Manual de usuario

Lenguaje Medievo

Denzel Delgado Urquieta

Darwin Tapia Urrutia

Tabla de contenido

1. Introducción a Medievo

Medievo es un lenguaje de programación que consta en una mezcla del lenguaje C++ y Python con la intencionalidad de agregar caracterizaciones medievales. Asimismo, cuenta con una función única relacionada a la edad media llamada "parias", cuyo uso de concepto difiere ligeramente, sin embargo, mantiene el significado mismo del periodo.

En este manual se presentan los conceptos básicos para iniciarse en la programación con este lenguaje.

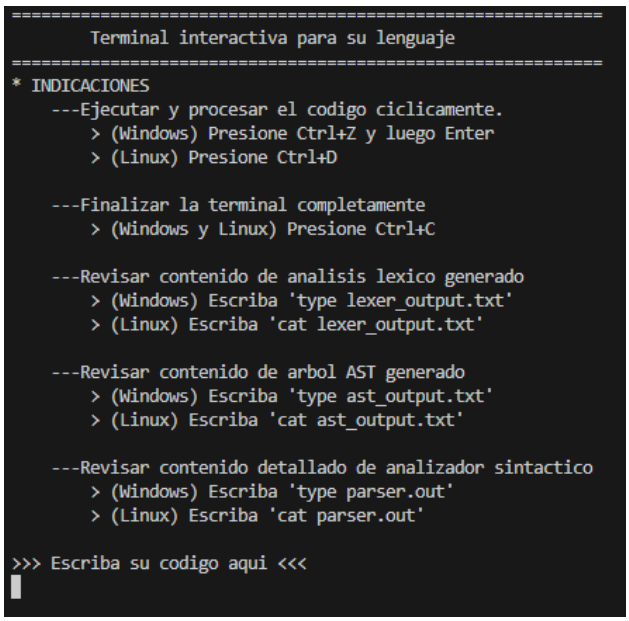
1.1 Tipos de ejecución

Existen dos formas para interactuar con Medievo.

1.1.1 Modo Interactivo

En esta instancia, ejecute el archivo “python test_parser.py” en la terminal que le acomode y se le abrirá la terminal interactiva, en la que podrá escribir código libremente.

Cabe mencionar que, si prefiere este modo, es de suma importancia que lea las indicaciones que le aparecerá al momento de ejecutar el parser en la terminal.



1.1.2 Modo Archivo

En este apartado, debe de escribir “python test_parser.py nombreArchivo.txt” en la terminal. Esto ejecutará el código escrito en el archivo de texto.

2. Tipos de datos

Números: Dígitos de longitud definida por el usuario. Puede incluir números negativos con el comando 'menos' que se explicará más adelante.

Ejemplos admitidos: 1, 0, 12.5, menos 5, menos 13456.241

Cadenas: Conjunto de caracteres en secuencia que representan texto. Se escribe entre comillas dobles.

Ejemplos admitidos: "Hola", " Chao "

Medievo no requiere que se especifique el tipo de dato. Se crea la variable y este se encargará en distinguir de un entero, flotante o cadena.

3. Declarar una variable

La lealtad antiguamente era mucho más estricta, porque una traición a ella conllevaba a la muerte, exilio u otra consecuencia dependiendo de la gravedad. Para que una variable tenga dicha responsabilidad necesitará la palabra **'devote'** con la siguiente estructura:

nombreVariable devote valor;

```
num devote 10;
texto devote "Hola";
```

3.1 Menos

Una variable numérica con valor negativo debe de tener definido **'menos'** para que sea válido.

```
num devote menos 10;
num devote num plunder 20;
```

Nota: La palabra 'plunder' se encuentra definida en el apartado de operadores aritméticos. También, debe respetar el espacio entre 'menos' y su valor. Si lo deja junto lo tomará como una cadena.

