output | capture | filter
sub run_opts(){
  if ($next->{value} eq 'args'){
    args();
  }
  elsif ($next->{value} eq 'kill'){
    killcmd();
  }
  elsif ($next->{value} eq 'input'){
    input();
  }
  elsif ($next->{value} eq 'outfile'){
    output();
  }
  elsif ($next->{value} eq 'capture'){
    capt();
  }
  elsif ($next->{value} eq 'expect'){
    expt();
  }
  elsif ($next->{value} eq 'filter'){
    filter();
  }
  else {
    die &syntax_error;
  }
}

# read_file = "read_file" "quoted_text" {read_file_opts}+
```

```
sub readfile(){
  consume('RESERVED_WORD', 'read_file');
  consume('QUOTED_TEXT');
  do {
    read_file_opts();
  } while ($next->{type} eq 'RESERVED_WORD');
}

# read_file_opts = expect | capture
sub read_file_opts(){
  if ($next->{value} eq 'capture'){
    capt();
  }
  elsif ($next->{value} eq 'expect'){
    expt();
  }
  else {
    die &syntax_error;
  }
}

# run_sql = "run_sql" sql {run_sql_opts}*
sub runsql(){
  consume('RESERVED_WORD', 'run_sql');
  sql();
  while ($next->{type} eq 'RESERVED_WORD'){
    read_file_opts();
  }
}

# sql = "script" "quoted_text"
#     | {"quoted_text"}+
sub sql(){
  if($next->{type} eq 'RESERVED_WORD'){
    consume('RESERVED_WORD', 'script');
    consume('QUOTED_TEXT');
  }
  elsif($next->{type} eq 'QUOTED_TEXT'){
    do{
      consume('QUOTED_TEXT');
    }while($next->{type} eq 'QUOTED_TEXT');
  }
  else {
    die &syntax_error;
  }
}

# variable = "identifier" "<=" value
sub variable(){
  consume('IDENTIFIER');
  consume('ASSIGN_OP');
  value();
}

# print = "print" "quoted_text"
sub prnt(){
  consume('RESERVED_WORD', 'print');
  consume('QUOTED_TEXT');
}

# diff = "diff" "quoted_text" [","] "quoted_text"
```

```
sub diff(){
  consume('RESERVED_WORD', 'diff');
  consume('QUOTED_TEXT');
  consume('QUOTED_TEXT');
}

# unlink = "unlink" "quoted_text" {"," "quoted_text"}*
sub unlnk(){
  consume('RESERVED_WORD', 'unlink');
  consume('QUOTED_TEXT');
  while ($next->{type} eq 'COMMA'){
    consume('COMMA');
    consume('QUOTED_TEXT');
  }
}

# value = "quoted_text" | "expression"
#       | "variable" | function | "integer"
sub value(){
  if ($next->{type} eq 'QUOTED_TEXT'){
    consume('QUOTED_TEXT');
  }
  elsif($next->{type} eq 'FUNC_ID'){
    function();
  }
  elsif($next->{type} eq 'VARIABLE'){
    consume('VARIABLE');
  }
  elsif($next->{type} eq 'EXPRESSION'){
    consume('EXPRESSION')};
  }
  elsif($next->{type} eq 'INTEGER'){
    consume('INTEGER');
  }
  else {
    die &syntax_error;
  }
}

# function = "&" "identifier" "(" params ")"
sub function(){
  consume('FUNC_ID');
  consume('GROUPING', '(');
  params();
  consume('GROUPING', ')');
}

# params = value {"," value}
sub params(){
  value();
  while( $next->{type} eq 'COMMA' ) {
    consume('COMMA');
    value();
  }
}

# args = "args" "quoted_text";
sub args(){
  consume('RESERVED_WORD', 'args');
  consume('QUOTED_TEXT');
}
```

```
# input = "input" {"quoted_text"}+
sub input(){
  consume('RESERVED_WORD', 'input');
  do{
    consume('QUOTED_TEXT');
  } while($next->{type} eq 'QUOTED_TEXT');
}

# output = "outfile" "quoted_text"
sub output(){
  consume('RESERVED_WORD', 'outfile');
  consume('QUOTED_TEXT');
}

# capture = "capture" identifiers "<=" "quoted_text"
sub capt(){
  consume('RESERVED_WORD', 'capture');
  identifiers();
  consume('ASSIGN_OP');
  consume('QUOTED_TEXT');
}

# identifiers = "identifer" {"," identifiers}*
sub identifiers(){
  consume('IDENTIFIER');
  while( $next->{type} eq 'COMMA' ) {
    consume('COMMA');
    identifiers();
  }
}

# kill = "kill" ["signal"] "in" value "timeunit";
sub killcmd(){
  consume('RESERVED_WORD', 'kill');
  if($next->{type} eq 'SIGNAL'){
    consume('SIGNAL');
  }
  consume('RESERVED_WORD', 'in');
  value();
  consume('TIMEUNIT');
}

# filter = "filter" filter2
sub filter(){
  consume('RESERVED_WORD', 'filter');
  filter2();
}

# filter2 = "quoted_text" "=>" filter3 ["," filter2]
sub filter2(){
  consume('QUOTED_TEXT');
  consume('REPLACE_OP');
  filter3();
  if($next->{type} eq 'COMMA'){
    consume('COMMA');
    filter2();
  }
}

# filter3 = "quoted_text" | skip
```

```
sub filter3(){
  if($next->{type} eq 'QUOTED_TEXT'){
    consume('QUOTED_TEXT');
  } else {
    consume('RESERVED_WORD', 'skip');
  }
}

# utility = "identifier"
#         | "quoted_text"
sub utility(){
  if($next->{type} eq 'IDENTIFIER'){
    consume('IDENTIFIER');
  } elsif ($next->{type} eq 'QUOTED_TEXT'){
    consume('QUOTED_TEXT');
  }
  else {
    die &syntax_error;
  }
}

# expect = "expect" expect2
sub expt(){
  consume('RESERVED_WORD', 'expect');
  expt2();
}

# expect2 = "integer" "times" to_match
#         | {["not"] to_match}+
sub expt2(){
  if($next->{type} eq 'INTEGER' || $next->{type} eq 'VARIABLE'){
    consume($next->{type});
    consume('RESERVED_WORD', 'times');
    consume('QUOTED_TEXT');
  }
  else {
    do{
      if($next->{type} eq 'RESERVED_WORD' and $next->{value} eq 'not'){
        consume('RESERVED_WORD', 'not');
      }
      to_match();
    } while($next->{type} eq 'RESERVED_WORD' and $next->{value} eq 'not' or
            $next->{type} eq 'REGEX' or $next->{type} eq 'QUOTED_TEXT');
  }
}

