Scanner

Compiladores

Setup:

El **CLI** es el programa que interactúa con el usuario (**I**nterfaz de la **L**ínea de **C**omandos). Este se encarga de llamar a los paquetes y clases indicadas para todo el proceso de compilar. El compilador está dividido en 2 grandes partes: Análisis y Síntesis, y cada una de estas partes se divide en varias partes (dependiendo del compilador).

Para que esta interfaz sea más amigable al usuario; el **CLI** debe ser capaz de presentar opciones, ayuda y resultado (si lo hay) de la operación realizada. También hay que recordar que ninguna de las siguientes fases puede ejecutarse por sí sola, por lo que todo se debe ejecutar desde el **CLI.**

Tengan en cuenta de que lo que ejecute el compilador para esta asignación seguirá durante todo el proyecto, esto lo deben hacer para familiarizarse de nuevo con java y estructurar mejor su proyecto. El **CLI** seguirá siendo el mismo, pero cambiará la ejecución en entregas posteriores.

Estructura de los archivos

Deberán crear una serie de archivos que implementarán cada parte del compilador y por supuesto la clase Compiler.java (**CLI**) que controla todos esos archivos.

A continuación, se muestra la estructura de directorios que deben usar, es importante notar que TODOS los directorios comienzan con minúscula y las clases con Mayúsculas

```
/class
  /compiler
    Compiler.java

    Makefile

    /scanner
      Scanner.java

    /parser
      Parser.java

    /ast
      Ast.java

    /semantic
      Semantic.java

    /irt
```

```
    Irt.java

  /opt
    Algebraic.java
    ConstantF.java

  /codegen
    Codegen.java
```

Command Line Interface:

Para correr el **CLI** se debe escribir en la línea de comandos:

```
java compiler [option] <filename>
```

Si no se coloca un archivo de entrada debe mostrarse una pequeña sinópsis con la ayuda, tal y como si se colocara la opción -h. Las opciones que debe tener el compilador son las siguientes:

Parametro	Descripción
`-o <outname>`	Escribir el output a `<outname>`

`-target <stage>`	`<stage>` es uno de: `scan, parse, ast, semantic, irt, codegen`
	En este caso, la compilación debe proceder hasta la etapa indicada, es decir, si `<stage>` es `scan`, una instancia de `scan` debe ser creada imprimiendo al archivo de salida
	`"stage: scanning".`
	
	Si es parse una instancia de `parser` debe ser creada a partir de la instancia de `scanner` imprimiendo al archivo de salida
	`"stage: parsing"`
	`"stage: scanning"`
	
	Si no se especifíca ningún stage, entonces por defecto se llegará hasta la generación de código.

-opt <opt_stage>	<opt_stage> es uno de: `constant`, `algebraic`;
	En este caso, la compilación debe llegar a la fase especificada por `-target` y hacer solo la optimización que se le pida, por lo que debe imprimir al archivo de salida
	`"optimizing: constant folding"`
	
	o bien
	`"optimizing: algebraic simplification".`

-debug <stage>	Imprime información de debugging. Debe haber un mensaje por cada etapa listada en <stage> de la forma
	
	`"Debugging <stage>"`
	
	`<stage>` tiene las mismas opciones de `-target`, con la diferencia que se pueden "debuggear" varias etapas, separandolas con ':' de la forma `scan:parse:etc`

Hasta el momento únicamente hemos descrito el esqueleto de lo que eventualmente será su compilador, todos los archivos a excepción de Compiler.java deben pertenecer al paquete "compiler".

La clase principal `Compiler.java` debe llamar (dependiendo de los argumentos) a las demás clases, el orden de las clases comienza con scanner y termina codegen, cada clase recibe en el constructor un objeto de la clase anterior es decir, la clase `Scanner.java` recibe el nombre del archivo en el constructor (es la inicial), `Parser.java`, requiere de un objeto Scanner (lo recibe en el constructor), `Ast.java` recibe un objeto Parser, y así sucesivamente.

Al entrar a cada clase se debe imprimir al archivo de salida el mensaje:
stage: <stage> // ver las opciones de stage

Si el debugging está activo, debe imprimir a pantalla:
debugging <stage>

Scanner:

Su scanner debe ser capaz de identificar tokens del lenguaje Decaf.

Su scanner debe producir mensajes de error razonables y específicos para caracteres ilegales, comillas faltantes y otros errores léxicos. Su scanner debe encontrar tantos errores léxicos como sea posible, y debe continuar escaneando luego de encontrar errores. El scanner también debe filtrar comentarios y espacios en blanco.

