3er Reporte de proyecto

Pomares Angelino José Roman

Planteamiento del problema

Problema “Algoritmo para la construcción del Universo de Herbrand hasta un nivel i para una fórmula de la lógica de primer orden en forma clausal”.

La construcción del algoritmo de Herbrand dice que, sea S un vocabulario con al menos una constante, el universo de Herbrand de S, es el conjunto de todos los términos sin variables libres de S.

Diseño preliminar

El software a desarrollar será codificado en el lenguaje de programación Java, apoyado del IDE Netbeans versión 8.0.2, con JDK 1.8.0_201.

Se hará uso de analizadores léxicos y sintácticos para la validación de las fórmulas de entrada (más adelante definidas).

Se proporcionara una interfaz con formularios y componentes de Java. Tentativamente la carga de archivo .Txt para el testeo de fórmulas, así como un JTextarea donde se puedan ingresar manualmente las formulas. El resultado se mostrará en un componente de JTable.

Definición de formatos de entrada

La cadena de texto de entrada será una fórmula de forma clausal, definida como S donde:

S es un conjunto de cláusulas:

$$S_3 = \{ P(f(x)), a, g(y), b \}$$

Las cuales serán representadas de la forma:

$$S3 = \{\{P(f(x))\},\{a\},\{g(y)\},\{b\}\}$$

Además de la entrada del nivel i a computar, el cual será capturado como un entero.

Definición de formatos de salidas

La salida del procesamiento del Universo de Herbrand devolverá una lista finita de cadenas en el siguiente formato:

$$H_0 = \{ a, b \}$$
$$H_1 = \{ a, b, f(a), f(b), g(a), g(b) \}$$
$$H_2 = \{ a, b, f(a), f(b), g(a), g(b), f(f(a)), f(f(b), f(g(a)), f(g(b)), g(f(a)), g(f(b)), g(g(a)), g(g(b)) \}$$
$$\ldots$$

Analizador

Se definieron 11 tipos de tokens necesarios para el reconocimiento de cada carácter de la cadena de entrada los cuales son los siguientes:

Inicio: definido por “S=”

LlaveIzq: definido por “{”

LlaveDer: definido por “}”

ParenteIzq: definido por “(”

ParenteDer: definido por “)”

Coma: definido por “,”

Negación: definido por “~”

LetraConstante: definidas por el rango “a-e” (Minúsculas)

LetraPredicado: definida por el rango “P-Z” (Mayúsculas)

LetraVariable: definida por el rango “w-z” (Minúsculas)

LetraFuncion: definida por el rango “f-k” (Minúsculas)

Gramática

Cada una de estos tokens serán utilizados para formar la gramática que represente la estructura clausular necesaria como entrada para procesar el Universo de Herbrand.

Base = (Inicio) (LlaveIzq) Sentencias

Sentencias = (Llaveizq) Formula ((coma)|(LlaveDer))

Formula = (Predicados | Función | (constante)) ((Coma) Formula|LlaveDer)

Predicados = (Negación)(LetraPredicado)(ParenteIzq)inPredicados|(LetraPredicado)(ParenteIzq) inPredicados

inPredicados = (Funcion|(Constante)|(LetraVariable))((Coma)inPredicados|(ParenteDer))

Funcion = (LetraFuncion) (ParenteIzq) inFuncion

i**nFuncion** = (Funcion|(Constante)|(LetraVariable))((Coma)inFuncion|(ParenteDer))

Para asegurar que la validación de la cadena de entrada sea de forma clausular se hará uso de la herramienta Javacc, cuya descarga es gratuita y disponible para el sistema operativo windows, una vez instalada en el equipo basta con agregar al path de variables de entorno la dirección de la carpeta de instalación para hacer uso de los comandos desde cualquier ubicación.

En Javacc se comienza definiendo un archivo con extensión .jj, para este caso el código está dividido en 3 secciones explicadas a continuación:

Configuración de opciones y definición de la clase.