# to_match = "quoted_text" | "regex"
sub to_match(){
  if($next->{type} eq 'REGEX'){
    consume('REGEX');
  } elsif ($next->{type} eq 'QUOTED_TEXT'){
    consume('QUOTED_TEXT');
  }
  else {
    die &syntax_error;
  }
}

# ifstmt = "if" "expression" "{" {stmt}+ "}" [elsestmt]
sub ifstmt(){
  consume('RESERVED_WORD', 'if');
```

```
  consume('EXPRESSION');
  consume('GROUPING', '{');
  while ($next->{type} ne 'GROUPING'){
    stmt();
  }
  consume('GROUPING', '}');
  if($next->{type} eq 'RESERVED_WORD' and $next->{value} eq 'else'){
    elsestmt();
  }
}

# elsestmt = "else" "{" {stmt}+ "}"
sub elsestmt(){
  consume('RESERVED_WORD', 'else');
  consume('GROUPING', '{');
  while ($next->{type} ne 'GROUPING'){
    stmt();
  }
  consume('GROUPING', '}');
}

# whilestmt = "while" "expression" "{" {stmt}* "}"
sub whilestmt(){
  consume('RESERVED_WORD', 'while');
  consume('EXPRESSION');
  consume('GROUPING', '{');
  while ($next->{type} ne 'GROUPING'){
    stmt();
  }
  consume('GROUPING', '}');
}
```

Código en Perl 5. Implementación del parser

Al inicio de cada función se describen las reglas de producción del no terminal que corresponde a esa función.
Una vez implementadas las funciones ya se tiene un parser funcional el cual reconoce el lenguaje definido por la gramática y es capaz de reconocer y reportar errores de sintaxis.

Ahora si procesamos el programa del Código en utiltest 31 de la sección del *lexer* la salida es la siguiente

$ utiltest programa31.tst
ACCEPTED!!

Tabla 12

Modificando el programa del Código en utiltest 31 incluyendo errores de sintaxis nos queda el siguiente programa de *utiltest*.

```
1:  # variable para el archivo de salida
2:  salida = ":TESTDIR:/salida.out";
3:
4:  run id
5:    # capturar nombre y grupo actual
6:    capture nombre, grupo => "uid=\d+\((\w+)\) gid=\d+\((\w+)\)";
7:
8:  print "variables: nombre=:nombre:,grupo=:grupo:"
9:
10: run ls
11:   args '-l'
12:   # sustituir nombre y grupo en la salida
13:   filter ":nombre:" <= "USUARIO", ":grupo:" => "GRUPO",
14:          "total \d+" => squip # quitar de la salida el total
15:   # filtrar las fechas también
16:   filter "\w{3} +\d{1,2} +\d\d:\d\d" => "FECHA"
17:   outfile ":salida:"
18:   kill SIGKILL in 10 secons;
19: # comparar la salida contra la salida esperada
20: diff ":salida:" ":TESTDIR:/salida.mas"
21:
```

Código en utiltest 32. Programa de utiltest con errores de sintaxis

Al ejecutarlo e ir corrigiendo los errores presentados la salida es la siguiente:

```
$ utiltest lstest.tst
Unexpected lexical error at line 2 near
"= ":TESTDIR:/salida.out";

run id ..."
```

Tabla 13. Error léxico '='

Este es un error léxico, por lo que el error proviene del lexer. El error es que el símbolo '=' no está definido en el lenguaje, ninguna de las expresiones regulares del lexer acepta el caracter '='.

```
$ utiltest lstest.tst
Syntax error near "=> uid=\d+\((\w+)\) gid=\d+\((\" at line 6.
got REPLACE_OP:=> while expecting ASSIGN_OP:
```

Tabla 14. Error sintáctico '=>'

Este es un error de sintaxis, el *token* '=>' es un *token* válido pero en la línea 6 se está usando en el subcomando capture, el cual espera el operador de asignación.

```
$ utiltest lstest.tst
Syntax error near "run ls" at line 10.
got RESERVED_WORD:run while expecting DELIMITER:
```

Tabla 15. Error sintáctico ';'

Este es otro error de sintaxis, el problema fue ocasionado porque el parser esperaba un delimitador ‘;’ pero en lugar de eso encontró la palabra reservada ‘*run’*.

```
$ utiltest lstest.tst
Syntax error near "<= USUARIO , :grupo: => GRUPO ," at line 13.
got ASSIGN_OP:<= while expecting REPLACE_OP:
```

Tabla 16. Error sintáctico ‘<=’

Este es un caso similar al de la línea 6, en este caso se debió de haber usado el operador de reemplazo pero se encontró el operador de asignación.

```
$ utiltest lstest.tst
Syntax error near "squip" at line 14.
got IDENTIFIER:squip while expecting RESERVED_WORD:skip
```

Tabla 17. Error sintáctico ‘squip’

Aquí se encontró con la palabra *squip* cuando se esperaba la palabra *skip*.

```
$ utiltest lstest.tst
Syntax error near "secons ;" at line 18.
got IDENTIFIER:secons while expecting TIMEUNIT:
```

Tabla 18. Error sintáctico ‘secons’

secons no es una unidad válida de tiempo.

```
$ utiltest lstest.tst
Syntax error near "$" at line 21.
got EOF:$ while expecting DELIMITER:
```

Tabla 19. Error sintáctico ‘EOF’

Aquí faltó el delimitador ‘;’ al final del comando diff.

```
$ utiltest lstest.tst
ACCEPTED!!
```

Tabla 20. Programa aceptado

En la salida se demuestra que el parser es capaz de identificar y reportar errores.

Creación del árbol de sintaxis abstracto (AST)

Ser capaz de reconocer el lenguaje no es suficiente para un intérprete, es necesario construir una estructura que conserve la jerarquía de llamadas de las funciones del parser así como los datos de los tokens leídos. Éste tipo de estructura omite información trivial para la interpretación como lo son las comas, separadores y operadores de agrupación y se enfoca en los datos que son útiles para la ejecución del programa. A este tipo de estructura se le conoce comúnmente como Árbol de Sintaxis Abstracto (o AST por sus siglas en inglés).

Para construir un AST únicamente es necesario generar estructuras en las funciones del parser e ir capturando la información necesaria en estas estructuras. La estructura normalmente es un árbol que se va construyendo a medida que se llaman las funciones del parser. Cada función regresa un árbol más pequeño que se va insertando como nodo del árbol principal para así generar el AST.

En el caso de *utiltest* se utilizarán las estructuras de datos de Perl para generar el equivalente de un AST. Las estructuras de Perl son los arreglos de elementos, hashes (también llamados arreglos asociativos) y escalares (cadenas de caracteres y números).

La estructura que almacenará la sintaxis en *utiltest* será un arreglo en lugar de un árbol. Cada elemento del arreglo contendrá una referencia a una estructura con toda la información necesaria para la evaluación del programa.