4. Concatenar cadenas

Debe de utilizar **'unir'** para realizar concatenaciones entre cadenas de texto. Cabe mencionar que su uso es estrictamente para strings. Si ingresa un número le dará error.

```
cad1 devote "Hola ";
cad2 devote "Mundo";
union devote cad1 unir cad2;
```

5. Comentarios

Medievo permite únicamente comentarios lineales, no admite un bloque de varias líneas. Utilice **'//'** para comentar.

```
// Comentario 1
variable devote 10; // Comentario 2
```

6. Imprimir

Para poder mostrar un valor por pantalla necesitará usar **'print'**. Este comando admite la impresión de cadenas de texto y números en el mismo. Además, puede realizar operaciones aritméticas o lógicas en este.

```
x devote 10;
print(x);

edad devote 20;
print("Edad: ", edad, "Nombre: ", "Pepito");
```

7. Operadores

Como en todo lenguaje de programación, debe de existir alguna forma de relacionar las variables y producir un resultado.

7.1 Operadores aritméticos

7.1.1 Sumar

El operador **'inherit'** representa la unión de dos valores, es decir, recibir las tierras por herencia de un antepasado. Se utiliza para sumar dos valores.

```
suma devote 10 inherit 20;
```

7.1.2 Restar

Utilice **'plunder'** para tomar por la fuerza las pertenencias de otra variable. Sirve para restar un valor de otro.

```
resta devote 100 plunder 99;
```

7.1.3 Multiplicar

Escriba **‘forge’** si desea crear algo nuevo en base a dos objetos existentes. Multiplica dos valores.

```
mult devote 2 forge 20;
```

7.1.4 Dividir

El comando **‘cleave’** alude a particiones, como de tierras, grupo social u otros. Divide dos valores.

```
div devote 10 cleave 5;
```

7.1.5 Modulo

‘shatter’ simboliza la sobra del botín tras el saqueo de un castillo; aquello que no pudo dividirse ni ser reclamado. Es el residuo de una división.

```
modulo devote 10 shatter 5;
```

7.2 Operadores de comparación

Los operadores de comparación no se detallarán. Dado que Medievo es un lenguaje que presenta una mezcla de Python y C, no tendrá problema en reconocer las siguientes palabras clave:

- **‘>’** Mayor que
- **‘<’** Menor que
- **‘>=’** Mayor o igual
- **‘<=’** Menor o igual
- **‘==’** Igual a
- **‘!=’** Distinto a

7.3 Operadores lógicos

7.3.1 AND

Se representa como **‘&&’** para realizar comportamientos de conjunción.

7.3.2 OR

Se determina como **‘||’** para disyunciones.

7.3.3 NOT

Se determina como **‘!’** para negaciones.

Dichos operadores lógicos puede utilizarlos tanto en operaciones con resultados booleanos en la declaración de variables o introducirlo como una condición.

```
opAnd devote (10 > 5) && (5 > 20);
opOr devote (20 < 15.2) || (1.234 == 1.233);
opNot devote !(5 < 10);
```

8. Condicionales

Es bien sabido que una traición en la edad media implica un juicio y un resultado posteriormente del mismo. De manera simplificada, **'if-else'** para Medievo es **'judge-exile'**, es decir, si no se llega a un acuerdo será exiliado. Su estructura es representada en la siguiente imagen.

```
judge(condicion) {
        // Bloque de codigo
} exile {
        // Bloque de codigo
}
```

Por ejemplo:

```
x devote 10;
y devote "Hola";
judge(x > 5){
        print(x);
} exile {
        print(y);
}
```

9. Bucles

En esta instancia se presentarán los bucles más básicos para un lenguaje de programación.

