UNIVERSIDAD MARIANO GALVEZ DE GUATEMALA

FACULTAD DE INGENIERA EN SISTEMAS, CHIQUIMULILLA SANTA ROSA

INGA. MARIELOS CARPIOS

COMPILADORE

PROYECTO DE

FASES DE ANALISIS DE UN COMPILADOR EN PYTHON

INTEGRANTES:

Nombre: Carlos Abelino Jimenez García

Carne: 1790-15-10692

Nombre: Kender Antonio Sagastume

Carne:1790-15-12807

Índice

Introducción

El compilador es el encargado de realizar la traducción de un lenguaje pseudo natural a un lenguaje binario entendible para la computadora, para lo cual se vale de etapas para poder generar el archivo binario, primero que nada el análisis de léxico, posteriormente se verifica la sintáctica y por ultimo su significado semántico, en el presente texto veremos cómo realiza cada una de las fases de compilación y el compilador que se realizó, desde su diseño hasta su implementación en java, siendo este compilador para el lenguaje script Python.

Objetivos

- Poder desarrollar las partes de análisis de un compilador desde cero, sin utilizar herramientas externas, pero no obstante siendo los objetivos principales

- Crear el analizador de léxico para identificar los toquen que el lenguaje está reconociendo y esto poder ser utilizado en las siguientes fases del compilador, este utilizando el concepto de máquina de estados finitos

- luego poder desarrollar el analizador sintáctico con recuperación de errores, usando el concepto de pasar el error y obviarlos, en esta parte se intentó verificar la gramática regular mediante una máquina de estado apoyado de una pila

Por último, el desarrollo de la unidad de análisis semántico en el cual verificaremos los tipos de datos que se poseen en Python y su compatibilidad, aclarando que Python es un lenguaje script conde se usa mucho en concepto de object , por lo que su análisis semántico , no es de lo más fuerte ni complicado.

Análisis de léxico

Esta es la etapa de la fase del compilador donde se se buscan lo tokens validos que posee cierto lenguaje en nuestro caso Python posee los siguientes

```
//------------------------ TABLA DE SIMBOLOS ---------------------------------
        String Tabladesimbol[]={"#","def","class",":",".","\"","'","print","+","-","*","/","//","%","='
            "true","false","and","or","not","==","!=","<",">","<=",">=","(",")","[","]",
            "if","for","in","while","raw_input","print","int","float","str","tuple","list","dict"};
//-----------------------------------------------------------------------------
```

Como podemos ver en la imagen de la tabla de símbolos, esta no sirve para las tres etapas incluyendo el análisis de léxico.

El análisis de léxico, lo que se realiza es un parseo leve en búsqueda de toquens propios de lenguaje luego son mostrados, (una idea básica).

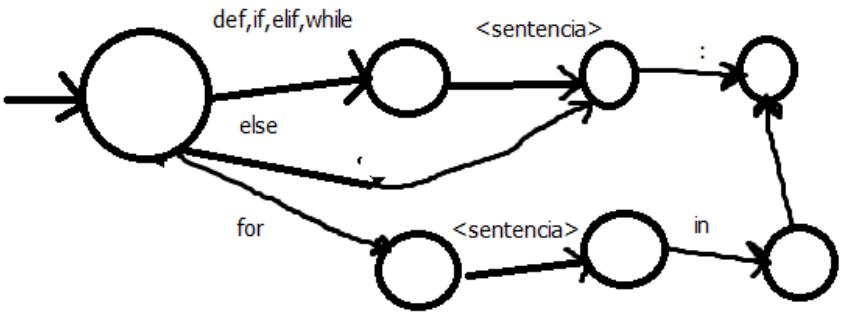
El código es de la siguiente manera:

```
    private void analisisLexico() {
        String[] lineas=txtcodigo.getText().split("\n");
        // primero
        txtresultados.setText("");
        txterror.setText("");
        String Lexico="";
        for(int i=0;i<lineas.length;i++){
            //System.out.println(lineas[i]);
            StringTokenizer stlinea=new StringTokenizer(lineas[i]," ");
            while (stlinea.hasMoreTokens()) {
                        String token=stlinea.nextToken();
                        boolean isvalid=false;
                        for(int k=0;k<token.length();k++){
                            if(token.charAt(k)=='ñ' || token.charAt(k)=='Ñ' || token.charAt(k)=='´' || toke
                                txterror.append("Error en token invalido "+token.charAt(k)+" en linea: "+(i
                                isvalid=true;
                            }
                        }

                        if(!isvalid && isSimboltable(token))
                          Lexico=Lexico+token+"\n";
                          // lex.add(tokek



             }

        }
        txtresultados.append(Lexico);
```

