Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Escuela de Ciencias y Sistemas

Organización de Lenguajes y Compiladores 2

Escuela de vacaciones de junio de 2017

Catedráticos: Ing. Bayron López

Tutores académicos: Enio González

SBScript

Práctica 1 - Un intérprete básico

1. ÍNDICE DE CONTENIDO

Contenido

2. OBJETIVOS

2.1. OBJETIVO GENERAL

Aplicar los conocimientos del curso de Organización de Lenguajes y Compiladores 2 en la creación de soluciones de software.

2.2. OBJETIVOS ESPECÍFICOS

- Aplicar los conceptos de análisis léxico, sintáctico y semántico con herramientas específicas para desarrollar un intérprete que ejecute instrucciones de un lenguaje de programación de alto nivel.
- Aplicar las técnicas adecuadas para la recuperación y reporte de errores producto del proceso de compilación y ejecución.
- Implementar un módulo para realizar gráficas de diversas estructuras lógicas relacionadas con el código de alto nivel del lenguaje a interpretar.

3. DESCRIPCIÓN DEL PROYECTO

Entre los conceptos más importantes de la teoría de compiladores, el AST (Abstract Syntax Tree) juega un papel fundamental; ante la necesidad de fortalecer este concepto se ha diseñado el lenguaje SBScript. SBScript es un lenguaje de programación que cuenta con sentencias básicas que se pueden encontrar en cualquier otro lenguaje, pero que incorpora entre sus funciones primitivas (propias del lenguaje) la posibilidad de crear gráficas de la estructura de sus propias instrucciones.

Todo lenguaje debe contar con un IDE, por eso se la ha encargado a usted, como estudiante del curso de

Compiladores 2, la creación de un intérprete para para SBScript. El IDE solicitado facilitará entre otras tareas la creación, el almacenamiento y la ejecución de los programas contenidos en archivos de texto que tendrán la extensión **sbs**; además soportará múltiples pestañas para poder trabajar con varios archivos a la vez. El IDE también contará con la capacidad de analizar e interpretar el contenido de cualquiera de los archivos abiertos; mostrará las distintas salidas (texto e imágenes) generadas y los reportes de errores generados como producto del análisis y ejecución de determinado archivo. A continuación se muestra una sugerencia de interfaz para el IDE solicitado.

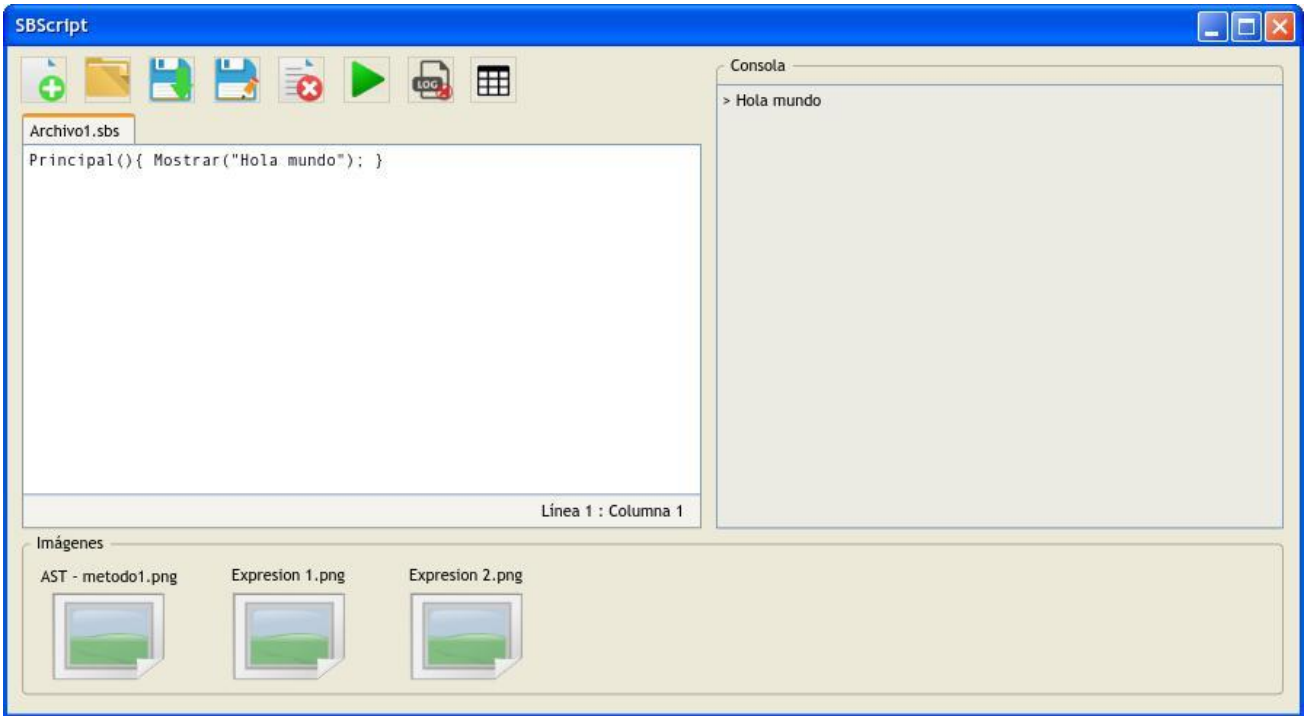


Ilustración 1 - Interfaz gráfica sugerida

Los botones (o menú) del IDE deberán contar con las siguientes funcionalidades:

- **Nuevo:** Creará una nueva pestaña en blanco en el editor.
- **Abrir:** Abrirá un cuadro de diálogo para seleccionar un archivo con extensión sbs y mostrará su contenido en una nueva pestaña.
- **Guardar:** Guardará el contenido de la pestaña actual en un archivo, si la pestaña es una pestaña nueva, se deberá mostrar un cuadro de diálogo para seleccionar la ubicación y nombre del archivo.
- **Guardar como:** Esta opción mostrará directamente un cuadro de diálogo para seleccionar la ubicación y el nombre del archivo a guardar.
- **Cerrar pestaña:** Esta opción eliminará la pestaña de la interfaz gráfica.
- **Ejecutar archivo:** Ejecutará el contenido de la pestaña actual.
- **Ver reporte de errores:** Enviará al usuario a la sección o al archivo de reporte de errores.
- **Abrir álbum:** Abrirá la carpeta del sistema en donde están todas las imágenes generadas durante la ejecución del programa.

