DOCUMENTACIÓN "INTERPRETE DE PROLOG"

Lenguajes de programación

Descripción breve

El presente documento pretende ilustrar al lector con el proceso de desarrollo del intérprete de Prolog. Mostrando decisiones de diseño, algoritmos y librerías utilizadas entre otras cosas, además de un manual de usuario.

Fabián Fernández & Pablo Corrales

Tecnológico de Costa Rica

TABLA DE CONTENIDOS

TABAL DE CONTENIDOS

DESCRIPCIÓN DEL PROBLEMA

El problema a resolver consta del diseño y creación de una herramienta en el lenguaje de programación Python que permita familiarizarse con los conceptos de la programación lógica, mediante la implementación de una base de conocimientos, y de un motor de inferencia, similar al que se usa en el lenguaje Prolog.

Mediante la utilización de los modos de definición y de consulta se pretende recopilar la información necesaria para poder simular el funcionamiento de un programa en Prolog. En el que se almacenan los hechos y predicados en la base de conocimientos.

Se deben implementar unificaciones de variables con los diferentes hechos en la base de conocimientos.

Al finalizar la lectura de la información ingresada por el usuario se debe mostrar YES o NO, o bien, el valor que toma cada variable, también muestra distintos mensajes de error.

DISEÑO DEL PROGRAMA

ESCOGENCIA DEL LENGUAJE DE PROGRAMACIÓN

Se decidió utilizar el lenguaje de programación Python debido a la experiencia previamente adquirida del grupo con este lenguaje, además la implementación de funciones, la manipulación de variables y cadenas es ciertas ocasiones es más simple. Dentro de los lenguajes propuestos para realizar esta tarea estaban C, C++, Ruby y Python por lo que se decidió descartar los otros tres lenguajes debido a la complejidad del proyecto y el poco tiempo con el que se disponía en un principio, además de cómo se mencionó anteriormente, la experiencia de los integrantes del proyecto con el lenguaje.

UTILIZACIÓN DE LISTAS

Para almacenar el código ingresado en formato de código de Prolog se decidió utilizar listas para organizar y acomodar de manera más sencilla cada palabra ingresada y así poder evaluar cada token con el fin de determinar su tipo o valor.

VERIFICACIÓN DE REGLAS

Es una función que permite validar las reglas de sintaxis de Prolog, a continuación se mostrará el extracto del código en dónde se evalúan las diferentes errores sintácticos que podría presentar el intérprete al usuario.

```
# Funcion que verifica las reglas
def regla(resp,tamanno):
    if resp[tamanno-1] != ".":
        return "Error en la regla"
    inicio = 0
    if resp[inicio].islower() != True:
        return "Error en la regla"
    while tamanno > inicio:
        if resp[inicio] == ":":
            atras = inicio -1
            while resp[atras] !="(":
                atras = atras - 1
                aux = atras
            while aux < inicio or aux < (inicio+1):
                if resp[aux+1].islower() != False and resp[aux+1] != ")":
                    return "Error en la regla"
                aux = aux + 1
            if resp[inicio+2].islower() != True and resp[inicio+2] != "!":
                return "Error en la regla"
        if resp[inicio]=="(":
            if resp[inicio-1] ==",":
                return "Error en la regla"
            aux = 1
            while resp[inicio+aux] != " ":
                if resp[inicio+aux] == ")":
                    break
                if resp[inicio+aux] == ".":
                    return "Error en la regla"
                aux = aux + 1
        inicio = inicio + 1
    return "Corecto"
```

FUNCIÓN ENTRADA

Esta función fue creada con el fin de definir si la entrada es un hecho o una regla, es de suma importancia para la clase hecho y predicado, ya que le permite identificar a cuál pertenece cada argumento ingresado.

```
#Funcion inicial, define si la entrada es una regla o hecho, ENTRADA PRINCIPAL DE REGLAS Y HECHOS
def entrada(entrada):
    resp = quitar(entrada)
    if (")" not in resp) or ("(" not in resp) or ("." not in resp):
        return "Error de estructura"
    tamanno = 0
    for i in resp:
        tamanno = tamanno + 1
    letra = 0
    while tamanno > letra:
        if resp[letra] == ":":
            if resp[letra+1] == "-":
                solu = "-"#regla(resp,tamanno)
                return solu
        letra= letra + 1
    solu = V_hecho(resp, tamanno)
    return solu
```