Para efectos de ejecutar el analizador más de una vez, se agregaran parámetros necesarios para evitar que el programa lance errores.

```
options{
  Ignore_Case = true;
        static=false;
}
```

PARSER_BEGIN es una palabra reservada, acompañada de PARSER_END acotan la importación de librerías así como la declaración de la clase e instanciación de objetos necesarios durante todo el parseo, para efectos de este proyecto, solo es necesario una lista de tokens, una lista de constantes, y una lista de identificadores de funciones.

```
PARSER_BEGIN(Analizador)
package compilador;
import java.util.ArrayList;

public class Analizador {
  public static ArrayList<String> listaTokens = new ArrayList();
  public static ArrayList<String> listaConstantes = new ArrayList();
  public static ArrayList<String> listafunciones = new ArrayList();
}

PARSER_END(Analizador)
```

}

Léxico

EL siguiente método llamado Input será una declaración de todos los tokens utilizados por el analizador, sin embargo, esta parte aún no define el significado de cada uno de ellos.

```
void Input() :
{}
{
  (<INICIO>|<LLAVEIZQ>|<PARENTEIZQ>|<PARENTEDER>|<COMA>|<NEGACION>|<CONSTANTE>|<LETRAPREDICADO>|<LETRAVARIABLE>|<LETRAFUNCION>)*
}
```

La declaración de cada uno de estos tokens se hace de la siguiente forma:

```
TOKEN:
{
    <INICIO:        "S=">{System.out.println("Token:  INICIO---> " + image);
                    Analizador.listaTokens.add("Token:  INICIO---> " + image);
                    }
    |<LLAVEIZQ:     "{">{System.out.println("Token:  LLAVEIZQ---> " + image);
                    Analizador.listaTokens.add("Token:  LLAVEIZQ---> " + image);
                    }
    |<LLAVEDER:     "}">{System.out.println("Token:  LLAVEDER---> " +image);
                    Analizador.listaTokens.add("Token:  LLAVEDER---> " + image);
                    }
    |<PARENTEIZQ:   "(">{System.out.println("Token:  PARENTEIZQ---> " +image);
                    Analizador.listaTokens.add("Token:  PARENTEIZQ---> " + image);
                    }
    |<PARENTEDER:   ")">{System.out.println("Token:  PARENTEDER---> " +image);
                    Analizador.listaTokens.add("Token:  PARENTEDER---> " + image);
                    }
    |<COMA:         ",">{System.out.println("Token:  COMA---> " +image);
                    Analizador.listaTokens.add("Token:  COMA---> " + image);
                    }
    |<NEGACION:     "~">{System.out.println("Token:  NEGACION---> " +image);
                    Analizador.listaTokens.add("Token:  NEGACION---> " + image);
                    }
    |<CONSTANTE:   (["a"-"e"])>{System.out.println("Token:  CONSTANTE---> " +image);
                    Analizador.listaTokens.add("Token:  CONSTANTE---> " + image);
                    }
    |<LETRAPREDICADO:   (["P"-"Z"])>{System.out.println("Token:  LETRAPREDICADO---> " +image);
                    Analizador.listaTokens.add("Token:  LETRAPREDICADO---> " + image);
                    }
    |<LETRAVARIABLE:    (["w"-"z"])+>{System.out.println("Token:  LETRAVARIABLE---> " +image);
                    Analizador.listaTokens.add("Token:  LETRAVARIABLE---> " + image);
                    }
    |<LETRAFUNCION:    (["f"-"k"])+>{System.out.println("Token:  LETRAFUNCION---> " +image);
                    Analizador.listaTokens.add("Token:  LETRAFUNCION---> " + image);
                    }
}
```

Además de definir, también será útil para indicar al analizador la impresión y adición del token a las listas declaradas anteriormente.

Nuestro analizador no es sensible a espacios o saltos de línea por lo que se especifica de la siguiente forma:

```
SKIP:
{
    " "  | "\t" | "\n" | "\r\n"
}
```

Sintáctico

Para la definición de la sintaxis, se utilizan funciones en las que se insertan expresiones, haciendo uso de operadores, Tokens y funciones. Para la expresión de la estructura general se definió la expresión iniciar() de la forma:

```
void iniciar() :{}
{
   <INICIO><LLAVEIZQ>sentencias()<EOF>
}
```

Donde <EOF> indica el final de la lectura, y sentencias() será definido como:

```
void sentencias():{}
{
  <LLAVEIZQ>formula()(<COMA>sentencias()|<LLAVEDER>)
}
```