Análisis sintáctico

El análisis sintáctico es la parte del compilador que verifica que las cosas estén estructuradas bien, que poseen el orden necesario y que se cumplan ciertas reglas de producción, nosotros proponemos una Gramatica libre de contexto que sigue el siguiente autómata:

Analisis de Lexico



Podemos ver que realmente Python no es un lenguaje rígido para verificar las instrucciones, de hecho es la razón por cual es considerado un buen lenguaje para aprender a programar, lo que realizamos en el código es la verificación que se hayan pasado por los estado que le corresponde.

El código es el siguiente:

```
private void AnalisisSintactico() {
     String[] lineas=txtcodigo.getText().split("\n");
   // primero
   txtresultados.setText("");
   txterror.setText("");
   String Lexico="";
   boolean error=false;
   for(int i=0;i<lineas.length;i++){
       if(!isclosebreak(lineas[i])){
          txterror.append("error no se cerro ( ) o [ ] o comillas, en linea: "+(i+1)+"\n");
          error=true;
       }

       //System.out.println(lineas[i]);
       StringTokenizer stlinea=new StringTokenizer(lineas[i]," ");
       while (stlinea.hasMoreTokens()) {
                    String token=stlinea.nextToken();
            lex.add(token);

         }
       boolean waitres=false,isfor=false;
       String tok="";

         while(!lex.isEmpty()){
             tok=lex.peek();
             lex.poll();
```

Podemos apreciar que manejamos minusiosamente cada uno de los posibles errores, y sus recuperación


```
                if((tok.equalsIgnoreCase("def") || tok.equalsIgnoreCase("if") || tok.equalsIgnoreCase("eli
                {
                    error=true;
                   // if()
                   //  JOptionPane.showMessageDialog(this,"Analisis Sintactico Exitoso");
                }else if(tok.equalsIgnoreCase("def") || tok.equalsIgnoreCase("if") || tok.equalsIgnoreCase
                {
                   //if(tok.equalsIgnoreCase("for")){
                     //  isfor=true;
                   //}
                    waitres=true;
                }
            }
            if(waitres){
                if(!tok.equalsIgnoreCase(":")){
                        txterror.append("Error falta ':' separado al final de la expresion, en linea: "+(i-
                          waitres=false;
                        error=true;
                }
            }
        }

        if(!error ){
            JOptionPane.showMessageDialog(this,"Analisis Sintactico Exitoso");
            txtresultados.append("Analisis de sintatico exitoso");
        }
    }

    private boolean ordentokencorrect() {
```

Dado que son un pocas la palabra reservada podría incluso no ser necesario el uso de un diccionario o la tabla de símbolos, aunque se deja a discreción.

Entre las partes más poderosas que posee el analizador es verificación de apertura y cierre de algunas de los agrupadores más usados como lo son ( ) [ ] “ ”,

A continuación, se puede ver el modulo encargado de verificar esto

```
private boolean isclosebreak(String linea) {
    int abre=0;

            for(int i=0;i<linea.length();i++){
                if(linea.charAt(i)=='[' || linea.charAt(i)=='('){

                    abre++;
                }
            }
             for(int i=0;i<linea.length();i++){
                 if(linea.charAt(i)==']' || linea.charAt(i)==')'){
                    abre--;
                }
            }
            boolean iscomilla=false;

             for(int i=0;i<linea.length();i++){
                 if(linea.charAt(i)=='\"'){
                     if(iscomilla){
                         abre--;
                         iscomilla=false;
                     }else{
                         iscomilla=true;
                     abre++;
                     }

                }
            }
```