CLASE HECHO

Es una clase que agrega los hechos a la base de conocimiento.

```
class hecho:
    def __init__(self, funtor, datos): #String funtor, lista datos.
        self.funtor = funtor #string[0]
        self.aridad = len(datos)#len(string[1])
        self.datos = datos#string[1]

    def agregarHecho(self): #agrega hecho a listaHechos
        global LargoBC
        BC.append(self)
        LargoBC+=1
```

CLASE PREDICADO

Es la clase encargada de agregar a la base de conocimientos al igual que la clase hecho, los predicados. Además, reemplaza ciertos elementos en el string para solamente obtener los datos relevantes para agregar a la base de conocimiento y separar cada hecho

```
class predicado:
    def __init__(self, funtor, datos,reglas): #String funtor, lista datos, lista reglas.
        self.funtor = funtor #string[0]
        self.aridad = len(datos)#len(string[1])
        self.datos = datos#string[1]
        self.reglas = reglas#string[2]

    def agregarPredicado(self): #agrega hecho a listaHechos
        global LargoBC
        BC.append(self)
        LargoBC+=1

def separaHechos(string): #Recibe string: "amigo(Juan,Carlos),agrada(Carlos,Isa);agrada(Carlos,Miguel)"
    string = string.replace('),',') and ')
    string = string.replace(');',') or ')
    string = string.split() #Lista con este formato: ['amigo(Luis,Carlos)', 'or', 'agrada(Luis,Ana)', 'and', 'agrada(Luis,Stef)']
    largo = len(string)
    cont = 0
    resultado = []
    while(cont<largo):
        if(cont%2 == 0):
            resultado = resultado + fragmenta(string[cont]) #Utiliza la funcion fragmenta para separa cada hecho.
        else:
            resultado = resultado + [string[cont]]
        cont = cont + 1
    return resultado #Devuelve lista como: ['amigo', ['Luis', 'Carlos'], 'or', 'agrada', ['Luis', 'Ana']]
```

FUNCIÓN DE UNIFICACIÓN

Esta función permite realizar la verificación que permite unificar las variables en el modo de consulta, recorriendo los hechos y predicados con los que se pueda unificar y hacer backtracking, en caso de que la variable esté vacía, se le asigna el valor de la segunda variable, o la variable con algún dato. Lo mismo sucede con la variable anónima. Para el caso de que sean constantes, compara la variable que almacena a cada una de las constantes y verifica que las variables que contienen las constantes sean completamente iguales.

```
def unificar(a,b):
    global l_vars
    if a[0].islower() and b[0].islower() and a==b:
        return True
    elif a[0].islower() and b[0].islower() and a != b:
        return False
    elif a in l_vars and a[0].isupper():
        if l_vars[a] == b and l_vars[a].islower():
            return True
        else:
            return False
    elif b in l_vars and b[0].isupper():
        if l_vars[b] == a:
            return True
        else:
            return False
    else:
        l_vars[a] = b
        l_vars[b] = a
        #print(l_vars)
        return True
```

FUNCIÓN COMPARA HECHO Y PREDICADO

Esta función recibe un objeto de tipo hecho, que permite saber si posee la misma aridad, functor y datos. Trabaja en conjunto con la función evaluar_pre e inBC para conocer si lo que se consulta existe en la base de datos.

```
def comp_hecho_predicado(hecho): #Recibe un objeto de tipo hecho, devuelve true o false si tiene mismo funtor, aridad y datos.
    global l_vars
    res = False
    cont = 0
    funtor = hecho.funtor
    aridad = hecho.aridad
    datos = hecho.datos
    while cont < LargoBC and not res:
        hechoCompara = BC[cont]
        #l_vars = {}
        if hechoCompara.funtor == funtor and hechoCompara.aridad == aridad: #and hechoCompara.datos == datos:
            if comp_datos(hechoCompara.datos,datos):
                if hecho_o_pre(hechoCompara) != "pre":
                    res = hecho_res(0,aridad,hechoCompara,datos)
                else:
                    i = 0
                    while i<aridad:
                        unificar(hechoCompara.datos[i],datos[i])
                        i+=1
                    print(l_vars)
                    reglas = separaHechos(hechoCompara.reglas)
                    res = evaluar_pre(reglas)
            else:
                res = False
        cont = cont + 1
    return res
```

