> form.cs		}	
using System;	this.buttonX12.Click		<pre>sr.Close();</pre>
using	+= new	private void	}
System. Collections. Gen	System. EventHandler (th	buttonX5_Click(object	
eric;	is.buttonX12_Click);	sender, EventArgs e)	}
using		{	
<pre>System.ComponentModel;</pre>	//this.button2.Click		private void
using System.Data;	+= new		buttonX8_Click(object
using System.Drawing;	System.EventHandler(th	OpenFileDialog	sender, EventArgs e)
using System.Text;	is.button1_Click);	openFileDialog1 = new	{
using	Code. KeyUp	OpenFileDialog();	if (Code
System. Windows. Forms;	$+= (s, e) => {$		!= nu11)
using System. IO;			{
using	List <string></string>	openFileDialog1.Initia	
Lexical_Analyzer;	DictionaryList = new	<pre>1Directory = @"C:\";</pre>	DialogResult result =
using Syntax_Analyzer;	List <string>(new</string>		MessageBox.Show("Clean
using	string[] { "Backup",	openFileDialog1.Title	Code Workspace?",
Semantics_Analyzer;	"commence", "deploy",	= "Browse Militari	
using	"go" }. ToList());	Solution";	"Clearing Option",
System. CodeDom. Compile			
r;	clsIntelliSense.AutoCo		MessageBoxButtons.YesN
using	mpleteTextBox(Code,	openFileDialog1.CheckF	o);
System. Diagnostics;	listBox1,	ileExists = true;	if
using	DictionaryList, e);		((result ==
Microsoft. CSharp;	};	openFileDialog1.CheckP	DialogResult.Yes) &&
using	}	athExists = true;	(Code != null))
System. Configuration;	,	,	{
using	int lines = 0;		·
System. Threading;	List <int></int>	openFileDialog1.Defaul	Code.Clear();
using	linetokens = new	tExt = "mltr";	osas, sisai (, ,
System. Runtime. Interop	List <int>();</int>	mier,	MessageBox. Show("Clean
Services;	Bise (ine) (),	openFileDialog1.Filter	ed!");
using	LexicalAnalyzer lex =	= "Militari Solutions	}
System. Text. RegularExp	new LexicalAnalyzer();	(*. mltr)   *. mltr   All	else
ressions;	new Bekreammaryzer (),	files (*.*)  *.*";	if(result ==
using	/*Literal	11105 (1.17)	DialogResult.No) {
IntellisenceTextBox;	List*/	openFileDialog1.Restor	Dialognesait. No,
using System. Linq;	List <string></string>	eDirectory = true;	MessageBox.Show("No
doing bystem. Lind,	intlist = new	edificationy true,	Changes");
namespace Militari	List <string>();</string>		changes /,
{	List(string)	openFileDialog1.ReadOn	else
public partial	doublelist = new	lyChecked = true;	CISC
class Form1 :	List <string>();</string>	Tychecked true,	MessageBox.Show("Code
DevComponents. DotNetBa	List(string)	openFileDialog1.ShowRe	Workspace is Empty!");
r. Metro. MetroAppForm	stringlist = new	adOnly = true;	"OIKSpace IS Empty. ),
1. Metro, MetroApprorm	List <string>();</string>	adonly - tide,	J
string fname =	List(string)	if	}
string mame -	charlist = new	(openFileDialog1. ShowD	}
,	List <string>();</string>	ialog() ==	J
string consolewrt = "";	List\string>(), List\string>	DialogResult.OK)	privata vaid
	boolist = new	Dialognesuit. On)	private void buttonX9 Click(object
<pre>string function = "";</pre>	List <string>();</string>	fname	sender, EventArgs e)
	List\string/(), List\string>	-	sender, Eventargs e)
string globdeclare = "";		-	l
string main =	<pre>funclist = new List<string>();</string></pre>	openFileDialog1.FileNa	this.Close();
String Main -	LISt\StIIng/();	me;	this, close();
,		StroomD1	}
	private void	StreamReader sr = new	
public Form1()	Form1_Load(object	StreamReader(fname);	private void
1	sender, EventArgs e)	C-1- T	buttonX6_Click(object
In:t::-1::C	1	Code. Text =	sender, EventArgs e)
<pre>InitializeComponent();</pre>		sr.ReadToEnd();	1

if (fname	if		return
== "")	(res ==	dataGridViewX3. Rows. Cl	sem;
{	DialogResult.Cancel)	ear(); lex =	}
openFileDialog1.Filter	·	new LexicalAnalyzer();	public
= "Militari	return;		List <semanticsinitiali< td=""></semanticsinitiali<>
Files * mltr";	} fname	Initializer Lexical = new Initializer();	zer.Tokens> tokenDumps(List <tokens< td=""></tokens<>
DialogResult res =	=	string	> tokens)
openFileDialog1.ShowDi	openFileDialog1.FileNa	<pre>txt = Code.Text;</pre>	{
alog();	me;	lex =	T
if (res ==	MessageBox.Show(fname)	Lexical.InitializeAnal yzer(txt, lex);	List <semanticsinitiali zer.Tokens&gt; token =</semanticsinitiali 
DialogResult.Cancel)	;	fact (one, ton),	new
{			List <semanticsinitiali< td=""></semanticsinitiali<>
return;	StreamWriter sw = new StreamWriter(fname);	DisplayTokens(lex);	zer.Tokens>();
}	Streamwifter (mame),	if	SemanticsInitializer.T
fname	sw. WriteLine (Code. Text	(lex.invalid == 0 &&	okens t = new
= energileDieleg1 FileNe	);	lex. token. Count != 0)	<pre>SemanticsInitializer.T okens();</pre>
openFileDialog1.FileNa me;	sw.Flush();	l	foreach
		<pre>buttonX3.Enabled =</pre>	(var item in tokens)
MessageBox. Show(fname)	sw.Close();	true;	{ t =
,	}	else {	new
StreamWriter sw = new			SemanticsInitializer.T
StreamWriter(fname);	private void buttonX11 Click(object	<pre>buttonX3.Enabled = false;</pre>	okens();
sw.WriteLine(Code.Text	sender, EventArgs e)	raise,	t.setAttributes(item.g
);	{	<pre>buttonX4.Enabled =</pre>	etAttributes());
sw. Flush();	Code. Cut();	false;	t.setLexemes(item.getL
Sw. I Tush () ,	}	<pre>dataGridViewX1.Show();</pre>	exemes());
sw.Close();		}	,
}	private void buttonX2 Click(object	ì	<pre>t. setLines(item. getLin es());</pre>
J	sender, EventArgs e)	J	C3()),
private void	{	private	t.setTokens(item.getTo
<pre>buttonX7_Click(object sender, EventArgs e)</pre>	richTextBoxEx1.Text =	SemanticsInitializer SemanticsStart(List <se< td=""><td>kens());</td></se<>	kens());
Sender, Eventargs e/	null;	manticsInitializer.Tok	token.Add(t);
	consolewrt	ens> tokens)	}
<pre>openFileDialog1.Filter = "Text Files *.txt";</pre>	= "";	{	return token;
TOAU TITES TO UAL ,	globdeclare = "";	SemanticsInitializer	}
openFileDialog1.ShowDi	main = "";	sem = null;	
alog(); fname =	lex = new LexicalAnalyzer();	try {	private void DisplayTokens(LexicalA
openFileDialog1.FileNa	Lexical Analyzei (),	sem =	nalyzer lex)
me;	buttonX3.Enabled =	new	{
if (fname == "")	false; if	SemanticsInitializer(t okens);	string result = "Successfully
{	(Code. Text != "")	}	Executed.";
P. 1. 1. 1. P. 1.	{	catch	int ctr =
openFileDialog1.Filter = "Militari	dataGridViewX1.Rows.Cl	(Exception e)	0, id = 1;
Files   *. mltr";	ear();	ι	LexGrid.Rows.Clear();
D. 1. D	1 . 0 . 177 . 70 . 7	MessageBox. Show(e.Mess	1 . 0
DialogResult res = openFileDialog1.ShowDi	dataGridViewX2.Rows.Cl ear();	age);	<pre>dataGridViewX4.Rows.Cl ear();</pre>
alog();	-ui (, ,	J	(V),

		C ::::::1:	
if		S_initialize.errors.ge	T
(lex.invalid != 0)	private int	tErrorMessage(),	List <string></string>
result =	GetErrorLine(int ctr)	S_initialize.errors.ge	ConstantvarList = new
"Encountered " +	{	tLines());	List <string>();</string>
lex.invalid.ToString()	int line =		
+ " error/s.\nPlease	0;	errornum++;	List <string></string>
try again.\n";	int cls =	}	GlobalvarList = new
	0;	else	List <string>();</string>
dataGridViewX1.Rows.Ad	for (int i	{	
d(id , "Lexical	= 0; i <		List <string></string>
Analyzer " + result);	linetokens.Count; i++)	dataGridViewX2.Rows.Ad	LocalvarList = new
•	{	d(i, s);	List <string>();</string>
foreach	cls =	3(1) 5/,	Else selling, () ,
(var token in	linetokens[i];	buttonX4.Enabled =	List <string> ReservedW</string>
lex. token)	if	true;	= new List <string> {</string>
Tex. token/		true,	"company", "unit",
( :e	(ctr + 1 <=	1-+-C: JV:V9 Ch().	"digit", "response",
if	linetokens[i])	dataGridViewX2.Show();	
(token.getTokens() ==	(* . 1)	}	"joe", "hold", "miss",
"INVALID")	return (i + 1);	}	"operation", "struct",
{	}		
	return	public	"PrimaryMission",
dataGridViewX1.Rows.Ad	line;	List <tokenlibrary.toke< td=""><td>"post", "capture",</td></tokenlibrary.toke<>	"post", "capture",
d(id, "Invalid input:	}	nsClass>	"backup", "campaign",
"		tokenDump(List <lexical< td=""><td>"abort", "deploy",</td></lexical<>	"abort", "deploy",
	private void	_Analyzer.Tokens>	"inquire",
+ token.getLexemes()	buttonX3 Click(object	tokens)	
	sender, EventArgs e)	. {	"inorder",
, " on line "	{	`	"otherorder", "phase",
, 0111110	· ·	List <tokenlibrary.toke< td=""><td>"go", "order", "action"</td></tokenlibrary.toke<>	"go", "order", "action"
+ token.getLines() +	SyntaxInitializer	nsClass> token = new	};
"\n");	S initialize = new	List TokenLibrary. Toke	J ,
\II <i>)</i> ,	_		List/string Operators
Ĵ	SyntaxInitializer();	nsClass>();	List <string> Operators</string>
4		m 1 .	T
else		Tokens t =	= new List <string> {</string>
if (token.getTokens()	dataGridViewX2.Rows.Cl	new Tokens();	"+", "-", "*", " <sup>-</sup> /",
	ear();	new Tokens(); foreach	
if (token.getTokens()		new Tokens();	"+", "-", "*", " <sup>-</sup> /",
if (token.getTokens()	ear();	new Tokens(); foreach	"+", "-", "*", "/", "=" }; List <string> disp =</string>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad</pre>	<pre>ear();     int i = 1;     string s;     s =</pre>	new Tokens(); foreach	"+", "-", "*", "/", "=" };
<pre>if (token.getTokens() == "NODELIM") {</pre>	ear(); int i = 1; string s;	new Tokens(); foreach (var item in tokens) {	"+", "-", "*", "/", "=" }; List <string> disp =</string>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad</pre>	<pre>ear();     int i = 1;     string s;     s =</pre>	<pre>new Tokens();     foreach (var item in tokens)     {         t =</pre>	"+", "-", "*", "/", "=" }; List <string> disp =</string>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok</pre>	<pre>new Tokens();     foreach (var item in tokens)     {         t =</pre>	"+", "-", "*", "/", "=" };  List <string> disp = new List<string>();</string></string>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok</pre>	<pre>new Tokens();     foreach (var item in tokens)     {         t = new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: "</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));     if (s !=</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes()</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer"</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: "</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));     if (s !=</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line "</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded") {</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() +</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded") {     int</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line "</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded") {</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") {  dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); }</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;  bool globexists;  bool locexists;  int idn = 0;  int Line =</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() +</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1; dataGridViewX2.Rows.C1</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;  bool globexists;  bool locexists;  int idn = 0;  int Line = 1;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") {  dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); }</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.C1 ear();</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;  bool globexists;  bool locexists;  int idn = 0;  int Line =</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") {  dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else {</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1; dataGridViewX2.Rows.C1 ear();     if</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;  bool globexists;  bool locexists;  int idn = 0;  int Line = 1;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") {  dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); }</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.Cl ear();     if (S_initialize.errors.g</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;  bool globexists;  bool locexists;  int idn = 0;  int Line = 1;  int x = 0;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++;</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1; dataGridViewX2.Rows.C1 ear();     if</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; bool locexists; int idn = 0; int Line = 1; int x = 0;</string></string></pre> ConstantvarList.Clear(
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id,</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.Cl ear();     if (S_initialize.errors.g</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists;  bool globexists;  bool locexists;  int idn = 0;  int Line = 1;  int x = 0;</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(),</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.Cl ear();     if (S_initialize.errors.g etColumn() == 1) {</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; bool locexists; int idn = 0; int Line = 1; int x = 0;  ConstantvarList.Clear( );</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(), token.getTokens(),</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.Cl ear();     if (S_initialize.errors.g etColumn() == 1)     { S_initialize.errors.se</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; bool locexists; int idn = 0; int Line = 1; int x = 0;</string></string></pre> ConstantvarList.Clear(
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(),</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.Cl ear();     if (S_initialize.errors.g etColumn() == 1) {</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; bool locexists; int idn = 0; int Line = 1; int x = 0;  ConstantvarList.Clear( );</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(), token.getTokens(),</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.Cl ear();     if (S_initialize.errors.g etColumn() == 1)     { S_initialize.errors.se</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; bool locexists; int idn = 0; int Line = 1; int x = 0;  ConstantvarList.Clear( );</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(), token.getTokens(),</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.C1 ear();     if (S_initialize.errors.g etColumn() == 1)     { S_initialize.errors.se tLines(S_initialize.er</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; int idn = 0; int Line = 1; int x = 0;  ConstantvarList.Clear();  LocalvarList.Clear();</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(), token.getTokens(),</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.C1 ear();     if (S_initialize.errors.g etColumn() == 1)     { S_initialize.errors.se tLines(S_initialize.er</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; bool locexists; int idn = 0; int Line = 1; int x = 0;  ConstantvarList.Clear();</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(), token.getAttributes()); ; }</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;  dataGridViewX2.Rows.C1 ear();     if (S_initialize.errors.g etColumn() == 1)     { S_initialize.errors.se tLines(S_initialize.er</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; int idn = 0; int Line = 1; int x = 0;  ConstantvarList.Clear();  LocalvarList.Clear();</string></string></pre>
<pre>if (token.getTokens() == "NODELIM") { dataGridViewX1.Rows.Ad d(id, "Proper delimiter expected: " + token.getLexemes() , " on line " + token.getLines() + "\n"); } else { id++; LexGrid.Rows.Add(id, token.getLexemes(), token.getAttributes()); ; }</pre>	<pre>ear();     int i = 1;     string s;     s = S_initialize.Start(tok enDump(lex.token));      if (s != "Syntax Analyzer Succeeded")     {         int errornum = 1;      dataGridViewX2.Rows.Cl ear();         if (S_initialize.errors.g etColumn() == 1)     { S_initialize.errors.se tLines(S_initialize.er rors.getLines() - 1);     } }</pre>	<pre>new Tokens();</pre>	<pre>"+", "-", "*", "/", "=" };  List<string> disp = new List<string>();  bool constexists; bool globexists; int idn = 0; int Line = 1; int x = 0;  ConstantvarList.Clear(); GlobalvarList.Clear(); dataGridViewX3.Rows.Cl</string></string></pre>

for $(x =$			[2]. Value. ToString()
0; x <	}	do	== "digit")
LexGrid. Rows. Count;		,	
X++)	else if	{	{
{	(LexGrid. Rows[x]. Cells	; f	ı
	[2].Value.ToString() == "="	if (LexGrid.Rows[x].Cells	}
/*Constant Semantics*/	LexGrid.Rows[x].Cells[	[2]. Value. ToString()	else {
if	2]. Value. ToString() ==	== "id")	
(LexGrid.Rows[x].Cells	"Numlit"		dataGridViewX3.Rows.Ad
[2]. Value. ToString()	LexGrid.Rows[x].Cells[	{	d(idn++,
== "hold")	2].Value.ToString() ==	,	"TypeMismatch: " +
{	";")	if (constexists =	LexGrid. Rows[x]. Cells[
x++;	1	ConstantvarList.Exists (element => element ==	2].Value.ToString(), " Line: " + Line);
if	(	LexGrid. Rows[x]. Cells[	Line. Line,
(LexGrid.Rows[x].Cells	}	1]. Value. ToString())	}
[2]. Value. ToString()		== true)	
== "unit")	else if		X++;
{	(LexGrid. Rows[x]. Cells	{	
1	[2]. Value. ToString()	1 . C . W. VO D . A .	if
do	== "unit")	dataGridViewX3.Rows.Ad d(idn++, "Multiple	((LexGrid. Rows[x]. Cell s[2]. Value. ToString()
{	{	declaration of	== ";"))
		variable: " +	, , , ,
if	}	LexGrid.Rows[x].Cells[	{
(LexGrid.Rows[x].Cells		1]. Value. ToString(), "	
[2]. Value. ToString()	else {	Line: " + Line);	Line++;
== "id")	dataGridViewX3.Rows.Ad	1	1
{	dataGridviewAS. Rows. Ad d(idn++,	,	l
	"TypeMismatch: " +	else {	} while
if (constexists =	LexGrid.Rows[x].Cells[		((LexGrid.Rows[x].Cell
ConstantvarList.Exists	2].Value.ToString(), "	ConstantvarList.Add(Le	s[2]. Value. ToString()
(element => element ==	Line: " + Line);	xGrid. Rows[x]. Cells[1]	!= ";"));
LexGrid. Rows[x]. Cells[	1	.Value.ToString());	}
1].Value.ToString()) == true)	}	doublelist.Add(LexGrid	if (LexGrid.Rows[x].Cells
tiuc)	x++;	. Rows[x]. Cells[1]. Valu	[2]. Value. ToString()
{	,	e.ToString());	== "company")
	if		{
dataGridViewX3.Rows.Ad	((LexGrid.Rows[x].Cell	}	
d(idn++, "Multiple	s[2].Value.ToString() == ";"))	1	do
declaration of variable: " +	; ))	,	1
LexGrid. Rows[x]. Cells[	{	else if	(
1].Value.ToString(), "		(LexGrid.Rows[x].Cells	if
Line: " + Line);	Line++;	[2]. Value. ToString()	(LexGrid.Rows[x].Cells
,		== "="	[2]. Value. ToString()
}	}	LexGrid. Rows[x]. Cells[	== "id")
else {	} while	2].Value.ToString() == "Declit"	{
erse (	((LexGrid. Rows[x]. Cell	LexGrid. Rows[x]. Cells[	t .
ConstantvarList.Add(Le	s[2]. Value. ToString()	2]. Value. ToString() ==	if (constexists =
xGrid.Rows[x].Cells[1]	!= ";"));	";")	ConstantvarList.Exists
.Value.ToString());	}		(element => element ==
1	if	{	LexGrid. Rows[x]. Cells[
intlist. Add (LexGrid. Ro	(LexGrid. Rows[x]. Cells	ì	1]. Value. ToString())
<pre>ws[x].Cells[1].Value.T oString());</pre>	<pre>[2]. Value. ToString() == "digit")</pre>	J	== true)
00 01 1116 (/ / ,	{	else if	{
}		(LexGrid.Rows[x].Cells	

			if
dataGridViewX3.Rows.Ad	if	}	(LexGrid. Rows[x]. Cells
d(idn++, "Multiple	((LexGrid.Rows[x].Cell	•	[2]. Value. ToString()
declaration of	s[2]. Value. ToString()	}	== "unit")
variable: " +	== ";"))	•	. {
LexGrid.Rows[x].Cells[		else if	do
1]. Value. ToString(), "	{	(LexGrid.Rows[x].Cells	{
Line: " + Line);	•	[2]. Value. ToString()	•
	Line++;	== "="	if
}		LexGrid.Rows[x].Cells[	(LexGrid.Rows[x].Cells
	}	2]. Value. ToString() ==	[2]. Value. ToString()
else {		"Charlit"	== "id")
	} while	LexGrid.Rows[x].Cells[	
ConstantvarList.Add(Le	((LexGrid.Rows[x].Cell	2]. Value. ToString() ==	{
xGrid.Rows[x].Cells[1]	s[2]. Value. ToString()	";")	
.Value.ToString());	!= ";"));		if (globexists =
	}	{	GlobalvarList.Exists(e
stringlist.Add(LexGrid	if		lement => element ==
.Rows[x].Cells[1].Valu	(LexGrid.Rows[x].Cells	}	LexGrid.Rows[x].Cells[
e.ToString());	[2]. Value. ToString()		1].Value.ToString())
	== "joe")	else if	== true)
}	{	(LexGrid.Rows[x].Cells	
		[2]. Value. ToString()	{
}	do	== "joe")	
			dataGridViewX3.Rows.Ad
else if	{	{	d(idn++, "Multiple
(LexGrid.Rows[x].Cells			declaration of
[2]. Value. ToString()	if	}	variable: " +
== "="	(LexGrid.Rows[x].Cells		LexGrid.Rows[x].Cells[
LexGrid.Rows[x].Cells[	[2]. Value. ToString()	else {	1].Value.ToString(), "
2].Value.ToString() ==	== "id")		Line: " + Line);
"Stringlit"		dataGridViewX3.Rows.Ad	
LexGrid.Rows[x].Cells[	{	d (i dn++,	}
2].Value.ToString() ==		"TypeMismatch: " +	
";")	if (constexists =	LexGrid.Rows[x].Cells[	else {
	ConstantvarList.Exists	2].Value.ToString(), "	
{	(element => element ==	Line: " + Line);	GlobalvarList.Add(LexG
	LexGrid. Rows[x]. Cells[		rid.Rows[x].Cells[1].V
}	1]. Value. ToString())	}	alue.ToString());
	== true)		
else if	,	X++;	intlist.Add(LexGrid.Ro
(LexGrid. Rows[x]. Cells	{		ws[x]. Cells[1]. Value. T
[2]. Value. ToString()		if	oString());
== "company")	dataGridViewX3. Rows. Ad	((LexGrid. Rows[x]. Cell	1
r	d(idn++, "Multiple	s[2]. Value. ToString()	}
{	declaration of	== ";"))	1
	variable: " +	ſ	1
}	LexGrid. Rows[x]. Cells[	1	1
1. (	1]. Value. ToString(), "	Line	else if
else {	Line: " + Line);	Line++;	(LexGrid. Rows[x]. Cells
dataCnidViewV2 Dewa Ad	ì	1	[2].Value.ToString() == "="
<pre>dataGridViewX3.Rows.Ad d(idn++,</pre>	J	}	== =    LexGrid.Rows[x].Cells[
"TypeMismatch: " +	else {	} while	2]. Value. ToString() ==
LexGrid. Rows[x]. Cells[	C19C (	((LexGrid. Rows[x]. Cell	"Numlit"
2]. Value. ToString(), "	ConstantvarList.Add(Le	s[2]. Value. ToString()	LexGrid. Rows[x]. Cells[
Line: " + Line);	xGrid. Rows[x]. Cells[1]	!= ";"));	2]. Value. ToString() ==
Dine. · Dine/,	. Value. ToString());	. , //,	2]. value, 103t11lig() ";")
}	· · · · · · · · · · · · · · · · · · ·	}	, /
J	charlist.Add(LexGrid.R	J	{
x++;	ows[x]. Cells[1]. Value.		·
Α ,	ToString());	/*Global Semantics*/	}
		, orosar somanoros.,	,

		[2]. Value. ToString()	
else if	dataGridViewX3.Rows.Ad	[2]. varue. lostring() != "(");	{
(LexGrid. Rows[x]. Cells	d(idn++,	}	(
[2]. Value. ToString()	"TypeMismatch: " +	,	}
== "unit"	LexGrid.Rows[x].Cells[	if	
LexGrid.Rows[x].Cells[	2]. Value. ToString(), "	(LexGrid.Rows[x].Cells	else if
2]. Value. ToString() ==	Line: " + Line);	[2]. Value. ToString()	(LexGrid. Rows[x]. Cells
","    	1	== "digit")	[2]. Value. ToString()
LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	}	{ do	== "digit"    LexGrid.Rows[x].Cells[
"("	x++;	{	2]. Value. ToString() ==
LexGrid. Rows[x]. Cells[	A · · · ,	· ·	","
2]. Value. ToString() ==	if	if	LexGrid. Rows[x]. Cells[
")"	((LexGrid.Rows[x].Cell	(LexGrid.Rows[x].Cells	2].Value.ToString() ==
LexGrid.Rows[x].Cells[	s[2]. Value. ToString()	[2]. Value. ToString()	"("
2].Value.ToString() ==	== ";")	== "id")	LexGrid. Rows[x]. Cells[
" {")	(LexGrid. Rows[x]. Cells	ſ	2]. Value. ToString() ==
ı	[2]. Value. ToString() == "{")	1	")"    LexGrid.Rows[x].Cells[
1	( ))	if (globexists =	2]. Value. ToString() ==
//x++;	{	GlobalvarList.Exists(e	"{")
// A ,	·	lement => element ==	
//do	Line++;	LexGrid. Rows[x]. Cells[	{
		1].Value.ToString())	
//{	}	== true)	}
	}		
// if	while	{	else if
(LexGrid.Rows[x].Cells [2].Value.ToString()	((LexGrid.Rows[x].Cell s[2].Value.ToString()	dataGridViewX3.Rows.Ad	(LexGrid. Rows[x]. Cells [2]. Value. ToString()
== "id")	!= ";") &&	d(idn++, "Multiple	== "["
14 /	(LexGrid. Rows[x]. Cells	declaration of	LexGrid. Rows[x]. Cells[
// {	[2]. Value. ToString()	variable: " +	2]. Value. ToString() ==
	!= "{"));	LexGrid.Rows[x].Cells[	"]")
//	}	1].Value.ToString(), "	
funclist. Add (LexGrid. R		Line: " + Line);	{
ows[x]. Cells[1]. Value.	if (LexGrid. Rows [x]. Cel	1	1
ToString());	ls[2].Value.ToString() == "miss")	}	}
// x++;	m133 /	else {	else {
,,		(100	0150 (
// }	X <sup>++</sup> ;	GlobalvarList.Add(LexG	dataGridViewX3.Rows.Ad
	do	rid.Rows[x].Cells[1].V	d(idn++,
//} while	{	alue.ToString());	"TypeMismatch: " +
(LexGrid. Rows[x]. Cells		1 11 11	LexGrid. Rows[x]. Cells[
[2].Value.ToString() != "(");	if (LexGrid.Rows[x].Cells	doublelist.Add(LexGrid .Rows[x].Cells[1].Valu	2].Value.ToString(), " Line: " + Line);
:- (),	[2]. Value. ToString()	e. ToString());	Line. + Line),
}	== "id")	c. 105t1 Ing (// ,	}
,	,	}	,
else if	{		X++;
(LexGrid.Rows[x].Cells		}	
[2]. Value. ToString()	funclist. Add (LexGrid. R		if
== "["	ows[x]. Cells[1]. Value.	else if	((LexGrid. Rows[x]. Cell
LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	ToString());	(LexGrid.Rows[x].Cells [2].Value.ToString()	s[2]. Value. ToString() == ";")
2]. value. lostring() == "]")	x++;	[2]. value. lostring() == "="	(LexGrid. Rows[x]. Cells
٠, د	. ,	LexGrid.Rows[x].Cells[	[2]. Value. ToString()
{	}	2]. Value. ToString() ==	== "{"))
	}	"Declit"	
}	while	LexGrid.Rows[x].Cells[	{
1	(LexGrid.Rows[x].Cells	2]. Value. ToString() ==	T *
else {		";")	Line++;