Análisis semántico

Este es el encargado de que las operaciones tengan un sentido lógico de lo que hacen congruencia de datos, para lo cual, se implementó un analizador de asignaciones y comparaciones de compatibilidad de tipos de dato

El analizado fue codificado de la siguiente manera

```
    Vector<variable> idvar=new Vector<>();
private void analissemantico(){
        String[] lineas=txtcodigo.getText().split("\n");

    // primero
    txtresultados.setText("");
    txterror.setText("");

    for(int i=0;i<lineas.length;i++){
        String[] TOken=lineas[i].split(" ");
        System.out.println(""+TOken.length);
        if(TOken.length==3 ){
            variable varb=new variable();
            varb.nombre=TOken[0];
            varb.tipo=vertipo(TOken[2]);
            varb.valor=TOken[2];
            idvar.add(varb);
            System.out.println(""+ varb.tipo);
        }else if(TOken.length==5){
            System.out.println("token 1 "+TOken[2]+" token 2 "+TOken[4]);
            String tip1=buscar(TOken[2]);
            String tip2=buscar(TOken[4]);
            System.out.println(" "+tip1 + " "+tip2);
            if(tip1.equals(tip2)){
                txtresultados.append("tipos de datos valido\n");
            }else{
                txtresultados.append("tipos de datos invalido para operacion\n");
            }
```

Podemos ver además que se tuvo que implementar un reconocedor de tipos de datos para su indexación de tipos de variable dentro de la estructura de datos

```
// idvar.clear();
 }

 private String vertipo(String dato) {
      boolean isnum=true,isletra=false;
        for(int i=0;i<dato.length();i++){
            if(dato.charAt(i)<'0' || dato.charAt(i)>'9'){
             isnum=false;
            }
        }
        if(isnum)return "number";
        else if(dato.charAt(0)=='\"' && dato.charAt(dato.length()-1)=='\"'){
             return "string";
        }

        return "object";
 }

 private String buscar(String vars) {
     for(int i=0;i<idvar.size();i++){
             if(idvar.get(i).nombre.equalsIgnoreCase(vars)){
                 return idvar.get(i).tipo;
             }
     }
     return "notfound";
 }
```

Además de las búsquedas dentro del anexado de páginas que sale en aproximadamente 10 lineas.

```
    private String buscar(String vars) {
        for(int i=0;i<idvar.size();i++){
                if(idvar.get(i).nombre.equalsIgnoreCase(vars)){
                    return idvar.get(i).tipo;
                }
        }
        return "notfound";
    }


}
```

Conclusión

-Las tres fases de análisis son de vital importancia para la realización de compilación de esta, es necesario tener bases solida teórica, prácticas para poder construir estos analizadores

- el lenguaje de programación Python es un lenguaje bastante amigable que su desarrollo de su compilador no presenta grandes complicaciones por ser un leguaje script, a diferencia de compiladores para c++,.

- es necesario tener conocimientos de estructura de datos para su realización tales como tablas hash, vectores, listas, colas, pilas, arboles.

Recomendaciones

- Se debe de tratar de adquirir base teórica como practica para poder desarrollar efectivamente los analizadores en cualquier entorno que se desee.

- Las reglas de producción para la generación de el árbol pueden ser manejadas implícitamente mediante el uso de autómatas de pila, e incluso poder gestionar mejor los errores en tiempo real de análisis del lenguaje de dicha gramática libre de contexto.