```
# start = {stmt}+
sub start(){
  my $parsetree = [];
  do {
    push(@$parsetree, stmt());
  } while( $next->{type} ne 'EOF');
  print "ACCEPTED!!\n";
  return $parsetree;
}

# stmt = ifstmt  | whilestmt
#      | fuction | variable
#      | cmd ";"
sub stmt(){
  my $stmt;
  if($next->{type} eq 'RESERVED_WORD'){
    if ($next->{value} eq 'if'){
      $stmt = ifstmt();
    }
    elsif( $next->{value} eq 'while'){
      $stmt = whilestmt();
    }
    else{
      $stmt = cmd();
      consume('DELIMITER');
    }
  }
  elsif($next->{type} eq 'FUNC_ID'){
    $stmt = function();
    consume('DELIMITER');
  }
  elsif($next->{type} eq 'IDENTIFIER'){
    $stmt = variable();
    consume('DELIMITER');
  }
  else {
    die &syntax_error;
  }
  return $stmt;
}
```

```
sub cmd(){
  my $cmd;
  if ($next->{value} eq 'run'){
    $cmd = runutil();
  }
  elsif ($next->{value} eq 'run_sql'){
    $cmd = runsql();
  }
  elsif ($next->{value} eq 'read_file'){
    $cmd = readfile();
  }
  elsif ($next->{value} eq 'print'){
    $cmd = prnt();
  }
  elsif ($next->{value} eq 'unlink'){
    $cmd = unlnk();
  }
  elsif ($next->{value} eq 'diff'){
    $cmd = diff();
  }
  else {
    die &syntax_error();
  }
  return $cmd;
}

# run = "run" utility {run_opts}*
sub runutil(){
  my $cmd = {};
  $cmd->{type} = consume('RESERVED_WORD', 'run');
  $cmd->{program} = utility();
  do {
    run_opts($cmd);
  } while ($next->{type} eq 'RESERVED_WORD');
  return $cmd;
}

# run_opts = expect | args    | kill | input
#              | output | capture | filter
sub run_opts(){
  my $cmd = shift;
  if ($next->{value} eq 'args'){
    $cmd->{args} = args();
  }
  elsif ($next->{value} eq 'kill'){
    $cmd->{kill} = killcmd();
  }
  elsif ($next->{value} eq 'input'){
    $cmd->{input} = input();
  }
  elsif ($next->{value} eq 'outfile'){
    $cmd->{outfile} = output();
  }
  elsif ($next->{value} eq 'capture'){
    $cmd->{capture} = [] unless(defined $cmd->{capture});
    push(@{$cmd->{capture}}, capt());
  }
  elsif ($next->{value} eq 'expect'){
    $cmd->{expect} = [] unless(defined $cmd->{expect});
    expt($cmd->{expect});
  }
```

```
  elsif ($next->{value} eq 'filter'){
    $cmd->{filters} = [] unless(defined $cmd->{filters});
    filter($cmd->{filters});
  }
  else {
    die &syntax_error;
  }
}

# read_file = "read_file" "quoted_text" {read_file_opts}+
sub readfile(){
  my $cmd = {};
  $cmd->{type} = consume('RESERVED_WORD', 'read_file');
  $cmd->{file} = consume('QUOTED_TEXT');
  do {
    &read_file_opts($cmd);
  } while ($next->{type} eq 'RESERVED_WORD');
  $cmd;
}

# read_file_opts = expect | capture
sub read_file_opts(){
  my $cmd = shift;
  if ($next->{value} eq 'capture'){
    $cmd->{capture} = [] unless(defined $cmd->{capture});
    push(@{$cmd->{capture}}, capt());
  }
  elsif ($next->{value} eq 'expect'){
    $cmd->{expect} = [] unless(defined $cmd->{expect});
    expt($cmd->{expect});
  }
  else {
    die &syntax_error;
  }
  return $cmd;
}

# run_sql = "run_sql" sql {run_sql_opts}*
sub runsql(){
  my $cmd = {sql => []};
  $cmd->{type} = consume('RESERVED_WORD', 'run_sql');
  &sql($cmd);
  while ($next->{type} eq 'RESERVED_WORD'){
    &read_file_opts($cmd);
  }
  return $cmd;
}

# sql = "script" "quoted_text"
#     | {"quoted_text"}+
sub sql(){
  my $cmd = shift;
  if($next->{type} eq 'RESERVED_WORD'){
    consume('RESERVED_WORD', 'script');
    $cmd->{script} = consume('QUOTED_TEXT');
  }
  elsif($next->{type} eq 'QUOTED_TEXT'){
    do{
      push(@{$cmd->{sql}}, consume('QUOTED_TEXT'));
    }while($next->{type} eq 'QUOTED_TEXT');
  }
```

```
  else {
    die &syntax_error;
  }
}

# variable = "identifier" "<=" value
sub variable(){
  my $cmd = {};
  $cmd->{type} = 'var';
  $cmd->{id} = consume('IDENTIFIER');
  consume('ASSIGN_OP');
  $cmd->{value} = value();
  return $cmd;
}

# print = "print" "quoted_text"
sub prnt(){
  my $cmd = {};
  $cmd->{type} = consume('RESERVED_WORD', 'print');
  $cmd->{text} = consume('QUOTED_TEXT');
  return $cmd;
}

# diff = "diff" "quoted_text" [","] "quoted_text"
sub diff(){
  my $cmd = {};
  $cmd->{type} = consume('RESERVED_WORD', 'diff');
  $cmd->{file1} = consume('QUOTED_TEXT');
  $cmd->{file2} = consume('QUOTED_TEXT');
  return $cmd;
}

# unlink = "unlink" "quoted_text" {"," "quoted_text"}*
sub unlnk(){
  my $cmd = { globs => [] };
  $cmd->{type} = consume('RESERVED_WORD', 'unlink');
  push (@{$cmd->{globs}}, consume('QUOTED_TEXT'));
  while ($next->{type} eq 'COMMA'){
    consume('COMMA');
    push (@{$cmd->{globs}}, consume('QUOTED_TEXT'));
  }
  return $cmd;
}