	== "="	s[2]. Value. ToString()	2].Value.ToString() ==
}	LexGrid.Rows[x].Cells[	!= ";") &&	";")
}	2]. Value. ToString() ==	(LexGrid.Rows[x].Cells	
while	"Stringlit"	[2]. Value. ToString()	{
((LexGrid. Rows[x]. Cell	LexGrid. Rows[x]. Cells[	!= " {"));	1
s[2].Value.ToString() != ";") &&	2].Value.ToString() == ":")	if	}
(LexGrid. Rows[x]. Cells	, )	(LexGrid.Rows[x].Cells	else if
[2]. Value. ToString()	{	[2]. Value. ToString()	(LexGrid. Rows[x]. Cells
!= "{"));		== "joe")	[2]. Value. ToString()
}	}	{	== "joe"
if		do	LexGrid.Rows[x].Cells[
(LexGrid. Rows[x]. Cells	else if	{	2]. Value. ToString() ==
<pre>[2].Value.ToString() == "company")</pre>	(LexGrid.Rows[x].Cells [2].Value.ToString()	if	","    LexGrid.Rows[x].Cells[
Company )	== "company"	(LexGrid.Rows[x].Cells	2]. Value. ToString() ==
do	LexGrid. Rows[x]. Cells[	[2]. Value. ToString()	"("
{	2].Value.ToString() ==	== "id")	LexGrid.Rows[x].Cells[
	","		2]. Value. ToString() ==
if	LexGrid. Rows[x]. Cells[	{	")"
(LexGrid. Rows[x]. Cells	2].Value.ToString() == "("	:£ (-1-1:-+- =	LexGrid.Rows[x].Cells[ 2].Value.ToString() ==
<pre>[2]. Value. ToString() == "id")</pre>	LexGrid.Rows[x].Cells[	if (globexists = GlobalvarList.Exists(e	2]. value. lostring() == "{")
14 /	2]. Value. ToString() ==	lement => element ==	( )
{	")"	LexGrid.Rows[x].Cells[	{
	LexGrid.Rows[x].Cells[	1]. Value. ToString())	
if (globexists =	2]. Value. ToString() ==	== true)	}
GlobalvarList.Exists(e	" {")	ſ	1
<pre>lement =&gt; element == LexGrid.Rows[x].Cells[</pre>	ſ	1	else {
1]. Value. ToString())	t	dataGridViewX3.Rows.Ad	dataGridViewX3.Rows.Ad
== true)	}	d(idn++, "Multiple	d(idn++,
		declaration of	"TypeMismatch: " +
{	else {	variable: " +	LexGrid. Rows[x]. Cells[
l . C . W. VO D	1 . C . W. VO D . A .	LexGrid. Rows[x]. Cells[	2]. Value. ToString(), "
dataGridViewX3.Rows.Ad d(idn++, "Multiple	dataGridViewX3.Rows.Ad d(idn++,	1].Value.ToString(), " Line: " + Line);	Line: " + Line);
declaration of	"TypeMismatch: " +	Line. Line,	}
variable: " +	LexGrid. Rows[x]. Cells[	}	,
LexGrid.Rows[x].Cells[	2]. Value. ToString(), "		X <sup>++</sup> ;
1]. Value. ToString(), "	Line: " + Line);	else {	
Line: " + Line);	1		if
l	}	GlobalvarList.Add(LexG rid.Rows[x].Cells[1].V	((LexGrid.Rows[x].Cells[2].Value.ToString()
J	X++;	alue. ToString());	== ";")
else {	,	2100. 100.01 1118 (//,	(LexGrid.Rows[x].Cells
	if	charlist.Add(LexGrid.R	[2]. Value. ToString()
GlobalvarList.Add(LexG	//I 0 · 1 D [ ] 0 11	[] C-11-[1] V-1	== " {"))
rid. Rows[x]. Cells[1]. V	((LexGrid.Rows[x].Cell	ows[x]. Cells[1]. Value.	( //
alue.ToString());	s[2].Value.ToString()	ToString());	
	s[2].Value.ToString() == ";")		{
stringlist Add(LexGrid	s[2].Value.ToString() == ";")    (LexGrid.Rows[x].Cells		{
stringlist.Add(LexGrid .Rows[x].Cells[1].Valu	s[2].Value.ToString() == ";")		
_	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString()</pre>		{
.Rows[x].Cells[1].Valu	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString()</pre>	<pre>ToString()); } else if</pre>	{     Line++; }
.Rows[x].Cells[1].Valu	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString() == "{"))</pre>	<pre>ToString()); } else if (LexGrid.Rows[x].Cells</pre>	<pre>{ Line++; } while</pre>
.Rows[x].Cells[1].Valu	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString() == "{"))</pre>	<pre>ToString()); } else if (LexGrid.Rows[x].Cells [2].Value.ToString()</pre>	<pre>{ Line++; } while ((LexGrid.Rows[x].Cell</pre>
.Rows[x].Cells[1].Valu	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString() == "{")) { Line++;</pre>	<pre>ToString()); } else if (LexGrid.Rows[x].Cells [2].Value.ToString() == "="   </pre>	<pre>{ Line++; } while ((LexGrid. Rows[x]. Cell s[2]. Value. ToString()</pre>
.Rows[x].Cells[1].Valu	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString() == "{"))</pre>	<pre>ToString()); } else if (LexGrid.Rows[x].Cells [2].Value.ToString()</pre>	<pre>{ Line++; } while ((LexGrid.Rows[x].Cell</pre>
<pre>. Rows[x]. Cells[1]. Valu e. ToString()); }</pre>	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString() == "{")) { Line++; }</pre>	<pre>ToString()); } else if (LexGrid.Rows[x].Cells [2].Value.ToString() == "="    LexGrid.Rows[x].Cells[</pre>	<pre>{ Line++; } while ((LexGrid.Rows[x].Cell s[2].Value.ToString() != ";") &amp;&amp;</pre>
<pre>. Rows[x]. Cells[1]. Valu e. ToString()); } else if</pre>	<pre>s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells [2]. Value. ToString() == "{")) {     Line++; } }</pre>	<pre>ToString()); } else if (LexGrid.Rows[x].Cells [2].Value.ToString() == "="    LexGrid.Rows[x].Cells[ 2].Value.ToString() ==</pre>	<pre>{ Line++; } while ((LexGrid.Rows[x].Cell s[2].Value.ToString() != ";") &amp;&amp; (LexGrid.Rows[x].Cells</pre>

1			
} <b>if</b>	ſ	if (LexGrid.Rows[x].Cells	: c
(LexGrid. Rows[x]. Cells	1	[2]. Value. ToString()	if (LexGrid.Rows[x].Cells
[2]. Value. ToString()	}	== "id")	[2]. Value. ToString()
== "response")	j	1 <b>u</b> )	== ")") { }
{	else if	· ·	, , ( )
do	(LexGrid. Rows[x]. Cells	bool exist;	if
{	[2]. Value. ToString()	Sect office,	(LexGrid.Rows[x].Cells
	== "response"	if(exist =	[2]. Value. ToString()
if	LexGrid. Rows[x]. Cells[	GlobalvarList.Exists(e	== "{") { }
(LexGrid.Rows[x].Cells	2]. Value. ToString() ==	lement => element ==	
[2]. Value. ToString()	″, ″	LexGrid.Rows[x].Cells[	if
== "id")	LexGrid.Rows[x].Cells[	1]. Value. ToString())	(LexGrid.Rows[x].Cells
	2].Value.ToString() ==	== false) {	[2]. Value. ToString()
{	"("		== "unit")
	LexGrid.Rows[x].Cells[	dataGridViewX3.Rows.Ad	
if (globexists =	2]. Value. ToString() ==	d(idn++, "Accessing	{
GlobalvarList.Exists(e	")"	undeclared Variable: "	
lement => element ==	LexGrid. Rows[x]. Cells[	+	do
LexGrid. Rows[x]. Cells[	2].Value.ToString() ==	LexGrid. Rows[x]. Cells[	,
1]. Value. ToString())	" {")	1]. Value. ToString());	{
== true)	ſ	}	: c
t.	1	else if (exist =	if (LexGrid.Rows[x].Cells
l	1	funclist.Exists(elemen	[2]. Value. ToString()
dataGridViewX3.Rows.Ad	J	t => element ==	== "id")
d(idn++, "Multiple	else {	LexGrid. Rows[x]. Cells[	1u )
declaration of	CISC (	1]. Value. ToString())	{
variable: " +	dataGridViewX3.Rows.Ad	== false)	
LexGrid. Rows[x]. Cells[	d(idn++,	{	if (locexists =
1]. Value. ToString(), "	"TypeMismatch: " +	•	LocalvarList.Exists(el
Line: " + Line);	LexGrid.Rows[x].Cells[	dataGridViewX3.Rows.Ad	ement => element ==
	2].Value.ToString(), "	d(idn++, "Accessing	LexGrid.Rows[x].Cells[
}	Line: " + Line);	undeclared Variable: "	1].Value.ToString())
		+	== true)
else {	}	LexGrid.Rows[x].Cells[	
		1]. Value. ToString());	{
GlobalvarList. Add (LexG	X++;	}	1 . 6 . 111 . 12 . 1
rid. Rows[x]. Cells[1]. V	• 6	}	dataGridViewX3. Rows. Ad
alue.ToString());	if	else	d(idn++, "Multiple declaration of
haalist Add/LarCrid Da	((LexGrid.Rows[x].Cell s[2].Value.ToString()	1	variable: " +
<pre>boolist.Add(LexGrid.Ro ws[x].Cells[1].Value.T</pre>	s[z]. varue. lostring() == ";")	ì	LexGrid. Rows[x]. Cells[
oString());	(LexGrid.Rows[x].Cells	J	1]. Value. ToString(), "
obtling()),	[2]. Value. ToString()	/*Local Declaration*/	Line: " + Line);
}	== "{"))	if	Bine, Bine,
		(LexGrid.Rows[x].Cells	}
}	{	[2]. Value. ToString()	
		== "PrimaryMission")	else {
else if	Line++;	{	
(LexGrid.Rows[x].Cells			LocalvarList.Add(LexGr
[2]. Value. ToString()	}	do	id. Rows[x]. Cells[1]. Va
== "="	}		lue.ToString());
LexGrid. Rows[x]. Cells[	while	{	
2]. Value. ToString() ==	((LexGrid. Rows[x]. Cell		intlist. Add (LexGrid. Ro
"AFFIRMATIVE"	s[2]. Value. ToString()	X++;	ws[x]. Cells[1]. Value. T
LexGrid. Rows[x]. Cells[	!= ";") &&	; f	oString());
2]. Value. ToString() == "NECATIVE"	(LexGrid. Rows[x]. Cells	if	1
"NEGATIVE"    LexGrid.Rows[x].Cells[	[2].Value.ToString() != "{"));	(LexGrid.Rows[x].Cells [2].Value.ToString()	l
2]. Value. ToString() ==	. (//,	== "(") { }	}
z]. varue. 103t1111g() ";")	J	( / ( )	J
, /			

else if (LexGrid.Rows[x].Cells	if ((LexGrid.Rows[x].Cell	else {	X++;
[2]. Value. ToString() == "="    LexGrid. Rows[x]. Cells[	s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells	LocalvarList.Add(LexGr id.Rows[x].Cells[1].Va lue.ToString());	<pre>if ((LexGrid.Rows[x].Cell s[2].Value.ToString()</pre>
2]. Value. ToString() == "Numlit"	[2]. Value. ToString() == "{"))	doublelist.Add(LexGrid	== ";")    (LexGrid. Rows[x]. Cells
<pre>LexGrid.Rows[x].Cells[ 2].Value.ToString() ==</pre>	{	.Rows[x].Cells[1].Valu e.ToString());	[2]. Value. ToString() == "{"))
";")	Line++;	}	{
{	}	}	Line++;
}	} while	else if	}
<pre>else if (LexGrid.Rows[x].Cells [2].Value.ToString()</pre>	((LexGrid.Rows[x].Cell s[2].Value.ToString() != ";") &&	(LexGrid.Rows[x].Cells [2].Value.ToString() == "="	<pre>} while ((LexGrid.Rows[x].Cell</pre>
== "unit"    LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	(LexGrid.Rows[x].Cells [2].Value.ToString() != "{");	<pre>LexGrid.Rows[x].Cells[ 2].Value.ToString() == "Declit"   </pre>	s[2]. Value. ToString() != ";") && (LexGrid. Rows[x]. Cells
","    LexGrid.Rows[x].Cells[	}	<pre>LexGrid.Rows[x].Cells[ 2].Value.ToString() ==</pre>	[2]. Value. ToString() != "{");
2]. Value. ToString() == "("	if	";")	}
LexGrid.Rows[x].Cells[ 2].Value.ToString() == ")"	<pre>(LexGrid.Rows[x].Cells [2].Value.ToString() == "digit")</pre>	1	if (LexGrid.Rows[x].Cells
<pre>LexGrid.Rows[x].Cells[ 2].Value.ToString() ==</pre>	- digit /	else if	[2]. Value. ToString() == "company")
"{") {	do	(LexGrid.Rows[x].Cells [2].Value.ToString() == "digit"	{
}	{	<pre>LexGrid.Rows[x].Cells[ 2].Value.ToString() ==</pre>	do
else if	if (LexGrid.Rows[x].Cells	","    LexGrid.Rows[x].Cells[	{
(LexGrid.Rows[x].Cells [2].Value.ToString() == "["	<pre>[2].Value.ToString() == "id")</pre>	2].Value.ToString() == "("    LexGrid.Rows[x].Cells[	<pre>if (LexGrid.Rows[x].Cells [2].Value.ToString()</pre>
LexGrid. Rows[x]. Cells[ 2]. Value. ToString() ==	{	2]. Value. ToString() == "")"	== "id")
"]")	if (locexists = LocalvarList.Exists(el	LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	{
{	<pre>ement =&gt; element == LexGrid.Rows[x].Cells[</pre>	" {")	if (locexists = LocalvarList.Exists(el
}	1].Value.ToString()) == true)	1	<pre>ement =&gt; element == LexGrid.Rows[x].Cells[</pre>
else { dataGridViewX3.Rows.Ad	{	else {	1].Value.ToString()) == true)
d(idn++, "TypeMismatch: " +	dataGridViewX3.Rows.Ad d(idn++, "Multiple	dataGridViewX3.Rows.Ad	{
LexGrid.Rows[x].Cells[ 2].Value.ToString(), "	declaration of variable: " +	d(idn++, "TypeMismatch: " +	dataGridViewX3.Rows.Add(idn++, "Multiple
Line: " + Line);	LexGrid.Rows[x].Cells[ 1].Value.ToString(), "	LexGrid.Rows[x].Cells[ 2].Value.ToString(), "	declaration of variable: " +
}	Line: " + Line);	Line: " + Line);	LexGrid.Rows[x].Cells[ 1].Value.ToString(), "
X <sup>++</sup> ;	}	}	Line: " + Line);

}	}	1].Value.ToString(), " Line: " + Line);	2].Value.ToString(), " Line: " + Line);
else {	x++;	}	}
LocalvarList.Add(LexGr id.Rows[x].Cells[1].Va lue.ToString());	if ((LexGrid.Rows[x].Cell s[2].Value.ToString()	else { LocalvarList.Add(LexGr	x++; if
<pre>stringlist.Add(LexGrid .Rows[x].Cells[1].Valu e.ToString());</pre>	== ";")    (LexGrid.Rows[x].Cells [2].Value.ToString() == "{"))	<pre>id. Rows[x].Cells[1].Va lue.ToString()); charlist.Add(LexGrid.R</pre>	<pre>((LexGrid. Rows[x]. Cell s[2]. Value. ToString() == ";")    (LexGrid. Rows[x]. Cells</pre>
}	{	<pre>ows[x].Cells[1].Value. ToString());</pre>	[2]. Value. ToString() == "{"))
}	Line++;	}	{
else if	}	}	Line++;
<pre>(LexGrid. Rows[x]. Cells [2]. Value. ToString() == "="    LexGrid. Rows[x]. Cells[ 2]. Value. ToString() == "Stringlit"    LexGrid. Rows[x]. Cells[ 2]. Value. ToString() == ";")</pre>	<pre>} while ((LexGrid.Rows[x].Cell s[2].Value.ToString() != ";") &amp;&amp; (LexGrid.Rows[x].Cells [2].Value.ToString() != "{");;</pre>	<pre>else if (LexGrid.Rows[x].Cells [2].Value.ToString() == "="    LexGrid.Rows[x].Cells[ 2].Value.ToString() == "Charlit"    LexGrid.Rows[x].Cells[ 2].Value.ToString() == ";")</pre>	<pre>} while ((LexGrid.Rows[x].Cell s[2].Value.ToString() != ";") &amp;&amp; (LexGrid.Rows[x].Cells [2].Value.ToString() != "{");</pre>
}	if (LexGrid.Rows[x].Cells	, /	}
else if (LexGrid.Rows[x].Cells [2].Value.ToString()	[2]. Value. ToString() == "joe") {	else if	<pre>if (LexGrid.Rows[x].Cells [2].Value.ToString() == "response")</pre>
== "company"    LexGrid. Rows[x]. Cells[ 2]. Value. ToString() == ","	do {	(LexGrid.Rows[x].Cells [2].Value.ToString() == "joe"    LexGrid.Rows[x].Cells[	{ do
LexGrid. Rows[x]. Cells[ 2]. Value. ToString() == "("    LexGrid. Rows[x]. Cells[	if (LexGrid.Rows[x].Cells [2].Value.ToString()	2]. Value. ToString() == ","    LexGrid. Rows[x]. Cells[ 2]. Value. ToString() ==	{ if
2].Value.ToString() == ")"    LexGrid.Rows[x].Cells[	== "id") {	"("    LexGrid.Rows[x].Cells[ 2].Value.ToString() == ")"	<pre>(LexGrid.Rows[x].Cells [2].Value.ToString() == "id")</pre>
2].Value.ToString() == "{")	<pre>if (locexists = LocalvarList.Exists(el ement =&gt; element ==</pre>	LexGrid.Rows[x].Cells[ 2].Value.ToString() == "{")	<pre>if (locexists =</pre>
}	LexGrid.Rows[x].Cells[ 1].Value.ToString()) == true)	{	LocalvarList. Exists(el ement => element == LexGrid. Rows[x]. Cells[
else {	{	}	1].Value.ToString()) == true)
dataGridViewX3. Rows. Add(idn++, "TypeMismatch: " +	dataGridViewX3. Rows. Ad	else { dataGridViewX3. Rows. Ad	{
LexGrid.Rows[x].Cells[ 2].Value.ToString(), " Line: " + Line);	<pre>declaration of variable: " + LexGrid.Rows[x].Cells[</pre>	<pre>d(idn++, "TypeMismatch: " + LexGrid.Rows[x].Cells[</pre>	<pre>dataGridViewX3. Rows. Ad d(idn++, "Multiple declaration of variable: " +</pre>

LexGrid.Rows[x].Cells[	d(idn++,	d(idn++, "Multiple	2]. Value. ToString(), "
<pre>1].Value.ToString(), " Line: " + Line);</pre>	"TypeMismatch: " + LexGrid.Rows[x].Cells[	<pre>declaration of variable: " +</pre>	Line: " + Line);
	2]. Value. ToString(), "	LexGrid.Rows[x].Cells[	}
}	Line: " + Line);	<pre>1].Value.ToString(), " Line: " + Line);</pre>	x++;
else {	}	}	if
LocalvarList.Add(LexGr	x++;	,	((LexGrid.Rows[x].Cell
<pre>id. Rows[x]. Cells[1]. Va lue. ToString());</pre>	if	else {	s[2]. Value. ToString() == "{"))
ruc. rootiring (/),	((LexGrid.Rows[x].Cell	LocalvarList.Add(LexGr	( ) /
<pre>boolist.Add(LexGrid.Ro ws[x].Cells[1].Value.T</pre>	s[2].Value.ToString() == ";")	id.Rows[x].Cells[1].Va lue.ToString());	{
oString());	(LexGrid. Rows[x]. Cells		Line++;
}	[2]. Value. ToString() == "{")	}	}
)		}	j
}	{		} while
also if	Lingth	else if (LexGrid.Rows[x].Cells	((LexGrid.Rows[x].Cell s[2].Value.ToString()
else if (LexGrid.Rows[x].Cells	Line++;	[2]. Value. ToString()	!= "{"));
[2]. Value. ToString()	}	== "Stringlit"	. (///
== "="		LexGrid.Rows[x].Cells[	}
LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	<pre>} while ((LexGrid.Rows[x].Cell</pre>	2].Value.ToString() == "Declit"	-1 (
"AFFIRMATIVE"	s[2]. Value. ToString()	LexGrid. Rows[x]. Cells[	else {
LexGrid. Rows[x]. Cells[	!= ";") &&	2]. Value. ToString() ==	dataGridViewX3.Rows.Ad
2]. Value. ToString() ==	(LexGrid. Rows[x]. Cells	"Numlit"	d(idn++, "Semantics
"NEGATIVE"    LexGrid.Rows[x].Cells[	[2].Value.ToString() != "{");	<pre>LexGrid.Rows[x].Cells[ 2].Value.ToString() ==</pre>	Analyzer Succeeded");
2]. Value. ToString() ==	:- ( )),	"Charlit"	Succeeded),
";")	}	LexGrid.Rows[x].Cells[	}
ſ	if	2].Value.ToString() == "="	
1	(LexGrid. Rows[x]. Cells	-    LexGrid.Rows[x].Cells[	x++;
}	[2]. Value. ToString()	2].Value.ToString() ==	,
	== "miss")	"joe"	
else if (LexGrid.Rows[x].Cells	ſ	LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	<pre>} while (LexGrid.Rows[x].Cells</pre>
[2]. Value. ToString()	l	","	[2]. Value. ToString()
== "response"	do	LexGrid.Rows[x].Cells[	== "deploy");
LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	ſ	2].Value.ToString() == "("	//
", "	(	LexGrid.Rows[x].Cells[	MessageBox. Show(x. ToSt
LexGrid.Rows[x].Cells[	if	2].Value.ToString() ==	ring());
2].Value.ToString() ==	(LexGrid. Rows[x]. Cells	")"    	}
"("    LexGrid.Rows[x].Cells[	[2]. Value. ToString() == "id")	LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	else
2]. Value. ToString() ==	14 /	"{")	{
")"	{		
LexGrid.Rows[x].Cells[ 2].Value.ToString() ==	if (locexists =	{	continue;
"{")	LocalvarList. Exists (el	}	}
•	ement => element ==		/*Check if
{	LexGrid. Rows[x]. Cells[	else {	Constant var Exist to
}	1].Value.ToString()) == true)	dataGridViewX3.Rows.Ad	Global var*/ foreach
J	or ac/	d(idn++,	(string constlist in
else {	{	"TypeMismatch: " +	ConstantvarList)
dataCridViowY3 Down Ad	dataCridViowY? Down Ad	LexGrid.Rows[x].Cells[	{
dataGridViewX3.Rows.Ad	dataGridViewX3.Rows.Ad		

if	1-4-C-: JV:V2 D AJ	{	: 6
(GlobalvarList.Contain s(constlist) == true)	dataGridViewX3.Rows.Ad d(idn++, "Reserved	x += 2;	if (LexGrid.Rows[x].Cells
s(constitist) tide)	Identifier Misused: "	x '- 2,	[2]. Value. ToString()
ί	+ globallist, "Line:	main +=	== "digit")
dataGridViewX3.Rows.Ad	" + Line);	"Console. ReadLine();	digit /
d(idn++, "Multiple	}	\n } \n";	{
declaration of Global	else	}	
variable: " +	{		<pre>main += "double ";</pre>
constlist, "Line: " +		else	
Line);	continue;	{	X <sup>++</sup> ;
}	}	_	
	}	main += LexGrid.Rows[x	}
if (LocalvarList. Contai	}	-	1
ns(constlist) == true)		1]. Cells[1]. Value. ToSt	else if
1	private void buttonX12 Click(object	ring() + " \n";	(LexGrid.Rows[x].Cells [2].Value.ToString()
dataGridViewX3.Rows.Ad	sender, EventArgs e)	// x;	== "unit")
d(idn++, "Multiple	{	// A ,	unit )
declaration of Local		}	{
variable: " +	richTextBoxEx1.Text =	switch	•
constlist, "Line: " +	"";	(LexGrid.Rows[x].Cells	main += "int ";
Line);	int	[2]. Value. ToString())	
}	checktemp = 0;	{	X++;
if	int		
(ReservedW. Contains (co	checkfunc = 0;	case "PrimaryMission":	if
nstlist) == true)		0	((LexGrid. Rows[x+2]. Ce
{	List <string> disp =</string>	X += 3;	11s[1]. Value. ToString(
1-4-C+: 1V:V2 D A1	new List <string>();</string>		) == "[") && (LexGrid.Rows[x +
<pre>dataGridViewX3.Rows.Ad d(idn++, "Reserved</pre>	List <string> outp =</string>	main += "public static void Main() \n { \n";	5]. Cells[1]. Value. ToSt
Identifier Misused: "	new List <string> ();</string>	void main() \n (\n,	ring() = "["])
+ constlist, "Line: "	new List(String/(),	checktemp = 1;	11lig (/ [ //
+ Line);	richTextBoxEx1.Text =	checktomp 1,	{
}	"using System; \n	break;	•
else	namespace test \n { \n		main +=
{	class test \n { \n ";	case "unit":	LexGrid.Rows[x].Cells[
			1].Value.ToString() +
continue;	//globdeclare =	//Check if Global unit	", " + "]" + " " +
}	"public static class	10 (1 1 2 0 1	LexGrid.Rows[x -
}	GlobalVar { \n";	if (checktemp == 0	1]. Cells[1]. Value. ToSt
foreach	//main =	checkfunc == 0)	ring();
(string globallist in GlobalvarList)	"class funct \n { \n"; //function	1	x++;
(10balvalL1st)	= "class func \n {	t	Α'',
(	\n";	if (LexGrid.Rows[x +	}
if	for (int x	2]. Cells[2]. Value. ToSt	,
(LocalvarList.Contains	= 0; x <	ring() == "(")	}
(globallist) == true)	LexGrid.Rows.Count;		
{	X++)	{	else if
	{		(LexGrid.Rows[x].Cells
dataGridViewX3. Rows. Ad	if	checkfunc = 1;	[2]. Value. ToString()
d(idn++, "Multiple	(LexGrid. Rows[x]. Cells		== "joe")
declaration of Local	[2]. Value. ToString()	X++;	ſ
variable: " + globallist, " Line: "	== "}")	main += "public static	l
+ Line);	ι	int ";	main += "char";
bine,,	x++;	· · · · · · · · · · · · · · · · · · ·	metri · Clief ,
if	if	do	x++;
(ReservedW. Contains (gl	(LexGrid.Rows[x].Cells		•
oballist) == true)	[2]. Value. ToString()	{	}
{	== "deploy")		