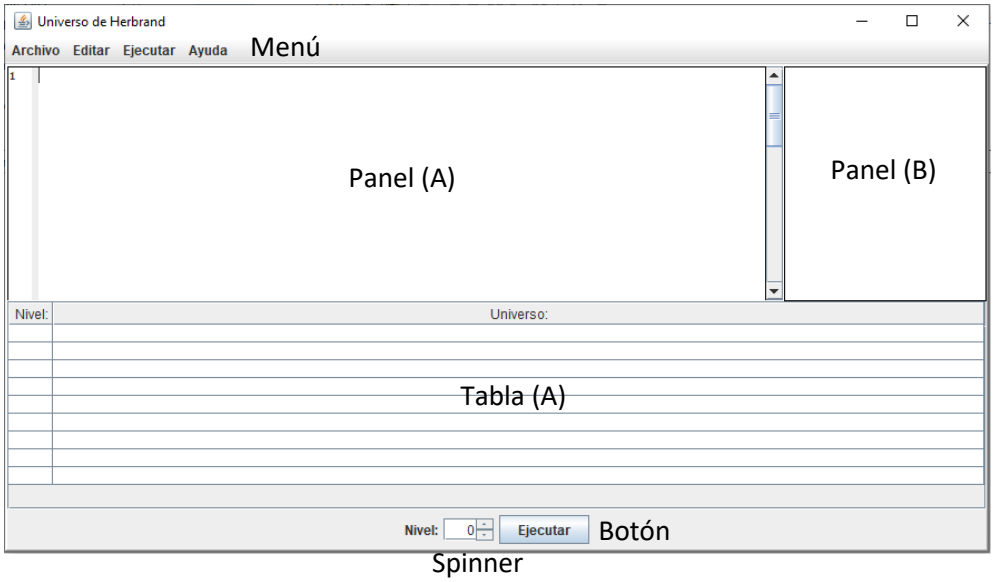
JavaCC permite la recursión por la derecha, de esta forma es posible invocar la expresión de sentencias dentro de sentencias.

Las demás expresiones quedan definidas de la siguiente forma:

```
void formula() :{}
{
    (predicados()|funcion()|<CONSTANTE>)(<COMA>formula()|<LLAVEDER>)
}

void predicados() :{}
{
    <NEGACION><LETRAPREDICADO><PARENTEIZQ>inPredicados()|<LETRAPREDICADO><PARENTEIZQ>inPredicados()
}

void inPredicados():{}
{
    (funcion()|<CONSTANTE>|<LETRAVARIABLE>)(<COMA>inPredicados()|<PARENTEDER>)
}

void funcion() :{}
{
    <LETRAFUNCION><PARENTEIZQ>inFuncion()
}

void inFuncion():{}
{
    (funcion()|<CONSTANTE>|<LETRAVARIABLE>)(<COMA>inFuncion()|<PARENTEDER>)
}
```

Desarrollo de interfaz

La interfaz del usuario es desarrollada con el asistente de diseño de Netbeans y haciendo uso de la paleta de componentes se logró el diseño siguiente:



Donde el Panel(A) será el área donde se ingrese la cadena de texto, el panel(B) es el área donde se hará impresión del análisis léxico y sintáctico, en caso de existir un error en la entrada, en este panel se mostrara el carácter faltante, o causante del error. En caso de que la cadena sea léxica y sintácticamente correcta, se imprimirá la lista de tokens con el reconocimiento de cada uno de los caracteres en la cadena asignado a su token.

La Tabla(A) será destinada para el despliegue del resultado una vez que la cadena pase por el algoritmo que devuelva el Universo de Herbrand por cada uno de los niveles seleccionados.

El botón colocado en la parte inferior central, con la etiqueta “Ejecutar” será el responsable de dar comienzo a la captura de la cadena ingresada, analizarla, procesar el algoritmo de Herbrand y devolver el resultado.

El Spinner colocado en la parte izquierda del botón tendrá la funcionalidad de capturar en formato de entero, el número de niveles calculados por el algoritmo de Herbrand.

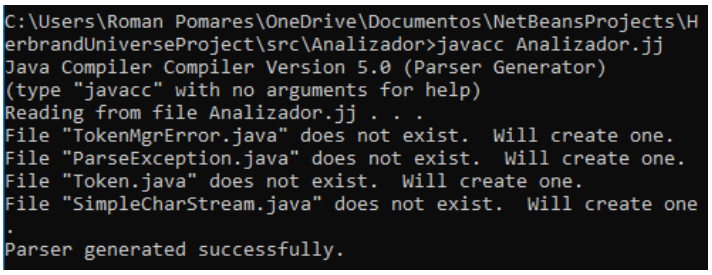
En la parte superior, existirá un menú genérico con opciones de **Archivo**, el cual permitirá abrir un archivo, guardar un archivo, y salir del programa. Seguida de la opción **Editar**, cuya opción solo será limpiar todas las áreas. Una opción de **Ejecutar** que tendrá nuevamente la funcionalidad del botón mencionado anteriormente con la etiqueta “Ejecutar”, y por último la opción de **Ayuda**, cuya funcionalidad será dar información del programa, con respecto a la sintaxis a utilizar.