La funcionalidad del IDE de SBScript se ejemplifica en la siguiente imagen, en donde, producto del análisis de un lote de archivos con extensión sbs, se obtiene una salida en consola, un repositorio de imágenes y un

reporte de los posibles errores existentes.

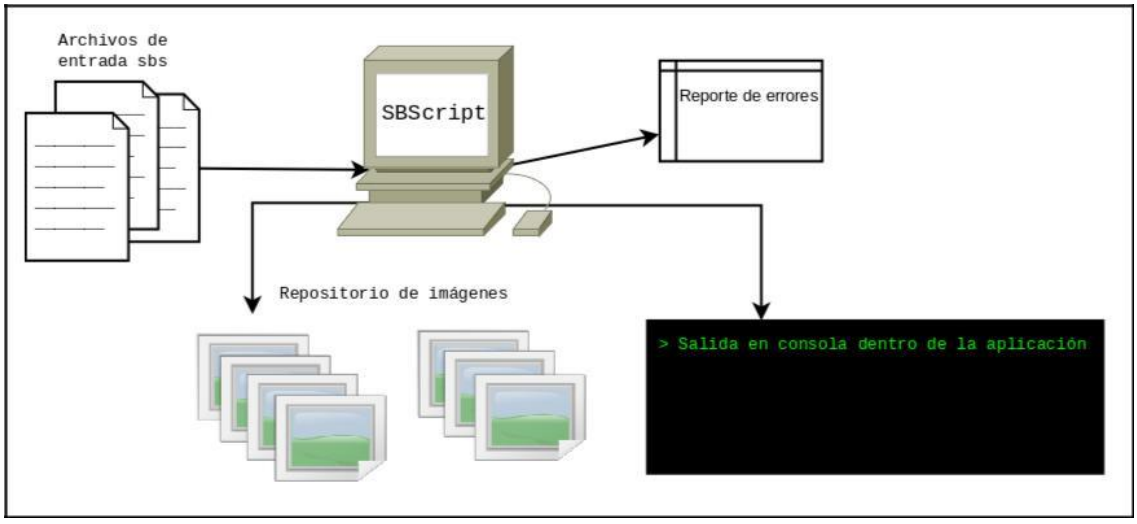


Ilustración 2 - Esquema de funcionamiento SBScript

Cómo última parte de esta sección se propone una arquitectura para SBScript, en donde se establezca un sistema de paquetes para agrupar los componentes de software (clases) según su funcionalidad. En los paquetes sugeridos podemos resaltar:

- Un paquete para la fase de análisis, que incluirá todo lo relacionado con Gold Parser
- Un paquete de utilidades que tendrá todo lo relacionado a manejo de archivos, gestión de errores y generación de reportes
- Un paquete para agrupar todo lo relacionado al intérprete, que a su vez contendrá cuántos sub paquetes considere necesario el estudiante para realizar todo lo referente a la ejecución de las instrucciones
- Un paquete para todo lo referente a interfaz gráfica.

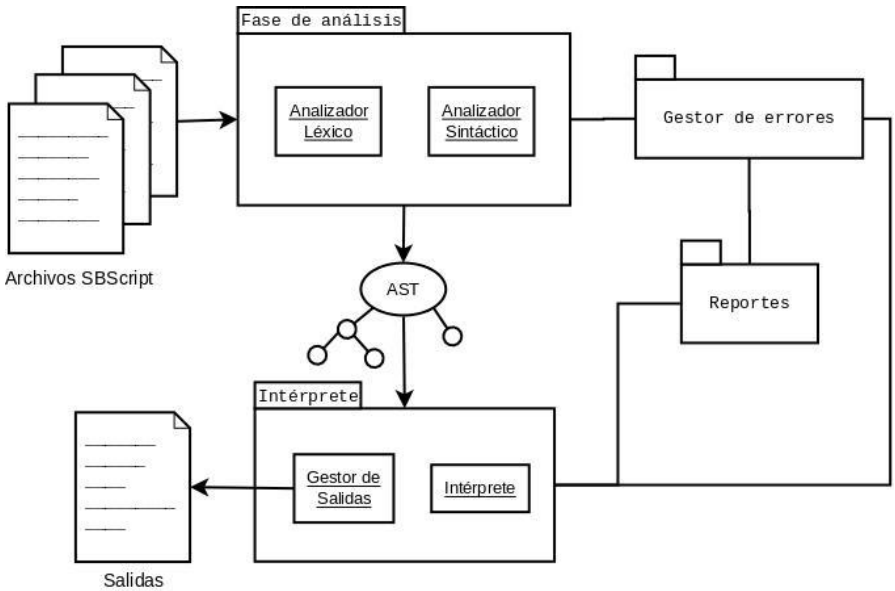


Ilustración 3 - Arquitectura propuesta para la construcción de un intérprete

4. EL LENGUAJE SBSCRIPT

SBScript es un lenguaje que soporta:

- Comentarios.
- Sensibilidad entre mayúsculas y minúsculas.
- Datos de tipo numérico (no hay diferencia entre entero o decimal), booleano y cadena.
- Realización de operaciones aritméticas, lógicas y relacionales bajo un estricto sistema de tipos.
- Declaración de variables (globales y locales).
- Definición de funciones con un tipo asociado a su retorno y un conjunto de parámetros.
- Sobrecarga de funciones por medio de sus parámetros.
- Llamadas a funciones (incluyendo recursividad directa o indirecta).
- Uso de sentencias propias del lenguaje.
- Ciclos y sentencias de control de flujo.