	(LexGrid.Rows[x +		
else if	3]. Cells[1]. Value. ToSt	}	}
(LexGrid.Rows[x].Cells	ring() == "["))		
[2]. Value. ToString()		else if	else if
== "company")	{	(LexGrid.Rows[x].Cells	(LexGrid.Rows[x].Cells
		[1]. Value. ToString()	[1]. Value. ToString()
{	main += "public static	== "[")	== "(")
	int ";		
main += "string";		{	{
	main +=		
X++;	LexGrid.Rows[x].Cells[	main += "public static	main += "public static
	1]. Value. ToString() +	int ";	int ";
}	", " + "]" + " " +		
	LexGrid. Rows[x -	main +=	main +=
else	1]. Cells[1]. Value. ToSt	LexGrid.Rows[x].Cells[	LexGrid.Rows[x].Cells[
	ring();	1].Value.ToString() +	1]. Value. ToString();
{		"]" + " " +	
	x += 6;	LexGrid.Rows[x -	do
main +=		1]. Cells[1]. Value. ToSt	
LexGrid.Rows[x].Cells[	if	ring();	{
1]. Value. ToString();	(LexGrid.Rows[x].Cells	_	
	[1]. Value. ToString()	x += 3;	main +=
x++;	== "=")	•	LexGrid.Rows[x].Cells[
•		if	1]. Value. ToString();
}	{	(LexGrid.Rows[x].Cells	-3
,		[1]. Value. ToString()	X++;
} while	do	== "=")	,
(LexGrid. Rows[x]. Cells	40	,	} while
[1]. Value. ToString()	{	{	(LexGrid. Rows[x]. Cells
!= "{");	C		[2]. Value. ToString()
. (),	main +=	do	!= "{");
main += "{\n";	LexGrid. Rows[x]. Cells[	uo	. ( ) ,
main ( \mathridage ( \mathridage )	1]. Value. ToString();	{	main +=
break;	ij. varue. 100 tring (),	(	LexGrid. Rows[x]. Cells[
bicak,	x++;	main +=	1]. Value. ToString();
}	A · · · ,	LexGrid. Rows[x]. Cells[	ij. value. losti ing (),
J	} while	1]. Value. ToString();	}
if(checkfunc == 0 &&	(LexGrid. Rows[x]. Cells	ij. varde. 100 ti ing () ,	J
checktemp == 0)	[1]. Value. ToString()	x++;	else if
checktemp 0)	!= ";");	A · · ,	(LexGrid. Rows[x]. Cells
{	. , /,	} while	[1]. Value. ToString()
(	main += "; \n";	(LexGrid. Rows[x]. Cells	== "=")
x++;	main - , \n ,	[1]. Value. ToString()	,
Λ'',	// x += 2;	!= "}");	1
do	// A · 2,	· , , ,	(
uo	}	main += "}; \n";	main += "public static
{	J	main , , , ii ,	int " + LexGrid. Rows[x
(	else	x++;	- Lexol 1d, Rows [x
switch	6156	Δ'',	1]. Cells[1]. Value. ToSt
(LexGrid. Rows[x]. Cells	{	}	ring() + "=" +
[2]. Value. ToString())	(	J	LexGrid. Rows[x +
[2]. value: 103tling(//	main += "= new int[" +	else	1]. Cells[1]. Value. ToSt
{	LexGrid. Rows[x -	6136	ring() + "; \n";
·	5]. Cells[1]. Value. ToSt	{	11116 (/ ' , \11 ,
case "id":	ring() + "," +	· ·	x = x + 2;
case iu .	LexGrid. Rows [x -	main += "= new int[" +	$\Lambda = \Lambda + \Delta$ ,
x++;	2]. Cells[1]. Value. ToSt	LexGrid. Rows[x -	}
Α···,	ring();	2]. Cells[1]. Value. ToSt	J
if	11118 (/ ,	ring() + "]; \n";	else
((LexGrid.Rows[x].Cell	main += "]; \n";	ring(/ ' ], \II ,	0190
s[1]. Value. ToString()	main - J, M,	1	ſ
alitaratus, footi iiik ()		}	
== "[") &&	}	}	{

main += "public static	(LexGrid.Rows[x + 3].Cells[1].Value.ToSt	}	else if
int " + LexGrid.Rows[x	ring() == "["))	else if	(LexGrid.Rows[x].Cells [1].Value.ToString()
1]. Cells[1]. Value. ToSt	{	(LexGrid. Rows[x]. Cells	== "(")
ring() + "; \n";	•	[1]. Value. ToString()	. ,
	main += "int ";	== "[")	{
}		r	
break;	main += LexGrid.Rows[x].Cells[	1	main += "int ";
Dieak,	1]. Value. ToString() +	main += "int ";	main +=
case ", ":	"," + "]" + " " +	,	LexGrid.Rows[x].Cells[
	LexGrid.Rows[x -	main +=	1].Value.ToString();
X++;	1]. Cells[1]. Value. ToSt	LexGrid. Rows [x]. Cells [	1
break;	ring();	1].Value.ToString() + "]" + "" +	do
bicak,	x += 6;	LexGrid.Rows[x -	{
}		1].Cells[1].Value.ToSt	
1	if	ring();	main +=
<pre>} while (LexGrid.Rows[x].Cells</pre>	(LexGrid.Rows[x].Cells [1].Value.ToString()	x += 3;	LexGrid.Rows[x].Cells[ 1].Value.ToString();
[2]. Value. ToString()	== "=")	x '- 3,	ij. vaiue. iostiing(),
!= ";");	,	if	x++;
	{	(LexGrid.Rows[x].Cells	
}	1	[1]. Value. ToString() == "=")	} while
else	do	== = )	(LexGrid.Rows[x].Cells [2].Value.ToString()
CISC	{	{	!= "{");
{			
	main +=	do	main +=
main += "int ";	<pre>LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	ſ	<pre>LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>
}	ij. vaiue. 103tiing(),	l .	IJ. varue. 103tIIIIg(),
•	X <sup>++</sup> ;	main +=	}
break;		LexGrid. Rows[x]. Cells[	
}	<pre>} while (LexGrid.Rows[x].Cells</pre>	1].Value.ToString();	else if (LexGrid.Rows[x].Cells
,	[1]. Value. ToString()	x++;	[1]. Value. ToString()
else {	!= ";");	,	== "=")
		} while	
//Check if Local unit	main $+=$ "; $\setminus$ n";	(LexGrid. Rows[x]. Cells	{
x++;	// x += 2;	[1].Value.ToString() != "}");	main += "int " +
A · · · ,	// A · 2,	. , , ,	LexGrid. Rows[x -
do	}	main += "}; \n";	1].Cells[1].Value.ToSt
ſ	1		ring() + "=" +
{	else	X <sup>++</sup> ;	LexGrid.Rows[x + 1].Cells[1].Value.ToSt
switch	{	}	ring() + "; \n";
(LexGrid.Rows[x].Cells		•	
[2]. Value. ToString())	main += "= new int[" +	else	x = x + 2;
ſ	LexGrid. Rows[x -	{	1
l	5].Cells[1].Value.ToSt ring() + "," +	l	}
case "id":	LexGrid. Rows[x -	main += "= new int[" +	else
	2]. Cells[1]. Value. ToSt	LexGrid.Rows[x -	
X++;	ring();	2]. Cells[1]. Value. ToSt	{
if	main += "]; \n";	ring() + "]; \n";	main += "int " +
((LexGrid.Rows[x].Cell	main - J, \II,	}	LexGrid. Rows[x -
s[1].Value.ToString()	}		1].Cells[1].Value.ToSt
== "[") &&		}	ring() + "; \n";

```
else if
                                                                    if (checkfunc == 0 &&
                                                                                                     } while
                                  (LexGrid. Rows[x]. Cells
                                                                    checktemp == 0)
                                                                                                      (LexGrid. Rows[x]. Cells
break;
                                  [2]. Value. ToString()
                                                                                                      [1]. Value. ToString()
                                 == "unit")
                                                                                                      != ":"):
case ", ":
                                  {
                                                                                                     main += "; \n";
                                                                    x++:
X^{++};
                                  main += "int ";
                                                                                                     //_{X} += 2;
                                                                    do
break;
                                 x++;
                                                                    switch
                                                                                                     else
                                                                    (LexGrid. Rows[x]. Cells
} while
(LexGrid. Rows[x]. Cells
                                                                    [2]. Value. ToString())
                                  else if
[2]. Value. ToString()
                                  (LexGrid.Rows[x].Cells
!= ";");
                                  [2]. Value. ToString()
                                                                                                     main += "= new
                                  == "joe")
                                                                                                     double[" +
                                                                    case "id":
                                                                                                     LexGrid.Rows[x -
                                  {
                                                                                                     5]. Cells[1]. Value. ToSt
                                                                                                     ring() + "," +
break;
                                                                    x++;
                                                                                                     LexGrid.Rows[x -
                                  main += "char";
                                                                                                     2]. Cells[1]. Value. ToSt
                                                                    ((LexGrid.Rows[x].Cell
                                  x++;
                                                                                                     ring();
case "digit":
                                                                    s[1]. Value. ToString()
                                                                    == "[") &&
                                                                                                     main += "]; \n";
                                                                    (LexGrid. Rows[x +
//Check if Global unit
                                                                    3]. Cells[1]. Value. ToSt
                                  else if
if (checktemp == 0 ||
                                  (LexGrid. Rows[x]. Cells
                                                                    ring() = "["))
checkfunc == 0)
                                  [2]. Value. ToString()
                                  == "company")
                                                                                                     else if
                                  {
                                                                   main += "public static
                                                                                                      (LexGrid. Rows[x]. Cells
if (LexGrid.Rows[x +
                                                                   double ";
                                                                                                      [1]. Value. ToString()
2]. Cells[2]. Value. ToSt
                                 main += "string";
                                                                                                     == "[")
ring() = "(")
                                                                   main +=
                                                                                                      {
                                                                   LexGrid.Rows[x].Cells[
                                  x++;
                                                                    1].Value.ToString() +
                                                                    "," + "]" + " " +
                                                                                                     main += "public static
checkfunc = 1;
                                                                   LexGrid.Rows[x -
                                                                                                     double ";
                                                                    1]. Cells[1]. Value. ToSt
                                  else
                                                                                                     main +=
x++:
                                                                   ring();
                                                                                                     LexGrid.Rows[x].Cells[
                                                                                                     1]. Value. ToString() + "]" + "" +
main += "public static
                                                                   x += 6;
double ";
                                  main +=
                                 LexGrid.Rows[x].Cells[
                                                                                                     LexGrid.Rows[x -
                                  1]. Value. ToString();
                                                                    (LexGrid. Rows[x]. Cells
                                                                                                     1]. Cells[1]. Value. ToSt
                                                                    [1]. Value. ToString()
                                                                                                     ring();
                                                                   == "=")
                                 X^{++};
                                                                                                     X += 3;
(LexGrid.Rows[x].Cells
[2]. Value. ToString()
                                                                                                      (LexGrid. Rows[x]. Cells
                                 } while
                                                                    do
== "digit")
                                  (LexGrid.Rows[x].Cells
                                                                                                     [1]. Value. ToString()
                                  [1]. Value. ToString()
                                                                                                     == "=")
                                  != "{");
                                                                   main +=
                                                                                                      {
main += "double ";
                                 main += "{\n";
                                                                   LexGrid.Rows[x].Cells[
                                                                   1]. Value. ToString();
                                                                                                     do
X^{++};
                                 break:
                                                                                                      {
                                                                   X^{++};
```

main +=	LexGrid.Rows[x].Cells[ 1].Value.ToString();	break;	LexGrid.Rows[x].Cells[ 1].Value.ToString();
LexGrid.Rows[x].Cells[ 1].Value.ToString();	}	}	x++;
x++;	else if (LexGrid.Rows[x].Cells	else	<pre>} while (LexGrid.Rows[x].Cells</pre>
<pre>} while (LexGrid.Rows[x].Cells</pre>	[1]. Value. ToString() == "=")	{	[1]. Value. ToString() != ";");
[1]. Value. ToString() != "}");	{	//Check if Local unit	main += "; \n";
main += "}; \n";	main += "public static	x++;	// x += 2;
x++;	double " + LexGrid.Rows[x -	do	}
}	1].Cells[1].Value.ToSt ring() + "=" +	{	else
else	LexGrid.Rows[x + 1].Cells[1].Value.ToSt	<pre>switch (LexGrid.Rows[x].Cells</pre>	{
{	ring() + "; \n";	[2]. Value. ToString())	main += "= new
main += "= new	x = x + 2;	{	double[" + LexGrid.Rows[x -
double[" + LexGrid.Rows[x -	}	case "id":	5]. Cells[1]. Value. ToSt ring() + "," +
2].Cells[1].Value.ToSt ring() + "]; \n";	else	x++;	LexGrid. Rows[x - 2]. Cells[1]. Value. ToSt
}	{	if ((LexGrid.Rows[x].Cell	ring();
}	<pre>main += "public static double " + LexGrid.Rows[x -</pre>	s[1].Value.ToString() == "[") && (LexGrid.Rows[x +	main += "]; \n";
else if (LexGrid.Rows[x].Cells	1]. Cells[1]. Value. ToSt ring() + "; \n";	3]. Cells[1]. Value. ToSt ring() == "["))	}
[1]. Value. ToString() == "(")	}	{	else if
{	break;	<pre>main += "double ";</pre>	<pre>(LexGrid.Rows[x].Cells [1].Value.ToString() == "[")</pre>
<pre>main += "public static double ";</pre>	case ",":	main += LexGrid.Rows[x].Cells[	{
main +=	X++;	1]. Value. ToString() + "," + "]" + " " +	main += "int ";
LexGrid.Rows[x].Cells[ 1].Value.ToString();	break;	LexGrid.Rows[x - 1].Cells[1].Value.ToSt	main +=
do	} while	ring(); x += 6;	LexGrid.Rows[x].Cells[ 1].Value.ToString() + "]" + "" +
{	(LexGrid.Rows[x].Cells [2].Value.ToString()	x 0, if	LexGrid.Rows[x - 1].Cells[1].Value.ToSt
main += LexGrid.Rows[x].Cells[	!= ";");	(LexGrid.Rows[x].Cells [1].Value.ToString()	ring();
1]. Value. ToString();	}	== "=")	x += 3;
X++;	else	l do	<pre>if (LexGrid.Rows[x].Cells [1].Value.ToString()</pre>
<pre>} while (LexGrid. Rows[x]. Cells [2]. Value. ToString()</pre>	main += "double ";	do	[1]. value. lostring() == "=")
[2]. value. lostring() != "{");	main double ,	main +=	{
main +=	,		do

```
{
                                 main +=
                                                                   if (checktemp == 0 ||
                                                                                                     else if
                                 LexGrid.Rows[x].Cells[
                                                                   checkfunc == 0)
                                                                                                     (LexGrid. Rows[x]. Cells
main +=
                                 1]. Value. ToString();
                                                                                                     [2]. Value. ToString()
LexGrid. Rows[x]. Cells[
                                                                                                     == "company")
1]. Value. ToString();
                                                                   if (LexGrid.Rows[x +
                                 else if
                                                                   2]. Cells[2]. Value. ToSt
X^{++};
                                                                   ring() = "(")
                                                                                                     main += "string";
                                 (LexGrid.Rows[x].Cells
} while
                                 [1]. Value. ToString()
(LexGrid.Rows[x].Cells
                                 == "=")
                                                                                                     x++;
[1]. Value. ToString()
!= "}");
                                 {
                                                                   checkfunc = 1;
main += "}; \n";
                                 main += "double" +
                                                                   x++:
                                                                                                     else
                                 LexGrid. Rows[x -
X^{++};
                                 1]. Cells[1]. Value. ToSt
                                                                   main += "public static
                                 ring() + "=" +
                                                                   string ";
                                 LexGrid.Rows[x +
                                                                                                     main +=
                                 1]. Cells[1]. Value. ToSt
                                                                   do
                                                                                                     LexGrid.Rows[x].Cells[
else
                                 ring() + "; \n";
                                                                                                     1]. Value. ToString();
{
                                 X = X + 2;
                                                                                                     X^{++};
main += "= new
                                                                   (LexGrid. Rows[x]. Cells
double[" +
                                                                   [2]. Value. ToString()
LexGrid.Rows[x -
                                                                   == "digit")
                                                                                                     } while
                                 else
2]. Cells[1]. Value. ToSt
                                                                                                     (LexGrid. Rows[x]. Cells
ring() + "]; \n";
                                 {
                                                                                                     [1]. Value. ToString()
                                                                                                     != "{");
                                 main += "double " +
                                                                   main += "double ":
                                 LexGrid.Rows[x -
                                                                                                     main += "{\n";
                                 1]. Cells[1]. Value. ToSt
                                                                   X^{++};
                                 ring() + "; \n";
                                                                                                     break;
else if
(LexGrid.Rows[x].Cells
[1]. Value. ToString()
                                                                   else if
== "(")
                                 break;
                                                                   (LexGrid.Rows[x].Cells
                                                                                                     if (checkfunc == 0 \&\&
                                                                   [2]. Value. ToString()
                                                                                                     checktemp == 0
                                                                   == "unit")
{
                                 case ", ":
                                                                                                     {
main += "double":
                                 x++:
                                                                                                     X^{++};
                                                                   main += "int ";
main +=
                                 break;
LexGrid.Rows[x].Cells[
                                                                                                     do
1]. Value. ToString();
                                                                   X^{++};
do
                                 } while
                                 (LexGrid. Rows[x]. Cells
                                                                                                     switch
{
                                 [2]. Value. ToString()
                                                                   else if
                                                                                                     (LexGrid. Rows[x]. Cells
                                 != ";");
                                                                   (LexGrid.Rows[x].Cells
                                                                                                     [2]. Value. ToString())
main +=
                                                                   [2]. Value. ToString()
LexGrid.Rows[x].Cells[
                                                                   == "joe")
                                                                                                     {
1]. Value. ToString();
                                                                                                     case "id":
                                 break;
X++;
                                 //Check if Global
                                                                   main += "char";
                                                                                                     X^{++};
} while
                                 Company
(LexGrid. Rows[x]. Cells
                                                                   X^{++};
[2]. Value. ToString()
                                 case "company":
                                                                                                     ((LexGrid.Rows[x].Cell
!= "{");
                                                                                                     s[1]. Value. ToString()
                                 //Check if Global unit
                                                                                                     == "[") &&
```

```
(LexGrid.Rows[x +
3]. Cells[1]. Value. ToSt
                                                                   }
ring() == "["))
                                                                                                     main += "public static
                                                                                                     string " +
                                  else if
                                                                    else if
                                                                                                     LexGrid.Rows[x -
main += "public static
                                  (LexGrid. Rows[x]. Cells
                                                                    (LexGrid. Rows[x]. Cells
                                                                                                     1]. Cells[1]. Value. ToSt
string ";
                                  [1]. Value. ToString()
                                                                    [1]. Value. ToString()
                                                                                                     ring() + "; \n";
                                                                   == "(")
                                  == "[")
main +=
                                  {
LexGrid.Rows[x].Cells[
1].Value.ToString() +
                                                                                                     break;
"," + "]" + " " +
                                  main += "public static
                                                                   main += "public static
LexGrid.Rows[x -
                                  string ";
                                                                    string ";
                                                                                                     case ", ":
1]. Cells[1]. Value. ToSt
ring();
                                  main +=
                                                                    main +=
                                                                                                     X^{++};
                                  LexGrid. Rows[x]. Cells[
                                                                   LexGrid.Rows[x].Cells[
x += 6;
                                  1]. Value. ToString() +
                                                                    1]. Value. ToString();
                                                                                                     break;
                                  "]" + " " +
                                  LexGrid.Rows[x -
(LexGrid.Rows[x].Cells
                                  1]. Cells[1]. Value. ToSt
[1]. Value. ToString()
                                 ring();
                                                                                                     } while
== "=")
                                                                                                      (LexGrid. Rows[x]. Cells
                                                                                                     [2]. Value. ToString()
                                  X += 3;
                                                                    main +=
{
                                                                   LexGrid.Rows[x].Cells[
                                                                                                     != ";");
                                                                    1]. Value. ToString();
                                  (LexGrid.Rows[x].Cells
do
                                  [1]. Value. ToString()
                                                                    x++;
{
                                  == "=")
                                                                                                     else
                                                                   } while
                                  {
                                                                    (LexGrid. Rows[x]. Cells
main +=
LexGrid. Rows[x]. Cells[
                                                                    [2]. Value. ToString()
1]. Value. ToString();
                                                                    != "{");
                                                                                                     main += "string";
                                  do
                                                                                                     }
                                  {
X^{++};
                                                                    main +=
                                                                   LexGrid.Rows[x].Cells[
} while
                                                                    1]. Value. ToString();
                                  main +=
                                                                                                     break;
(LexGrid.Rows[x].Cells
                                 LexGrid.Rows[x].Cells[
[1]. Value. ToString()
                                                                                                     }
                                 1]. Value. ToString();
!= ";");
                                                                    else if
                                 x++:
                                                                                                     else
main += "; \n";
                                                                    (LexGrid.Rows[x].Cells
                                  } while
                                                                    [1]. Value. ToString()
// x += 2;
                                  (LexGrid.Rows[x].Cells
                                                                    == "=")
                                  [1]. Value. ToString()
                                                                                                     //Check if Local unit
                                  != "}");
                                                                                                     x++;
                                                                   main += "public static
string " +
else
                                  main += "}; \n";
                                                                                                     do
                                                                   LexGrid.Rows[x -
                                  X^{++};
                                                                    1]. Cells[1]. Value. ToSt
                                                                   ring() + "=" +
main += "= new
string[" +
                                                                   LexGrid.Rows[x +
                                                                                                     switch
                                                                                                      (LexGrid. Rows[x]. Cells
LexGrid.Rows[x -
                                                                   1]. Cells[1]. Value. ToSt
                                  else
5]. Cells[1]. Value. ToSt
                                                                   ring() + "; \n";
                                                                                                     [2]. Value. ToString())
ring() + "," +
LexGrid.Rows[x -
                                                                   X = X + 2;
                                                                                                     {
                                  main += "= new
2]. Cells[1]. Value. ToSt
                                  string[" +
ring():
                                                                                                     case "id":
                                 LexGrid.Rows[x -
main += "]; \n";
                                  2]. Cells[1]. Value. ToSt
                                                                   else
                                                                                                     X^{++};
                                  ring() + "]; \n";
```

if ((LexGrid.Rows[x].Cell	2].Cells[1].Value.ToSt ring();	LexGrid.Rows[x - 2].Cells[1].Value.ToSt ring() + "]; \n";	{
s[1]. Value. ToString() == "[") &&	main += "]; \n";	}	main += "string" + LexGrid.Rows[x -
(LexGrid.Rows[x + 3].Cells[1].Value.ToSt	}	}	1].Cells[1].Value.ToSt ring() + "; \n";
ring() == "["))	}	else if	}
{	else if (LexGrid.Rows[x].Cells	<pre>(LexGrid.Rows[x].Cells [1].Value.ToString()</pre>	break;
main += "string";	[1]. Value. ToString() == "[")	== "(")	case ", ":
main +=		{	
LexGrid.Rows[x].Cells[ 1].Value.ToString() +	{	main += "string";	X <sup>++</sup> ;
", " + "]" + " " +	main += "int ";		break;
LexGrid. Rows[x -		main +=	1
1].Cells[1].Value.ToSt ring();	main += LexGrid.Rows[x].Cells[	<pre>LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	}
·· +- 6.	1].Value.ToString() + "" + "" +	do	} while (LexGrid.Rows[x].Cells
x += 6;	LexGrid.Rows[x -	do	[2]. Value. ToString()
if	1]. Cells[1]. Value. ToSt	{	!= ";");
(LexGrid.Rows[x].Cells	ring();		
[1]. Value. ToString()		main +=	}
== "=")	x += 3;	<pre>LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	break;
{	if	1]. varue. 105tring(),	Dieak,
	(LexGrid.Rows[x].Cells	x++;	case "joe":
do	[1]. Value. ToString()		
ſ	== "=")	} while	//Check if global joe
1	{	(LexGrid.Rows[x].Cells [2].Value.ToString()	
main +=	į	!= "{");	if (checktemp == 0
LexGrid.Rows[x].Cells[	do	. , ,	checkfunc == 0)
1].Value.ToString();		main +=	
	{	LexGrid. Rows[x]. Cells[	{
X++;	main +=	1].Value.ToString();	if (LexGrid.Rows[x +
} while	LexGrid. Rows[x]. Cells[	}	2]. Cells[2]. Value. ToSt
(LexGrid. Rows[x]. Cells	1].Value.ToString();		ring() == "(")
[1]. Value. ToString()		else if	
!= ";");	x++;	(LexGrid. Rows[x]. Cells	{
main += "; \n";	} while	[1]. Value. ToString() == "=")	checkfunc = 1;
, 11	(LexGrid. Rows[x]. Cells	,	checkrune 1,
// x += 2;	[1]. Value. ToString() != "}");	{	x++;
}	. , ,	main += "string" +	main += "public static
	main += "}; \n";	LexGrid.Rows[x -	char ";
else		1]. Cells[1]. Value. ToSt	
ſ	X++;	ring() + "=" +	do
{	}	LexGrid.Rows[x + 1].Cells[1].Value.ToSt	{
main += "= new	j	ring() + "; \n";	· ·
string[" +	else	- • •	if
LexGrid.Rows[x -		X = X + 2;	(LexGrid. Rows[x]. Cells
5]. Cells[1]. Value. ToSt	{	1	[2]. Value. ToString()
ring() + "," + LexGrid.Rows[x -	main += "= new	}	== "digit")
POVOLIA" MORSEY	string[" +	else	{