MODO DEFINICIÓN

Este es el modo que permite ingresar al usuario los hechos a la base de conocimiento, el intérprete ingresa a este modo con la palabra reservada "<define>" y concluye el modo de definición para ingresar al modo consulta con "</define>".

```
def consulta(): #Modo definicion
    global BC
    global LargoBC
    string = input("Modo Definición: ?- ")
    while string != "</define>":
        val = inicio_verificaciones(string)
        string = fragmenta(string)
        if val[0:5] == "Error":
            print(val)
        elif len(string) == 3: #Si es igual a 3 es predicado y lo evalua
            new_str = predicado(string[0],string[1],string[2])
            if inBC(new_str):
                print("Predicado ya existe en la Base de Conocimientos")
            else:
                new_str.agregarPredicado()
                print("Predicado agregado exitosamente!")
        else: #Sino es hecho y lo agrega a la base de conocimientos
            new_str = hecho(string[0],string[1])
            if inBC(new_str):
                print("Hecho ya existe en la Base de Conocimientos")
            else:
                new_str.agregarHecho()
                print("Hecho agregado exitosamente!")
                print("Tamaño de Base de conocimientos: ",LargoBC)
        string = input("Modo Definición: ?- ")
```

MODO CONSULTA

El programa por defecto ingresa directamente al modo de consulta, por lo tanto el usuario deberá ingresar al modo de definición con la palabra reservada "<define" y vuelve al modo consulta con "</define>" valida además las diferentes palabras reservadas de Prolog para realizar las acciones correspondientes a estas palabras.

```
def inicio(): #Modo consulta (Inicio de programa)
    global ListaRes
    global l_vars
    estado = 1
    while estado:
        string = input("?- ")
        val = inicio_verificaciones(string)
        if string == '':
            pass
        elif string == "<define>":
            consulta()
        elif string == "Exit":
            estado = 0
        elif string == "nl":
            print()
        elif string == "i_BC":
            print(BC)
        elif val[0:5] == "Error":
            print(val)
        else:
            string = fragmenta(string)
            if string[0] == "write":
                escribe(string[1])
            else:
                new_str = hecho(string[0],string[1])
                l_vars = {}
                if comp_hecho_predicado(new_str) == True:
                    print("YES")
                else:
                    print("NO")
    print("Fin de Tarea Programada 2")
```

LIBRERÍAS USADAS

No se requirieron librerías externas para el desarrollo de esta tarea programada, todo se desarrolló por los miembros del equipo sin necesidad de recurrir a librerías. Decidimos hacerlo de esta manera para propiciar un mayor y mejor aprendizaje en el proyecto, sin olvidar que la información para desarrollar un intérprete de Prolog es muy limitada en el aspecto de librerías y reutilización de código. Además no se sintió la necesidad de utilizar dichas librerías.

