EINA - Universidad de Zaragoza
Departamento de Informática e Ingeniería de Sistemas
3º Grado de Ingeniería Informática
30230 Procesadores de Lenguajes. **Curso 2018-2019**
Profesor: Javier Fabra – jfabra@unizar.es
==========

PRÁCTICA 4: Análisis semántico.
Parte I: La tabla de símbolos

==========

I. OBJETIVOS

En esta práctica vas a implementar una tabla de símbolos, que integrarás en tu gramática realizada en la Práctica 3, para almacenar la información relativa a los programas escritos en MiniLeng. Además, vas a implementar una primera verificación semántica sencilla utilizando la tabla.

II. CONTENIDO

NOTA: A lo largo de la práctica se proponen los nombres de los atributos, campos, tipos de datos, etc. Evidentemente, tenéis libertad para utilizar otros nombres (que sean descriptivos), así como para aplicar las reglas de estilo que consideréis más convenientes o que prefiráis (siempre dentro de los márgenes del código del buen programador, ver `http://en.wikipedia.org/wiki/Naming_convention_(programming)`).

En clase se ha descrito la utilidad e importancia de la tabla de símbolos en los traductores/compiladores. Como recordarás, la **tabla de símbolos** es la estructura utilizada por el compilador para almacenar información (*atributos*) asociada a los símbolos declarados en el programa en compilación.

A continuación, se van a dar las pautas para poder desarrollar una tabla de símbolos, similar a las que utilizan los compiladores que utilizáis diariamente, y que nos servirá para realizar el analizador semántico de MiniLeng.

La clase `Simbolo`

En primer lugar, vamos a definir los símbolos que conforman los programas MiniLeng. Para ello, comenzaremos por crear una clase llamada "**`Simbolo`**". Esta clase implementa los símbolos que se encuentran en nuestro lenguaje, por lo que tendrá por atributos una serie de campos de los siguientes tipos:

- `Tipo_simbolo`: Representa el tipo de símbolo. Podrá tomar los siguientes valores (una posible implementación es como un tipo enumerativo, por ejemplo): `PROGRAMA`, `VARIABLE`, `ACCION`, `PARAMETRO`
- `Tipo_variable`: Representa el tipo de variable. Podrá tomar los siguientes valores: `DESCONOCIDO`, `ENTERO`, `BOOLEANO`, `CHAR`, `CADENA`
- Clase_parametro: Representa la clase de los parámetros en las acciones: `VAL`, `REF` (valor y referencia, respectivamente)

Los atributos necesarios para un símbolo son:

- `nombre`: será una cadena de texto;
- `nivel`: entero que representa el nivel (bloque) en el que se ha declarado el símbolo. El nivel comienza en 0 y se va incrementando conforme aumenta el anidamiento al declarar acciones;
- `tipo`: del `Tipo_simbolo`;
- `variable`: del `Tipo_variable`;
- `parametro`: del tipo `Clase_parametro`;
- `visible`: booleano que indica si el símbolo es visible o no;
- `lista_parametros`: lista de símbolos que representan los parámetros de una acción;
- `dir`: long o entero representando la dirección del símbolo (**de momento no utilizaremos su valor**, pero podemos añadir el atributo correspondiente).

Evidentemente, muchos de estos atributos sólo tendrán sentido si el símbolo es una variable, o un parámetro, o una acción, etc.

Deberás implementar los métodos *getter* y *setter* necesarios para dar valor a los atributos.

EXTRA: Además, puedes definir métodos directos para configurar ciertos tipos de símbolos. Por ejemplo: un método `simbolo.introducir_parametro(nombre, tipo_var, clase_param, nivel)`, que configura los campos del símbolo correspondientes a un parámetro (nombre del parámetro, tipo de variable, clase del parámetro y nivel en el que ha sido declarado).