<pre>main += "double ";</pre>	main += "{\n";	main += LexGrid.Rows[x].Cells[	do
X <sup>++</sup> ;	break;	1]. Value. ToString();	{
}	}	x <sup>++</sup> ;	main += LexGrid.Rows[x].Cells[
else if	if(checkfunc == 0 &&	} while	1]. Value. ToString();
<pre>(LexGrid.Rows[x].Cells [2].Value.ToString() == "unit")</pre>	<pre>checktemp == 0) {</pre>	(LexGrid.Rows[x].Cells [1].Value.ToString() != ";");	x++;
			} while
{	X <sup>++</sup> ;	main += "; \n";	<pre>(LexGrid.Rows[x].Cells [1].Value.ToString()</pre>
main += "int ";	do	// x += 2;	!= "}");
X <sup>++</sup> ;	{	}	main $+=$ "}; \n";
}	switch	else	X++;
else if	<pre>(LexGrid. Rows[x]. Cells [2]. Value. ToString())</pre>	{	}
(LexGrid. Rows[x]. Cells	[2]. varue. rostrring())	l	<b>\</b>
[2]. Value. ToString() == "joe")	{	main += "= new char[" + LexGrid.Rows[x -	else
{	case "id":	5].Cells[1].Value.ToSt ring() + "," +	{
	x++;	LexGrid. Rows[x -	main += "= new char["
main += "char";	if	2]. Cells[1]. Value. ToSt ring();	+ LexGrid.Rows[x - 2].Cells[1].Value.ToSt
X++;	((LexGrid.Rows[x].Cell	1 111g (/ ,	ring() + "]; \n";
,	s[1]. Value. ToString()	main += "]; \n";	0.0 27 ( 7
}	== "[") &&	,	}
-1 :6	(LexGrid.Rows[x + 3].Cells[1].Value.ToSt	}	1
else if (LexGrid.Rows[x].Cells	ring() == "["))	}	,
[2]. Value. ToString()		•	else if
== "company")	{	else if	(LexGrid. Rows[x]. Cells
ſ		(LexGrid.Rows[x].Cells [1].Value.ToString()	[1]. Value. ToString() == "(")
{	<pre>main += "public static char ";</pre>	== "[")	( )
main += "string";	char ,	. /	{
	main +=	{	
x++;	LexGrid. Rows[x]. Cells[		main += "public static
}	1]. Value. ToString() + "," + "]" + " " +	<pre>main += "public static char ";</pre>	char ";
,	LexGrid.Rows[x -	char ,	main +=
else	1]. Cells[1]. Value. ToSt	main +=	LexGrid.Rows[x].Cells[
{	ring();	LexGrid.Rows[x].Cells[ 1].Value.ToString() +	1].Value.ToString();
	x += 6;	"]" + " " +	do
main +=	• 6	LexGrid. Rows[x -	ſ
<pre>LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	if (LexGrid.Rows[x].Cells	1].Cells[1].Value.ToString();	1
ij. varue. iooti ing () ,	[1]. Value. ToString()	ring (),	main +=
x++;	== "=")	x += 3;	<pre>LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>
}	{	if	_
\ while	do	(LexGrid. Rows[x]. Cells	X++;
<pre>} while (LexGrid.Rows[x].Cells</pre>	do	[1]. Value. ToString() == "=")	} while
[1]. Value. ToString()	{	•	(LexGrid.Rows[x].Cells
!= "{");		{	

```
[2]. Value. ToString()
!= "{");
                                  main += "char";
                                                                                                       {
main +=
                                                                    main +=
                                                                                                       do
LexGrid. Rows[x]. Cells[
                                                                    LexGrid. Rows[x]. Cells[
1]. Value. ToString();
                                  break;
                                                                    1]. Value. ToString();
                                                                                                       {
                                                                                                       main +=
                                                                    X^{++};
                                                                                                      LexGrid.Rows[x].Cells[
else if
                                                                    } while
                                                                                                       1]. Value. ToString();
                                  else
(LexGrid.Rows[x].Cells
                                                                    (LexGrid.Rows[x].Cells
[1]. Value. ToString()
                                                                    [1]. Value. ToString()
                                                                                                       X^{++};
== "=")
                                                                    != ";");
                                  //Check if Local unit
                                                                                                       } while
                                                                    main += "; \n";
                                                                                                       (LexGrid. Rows[x]. Cells
{
                                                                                                       [1]. Value. ToString()
                                  x++;
main += "public static
                                                                    //_{X} += 2;
                                                                                                       != "}");
char " +
                                  do
                                                                                                       main += "}; \n";
LexGrid.Rows[x -
1]. Cells[1]. Value. ToSt
ring() + "=" +
                                                                    else
                                                                                                       X^{++};
LexGrid.Rows[x +
                                  switch
1]. Cells[1]. Value. ToSt
                                  (LexGrid. Rows[x]. Cells
ring() + "; \n";
                                  [2]. Value. ToString())
                                                                    main += "= new char["
                                                                                                       else
                                                                    + LexGrid.Rows[x -
X = X + 2;
                                                                    5]. Cells[1]. Value. ToSt
                                                                    ring() + "," +
                                  case "id":
                                                                    LexGrid.Rows[x -
                                                                                                       main += "= new char["
else
                                  x++;
                                                                    2]. Cells[1]. Value. ToSt
                                                                                                       + LexGrid. Rows[x -
                                                                                                      2]. Cells[1]. Value. ToSt
                                                                    ring():
                                                                                                      ring() + "]; \n";
                                  ((LexGrid.Rows[x].Cell
                                                                    main += "]; \n";
main += "public static
                                  s[1]. Value. ToString()
                                                                                                      }
char " +
                                  == "[") &&
                                                                                                      }
                                  (LexGrid.Rows[x +
LexGrid.Rows[x -
1]. Cells[1]. Value. ToSt
                                  3]. Cells[1]. Value. ToSt
ring() + "; \n";
                                  ring() == "["))
                                                                                                       else if
                                                                    else if
                                                                                                       (LexGrid.Rows[x].Cells
                                                                    (LexGrid.Rows[x].Cells
                                                                                                       [1]. Value. ToString()
                                                                    [1]. Value. ToString()
                                                                                                       == "(")
                                  main += "char":
                                                                    == "[")
break:
                                                                                                       {
case ", ":
                                  main +=
                                                                                                       main += "char";
                                  LexGrid.Rows[x].Cells[
                                  1]. Value. ToString() + "," + "]" + " " +
                                                                    main += "char";
X^{++};
                                                                                                      main +=
                                  LexGrid.Rows[x -
                                                                                                      LexGrid.Rows[x].Cells[
break;
                                                                    main +=
                                  1]. Cells[1]. Value. ToSt
                                                                    LexGrid. Rows[x]. Cells[
                                                                                                       1]. Value. ToString();
                                  ring();
                                                                    1]. Value. ToString() +
                                                                    "]" + " " +
                                                                                                      do
                                                                    LexGrid.Rows[x -
} while
                                  X += 6;
(LexGrid. Rows[x]. Cells
                                                                    1]. Cells[1]. Value. ToSt
[2]. Value. ToString()
                                                                    ring();
!= ";");
                                  (LexGrid. Rows[x]. Cells
                                                                                                      main +=
                                  [1]. Value. ToString()
                                                                    X += 3;
                                                                                                      LexGrid.Rows[x].Cells[
                                  == "=")
                                                                                                      1]. Value. ToString();
                                                                    (LexGrid. Rows[x]. Cells
                                  {
                                                                                                      X^{++};
else
                                                                    [1]. Value. ToString()
{
                                                                    == "=")
                                                                                                       } while
                                  do
                                                                                                       (LexGrid. Rows[x]. Cells
```

```
[2]. Value. ToString()
                                                                                                     [2]. Value. ToString()
!= "{");
                                  //Check if Global
                                                                   x++;
                                                                                                     == "joe")
                                  response
main +=
                                                                   main += "public static
LexGrid. Rows[x]. Cells[
                                  if (checktemp == 0)
                                                                   void ";
                                                                                                     main += "char":
1]. Value. ToString();
                                                                   do
}
                                                                                                     X^{++};
                                  X++;
else if
(LexGrid.Rows[x].Cells
                                  if
                                                                   if
[1]. Value. ToString()
                                  (LexGrid. Rows[x]. Cells
                                                                   (LexGrid.Rows[x].Cells
                                                                                                     else if
== "=")
                                  [2]. Value. ToString()
                                                                   [2]. Value. ToString()
                                                                                                     (LexGrid. Rows[x]. Cells
                                                                                                     [2]. Value. ToString()
                                  == "id")
                                                                   == "digit")
                                                                                                     == "company")
{
                                  {
main += "char" +
LexGrid.Rows[x -
                                  main += "boolean";
                                                                   main += "double ";
                                                                                                     main += "string";
1]. Cells[1]. Value. ToSt
ring() + "=" +
                                  main +=
                                                                   x++;
LexGrid.Rows[x +
                                 LexGrid.Rows[x].Cells[
                                                                                                     X^{++};
1]. Cells[1]. Value. ToSt
                                  1]. Value. ToString() +
ring() + "; \n";
                                  "=false; \n";
                                                                   else if
                                                                   (LexGrid.Rows[x].Cells
x = x + 2;
                                                                                                     else
                                                                   [2]. Value. ToString()
                                                                   == "unit")
                                 break;
else
                                                                                                     main +=
                                                                                                     LexGrid.Rows[x].Cells[
{
                                  //Check if local
                                                                   main += "int ":
                                                                                                     1]. Value. ToString():
                                  response
main += "char" +
                                                                   X^{++};
LexGrid.Rows[x -
                                  else
                                                                                                     }
1]. Cells[1]. Value. ToSt
ring() + "; \n";
                                  {
                                                                   ((LexGrid.Rows[x].Cell
                                                                                                     } while
                                  X++;
                                                                   s[1]. Value. ToString()
                                                                                                     (LexGrid.Rows[x].Cells
                                                                   == "[") &&
                                                                                                     [2]. Value. ToString()
                                                                   (LexGrid.Rows[x +
                                                                                                     != "{");
break;
                                  if
                                  (LexGrid. Rows[x]. Cells
                                                                   3]. Cells[1]. Value. ToSt
case ", ":
                                  [2]. Value. ToString()
                                                                   ring() = "["))
                                                                                                     main += "\n" +
                                 == "id")
                                                                                                     LexGrid.Rows[x].Cells[
                                                                                                     1]. Value. ToString() +
X^{++};
                                  {
                                                                                                     "\n";
                                                                   main +=
break;
                                  main += "boolean ";
                                                                   LexGrid.Rows[x].Cells[
                                                                                                     break:
                                                                   1]. Value. ToString() +
                                                                   "," + "]" + " " +
                                                                                                     case "struct":
                                  main +=
                                                                   LexGrid.Rows[x -
} while
                                 LexGrid.Rows[x].Cells[
(LexGrid. Rows[x]. Cells
                                  1]. Value. ToString() +
                                                                   1]. Cells[1]. Value. ToSt
                                                                                                     int temp=0;
[2]. Value. ToString()
                                  "=false; \n";
                                                                   ring();
!= ";");
                                                                                                     main += "public struct
                                                                   x++;
                                 break;
                                                                                                     X^{++};
break;
                                                                                                     main +=
                                                                                                     LexGrid.Rows[x].Cells[
case "response":
                                 case "miss":
                                                                                                     1]. Value. ToString() +
                                                                   (LexGrid. Rows[x]. Cells
                                                                                                     "{ \n";
                                  checkfunc = 1;
```

```
LexGrid.Rows[x].Cells[
x += 2;
                                                                                                    1]. Value. ToString();
                                                                   do
do
                                 else if
                                                                                                    X^{++};
                                 (LexGrid. Rows[x]. Cells
                                 [2]. Value. ToString()
{
                                 == "company")
                                                                   (LexGrid.Rows[x].Cells
                                                                   [2]. Value. ToString()
                                                                  == "digit")
((LexGrid.Rows[x].Cell
                                                                                                    } while
s[2]. Value. ToString()
                                                                                                     (LexGrid.Rows[x].Cells
== "}") &&
                                 temp = 0;
                                                                                                    [2]. Value. ToString()
(LexGrid.Rows[x +
                                                                                                    != ";");
                                 main += "public string
1]. Cells[2]. Value. ToSt
                                                                  main += "double ";
ring() == ";"))
                                                                                                    main += "; \n";
                                                                   X^{++};
{
                                 x++;
                                                                                                    break;
temp = 1;
                                                                                                    case "capture":
                                                                   else if
main += "};\n";
                                                                   (LexGrid.Rows[x].Cells
                                                                                                    X++;
                                 else if
                                                                   [2]. Value. ToString()
                                                                  == "unit")
X^{++};
                                 (LexGrid.Rows[x].Cells
                                 [2]. Value. ToString()
                                                                                                    (LexGrid. Rows[x]. Cells
                                 == "joe")
                                                                                                    [2]. Value. ToString()
                                                                                                    == "(")
                                 {
                                                                   main += "int ";
                                                                                                    {
else
                                 temp = 0:
                                                                   X++;
{
                                                                                                    x++:
                                 main += "public char
temp = 0;
                                 ";
                                                                                                    do
                                 X^{++};
                                                                   (LexGrid. Rows[x]. Cells
(LexGrid. Rows[x]. Cells
                                                                   [2]. Value. ToString()
[2]. Value. ToString()
                                                                  == "joe")
                                                                                                    if
== "unit")
                                                                                                    (LexGrid.Rows[x].Cells
                                                                                                    [2]. Value. ToString()
{
                                 else {
                                                                                                    == "#")
                                                                   main += "char ";
                                 temp = 0;
temp = 0;
                                                                                                    {
                                                                   X^{++};
main += "public int ";
                                 main +=
                                                                                                    x++:
                                 LexGrid.Rows[x].Cells[
                                 1]. Value. ToString();
X^{++};
                                                                                                    if
                                                                   else if
                                                                                                    (LexGrid.Rows[x].Cells
                                 x++;
                                                                   (LexGrid.Rows[x].Cells
                                                                                                    [2]. Value. ToString()
                                                                                                    == "id")
                                                                   [2]. Value. ToString()
                                                                   == "company")
else if
                                                                                                    {
(LexGrid.Rows[x].Cells
                                                                                                    foreach (string a in
[2]. Value. ToString()
== "digit")
                                                                   main += "string";
                                                                                                    intlist)
                                 } while (temp == 0);
{
                                                                                                    {
                                                                   x++;
                                 break;
                                                                                                    if (a ==
temp = 0;
                                 case "hold":
                                                                                                    LexGrid.Rows[x].Cells[
main += "public double
                                                                   else
                                                                                                    1]. Value. ToString())
                                 main += "const";
";
                                                                                                    {
x++;
                                 X^{++};
                                                                   main +=
```

<pre>if   (LexGrid.Rows[x].Cells [1].Value.ToString() == "[") {   if (LexGrid.Rows[x + 3].Cells[1].Value.ToSt   ring() == "[") {   x;   do</pre>	<pre>//main += "GlobalVar." + LexGrid. Rows[x - 1]. Cells[1]. Value. ToSt ring() + LexGrid. Rows[x]. Cells[ 1]. Value. ToString() + LexGrid. Rows[x + 1]. Cells[1]. Value. ToSt ring() + LexGrid. Rows[x + 2]. Cells[1]. Value. ToSt ring(); main += " = Convert. ToInt32 (Consol e. ReadLine()); \n";</pre>	<pre>//main += "GlobalVar." + LexGrid. Rows[x - 1]. Cells[1]. Value. ToSt ring() + LexGrid. Rows[x]. Cells[ 1]. Value. ToString() + LexGrid. Rows[x + 1]. Cells[1]. Value. ToSt ring() + LexGrid. Rows[x + 2]. Cells[1]. Value. ToSt ring(); main += " = Convert. ToInt32 (Consol e. ReadLine()); \n";</pre>	<pre>{   if (LexGrid.Rows[x +     3].Cells[1].Value.ToSt   ring() == "[")   {   x;   do   {     switch     (LexGrid.Rows[x].Cells   [2].Value.ToString())</pre>
{	}	}	{
<pre>switch (LexGrid.Rows[x].Cells</pre>	else	}	case "id":
[2]. Value. ToString())	{	else	
{	x;	{	<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>
case "id":	do	outp. Add (LexGrid. Rows[x]. Cells[1]. Value. ToSt	break;
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	{ switch	ring()); main += LexGrid.Rows[x	default:
break;	(LexGrid.Rows[x].Cells [2].Value.ToString())	- 1].Cells[1].Value.ToSt ring() + "=";	<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString() +</pre>
default:	{	main +=	LexGrid. Rows[x + 1]. Cells[1]. Value. ToSt
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString() + LexGrid.Rows[x+1].Cell s[1].Value.ToString() + "," +</pre>	<pre>case "id":  main += LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	"Convert. ToInt32(Conso le. ReadLine()); \n"; }	ring() + "," + LexGrid. Rows[x + 4]. Cells[1]. Value. ToSt ring() + LexGrid. Rows[x + 5]. Cells[1]. Value. ToSt
LexGrid.Rows[x+4].Cells[1].Value.ToString()	break;	}	ring();
+ LexGrid.Rows[x+5].Cell	default:	foreach (string b in	x = x + 5;
s[1]. Value. ToString();	main +=	doublelist)	break;
x = x + 5;	LexGrid.Rows[x].Cells[ 1].Value.ToString();	{	}
break;	break;	if (b == LexGrid.Rows[x].Cells[	X++;
}	}	1]. Value. ToString())	<pre>} while (LexGrid.Rows[x].Cells</pre>
x++;	x++;	{	[1]. Value. ToString() != ")");
<pre>} while (LexGrid.Rows[x].Cells [1].Value.ToString() != ")");</pre>	<pre>} while (LexGrid.Rows[x].Cells [1].Value.ToString() != ")");</pre>	<pre>x++; if (LexGrid.Rows[x].Cells [1].Value.ToString() == "[")</pre>	<pre>//main += "GlobalVar." + LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() +</pre>

LexGrid.Rows[x].Cells[ 1].Value.ToString() +	1].Value.ToString() + LexGrid.Rows[x +	x++;	[1]. Value. ToString() != ")");
LexGrid.Rows[x +	1].Cells[1].Value.ToSt		
1]. Cells[1]. Value. ToSt	ring() +	if	//
ring() + LexGrid.Rows[x +	LexGrid.Rows[x + 2].Cells[1].Value.ToSt	(LexGrid. Rows[x]. Cells	//main += "GlobalVar."
2]. Cells[1]. Value. ToSt	2]. Cells[1]. Value. 10St ring();	[1].Value.ToString() == "[")	+ LexGrid.Rows[x - 1].Cells[1].Value.ToSt
ring();	1111g(),	L )	ring() +
1110 (/ )	main += " =	{	LexGrid. Rows[x]. Cells[
main += " =	Convert.ToDouble(Conso		1].Value.ToString() +
Convert. ToDouble(Conso	<pre>le.ReadLine()); \n";</pre>	if (LexGrid.Rows[x +	LexGrid.Rows[x +
<pre>le.ReadLine()); \n";</pre>		3].Cells[1].Value.ToSt	1].Cells[1].Value.ToSt
,	}	ring() == "[")	ring() +
}	1	ſ	LexGrid. Rows[x +
else	}	{	2]. Cells[1]. Value. ToSt ring();
6126	else	x;	1111g(),
{	0100	,	main += " =
	{	do	<pre>Console.ReadLine();</pre>
x;			n'';
	//outp. Add (LexGrid. Row	{	
do	s[x]. Cells[1]. Value. To		}
ſ	String());	switch	.1.
1	main += LexGrid.Rows[x	<pre>(LexGrid.Rows[x].Cells [2].Value.ToString())</pre>	else
switch	- Lexoriu. Rows[x	[2]. varue. lostring())	{
(LexGrid. Rows[x]. Cells	1]. Cells[1]. Value. ToSt	{	
[2]. Value. ToString())	ring() + "=";		x;
		case "id":	
{	main +=		do
// - 1//	"Convert. ToDouble (Cons	main +=	ſ
case "id":	ole.ReadLine()); \n";	LexGrid. Rows[x]. Cells[	{
main +=	}	1].Value.ToString();	switch
LexGrid. Rows[x]. Cells[	J	break;	(LexGrid. Rows[x]. Cells
1]. Value. ToString();	//outp. Add (LexGrid. Row	,	[2]. Value. ToString())
	s[x].Cells[1].Value.To	default:	
break;	String());		{
		main +=	W W
default:	//consolewrt += LexGrid.Rows[x].Cells[	LexGrid. Rows[x]. Cells[	case "id":
main +=	1]. Value. ToString() +	1].Value.ToString() + LexGrid.Rows[x +	main +=
LexGrid. Rows[x]. Cells[	"=";	1]. Cells[1]. Value. ToSt	LexGrid. Rows[x]. Cells[
1]. Value. ToString();	,	ring() + "," +	1]. Value. ToString();
	//consolewrt +=	LexGrid.Rows[x +	
break;	"Convert.ToDouble(Cons	4]. Cells[1]. Value. ToSt	break;
,	ole.ReadLine()); \n";	ring() +	
}	1	LexGrid. Rows[x +	default:
v44.	}	5].Cells[1].Value.ToSt ring();	main +=
X++;	}	1111g(),	LexGrid. Rows[x]. Cells[
} while	,	x = x + 5;	1]. Value. ToString();
(LexGrid. Rows[x]. Cells	foreach (string c in	•	
[1]. Value. ToString()	stringlist)	break;	break;
!= ")");		,	1
	{	}	}
//main += "GlobalVar."	if (c ==	v++·	v++·
+ LexGrid. Rows[x -	LexGrid. Rows[x]. Cells[	X++;	X++;
1]. Cells[1]. Value. ToSt	1]. Value. ToString())	} while	} while
ring() +	5	(LexGrid. Rows[x]. Cells	(LexGrid.Rows[x].Cells
LexGrid.Rows[x].Cells[	{		

[1].Value.ToString() != ")");	foreach (string c in charlist)	x = x + 5;	LexGrid.Rows[x].Cells[ 1].Value.ToString();
// •		break;	break;
//main += "GlobalVar." + LexGrid.Rows[x -	{	}	}
1].Cells[1].Value.ToSt ring() +	if (c == LexGrid.Rows[x].Cells[	x++;	x++;
LexGrid.Rows[x].Cells[ 1].Value.ToString() +	1].Value.ToString())	} while	} while
LexGrid.Rows[x + 1].Cells[1].Value.ToSt	{	<pre>(LexGrid.Rows[x].Cells [1].Value.ToString()</pre>	<pre>(LexGrid.Rows[x].Cells [1].Value.ToString()</pre>
ring() + LexGrid.Rows[x +	X++;	!= ")");	!= ")");
2].Cells[1].Value.ToSt	if		
ring();	<pre>(LexGrid.Rows[x].Cells [1].Value.ToString()</pre>	//main += "GlobalVar." + LexGrid.Rows[x -	//main += "GlobalVar." + LexGrid.Rows[x -
<pre>main += " = Console.ReadLine();</pre>	== "[")	1].Cells[1].Value.ToSt ring() +	1]. Cells[1]. Value. ToSt ring() +
\n";	{	LexGrid.Rows[x].Cells[	LexGrid.Rows[x].Cells[
}	if (LexGrid.Rows[x +	1].Value.ToString() + LexGrid.Rows[x +	1].Value.ToString() + LexGrid.Rows[x +
}	3].Cells[1].Value.ToSt ring() == "[")	1]. Cells[1]. Value. ToSt ring() +	1]. Cells[1]. Value. ToSt ring() +
else	{	<pre>LexGrid.Rows[x + 2].Cells[1].Value.ToSt ring();</pre>	<pre>LexGrid. Rows[x + 2]. Cells[1]. Value. ToSt ring();</pre>
{	х;	I Ilig (),	1 1 lig (),
//outp. Add (LexGrid. Row	do	main += " = Console.ReadKey().KeyC	main += " = Console.ReadKey().KeyC
<pre>s[x].Cells[1].Value.To String());</pre>	{	har; \n";	har; \n";
		}	}
main += LexGrid. Rows[x	switch (LexGrid.Rows[x].Cells	else	}
1].Cells[1].Value.ToSt ring() + "=";	<pre>[2]. Value. ToString()) {</pre>	{	else
main +=	(	х;	{
"Console.ReadLine(); \n";	case "id":	do	//outp.Add(LexGrid.Row
\ ,	main +=		s[x].Cells[1].Value.To
}	<pre>LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	{	String());
//outp.Add(LexGrid.Row	break;	switch (LexGrid.Rows[x].Cells	main += LexGrid.Rows[x
s[x]. Cells[1]. Value. To	or cak,	[2]. Value. ToString())	1].Cells[1].Value.ToSt
String());	default:	ſ	ring() + "=";
//consolewrt +=	main +=	{	main +=
LexGrid.Rows[x].Cells[	LexGrid.Rows[x].Cells[	case "id":	"Console.ReadKey().Key
1].Value.ToString() + "=":	1].Value.ToString() + LexGrid.Rows[x +	main +=	Char; \n";
//consolewrt +=	1]. Cells[1]. Value. ToSt ring() + "," +	LexGrid.Rows[x].Cells[ 1].Value.ToString();	}
"Console. ReadLine();	LexGrid.Rows[x +	- 0	//outp. Add (LexGrid. Row
\n";	4].Cells[1].Value.ToSt ring() +	break;	<pre>s[x].Cells[1].Value.To String());</pre>
}	LexGrid.Rows[x +	default:	-
}	5].Cells[1].Value.ToSt ring();	main +=	<pre>//consolewrt += LexGrid.Rows[x].Cells[</pre>

```
1]. Value. ToString() +
                                  2]. Value. ToString() ==
                                  "id")
                                                                   case "NEGATIVE":
                                                                                                     main += " ] ";
//consolewrt +=
                                  {
                                                                   main += "false";
                                                                                                     X^{++};
"Console. ReadLine();
n'';
                                  do
                                                                    x++;
                                                                                                     break;
                                                                                                     case "(":
                                                                   break;
                                                                   case "id":
                                                                                                     main += "(";
                                  switch
                                  (LexGrid. Rows[x]. Cells
                                  [2]. Value. ToString())
                                                                   main +=
                                                                                                     X^{++};
                                                                   LexGrid.Rows[x].Cells[
                                  {
                                                                   1]. Value. ToString();
                                                                                                     break;
                                  case "Stringlit":
                                                                                                     case ")":
X^{++};
                                                                   X^{++};
} while
                                  main +=
                                                                   break;
                                                                                                     main += ")";
(LexGrid.Rows[x].Cells
                                  LexGrid.Rows[x].Cells[
[2]. Value. ToString()
                                  1]. Value. ToString();
                                                                    case ", ":
                                                                                                     X++;
!= ";");
                                                                   main += ", ";
                                  X++;
                                                                                                     break;
                                                                    x++;
                                                                                                     default:
                                  break;
                                  case "Numlit":
                                                                                                     main +=
break;
                                                                   break;
                                                                                                     LexGrid.Rows[x].Cells[
                                                                   case "+":
                                  main +=
                                                                                                     1]. Value. ToString();
case "post":
                                  LexGrid.Rows[x].Cells[
                                  1]. Value. ToString();
                                                                   main += " + ":
                                                                                                     x++;
x++:
                                                                   X^{++};
                                                                                                     break;
                                  X^{++};
(LexGrid. Rows[x]. Cells
                                  break;
                                                                   break;
[2]. Value. ToString()
== "(")
                                  case "Declit":
                                                                   case "[":
                                                                                                     } while
                                                                                                      (LexGrid. Rows[x]. Cells
{
                                  main +=
                                                                   main += " [ ";
                                                                                                      [2]. Value. ToString()
                                  LexGrid.Rows[x].Cells[
                                                                                                      != ";");
main +=
                                  1]. Value. ToString();
                                                                   X^{++};
"Console. Write(";
                                  x++:
do
                                                                    (LexGrid. Rows[x+2]. Cel
                                                                    ls[1]. Value. ToString()
                                  break;
                                                                                                     } while
                                                                   == "[")
                                                                                                      (LexGrid.Rows[x].Cells
                                  case "Charlit":
                                                                                                      [2]. Value. ToString()
                                                                                                      != ";");
                                  main +=
                                                                                                     main += "; \n";
                                  LexGrid. Rows [x]. Cells[
                                                                    main +=
(LexGrid.Rows[x].Cells
                                                                   LexGrid.Rows[x].Cells[
                                  1]. Value. ToString();
[2]. Value. ToString()
                                                                   1]. Value. ToString() +
                                                                    "," + LexGrid.Rows[x +
== "Stringlit" ||
                                  X++;
LexGrid.Rows[x].Cells[
                                                                   3]. Cells[1]. Value. ToSt
                                                                                                     break;
2].Value.ToString() ==
                                                                   ring() + "]";
                                  break;
"Numlit" ||
                                                                                                     case "id":
LexGrid.Rows[x].Cells[
                                  case "AFFIRMATIVE":
                                                                   X += 5;
2]. Value. ToString() ==
                                                                                                     do
"Charlit" ||
                                  main += "true";
LexGrid.Rows[x].Cells[
                                                                                                      {
2]. Value. ToString() ==
                                  X^{++};
                                                                   break;
"Declit" ||
                                                                                                     switch
                                                                   case "]":
LexGrid.Rows[x].Cells[
                                  break;
```

```
(LexGrid. Rows[x]. Cells
[2]. Value. ToString())
                                                                    case "^":
                                                                                                      do
                                  break;
                                                                    main +=
                                                                    LexGrid. Rows[x]. Cells[
                                  case "=":
case "id":
                                                                    1]. Value. ToString();
                                                                                                      main +=
                                                                                                      LexGrid. Rows[x]. Cells[
main += " " +
                                  main += "=";
                                                                                                      1]. Value. ToString();
                                                                    X^{++};
LexGrid.Rows[x].Cells[
1]. Value. ToString();
                                  x++;
                                                                    break;
                                                                                                      X^{++};
                                                                    case "(":
                                                                                                      } while
X^{++};
                                  break;
                                                                                                      (LexGrid. Rows[x]. Cells
                                  case "+":
if (LexGrid. Rows[x]. Cel
                                                                    main +=
                                                                                                      [1]. Value. ToString()
                                                                                                      != ";");
ls[1]. Value. ToString()
                                                                    LexGrid.Rows[x].Cells[
== "[")
                                  main +=
                                                                    1]. Value. ToString();
                                  LexGrid.Rows[x].Cells[
                                                                                                      break;
{
                                  1]. Value. ToString();
                                                                    X^{++};
                                                                                                      case "sqrt":
                                  X++;
                                                                    break;
(LexGrid. Rows[x+3]. Cel
                                                                                                      X^{++};
                                                                    case ")":
1s[1]. Value. ToString()
                                  break;
== "[")
                                                                                                      main += "Math. Sqrt";
                                  case "-":
                                                                    main +=
{
                                                                    LexGrid.Rows[x].Cells[
                                                                                                      break;
                                  main +=
                                                                    1]. Value. ToString();
                                  LexGrid.Rows[x].Cells[
                                                                                                      default:
main +=
LexGrid.Rows[x].Cells[
                                  1]. Value. ToString();
                                                                    x++;
1]. Value. ToString() +
                                                                                                      //if
LexGrid. Rows[x +
                                  x++;
                                                                    break:
                                                                                                      (LexGrid.Rows[x].Cells
                                                                                                      [1]. Value. ToString()
1]. Cells[1]. Value. ToSt
                                                                    case "~":
ring() + "," +
                                  break;
                                                                                                      == "[")
LexGrid. Rows[x +
4]. Cells[1]. Value. ToSt
                                  case "*":
                                                                    main += " -";
                                                                                                      //{
ring() + "]";
                                                                                                            main += "[";
                                  main +=
                                                                    x++;
x += 6;
                                  LexGrid.Rows[x].Cells[
                                  1]. Value. ToString();
                                                                                                            X^{++};
                                                                    break;
break;
                                                                    case "Extent":
                                  X^{++};
                                                                                                            do
                                                                    main += "Length";
                                                                                                      //
                                                                                                          {
                                  break:
else
                                  case "/":
                                                                    X^{++};
                                                                                                                 if
                                                                                                      (LexGrid.Rows[x +
{
                                  main +=
                                                                                                      3]. Cells[2]. Value. ToSt
                                                                    break;
main +=
                                  LexGrid. Rows[x]. Cells[
                                                                                                      ring() == "=")
LexGrid. Rows[x]. Cells[
                                  1]. Value. ToString();
                                                                    case "ToJoeRange":
                                                                                                            {
1]. Value. ToString() +
                                                                    main +=
LexGrid. Rows [x +
                                  X^{++};
1]. Cells[1]. Value. ToSt
                                                                                                      //
                                                                    "ToCharArray()";
                                                                                                                     do
ring() +
                                  break;
LexGrid.Rows[x +
                                                                                                      //
                                                                                                                     {
                                                                    x++;
2]. Cells[1]. Value. ToSt
                                  case "%":
ring();
                                                                                                      //
                                                                    break;
                                                                                                                         main
                                  main +=
x += 3;
                                  LexGrid.Rows[x].Cells[
                                                                    case "Carry":
                                                                                                      LexGrid.Rows[x].Cells[
                                  1]. Value. ToString();
                                                                                                      1]. Value. ToString();
                                                                    main += "Contains";
break;
                                  X^{++};
                                                                                                      //
                                                                                                                         X^{++};
                                                                    X^{++};
                                  break;
                                                                                                                     } while
```