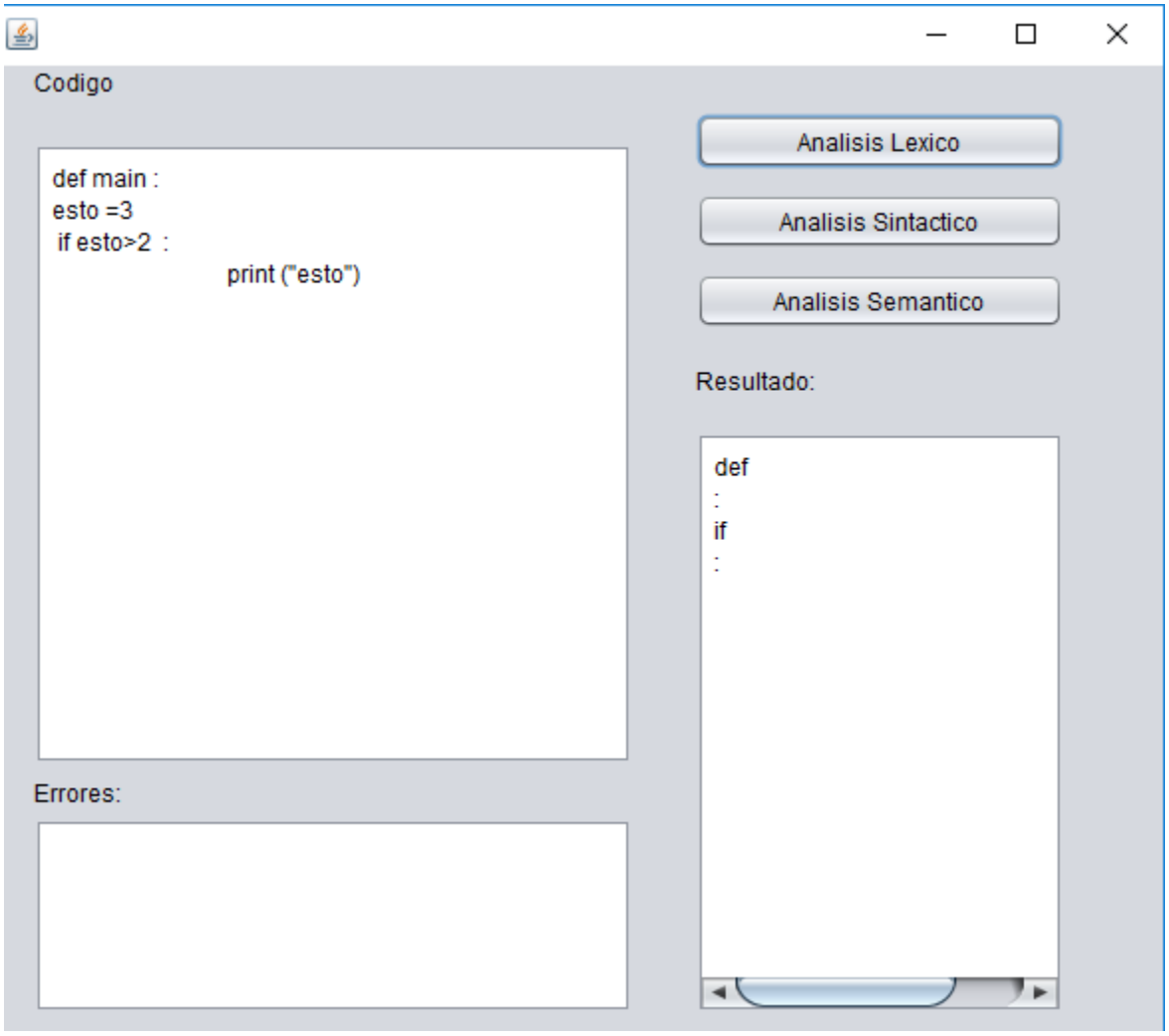
- Python posee una forma de ver los objetos como tipo Object o genéricos por lo que el analizado semántico no detecta casi incompatibilidades de datos, acepciones, por lo que al desarrollarse el módulo de análisis semántico de debe de observarse cuidadosamente esto para no realizar programación demás