# value = "quoted_text" | "expression"
#       | "variable" | function | "integer"
sub value(){
  my $val;
  if ($next->{type} eq 'QUOTED_TEXT'){
    $val = consume('QUOTED_TEXT');
  }
  elsif($next->{type} eq 'FUNC_ID'){
    $val = function();
  }
  elsif($next->{type} eq 'VARIABLE'){
    $val = {type => "var",
            id   => consume('VARIABLE');
  }
  elsif($next->{type} eq 'EXPRESSION'){
    $val = {type => 'expression',
            expr => consume('EXPRESSION')};
```

```
  }
  elsif($next->{type} eq 'INTEGER'){
    $val = consume('INTEGER');
  }
  else {
    die &syntax_error;
  }
  return $val;
}

# function = "&" "identifier" "(" params ")"
sub function(){
  my $func = {params => []};
  $func->{type} = 'function';
  $func->{name} = consume('FUNC_ID');
  consume('GROUPING', '(');
  $func->{params} = params();
  consume('GROUPING', ')');
  return $func;
}

# params = value {"," value}
sub params(){
  my $parms = [];
  push (@$parms, &value());
  while( $next->{type} eq 'COMMA' ) {
    consume('COMMA');
    push (@$parms, &value());
  }
  return $parms;
}

# args = "args" "quoted_text";
sub args(){
  consume('RESERVED_WORD', 'args');
  return consume('QUOTED_TEXT');
}

# input = "input" {"quoted_text"}+
sub input(){
  my $input = [];
  consume('RESERVED_WORD', 'input');
  do{
    push(@$input, consume('QUOTED_TEXT'));
  } while($next->{type} eq 'QUOTED_TEXT');
  return $input;
}

# output = "outfile" "quoted_text"
sub output(){
  consume('RESERVED_WORD', 'outfile');
  return consume('QUOTED_TEXT');
}

# capture = "capture" identifiers "<=" "quoted_text"
sub capt(){
  my $capture = {identifiers => []};
  consume('RESERVED_WORD', 'capture');
  identifiers($capture->{identifiers});
  consume('ASSIGN_OP');
  $capture->{regex} = consume('QUOTED_TEXT');
```

```
  return $capture;
}

# identifiers = "identifer" {"," identifiers}*
sub identifiers(){
  my $ids = shift;
  push (@$ids, consume('IDENTIFIER'));
  while( $next->{type} eq 'COMMA' ) {
    consume('COMMA');
    &identifiers($ids);
  }
}

# kill = "kill" ["signal"] "in" value "timeunit";
sub killcmd(){
  my $kill = {};
  consume('RESERVED_WORD', 'kill');
  if($next->{type} eq 'SIGNAL'){
    $kill->{signal} = consume('SIGNAL');
  }
  consume('RESERVED_WORD', 'in');
  $kill->{number} = value(); #consume('INTEGER');
  $kill->{unit} = consume('TIMEUNIT');
  return $kill;
}

# filter = "filter" filter2
sub filter(){
  my $filters = shift;
  consume('RESERVED_WORD', 'filter');
  &filter2($filters);
}

# filter2 = "quoted_text" "=>" filter3 {"," filter2}
sub filter2(){
  my $arr = shift;
  my $filter = {};
  $filter->{from} = consume('QUOTED_TEXT');
  consume('REPLACE_OP');
  $filter->{to} = filter3();
  push (@$arr, $filter);
  if($next->{type} eq 'COMMA'){
    consume('COMMA');
    &filter2($arr);
  }
}

# filter3 = "quoted_text" | skip
sub filter3(){
  my $val;
  if($next->{type} eq 'QUOTED_TEXT'){
    $val = consume('QUOTED_TEXT');
  } else {
    consume('RESERVED_WORD', 'skip');
    $val = undef;
  }
  return $val;
}

# utility = "identifier"
#         | "quoted_text"
```

```
sub utility(){
  my $cmdstr;
  if($next->{type} eq 'IDENTIFIER'){
    $cmdstr = consume('IDENTIFIER');
  } elsif ($next->{type} eq 'QUOTED_TEXT'){
    $cmdstr = consume('QUOTED_TEXT');
  }
  else {
    die &syntax_error;
  }
  return $cmdstr;
}

# expect = "expect" expect2
sub expt(){
  consume('RESERVED_WORD', 'expect');
  return expt2($_[0]);
}

# expect2 = "integer" "times" to_match
#         | {["not"] to_match}+
sub expt2(){
  my $expects = shift;
#accept function?
  if($next->{type} eq 'INTEGER' || $next->{type} eq 'VARIABLE'){
    my $item = {};
    $item->{times} = consume($next->{type});
    consume('RESERVED_WORD', 'times');
    $item->{regex} = consume('QUOTED_TEXT');
    push(@$expects, $item);
  }
  else {
    do{
      my $item = {};
      if($next->{type} eq 'RESERVED_WORD' and $next->{value} eq 'not'){
        $item->{not} = consume('RESERVED_WORD', 'not');
      }
      $item->{regex} = to_match();
      push(@$expects, $item);
    } while($next->{type} eq 'RESERVED_WORD' and $next->{value} eq 'not' or
            $next->{type} eq 'REGEX' or $next->{type} eq 'QUOTED_TEXT');
  }
  return $expects;
}

# to_match = "quoted_text" | "regex"
sub to_match(){
  my $val;
  if($next->{type} eq 'REGEX'){
    $val = consume('REGEX');
  } elsif ($next->{type} eq 'QUOTED_TEXT'){
    $val = consume('QUOTED_TEXT');
  }
  else {
    die &syntax_error;
  }
  return $val;
}

# ifstmt = "if" "expression" "{" {stmt}+ "}" [elsestmt]
sub ifstmt(){
```

```
  my $if = {ifcmds => []};
  $if->{type} = consume('RESERVED_WORD', 'if');
  $if->{condition} = consume('EXPRESSION');
  consume('GROUPING', '{');
  while ($next->{type} ne 'GROUPING'){
    push(@{$if->{ifcmds}}, stmt());
  }
  consume('GROUPING', '}');
  if($next->{type} eq 'RESERVED_WORD' and $next->{value} eq 'else'){
    $if->{elsecmds} = elsestmt();
  }
  return $if;
}

# elsestmt = "else" "{" {stmt}+ "}"
sub elsestmt(){
  my $elsecmds = [];
  consume('RESERVED_WORD', 'else');
  consume('GROUPING', '{');
  while ($next->{type} ne 'GROUPING'){
    push(@$elsecmds, stmt());
  }
  consume('GROUPING', '}');
  return $elsecmds;
}

# whilestmt = "while" "expression" "{" {stmt}* "}"
sub whilestmt(){
  my $while = {cmds => []};
  $while->{type} = consume('RESERVED_WORD', 'while');
  $while->{condition} = consume('EXPRESSION');
  consume('GROUPING', '{');
  while ($next->{type} ne 'GROUPING'){
    push(@{$while->{cmds}}, stmt());
  }
  consume('GROUPING', '}');
  return $while;
}
```

Código en Perl 6. Implementación del parser con la creación del AST

Con los cambios anteriores es posible construir la estructura abstracta que representa al programa a ejecutar. Esta estructura no contiene absolutamente toda la información del programa, si no que únicamente incluye la información necesaria para su evaluación.