(LexGrid.Rows[x].Cells			
[1]. Value. ToString()	x++;	} while	case "order":
!= ";");		(LexGrid.Rows[x].Cells	
	} while	[2]. Value. ToString()	x++;
// }	(LexGrid.Rows[x].Cells	!= "{");	
	[1]. Value. ToString()		<pre>main += "else {\n";</pre>
// else	!= ";");	main +=	
		LexGrid.Rows[x].Cells[	break;
// {	main += ";";	1]. Value. ToString() +	
		"\n";	case "campaign":
// main +=	break;		
LexGrid. Rows[x]. Cells[	<i>" - "</i>	// <sub>X</sub> ++;	x++;
1]. Value. ToString() +	case "@":	1 1	" 1 "
", " + LexGrid. Rows[x +	1	break;	main += "switch";
2].Cells[1].Value.ToSt ring() + "]";	do	case "otherorder":	J -
ring() + ];	{	case otherorder:	do
// x += 5;	l	x++;	{
// x '- 3,	x++;	A · · · ,	(
// }	A··,	<pre>main += "else if";</pre>	if
,,	} while	main olse ii ,	(LexGrid.Rows[x].Cells
// } while	(LexGrid. Rows[x]. Cells	do	[2]. Value. ToString()
(LexGrid. Rows[x]. Cells	[1]. Value. ToString()		== "id")
[1]. Value. ToString()	!= "@");	{	
!= ";");			{
	break;	if	
//} else {		(LexGrid.Rows[x].Cells	main +=
	case "inorder":	[2]. Value. ToString()	LexGrid.Rows[x].Cells[
main +=		== "Carry")	1].Value.ToString();
LexGrid.Rows[x].Cells[	X <sup>++</sup> ;		
1]. Value. ToString();	. ". "	{	x++;
X++;	main += "if";	·	1
//}	do	<pre>main += "Contains";</pre>	}
//]	do	x++;	else
break;	{	A · · · ,	C13C
si san,		}	{
}	if	•	•
	(LexGrid.Rows[x].Cells	else	main +=
} while	[2]. Value. ToString()		LexGrid.Rows[x].Cells[
(LexGrid.Rows[x].Cells	== "Carry")	{	1].Value.ToString();
[2]. Value. ToString()			
!= ";") ;	{	main +=	x++;
		LexGrid.Rows[x].Cells[	
main += "; \n";	main += "Contains";	1].Value.ToString();	}
1 1			1 1 1
break;	X++;	X++;	} while
case "Swap":	}	1	(LexGrid.Rows[x].Cells [2].Value.ToString()
case Swap .	J	J	!= ")");
main +=	else	} while	. , , ,
"Array. Reverse";	0150	(LexGrid. Rows[x]. Cells	main +=
,	{	[2]. Value. ToString()	LexGrid.Rows[x].Cells[
X++;		!= "{");	1]. Value. ToString() +
	main +=		"\n" + "{ \n";
do	<pre>LexGrid.Rows[x].Cells[</pre>	main +=	•
	1].Value.ToString();	LexGrid.Rows[x].Cells[	break;
{		1].Value.ToString() +	
	x++;	"\n";	<pre>case "operation":</pre>
main +=			
LexGrid. Rows[x]. Cells[	}	break;	main += "case";
1].Value.ToString();			

```
case "inquire":
                                                                    main += "--";
                                                                                                     main += "while";
X^{++};
do
                                  x++:
                                                                    x++;
                                  main += "for":
{
                                                                    do {
                                                                                                      {
                                  do
(LexGrid.Rows[x].Cells
                                                                    main +=
                                                                                                      (LexGrid.Rows[x].Cells
[2]. Value. ToString()
                                                                   LexGrid.Rows[x].Cells[
                                                                                                      [2]. Value. ToString()
== "id")
                                                                    1]. Value. ToString();
                                                                                                      == "id")
{
                                  (LexGrid. Rows[x]. Cells
                                                                    x++;
                                  [2]. Value. ToString()
                                  == "id")
main +=
                                                                                                     main +=
LexGrid.Rows[x].Cells[
                                                                                                     LexGrid.Rows[x].Cells[
1]. Value. ToString();
                                  {
                                                                    while (LexGrid. Rows [x].
                                                                                                      1]. Value. ToString();
                                                                   Cells[2]. Value. ToStrin
                                                                   g() != ";");
                                  main +=
                                  LexGrid.Rows[x].Cells[
                                  1]. Value. ToString();
                                                                    main += "; \n";
                                                                                                      else if
                                                                   break;
else
                                  X^{++};
                                                                                                      (LexGrid. Rows[x]. Cells
{
                                                                    case "++":
                                                                                                      [2]. Value. ToString()
                                                                                                     == "&")
                                                                    main += "++":
main +=
                                  else {
LexGrid.Rows[x].Cells[
1]. Value. ToString();
                                  main +=
                                                                    x++;
                                  LexGrid.Rows[x].Cells[
                                                                                                     main += " && ":
                                  1]. Value. ToString();
x++:
                                                                    do
                                                                                                     X^{++};
                                  X^{++};
} while
(LexGrid.Rows[x].Cells
                                                                    main +=
                                                                                                      else
[1]. Value. ToString()
                                  } while
                                                                   LexGrid.Rows[x].Cells[
                                  (LexGrid.Rows[x].Cells
!= ":");
                                                                    1]. Value. ToString();
                                                                                                      {
                                  [2]. Value. ToString()
                                  != ")");
main +=
                                                                                                     main +=
                                                                    X++;
LexGrid.Rows[x].Cells[
                                                                                                     LexGrid.Rows[x].Cells[
1]. Value. ToString() +
                                  main +=
                                                                                                      1]. Value. ToString();
                                  LexGrid.Rows[x].Cells[
                                                                    } while
                                  1]. Value. ToString() +
                                                                    (LexGrid.Rows[x].Cells
                                                                                                     X^{++};
break;
                                                                    [2]. Value. ToString()
                                  "\n";
                                                                    != ";");
case "abort":
                                  x++;
                                                                    main += ";\n";
                                                                                                      } while
                                  main += "{ \setminus n"};
                                                                                                      (LexGrid. Rows[x]. Cells
x += 3;
                                                                                                      [2]. Value. ToString()
                                                                   break;
main += "break; \n";
                                                                                                      != ")");
                                  break;
                                                                    case "go":
                                  case "action":
                                                                                                     main +=
break;
                                                                                                     LexGrid.Rows[x].Cells[
                                                                    x++;
case "sqrt":
                                  x++;
                                                                                                     1]. Value. ToString();
                                                                    main += "do {";
X^{++};
                                  main += "default: \n";
                                                                                                     X^{++};
                                                                   break;
main += "Math.Sqrt";
                                  break;
                                                                    case "phase":
                                                                                                      (LexGrid. Rows[x]. Cells
                                  case "--":
break;
                                                                                                      [1]. Value. ToString()
                                                                    X++;
                                                                                                      == ";")
```

			; £
{	break;	CodeDomProvider	if (CompErr.ErrorText.Con
·	break,	codeProvider =	tains ("test. test"))
main += "; \n";	case "commence":	CodeDomProvider.Create	{
		<pre>Provider("CSharp");</pre>	
}	main +=	string	}
1	"Console.Clear(); \n";	Output = "Out. exe";	1 (
else	v++.	// Button ButtonObject =	else {
{	X++;	(Buttonobject -	x++;
	break;	(Barron, Benaul,	,
main += " { \n";	}		dataGridViewX4.Rows.Ad
		System.CodeDom.Compile	d(x,
}	if	r.CompilerParameters	CompErr.ErrorText);
break;	<pre>(LexGrid.Rows[x].Cells [2].Value.ToString()</pre>	<pre>parameters = new CompilerParameters();</pre>	}
Dieak,	== "}")	compiler arameters (),	}
case "backup":	{	parameters.GenerateExe	else
		cutable = true;	{
X <sup>++</sup> ;	X <sup>++</sup> ;		
main += "return ":	if	parameters.OutputAssem	textBox2. ForeColor =
main += return ;	(LexGrid.Rows[x].Cells [2].Value.ToString()	b1y = Output;	Color.Blue;
do	== "deploy")	CompilerResults	textBox2. Text =
	{	results =	"Success!";
{		codeProvider.CompileAs	
• 6	x += 2;	semblyFromSource(param	Process. Start (Output);
if (LexGrid.Rows[x].Cells	main +=	eters, richTextBoxEx1.Text);	buttonX12.Enabled =
[2]. Value. ToString()	"Console. ReadLine();	TICHTEX (DOXEAT: TEXT),	false;
== "id")	\n } \n";	if	•
14 /	(11 ) (11 ,	11	
14 /	\n' , \n' , \right	(results.Errors.Count	<pre>buttonX13.Enabled =</pre>
{	}	==	<pre>buttonX13.Enabled = true;</pre>
{	else {	(results.Errors.Count	
{ main += LexGrid.Rows[x].Cells[	}	(results.Errors.Count	
{ main +=	}	<pre>(results.Errors.Count &gt; 0) {</pre>	
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	else { main += LexGrid.Rows[x	<pre>(results.Errors.Count &gt; 0)</pre>	<pre>true;</pre>
<pre>f main += LexGrid.Rows[x].Cells[</pre>	else { main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt	<pre>(results.Errors.Count &gt; 0)</pre>	<pre>true;</pre>
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	else { main += LexGrid.Rows[x	<pre>(results.Errors.Count &gt; 0)</pre>	<pre>true;</pre>
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";</pre>	<pre>(results.Errors.Count &gt; 0)</pre>	<pre>true;</pre>
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString();</pre>	else { main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt	<pre>(results.Errors.Count &gt; 0)</pre>	<pre>true;</pre>
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; }</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";</pre>	<pre>(results.Errors.Count &gt; 0)</pre>	<pre>true;</pre>
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; }</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else {</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; }</pre>	<pre></pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs
<pre>{ main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++;</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else {</pre>	<pre></pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  }
<pre>{ main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++;</pre>	<pre></pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender,
<pre>{ main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++;</pre>	<pre></pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha
<pre>{ main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++; continue; }</pre>	<pre></pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender,
<pre>{ main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++; continue; } while</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";  x; }  globdeclare += "\n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender,
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++; continue; } while (LexGrid.Rows[x].Cells [2].Value.ToString()</pre>	<pre></pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender,
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++; continue; } while (LexGrid.Rows[x].Cells</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";  x; }  globdeclare += "\n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender, EventArgs e)  {  private void
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString();  x++; } else {     x++;     continue; }     while     (LexGrid.Rows[x].Cells [2].Value.ToString() != ";");</pre>	<pre>else  main += LexGrid. Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";  x; }  globdeclare += "\n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender, EventArgs e)  {  private void buttonX13_Click(object
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++; continue; } while (LexGrid.Rows[x].Cells [2].Value.ToString() != ";"); main +=</pre>	<pre>else  main += LexGrid.Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";  x; }  globdeclare += "\n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender, EventArgs e)  {  private void
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++; continue; }  while (LexGrid.Rows[x].Cells [2].Value.ToString() != ";"); main += LexGrid.Rows[x].Cells[</pre>	<pre>else  main += LexGrid. Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";  x;  }  globdeclare += "\n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender, EventArgs e)  {  private void buttonX13_Click(object
<pre>main += LexGrid.Rows[x].Cells[ 1].Value.ToString(); x++; } else { x++; continue; } while (LexGrid.Rows[x].Cells [2].Value.ToString() != ";"); main +=</pre>	<pre>else  main += LexGrid. Rows[x - 1].Cells[1].Value.ToSt ring() + " \n";  x; }  globdeclare += "\n";</pre>	<pre>(results. Errors. Count &gt; 0)</pre>	true;  }  private void superTabControl1_Selec tedTabChanged(object sender, DevComponents.DotNetBa r.SuperTabStripSelecte dTabChangedEventArgs e)  {  private void richTextBoxEx1_TextCha nged(object sender, EventArgs e)  {  private void buttonX13_Click(object

	new List <string> {</string>	public	public
File.Delete("Out.exe")	"go", "order" };	List <string> rs_2 =</string>	List <string> rs_17 =</string>
:	public	new List <string> {</string>	new List <string> {</string>
consolewrt	List <string> rw 4 =</string>	"++", "" };	"  " };
= "":	new List <string> {</string>	public	// public
•	"AFFIRMATIVE",	List <string> rs 3 =</string>	List <string> rs 18 =</string>
richTextBoxEx1.Text =	"NEGATIVE",	new List <string> {</string>	new List <string> { "\$"</string>
"";	"commence",	"+=", "+(", "-=", "-	};
	"ToJoeRange", "Extent"	(", "*(", "/(", "=="	public
buttonX12.Enabled =	};	};	List <string> rs_18 =</string>
true;	public	public	new List <string> { "}"</string>
	List <string> rw 5 =</string>	List <string> rs 4 =</string>	};
buttonX13.Enabled =	new List <string> {</string>	new List <string> {</string>	public
false;	"action" };	"\\" };	List <string> rs_19 =</string>
	}	public	new List <string> { "!"</string>
}		List <string> rs_5 =</string>	};
	public class	new List <string> { ";"</string>	public
}	ReservedWordsDelims	};	List <string> rs_20 =</string>
}	{	public	new List <string> { "&amp;"</string>
	public	List <string> rs_6 =</string>	};
Lexical Analyzer:	List <char> delim_1 =</char>	new List <string> { "("</string>	public
Dictionary.cs	new List <char> { ''</char>	};	List <string> rs_21 =</string>
using	};	public	new List <string> { "["</string>
System.Collections.Gen	public	List <string> rs_7 =</string>	};
eric;	List <char> delim_2 =</char>	new List <string> { ")"</string>	public
	new List <char> { '('</char>	};	List <string> rs_22 =</string>
namespace	};	public	new List <string> { "]"</string>
Lexical_Analyzer	public	List <string> rs_8 =</string>	};
{	List <char> delim_3 =</char>	new List <string> { "{"</string>	public
public class	new List <char> { '',</char>	};	List <string> rs_23 =</string>
Dictionary	'{'};	public	new List <string> { "."</string>
{	public	List <string> rs_9 =</string>	};
	List <char> delim_end =</char>	new List <string> {</string>	public
//RESERVED	new List <char> { '.',</char>	"++", "" };	List <string> rs_24 =</string>
SYMBOLS	' ', '\n', '\t','(' ,	public	new List <string> {</string>
public class	':', ',', '\'', '[',	List <string> rs_10 =</string>	"\"" };
ReservedWords	`]`, `?`, `#`, `\$`,	new List <string> {</string>	1
{	, %' , '\\',	"<=", ">=", "!=" };	}
public	,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,	public	111
List <string> rw_1 =</string>	) , , , , , , , , , , , , , , , , , , ,	List <string> rs_11 =</string>	public class
new List <string> {</string>	, , , , _ ,	new List <string> { ","</string>	ReservedSymbolsDelims
"company", "unit",	!, <, >, *, /	};	(
"digit", "response",	}; h1:-	public	public
"joe", "hold", "miss", "operation", "struct"	public List <char> delim 4 =</char>	List <string> rs_12 = new List<string> {</string></string>	List <char> del1 = new List<char> { '</char></char>
<pre>};</pre>	new List <char> { ';'</char>	"<", ">" ,"=" };	','(','=','a','b','c',
public	lew List\chai> ( , )   ;	public	'd','e','f','g','h','i
List <string> rw 2 =</string>	public	List <string> rs_13 =</string>	','j','k','l','m',
new List <string> Iw_Z =</string>	List <char> delim 5 =</char>	new List <string> 15_13 -</string>	, J , K , I , III ,
"PrimaryMission",	new List <char> { ':'</char>	}; //Negation	'n','o','p','q','r','s
"post", "capture",	};	public	','t','u','v','w','x',
"backup", "campaign",	,	List <string> rs_14 =</string>	y, z,
backap, campaign,	}	new List <string> { "^"</string>	<i>y</i> , <i>z</i> ,
"abort", "deploy",	j	}; //Power	'0', '1', '2', '3',
"inquire", "inorder",	public class	public	'4', '5', '6', '7',
"otherorder",	ReservedSymbols	List <string> rs_15 =</string>	'8', '9',
- ,	{	new List <string> { "#"</string>	, ,
"phase", "Swap",	public	}; //Address	'A', 'B', 'C', 'D', 'E', 'F
"Carry", "sqrt" };	List <string> rs_1 =</string>	public	','G','H','I','J','K',
public	new List <string> {</string>	List <string> rs_16 =</string>	'L', 'M',
List <string> rw_3 =</string>	"+", "-", "*", "%" ,	new List <string> { ":"</string>	
_	"/"};	};	'N','O','P','Q','R','S

','T','U','V','W','X',
'Y','Z'};

public
List<char> del2 = new
List<char> { ')','
','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',

'1', '2', '3', '4', '5', '6', '7', '8', '9',

'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'};

public
List<char> del3 = new
List<char> {
 '"', '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',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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'};

public
List<char> del4 = new
List<char> {
 'a','b','c','d','e','f
 ','g','h','i','j','k',
 '1','m',

'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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','(',

')', '"', '{', '}',
'@', '|', '&', '\_,
'&', '[', ']', '+',

'-', '\*', '/', '>',
'<', '=', '?'};
public
List<char> del5 = new
List<char> { '\n','
',')', 'a', 'b', 'c', 'd',

',')','a','b','c','d',
'e','f','g','h','i','j
','k','l','m',
'n','o','p','q','r','s

n, o, p, q, r, s
', 't', 'u', 'v', 'w', 'x',
'y', 'z',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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'};

public
List<char> de16 = new
List<char> {
 '(','''',')','#',
 'a','b','c','d','e','f
 ','g','h','i','j','k',
 '1','m',

'n','o','p','q','r','s
','t','u','v','w','x',
'y','z',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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'};

public List<char> del7 = new List<char> { ')',' ','{', ';','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',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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'};

public
List<char> del8 = new
List<char> {
 'a','b','c','d','e','f
 ','g','h','i','j','k',
 '1','m',

'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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','', '\n',

'"', '\'' };

public

List<char> del9 = new List<char> { ';','a','b','c','d','e ','f','g','h','i','j', 'k','1','m',

'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'A','B','C','D','E','F','G','H','I','J','K',

'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z'};

public
List<char> del10 = new
List<char> {
 ',''','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',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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'};

public
List<char> del11 = new
List<char> {
 'a','b','c','d','e','f
 ','g','h','i','j','k',
 '1','m',

'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',

'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',

'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','&','

'=','"','\','A','(','
)','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',

		List <char> { ';','</char>	
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',	'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',	','\n','a','b','c','d' ,'e','f','g','h','i',' j','k','l','m',	'A','B','C','D','E','F ','G','H','I','J','K', 'L','M',
'B','C','D','E','F','G ','H','I','J','K','L', 'M',	'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',	'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z'};
'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z',''};	'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z'};	'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',	<pre>public List<char> del21 = new List<char> {    ']','a','b','c','d','e</char></char></pre>
<pre>public List<char> del13 = new List<char> {</char></char></pre>	<pre>public List<char> del16 = new List<char> { '</char></char></pre>	'A','B','C','D','E','F ','G','H','I','J','K',	','f','g','h','i','j', 'k','l','m',
'a','b','c','d','e','f ','g','h','i','j','k', 'l','m',	','\n','a','b','c','d' ,'e','f','g','h','i',' j','k','l','m',	'L','M', 'N','O','P','Q','R','S	'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',
'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',	'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',	','T','U','Y','W','X', 'Y','Z'};  public List <char> del19 = new</char>	'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',	'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',	List <char> {     'a','b','c','d','e','f     ','g','h','i','j','k',     '1','m',</char>	'A','B','C','D','E','F ','G','H','I','J','K', 'L','M',
'A','B','C','D','E','F ','G','H','I','J','K', 'L','M',	'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',	'N','O','P','Q','R','S ','T','U','Y','W','X', 'Y','Z'};
'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z'};	'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z'};	'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',	<pre>public List<char> del22 = new List<char> { ' ','=','[',';',')','a',</char></char></pre>
<pre>public List<char> del14 = new List<char> {   '(','a','b','c','d','e</char></char></pre>	public List <char> del17 = new List<char> { ' ','(','a','b','c','d',</char></char>	'A','B','C','D','E','F ','G','H','I','J','K', 'L','M',	'b','c','d','e','f','g ','h','i','j','k','l', 'm',
','f','g','h','i','j', 'k','l','m',	'e','f','g','h','i','j ','k','l','m',	'N','O','P','Q','R','S ','T','U','V','W','X',	'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',
'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',	'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',	'Y','Z' };  // public  List <char> de121 = new  List<char> { 'O', '1',</char></char>	'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',	'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',	'2', '3', '4', '5', '6', '7', '8', '9' };  public  List <char> de120 = new</char>	'A','B','C','D','E','F ','G','H','I','J','K', 'L','M',
'A','B','C','D','E','F ','G','H','I','J','K', 'L','M',	'A','B','C','D','E','F ','G','H','I','J','K', 'L','M',	List <char> { ' ','(','a','b','c','d', 'e','f','g','h','i','j ','k','l','m',</char>	'N','O','P','Q','R','S ','T','U','Y','W','X', 'Y','Z'};
'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z' };	'N','O','P','Q','R','S ','T','U','V','W','X', 'Y','Z' };	'n','o','p','q','r','s ','t','u','v','w','x', 'y','z',	<pre>public List<char> de123 = new List<char> {     'a', 'b', 'c', 'd', 'e', 'f</char></char></pre>
<pre>public List<char> del15 = new List<char> {</char></char></pre>	/*public List <char> del18 = new List<char> {</char></char>	'0', '1', '2', '3',	a, b, c, d, e, f ','g','h','i','j','k', 'l','m',
'a','b','c','d','e','f ','g','h','i','j','k', '1','m',	'd','f','s' };*/ public List <char> del18 = new</char>	'4', '5', '6', '7', '8', '9',	'n','o','p','q','r','s

','t','u','v','w','x',	Li-+/-h\ [ '~' '0'	, , T, , 11, , V, , W, , V,	
	List <char> { '~', '0', '1', '2', '3', '4',</char>	','T','U','V','W','X', 'Y','Z' };	public
'y', 'z',			List <char> delim_lit =</char>
202 212 202 202	'5', '6', '7', '8',	public	new List <char>();</char>
'0', '1', '2', '3',	'9'};	List <char></char>	public
'4', '5', '6', '7',	}	delim_undscr = new	List <char> delim_12 =</char>
'8', '9',	public class	List <char> { '_' };</char>	new List <char>();</char>
	LiteralsDelims	public	public
'A','B','C','D','E','F	{	List <char> id = new</char>	List <char> delim_16 =</char>
','G','H','I','J','K',	/* public	List <char> {};</char>	new List <char> { ''</char>
'L','M',	List <char> delim_txt =</char>	}	};
	new List <char> { ' ',</char>	public class	}
'N','O','P','Q','R','S	';', ',', ')', '.' };	IdentifierDelims	
','T','U','V','W','X',	public	{	}
'Y','Z' };	List <char> delim_num =</char>	public	}
public	new List <char> { '+',</char>	List <char> delim_end =</char>	
List <char> del24 = new</char>	'-', '*', '/', ',',	new List <char> {</char>	Lexical Analyzer:
List <char> { ')' };</char>	'&', ' ', ')', ';' };	'.','\n', '=',	Initializer.cs
	*/	'\t','(', ':', ',',	
	public	'\'', '[', ']', '?',	using System;
public	List <char> delim txt =</char>	`#`, `\$`, `\ <sub>%</sub> `, `\\`,	using
List <char> delim end =</char>	new List <char> { ' ',</char>	., , , , , , , , , , , , , , , , , , ,	System. Collections. Gen
new List <char> { '.',</char>	'\n', ';', ',', ')',	')', '"', ';', '@',	eric;
	· · · · · · · · · · · · · · · · · · ·	, ^, , ~, , , , ,	using System.Ling;
' ', '\n', '\t','(' , ':', ',', '[', ']',	public	`!`, `<`, `, <del>_</del> ,	daing bystem. Lind,
· , , , , , , , , , , , , , , , , , , ,	List <char> delim num =</char>	· , · , · , · , · , · , · , · , · , · ,	//Unused Libraries
	new List <char> { '+',</char>	/ <b>,</b> <sup>+</sup> <b>,</b> <sup>-</sup> <b>,</b> ∫ <b>,</b>	//using System. Text;
`\\`,	new List char/ ( +,	,	- ·
, , , , , , , , , , , , , , , , , , , ,	`-', `*', `/', , .', ', `\n', `;', `&',	//OTHER	//using
), ;, w, , ,~, ,`, , , , ,	, \n , ; , & ,		System. Threading. Tasks
, , , _ , , ,	` `, `)`, `,`, `&`,	DELIMITERS	;
ζ, , , ,	']'};	public class	//using
1.1.1.1.1	}	Delims	System. Collections. Gen
'>', '*', '/',		{	eric;
'&','a','b','c','d','e	//Identifier	public	
','f','g','h','i','j',	public class	List <char> delim_zero</char>	namespace
'k','1','m',	Identifier	= new List <char> { '0'</char>	Lexical_Analyzer
	{	};	{
'n','o','p','q','r','s	public	public	public class
','t','u','v','w','x',	List <char> delim_digit</char>	List <char></char>	Initializer
'y', 'z',	= new List <char> {</char>	delim_lowlet = new	{
	'1', '2', '3', '4',	List <char> {</char>	public int
'0', '1', '2', '3',	'5', '6', '7', '8',	'a','b','c','d','e','f	tokens = 0;
'4', '5', '6', '7',	'9', '0' };	','g','h','i','j','k',	
'8', '9',	public	'1','m',	//INITIALIZATION
	List <char></char>		public
'A', 'B', 'C', 'D', 'E', 'F	delim lowlet = new	'n','o','p','q','r','s	LexicalAnalyzer
','G','H','I','J','K',	List <char> {</char>	','t','u','v','w','x',	InitializeAnalyzer(str
'L','M',	'a','b','c','d','e','f	'y', 'z' };	ing txt,
	','g','h','i','j','k',	public	LexicalAnalyzer lex)
'N','O','P','Q','R','S	'1','m',	List <char></char>	{
', 'T', 'U', 'V', 'W', 'X',	- , ,	delim mathOp = new	Boolean
'Y', 'Z' };	'n','o','p','q','r','s	List <char> { '+', '-',</char>	hastoken = false;
1, 2, ,	', 't', 'u', 'v', 'w', 'x',	'*', '/' };	Tokens t =
	'y', 'z' };	public	new Tokens();
}	public	List <char></char>	//txt =
J	List <char></char>	delim undscr = new	txt. TrimStart();
	delim caplet = new	List <char> { '_' };</char>	cat. IIImotait(/,
//LITERALS	List <char></char>	public	lex.token.Clear();
	{'A', 'B', 'C', 'D', 'E', '	*	rea. token. Crear (),
public class		List <char></char>	low invol: 1 = 0.
Literals	F','G','H','I','J','K'	delim_identifier = new	lex.invalid = 0;
111:	,'L','M',	List <char>();</char>	lex. valid
public	יאי ירי יהי ירי יהי יר		= 0;
List <char> nums = new</char>	'N','O','P','Q','R','S		