4.1. ENCABEZADO

El inicio de todo archivo SBScript cuenta con una sección de declaraciones en donde se declararán aspectos generales que a continuación se definen y ejemplifican:

- Se pueden incluir otros archivos SBScript con la directiva *'Incluye'* acompañada de un nombre de archivo sbs, que como su nombre lo indica, se encargará de incluir el contenido de dicho archivo sbs adjunto al contenido del archivo que se está analizando (como que todos fueran un solo archivo). Si el archivo a incluir no existe debe reportarse un error de tipo general.
- Se puede definir un valor de incerteza para realizar comparaciones entre valores numéricos (ver sección Expresiones Relacionales) por medio de la directiva 'Define' acompañada de un valor numérico. Si no se define valor para la incerteza se tomará un valor por defecto de **0.5**. La incerteza es independiente para cada archivo.
- Se puede definir una ruta específica para guardar los reportes generados por la aplicación por medio de la directiva (ver sección Reportes) por medio de la directiva 'Define' acompañada de una cadena.

  Si la ruta indicada no existe deberá reportarse un error de tipo general y utilizarse una ruta por defecto definida por el desarrollador. La ruta de salida definida es independiente para cada archivo.

Ejemplo de un encabezado:

```
Incluye aritmeticas.sbs
Incluye relacionales.sbs
Define 0.0002
Define "/home/user1/pictures/reports"
```

Este ejemplo incluiría el contenido (ya analizado) de los archivos **aritmeticas.sbs** y **relacionales.sbs**; utilizaría una incerteza de **0.0002** y guardaría todo el contenido generado por los reportes en la carpeta **/home/user1/pictures/reports**.

4.2. COMENTARIOS

En SBScript se permite escribir comentarios. Los comentarios de una sola línea empezarán con una almohadilla **#** y terminarán con un salto de línea; los comentarios de múltiples líneas empiezan con almohadilla **#** y un asterístico *****, terminarán con un ***** y una almohadilla **#**.

4.3. TIPOS DE DATOS

Para las variables se permiten los tipos de datos indicados en la siguiente tabla, mientras que para las funciones se permiten los mismos tipos de datos y adicional a ello se admite también el tipo vacío, cuya palabra reservada será **Void** que indicará que una función no tiene tipo de retorno asignado. En las cadenas se permiten los caracteres de escape "\n" y "\t" para mostrar saltos de línea y tabulaciones respectivamente.

Tipo de dato	**Ejemplos**
Number	90 -120.210 20.06 -30
Bool	true false
String	"Esto es una cadena" "\n"

4.4. EXPRESIONES EN GENERAL

- Al momento de que un tipo booleano se vea implicado en una operación numérica permitida, este debe interpretarse como un 0 si su valor es falso, y como un 1 si su valor es verdadero.

Operación de ejemplo	**Resultado**
true + 5.90	6.90
90.206 * false	0

- Para realizar una concatenación entre una cadena y un valor booelano es necesario convertir los valores booleanos a su equivalente en texto, es decir, sí el valor booleano es verdadero, en la cadena resultante de la concatenación se debe agregar el caracter "1"; caso contrario el valor booleano sea falso, el texto a concatenar será el caracter "0".

Operación de ejemplo	**Resultado**
false + " hola mundo"	"0 hola mundo"
"hola mundo" + true	"hola mundo 1"

- En las concatenaciones donde intervienen datos de tipo *Number* se debe obtener el texto que representa el valor numérico y agregarlo a la cadena resultante.

Operación de ejemplo	**Resultado**
"La suma de 5 + 3 es: " + 8	"La suma de 5 + 3 es: 8"
3.1416 + " es el valor de Pi"	"3.1416 es el valor de Pi"

4.5. EXPRESIONES ARITMÉTICAS

Dentro de SBScript es posible escribir expresiones aritméticas con los siguientes operadores. Cada una de las siguientes matrices de tipos tiene en las filas el tipo de dato del operando izquierdo y en los encabezados el tipo de dato del operando derecho, en sus celdas internas se almacena el tipo del resultado. El operador - unario, solo se aplica a valores de tipo *Number*, el resto de tipos son errores.

+		Boolean	Number	String
	Boolean	OR lógica	Suma	Concatenación
	Number	Suma	Suma	Concatenación
	String	Concatenación	Concatenación	Concatenación

-		Boolean	Number	String
	Boolean	Error	Resta	Error
	Number	Resta	Resta	Error
	String	Error	Error	Error

*		Boolean	Number	String
	Boolean	AND lógica	Multiplicación	Error
	Number	Multiplicación	Multiplicación	Error
	String	Error	Error	Error

/		Boolean	Number	String
	Boolean	Error	División	Error
	Number	División	División	Error
	String	Error	Error	Error

%		Boolean	Number	String
	Boolean	Error	División	Error
	Number	División	División	Error
	String	Error	Error	Error

^		Boolean	Number	String
	Boolean	Error	Potencia	Error
	Number	Potencia	Potencia	Error
	String	Error	Error	Error

4.6. EXPRESIONES RELACIONALES

Además de las expresiones aritméticas, también es posible definir operaciones relaciones, cuyo valor siempre será un valor booleano. Estas operaciones solo son aplicables entre expresiones del mismo tipo, de lo contrario se debe reportar error. Entre datos numéricos, las comparaciones se comportan de manera natural, haciendo la comparación que su nombre indica; Para los booleanos true = 1 y false = 0.