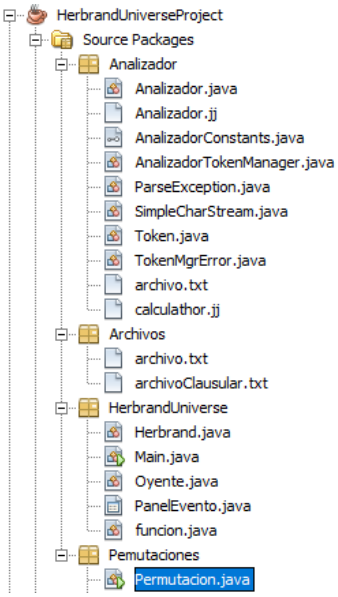
Desarrollo de Clases

La estructura del proyecto está organizada en 3 paquetes llamados: Archivos, HebrandUniverse, y Analizador.

El paquete llamado “Analizador”, tendrá las los archivos que corresponden a las funciones del análisis léxico y sintáctico.

En este paquete, el archivo Analizador.jj es el principal, pues la ejecución de este archivo genera automáticamente los demás archivos.





El paquete llamado “Archivos”, solo serán archivos con extensión .txt los cuales son auxiliares para el análisis, pues el parser generado, recibe de entrada un documento .txt

El paquete HerbrandUniverse contendrá clases importantes como el Formulario de interfaz, la clase oyente, el modelo función y la clase Herbrand.

Las clases relevantes para el desarrollo del proyecto es Analizador.jj y todas sus clases generadas .java por la herramienta javacc, la clase Oyente que implementa de ActionListener, su funcionalidad es ser intermediario entre la vista y el modelo, y la clase funcion, cuya utilidad es la abstracción de una función en nuestra entrada y manipularla como un objeto.

La clase Herbrand.java tendrá métodos asociados directamente con el algoritmo del Universo de Herbrand, apoyado de la clase Permutacion.java del paquete Permutaciones, su principal función será la declaración de listas y variables útiles en el algoritmo, funciones para división de cadenas de texto, y principalmente el algoritmo de Herbrand.

La clase Permutacion.java ofrece una función que recibe como entrada una la lista de elementos a permutar (inicialmente será la lista de constantes en nuestra formula), un String que será utilizado para concatenar recursivamente, un entero n que representa el número de elementos a tomar por cada permutación (representa el número de argumentos en cada función), otro entero r que significa el número de elementos en la lista de entrada, y finalmente un String que recibe la letra que identifica a la función que va a permutar elementos.

Desarrollo de algoritmo

La generación del universo de Herbrand se define de la manera siguiente:

Dado un conjunto de cláusulas universalmente cuantificadas (En forma estándar de Skolen):

- H0 son todas las constantes en S, si no hay constantes en S entonces H0=a
- Para i>0 Hi=Hi-1 unido al conjunto de todos los términos $f^n(t_1,\ldots t_n)$ para todas las funciones f^n que ocurren en S donde t_j pertenece H_{i-1} J = 1,…,n.
- H_i es llamado el i nesimo nivel conjunto constante de S
- H_n es llamado el universo de Herbrand de S

De esta forma, en la clase de Herbrand.java se define el nivel H0 con la agregación de las constantes a una lista (En caso de no existir constantes, se agrega “a”).

```
//Si no hay constantes, agrega 'a' por default al Universo
if (listaConstantes.size() == 0) {
    listaConstantes.add(new String("a"));
}
```

En caso de no existir funciones en la cláusula, el valor de cada nivel del universo es igual a la lista de constantes.

```
//Si no hay funciones, devuelve las constantes, en numero veces deacuerdo al nivel seleccionado
if (listaFunciones.size() == 0) {
    for (int i = 0; i < nivel; i++) {
        auxUniverso.clear();
        for (String strCons : listaConstantes) {
            auxUniverso.add(new String(strCons));
        }
        Universo.add(auxUniverso.toString());
    }
}
```

Estos dos fragmentos de código, cubren la generación del universo de la forma más básica, es decir, ninguna sola constante, y ninguna sola función.