```
$AST =  [
          {
            'value' => ':TESTDIR:/salida.out',
            'id' => 'salida',
            'type' => 'var'
          },
          {
            'capture' => [
                           {
                             'identifiers' => [
                                                'nombre',
                                                'grupo'
                                              ],
```

```
                        'regex' => 'uid=\\d+\\(\\w+)\\) gid=\\d+\\(\\w+)\\)'
                      }
                    ],
      'type' => 'run',
      'program' => 'id'
    },
    {
      'text' => 'variables: nombre=:nombre:,grupo=:grupo:',
      'type' => 'print'
    },
    {
      'outfile' => ':salida:',
      'args' => '-l',
      'filters' => [
                      {
                        'to' => 'USUARIO',
                        'from' => ':nombre:'
                      },
                      {
                        'to' => 'GRUPO',
                        'from' => ':grupo:'
                      },
                      {
                        'to' => 'BLOQUES',
                        'from' => 'total \\d+'
                      },
                      {
                        'to' => 'FECHA',
                        'from' => '\\w{3} +\\d{1,2} +\\d\\d:\\d\\d'
                      }
                    ],
      'type' => 'run',
      'kill' => {
                  'unit' => 'seconds',
                  'signal' => 'SIGKILL',
                  'number' => '10'
                },
      'program' => 'ls'
    },
    {
      'file2' => ':TESTDIR:/salida.mas',
      'file1' => ':salida:',
      'type' => 'diff'
    }
  ];
```

Árbol 1. Código en utiltest 33

En la creación de la estructura se hicieron simplificaciones para facilitar la implementación de la evaluación de la misma, por ejemplo en el programa anterior, el cuarto comando es la ejecución del comando ls -l el cuál originalmente contenía 2 subcomandos *filter*.

```
1: run ls
2:  args '-l'
3:  # sustituir nombre y grupo en la salida
4:  filter ":nombre:" => "USUARIO", ":grupo:" => "GRUPO",
5:         "total \d+" => skip # quitar de la salida el total
6:  # filtrar las fechas también
```

```
7:  filter "\w{3} +\d{1,2} +\d\d:\d\d" => "FECHA"
8:  outfile ":salida:"
9:  kill SIGKILL in 10 seconds;
```

Código en utiltest 33

A la hora de crear la estructura se hizo la optimización de juntar todos los datos de los subcomandos como *filter*, *capture* o *expect* en un solo arreglo en lugar de obtener un arreglo por subcomando, esto simplifica la implementación de la evaluación de la estructura al momento de la ejecución del programa.

```
'filters' => [
               [
                 {
                   'to' => 'USUARIO',
                   'from' => ':nombre:'
                 },
                 {
                   'to' => 'GRUPO',
                   'from' => ':grupo:'
                 },
                 {
                   'to' => 'BLOQUES',
                   'from' => 'total \\d+'
                 },
               ],
               [
                 {
                   'to' => 'FECHA',
                   'from' => '\\w{3} +\\d{1,2} +\\d\\d:\\d\\d'
                 }
               ]
             ],
```

Árbol 2. AST del Código en utiltest 33

La gramática presentada es una gramática simplificada que permite especificar algunos subcomandos más de una vez a pesar de que deberían ser únicos. Ésta decisión fue tomada para agregar flexibilidad al lenguaje sin tener que complicar la gramática y así ocasionar la complejidad innecesaria de la implementación del parser, un ejemplo de esto es el comando *run*.

```
run = “run” utility {run_opts}*

run_opts = expect
         | args
         | kill
         | input
         | output
         | capture
         | filter
```

Se puede observar que run_opts puede generar varias veces subcomandos como *kill* o *args* que normalmente se esperaría que fuera únicos. La razón de éste comportamiento es para permitir especificar la lista de comandos en cualquier orden sin tener que especificar en la gramática explícitamente todos los órdenes posibles de cada subcomando de *run*. Sin

embargo, se puede forzar fácilmente la unicidad de estos subcomandos al momento de la creación de la estructura y, por ejemplo, forzar un error de sintaxis al momento de tratar de agregar el valor a la estructura si ya se encuentra presente. Otra solución es tomar el primer o último valor del subcomando e ignorar el resto. *Utiltest* tomará como válido el último subcomando especificado para estos casos.

Otra de las grandes ventajas del parseo es la posibilidad de anidación. Por ejemplo, los comandos *if* y *while* contienen un bloque de sentencias a ejecutar en caso de que la condición sea verdadera. Debido a que *if* y *while* son también sentencias es posible que dentro del bloque de sentencias a ejecutar se encuentren sentencias *if* y *while* que a su vez tendrán más sentencias a ejecutar y así sucesivamente permitiendo varios niveles de anidación. También es posible anidar funciones dentro de los parámetros de funciones.

Considere el siguiente programa que utiliza anidación de funciones y que genera un número aleatorio y lo imprime.

```
1: random <= &rnd(&rnd(0,&rnd(2,5)),&rnd(5,10));
2: print "Número=:random:";
```

Código en utiltest 34

El parser genera la siguiente estructura para el programa anterior.

```
$AST  = [
          {
            'value' => {
                         'params' => [
                                       {
                                         'params' => [
                                                       '0',
                                                       {
                                                         'params' => [
                                                                       '2',
                                                                       '5'
                                                                     ],
                                                         'name' => 'rnd',
                                                         'type' => 'function'
                                                       }
                                                     ],
                                         'name' => 'rnd',
                                         'type' => 'function'
                                       },
                                       {
                                         'params' => [
                                                       '5',
                                                       '10'
                                                     ],
                                         'name' => 'rnd',
                                         'type' => 'function'
                                       }
                                     ],
                         'name' => 'rnd',
                         'type' => 'function'
                       },
            'id' => 'random',
            'type' => 'var'
          },
          {
```

```
            'text' => 'Número=:random:',
            'type' => 'print'
          }
        ];
```

Árbol 3. AST del código en utiltest 34

Considere el siguiente programa de utiltest que utiliza anidación de bloques de control.