Igual	Diferente	Menor	Mayor	Menor o igual	Mayor o igual
==	!=	<	>	<=	>=

Para las cadenas, las comparaciones deben realizarse siguiendo el orden lexicográfico de cada uno de los caracteres que componen las cadenas a comparar. Por ejemplo:

Cadena 1	**Cadena 2**	**Resultado**
“Agua”	“Aguacate”	Cadena 2 es mayor que Cadena 1
La cadena es idéntica en sus primeros 4 caracteres, pero como la cadena 2 aún tiene caracteres el resultado sería que la cadena 2 es “mayor que” la cadena 1.		
“compiladores 2”	“compiladores2”	Cadena 1 es menor que Cadena 2
En este caso la cadena 1 es menor que la cadena 2 porque el ascii asociado al espacio en blanco (32) es menor que el valor del ascii asociado al dígito ‘2’ (50).		
“japón”	“japón”	Cadena 2 es igual que Cadena 1
Las cadenas son idénticas.		

Operador semejantes ~ (el código ascii del operador es 126)

Al principio de todo archivo SBScript es posible definir un grado de incerteza, que será un valor que servirá para utilizar este operador (ver sección de encabezados), dicho grado de incerteza se aplicará para comparar valores numéricos, como por ejemplo:

10.54 ~ 10.50

Con un valor de incerteza de **0.5**, el resultado de esta comparación sería **true**. Porque el valor absoluto de la diferencia que existe entre los dos valores (|10.54 - 10.50| = **0.04**) es menor o igual que el valor de incerteza declarada.

Si en otro caso la incerteza fuese de **0.01**, el resultado de la misma comparación sería false. Porque el valor absoluto de la diferencia entre los valores sería mayor que la incerteza declarada.

El mismo operador “~” será utilizado para comparar cadenas, dicha comparación ignorará los espacios en blanco al inicio y al final de las cadenas, tampoco distinguirá entre mayúsculas y minúsculas, bajo estas condiciones se realizará la comparación de las cadenas (la incerteza NO tiene nada que ver en las comparaciones entre cadenas).

Por ejemplo, la siguiente comparación sería verdadera:

" correo@mail.com" ~ "Correo@mail.com "

Se ignoraría el hecho de que una de las dos palabras inicia con C mayúscula y la otra con c minúscula, y además ignoraría los espacios en blando al final de la cadena de la derecha, por lo tanto las cadenas son semejantes.

Otro ejemplo, la siguiente comparación sería falsa:

"Compi2" ~ "Compi 2"

A pesar de que las dos cadenas cuentan con los mismo caracteres, los espacios que se encuentre en medio de cada cadena no deben ser ignorados, por lo tanto estas cadenas no cumplen con ser semejantes.

4.7. EXPRESIONES LÓGICAS

Por último en las expresiones, también es posible utilizar expresiones de índole lógico, para este tipo de expresiones los operandos siempre tienen que ser de tipo booleano y el resultado de igual forma siempre será un valor booleano, de lo contrario será error.

And (y lógico)	Or (o lógica)	Xor	Not (negación lógica)
&&	||	|&	!

4.8. PRECEDENCIA DE OPERADORES Y SÍMBOLOS DE AGRUPACIÓN

En la siguiente tabla se definirá la precedencia y asociatividad para cada uno de los operadores definidos en esta sección de expresiones, ordenados de menor a mayor precedencia. Los paréntesis **( )** son los únicos símbolos que se utilizarán para agrupar operaciones y "romper" la precedencia normal de las operaciones.

Símbolo	Precedencia	Asociatividad
+ -	1	Izquierda
*** / %**	2	Izquierda
^	3	Derecha
- (unario)	4	No aplica
== != < > <= >= ~	5	No aplica
||	6	Izquierda
!&	7	Izquierda
&&	8	Izquierda
!	9	Izquierda
( )	10	No aplica

4.10. IDENTIFICADORES

Un identificador estará definido como una sucesión de caracteres que inicia con una letra y seguido de ella pueden haber cero o más caracteres, entre letras, dígitos o guiones bajos.

Nota: A partir de esta sección del enunciado los elementos encerrados entre corchetes **[ ]** y escritos con un color gris indican opcionalidad, es decir que, dichos elementos pueden o no venir dentro de la sintaxis de SBScript.

4.11. DECLARACIÓN DE VARIABLES (GLOBALES O LOCALES)

Para la declaración de variables se utilizará una sintaxis en donde primero se escriba el tipo de la variable y luego una lista de identificadores separados por coma, para finalizar siempre con un punto y coma (;), de manera opcional se podrá realizar una asignación inicial anteponiendo al punto y coma un signo igual (=) seguido de una expresión cuyo valor será asignado directamente a cada una de las variables definidas en la lista de identificadores. Las variables globales son aquellas declaradas fuera del cuerpo de las funciones y son accesibles desde cualquier función. Mientras que las variables locales son accesibles únicamente dentro del ámbito donde fueron declaradas.

Ejemplo de sintaxis:

<TIPO> <LISTA_ID> [= <EXPRESION>]**;**

```
Number a, b, c = 45.8;
String str1 = hola("mundo");
```

4.12. ASIGNACIONES

Las asignaciones siguen una sintaxis básica; Primeramente el identificador de la variable sobre la que se realiza la asignación, seguido del signo igual (=), luego la expresión que representa el nuevo valor para la variable y por último punto y coma (;).

Ejemplo de sintaxis:

<ID> = <EXPRESION>;

```
a = 4 * factorial(5);
str1 = Saludame("Ana");
```

Durante la asignación de una expresión a una variable el valor de dicha expresión puede ser convertido de manera implícita siguiendo la siguiente convención de tipos.