while (txt		if	
!= "")	break;	(ctr + 1 <=	}
:- )	bieak,	lex. linetokens[i])	}
if	}	fex. Timetokens[1])	J
(hastoken =	}	(	else ifEnd = true;
lex.GetTokenLines(txt,	if	lex.token[ctr].setLine	}
tokens))	(lex.ctr == 0 &&	s(i + 1);	if
{	txt.Length != 1)	- ,</td <td>(ifEnd != true)</td>	(ifEnd != true)
`	<pre>lex.ctr = GetCtr(txt);</pre>	break;	,
txt = txt. Remove(0,	` ,	}	ctr++;
lex.ctr);	else if (lex.ctr == 0	}	}
	&& txt.Length == 1)	}	
tokens;	lex.ctr = 1;	return	if
}		lex;	(!(txt.Length >= ctr))
else	else if (lex.ctr >=	}	ctr;
if (hastoken =	txt.Length) lex.ctr =		return
lex.GetReservedWords(t	txt.Length;	//GET CTRS	ctr;
xt))		private int	}
	t.setTokens("INVALID")	GetCtr(string txt)	private int
txt = txt.Remove(0,	;	{	GetCtr(string txt, int
lex.ctr);			ctr)
else	t.setLexemes(txt.Subst	Dictionary.ReservedWor	{
if (hastoken =	ring(0, lex.ctr));	dsDelims rwd = new	Boolean
lex.GetReservedSymbols		Dictionary.ReservedWor	notEnd = true;
(txt))	<pre>lex. token. Add(t);</pre>	dsDelims();	List <char></char>
		Dictionary	delims = new
txt = txt.Remove(0,	txt = txt.Remove(0,	td = new Dictionary();	List <char>{ '"', '\\',</char>
lex.ctr);	lex.ctr);		'\n' };
else		Boolean	while
if (hastoken =	}	ifEnd = false;	(notEnd && (txt.Length
<pre>lex.GetLiterals(txt))</pre>		int ctr =	-1) >= ctr
	tokens++;	0;	{
txt = txt.Remove(0,	tokens++; //txt		{
<pre>txt = txt.Remove(0, lex.ctr);</pre>	·		{ foreach (char c in
	//txt	0;	foreach (char c in delims)
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); }</pre>	0; foreach	delims)
<pre>lex.ctr);</pre>	//txt = txt.TrimStart(); } lex.linetokens.Add(tok	0; foreach (var item in rwd.delim_end) {	delims) {
<pre>lex.ctr);</pre>	//txt = txt.TrimStart(); } lex.linetokens.Add(tokens);	0; foreach (var item in rwd.delim_end) { if	<pre>delims)</pre>
<pre>lex.ctr);</pre>	//txt = txt.TrimStart(); } lex.linetokens.Add(tokens); lex =	0; foreach (var item in rwd.delim_end) {     if (txt.ElementAt(ctr -	delims) {
<pre>lex.ctr);</pre>	//txt = txt.TrimStart(); } lex.linetokens.Add(tokens);	0; foreach (var item in rwd.delim_end) { if	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);</pre>	<pre>foreach (var item in rwd. delim_end)  {          if (txt. ElementAt(ctr - 1) == item)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex); return</pre>	0; foreach (var item in rwd.delim_end) {     if (txt.ElementAt(ctr -	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex); return</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);     return lex; }</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);     return lex; }</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);     return lex; } private LexicalAnalyzer</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);     return lex; } private LexicalAnalyzer</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);      return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) {</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) { for (int</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);      return lex; }      private LexicalAnalyzer setLines(LexicalAnalyz er lex)     {         for (int ctr = 0; ctr </pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) { for (int ctr = 0; ctr &lt; lex.token.Count;</pre>	<pre>foreach (var item in rwd. delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens);     lex = setLines(lex);      return lex; }      private LexicalAnalyzer setLines(LexicalAnalyz er lex)     {         for (int ctr = 0; ctr </pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) { for (int ctr = 0; ctr &lt; lex.token.Count; ctr++) {</pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) { for (int ctr = 0; ctr &lt; lex.token.Count; ctr++) { for</pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) {     for (int ctr = 0; ctr &lt; lex.token.Count; ctr++)     {         for (int i = 0; i </pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) { for (int ctr = 0; ctr &lt; lex.token.Count; ctr++) { for</pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) {     for (int ctr = 0; ctr &lt; lex.token.Count; ctr++)     for (int i = 0; i &lt; lex.linetokens.Count;</pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) {     for (int ctr = 0; ctr &lt; lex.token.Count; ctr++)     for (int i = 0; i &lt; lex.linetokens.Count;</pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>
<pre>lex.ctr);</pre>	<pre>//txt = txt.TrimStart(); } lex.linetokens.Add(tok ens); lex = setLines(lex);  return lex; }  private LexicalAnalyzer setLines(LexicalAnalyz er lex) {     for (int ctr = 0; ctr &lt; lex.token.Count; ctr++)     for (int i = 0; i &lt; lex.linetokens.Count;</pre>	<pre>foreach (var item in rwd.delim_end)</pre>	<pre>delims)</pre>

```
ctr++;
                                                                            int
                                                                tempctr = 0, limit =
                                                                                                delims = rwd. delim 4;
                                            Boolean
            return
                                hastokenlines = false;
ctr;
                                            if
                                                                                                break;
                                (txt.ElementAt(0) ==
                                '\n')
                                                                 (txt.Length != 1)
                                                                                                case 4:
                                                                                while
                                                                                                words = rw.rw_5;
Lexical Analyzer:
                                lines++;
                                                                ((txt.Length - 1))
Lexical Analyzer.cs
                                                                tempctr &&
                                                                                                delims = rwd.delim_5;
                                linetokens.Add(tokenct
                                                                !isEnd(txt[tempctr +
                                                                1], rwd))
using System;
                                                                                                break;
                                r):
using
System. Collections. Gen
                                hastokenlines = true;
eric:
                                                                                                //Check Reserved Words
                                                ctr =
                                                                tempctr++;
using System. Ling;
                                1:
                                                                                                 foreach (char c in
//Unused Libraries
                                            else if
                                                                tempctr++;
//using System.Text;
                                (txt.ElementAt(0) == '
//using
System. Threading. Tasks
                                                                            for (int i
                                                                                                limit = words.Count -
                                                                = 0; i < 5; i++)
using TokenLibrary;
                                hastokenlines = true;
                                                                                                 temp = new
                                                ctr =
                                                                                 ctr =
                                                                                                List<string>();
namespace
                                                                0:
Lexical_Analyzer
                                                                                words
                                                                = new List<String>();
                                                                                                found = false;
                                            return
    public class
                                                                                 delims
                                hastokenlines;
Tokens: TokensClass
                                                                = new List<char>();
                                                                                                foreach (string w in
   {
                                                                                 found
                                                                                                words)
                                        //GET TOKENS
                                                                = true:
                                                                                 switch
                                        public Boolean
                                GetReservedWords(strin
                                                                (i)
                                                                                                //IF NOT OUT OF RANGE
   public class
LexicalAnalyzer
                                g txt)
                                                                                                 if ((w.Length - 1) >=
        public
                                                                case 0:
                                                                                                ctr)
List<Tokens> token =
                                Dictionary.ReservedWor
new List<Tokens>();
                                dsDelims rwd = new
                                                                words = rw.rw_1;
       public
                                Dictionary. ReservedWor
List<int> linetokens =
                                                                delims = rwd.delim_1;
                                                                                                //IF LETTER MATCHED
                                dsDelims();
new List<int>();
        Boolean
                                                                                                if (c ==
                                Dictionary. ReservedWor
                                                                break:
isReserved = false;
                                ds rw = new
                                                                                                w.ElementAt(ctr))
        public int
                                Dictionary. ReservedWor
                                                                case 1:
invalid = 0;
                                ds();
       public int
                                            Tokens t =
                                                                words = rw.rw 2;
valid = 0;
                                new Tokens();
                                                                                                found = true;
        public int ctr
                                                                delims = rwd. delim 2;
                                                                                                //CHECK SIZE OF WORD
        public byte
                                List<String> words;
                                                                break;
                                                                                                AND INPUT
state = 0;
                                            List<char>
        public int
                                delims;
                                                                case 2:
                                                                                                if (w.Length ==
lines = 0;
                                                                                                tempctr)
        public int
                                List<String> temp;
                                                                words = rw.rw_3;
idnum = 1:
                                            Boolean
        Dictionary td
                                found = false,
                                                                delims = rwd.delim 3;
= new Dictionary();
                                hastoken = false,
                                                                                                //CHECK DELIMITER
                                exitfor = false, ifEnd
                                                                hreak.
        public Boolean
                                = false, nodelim =
                                                                                                if ((tempctr - 1) ==
GetTokenLines(string
                                true;
                                                                case 3:
                                                                                                ctr)
txt, int tokenctr)
                                                                words = rw.rw_4;
```

```
if
foreach (char delim in
                                t.setLexemes(w);
                                                                 (hastoken)
                                                                                                 else ifEnd = true;
delims)
                                t. setAttributes(w);
                                                                                                                      if
{
                                                                 exitfor = true;
                                                                                                 (ifEnd != true)
                                token. Add(t);
//IF NOT OUT OF RANGE
                                                                break:
                                                                                                 ctr++;
                                invalid++;
if ((txt.Length - 1) >
                                                                                 if
ctr)
                                                                 (exitfor)
                                                                                                              if
                                                                                                 (!(txt.Length >= ctr))
                                else if (nodelim)
                                                                                                 ctr--;
//IF FOUND DELIMITER
                                                                 exitfor = false;
                                                                                                              return
if (txt[ctr + 1] ==
                                                                                                 hastoken;
                                                                break:
delim)
                                hastoken = true;
                                                                                                            public
                                found = true;
                                                                                                 Boolean
                                                                                                 GetReservedSymbols(str
hastoken = true;
                                t. setTokens("INVALID")
                                                                 NOTHING FOUND
                                                                                                 ing txt)
                                                                             if (found
                                                                 == false)
nodelim = false;
                                t.setLexemes(w);
                                                                                                 Dictionary td = new
t.setTokens(w);
                                                                                                 Dictionary();
                                t. setAttributes(w);
                                                                hastoken = false;
t.setLexemes(w);
                                                                                                 Dictionary.ReservedSym
                                token. Add(t);
                                                                 foreach (var item in
                                                                                                 bols rs = new
t.setAttributes(w):
                                                                 rwd.delim end)
                                                                                                 Dictionary. ReservedSym
                                invalid++:
                                                                                                 bols();
token. Add(t):
                                                                                     if
                                break;
                                                                 (txt.ElementAt(ctr -
                                                                                                 Dictionary.ReservedSym
valid++;
                                                                 1) == item
                                                                                                 bolsDelims rsd = new
                                                                                                 Dictionary. ReservedSym
break;
                                                                 ifEnd = true;
                                                                                                 bolsDelims();
                                if (hastoken)
                                                                                                               Boolean
                                                                                 while
                                                                                                 found = false,
                                                                 (ifEnd != true)
                                                                                                 hastoken = false,
                                                                                                 exitfor = false;
                                break:
                                                                 foreach (var item in
else if (w ==
                                                                                                 List<String> words;
words[limit] &&
                                                                 rwd.delim end)
hastoken == false) {
                                                                                                 List<char> delims;
found = false; }
                                                                 if ((txt.Length) >
                                                                                                 List<String> temp;
                                else temp. Add(w);
                                                                 ctr)
                                                                                                               int
                                                                                                 tempctr = 0, limit =
                                                                                                 0, sctr= 0;
if (hastoken == false)
                                                                 if (txt.ElementAt(ctr)
                                                                                                                if
                                                                 == item)
                                                                                                  (txt.Length != 1)
                                                                                                 while ((txt.Length -
hastoken = true;
                                                                                                 1) > tempctr &&
                                                                 ifEnd = true;
nodelim = false;
                                ctr++;
                                                                                                 !isEnd(txt[tempctr +
                                                                break;
                                                                                                 1], rsd))
found = true:
                                words = temp;
                                                     if
t. setTokens("NODELIM")
                                (found == false)
                                                                                                 tempctr++;
                                break;
```

```
words = rs.rs 6;
                                                                  case 13:
                                                                                                   //CHECK SIZE OF WORD
tempctr++;
                                                                                                   AND INPUT
                                 delims = rsd. del6;
                                                                  words = rs.rs 14;
              for (int
                                                                                                   if (w.Length ==
i = 0; i < 16; i++)
                                 break:
                                                                  delims = rsd. del14;
                                                                                                   tempctr)
                                 case 6:
                   \operatorname{sctr}
                                                                  break;
= 0;
                                                                                                   //CHECK DELIMITER
                                                                  case 14:
                                 words = rs.rs_7;
words = new
List<String>();
                                 delims = rsd. del7;
                                                                  words = rs.rs_15;
                                                                                                   if ((tempctr - 1) ==
                                                                                                   sctr)
delims = new
                                 break:
                                                                  delims = rsd. del15;
List<char>();
                                 case 7:
                                                                  break;
found = true;
                                                                                                   foreach (char delim in
                                 words = rs.rs 8;
                                                                  case 15:
                                                                                                   delims)
switch (i)
                                 delims = rsd. del8;
                                                                  words = rs.rs_16;
case 0:
                                 break;
                                                                  delims = rsd.del16;
                                                                                                   //IF NOT OUT OF RANGE
words = rs.rs 1;
                                 case 8:
                                                                  break:
                                                                                                   if ((txt.Length - 1) >
                                                                                                   sctr)
delims = rsd.del1;
                                 words = rs.rs_9;
                                 delims = rsd. del9:
break:
                                                                  //Check Reserved
                                                                  Symbols
                                                                                                   if (txt[sctr + 1] ==
case 1:
                                 break:
                                                                                                   delim)
                                                                  foreach (char c in
words = rs.rs 2;
                                 case 9:
                                                                                                   //IF FOUND DELIMITER
                                                                  txt)
delims = rsd. del2;
                                 words = rs.rs 10;
                                                                  limit = words.Count -
break;
                                 delims = rsd. del10;
                                                                  1;
                                                                                                   found = true;
                                 break;
                                                                  temp = new
                                                                                                   hastoken = true;
case 2:
                                                                  List<string>();
words = rs.rs_3;
                                 case 10:
                                                                                                   tokens. Add (w);
                                                                  found = false:
delims = rsd.del3:
                                                                                                   lexemes. Add(w):
                                 words = rs.rs 11;
                                                                  foreach (string w in
                                 delims = rsd. del11;
                                                                                                   valid++;
break;
                                                                  words)
                                                                                                   break;
case 3:
                                 break;
words = rs.rs 4;
                                 case 11:
                                                                  //IF NOT OUT OF RANGE
                                                                  if ((w.Length - 1) >=
delims = rsd. del4;
                                 words = rs.rs_12;
                                                                  sctr)
break;
                                 delims = rsd.del12;
                                                                                                   else if (w ==
                                                                                                   words[limit] &&
case 4:
                                 break;
                                                                  if (c ==
                                                                                                   hastoken == false)
words = rs.rs 5;
                                 case 12:
                                                                  w.ElementAt(sctr))
                                                                                                   found = false;
delims = rsd. del5:
                                 words = rs.rs 13;
break;
                                 delims = rsd. del13;
                                                                  found = true;
                                                                                                   if (hastoken) break;
case 5:
                                 break;
```

```
Boolean
                                found = false,
                                                                 delims = rsd. del2;
                                                                                                 words = rs.rs 10;
                                hastoken = false,
else temp. Add(w);
                                exitfor = false;
                                                                 break;
                                                                                                 delims = rsd. del10;
                                            Tokens t =
                                                                 case 2:
                                                                                                 break;
                                new Tokens();
                                             //rsd =
                                                                 words = rs.rs_3;
                                                                                                 case 10:
                                td.AddRange(rsd);
                                                                 delims = rsd. del3;
                                                                                                 words = rs.rs_11;
                                List<String> words;
                                            List<char>
                                                                                                 delims = rsd. del11;
                                                                 break;
                                delims:
                                                                 case 3:
sctr++;
                                                                                                 break;
                                List<String> temp;
                                                                 words = rs.rs 4;
                                                                                                 case 11:
words = temp;
                                            int
                                tempctr = 0, limit =
if (found == false)
                                0, sctr = 0;
                                                                 delims = rsd. del4;
                                                                                                 words = rs.rs 12;
break;
                                            if
                                                                 break;
                                                                                                 delims = rsd. del12;
if (hastoken)
                                (txt.Length != 1)
                                                                 case 4:
                                                                                                 break;
                                                 while
                                ((txt.Length - 1))
                                                                 words = rs.rs 5;
                                                                                                 case 12:
exitfor = true;
                                tempctr &&
                                !isEnd(txt[tempctr +
                                                                 delims = rsd.del5;
                                                                                                 words = rs.rs_13;
break;
                                1], rsd))
                                                                 break:
                                                                                                 delims = rsd. del13:
                                tempctr++;
                                                                 case 5:
                                                                                                 break:
                  if
(exitfor)
                                                                 words = rs.rs 6;
                                                                                                 case 13:
                                tempctr++;
                                                                 delims = rsd. del6;
                                                                                                 words = rs.rs 14;
exitfor = false;
                                             for (int i
                                                                 break;
                                                                                                 delims = rsd. del14;
break;
                                = 0; i < 24; i++)
                                                                 case 6:
                                                                                                 break;
                                                 sctr =
              if
                                                                 words = rs.rs_7;
                                                                                                 case 14:
(hastoken) ctr = sctr;
                                                 words
                                = new List < String > ();
                                                                 delims = rsd. del7;
                                                                                                 words = rs.rs 15;
return hastoken;
                                                 delims
         } */
                                = new List<char>();
                                                                 break;
                                                                                                 delims = rsd. del15;
                                                 found
        public Boolean
                                = true;
                                                                 case 7:
                                                                                                 break;
GetReservedSymbols(str
                                                 switch
ing txt)
                                (i)
                                                                 words = rs.rs 8;
                                                                                                 case 15:
            Dictionary
                                                                 delims = rsd. del8;
                                                                                                 words = rs.rs_16;
td = new Dictionary();
                                case 0:
                                                                 break;
                                                                                                 delims = rsd. del16;
Dictionary.ReservedSym
                                words = rs.rs_1;
bols rs = new
                                                                 case 8:
                                                                                                 break;
                                delims = rsd.del1;
Dictionary.ReservedSym
bols();
                                                                 words = rs.rs_9;
                                                                                                 case 16:
                                break;
                                                                                                 words = rs.rs_17;
                                                                 delims = rsd. del9;
Dictionary. ReservedSym
                                case 1:
bolsDelims rsd = new
Dictionary. ReservedSym
                                                                 break;
                                                                                                 delims = rsd. del17;
bolsDelims();
                                words = rs.rs 2;
                                                                 case 9:
                                                                                                 break;
```

```
{
case 17:
                                limit = words.Count -
                                                                hastoken = true;
                                                                                                exitfor = false;
words = rs.rs 18;
                                                                t = new Tokens();
                                temp = new
                                                                                                break;
delims = rsd. del18:
                               List<string>();
                                                                t.setTokens(w);
                                found = false;
break;
                                                                t.setLexemes(w);
                                                                                                             if
                                                                                                (hastoken) ctr = sctr;
case 18:
                                foreach (string w in
                                                                t. setAttributes(w);
                                                                                                            return
                                words)
                                                                                                hastoken;
words = rs.rs_19;
                                                                token. Add(t);
                                //IF NOT OUT OF RANGE
                                                                                                        /* public
delims = rsd. del19;
                                                                valid++;
                                                                                                Boolean
break;
                                if ((w.Length - 1) >=
                                                                break;
                                                                                                GetLiterals(string
                                sctr)
                                                                                                txt)
case 19:
words = rs.rs_20;
                                                                                                Dictionary.LiteralsDel
                                if (c ==
                                                                                                ims 1d = new
delims = rsd.del20;
                                w. ElementAt(sctr))
                                                                                                Dictionary.LiteralsDel
                                                                else if (w ==
                                                                                                ims();
                                                                words[limit] &&
break;
                                                                hastoken == false)
                                                                                                Dictionary.Literals 1
                                                                found = false;
case 20:
                                found = true;
                                                                                                = new
                                                                                                Dictionary.Literals();
                                //CHECK SIZE OF WORD
words = rs.rs 21;
                                                                                                           List<char>
                                AND INPUT
                                                                                                delims = new
delims = rsd. del21:
                                                                                                List<char>();
                                if (w.Length ==
                                                                if (hastoken) break:
                                                                                                            Boolean
break;
                                tempctr)
                                                                                                hastoken = false,
                                                                                                validtxt = false;
case 21:
                                                                                                            string
                                                                                                literal = "";
                                //CHECK DELIMITER
words = rs.rs_22;
                                                                else temp. Add(w);
                                                                                                            state = 0;
                                                                                                            int 1ctr =
delims = rsd.del22;
                                if ((tempctr - 1) ==
                                                                                                0:
                                sctr)
break;
                                                                                                            if
                                                                                                (txt.ElementAt(lctr)
case 22:
                                                                                                == ',"')
                                foreach (char delim in
                                                                                    }
                                                                                                                state
words = rs.rs_23;
                                delims)
                                                                                                = 1;
                                                                sctr++;
                                                                                                            else if
delims = rsd. del23;
                                                                                                (txt.ElementAt(lctr)
                                                                words = temp;
                                                                                                == '\'')
                                //IF NOT OUT OF RANGE
break;
                                                                                                                state
                                                                (found = false)
                                                                                                = 2:
                                if ((txt.Length - 1) >
case 23:
                                                                break;
                                                                                                            else
                                                                                                            {
                                sctr)
                                                                                    if
words = rs.rs_24;
                                                                (hastoken)
                                                                                                foreach (char num in
delims = rsd. del24;
                                                                                                1. nums)
                                //IF FOUND DELIMITER
                                                                exitfor = true;
hreak.
                                                                                                                    if
                                if (txt[sctr + 1] ==
                                                                break;
                                                                                                (txt.ElementAt(1ctr)
                                delim)
                                                                                    }
                                                                                                == num)
foreach (char c in
                                                                                                state = 3;
txt)
                                                                                if
                                found = true;
                                                                (exitfor)
```

```
if (state
                                if ((txt.Length - 1)
! = 0)
                                >= (1ctr + 1))
                                                                                                if ((txt.Length - 1)
                                if (txt[lctr + 1] ==
                                                                while ((txt.Length -
                                                                                                >= (1ctr + 1) &&
                switch
                                                                                                txt[1ctr + 1] == '\'')
                                                                1) > 1ctr &&
(state)
                                                                !(txt[1ctr + 1] ==
                                                                '\'') && !(txt[lctr +
                                                                1] == '\n'))
case 1: case 2:
                                                                                                lctr++;
                                hastoken = true;
delims = 1d.delim_txt;
                                break:
                                                                                                foreach (char c in
//String Literal
                                                                literal +=
                                                                                                delims)
Analyzer
                                                                txt[lctr].ToString();
if (state == 1)
                                                                lctr++;
                                                                                                if ((txt.Length - 1)
                                                                                                >= (1ctr + 1))
                                if (hastoken &&
                                                                if (1ctr >= 3)
if (txt.Length != 1)
                                                                                                if (txt[lctr + 1] ==
                                validtxt)
while ((txt.Length -
                                                                hastoken = false;
1) > 1ctr &&
                                valid++;
                                                                                                hastoken = true;
!(txt[lctr + 1] ==
                                                                ctr = 1ctr + 2;
"") && !(txt[lctr +
                                tokens.Add("Stringlit"
                                                                                                break;
1] == '\n'))
                                                                if (ctr > txt.Length)
{
                                lexemes. Add (txt. Substr
                                                                ctr = txt.Length;
                                ing(0, (1ctr + 1)));
literal +=
txt[lctr]. ToString();
                                ctr = 1ctr + 1;
                                                                else
                                                                                                if (hastoken)
lctr++;
                                else if (!validtxt)
                                                                                                {
                                                                if ((txt[1] == '\\' &&
if ((txt.Length - 1)
                                                                1ctr == 2) || (1ctr ==
                                                                                                valid++;
== 1ctr && (txt[1ctr]
                                                                1 && txt[1] != '\\')
!= '"'))
                                ctr = 1ctr + 2:
                                                                | | 1ctr == 0 
                                                                                                tokens.Add("Charlit");
hastoken = false;
                                hastoken = false;
                                                                validtxt = true;
                                                                                                lexemes. Add(txt. Substr
                                                                                                ing(0, (1ctr + 1)));
else
                                                                else
                                                                                                ctr = 1ctr + 1;
if (!(1ctr == 1 \&\&
                                                                validtxt = false;
txt[lctr] = ' \')
                                                                                                else
                                                                hastoken = false;
                                                                ctr = 1ctr + 2;
validtxt = true;
                                //Character Literal
                                                                if (ctr > txt.Length)
                                                                                                ctr = 1ctr + 1;
lctr++;
                                Analyzer
                                                                ctr = txt.Length;
                                                                                                if (ctr > txt.Length)
foreach (char c in
                                else
delims)
                                                                                                ctr = 1ctr;
{
                                                                if (validtxt)
                                if (txt.Length != 1)
```

```
lctr++;
                                                                 foreach (char n in
                                                                                                  if ((txt.Length - 1))
                                                                                                  1ctr)
                                                                 num)
                                                                                                  if (txt.ElementAt(lctr
                                                                                                  + 1) == delim
                                                                 if (txt.ElementAt(1ctr
                                                                 + 2) == n
                                                                 isDouble = true;
                                                                                                  hastoken = true;
break;
                                if (hasnum)
                                                                                                  break;
case 3:
                                while (isNumNext)
Dictionary. Identifier
id = new
Dictionary. Identifier (
                                                                 //Double Literal
                                                                 Analyzer
                                                                                                  if (hastoken)
                                 isNumNext = false;
delims = 1d. delim_num;
                                                                 if (isDouble)
                                 foreach (char n in
Boolean isNumNext =
                                                                                                  valid++;
true, hasnum = true,
hasid = false;
                                                                 1ctr++;
                                                                                                  tokens.Add("Declit");
List<char> num = new
                                if ((txt.Length - 1) >
                                                                 isNumNext = true;
List<char> { '0', '1',
                                                                                                  lexemes.Add(txt.Substr
                                 lctr)
'2', '3', '4', '5', '6', '7', '8', '9' };
                                                                 while (isNumNext)
                                                                                                  ing(0, (1ctr + 1)));
                                if (txt.ElementAt(lctr
                                 + 1) == n
id. id. AddRange (id. deli
                                                                 isNumNext = false;
                                                                                                  else
m caplet);
                                 lctr++;
                                                                 foreach (char n in
id.id.AddRange(id.deli
                                                                 num)
                                                                                                  foreach (char c in
m_caplet);
                                 isNumNext = true;
                                                                                                  id.id)
//If Negative
                                                                 if ((txt.Length - 1))
                                                                                                  {
                                                                                                  if ((txt.Length - 1))
if
(txt.ElementAt(lctr)
                                                                 if (txt.ElementAt(1ctr
                                                                                                  1ctr)
== ',-')
                                                                 + 1) == n
                                                                                                  if (txt.ElementAt(lctr
                                //Double Literal
                                                                                                  + 1) == c
                                Analyzer
hasnum = false;
                                                                 lctr++;
                                Boolean isDouble =
foreach (char n in
                                false:
                                                                 isNumNext = true;
                                                                                                  hasid = true;
num)
                                if ((txt.Length - 1) >
                                1ctr)
if ((txt.Length - 1) >
                                if (txt.ElementAt(lctr
1ctr)
                                + 1) == '.')
if (txt.ElementAt(1ctr
+ 1) == n
                                                                 foreach (char delim in
                                                                                                  if (!hasid)
                                 if ((txt.Length - 1) >
                                                                 delims)
                                1ctr + 1
                                                                                                  ctr = 1ctr + 1;
hasnum = true;
```

```
//Integer Literal
                                if (!hasid)
                                                                foreach (char num in
                                                                                                 if (!(lctr == 1 &&
Analyzer
                                                                1. nums)
                                                                                                 txt[1ctr] = ' \')
                                ctr = 1ctr + 1;
else
                                                                                     if
                                                                (txt.ElementAt(lctr)
{
                                                                == num)
                                                                                                 validtxt = true;
foreach (char delim in
                                                                state = 4;
                                                                                                 lctr++;
delims)
                                break;
                                                                                                 foreach (char c in
                                                                                                 delims)
                                            return
                                                                             if (state
if (txt.ElementAt(lctr
                                hastoken;
                                                                ! = 0)
+ 1) == delim
                                                                                                 if ((txt.Length - 1)
                                                                                 switch
                                        public Boolean
                                                                (state)
                                                                                                 >= (1ctr + 1))
                                GetLiterals(string
hastoken = true;
                                txt)
                                                                                                 if (txt[lctr + 1] ==
                                                                case 1:
break;
                                Dictionary.LiteralsDel
                                                                case 2:
                                ims 1d = new
                                Dictionary. LiteralsDel
                                                                                                hastoken = true;
                                                                case 3:
                                ims();
                                                                delims = 1d.delim_txt;
                                                                                                break;
                                Dictionary.Literals 1
if (hastoken)
                                = new
                                                                //String Literal
                                Dictionary.Literals();
                                                                Analyzer
                                            Tokens t =
                                                                if (state == 1)
                                new Tokens():
valid++;
                                            List<char>
                                delims = new
tokens.Add("Numlit");
                                List<char>();
                                                                                                 if (hastoken &&
                                                                if (txt.Length != 1)
                                                                                                 validtxt)
                                            Boolean
lexemes.Add(txt.Substr
                                hastoken = false,
                                                                                                 {
ing(0, (1ctr + 1)));
                                validtxt = false;
                                            string
                                literal = "";
                                                                while ((txt.Length -
                                                                                                valid++;
                                            state = 0;
                                                                1) > 1ctr &&
                                            int 1ctr =
                                                                !(txt[1ctr + 1] ==
                                                                                                 t = new Tokens():
else
                                0:
                                                                "") && !(txt[lctr +
                                                                1] == '\n'))
                                                                                                 t.setTokens("Stringlit
                                            if
                                                                                                 ");
foreach (char c in
                                (txt.ElementAt(lctr)
id.id)
                                == ',"')
                                                                                                 t.setLexemes(txt.Subst
                                                state
                                                                literal +=
                                                                                                 ring(0, (1ctr + 1)));
                                = 1;
                                                                txt[lctr]. ToString();
                                                                                                 t.setAttributes("Strin
                                            else if
if (txt.ElementAt(lctr
                                (txt.ElementAt(lctr)
                                                                1ctr++;
                                                                                                 glit");
                                == '\',')
+ 1) == c
                                                state
                                                                                                 token.Add(t);
                                = 2;
                                            else if
                                                                if ((txt.Length - 1)
                                                                                                ctr = 1ctr + 1;
hasid = true;
                                (txt.ElementAt(lctr)
                                                                == 1ctr && (txt[1ctr]
                                                                != '"'))
                                == '@')
                                                state
                                                                                                else if (!validtxt)
                                = 3;
                                                                hastoken = false;
                                            else
                                                                else
                                                                                                ctr = 1ctr + 2;
```

```
hastoken = false;
                               validtxt = true;
                                                                t.setTokens("Charlit")
                                                                                                hastoken = false;
                                else
                                                                                                else
                                                                t. setLexemes (txt. Subst
                                                                ring(0, (lctr + 1)));
                                                                                                if (!(lctr == 1 &&
                                validtxt = false;
                                                                t.setAttributes("Charl
                                                                                                txt[lctr] = ' \setminus (')
                                                                it");
                               hastoken = false;
                                                                token.Add(t);
                                ctr = 1ctr + 2;
                                                                ctr = 1ctr + 1;
                                                                                                validtxt = true;
//Character Literal
                                if (ctr > txt.Length)
Analyzer
                                                                                                lctr++:
                                ctr = txt.Length;
else if(state == 2)
                                                                else
                                                                                                foreach (char c in
                                                                                                delims)
                                if (validtxt)
if (txt.Length != 1)
                                                                ctr = 1ctr + 1;
                                                                                                if ((txt.Length - 1)
                                                                                                >= (1ctr + 1))
                                if ((txt.Length - 1)
                                                                if (ctr > txt.Length)
                                >= (1ctr + 1) &&
                                                                                                if (txt[lctr + 1] ==
                                txt[lctr + 1] == '\')
while ((txt.Length -
                                                                ctr = 1ctr;
1) > 1ctr &&
!(txt[lctr + 1] ==
'\'') && !(txt[lctr +
1] = '(n')
                                lctr++:
                                                                                                hastoken = true:
{
                                foreach (char c in
                                                                                                break;
                                delims)
literal +=
txt[lctr].ToString();
                                if ((txt.Length - 1)
lctr++;
                                \geq (1ctr + 1)
                                                                else if(state == 3) {
                                if (txt[1ctr + 1] ==
                                                                if (txt.Length != 1)
                                                                                                if (hastoken &&
if (1ctr >= 3)
                                                                                                validtxt)
                                                                                                {
                                                                while ((txt.Length -
hastoken = false;
                                                                1) > 1ctr &&
                                                                                                valid++;
                               hastoken = true;
                                                                !(txt[1ctr + 1] ==
ctr = 1ctr + 2;
                                break;
                                                                '@') && !(txt[lctr +
                                                                                                t = new Tokens();
                                                                1] == '\n'))
if (ctr > txt. Length)
                                                                                                t. setTokens("comment")
ctr = txt.Length;
                                                                literal +=
                                                                                                t.setLexemes(txt.Subst
                                                                txt[lctr].ToString();
                                                                                                ring(0, (1ctr + 1)));
                                if (hastoken)
                                                                1ctr++;
                                                                                                t.setAttributes("comme
else
                                                                                                nt");
                                                                                                token. Add(t);
if ((txt[1] == ' \' \&\&
                                                                if ((txt.Length - 1)
                                valid++;
lctr == 2) || (lctr ==
                                                                == lctr && (txt[lctr]
                                                                                                ctr = 1ctr + 1;
1 && txt[1] != '\\')
                                t = new Tokens();
                                                                != '@'))
| | 1ctr == 0
```

```
else if (!validtxt)
                                if ((txt.Length - 1) >
                                                                                                  if ((txt.Length - 1))
                                 lctr)
                                                                                                  1ctr)
                                                                 }
                                if (txt.ElementAt(lctr
                                                                                                  if (txt.ElementAt(lctr
                                                                                                  + 1) == n
ctr = 1ctr + 2;
                                 + 1) == n
                                                                                                  {
hastoken = false;
                                hasnum = true;
                                                                                                  lctr++;
                                                                 //Double Literal
                                lctr++;
                                                                 Analyzer
                                                                                                  isNumNext = true;
                                                                 Boolean isDouble =
                                                                 false:
                                                                 if ((txt.Length - 1))
break;
                                                                 if (txt.ElementAt(lctr
                                                                 + 1) == '.')
                                 if (hasnum)
                                                                                                  foreach (char delim in
case 4:
                                                                                                  delims)
Dictionary. Identifier
id = new
                                while (isNumNext)
                                                                 if ((txt.Length - 1))
                                                                 1ctr + 1)
                                                                                                  if ((txt.Length - 1) >
Dictionary. Identifier (
                                                                                                  lctr)
                                                                 foreach (char n in
delims = 1d. delim num;
                                isNumNext = false:
                                                                 num)
                                                                                                  if (txt.ElementAt(1ctr
                                                                                                  + 1) == delim
Boolean isNumNext =
                                foreach (char n in
true, hasnum = true,
                                num)
hasid = false;
                                                                 if (txt.ElementAt(lctr
                                                                 + 2) == n
                                                                                                  hastoken = true;
List<char> num = new
List<char> { '0', '1',
                                if ((txt.Length - 1) >
                                                                 isDouble = true;
                                                                                                  break;
'2', '3', '4', '5', '6', '7', '8', '9' };
                                lctr)
                                if (txt.ElementAt(1ctr
                                 + 1) == n
id. id. AddRange (id. deli
m caplet);
                                                                 if (isDouble)
                                                                                                  if (hastoken)
id.id.AddRange(id.deli
                                storedval++;
m_caplet);
                                                                                                  {
                                 if (storedval <= 8)
int storedval = 0;
                                                                 lctr++;
                                                                                                  valid++;
//If Negative
                                                                 isNumNext = true;
                                                                                                  t = new Tokens();
                                 lctr++;
                                                                 while (isNumNext)
                                                                                                  t.setTokens("Declit");
(txt.ElementAt(lctr)
                                 isNumNext = true;
== ', ~', )
                                                                                                  t.setLexemes(txt.Subst
                                                                                                  ring(0, (lctr + 1)));
                                                                 isNumNext = false;
                                 else if(storedval > 8)
                                                                                                  t.setAttributes("Decli
hasnum = false;
                                                                 foreach (char n in
                                                                                                  t");
                                                                 num)
foreach (char n in
                                isNumNext = false;
                                                                                                  token. Add(t);
num)
                                hastoken = false;
                                                                                                  }
```

```
{
                                                                Dictionary. Identifier
                                                                                                 if ((txt.Length - 1) >
else
                                                                 id = new
                                                                                                 ictr)
                                valid++;
                                                                Dictionary. Identifier (
                                                                                                 if (txt.ElementAt(ictr
                                                                                                 + 1) == n
foreach (char c in
                                t = new Tokens();
                                                                Dictionary. IdentifierD
id. id)
                                t. setTokens("Numlit");
                                                                 elims delims = new
                                                                Dictionary. IdentifierD
                                                                                                 ictr++;
                                t.setLexemes(txt.Subst
                                                                 elims();
if ((txt.Length - 1) >
                                ring(0, (1ctr + 1)));
                                                                             Boolean
1ctr)
                                                                 hastoken = false,
                                                                                                 isvalID = true;
                                t.setAttributes("Numli
                                                                 valID = false, isvalID
if (txt.ElementAt(1ctr
                                t");
                                                                 = true;
+ 1) == c
                                token. Add(t);
                                                                                                                      if
                                                                 id.id.AddRange(id.deli
                                                                                                 (ictr > 17)
                                                                 m lowlet);
hasid = true;
                                                                                                 valID = false;
                                else
                                                                 id.id.AddRange(id.deli
                                                                 m_caplet);
                                                                                                                  if
                                                                 id. id. AddRange (id. deli
                                                                                                 (valID)
                                foreach (char c in
                                                                m undscr);
                                id.id)
                                                                 id.id.AddRange(id.deli
                                                                                                 foreach (char delim in
                                                                                                 delims.delim_end)
                                                                 m_digit);
if (!hasid)
                                if (txt.ElementAt(1ctr
                                                                             int ictr =
ctr = 1ctr + 1:
                                + 1) == c
                                                                 0:
                                                                                                 if ((txt.Length - 1))
                                                                                                 ictr)
                                                                             foreach
                                                                 (char c in id.id)
                                                                                                 if (txt.ElementAt(ictr
//Integer Literal
                                hasid = true;
                                                                                                 + 1) == delim
Analyzer
                                                                 (txt.ElementAt(ictr)
else
                                                                 == c)
                                                                                                 hastoken = true;
{
                                                                 valID = true;
                                                                                                 break;
foreach (char delim in
delims)
                                if (!hasid)
                                ctr = 1ctr + 1;
                                                                 id.id.AddRange(id.deli
//if
                                                                 m_digit);
(txt.ElementAt(lctr +
                                                                             if (valID)
                                                                                                 if(hastoken)
1) == delim
                                                                 //ictr++;
                                                                                                 valid++;
                                break;
hastoken = true;
                                                                 isvalID = true;
                                                                                                 tokens.Add("id");
                                            return
                                                                                 while
                                                                                                 lexemes.Add(txt.Substr
break;
                                                                 (isvalID)
                                hastoken;
                                                                                                 ing(0, (ictr + 1)));
                                                                 isvalID = false;
                                         /* public
                                                                                                                  ctr =
                                Boolean
                                                                 foreach (char n in
                                                                                                 ictr + 1;
                                GetIdentifiers(string
                                                                 id.id)
                                txt)
                                                                                                             return
if (hastoken)
                                                                                                 hastoken;
```

```
if
                                                                 id.id.AddRange(id.deli
                                result = true;
                                                                                                  (hastoken)
        //IS ENDS
                                                                 m_digit);
        public Boolean
                                 break;
                                                                              if (valID)
isEnd(char c,
                                                                                                  valid++;
Dictionary. ReservedWor
dsDelims rwd)
                                             return
                                                                 isvalID = true;
                                                                                                  = new Tokens();
                                 result;
                                                                                  while
            Boolean
                                                                 (isvalID)
                                                                                                  t.setTokens("id");
result = false;
                                                                                                  t.setLexemes(txt.Subst
            foreach
                                 } */
                                                                 isvalID = false;
                                                                                                  ring(0, (ictr + 1)));
(var item in
rwd.delim_end)
                                        public Boolean
                                 GetIdentifiers(string
                                                                 foreach (char n in
                                                                                                  t.setAttributes("ident
                                                                                                  ifier" + idnum);
                                 txt)
                                                                 id. id)
                if
(item == c)
                                                                                                  token. Add(t);
                                 Dictionary. Identifier
                                                                 if ((txt.Length - 1))
result = true;
                                id = new
                                                                                                  idnum++;
                                Dictionary. Identifier (
break;
                                                                 if (txt.ElementAt(ictr
                                                                 + 1) == n
                                                                                                                   ctr =
                                Dictionary. IdentifierD
                                                                                                  ictr + 1;
                                elims delims = new
            return
                                Dictionary. IdentifierD
result;
                                                                                                               return
                                 elims();
                                                                 ictr++;
                                                                                                  hastoken;
        public Boolean
                                             Boolean
isEnd(char c,
                                 hastoken = false,
                                                                 isvalID = true;
Dictionary. ReservedSym
                                 valID = false, isvalID
                                                                                                          public Boolean
bolsDelims rsd)
                                 = true:
                                                                                                  isEnd(char c.
                                                                                                  Dictionary. ReservedWor
                                             Tokens t =
                                 new Tokens();
                                                                                                  dsDelims rwd)
            Boolean
                                                                                      if
result = false;
                                                                 (ictr >= 18)
                                                                                                               Boolean
            foreach
(var item in
                                                                 valID = false;
                                                                                                  result = false;
rsd.delim_end)
                                 id.id.AddRange(id.deli
                                                                                                               foreach
                                 m_lowlet);
                                                                                                  (var item in
                                                                                                  rwd.delim_end)
                if
                                                                                  if
(item == c)
                                 id.id.AddRange(id.deli
                                                                 (valID)
                                 m_caplet);
                                                                                                                   if
                                                                                                  (item == c)
                                 id.id.AddRange(id.deli
result = true:
                                                                 foreach (char delim in
                                 m undscr);
                                                                 delims. delim end)
                                                                                                  result = true;
break:
                                 id.id.AddRange(id.deli
                                                                 if ((txt.Length - 1) >
                                 m digit);
                                                                                                  break;
            return
result;
                                             int ictr =
                                                                 if (txt.ElementAt(ictr
                                                                                                               return
                                                                 + 1) == delim
        public Boolean
                                                                                                  result;
isEnd(char c,
                                             foreach
List<char> 1d)
                                 (char c in id. id)
                                                                                                          public Boolean
                                                                                                  isEnd(char c,
                                                                                                  Dictionary.ReservedSym
            Boolean
                                                 if
                                                                 hastoken = true;
                                 (txt.ElementAt(ictr)
                                                                                                  bolsDelims rsd)
result = false;
            foreach
                                 == c)
                                                                 break;
(var item in 1d)
                                                                                                               Boolean
                                                                                                  result = false;
                                 valID = true;
                                                                                                               foreach
                if
(item == c)
                                                                                                  (var item in
                                                                                                  rsd.delim end)
```

