Universidad de San Carlos de Guatemala
Facultad de Ingeniería

Curso: **Lenguajes Formales y de Programación**

Manual Técnico: Proyecto 3

Erik Vladimir Girón Márquez
Carnet # 200313492
Sección A+

Guatemala, 04 de Noviembre de 2005

Introducción.

Conjuntivity es un intérprete para operaciones de conjuntos, implementando analisis léxico y sintáctico sobre el código escrito por el usuario y al mismo tiempo generando los resultados para las operaciones especificadas en el código.

El programa implementa un diseño puramente orientado a objetos, que junto con las carácterísticas de programacion por eventos de Visual Basic, simplificaron en gran parte el desarrollo visual e interactivo de la aplicación, sin embargo, debido a la falta de características de "bajo nivel"(como por ejemplo, la falta de algún equivalente a la función unputc() o incluso getch() del lenguaje c) complicó bastante la creacion del scanner y del parser.

Se utilizó entonces para el desarrollo del proyecto, la plataforma .net utlizando como se ha dicho anteriormente, el lenguaje Visual Basic. Permitiéndose compilar la aplicación bajo cualquier Visual Studio .net actual que disponga de Visual Basic, siempre y cuando la version del framework sea mayor o igual 1.0.

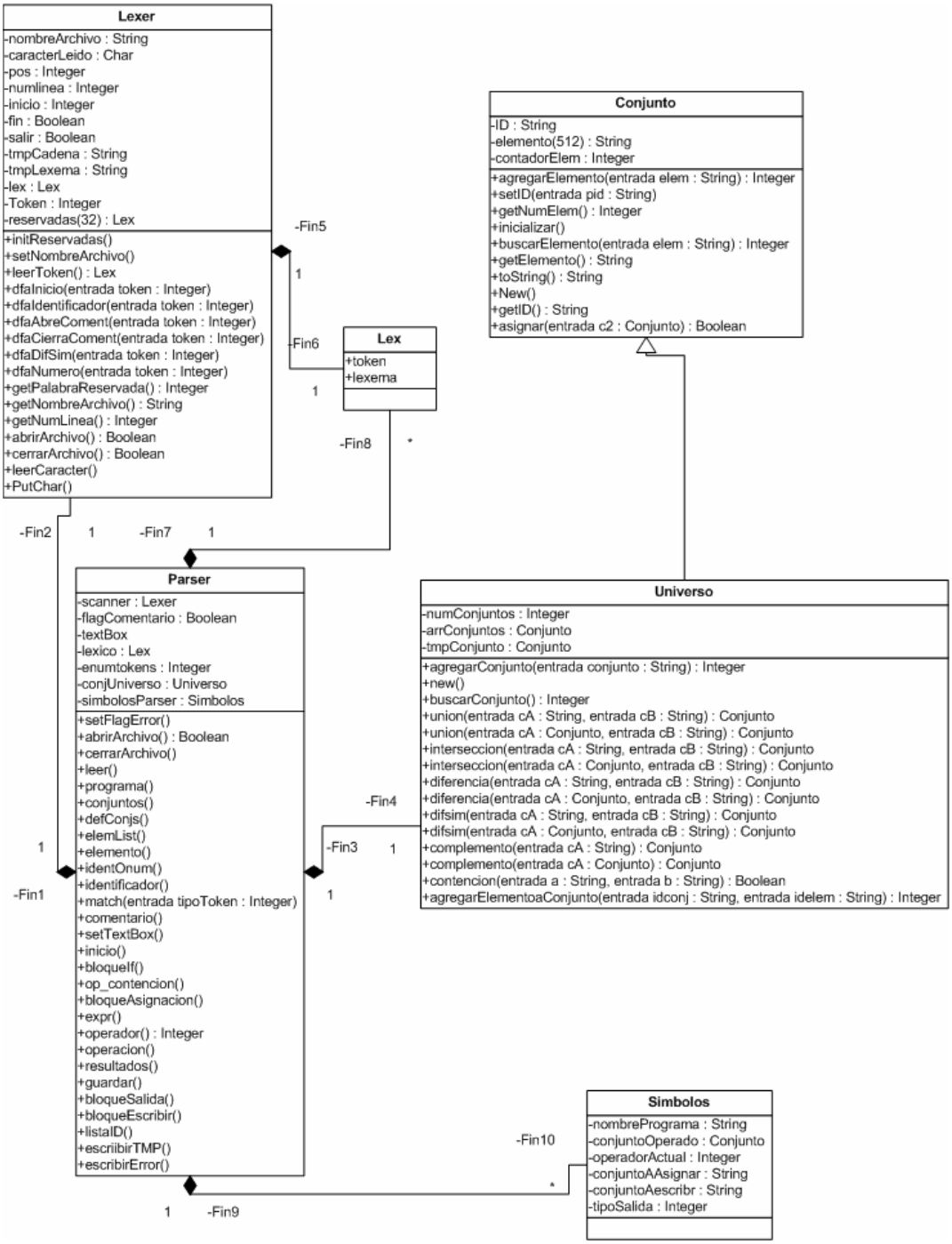
En este documento, el programador se podrá guiar por medio de diagramas UML a través del diseño del proyecto y de cada una de las clases que la componen, además de poder examinar las expresiones regulares utilizadas, su implementación por medio de autómatas finitos determinísticos y la gramática libre de contexto en formato Backus-Naur

Índice

Diseño General de las Clases.