Destino	Tipo de la expresión	Acción de conversión (casteo implícito)
String	String	Sin acción
	Number	El número se convierte a cadena
	Boolean	La cadena almacenará "1" si es true y "0" si es false
Number	String	Error en la asignación
	Number	Sin acción
	Boolean	Si es true asigna un 1, de lo contrario asigna un 0
Boolean	String	Error en la asignación
	Number	Error en la asignación
	Boolean	Sin acción

4.13. FUNCIONES

Una función es una subrutina de código que se identifica con un nombre y que puede contar con parámetros (y un tipo para su retorno). Para SBScript las funciones serán declarados definiendo primero su tipo, luego su identificador, seguido una lista de parámetros (que puede no venir) delimitada por paréntesis. Cada parámetro estará compuesto por su tipo, seguido de su identificador y cada uno de ellos estará separado del otro por una coma. Después de la definición de la función se declarará el cuerpo de la misma, formado por una lista de instrucciones delimitadas por un juego de llaves **{ }** .

Ejemplo de sintaxis:

```
<TIPO> <ID> ( [ <PARAMETROS> ] ){
        <INSTRUCCIONES>
}
#Ejemplo de declaración de una función
String Saludame(String nombre){
        Retorno “Hola ” + nombre + “!!!”;
}
```

Las funciones pueden ser sobrecargadas, lo que significa que pueden haber dos o más funciones con el mismo nombre, pero con distinta “llave”. La llave que identifica a una función de otra serán los tipos y/o cantidad de sus parámetros. Por ejemplo:

```
Number Funcion1(Number p1) { · · · }
Number Funcion1(String p1) { · · · }
Number Funcion1(String p1, Number n1) { · · · }
Number Funcion1(String p1, Boolean n1) { · · · }
```

Estas declaraciones son permitidas en el lenguaje ya que a pesar de que las funciones cuentan con un nombre idéntico entre sí, tienen su identidad definida acorde a los tipos de sus parámetros.

4.14. FUNCIÓN PRINCIPAL

El función principal es la subrutina que arrancará las acciones del intérprete, esta subrutina tendrá siempre la misma estructura.

Ejemplo de sintaxis:

```
Principal(){
        <INSTRUCCIONES>
}
#Ejemplo de declaración de una función principal
Principal(){
        IniciaCalificacion();
}
```

4.15. LLAMADAS A FUNCIONES

Una llamada a función se realiza escribiendo el identificador de la función; seguido de los valores que tomarán los parámetros para dicha llamada, separados por coma y envueltos en un juego de paréntesis; finalizando con un punto y coma (;).

Ejemplo de sintaxis:

```
<ID>( [ <LISTA_EXPRESIONES> ] );
#Ejemplos de llamadas a funciones válidas son:
IniciaCalificacion();
Saludame("Julia");
```

4.16. RETORNO

Esta sentencia finalizará la ejecución de la función y devolverá el valor indicado por la expresión que se encuentra adyacente a ella; Si durante la ejecución del código se encuentra la palabra 'Retorno' sin ninguna expresión asociada, se terminará la ejecución del cuerpo de la función. Esta sentencia inicia con la palabra reservada '*Retorno*' luego, puede o no venir una expresión y finaliza con un punto y coma (;).

Ejemplo de sintaxis:

```
Retorno [ <EXPRESION> ];
#Ejemplo de un retorno válido
Retorno a ^ 2;
```

4.17. CONTROL DE FLUJO SI

Esta instrucción inicia con la palabra reservada *'Si'*, seguida de una condición encerrada entre paréntesis y que cuenta con un cuerpo de instrucciones que se realizarán en caso de que la condición sea verdadera. Además cuenta de manera opcional con un cuerpo de instrucciones que se ejecutará en caso la condición no se cumpla, este cuerpo de instrucciones viene seguido del primer conjunto de sentencias separado únicamente por la palabra reservada *'Sino'*, este cuerpo de instrucciones también viene delimitado por un juego de llaves ({ }).

Ejemplo de sintaxis:

```
Si( <EXPRESION> ){
    <INSTRUCCIONES>
}
[ Sino {
        <INSTRUCCIONES>
  } ]
#Ejemplo de un Si-Sino válido Si(a <
max){
    Mostrar("'a' es menor que el valor máximo.");
}Sino{
    Mostrar("'a' ha llegado al máximo.");
}
```

4.18. CONTROL DE FLUJO SELECCIONA

Este control de flujo evalúa una expresión y compara el resultado de dicha expresión con cada uno de los valores definidos en los casos fijos del cuerpo de dicha sentencia, cada caso cuenta con un valor fijo y su correspondiente cuerpo de instrucciones que ejecutará toda vez el valor obtenido producto de haber evaluado la expresión sea igual al valor fijo de dicho caso.

Su sintaxis inicia con la palabra reservada 'Selecciona' seguido de la expresión a evaluar encerrada entre paréntesis; a continuación inicia la lista de casos, cada caso es una pareja de valor puntual, dos puntos (:), sentencias delimitadas por su propio juego de llaves. Los tipos de datos permitidos en este control de flujo son solamente *Strings* y *Number*, que a su vez son mutuamente excluyentes, es decir: Si deseo SELECCIONAR una expresión de tipo *String* los valores para los CASOS deben ser valores de tipo *String*, de igual forma sucedería con el tipo *Number*.