JFlex

No van a tener que escribir el scanner desde cero. Usarán un generador de scanners llamado JFlex para crear su scanner. JFlex es un generador de scanners creado en la Universidad de Princeton. Este programa lee un archivo de especificación que contiene expresiones regulares que describen los tokens y produce un programa en Java que escanea el lenguaje descrito. Pueden encontrar más información acerca de JFlex (incluyendo el manual y algunos ejemplos) en: http://jflex.de/

Se les sugiere que sigan el diseño propuesto en el libro para que sea más fácil programar el parser. Debe definir todos sus tokens en una clase sym para asegurarse que su scanner sea compatible con el generador de parsers CUP.

Detalles

JFlex va a leer las expresiones regulares de un archivo de entrada y producirá otro archivo con el resultado del análisis de las expresiones regulares. Noten que Jlex solamente genera un archivo fuente de Java para la clase scanner; no lo compila y ni siquiera chequea que la salida sea sintácticamente correcta. Por lo tanto, cualquier typo o error sintáctico en el archivo de especificación (el .flex) va a ser propagado a la salida. Ustedes deben arreglar cualquier error de este tipo en el archivo `.flex` antes de entregar su proyecto.

Debe investigar los requisitos que debe cumplir para integrarse con el parser.

Por ejemplo, proveer una representación básica abstracta para los `tokens`. Dicha representación generalmente contiene cuatro campos básicos por token:

`type` - el tipo de símbolo (un valor de tipo int de la clase sym)
`left` - es la posición izquierda original en el archivo de entrada
`right` - es la posición derecha original en el archivo de entrada
`value` - el valor léxico de tipo Object

Estos valores son pasados al constructor para `Symbol` en este orden.

El tipo (`type`) de un `Symbol` va a ser usado por el generador de parsers CUP en la siguiente fase del proyecto.

Todo token declarado para Decaf será utilizado como parte de la gramática, así que cada uno debe tener asignado un identificador único. Algunas clases de tokens toman muchos valores, pero no es necesario que tengan un tipo único para cada valor. Un ejemplo de esto son los tokens tipo "integer". La gramática no distingue entre dos valores distintos de un entero. Por lo tanto, todos los enteros pueden tener el mismo tipo. El valor de cada token tipo entero es guardado en el campo `value` de la clase `Symbol`.

El campo `value` de `Symbol` contiene información descriptiva acerca del token. Noten que el campo `value` es de tipo `java.lang.Object`, así que puede guardar una referencia a un objeto de cualquier tipo. Para esta parte del proyecto, simplemente lo vamos a usar para guardar los strings de atributos de algunos tokens. Sin embargo, en las fases siguientes va a ser usado para guardar información más compleja.

La clase `Symbol` no contiene ninguna información acerca del número de línea del token; en vez de esto, guarda los valores de la posición absoluta del token en el archivo de entrada. Hay dos approaches posibles que pueden usar para generar mensajes de error que incluyan el número de línea. El approach usado en el libro de Appel es definir un objeto global `ErrorMsg` que guarda las posiciones de los cambios de línea en el archivo de entrada, y calcula la posición fila y columna de cualquier token usando esta información. Si no les gusta este approach, otra forma es crear una subclase de la clase `Symbol` que guarde más información (específicamente el número de línea) acerca del token para usarla en los mensajes de error. Un tercer approach derivado del segundo seria utilizar los campos left y right de la clase Symbol para guardar información del número de línea. Pueden usar cualquiera de estos dos approaches, pero sus mensajes de error *deben* incluir el número de línea.

A pesar de que la mayor parte del lexer va a ser generada automáticamente y no programada por ustedes, todavía deben crear una clase `Compiler` que lea la línea de comandos y llame al lexer generado si se le solicita la etapa `scan`. Es importante que esta clase se comporte como se especifica en la sección *¿Qué entregar?*.

¿Qué entregar?

Deben entregar un archivo `.zip` llamado `grupoNN.zip` mediante Miu, donde `NN` es su número de grupo. Este archivo debe contener un directorio llamado `grupoNN` en donde esté el directorio compiler creado en su proyecto 0 con los archivos de su proyecto. Debe seguir con la estructura de archivos y directorios descrita anteriormente, esto es muy importante.

Al desempacar el archivo zip, cambiarse al directorio creado y ejecutar `make` debe producirse el archivo `Compiler.class`. Si le asigna al CLASSPATH el string "`grupoNN:.`", debe poder ejecutarse su compilador desde el directorio compiler de la siguiente forma:

```
java Compiler <filename> -target <target>
```

Su compilador debe manejar el argumento `scan` de la bandera `-target`, y debe asumir que el default es `scan`.

Su scanner (cuándo se usa `scan` como target) debe tener como salida una lista de errores (si los hay) y crear un archivo de salida que contenga una tabla con una línea para cada token detectado e indicando el número de línea y columna del token, el tipo de token, y el string de atributos.

Si se activa el flag de debug para el scanner también deben imprimirse los tokens que se están siendo leídos.

Finalmente, asegúrense de incluir documentación para este proyecto. Incluyan una descripción completa de cómo su scanner está organizado.