El siguiente código, representa el caso en que la lista de funciones sea diferente de 0.

```
if (listaFunciones.size() != 0) {
    String toaddAuxUniverso = "";
    String totalPerm = "";
    auxUniverso.clear();
    //El proceso se repite n veces donde n el el numero de niveles
    for (int ni = 0; ni < nivel; ni++) {
        //Para cada nivel, se comienza agregando las constantes
        for (String strCons : listaConstantes) {
            auxUniverso.add(new String(strCons));
        }
        for (funcion func : funcionesArg) {
            //genera Permutaciones de la lista auxUniverso para cada funcione
            Permutacion perm = new Permutacion();
            perm.setPer("");
            perm.perm1(auxUniverso, "", func.getParametros().size(), auxUniverso.size(), func.identificador);
            //se concatena las salidas para cada funcion
            totalPerm += perm.getPer();
        }
        //El string devuelto se splitea en una lista para agregar al Aux universo
        String str[] = totalPerm.split("&");
        List<String> al = new ArrayList<String>();
        al = Arrays.asList(str);
        auxUniverso = Stream.concat(auxUniverso.stream(), al.stream()).distinct().collect(Collectors.toList());
        Universo.add(auxUniverso.toString());
    }

}
```

Una vez dentro de esta condición (listaFunciones.size()!=0) se genera una lista llamada AuxUniverso por cada nivel y esta es agregada a la lista definitiva llamada Universo, como la lista de auxUniverso se estará utilizando repetidas veces, es necesario limpiarla y agregar las constantes nuevamente. Dentro de cada iteración, por cada una de las funciones dentro de la lista de funcionesArg, se genera una permutación con la lista de auxUniverso con la clase Permutacion. El resultado de esta permutación será un String que será concatenada con la siguiente permutación de la función correspondiente.

La clase permutación, queda definida de la siguiente forma:

```
public class Permutacion {

public static String per="";

    public Permutacion() {...2 lines }

    public static String getPer() {...3 lines }

    public static void setPer(String per) {...3 lines }

    public static void main(String[] args) {...12 lines }

    public static void perm1(List<String> elem, String act, int n, int r,String idenFunc) {
        if (n == 0) {
//            System.out.println(act);
            per+=idenFunc+"("+act+")&";
        } else {
            for (int i = 0; i < r; i++) {
                if(n-1==0){
                perm1(elem, act + elem.get(i) + "", n - 1, r,idenFunc);
                }else{
                perm1(elem, act + elem.get(i) + ",", n - 1, r,idenFunc);
                }
            }
        }
    }
}
```

Diseñada de manera recursiva donde el método perm1 verifica su caso base n==0,en caso de será asi, concantena el identificador de la formula, seguido de todo el string en este caso llamado act, de no ser asi, para cada elemento de la cadena se vuelve a llamar a la función perm1 con la cadena concatenada con el elemento i del arreglo, enviando n-1 elementos para seleccionar.

Pruebas funcionales

En el siguiente apartado, el proyecto será testado de acuerdo a diferentes categorías de pruebas, manteniendo un registro de los resultaos.

Pruebas Unitarias

Las pruebas unitarias son hechas a un bajo nivel, es decir se probaran los métodos y funciones individuales de las clases, componentes o módulos usados en el proyecto.

Caso 1	
Objetivo	Respuesta de Menú “Archivo”
Descripción	La opción de “Archivo” debe responder al hacer clic sobre él y desplegar el listado de ítems que contiene el submenú.
Entrada	Clic en menú “Archivo”
Salida Esperada	SubMenus “Abrir Archivo”, “Guardar Archivo”, “Salir”
Resultado	Despliegue de SubMenus “Abrir Archivo”, “Guardar Archivo”, “Salir”
Observaciones	El menú “Archivo” responde adecuadamente

Caso 2	
Objetivo	Respuesta de Menú “Editar”
Descripción	La opción de “Editar” debe responder al hacer clic sobre él y desplegar el listado de ítems que contiene el submenú.
Entrada	Clic en menú “Editar”
Salida Esperada	SubMenus “Limpiar todo”
Resultado	Despliegue de SubMenus “Limpiar todo”
Observaciones	El menú “Editar” responde adecuadamente