El comportamiento de este control de flujo pregunta cuál es la condición que se cumple y de ahí en más ejecuta todas las sentencias de todos los casos (sin incluir el cuerpo asociado al bloque *'Defecto'*) hasta encontrar una sentencia '*Detener'*. Si no se cumple ninguna de las condiciones se debe ejecutar el bloque de sentencias asociado a la palabra reservada *'Defecto'*. Para ver el flujo puede consultar la siguiente imagen: https://drive.google.com/file/d/0B9RsfIlp1lEYcDBXT0Q3cVpLRlU/view?usp=sharing

Ejemplo de sintaxis:

```
Selecciona ( <EXPRESION> )
        <VALOR_1> : { <INSTRUCCIONES_CASO_1> }
        [ <VALOR_N> : { <INSTRUCCIONES_N> } ]
        [ Defecto : { <INSTRUCCIONES_DEF> } ]
#Ejemplo de Selecciona, *val1 es una variable numérica
Selecciona( val1 * 2 )
   10 : { Mostrar("El doble de 'val1' es 10"); }
   100: { Mostrar("El doble de 'val1' es 100"); }
   Defecto : { Mostrar("No sé cuánto es el doble de 'val1'"); }
```

4.19. CICLO PARA

Este ciclo inicia con la palabra reservada *'Para'* consta de una declaración inicial de una variable de tipo *Number*; un punto y coma; una condición para mantenerse en el ciclo; un incremento (++) o decremento (--) que al final de cada iteración realizada afectará directamente a la variable declarada al inicio de este ciclo y un cuerpo de sentencias a ejecutar siempre y cuando la condición sea verdadera. El ciclo de cada iteración será: Evaluar la condición; si es verdadera, ejecutar cuerpo y realizar incremento o decremento, regresar a evaluar la condición; si es falsa, salir del ciclo.

Ejemplo de sintaxis:

```
Para (Number <ID> = <EXPRESION> ; <EXPRESION> ; <OP*> ){
    <INSTRUCCIONES>
}
#Ejemplo de un ciclo Para, *<OP> puede ser ++ o --
Para(Number i=1; i < 11; ++){
    Mostrar("5 x {0} = {1}s", i, 5*i);
}
```

4.20. CICLO HASTA

Este ciclo consta de una estructura que inicia con la palabra reservada *'Hasta'*, seguido de una condición (envuelta entre paréntesis) y al final constará con un cuerpo de sentencias (delimitadas por un juego de llaves { }) que se ejecutarán toda vez la condición del ciclo sea falsa, o dicho de otra forma, las instrucciones se ejecutarán HASTA que llegue el momento en que la condición se vuelva verdadera.

Ejemplo de sintaxis:

```
Hasta ( <EXPRESION> ){
    <INSTRUCCIONES>
}
#Ejemplo de un ciclo Hasta
Hasta(i == 10){
    Si(i>10){
        i = i - 1;
        Mostrar("Decrementando i, {0}", i);
    }Sino{
        i = i + 1;
        Mostrar("Incrementando i, {0}", i);
    }
}
```

4.21. CICLO MIENTRAS

Este ciclo consta de una estructura que inicia con la palabra reservada *'Mientras'*, seguido de una condición (envuelta entre paréntesis) y al final constará con un cuerpo de sentencias (delimitadas por un juego de llaves { }) que se ejecutarán toda vez la condición del ciclo sea verdadera.

Ejemplo de sintaxis:

```
Mientras( <EXPRESION> ){
    <INSTRUCCIONES>
}
#Ejemplo de un ciclo Mientras
Mientras(hayVida){
    Mostrar("Hay esperanza de ganar Compi2");
}
```

4.22. SENTENCIA DETENER

Esta sentencia es aplicable toda vez se encuentre dentro del cuerpo de una sentencia ***Mientras***, ***Hasta***, ***Para*** o dentro del cuerpo de un caso específico de una sentencia ***Selecciona***. Al encontrarla la ejecución deberá detenerse y regresar el foco de control al nivel superior de la sentencia que ha sido detenida.

Ejemplo de sintaxis:

```
#Puede venir en un Mientras, Hasta, Para o Selecciona
Detener;
```

4.23. SENTENCIA CONTINUAR

Esta sentencia es aplicable toda vez se encuentre dentro del cuerpo de una sentencia ***Mientras***, ***Hasta*** o ***Para***. Al encontrar esta sentencia dentro de un ciclo se detendrá la ejecución del mismo y saltará directamente a evaluar la condición para la siguiente iteración; en el caso particular de la sentencia ***Para***, se deberá ejecutar el incremento o decremento antes de saltar a evaluar la condición para la próxima iteración del ciclo.

Ejemplo de sintaxis:

```
#Solo puede venir en un Mientas, Hasta o Para
Continuar;
```

4.24. FUNCIÓN MOSTRAR

La función '*Mostrar*' es una función polimórfica de SBScript que se encargará de imprimir en consola la lista de expresiones que reciba como parámetros colocando dichos parámetros bajo el formato de una máscara determinada por un *String*. En dicha máscara, cada parámetro está asociado con los caracteres {#}, donde #, es el índice del parámetro que acompaña al formato. Los caracteres "{#}" serán sustituidos por el valor del parámetro que corresponda, los índices inician en 0.

Ejemplo de sintaxis:

```
Mostrar ( <STRING> [ , <EXPRESIONES> ] );
#Ejemplos válidos de la función Mostrar son:

Mostrar("Curso de Compiladores 2\n");
>Curso de Compiladores 2

Mostrar("Fecha: {0} de {1} de {2}\n", 24, "Enero", 2017);
>Fecha: 24 de Enero de 2017

Mostrar("{0} es ingeniero, {0} no es {1}", "Luis", "doctor");
>Luis es ingeniero, Luis no es doctor
```

