**Proyecto - Fase #1 - Analizador léxico**

**Mini-C# y Mini-Java**

Fecha de entrega: 27-28 de agosto, 2020, 19:00 horas

**El objetivo**

En el primer proyecto de programación, ustedes iniciarán su compilador con la aplicación del análisis léxico. Para la primera tarea del front-end, crearán un escáner para el lenguaje de programación asignado. El escáner irá reconociendo los tokens en el orden en que se leen, hasta el final del archivo. Para cada lenguaje, el escáner determinará sus atributos adecuadamente (estos eventualmente serán utilizados por otros componentes de su compilador) para que la información sobre cada símbolo deba estar correctamente impresa.

**Estructura Lexicográfica**

Estos son los tokens para ambos lenguajes:

* Palabras clave (reservadas):

Para mini-C#:

**void int double bool string class const interface null this for while foreach if else return break New NewArray Console WriteLine**

Para mini-Java:

**void int double boolean string class const interface null this extends implements for while if else return break New System out println**

* Identificadores:
  + Para mini-C#:

Un identificador es una secuencia de letras, dígitos y guiones bajos siempre comenzando

con una letra.

* + Para mini-Java:

Un identificador es una secuencia de letras, dígitos y signo dólar. Puede comenzar con cualquiera excepto un número.

Mini C# y Mini Java distingue entre mayúsculas y minúsculas (case sensitive).

* **if** es una palabra clave pero **IF** es un identificador
* **binky** y **Binky** son dos identificadores distintos.
* Espacios en blanco

El espacio en blanco (es decir, espacios, tabuladores y saltos de línea) sirve para

separar tokens, pero por lo demás debe ser ignorado. Palabras clave y los

identificadores deben estar separados por espacios en blanco, o por una señal de

que no es ni una palabra ni un identificador.

* **if ( 23 this** se escanea como cuatro tokens, al igual que **if(23this**
* Comentarios

Ambos lenguajes han adoptado los dos tipos de comentarios disponibles en C++. Un comentario de una sola línea se inicia con // y se extiende hasta el final de la línea. Comentarios de varias líneas comienzan con /\* y terminan con la primera \*/ posterior. Cualquier símbolo se permite en un comentario, excepto el de secuencia \*/ que pone fin al comentario actual. Los comentarios de varias líneas no se anidan.

Si un archivo incluye un comentario sin terminar, el escáner debe informar de un error.

* Constantes
  + Las constantes booleanas son: **true** o **false**.
  + Una constante entera puede ser expresada en decimal (base 10) o en hexadecimal (Base 16).
    - Un entero decimal es una secuencia de dígitos decimales (**0-9**).
    - Un entero hexadecimal debe comenzar con **0X** o **0x** (es el carácter cero) y es seguida por una secuencia de dígitos hexadecimales. Los dígitos hexadecimales incluyen los dígitos hexadecimales y las letras **a** a la **f** (ya sea en minúsculas o mayúsculas).
    - Ejemplos de enteros válidos: 8,012,0X0,0x12aE.
  + Una constante double es una secuencia de dígitos, un punto, seguido de una secuencia de dígitos, o nada. Así, **.12** no es una constante de tipo double, pero **12.** y **0.12** lo son. Una constante doble puede tener parte exponencial, por ejemplo, **12.2E+2**. Para la constante doble en la notación científica el punto decimal es requerido, el signo del exponente es opcional (si no está especificado, + es asumido), y el **E** puede ser en mayúscula y minúscula. Entonces, **.12E+2** es inválido, pero **12.E+2** es válido. Ceros al inicio de la mantisa y el exponente son permitidos.
  + Una constante string o cadena de caracteres es una secuencia de caracteres encerrada por comillas dobles “”. Los strings pueden contener cualquier carácter excepto una línea nueva, doble comilla o carácter nulo. Una constante string debe comenzar y finalizar en una misma línea, y no puede partirse en líneas múltiples. Ejemplos:

"Está es una cadena de caracteres que no tiene su doble comilla

Esta no es parte de la cadena de arriba // Todos deben ser ids

* Operadores y caracteres de puntuación:

+ - \* / % < <= > >= = == != && || ! ; , . [ ] ( ) { } [] () {}

Note que **[**, **]**, y **[]** son tres tokens diferentes y que para el operador **[]**, al igual que

los otros dos operadores de dos caracteres, no debe haber ningún espacio en

blanco entre los dos caracteres.