ANÁLISIS DE RESULTADOS

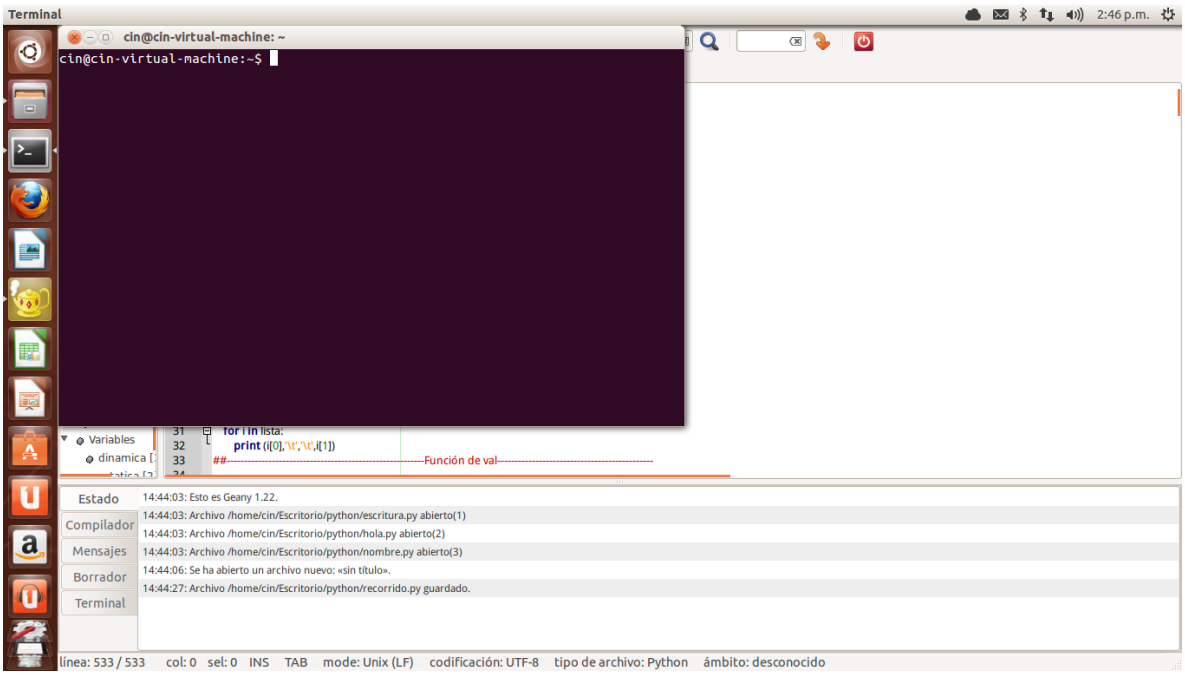
- La utilización del lenguaje de programación Python fue una decisión acertada ya que nos permitió agilizar el proceso de trabajo con listas y almacenamiento de datos.
- Los requisitos solicitados por el profesor se fueron abarcando por partes y distribuyendo entre los integrantes del grupo con el fin de poder interiorizar el problema y la posible solución a desarrollar.
- El trabajo grupal gestionado por GitHub fue un aspecto positivo que facilitó el manejo de los archivos y documentos necesarios para el desarrollo del trabajo.
- La distribución total del trabajo en el equipo tuvo aspectos positivos y negativos ya que se realizaron funciones individualmente que agilizaron el proceso de construcción de la tarea pero que dificultaron la unión y complementación de unas partes con otras.
- El funcionamiento general de la aplicación no es en su totalidad satisfactorio ya que no logramos pulir el código fuente de manera tal que se mejorara su efectividad y rendimiento en ejecución así como la robustez y estabilidad del mismo.
- El manejo de listas en Prolog no fue realizado con éxito ya que el programa no tiene un buen funcionamiento cuando se ingresan listas.

MANUAL DE USUARIO

Para ejecutar la aplicación se requiere tener instalado Python 3.2, a continuación se mostrará como correr el programa en el sistema operativo Ubuntu.