4.26. FUNCIÓN DIBUJARAST

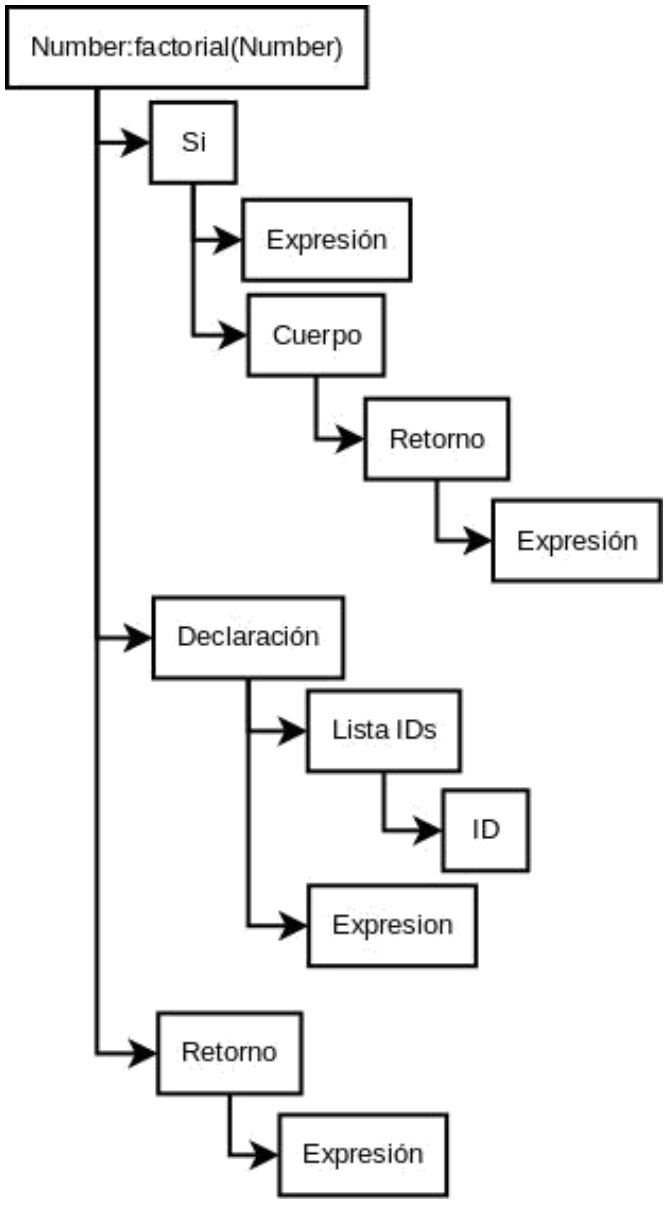
La función '*DibujarAST*' es una función de SBScript que se encargará de dibujar el AST del cuerpo de una función, si existen varios funciones con el mismo nombre se generarán, tantas imágenes como funciones con determinado nombre existan, agregando al nombre del método, su "llave" de parámetros; En esta imagen

NO se deberán dibujar los nodos de expresiones, en su lugar se colocarán nodos para representar la existencia de una expresión de forma resumida.

Ejemplo de sintaxis:

```
#El ejemplo de esta función se muestra en la siguiente tabla
DibujarAST( <ID> );
```

Función, **factorial**	Imagen generada por **DibujarAST(factorial)**

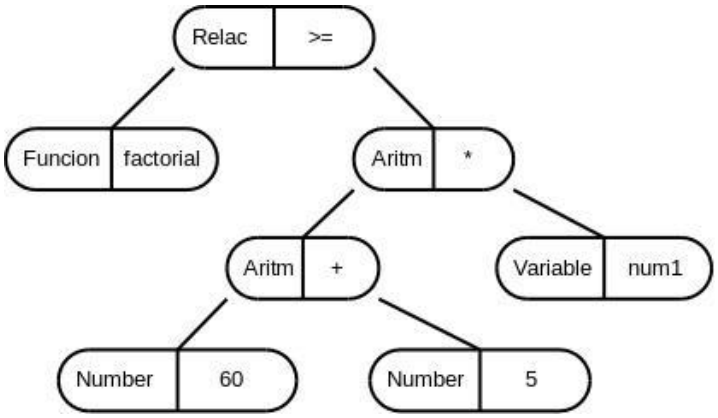
Number factorial(Number n){
Si(n <= 0){
Retorno 1;
}
Number res = n*factorial(n-1);
Retorno res;
}
 	
#Generaría la imagen
#**AST_factorial_Number.png**
	

4.27. FUNCIÓN DIBUJAREXP

La función *'DibujaEXP'* crea una imagen del AST asociado a una expresión en particular, incluyendo todos los tipos de operaciones y los datos más específicos, es decir, valores puntuales *Number*, *String*, *Boolean*, Identificadores y también llamadas a funciones (sin parámetros).

Ejemplo de sintaxis:

```
DibujarEXP( <EXPRESION> );
#Ejemplo del uso de la función DibujarEXP
DibujarEXP(factorial(5) >= (60+5)*num1);
#Generaría una imagen similar a la siguiente
```



5. REPORTES

5.1. REPORTE DE ERRORES

Todo compilador o intérprete es mucho más útil si tiene un reporte detallado de la mayor cantidad de errores posibles. Por esta razón se debe desarrollar un reporte de errores que tenga, para cada error como mínimo:

- Tipo de error (léxicos, sintácticos, semánticos o de tiempo de ejecución)
- Descripción detallada (que describa el error y/o la causa del mismo)
- Ubicación (que idealmente debería de ser, archivo, línea y columna del error)

En la siguiente imagen encontrarán un ejemplo de cómo debe ser el reporte de errores, para el reporte de errores **es obligatorio mostrar la fecha y hora** en la que este fue creado.