Caso 3	
Objetivo	Respuesta de Menú “Ejecutar”
Descripción	La opción de “Ejecutar” debe responder al hacer click sobre él y desplegar el listado de ítems que contiene el submenú.
Entrada	Clic en menú “Ejecutar”
Salida Esperada	SubMenus “Ejecutar”
Resultado	Despliegue de SubMenus “Ejecutar”
Observaciones	El menú “Ejecutar” responde adecuadamente

Caso 4	
Objetivo	Respuesta de Menú “Ayuda”
Descripción	La opción de “Ayuda” debe responder al hacer click sobre él y desplegar el listado de ítems que contiene el submenú.
Entrada	Click en menú “Ayuda”

Salida Esperada	SubMenus “Información”
Resultado	Despliegue de SubMenus “Información”
Observaciones	El menú “Ayuda” responde adecuadamente

Caso 5	
Objetivo	Respuesta de Spinner de selección de niveles
Descripción	El Spinner asociado a la selección de niveles debe incrementar o decrementar su valor
Entrada	Modificación de valores en Spinner con los botones
Salida Esperada	Incremento o decremento del valor del Spinner
Resultado	Un incremento y decremento adecuado
Observaciones	El spinner responde adecuadamente

Caso 6	
Objetivo	Comportamiento correcto de área de entrada
Descripción	El TextArea del panel(a) debe estar habilitado para el ingreso de cadenas de texto.
Entrada	Escritura con el teclado en al área principal
Salida Esperada	Comportamiento adecuado del TextArea
Resultado	Comportamiento adecuado del TextArea
Observaciones	El TextArea no representa ninguna anomalía con respecto a su comportamiento.

Caso 7	
Objetivo	Comportamiento correcto de área de impresión de análisis
Descripción	El TextArea del panel(b) debe estar deshabilitado para el ingreso de cadenas de texto.
Entrada	Intento de escritura con el teclado en al área principal
Salida Esperada	Comportamiento adecuado del TextArea (No debe permitir escritura)
Resultado	Comportamiento adecuado del TextArea
Observaciones	El TextArea no representa ninguna anomalía con respecto a su comportamiento.

Caso 8	
Objetivo	Método rellenatabla()
Descripción	Dada un modelo de datos, la tabla debe rellenarse correctamente con respecto a sus filas y columnas
Entrada	Ejecución del método rellenaTabla()
Salida Esperada	La JTable colocada en la interfaz deberá rellenarse con datos ficticios.
Resultado	La tabla se rellena correctamente con los datos seteados.
Observaciones	El método se comporta adecuadamente

Caso 9	
Objetivo	Método imprimirFuncion ()
Descripción	Este método es propio de la clase funcion.java, útil para la impresión recursiva de los valores dentro de una función
Entrada	Paramentos de relleno del objeto funcion
Salida Esperada	String formateado en forma de Universo de Herbrand
Resultado	String formateado en forma de Universo de Herbrand
Observaciones	El método funciona correctamente

Caso 10	
Objetivo	Método imprimirLista ()
Descripción	Este método es propio de la clase oyente.java, útil para la impresión de lista de funciones y constantes, es adicional al método anterior (imprimirFuncion())
Entrada	Una lista rellena de funciones y constantes
Salida Esperada	String formateado en forma de Universo de Herbrand
Resultado	String formateado en forma de Universo de Herbrand
Observaciones	El método funciona correctamente

Pruebas de Integración

En las siguientes pruebas se verifican que los distintos módulos o servicios utilizados funcionan bien en conjunto.

Caso 1	
Objetivo	Comportamiento correcta de la opción “Abrir Archivo”
Descripción	La opción “Abrir Archivo” ligado a la clase oyente, con su respectivo método llamado abrirArchivo()
Entrada	Clic sobre la opción “Abrir archivo”
Salida Esperada	Despliegue de JFileChooser para selección de archivo .txt
Resultado	Despliegue de JFileChooser para selección de archivo .txt
Observaciones	Opción “Abrir archivo” es correcta

Caso 2	
Objetivo	Comportamiento correcta de la opción “Guardar Archivo”
Descripción	La opción “Guardar Archivo” ligado a la clase oyente, con su respectivo método llamado guardarArchivo()
Entrada	Clic sobre la opción “Guardar Archivo”
Salida Esperada	Despliegue de JFileChooser para selección de ubiación en carpeta y el nombrado del archivo con extensión .txt
Resultado	Despliegue de JFileChooser para selección de ubiación en carpeta y el nombrado del archivo con extensión .txt
Observaciones	Opción “Guardar archivo” es correcta