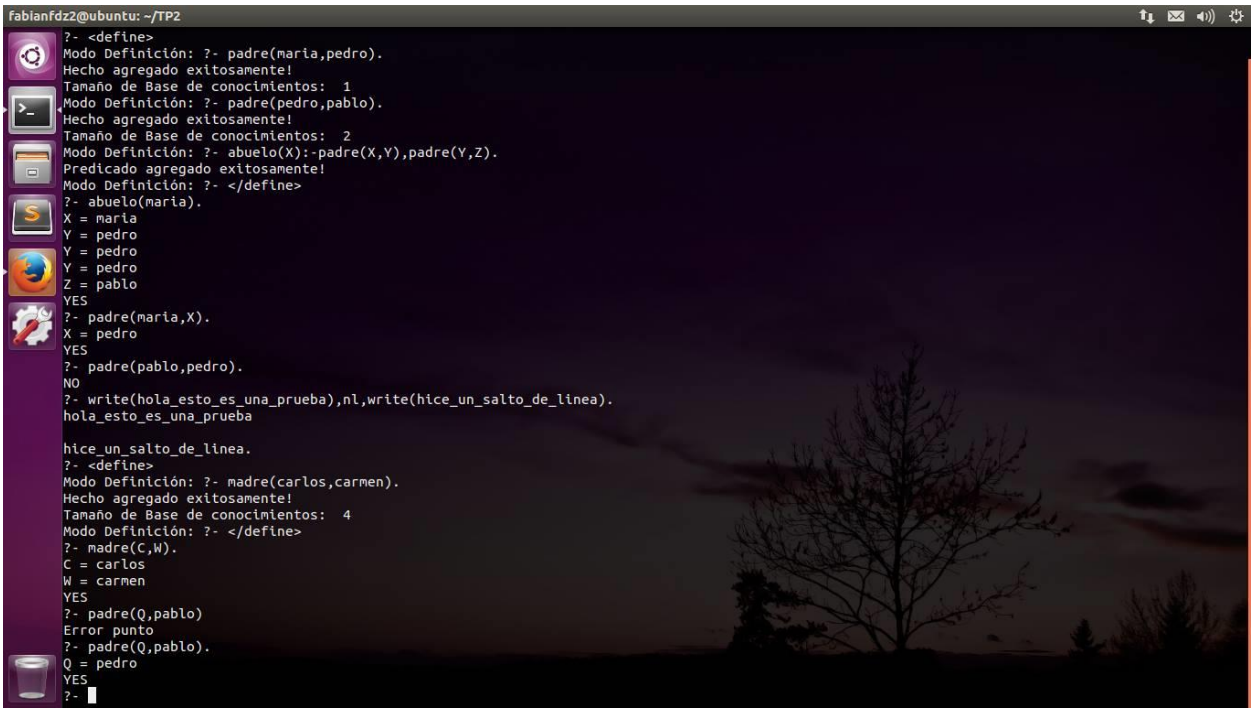
Si usted no cuenta con Python se le recomienda descargar cualquier editor de texto e instalar el intérprete de Python este lo pueden obtener de "Centro de software de Ubuntu". La recomendación sería utilizar Sublime Text o Geany.

1. Abrir la terminal con la combinación de teclas ctrl+alt+T



2. Se procede a buscar el archivo en el equipo; para ello se utilizarán los comandos "cd" para seleccionar ya sea carpetas o archivos y para conocer los archivos que contiene una carpeta se utiliza "ls". Posterior a esto, una vez conociendo el nombre de la carpeta en donde está nuestro programa se procede a correr el programa
3. A continuación se procede a correr el archivo Progra2.py, ya que este es el script que contiene el código que ocupamos. Para esto en consola escribimos "Progra2.py".
4. Sin mayor complicación se procede a ejecutar el programa, a continuación se presenta una breve demostración del funcionamiento del programa, cabe destacar que para poder utilizarlo, el usuario

debe tener conocimiento previo del lenguaje de programación Prolog, también es prudente advertir que a cada vez que el usuario ingresa un nuevo hecho en la base de conocimientos, el programa le mostrará al usuario un mensaje de alerta que notifica que el hecho fue agregado exitosamente. También muestra el tamaño de la base de conocimientos. Seguidamente la imagen de la demostración de uso del programa.



Para cerrar el programa se debe utilizar la palabra reservada “Exit”.

CONCLUSIÓN PERSONAL

El trabajo realizado nos permitió expandir nuestros conocimientos en el área de los lenguajes de la programación ya que para poder desarrollar un lenguaje de programación utilizando otro similar es necesario conocer aspectos relevantes sobre ambos, es decir se debe manejar el funcionamiento del lenguaje a desarrollar, en nuestro caso Prolog y de igual manera el lenguaje en el cual se desarrolló (Python).

Nos permitió conocer y familiarizarnos con los diferentes tipos de paradigmas de programación, lo cual es de suma importancia para el desarrollo de la lógica del programador así como de las habilidades y aptitudes requeridas para implementar una solución computacional

Esta tarea además de fomentar el trabajo grupal, permitió conocer aspectos claves, de gran importancia y utilidad sobre el lenguaje Prolog, permitiéndonos como estudiantes ampliar nuestros conocimientos en el área informática y mantener un punto de comparación más elaborado respecto a otros lenguajes. El conocimiento de este lenguaje es de suma importancia, ya que nos permite tener una noción de lo que es el comportamiento Prolog, su funcionalidad, características, limitaciones, usos, facilidades y demás aspectos negativos como positivos que poseen todos los lenguajes de programación utilizados en alrededor del mundo.