if	* <exception< th=""><th></th><th>case (int)</th></exception<>		case (int)
(item == c)	cref='ParseException'>	EnterBreak((Token)	SyntaxConstants.DEFAUL
{	if the node analysis	node);	T:
`	* discovered	break;	
result = true;	errors	case (int)	EnterDefault((Token)
resurt true,	*/	SyntaxConstants.FOR N:	node);
1 1	<i>,</i>	Syntaxconstants, ron_n.	
break;	public override	D 2 1/(D 1 )	break;
}	void Enter (Node node)	EnterForN((Token)	case (int)
}	{	node);	SyntaxConstants.CLEAR:
return	switch	break;	
result;	(node. Id) {	case (int)	EnterClear((Token)
}	case (int)	SyntaxConstants. IF:	node);
public Boolean	SyntaxConstants.MAIN N		break;
isEnd(char c,	:	<pre>EnterIf((Token) node);</pre>	case (int)
List <char> 1d)</char>	•	break;	SyntaxConstants. SQROOT
Elst (chai) (a)	EnterMainN((Token)	case (int)	:
D 1			•
Boolean	node);	SyntaxConstants.ELSEIF	D ( ( T )
result = false;	break;	_N:	EnterSqroot((Token)
foreach	case (int)		node);
(var item in ld)	SyntaxConstants.PRINT_	EnterElseifN((Token)	break;
{	N:	node);	case (int)
if		break;	SyntaxConstants.PLUS:
(item == c)	EnterPrintN((Token)	case (int)	
	node);	SyntaxConstants.ELSE_N	EnterPlus((Token)
· ·	break;	:	node);
result = true;	case (int)	•	break;
resurt - true,		EnterElseN((Token)	
1 1 .	SyntaxConstants.SCAN_N		case (int)
break;	;	node);	SyntaxConstants.MINUS:
}		break;	
}	EnterScanN((Token)	case (int)	EnterMinus((Token)
return	node);	SyntaxConstants.DO:	node);
result;	break;		break;
}	case (int)	<pre>EnterDo((Token) node);</pre>	case (int)
}	SyntaxConstants.CONST_	break;	SyntaxConstants. TIMES:
}	N:	case (int)	-,
,	11.	SyntaxConstants.WHILE_	EnterTimes((Token)
Syntax Analyzer:	EnterConstN((Token)	N:	node);
-		11.	
SyntaxAnalyzer.cs	node);	F , WI 1 M / /T 1	break;
	break;	EnterWhileN((Token)	case (int)
using Core.Library;	case (int)	node);	SyntaxConstants.DIVIDE
	SyntaxConstants.RETURN	break;	:
/**	:	case (int)	
* <remarks>A class</remarks>		SyntaxConstants. VOID:	EnterDivide((Token)
providing callback	EnterReturn((Token)		node);
methods for the	node);	EnterVoid((Token)	break;
* parser.	break:	node);	case (int)
*/	case (int)	break;	SyntaxConstants. MODULU
public abstract class	SyntaxConstants. SWITCH	case (int)	S:
=			з.
SyntaxAnalyzer:	_N:	SyntaxConstants.GETCH:	
Analyzer {			EnterModulus((Token)
	EnterSwitchN((Token)	EnterGetch((Token)	node);
/**	node);	node);	break;
* <summary>Called</summary>	break;	break;	case (int)
when entering a parse	case (int)	case (int)	SyntaxConstants. EQUALS
tree node.	SyntaxConstants.CASE_N	SyntaxConstants.STRUCT	:
*	:	_N:	
* <param< td=""><td></td><td>_</td><td>EnterEquals((Token)</td></param<>		_	EnterEquals((Token)
name='node'>the node	EnterCaseN((Token)	EnterStructN((Token)	node);
being entered	node);	node);	break;
*	break;	break;	case (int)
	case (int)		SyntaxConstants.SEMIC:
	SyntaxConstants.BREAK:		

	case (int)		
<pre>EnterSemic((Token) node);</pre>	SyntaxConstants.D_E:	<pre>EnterCBracket((Token) node);</pre>	EnterHash((Token) node);
break;	<pre>EnterDE((Token) node);</pre>	break;	break;
case (int)	break;	case (int)	case (int)
SyntaxConstants.DOT:	case (int) SyntaxConstants.MOD E:	SyntaxConstants.GREATE R:	SyntaxConstants.NEGA:
EnterDot((Token)	5)		EnterNega((Token)
node);	EnterModE((Token)	EnterGreater((Token)	node);
break;	node);	node):	break;
case (int)	break;	break;	case (int)
SyntaxConstants. COMMA:	case (int)	case (int)	SyntaxConstants. INT:
Syntaxconstants. Comma.	SyntaxConstants. NEWLIN	SyntaxConstants.LESS:	SyntaxConstants. INT.
EnterComma((Token)	E:	Syntaxconstants. EESS.	EnterInt((Token)
node);	L.	EnterLess ((Token)	node);
break;	EnterNewline((Token)	node);	
case (int)	node);		break; case (int)
SyntaxConstants. AND:	hode), break;	break; case (int)	SyntaxConstants.CHAR:
SyntaxConstants. And.	case (int)	SyntaxConstants.GREATE	SylitaxColls tallts. Char.
EnterAnd((Token)	SyntaxConstants. N E:		EnterChar((Token)
node);	_	R_E:	node);
break;	<pre>EnterNE((Token) node);</pre>	EnterGreaterE((Token)	break;
case (int)	break;	node);	case (int)
SyntaxConstants.OR:	case (int)	break;	SyntaxConstants.FLOAT:
	SyntaxConstants.O_PARE	case (int)	
EnterOr((Token) node); break;	N:	SyntaxConstants.LESS_E:	EnterFloat((Token) node);
case (int)	EnterOParen((Token)		break;
SyntaxConstants.NOT:	node);	EnterLessE((Token)	case (int)
-,	break;	node);	SyntaxConstants. STRING
EnterNot((Token)	case (int)	break;	:
node);	SyntaxConstants. C PARE	case (int)	·
break;	N:	SyntaxConstants. S_OBRA	EnterString((Token)
case (int)	11.	CKET:	node);
SyntaxConstants. INCREM	EnterCParen((Token)	CRET.	break;
ENT:	node);	EnterSObracket((Token)	case (int)
ENI.			
[ T + ] + ( (T -	break;	node);	SyntaxConstants.BOOL_N
EnterIncrement((Token)	case (int)	break;	:
node);	SyntaxConstants.D_QUOT	case (int)	
break;	E:	SyntaxConstants. S_CBRA	EnterBoolN((Token)
case (int)	D (/m )	CKET:	node);
SyntaxConstants.DECREM	EnterDQuote((Token)	(/ )	break;
ENT:	node);	EnterSCbracket((Token)	case (int)
	break;	node);	SyntaxConstants. ID:
EnterDecrement((Token)	case (int)	break;	
node);	SyntaxConstants.COLON:	case (int)	<pre>EnterId((Token) node);</pre>
break;		SyntaxConstants.DOLLAR	break;
case (int)	EnterColon((Token)	:	case (int)
SyntaxConstants.P_E:	node);		SyntaxConstants.NUM:
	break;	EnterDollar((Token)	
<pre>EnterPE((Token) node);</pre>	case (int)	node);	EnterNum((Token)
break;	SyntaxConstants.O_BRAC	break;	node);
case (int)	KET:	case (int)	break;
SyntaxConstants.M E:		SyntaxConstants. POWER:	case (int)
_	EnterOBracket((Token)	•	SyntaxConstants.DECIMA
<pre>EnterME((Token) node);</pre>	node);	EnterPower ((Token)	L:
break;	break;	node);	
case (int)	case (int)	break;	EnterDecimal((Token)
SyntaxConstants. T E:	SyntaxConstants.C BRAC	case (int)	node);
S, Avancons vants. I_D.	KET:	SyntaxConstants. HASH:	break;
<pre>EnterTE((Token) node);</pre>		Symposis varies, intoit.	DI Can,
Enter ID ((Token) Houe),			

break;

/• · · \	(* .)	(* .)	, ,
case (int) SyntaxConstants.S CHAR	case (int) SyntaxConstants.S:	case (int) SyntaxConstants.PROD C	break; case (int)
:	Syntaxconstants. 5:	OMMENTS:	SyntaxConstants. PROD A
•	<pre>EnterS((Token) node);</pre>	OMMENTO.	DD ID:
EnterSChar((Token)	break;	EnterProdComments((Pro	
node);	case (int)	duction) node);	EnterProdAddId((Produc
break;	SyntaxConstants.ZERO:	break;	tion) node);
case (int)		case (int)	break;
SyntaxConstants.TEXT:	EnterZero((Token)	SyntaxConstants.PROD_N	case (int)
F. ( T. ( (T. 1 . )	node);	EGATE:	SyntaxConstants.PROD_N
EnterText((Token) node);	break; case (int)	EnterProdNegate((Produ	1:
break;	SyntaxConstants. TOCHAR	ction) node);	EnterProdN1((Productio
case (int)	:	break;	n) node);
SyntaxConstants.COM:	•	case (int)	break;
	EnterTochar((Token)	SyntaxConstants.PROD_D	case (int)
EnterCom((Token)	node);	ATATYPE:	SyntaxConstants.PROD_N
node);	break;		2:
break;	case (int)	EnterProdDatatype((Pro	
case (int)	SyntaxConstants.LENGTH	duction) node);	EnterProdN2((Productio
SyntaxConstants.YES:	F:	break;	n) node);
EnterYes((Token)	EnterLengthf((Token)	case (int) SyntaxConstants.PROD L	break; case (int)
node);	node);	ITERALS:	SyntaxConstants. PROD I
break;	break;	TIERRES.	NDEX:
case (int)	case (int)	EnterProdLiterals((Pro	
SyntaxConstants.NO:	SyntaxConstants.CONTAI	duction) node);	EnterProdIndex((Produc
	NS:	break;	tion) node);
<pre>EnterNo((Token) node);</pre>		case (int)	break;
break;	EnterContains((Token)	SyntaxConstants.PROD_L	case (int)
case (int)	node);	ITERALS2:	SyntaxConstants.PROD_S
SyntaxConstants.FUNCTN AME:	break; case (int)	E-4D	MATH:
AME:	SyntaxConstants.REVERS	EnterProdLiterals2((Pr oduction) node);	EnterProdSmath((Produc
EnterFunctname((Token)	E:	break;	tion) node);
node);	2.	case (int)	break;
break;	EnterReverse((Token)	SyntaxConstants.PROD G	case (int)
case (int)	node);	LOBAL_DEC:	SyntaxConstants.PROD_A
SyntaxConstants.STRUCT	break;		RRAY_AID:
NAME:	case (int)	EnterProdGlobalDec((Pr	
	SyntaxConstants.PROD_S	oduction) node);	EnterProdArrayAid((Pro
EnterStructname((Token	TART_PROGRAM:	break;	duction) node);
) node);	EnterProdStartProgram(	case (int) SyntaxConstants.PROD D	break; case (int)
break; case (int)	(Production) node);	ECLARE:	SyntaxConstants. PROD_E
SyntaxConstants. IDSTRU	break;	ECEME.	LEM CHOICE:
CT:	case (int)	EnterProdDeclare((Prod	BB.m_emereB.
	SyntaxConstants.PROD_P	uction) node);	EnterProdElemChoice((P
<pre>EnterIdstruct((Token)</pre>	ROGRAM:	break;	roduction) node);
node);		case (int)	break;
break;	EnterProdProgram((Prod	SyntaxConstants.PROD_D	case (int)
case (int)	uction) node);	ECLARE_CHOICE:	SyntaxConstants.PROD_E
SyntaxConstants.F:	break;	EntanDrodDro-1Ch-:-	LEMENT:
<pre>EnterF((Token) node);</pre>	case (int) SyntaxConstants.PROD C	EnterProdDeclareChoice ((Production) node);	EnterProdElement((Prod
break;	LEAR:	((Production) node); break;	uction) node);
case (int)	DDM.	case (int)	break;
SyntaxConstants.D:	EnterProdClear((Produc	SyntaxConstants.PROD I	case (int)
	tion) node);	NIT_CHOICE:	SyntaxConstants.PROD_A
<pre>EnterD((Token) node);</pre>	break;		DD_ELEM:
break;		EnterProdInitChoice((P	
		roduction) node):	

roduction) node);