Caso 3	
Objetivo	Comportamiento correcta de la opción “Limpiar todo”
Descripción	La opción “Limpiar todo” ligado a la clase oyente, con su respectivo método llamado limpiar()
Entrada	Clic sobre la opción “Guardar Archivo”
Salida Esperada	Todos los paneles y la tabla debe volver a estar en blanco
Resultado	Solo el panel(a) y panel(b) respondieron
Observaciones	Falta por reiniciar la tabla

Caso 4	
Objetivo	Comportamiento correcta de la opción “Información”
Descripción	La opción “Información” ligado a la clase oyente, con su respectivo método llamado info()
Entrada	Clic sobre la opción “Información”
Salida Esperada	Visualización de un JDialog con información del programa
Resultado	JDialog con versión e información del programa
Observaciones	Responde correctamente

Caso 5	
Objetivo	Comportamiento correcta de la opción “Información”
Descripción	La opción “Información” ligado a la clase oyente, con su respectivo método llamado info()
Entrada	Clic sobre la opción “Información”
Salida Esperada	Visualización de un JDialog con información del programa
Resultado	JDialog con versión e información del programa
Observaciones	Responde correctamente

Caso 6	
Objetivo	Metodo imprimirLista() y imprimirFuncion() integrados
Descripción	Metodo de integración de ambos metodos
Entrada	Listado de funciones (rellenas de funciones y constantes)
Salida Esperada	Visualización de un JDialog con información del programa
Resultado	JDialog con versión e información del programa
Observaciones	Responde correctamente

Caso 7	
Objetivo	Pruebas de Análisis Léxico (1)
Descripción	La cadena de entrada debe ser analizada y cada carácter debe ser reconocido dentro del alfabeto.
Entrada	Una cadena que corresponde con el alfabeto
Salida Esperada	Mensaje de análisis exitoso
Resultado	Mensaje de análisis exitoso
Observaciones	El análisis léxico se realiza correctamente

Caso 8	
Objetivo	Pruebas de Análisis Léxico (2)
Descripción	La cadena de entrada debe ser analizada y cada carácter debe ser reconocido dentro del alfabeto.
Entrada	Una cadena que **NO** corresponde con el alfabeto
Salida Esperada	Mensaje de token desconocido “Error”
Resultado	Mensaje token desconocido en el análisis
Observaciones	El análisis léxico se realiza correctamente

Caso 9	
Objetivo	Pruebas de Análisis Sintáctico (1)
Descripción	La cadena de entrada debe ser analizada y cada gramática debe ser reconocida dentro de la formula.
Entrada	Una cadena que corresponde con la gramática
Salida Esperada	Mensaje análisis exitoso
Resultado	Mensaje de análisis exitoso
Observaciones	El análisis sintáctico se realiza correctamente

Caso 9	
Objetivo	Pruebas de Análisis Sintáctico (2)
Descripción	La cadena de entrada debe ser analizada y cada gramática debe ser reconocida dentro de la formula.
Entrada	Una cadena que **NO** corresponde con la gramática
Salida Esperada	Mensaje de error en la gramática seguido del token que genera el error
Resultado	Mensaje de error “Token esperado y fue encontrado: ”
Observaciones	El análisis sintáctico se realiza correctamente

Caso 10	
Objetivo	Probas el método de generación de permutaciones
Descripción	Se genera una instanciación para probar que las permutaciones sean correctas
Entrada	Una lista de constantes, el identificador de la función
Salida Esperada	Impresión de la cadena de las permutaciones
Resultado	Se imprimió las permutaciones necesarias para una función dado una lista
Observaciones	La generación de permutaciones es correcta

Caso 11	
Objetivo	Pruebas de Algoritmo de generación de universo de Herbrand
Descripción	Dada una formula clausular, deberá generar el universo para cada nivel
Entrada	Una formula representada de forma clausular
Salida Esperada	Tabla rellena de los niveles
Resultado	Las cadenas para cada nivel se generó y se agregó a la tabla
Observaciones	El nivel H0 deben ser solo constantes, y el nivel H0 devuelve funciones como corresponde al H1

Código

El proyecto se localiza como un repositorio de github, es posible acceder a través del siguiente link:

https://github.com/RomanPomares/HerbrandUniverse