EXTRA: Puedes definir también una serie de funciones que nos facilitarán en la siguiente práctica el análisis semántico:

- `ES_VARIABLE(..)` → devuelve `TRUE` si `tipo == VARIABLE`
- `ES_PARAMETRO(..)` → devuelve `tipo == PARAMETRO`
- `ES_ACCION(..)` → `tipo == ACCION`
- `ES_VALOR(..)` → `(tipo == PARAMETRO) AND (parametro == VAL)`
- `ES_REFERENCIA(..)` → `(tipo == PARAMETRO ) AND (parametro == REF)`

La clase `Tabla_simbolos`

Con esto, habrás creado la clase **`Simbolo`**. Ahora vamos a implementar la tabla de símbolos. La forma más usual y práctica de implementar una tabla de símbolos es mediante una **tabla de dispersión (colisiones, *tabla hash*) abierta**. Por tanto, ahora debes crear una clase llamada "**`Tabla_simbolos`**" que implemente una tabla hash de instancias de objetos de la clase **`Simbolo`**.

ATENCIÓN: Debes implementar el TAD tabla de dispersión (colisiones, *tabla hash*) abierta incluyendo la función de dispersión, no utilizar el propio tipo de dato ni la función de *hash* que nos proporciona Java. Esto es un requisito obligatorio para la correcta evaluación de las prácticas.

El tamaño de la tabla se deja a tu elección (decisiones de diseño). La función de hash también, pero debe generar el hash considerando sólo el nombre del símbolo. Sin embargo, el mecanismo para manejar colisiones debe ser el estudiado en clase, el **encadenamiento de colisiones** (generando, por tanto, una tabla abierta).

La tabla de símbolos debe ofrecer los siguientes métodos (se muestran con la sintaxis del lenguaje Java), que **debes implementar siguiendo un enfoque orientado a objetos**:

```
public Tabla_simbolos(){
...

/*********************************************************************/
public void inicializar_tabla(){ … }
/*********************************************************************
Crea una tabla de símbolos vacía.  Este procedimiento debe invocarse
antes de hacer ninguna operación con la tabla de símbolos.
*********************************************************************/

/*********************************************************************/
public Simbolo buscar_simbolo(String nombre)
     [ throws SimboloNoEncontradoException ] { … }
/*********************************************************************
Busca en la tabla el símbolo de mayor nivel cuyo nombre coincida
con el del parámetro (se distinguen minúsculas y mayúsculas).  Si
existe, devuelve un puntero como resultado, de lo contrario devuelve
NULL o lanza una excepción (se deja a vuestra elección un mecanismo
u otro).
*********************************************************************/

/*********************************************************************/
public Simbolo introducir_programa(String nombre, int dir){ … }
/*********************************************************************
Introduce en la tabla un simbolo PROGRAMA,  con el nombre
del parametro, de nivel 0, con la direccion del parametro. Dado que debe
ser el primer simbolo a introducir, NO SE VERIFICA QUE EL SIMBOLO YA
EXISTA.
*********************************************************************/

/*********************************************************************/
public Simbolo introducir_variable(String nombre,
     TipoVariable variable,
     int nivel,
     int dir) { … }
/*********************************************************************
Si existe un símbolo en la tabla del mismo nivel y con el mismo
nombre, devuelve NULL (o una excepción, esto se deja a vuestra
elección.  De lo contrario, introduce un símbolo VARIABLE (simple)
con los datos de los argumentos.
*********************************************************************/

/*********************************************************************/
public Simbolo introducir_accion (String nombre,
                          int nivel,
                          int dir);
/*********************************************************************
Si existe un símbolo en la tabla del mismo nivel y con el mismo
nombre, devuelve NULL.  De lo contrario, introduce un símbolo
ACCION con los datos de los argumentos.
*********************************************************************/
```

```
/*********************************************************************/
public Simbolo introducir_parametro (String nombre,
                                     Tipo_variable variable,
                                     Clase_parametro parametro,
                                     int nivel,
                                     int dir);
/*********************************************************************
Si existe un símbolo en la tabla del mismo nivel y con el mismo
nombre, devuelve NULL.  De lo contrario, introduce un símbolo
PARAMETRO con los datos de los argumentos.
*********************************************************************/

/*********************************************************************/
public void eliminar_programa() { … }
/*********************************************************************
Elimina de la tabla todos los símbolos PROGRAMA de nivel 0 (debería
haber uno solo).
*********************************************************************/

/*********************************************************************/
public void eliminar_variables(int nivel) { … }
/*********************************************************************
Elimina de la tabla todas las variables que sean del nivel del argumento.
NO ELIMINA PARÁMETROS.
*********************************************************************/

/*********************************************************************/
public void ocultar_parametros(int nivel) { … }
/*********************************************************************
Marca todos los parámetros de un nivel como ocultos para que no puedan
ser encontrados, pero se mantenga la definición completa de la
acción donde están declarados para verificación de
invocaciones a la acción.
*********************************************************************/

/*********************************************************************/
public void eliminar_parametros_ocultos(int nivel) { … }
/*********************************************************************
Elimina de la tabla todas los parámetros que hayan sido ocultados
previamente.  LOS PROCEDIMIENTOS Y FUNCIONES DONDE ESTABAN DECLARADOS
DEBEN SER ELIMINAODS TAMBIEN PARA MANTENER LA COHERENCIA DE LA TABLA.
*********************************************************************/

/*********************************************************************/
public void eliminar_acciones(int nivel) { … }
/*********************************************************************
Elimina de la tabla todas los procedimientos de un nivel.
LOS PARAMETROS DE ESTAS ACCIONES
DEBEN SER ELIMINADOS TAMBIEN PARA MANTENER LA COHERENCIA DE LA TABLA.
*********************************************************************/
```