A continuación se presenta el diseño general de las clases que componen la lógica del proyecto, por medio de diagramas UML. No se incluyen las generadas por el editor de formas de VB.NET, ya que éstas no influyen en la lógica de la aplicación.

Jerarquía de Clases



Elementos del Lenguaje

Expresiones Regulares:

Aquí podrá encontrar la definición formal de los elementos lexicográficos que componen el proyecto para la solución, dando el método del arbol que genera DFA para cada expresión regular, con sus respectivos diagramas y tablas de transición.

Expresion Regular:
LETRA->(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)

Expresion Regular:
DIGITO-> (0|1|2|3|4|5|6|7|8|9)

Expresion Regular:
IDENTIFICADOR-> LETRA(LETRA|NUMERO)*

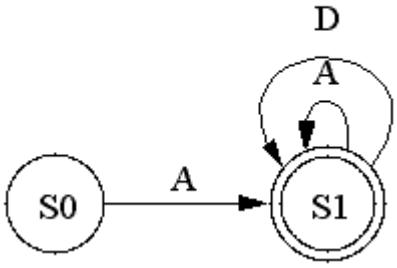
Matriz de Transición:

	L	N	Tipo
S0	S1		
S1	S1	S1	Aceptacion

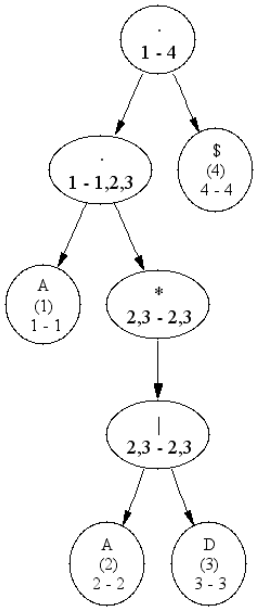
Funcion Follow:

Hoja	Función siguiente
1	2,3,4
2	2,3,4
3	2,3,4
4	

Diagrama de Transicion:



Arbol



Expresion Regular:

CIFRA-> (DIGITO)|(DIGITO)*

Matriz de Transición:

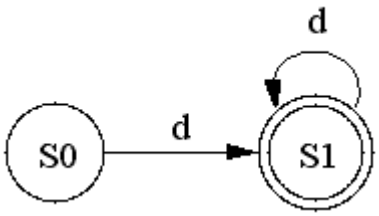
	d	Tipo
S0	S1	
S1	S1	Aceptacion

Funcion Follow:

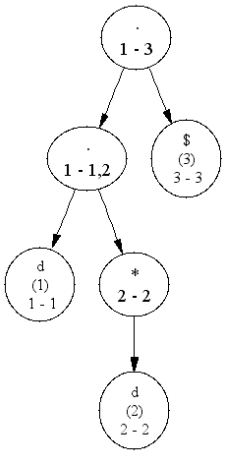
Función siguiente

Hoja	Siguiente
1	2,3
2	2,3
3	

Diagrama de Transicion:



Arbol



Expresion Regular:

ELEMENTO-> (IDENTIFICADOR|DIGITO+)

Matriz de Transición:

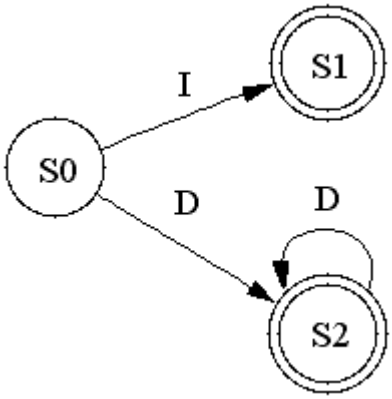
	I	D	Tipo
S0	S1	S2	
S1			Aceptacion
S2		S2	Aceptacion

Funcion Follow:

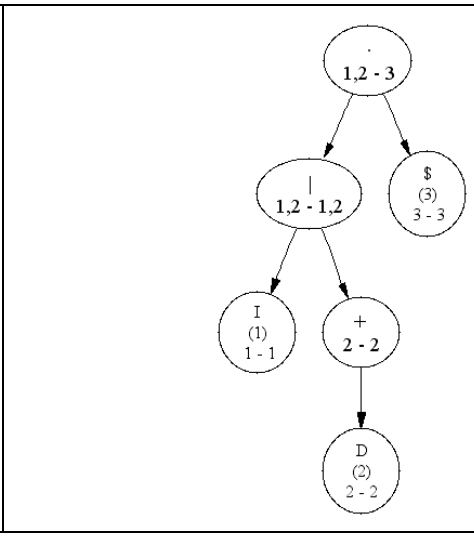
Función siguiente

Hoja	Siguiente
1	3
2	2,3
3	

Diagrama de Transicion:



Arbol



Implementacion del DFA:

```
Public Function leerToken() As lex
        Dim tipo As Integer
        tipo = 1

        Dim tokenA As Token
        Dim regLex As lex
        tokenA = tokenA.s_inicio
        fin = False
        salir = False
        tmpLexema = ""
        While (salir = False) And (EOF(1) =
False)
            leerCaracter()
            tmpCadena = caracterleido
            tmpLexema += tmpCadena
            Select Case (tokenA)
                Case Token.s_inicio
                    dfaInicio(tokenA)
                Case Token.tk_identificador
                    dfaIdentificador(tokenA)
                Case Token.tk_digito
                    dfaNumero(tokenA)
                Case Token.tk_num
                    dfaNumero(tokenA)
                Case Token.tk_sm_lesst
                    dfaDifSim(tokenA)
                Case Token.tk_sm_diagonal
                    dfaAbreComent(tokenA)
                Case Token.tk_op_intersec
                    dfaCierraComent(tokenA)
                Case Else
                    salir = True
                    tmpLexema =
tmpLexema.TrimEnd(caracterleido)
                    PutChar()
            End Select
        End While
        If (EOF(1)) Then
            salir = 1
        End If
        If (tokenA = Token.tk_identificador)
Then
            If (tipo = 1) Then
                tokenA = getPalabraReservada()
            Else
                tokenA = Token.s_normal
            End If
        End If
        regLex.lexema = tmpLexema
        regLex.token = tokenA
        Return regLex
    End Function
```

Gramática Libre de Contexto:

A continuación se presenta la gramática libre de contexto (Nivel 2 según la jerarquía de Chomsky) en formato Backus-Naur, que se utilizó para implementar un parser LR simple y no recursivo, además de definir formalmente la gramatica del pseudo-lenguaje que interpreta el programa.

```
Input :=   "programa" identificador
```

```
          "conjuntos" defConj
          "Inicio"   bloqueInicio
          "Resultados" resultados
          "fin"

defConj :=            defConj asignacionElem
                      | asignacionElem

asinacionElem :=      identificador "=" conj

conj :=    "{" elemLista "}"

elemLista :=          elemLista "," elemento
                      | elemento

elemento :=           identificador
                      | numero
                      | Є

op_union :=           identificador "+" identificador
op_dif     :=         identificador "-" identificador
op_difsim :=          identificador "<>" identificador
op_inter :=           identificador "*" identificador
op_comp := identificador "@"
op_cont := identificador "%" identificador

operacion :=          op_union
                      | op_intersec
                      | op_dif
                      | op_difsim
                      | op_comp
                      | op_intersec
                      | op_cont

expr :=    expr operacion
           | operación


asignacionOp :=       identificador "=" expr ";"

asignacion :=         asignacionOp
                      | asignacionElem ";"

bloqueAsignacion :=   bloqueAsignacion asignacion
                                  | asignacion

bloqueIf :=           "si" op_cont "entonces" bloqueAsignacion "fin si;"
                      |"si" op_cont "entonces" bloqueAsignacion "sino"
            bloqueAsignacion "fin si;"

bloqueInicio :=       bloqueInicio bloqueAsignacion
                      | bloqueInicio bloqueIf
                      | bloqueAsignacion
                      | bloqueIf

resultados :=         exprSalida exprEscribir
                      | exprEscribir
```

```
exprSalida :=         “salida” tipoSalida

tipoSalida :=         tipoSalida chardev
                      | chardev

chardev := “pantalla”
                      | “impresora”
                      | “archivo”

exprEscribir :=       exprEscribir cmdEscribir
                      | cmdEscribir

cmdEscribir :=        “escribir” “(“ listaConj “)”

listaConj :=          listaConj “,” identificador
                      | identificador
```

Nota: Los símbolos terminales se represetan entre comillas.

Limitaciones y Extras.

Limitaciones

Precedencia de operadores

Debido a que se implementó un parser LR no recursivo y muy rústico, no fue posible implementar precedencia en las operaciones de conjuntos.

Comentarios

Es posible implementar comentarios en el código, sin embargo debido a la técnica de análisis sintactico, puede que en algunas partes los comentarios no sean ignorados por el parser.

Numero limitado de conjuntos y elementos

A causa de que se implementó tanto los elementos como los conjuntos de manera estática(sin utilizar ninguna estructura de datos), sólo es posible un máximo de 512 conjuntos definidos en el universo, cada uno con un máximo de 512 elementos de tipo string en total en cada conjunto.

Salida a impresora

Por la falta de familiaridad con el lenguaje, la opcion de salida a impresora no fue implementada.

Extras

Editor de Textos

Podrá editor el código del archivo de entrada previo a interpretarlo desde la ventana del editor que está en la ventana principal.