```
1: verbose <= 1;
2:
3: iters <= &rnd(10);
4:
5: while `:iters: > 0`{
6:   if `&rnd(0,3) == 0` {
7:    run ls
8:     args '[-ltr|-l]'
9:     expect 6 times 'gicorone'
10:     capture id1 <= 'gicorone(.*)';
11:     if `:verbose:`{
12:       print "captured id1 as ':id1:'\n\n";
13:     }
14:   } else {
15:    run ls
16:     args '-[la|ltra]'
17:     expect 8 times 'gicorone'
18:     capture id2 <= 'gicorone(.*)'
19:     outfile ":TESTDIR:/test.out";
20:     if `:verbose:`{
21:       print "captured id2 as ':id2:'\n";
22:     }
23:   }
24:   iters <= `:iters: - 1`;
25: }
```

Código en utiltest 35

El parser genera la siguiente estructura para el programa anterior.

```
$VAR1 = [
          {
            'value' => '1',
            'id' => 'verbose',
            'type' => 'var'
          },
          {
            'value' => {
                         'params' => [
                                       '10'
                                     ],
                         'name' => 'rnd',
                         'type' => 'function'
                       },
            'id' => 'iters',
            'type' => 'var'
          },
          {
            'cmds' => [
                        {
                          'elsecmds' => [
                                          {
                                            'outfile' => ':TESTDIR:/test.out',
                                            'capture' => [
                                                           {
                                                             'identifiers' => [
```

```
                                                                    'id2'
                                                                  ],
                                                    'regex' => 'gicorone(.*)'
                                                  }
                                                ],
                                    'args' => '-[la|ltra]',
                                    'type' => 'run',
                                    'expect' => [
                                                  {
                                                    'regex' => 'gicorone',
                                                    'times' => '8'
                                                  }
                                                ],
                                    'program' => 'ls'
                                  },
                                  {
                                    'ifcmds' => [
                                                  {
                                                    'text' => 'captured id2 as \':id2:\'\\n',
                                                    'type' => 'print'
                                                  }
                                                ],
                                    'type' => 'if',
                                    'condition' => ':verbose:'
                                  }
                                ],
                  'ifcmds' => [
                                {
                                  'capture' => [
                                                 {
                                                   'identifiers' => [
                                                                      'id1'
                                                                    ],
                                                   'regex' => 'gicorone(.*)'
                                                 }
                                               ],
                                  'args' => '[-ltr|-l]',
                                  'type' => 'run',
                                  'expect' => [
                                                {
                                                  'regex' => 'gicorone',
                                                  'times' => '6'
                                                }
                                              ],
                                  'program' => 'ls'
                                },
                                {
                                  'ifcmds' => [
                                                {
                                                  'text' => 'captured id1 as \':id1:\'\\n\\n',
                                                  'type' => 'print'
                                                }
                                              ],
                                  'type' => 'if',
                                  'condition' => ':verbose:'
                                }
                              ],
                  'type' => 'if',
                  'condition' => '&rnd(0,3) == 0'
                },
                {
                  'value' => {
                               'expr' => ':iters: - 1',
                               'type' => 'expression'
                             },
                  'id' => 'iters',
                  'type' => 'var'
                }
              ],
```

```
            'type' => 'while',
            'condition' => ':iters: > 0'
          }
        ];
```

Árbol 4. AST del código en utiltest 35

En la estructura se pueden observar la estructura del programa conservando los distintos niveles de anidación de las sentencias a ejecutar.

Evaluador

Aquí es donde el compilador y el intérprete toman rumbos distintos, el compilador suele hacer análisis semántico de la estructura obtenida y trata de realizar optimizaciones para hacer más eficiente el programa conservando la misma funcionalidad, después de eso transforma la estructura en código ensamblador para después generar código máquina para la arquitectura actual. En el caso de los intérpretes se suelen tomar distintos enfoques, algunos basados en el AST y otros basados en generar un lenguaje portable intermedio altamente optimizado llamado ***bytecode***.

La forma de evaluación que utilizará *utiltest* será la de recorrer el AST e ir ejecutando el comportamiento descrito por el programa mientras se recorre la estructura.

La función principal para la evaluación de la estructura se encargará de ir recorriendo el arreglo de sentencias y llamando una función que se encargará de seleccionar y llamar la función adecuada para su evaluación.

Para cada tipo de sentencia <sentencia> se creará una función que podrá evaluarla. Estas funciones serán nombradas "*eval_<sentencia>*" y podrán ser directa o indirectamente recursivas.

```perl
# Evalua un árbol de sintaxis
sub eval_tree($)
{
  my $tree = shift;
  my $rc = SUCCESS;
  foreach my $stmt( @$tree ){
    $rc = eval_stmt($stmt);
  }
  return $rc;
}
```

Código en Perl 7. Implementación de la función que evalúa AST

```perl
# Evalua una sentencia. Llama a la función
# adecuada de acuerdo al tipo de sentencia
sub eval_stmt($){
  my $stmt = shift;
  if ($stmt->{type} eq 'run'){
```

```
    $rc = eval_util($stmt);
  }
  elsif($stmt->{type} eq 'run_sql'){
    $rc = eval_sql($stmt);
  }
  elsif($stmt->{type} eq 'read_file'){
    $rc = eval_file($stmt);
  }
  elsif($stmt->{type} eq 'var'){
    $rc = eval_var($stmt);
  }
  elsif($stmt->{type} eq 'if'){
    $rc = eval_if($stmt);
  }
  elsif($stmt->{type} eq 'while'){
    $rc = eval_while($stmt);
  }
  elsif($stmt->{type} eq 'function'){
    $rc = eval_function($stmt);
  }
  elsif($stmt->{type} eq 'print'){
    $rc = eval_print($stmt);
  }
  elsif($stmt->{type} eq 'unlink'){
    $rc = eval_unlink($stmt);
  }
  elsif($stmt->{type} eq 'diff'){
    $rc = eval_diff($stmt);
  }
  return $rc;
}
```

Código en Perl 8. Implementación del evaluador de sentencias

Para la evaluación de las distintas sentencias también en necesario la creación de algunas funciones auxiliares. Tal vez la más importante de éstas sea la interpolación de cadenas, es decir, la expansión de variables y funciones embebidas dentro de cadenas de caracteres.

El manejo de variables y funciones se implementará utilizando hashes de Perl para almacenarlas. Las variables se irán almacenando en el hash **%**VARS donde la llave será el identificador de la variable y el valor será el valor actual de la variable. En el caso de las funciones, serán definidas en el hash **%**FUNCS donde la llave será el identificador de la función y el valor será una subrutina anónima escrita en Perl.