**Implementación**

1. Su analizador deberá leer un archivo de entrada en lenguaje Mini C# o Mini Java (según corresponda) y determinar si pertenece o no al lenguaje. Es decir, su analizador debe:
   1. “comerse” los espacios en blanco
   2. Reconocer todas las palabras claves, los símbolos de puntuación, operadores de un carácter, operadores de dos caracteres, las constantes (int, double, bool/boolean, and string) e identificadores, retornando el token correcto
   3. Guardar el número de línea, la primera y última columna para todos los tokens
   4. El escáner debe reconocer los errores con un token especial de error llamado ERROR. Los errores deben ser presentados claramente en pantalla y llevarlos también en un archivo de salida.
2. Su escáner debe ser robusto, debe funcionar para cualquier entrada imaginable. Por ejemplo, debe manejar errores como un EOF en medio de una cadena o comentario, así como cadenas demasiado largas. Estos son solo algunos de los errores que pueden ocurrir; debe saber recuperarse de un error fatal. Un pobre manejo de errores o las excepciones no detectadas serán inaceptables.

**Manejo de errores**

Los errores que su escáner puede reportar son:

* 1. Cadenas de caracteres no válidas, debe mostrar un mensaje de carácter no reconocido y el carácter en cuestión. Debe continuar con el análisis con el siguiente carácter.
  2. Strings sin terminar (sin comilla doble de cierre), debe reportarlos como cadena sin terminar y continuar el análisis en la línea siguiente.
  3. Identificadores de longitud no permitida y caracteres inválidos. Si un identificador es más grande que el máximo aceptado, muestre un mensaje de error truncando el identificador con los primeros 31 caracteres (descartando el resto), y continuar el escaneo. Si la cadena contiene caracteres no válidos (por ejemplo, un carácter nulo), debe reportar que la cadena tiene el carácter nulo que no es válido. En cualquier caso, debe continuar el análisis con el carácter siguiente a esa cadena. El fin de cadena será la nueva línea o la comilla doble de cierre.
  4. Si un comentario se queda abierto cuando se encuentre el final del archivo, reportar el error con el mensaje “EOF en comentario". Su escáner no debería analizar este último comentario y su contenido. De igual forma para las cadenas, si el final del archivo se encuentra antes de unas comillas de cierre, reportarlo como error “EOF en una cadena”.
  5. Si se escanea un “\*/” fuera de un comentario, reportarlo como fin de comentario sin emparejar, en lugar de reconocer los tokens “\*” y “/”.
  6. Recordar que en esta fase del compilador solo se reconoce una clase muy limitada de errores. **No busque errores que no sean de léxico para esta fase.** Por ejemplo, no se debe revisar que las variables se hayan declarado antes de su uso.

**Ejemplos de archivo de entrada**

|  |  |
| --- | --- |
| Example.frag | Example.out |

**Requerimientos generales**

* El analizador se trabajará de forma individual o en parejas únicamente. El lenguaje a utilizar es libre, pero se recomienda C#, java o Python. Deberá utilizar expresiones regulares para definir su analizador. NO se permitirá implementar algún generador de analizadores léxicos como lex, flex o jflex.
* El ejecutable debe ser llamado **minic o minij** y debe pedir el archivo de entrada y escribir todos los resultados en un archivo de salida del mismo nombre al de entrada con extensión **.out** además de mostrarlos en pantalla.
* El proyecto se entrega en un repositorio en GitHub; se calificará el proyecto hasta el último commit realizado antes de la hora de entrega. En el espacio del portal indicado deberán entrega el ejecutable que se evaluará. **No se aceptarán cambios en el momento de la calificación**.
* En el repositorio deberá incluir un archivo README.txt donde explique todo el funcionamiento y porque cree que su programa funciona correctamente y es robusto, así también detallando cómo se manejan los errores.

**Referencias**

<https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

<https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=netcore-3.1>

<https://www.w3schools.com/python/python_regex.asp>

**¡Buena suerte!**