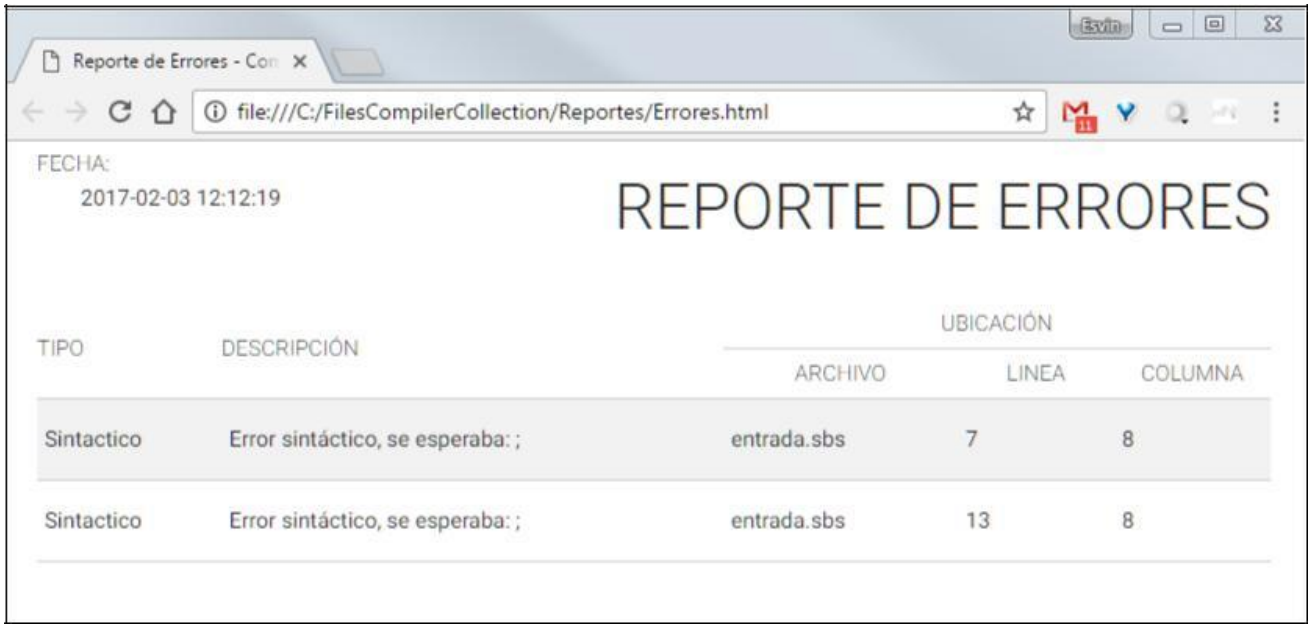


Ilustración 4 - Ejemplo de reporte de errores

Este reporte puede ser creado utilizando HTML (se debe tener un acceso directo desde la aplicación para abrir el reporte en el navegador web del sistema) o puede ser incorporado en una tabla dentro de la aplicación.

5.2. SALIDA EN CONSOLA

La salida en consola debe mostrarse en un área específica dentro de la aplicación, en ella se mostrarán las salidas impresas con la función **Mostrar** del lenguaje. Se recomienda utilizar un tamaño de fuente adecuado, y mejor aún sería que adicional a un tamaño correcto se utilice una fuente de tipo Monospace para poder observar los resultados de forma más clara.

5.3. IMÁGENES

Producto de la ejecución de las funciones DibujarAST y DibujarEXP se crean imágenes. Todas estas imágenes deben ser almacenadas en una ubicación previamente determinada. La ubicación de las imágenes puede ser definida en el encabezado de cada uno de los archivos SBS, o puede ser una ubicación por defecto. Cada imagen generada debe tener un nombre con un formato predeterminado; para las imágenes creadas por DibujarAST, el formato del nombre será: **AST_<ID>.png***; Mientras que para las imágenes creadas por DibujarEXP será únicamente necesario utilizar el prefijo EXP y un correlativo, quedando de la siguiente manera: **EXP_<NUM>.png***.

****El formato de las imágenes generadas quedará a discreción del estudiante.*

6. CONSIDERACIONES FINALES

6.1. EJEMPLOS

En la siguiente carpeta de Dropbox se encuentran algunos ejemplos de archivos de entrada para el lenguaje SBScript. Enlace: https://drive.google.com/file/d/0B9Rsfllp1lEYZFZEbTIxeGt6WVk/view?usp=sharing

6.2. HERRAMIENTAS A UTILIZAR

Para el desarrollo de esta práctica se utilizarán las siguientes herramientas.

- El lenguaje a utilizar es **Visual Basic**.
- El IDE a utilizar es **Visual Studio** en su versión **2012** o superior.
- Para la generación de imágenes se **recomienda** utilizar la herramienta Graphviz.
- Para todo lo que concierne a análisis léxico y/o sintáctico se debe utilizar Gold Parser

6.3. RESTRICCIONES

Para el correcto desarrollo de esta práctica deben respetarse los siguientes lineamientos

- Dudas respecto al enunciado deben hacerse mediante el grupo proporcionado, o en los días de laboratorio.
- Copias parciales o totales de proyectos tendrán una nota de 0 puntos y los estudiantes serán reportados a escuela de ciencias y sistemas.
- NO habrá prórroga de ningún tipo.
- La calificación se realizará a partir de los entregables enviados por el estudiante.

6.4. ENTREGABLES

Los entregables para esta práctica son:

- Aplicación funcional
- Código fuente
- Gramáticas utilizadas

6.5. REQUERIMIENTOS MÍNIMOS

Los requerimientos mínimos para tener derecho a la calificación en esta práctica son:

IDE (Interfaz Gráfica)

- Multipestañas
- Abrir
- Guardar
- Guardar como
- Salida de consola
- Acceso directo al reporte de errores (en el navegador si es HTML)

Intérprete

Encabezado

- Incerteza
- Ruta

Expresiones

- Todas las expresiones aritméticas, relacionales y lógicas
- Valores puntuales
- Acceso a variables
- Retorno de funciones

Declaración de variables

- Variables globales
- Variables locales

Asignaciones

- Casteo implícito

Funciones

- Método principal
- Funciones sin parámetros

Sentencias

- Llamadas a funciones
- Retorno (vacío y con expresión)
- Control de flujo SI
- Control de flujo SELECCIONA
- Ciclo PARA
- Ciclo MIENTRAS
- Función MOSTRAR
- Función DIBUJARAST
- Función DIBUJAREXP

Reportes

- Reporte de errores
- Salida en consola
- Imágenes

6.6. FECHA DE ENTREGA

La práctica se entregará el día 08 de junio de 2017 antes de las 19:00 horas.