```
# Función que interpola variables y evalua
# funciones de una cadena
sub expand_line
{
  my $line = shift;
  $line = &expand_vars($line);
  $line = &expand_funcs($line);
  return $line
}

# Expande variables de una cadena
```

```perl
sub expand_vars
{
  my $line = shift;
  # crea un arreglo de variables encontradas
  my @vars = $line =~ /:(\w+):/g;
  foreach (@vars){
    # si existe sustituye la variable por su valor
    if(exists($VARS{$_})){
      $line =~ s/:$_:/$VARS{$_}/;
    } else {
      warn "WARN: variable $_ is not defined\n";
    }
  }
  return $line;
}

# Función recursiva que evalúa funciones dentro de cadenas
sub expand_funcs
{
  my $line = shift;
  # Busca la función más interna
  if (my ($func) = $line =~ /(&\w+\([^\(\)]*\))/){
    my ($name,$args) = $func =~ /&(\w+)\((.*)\)/;
    my $val;
    #Evalúa la función y guarda el valor de retorno
    if(exists($FUNCS{$name})){
      $val = $FUNCS{lc($name)}->(split (/,/,$args));
    } else {
      warn "WARN: function $name is not defined\n";
      return;
    }
    # Sustituye la función por su valor de retorno
    $line =~ s/\Q$func\E/$val/;
    # Llama recursivamente por si hay más funciones a evaluar
    $line = &expand_funcs($line);
  }
  return $line
}
```

Código en Perl 9. Implementación de las funciones para interpolación de variables y funciones

Se deberá crear una función para cada sentencia. Esta función debe ser capaz de interpretar el nodo a evaluar, es decir, conocer la subestructura que le corresponde y traducirla a una serie de pasos a ejecutar.

```perl
sub eval_function($)
{
  my $func = shift;
  my @params;
  foreach my $p (@{$func->{params}}){
    # el parámetro es una función?
    if(ref $p eq 'HASH' and $p->{type} eq 'function'){
      push(@params, &eval_function($p));
    } # es una variable?
    elsif(ref $p eq 'HASH' and $p->{type} eq 'var') {
      push(@params, $VARS{$p->{id}});
    } # es un escalar
    else {
      push(@params, $p);
```

```
    }
  }
  die "Function $func->{name} does not exist\n" unless exists $FUNCS{$func->{name}};
  return $FUNCS{$func->{name}}->(@params);
}

sub eval_var($)
{
  my $var = shift;
  if (ref $var->{value} eq 'HASH'){
    # usar eval en caso de que sea un expresión
    if ($var->{value}->{type} eq 'expression'){
      my $value = &expand_line($var->{value}->{expr});
      $VARS{$var->{id}} = eval $value;
      if($@){
        warn "ERROR: Errors found when evaluating expression \"$value\"\n";
        return FAILURE;
      }
    }
    elsif($var->{value}->{type} eq 'function'){
      $VARS{$var->{id}} = eval_function($var->{value});
    }
  }
  else {
    my $value = &expand_line($var->{value});
    $VARS{$var->{id}} = $value;
  }
  return SUCCESS;
}

sub eval_if($){
  my $if = shift;
  if(eval &expand_line($if->{condition})){
    foreach my $stmt (@{$if->{ifcmds}}){
      eval_stmt($stmt);
    }
  }
  elsif(exists $if->{elsecmds}) {
    foreach my $stmt (@{$if->{elsecmds}}){
      eval_stmt($stmt);
    }
  }
}