Bibliografía

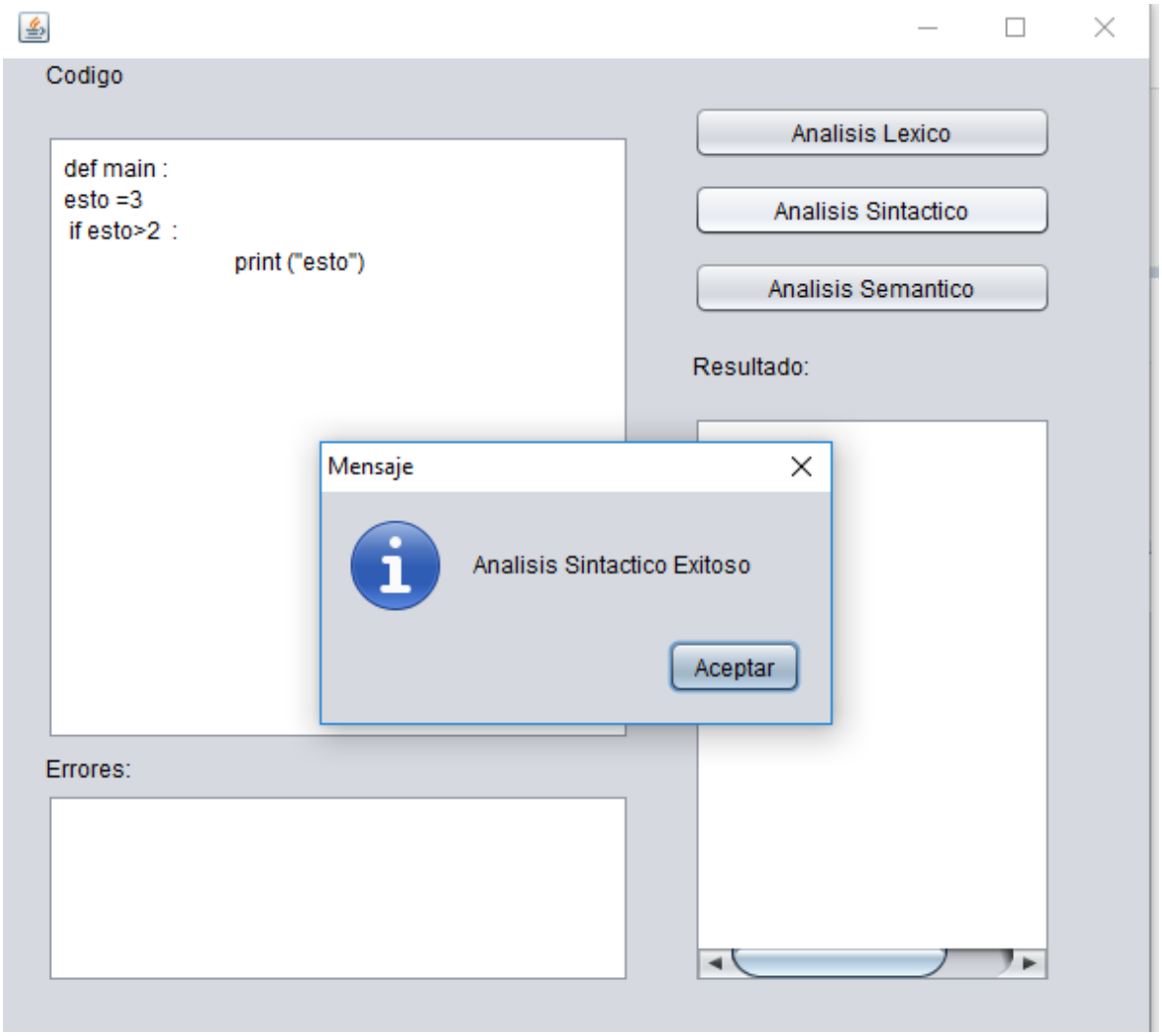
-Compiladores principios, técnicas y herramientas, 2da Ed. Alfred V. Aho, Monica S. Lam Ravi Sethi jeffrey D. Ullman

ANEXOS

Análisis de lexico



Análisis Sintactico



Analisis Semantico