Siguiendo el enfoque orientado a ejemplos, el método **`inicializar_tabla()`**, por ejemplo, se invocaría como **`tabla.inicializar_tabla()`**.

Se deja a vuestra elección implementar los mecanismos de detección de errores, si bien **se evaluará muy positivamente el implementar las diferentes excepciones que se deberían lanzar en cada caso**.

Integrando la tabla de símbolos en MiniLeng

Ahora que ya has implementado la tabla de símbolos, debes integrarla en tu compilador. Es el paso previo a la construcción de un analizador semántico. Para ello, debes considerar que **únicamente es necesaria una instancia** de la tabla de símbolos para todo el proceso de compilación. Partiendo del código JavaCC de la Práctica 3, crea esta instancia al comienzo del analizador. Ahora, debes añadir el código necesario para que:

- El nivel comience en 0 y se incremente conforme se declaran acciones anidadas.
- En las declaraciones de variables se añadan estas variables en el nivel actual. Vamos a realizar un primer análisis semántico: si detectas que una variable ya se ha declarado en ese nivel, debes emitir un mensaje de error (error semántico) y continuar la ejecución del analizador. Si la variable todavía no existía en la tabla, la añadiremos con el nivel y tipo correspondientes.
- En las declaraciones de acciones, debes añadir el identificador de la acción a la tabla, así como la lista de parámetros (variables del tipo parámetro), con los tipos y clases correspondientes. Si existe una acción o parámetro con el mismo nombre y en el mismo nivel, daremos un error semántico y continuaremos la ejecución.

Para verificar el correcto funcionamiento del enfoque de bloques, cada vez que se cierre un nivel (al alcanzar la palabra reservada "`Fin`"), puedes mostrar por pantalla el contenido de la tabla de símbolos y verificar que todo está funcionando como debería.

Finalmente, **debes preparar tu analizador para funcionar desde línea de comandos y que reciba un parámetro con el nombre del fichero .ml a analizar**. Si el fichero no existe, no se puede abrir o no se ha pasado el parámetro, la aplicación debe mostrar un error y las instrucciones para su manejo.