sub eval_while($){
  my $while = shift;
  while(eval &expand_line($while->{condition})){
    foreach my $stmt (@{$while->{cmds}}){
      eval_stmt($stmt);
    }
  }
}
```

Código en Perl 10. Implementación de funciones para evaluar distintas sentencias

Conclusiones

A lo largo del documento se ha descrito ampliamente el propósito de utiltest que puede ser resumido en “crear un lenguaje para facilitar la creación de pruebas, su traductor a AST y su evaluador”. Después de la implementación se puede observar que se cumplió el objetivo ya que se pueden crear pruebas con unas cuantas líneas, por ejemplo el Código de utiltest 18 (página 26) está compuesto de únicamente 3 líneas pero es capaz de probar que la salida del comando *ls -l* va de acuerdo a su especificación sin importar en qué directorio se corra mientras la plataforma sea compatible con POSIX. Otro punto importante es que el lenguaje y el intérprete pueden ser expandidos fácilmente para soportar nueva funcionalidad cuando ésta se vaya requiriendo usando la misma teoría descrita en este documento.

Elegí presentar este proyecto porque es de gran utilidad para mi trabajo y porque es en el que he usado más los conocimientos adquiridos en la carrera. En este documento se describe el uso de teoría de distintas asignaturas base de la carrera de Ingeniería en Computación para la creación de utiltest. Entre ellas se encuentran: “lenguajes de programación” para analizar las ventajas de los distintos lenguajes y elegir uno para su implementación; “computación para ingenieros” y “programación avanzada y métodos numéricos” para la implementación del intérprete en un lenguaje de programación; “algoritmos y estructuras de datos” para el diseño de la estructura de datos que contiene al programa parseado y para el entendimiento de las funciones recursivas que implementan al parser; “lenguajes formales y autómatas” para la base de la teoría detrás de un intérprete y las expresiones regulares; “compiladores” para los métodos de elaboración del lexer y el parser; “sistemas operativos” para el entendimiento sobre la ejecución de procesos, uso de shells, manejo de señales, redirecciones, etc.; y “bases de datos” para el soporte para consultas SQL.

En la práctica, *utiltest* está siendo usado actualmente para probar una nueva utilería del producto así como para la automatización de reproducciones de fallas de distintas utilerías para comprobar que realmente fueron corregidas. Miembros del equipo de pruebas en México están empezando a familiarizarse con *utiltest* para el desarrollo de futuras pruebas. También se espera expandir *utiltest* para incorporar recientes cambios que se han hecho a la infraestructura de pruebas del producto.

Glosario

API

Proviene de Application Programming Interface, es un conjunto de funciones, rutinas o herramientas para elaborar aplicaciones.

Bytecode

Código eficiente portable que generan algunos intérpretes.

Diff

Herramienta de UNIX para la comparación de archivos.

Expresión regular

Cadena que describe un patrón de búsqueda.

Golden file

Archivo que contiene la salida esperada de algun programa a probar.

Home

Directorio predeterminado asignado al usuario actual.

ODBC

Acrónimo de Open DataBase Connectivity, es un estándar multiplataforma para la conexión a las bases de datos.

Path

Variable de ambiente que contiene las distintas rutas en las que están localizados los binarios de los programas ejecutables.

PL/SQL

Lenguaje de programación procedural para las bases de datos de Oracle.

POSIX

Es una familia de estándares para el diseño de sistemas operativos basados en UNIX.

QA

Acronimo de Quality Assurance, es el área encargada de la calidad en el software.

RDBMS

Es un software manejador de bases de datos relacionales.

Regresión

Funcionalidad que solía ser correcta pero que deja de ser correcta como efecto secundario tras la introducción de funcionalidad nueva.

Sanity test

Prueba básica para la funcionalidad principal de un programa.

SQL

Lenguaje de consultas y de manejo de datos utilizado principalmente en bases de datos relacionales.

Timestamp

Secuencia de caracteres que describen una fecha con precisión de fracciones de segundo.

Token

En el contexto de compiladores es un conjunto de caracteres que representan una unidad abstracta o un significado.

Warning

Texto que arrojan ciertos programas y compiladores para advertir al programador de los posibles riesgos de la entrada que otorgó.

XLA

API de TimesTen para el manejo directo del log de transacciones la cual permite responder programáticamente a eventos en la base de datos.

Referencias

Alfred V. Aho, R. S. (1986). *Compilers Principles, Techniques, and Tools.* Addison Wesley.

Orwant, J. (1996). *Perl 5 Interactive Course.* Waite Group Press.

http://perldoc.perl.org/perlre.html último acceso Octubre 2014

http://en.wikipedia.org/ último acceso Octubre 2014

Anexos

Anexo 1: Plataformas soportadas por TimesTen

Platforma / Sistema Operativo	Procesador
Linux x86 (64-bit) Oracle Linux 4, 5 and 6 Red Hat Enterprise Linux 4, 5 and 6 SUSE Enterprise Server 10 and 11 MontaVista Linux Carrier Grade Edition 5.0 and 6.0 Asianux 3.0	Intel y AMD
Linux x86 (32-bit) Oracle Linux 4, 5 and 6 Red Hat Enterprise Linux 4, 5 and 6 SUSE Enterprise Server 10 and 11 MontaVista Linux Carrier Grade Edition 5.0 and 6.0 Asianux 3.0	Intel y AMD
Windows x64 (64-bit) Microsoft Windows XP, 2003, 2003 R2, 2008, 2008 R2, Vista, 7, 2012 R2	Intel y AMD
Windows x86 (32-bit) Microsoft Windows XP, 2003, 2003 R2, 2008, Vista, 7	Intel y AMD
Solaris Sparc (64-bit) Oracle Solaris 10 and 11	SPARC
Solaris x86-64 (64-bit) Oracle Solaris 10 and 11	Intel y AMD
IBM AIX (64-bit) AIX 6.1 and 7.1	POWER
IBM AIX (32-bit) - client only AIX 6.1 and 7.1	POWER
Solaris Sparc (32-bit) - client only Oracle Solaris 10	SPARC
Oracle VM for Linux3	x86 y x86-64

Anexo 2: Expresiones regulares de Perl

Metacaracteres

\	Escapa el siguiente metacaracter
^	Representa el inicio de la línea
.	Representa cualquier caracter excepto nueva línea
$	Representa fin de línea
|	Alternación
()	Agrupación
[]	Representa cualquiera de los caracteres entre corchetes

Cuantificadores

*	Representa 0 o más ocurrencias
+	Representa 1 o más ocurrencias
?	Representa 1 o 0 ocurrencias
{n}	Representa exactamente n ocurrencias
{n,}	Representa cuando menos n ocurrencias
{n,m}	Representa desde n hasta m ocurrencias

Clases de caracteres

\w	Representa cualquier carácter alfanumérico y _
\W	Representa cualquier carácter no incluido en \w
\s	Representa cualquier caracter de espacio
\S	Representa cualquier carácter no incluido en \s
\d	Representa cualquier dígito decimal
\D	Representa cualquier carácter no incluido en \d

Anexo 3: Documentación del formato largo de ls

The Long Format

If the **-l** option is given, the following information is displayed for each file: file mode, number of links, owner name, group name, number of bytes in the file, abbreviated month, day-of-month file was last modified, hour file last modified, minute file last modified, and the pathname. In addition, for each directory whose contents are displayed, the total number of 512-byte blocks used by the files in the directory is displayed on a line by itself, immediately before the information for the files in the directory. If the file or directory has extended attributes, the permissions field printed by the **-l** option is followed by a '@' character. Otherwise, if the file or directory has extended security information (such as an access control list), the permissions field printed by the **-l** option is followed by a '+' character.

If the modification time of the file is more than 6 months in the past or future, then the year of the last modification is displayed in place of the hour and minute fields.

If the owner or group names are not a known user or group name, or the **-n** option is given, the numeric ID's are displayed.

If the file is a character special or block special file, the major and minor device numbers for the file are displayed in the size field. If the file is a symbolic link, the pathname of the linked-to file is preceded by ``->''.

The file mode printed under the **-l** option consists of the entry type, owner permissions, and group permissions. The entry type character describes the type of file, as follows:

b	Block special file.
c	Character special file.
d	Directory.
l	Symbolic link.
s	Socket link.
p	FIFO.
-	Regular file.

The next three fields are three characters each: owner permissions, group permissions, and other permissions. Each field has three character positions:

1. If **r**, the file is readable; if **-**, it is not readable.

2. If **w**, the file is writable; if **-**, it is not writable.

3. The first of the following that applies:

   S If in the owner permissions, the file is not exe-

cutable and set-user-ID mode is set. If in the group permissions, the file is not executable and set-group-ID mode is set.

s If in the owner permissions, the file is executable and set-user-ID mode is set. If in the group permissions, the file is executable and set-group-ID mode is set.

x The file is executable or the directory is searchable.

- The file is neither readable, writable, executable, nor set-user-ID nor set-group-ID mode, nor sticky. (See below.)

These next two apply only to the third character in the last group (other permissions).

T The sticky bit is set (mode 1000), but not execute or search permission. (See chmod(1) or sticky(8).)

t The sticky bit is set (mode 1000), and is searchable or executable. (See chmod(1) or sticky(8).)