9.1 While

Es importante mantenerse en un estado de vigilia ante un conflicto inminente, como lo haría el vigía de un castillo, atento a cualquier comportamiento extraño en los alrededores. El único problema es que no sabe cuándo debe de terminar su patrullaje. Por eso, necesita escribir **'vigil'**. Este tiene la siguiente estructura:

```
vigil(condicion){
        // Bloque de codigo
}
```

Por ejemplo:

```
x devote 5;
vigil(x > 0) {
        print("Valor de x:");
        print(x);
        x devote x plunder 1;
}
```

9.2 For

Una marcha no es improvisada; es totalmente organizada y con un objetivo específico. Los soldados sabían su posición y cuantos pasos debían dar.

Así como las tropas marchaban bajo órdenes de un comandante, **'march'** permite repetir una acción un número determinado de veces. Su estructura es la que se presenta a continuación:

```
march(inicializacion; condicion; actualizacion) {
        // Bloque de codigo
}
```

Ejemplo:

```
march(i devote 5; i < 10; i devote i inherit 1) {
        print("Valor de i:");
        print(i);
}
```

10. Funciones

Medievo contiene dos funciones personalizadas (parias y conquistar), el cual, simplemente se invoca con sus respectivos argumentos. Asimismo, tiene la capacidad de crear funciones.

10.1 Parias

Las parias eran tributos que diversas taifas (digamos que facciones de un reino) debían de pagar, muchas veces para evitar una invasión o mantener una paz frágil. A veces, dichos pagos eran injustos, ya que el monto podía variar en cada ocasión.

El comando **'parias'** aplica este principio a una variable numérica. Esta representa la capital inicial de una facción, y el proceso de la función simula el cobro de un tributo aleatorio, resultando en una reducción incierta de la capital. La imagen expone la forma de invocarlo:

```
parias(variable);
```

Ejemplo:

```
montoInicial devote 100000;
parias(montoInicial);
```

Salida:

```
Impuesto: '57'%
Valor de entrada: 100000, Valor final: 43000.0
```

10.2 Conquistar

Las conquistas eran eventos cruciales a lo largo de la historia. Dos bandos opuestos con numerosos ejércitos se enfrentaban por dominio territorial, político, moral y reputacional.

La victoria no siempre dependía del número de soldados, sino del ingenio y estrategia. La batalla de Azincourt es un ejemplo claro, donde se evidenció la derrota francesa ante los ingleses que se encontraban en desventaja numérica.

En nuestro lenguaje, para términos simples, la función **'conquistar'** le dará la victoria al bando que tenga mayor ejército. Este requiere de dos argumentos, el nombre del pueblo a conquistar y una variable entera que representa la fuerza militar del atacante. Esta es su estructura:

```
conquistar(nombrePueblo, ejercito);
```

Ejemplo:

```
ejercito devote 15000;
conquistar("Valencia", ejercito);
```

Salida:

```
Pueblo 'Valencia' tiene defensa 1126. Ejército disponible: 15000
¡'Valencia' ha sido conquistado con éxito!
```

10.3 Crear una función

Los decretos son ordenes que debían ser cumplidas sin cuestionamientos. Este es claro, tiene un nombre como identificación y se ejecutaba cuando se requería.

Utilice **'decree'** para crear la función y opcionalmente **'yield'** para devolver un valor. Tiene la siguiente forma:

```
decree nombreFuncion(argumento1, ..., argumentoN){
        //Bloque de codigo
        yield variable; //Además de variable, puede ser un numero o texto
}
```

También, es posible escribir solamente **'yield'**.

Algunos ejemplos:

```
decree sumar(a, b){
        resultado devote a inherit b;
        yield resultado;
}
num devote sumar(10, 20);
```

```
decree imprimir(numero){
        judge(numero == 20) {
                print("No se permite este numero");
                yield;
        } exile {
                judge(numero >= 15 && numero < 20) {
                        print("Numero aceptado. Puede ser mejor");
                } exile {
                        print("Numero admitido: ", numero);
                }
        }
}

imprimir(10);
```

Nota: Si crea una variable que llame a una función que no contenga yield, su valor será None.