<pre>EnterProdAddElem((Prod uction) node);</pre>	case (int) SyntaxConstants.PROD_M EM DEC:	break; case (int)	EnterProdAssignSym((Pr
	EM DEC:	C C DDOD C	
	_	SyntaxConstants.PROD_C	oduction) node);
break;		ONSTANT1:	break;
case (int)	EnterProdMemDec((Produ		case (int)
SyntaxConstants.PROD_M	ction) node);	EnterProdConstant1((Pr	SyntaxConstants.PROD_A
_ELEM:	break;	oduction) node);	SSIGN_VALUE:
P. A. D. IMPI (/D. L.	case (int)	break;	P. J. D. 1A . J. W. L. (/
<pre>EnterProdMElem((Produc tion) node);</pre>	SyntaxConstants.PROD_I NIT DEC:	case (int) SyntaxConstants.PROD M	EnterProdAssignValue(( Production) node);
break;	NII_DEC.	AIN:	break;
case (int)	EnterProdInitDec((Prod	MIN.	case (int)
SyntaxConstants.PROD M	uction) node);	EnterProdMain((Product	SyntaxConstants. PROD C
2 ELEM:	break;	ion) node);	ONVERT:
_	case (int)	break;	
EnterProdM2Elem((Produ	SyntaxConstants.PROD_I	case (int)	EnterProdConvert((Prod
ction) node);	NIT_DEC_CHOICE:	SyntaxConstants.PROD_A	uction) node);
break;		SSIGN_CHOICE:	break;
case (int)	EnterProdInitDecChoice		case (int)
SyntaxConstants.PROD_F	((Production) node);	EnterProdAssignChoice(	SyntaxConstants.PROD_F
UNCTRET:	break;	(Production) node);	UNCT_PARAM:
	case (int)	break;	
<pre>EnterProdFunctret((Pro duction) node);</pre>	SyntaxConstants.PROD_C ONSTANT:	case (int)	EnterProdFunctParam((P
break;	UNSTAINT:	SyntaxConstants.PROD_A CCESS ASSIGN DTYPE:	roduction) node); break;
case (int)	EnterProdConstant((Pro	CCESS_ASSIGN_DITTE.	case (int)
SyntaxConstants.PROD_D	duction) node);	EnterProdAccessAssignD	SyntaxConstants. PROD_F
TYPE A:	break;	type ((Production)	UNCT IDPARAM:
_	case (int)	node);	_
EnterProdDtypeA((Produ	SyntaxConstants.PROD_L	break;	EnterProdFunctIdparam(
ction) node);	OCAL_CHOICE:	case (int)	(Production) node);
break;		SyntaxConstants.PROD_A	break;
case (int)	EnterProdLocalChoice((	SSIGN_VALUE_CHOICE:	case (int)
SyntaxConstants.PROD_E	Production) node);		SyntaxConstants.PROD_A
XDTYPE_A:	break;	EnterProdAssignValueCh	DDFUNCT_IDPARAM:
F. ( . D. 1F 14 A ( / D.	case (int)	oice((Production)	P. 4 . D. 14 11C 4 T.1
EnterProdExdtypeA((Pro duction) node);	SyntaxConstants.PROD_D ECLARE1:	node); break;	<pre>EnterProdAddfunctIdpar am((Production) node);</pre>
break;	ECLARET.	case (int)	break;
case (int)	EnterProdDeclare1((Pro	SyntaxConstants. PROD_A	case (int)
SyntaxConstants. PROD R	duction) node);	SSIGNING:	SyntaxConstants. PROD B
ETURN:	break;		ODY:
	case (int)	EnterProdAssigning((Pr	
EnterProdReturn((Produ	SyntaxConstants.PROD_F	oduction) node);	EnterProdBody((Product
ction) node);	UNCTRET1:	break;	ion) node);
break;		case (int)	break;
case (int)	EnterProdFunctret1((Pr	SyntaxConstants.PROD_A	case (int)
SyntaxConstants.PROD_F	oduction) node);	RRAY_ID:	SyntaxConstants.PROD_P
UNCTVOID:	break;	D . D . IA . T.1//D . I	RINT:
E	case (int)	EnterProdArrayId((Prod	EnterProdPrint((Produc
EnterProdFunctvoid((Pr oduction) node);	SyntaxConstants.PROD_F UNCTVOID1:	uction) node); break;	tion) node);
break:	UNCIVOIDI.	case (int)	break;
case (int)	EnterProdFunctvoid1((P	SyntaxConstants. PROD A	case (int)
SyntaxConstants. PROD S	roduction) node);	RRAY IDTAIL:	SyntaxConstants. PROD_P
TRUCT:	break;	_	OSTVAL:
	case (int)	EnterProdArrayIdtail((	
EnterProdStruct((Produ	SyntaxConstants.PROD_S	Production) node);	EnterProdPostval((Prod
ction) node);	TRUCT1:	break;	uction) node);
break;		case (int)	break;
	EnterProdStruct1((Prod	SyntaxConstants.PROD_A	
	uction) node);	SSIGN_SYM:	

case (int)	break;		case (int)
SyntaxConstants.PROD 0	case (int)	EnterProdElseState((Pr	SyntaxConstants.PROD C
UT:	SyntaxConstants.PROD M	oduction) node);	ASESTATEMENT:
	NT COND:	break;	
EnterProdOut((Producti		case (int)	EnterProdCasestatement
on) node):	EnterProdMntCond((Prod	SyntaxConstants. PROD_E	((Production) node);
, , , , , , , , , , , , , , , , , , ,	uction) node);	LSESTATEMENT:	
break;		LSESTATEMENT:	break;
case (int)	break;	D . D . ID1	case (int)
SyntaxConstants.PROD_0	case (int)	EnterProdElsestatement	SyntaxConstants.PROD_M
UT_C:	SyntaxConstants.PROD_M	((Production) node);	ATH_OP:
	NT_COND_T:	break;	
EnterProdOutC((Product		case (int)	EnterProdMathOp((Produ
ion) node);	EnterProdMntCondT((Pro	SyntaxConstants.PROD_D	ction) node);
break;	duction) node);	OWHILE:	break;
case (int)	break;		case (int)
SyntaxConstants.PROD S	case (int)	EnterProdDowhile((Prod	SyntaxConstants.PROD 0
TRUCT C:	SyntaxConstants.PROD M	uction) node);	PER COND:
11.001_01	NT:	break;	1 21 _ 001 10 1
EnterProdStructC((Prod	IVI .	case (int)	EnterProdOperCond((Pro
* *	EnterProdMnt((Producti	• ,	
uction) node);	* *	SyntaxConstants.PROD_D	duction) node);
break;	on) node);	OSTATEMENT:	break;
case (int)	break;		case (int)
SyntaxConstants.PROD_C	case (int)	EnterProdDostatement((	SyntaxConstants.PROD_0
ONCAT_LIT:	SyntaxConstants.PROD_I	Production) node);	PER_COND_CHOICE:
	FELSE:	break;	
EnterProdConcatLit((Pr		case (int)	EnterProdOperCondChoic
oduction) node);	EnterProdIfelse((Produ	SyntaxConstants.PROD W	e((Production) node);
break;	ction) node);	HILE_STATE:	break;
case (int)	break;	mibb_omib.	case (int)
SyntaxConstants. PROD S	case (int)	EnterProdWhileState((P	SyntaxConstants. PROD 0
-			_
CAN:	SyntaxConstants.PROD_I	roduction) node);	PER_SYM:
	FCONDITION:	break;	
EnterProdScan((Product		case (int)	EnterProdOperSym((Prod
ion) node);	EnterProdIfcondition((	SyntaxConstants.PROD_W	uction) node);
break;	Production) node);	HILESTATEMENT:	break;
case (int)	break;		case (int)
SyntaxConstants.PROD E	case (int)	EnterProdWhilestatemen	SyntaxConstants.PROD 0
XT I:	SyntaxConstants.PROD I	t((Production) node);	PER EQ:
	FSTATEMENT:	break;	
EnterProdExtI((Product	TOTALLINEIVI.	case (int)	EnterProdOperEq((Produ
ion) node);	EnterProdIfstatement((	SyntaxConstants. PROD S	ction) node);
	, ,		
break;	Production) node);	WITCH_STATE:	break;
case (int)	break;		case (int)
SyntaxConstants.PROD_F	case (int)	EnterProdSwitchState((	SyntaxConstants.PROD_0
OR_STATE:	SyntaxConstants.PROD_E	Production) node);	PER_EXT_S:
	LSEIF:	break;	
EnterProdForState((Pro		case (int)	EnterProdOperExtS((Pro
duction) node);	EnterProdElseif((Produ	SyntaxConstants.PROD C	duction) node);
break:	ction) node):	ASE STATE:	break;
case (int)	break;	AGE_STATE.	case (int)
		F. 4 . D. 10 . C. 4 (/D.	
SyntaxConstants.PROD_F	case (int)	EnterProdCaseState((Pr	SyntaxConstants. PROD_0
ORSTATEMENT:	SyntaxConstants.PROD_E	oduction) node);	PER_EXT_REP:
	LSEIFSTATEMENT:	break;	
EnterProdForstatement(		case (int)	EnterProdOperExtRep((P
(Production) node);	EnterProdElseifstateme	SyntaxConstants.PROD_D	roduction) node);
break;	<pre>nt((Production) node);</pre>	EF:	break;
case (int)	break;		case (int)
SyntaxConstants. PROD V	case (int)	EnterProdDef((Producti	SyntaxConstants. PROD 0
-			_
AL1:	SyntaxConstants. PROD_E	on) node);	PERAND:
D . D W 11//D 1	LSE_STATE:	break;	P + P 10 1//P 1
EnterProdVall((Product			EnterProdOperand((Prod
ion) node);			uction) node);

break:		return	case (int)
case (int)	EnterProdLogOper((Prod	ExitConstN((Token)	SyntaxConstants.GETCH:
SyntaxConstants. PROD_S	uction) node):	node):	return
IM MATH OP:	break;	case (int)	ExitGetch((Token)
IM_MATH_OI.	case (int)	SyntaxConstants.RETURN	node);
EnterProdSimMathOp((Pr	SyntaxConstants. PROD E		case (int)
oduction) node);	ND:		SyntaxConstants.STRUCT
	ND:	return	•
break;	D . D . ID . I / / D . I	ExitReturn((Token)	_N:
case (int)	EnterProdEnd((Producti	node);	return
SyntaxConstants.PROD_S	on) node);	case (int)	ExitStructN((Token)
_MATH_EXT:	break;	SyntaxConstants.SWITCH	node);
	}	_N:	case (int)
EnterProdSMathExt((Pro	}	return	SyntaxConstants.DEFAUL
duction) node);		ExitSwitchN((Token)	T:
break;	/**	node);	return
case (int)	* <summary>Called</summary>	case (int)	ExitDefault((Token)
SyntaxConstants.PROD_0	when exiting a parse	SyntaxConstants.CASE_N	node);
PER_COND_EXT:	tree node.	:	case (int)
	*	return	SyntaxConstants.CLEAR:
EnterProdOperCondExt((	* <param< td=""><td>ExitCaseN((Token)</td><td>return</td></param<>	ExitCaseN((Token)	return
Production) node);	name='node'>the node	node);	ExitClear((Token)
break:	being exited (/param)	case (int)	node);
case (int)	being exited\/param/		case (int)
` '		SyntaxConstants.BREAK:	` '
SyntaxConstants.PROD_R	* <returns>the</returns>	return	SyntaxConstants. SQROOT
EL_OP:	node to add to the	ExitBreak((Token)	:
	parse tree, or	node);	return
EnterProdRelOp((Produc	* null	case (int)	ExitSqroot((Token)
tion) node);	if no parse tree	SyntaxConstants.FOR_N:	node);
break;	should be	return	case (int)
case (int)	created	ExitForN((Token)	SyntaxConstants.PLUS:
SyntaxConstants.PROD_R	*	node);	return
ELOP EXT:	* <exception< td=""><td>case (int)</td><td>ExitPlus((Token)</td></exception<>	case (int)	ExitPlus((Token)
	<pre>cref='ParseException'&gt;</pre>	SyntaxConstants.IF:	node);
EnterProdRelopExt((Pro	if the node analysis	return	case (int)
duction) node);	* discovered	<pre>ExitIf((Token) node);</pre>	SyntaxConstants.MINUS:
break:	errors	case (int)	return
case (int)	*/	SyntaxConstants. ELSEIF	ExitMinus((Token)
SyntaxConstants. PROD 0	public override	N:	node):
P1:	Node Exit (Node node) {	return	case (int)
11.		ExitElseifN((Token)	
E + D 10.1//D 1 +:	switch		SyntaxConstants.TIMES:
EnterProdOp1((Producti	(node. Id) {	node);	return
on) node);	case (int)	case (int)	ExitTimes((Token)
break;	SyntaxConstants.MAIN_N	SyntaxConstants.ELSE_N	node);
case (int)	:	:	case (int)
SyntaxConstants.PROD_L	return	return	SyntaxConstants.DIVIDE
OG_OP:	ExitMainN((Token)	ExitElseN((Token)	:
	node);	node);	return
EnterProdLogOp((Produc	case (int)	case (int)	ExitDivide((Token)
tion) node);	SyntaxConstants.PRINT	SyntaxConstants.DO:	node);
break;	N:	return	case (int)
case (int)	return	<pre>ExitDo((Token) node);</pre>	SyntaxConstants. MODULU
SyntaxConstants. PROD E	ExitPrintN((Token)	case (int)	S:
XT LOG OP:	node);	SyntaxConstants. WHILE	return
AI_LOU_OI.		-	
End-approduction (/D)	case (int)	N:	ExitModulus((Token)
EnterProdExtLogOp((Pro	SyntaxConstants.SCAN_N	return	node);
		ExitWhileN((Token)	case (int)
duction) node);	:	1 \	0 . 0
break;	return	node);	SyntaxConstants. EQUALS
break; case (int)	•	case (int)	SyntaxConstants. EQUALS :
break; case (int) SyntaxConstants.PROD_L	return		: return
break; case (int)	return ExitScanN((Token)	case (int)	:
break; case (int) SyntaxConstants.PROD_L	return ExitScanN((Token) node);	case (int) SyntaxConstants.VOID:	: return

(:)			(:)
case (int)	return	return	case (int)
SyntaxConstants.SEMIC:	ExitNewline((Token)	ExitLessE((Token)	SyntaxConstants. ID:
return	node);	node);	return
ExitSemic((Token)	case (int)	case (int)	ExitId((Token) node);
node);	SyntaxConstants.N_E:	SyntaxConstants.S_OBRA	case (int)
case (int)	return	CKET:	SyntaxConstants.NUM:
SyntaxConstants.DOT:	ExitNE((Token) node);	return	return
return	case (int)	ExitSObracket((Token)	ExitNum((Token) node);
ExitDot((Token) node);	SyntaxConstants.O_PARE	node);	case (int)
case (int)	N:	case (int)	SyntaxConstants.DECIMA
SyntaxConstants. COMMA:	return	SyntaxConstants.S_CBRA	L:
return	ExitOParen((Token)	CKET:	return
ExitComma((Token)	node);	return	ExitDecimal((Token)
node);	case (int)	ExitSCbracket((Token)	node);
case (int)	SyntaxConstants.C_PARE	node);	case (int)
SyntaxConstants. AND:	N:	case (int)	SyntaxConstants.S_CHAR
return	return	SyntaxConstants. DOLLAR	:
ExitAnd((Token) node);	ExitCParen((Token)	:	return
case (int)	node);	return	ExitSChar((Token)
SyntaxConstants.OR:	case (int)	ExitDollar((Token)	node);
return	SyntaxConstants.D_QUOT	node);	case (int)
ExitOr((Token) node);	E:	case (int)	SyntaxConstants. TEXT:
case (int)	return	SyntaxConstants. POWER:	return
SyntaxConstants.NOT:	ExitDQuote((Token)	return	ExitText((Token)
return	node);	ExitPower((Token)	node);
<pre>ExitNot((Token) node);</pre>	case (int)	node);	case (int)
case (int)	SyntaxConstants.COLON:	case (int)	SyntaxConstants.COM:
SyntaxConstants. INCREM	return	SyntaxConstants.HASH:	return
ENT:	ExitColon((Token)	return	<pre>ExitCom((Token) node);</pre>
return	node);	ExitHash((Token)	case (int)
ExitIncrement((Token)	case (int)	node);	SyntaxConstants.YES:
node);	SyntaxConstants.O_BRAC	case (int)	return
case (int)	KET:	SyntaxConstants.NEGA:	ExitYes((Token) node);
SyntaxConstants.DECREM	return	return	case (int)
ENT:	ExitOBracket((Token)	ExitNega((Token)	SyntaxConstants.NO:
return	node);	node);	return
ExitDecrement((Token)	case (int)	case (int)	ExitNo((Token) node);
node);	SyntaxConstants.C_BRAC	SyntaxConstants.INT:	case (int)
case (int)	KET:	return	SyntaxConstants. FUNCTN
SyntaxConstants.P_E:	return	ExitInt((Token) node);	AME:
return	ExitCBracket((Token)	case (int)	return
<pre>ExitPE((Token) node);</pre>	node);	SyntaxConstants.CHAR:	ExitFunctname((Token)
case (int)	case (int)	return	node);
SyntaxConstants.M_E:	SyntaxConstants.GREATE	ExitChar((Token)	case (int)
return	R:	node);	SyntaxConstants.STRUCT
ExitME((Token) node);	return	case (int)	NAME:
case (int)	ExitGreater((Token)	SyntaxConstants.FLOAT:	return
SyntaxConstants.T_E:	node);	return	ExitStructname((Token)
return	case (int)	ExitFloat((Token)	node);
<pre>ExitTE((Token) node);</pre>	SyntaxConstants.LESS:	node);	case (int)
case (int)	return	case (int)	SyntaxConstants.IDSTRU
SyntaxConstants.D_E:	ExitLess((Token)	SyntaxConstants.STRING	CT:
return	node);	:	return
<pre>ExitDE((Token) node);</pre>	case (int)	return	ExitIdstruct((Token)
case (int)	SyntaxConstants.GREATE	ExitString((Token)	node);
SyntaxConstants.MOD_E:	R_E:	node);	case (int)
return	return	case (int)	SyntaxConstants.F:
ExitModE((Token)	ExitGreaterE((Token)	SyntaxConstants.BOOL_N	return
node);	node);	:	<pre>ExitF((Token) node);</pre>
case (int)	case (int)	return	case (int)
SyntaxConstants.NEWLIN	SyntaxConstants.LESS_E	ExitBoolN((Token)	SyntaxConstants.D:
E:	:	node);	

return return return return ExitD((Token) node); ExitProdNegate((Produc ExitProdN2((Production ExitProdDtypeA((Produc case (int) tion) node); ) node); tion) node); SyntaxConstants. S: case (int) case (int) case (int) return SyntaxConstants.PROD D SyntaxConstants.PROD I SyntaxConstants.PROD E ExitS((Token) node); ATATYPE: NDEX: XDTYPE A: case (int) return return return ExitProdDatatype((Prod ExitProdIndex((Product ExitProdExdtypeA((Prod SyntaxConstants. ZERO: return uction) node); ion) node); uction) node); ExitZero((Token) case (int) case (int) case (int) node): SyntaxConstants.PROD\_L SyntaxConstants.PROD\_S  ${\tt SyntaxConstants.\,PROD\_R}$ case (int) ITERALS: MATH: ETURN: SyntaxConstants. TOCHAR return return return ExitProdLiterals((Prod ExitProdSmath((Product ExitProdReturn((Produc uction) node): ion) node): tion) node): return case (int) ExitTochar((Token) case (int) case (int) SyntaxConstants.PROD L SyntaxConstants. PROD A SyntaxConstants. PROD F case (int) ITERALS2: RRAY AID: UNCTVOID: SyntaxConstants. LENGTH return return return ExitProdLiterals2((Pro ExitProdArrayAid((Prod ExitProdFunctvoid((Pro duction) node); uction) node); duction) node); return ExitLengthf ((Token) case (int) case (int) case (int) SyntaxConstants.PROD G SyntaxConstants.PROD E SyntaxConstants.PROD S node); case (int) LOBAL DEC: LEM CHOICE: TRUCT: SyntaxConstants. CONTAI return return return ExitProdGlobalDec((Pro ExitProdElemChoice((Pr ExitProdStruct((Produc duction) node); oduction) node); tion) node); return case (int) ExitContains((Token) case (int) case (int) SyntaxConstants.PROD D SyntaxConstants.PROD E SyntaxConstants.PROD M node): case (int) ECLARE: LEMENT: EM DEC: SyntaxConstants. REVERS return return return ExitProdDeclare((Produ ExitProdElement((Produ ExitProdMemDec((Produc ction) node); ction) node); tion) node); return ExitReverse ((Token) case (int) case (int) case (int) node); SyntaxConstants.PROD\_D SyntaxConstants.PROD\_A  ${\tt SyntaxConstants.\,PROD\_I}$ ECLARE\_CHOICE: case (int) DD ELEM: NIT DEC: SyntaxConstants.PROD\_S return return TART\_PROGRAM: ExitProdDeclareChoice( ExitProdAddElem((Produ ExitProdInitDec((Produ return (Production) node); ction) node); ction) node); ExitProdStartProgram(( case (int) case (int) case (int) SyntaxConstants.PROD\_I Production) node); SyntaxConstants.PROD M  ${\tt SyntaxConstants.\,PROD\_I}$ case (int) NIT CHOICE: ELEM: NIT DEC CHOICE: SyntaxConstants.PROD P return return return ExitProdInitChoice((Pr ExitProdMElem((Product ExitProdInitDecChoice( ROGRAM: oduction) node); ion) node); (Production) node); return ExitProdProgram((Produ case (int) case (int) case (int) ction) node); SyntaxConstants. PROD A SyntaxConstants.PROD M SyntaxConstants. PROD C case (int) DD ID: 2 ELEM: ONSTANT: SyntaxConstants.PROD C return return return ExitProdAddId((Product ExitProdConstant((Prod LEAR: ExitProdM2E1em((Produc tion) node); uction) node); return ion) node); ExitProdClear((Product case (int) case (int) case (int) SyntaxConstants.PROD\_F  ${\tt SyntaxConstants.\,PROD\_L}$ ion) node); SyntaxConstants.PROD\_N case (int) UNCTRET: OCAL\_CHOICE:  ${\tt SyntaxConstants.\,PROD\_C}$ return return return OMMENTS: ExitProdN1((Production ExitProdFunctret((Prod ExitProdLocalChoice((P ) node); uction) node); roduction) node); return ExitProdComments((Prod case (int) case (int) case (int) uction) node): SyntaxConstants.PROD\_N SyntaxConstants.PROD D SyntaxConstants.PROD D case (int) TYPE A: ECLARE1:

SyntaxConstants.PROD N

EGATE:

return return return return ExitProdDeclarel((Prod ExitProdArrayId((Produ ExitProdPostval((Produ ExitProdMntCond((Produ uction) node); ction) node); ction) node); ction) node); case (int) case (int) case (int) case (int) SyntaxConstants.PROD F SyntaxConstants. PROD A SyntaxConstants.PROD 0 SyntaxConstants.PROD M UNCTRET1: RRAY IDTAIL: NT COND T: UT: return return return return ExitProdArrayIdtail((P ExitProdFunctret1((Pro ExitProdOut((Productio ExitProdMntCondT((Prod duction) node); roduction) node); n) node); uction) node); case (int) case (int) case (int) case (int) SyntaxConstants.PROD F SyntaxConstants.PROD\_A SyntaxConstants.PROD\_0  ${\tt SyntaxConstants.\,PROD\_M}$ UNCTVOID1: SSIGN SYM: UT C: NT: return return return return ExitProdFunctvoid1((Pr ExitProdAssignSym((Pro ExitProdOutC((Producti ExitProdMnt((Productio oduction) node); duction) node); on) node): n) node): case (int) case (int) case (int) case (int) SyntaxConstants.PROD S SyntaxConstants. PROD A SyntaxConstants.PROD S SyntaxConstants.PROD I TRUCT1: SSIGN VALUE: TRUCT C: FELSE: return return return return ExitProdIfelse((Produc ExitProdStruct1((Produ ExitProdAssignValue((P ExitProdStructC((Produ ction) node); roduction) node); ction) node); tion) node); case (int) case (int) case (int) case (int) SyntaxConstants.PROD C SyntaxConstants.PROD C SyntaxConstants.PROD C SyntaxConstants.PROD I FCONDITION: ONSTANT1: ONVERT: ONCAT LIT: return return return return ExitProdConstant1((Pro ExitProdConvert((Produ ExitProdConcatLit((Pro ExitProdIfcondition((P ction) node); duction) node); roduction) node); duction) node); case (int) case (int) case (int) case (int) SyntaxConstants.PROD M SyntaxConstants.PROD F SyntaxConstants.PROD S SyntaxConstants.PROD I UNCT PARAM: FSTATEMENT: return return return return ExitProdMain((Producti ExitProdFunctParam((Pr ExitProdScan((Producti ExitProdIfstatement((P on) node); oduction) node); on) node); roduction) node); case (int) case (int) case (int) case (int) SyntaxConstants.PROD\_F  ${\tt SyntaxConstants.\,PROD\_A}$ SyntaxConstants.PROD\_E  ${\tt SyntaxConstants.\,PROD\_E}$ UNCT\_IDPARAM: SSIGN CHOICE: XT I: LSEIF: return return ExitProdAssignChoice(( ExitProdFunctIdparam(( ExitProdExtI((Producti ExitProdElseif((Produc Production) node); Production) node); on) node); tion) node); case (int) case (int) case (int) case (int) SyntaxConstants.PROD A SyntaxConstants. PROD A SyntaxConstants.PROD F SyntaxConstants.PROD E CCESS ASSIGN DTYPE: DDFUNCT IDPARAM: OR STATE: LSEIFSTATEMENT: return return return return  ${\tt ExitProdAccessAssignDt}$ ExitProdAddfunctIdpara ExitProdForState((Prod ExitProdElseifstatemen ype((Production) m((Production) node); uction) node); t((Production) node); case (int) node); case (int) case (int) case (int) SyntaxConstants.PROD B SyntaxConstants.PROD F SyntaxConstants.PROD E SyntaxConstants. PROD A ODY: ORSTATEMENT: LSE STATE: SSIGN VALUE CHOICE: return return return ExitProdBody((Producti ExitProdElseState((Pro return ExitProdForstatement(( ExitProdAssignValueCho on) node); duction) node); Production) node); ice((Production) case (int) case (int) case (int) SyntaxConstants.PROD\_E node); SyntaxConstants.PROD\_P SyntaxConstants.PROD\_V case (int) RINT: AL1: LSESTATEMENT: SyntaxConstants.PROD\_A return return return SSIGNING: ExitProdPrint((Product ExitProdVall((Producti ExitProdElsestatement( ion) node); on) node): (Production) node); return ExitProdAssigning((Pro case (int) case (int) case (int) duction) node); SyntaxConstants. PROD P SyntaxConstants.PROD M SyntaxConstants.PROD D case (int) OSTVAL: NT COND: OWHILE:

SyntaxConstants. PROD A

RRAY\_ID:

return	return	return	case (int)
ExitProdDowhile((Produ	${\tt ExitProdOperCondChoice}$	ExitProdRelopExt((Prod	${\tt SyntaxConstants.PROD\_S}$
ction) node);	((Production) node);	uction) node);	TART_PROGRAM:
case (int)	case (int)	case (int)	Ch: 1 dDr dC + + Dr (
SyntaxConstants.PROD_D OSTATEMENT:	SyntaxConstants.PROD_0 PER_SYM:	SyntaxConstants.PROD_0 P1:	<pre>ChildProdStartProgram( node, child);</pre>
return	return	return	break;
ExitProdDostatement((P	ExitProdOperSym((Produ	ExitProdOp1((Productio	case (int)
roduction) node);	ction) node);	n) node);	SyntaxConstants.PROD_P
case (int)	case (int)	case (int)	ROGRAM:
SyntaxConstants.PROD_W	SyntaxConstants.PROD_0	SyntaxConstants.PROD_L	
HILE_STATE: return	PER_EQ: return	OG_OP:	ChildProdProgram(node, child);
ExitProdWhileState((Pr	ExitProdOperEq((Produc	ExitProdLogOp((Product	break;
oduction) node);	tion) node);	ion) node);	case (int)
case (int)	case (int)	case (int)	SyntaxConstants.PROD_C
SyntaxConstants.PROD_W	SyntaxConstants.PROD_0	SyntaxConstants.PROD_E	LEAR:
HILESTATEMENT:	PER_EXT_S:	XT_LOG_OP:	OL : 1 ID 101 / 1
return ExitProdWhilestatement	return ExitProdOperExtS((Prod	return ExitProdExtLogOp((Prod	ChildProdClear(node, child);
((Production) node);	uction) node);	uction) node);	break;
case (int)	case (int)	case (int)	case (int)
SyntaxConstants. PROD_S	SyntaxConstants.PROD_0	SyntaxConstants.PROD_L	SyntaxConstants.PROD_C
WITCH_STATE:	PER_EXT_REP:	OG_OPER:	OMMENTS:
return	return	return	CI : 1 ID 10 ( )
<pre>ExitProdSwitchState((P roduction) node);</pre>	ExitProdOperExtRep((Pr oduction) node);	ExitProdLogOper((Produ ction) node);	ChildProdComments(node
case (int)	case (int)	ction, node,, case (int)	, child); break;
SyntaxConstants. PROD_C	SyntaxConstants.PROD 0	SyntaxConstants.PROD_E	case (int)
ASE_STATE:	PERAND:	ND:	SyntaxConstants.PROD_N
return	return	return	EGATE:
ExitProdCaseState((Pro	ExitProdOperand((Produ	ExitProdEnd((Productio	CL 11 ID IN / I
duction) node); case (int)	ction) node); case (int)	n) node);	ChildProdNegate(node, child);
SyntaxConstants. PROD_D	SyntaxConstants. PROD_S	return node;	break;
EF:	IM_MATH_OP:	}	case (int)
return	return		SyntaxConstants.PROD_D
ExitProdDef((Productio	ExitProdSimMathOp((Pro	/**	ATATYPE:
n) node);	duction) node);	* <summary>Called</summary>	
case (int) SyntaxConstants.PROD C	case (int) SyntaxConstants.PROD S	when adding a child to a parse tree	<pre>ChildProdDatatype(node , child);</pre>
ASESTATEMENT:	_MATH_EXT:	* node.	break;
return	return	*	case (int)
ExitProdCasestatement(	ExitProdSMathExt((Prod	* <param< td=""><td>SyntaxConstants.PROD_L</td></param<>	SyntaxConstants.PROD_L
(Production) node);	uction) node);	name='node'>the parent	ITERALS:
case (int)	case (int)	node	ChildProdLiterals(node
SyntaxConstants.PROD_M ATH OP:	SyntaxConstants.PROD_0 PER COND EXT:	* <param name="child"/> the child	, child);
return	return	node, or null	break;
ExitProdMathOp((Produc	ExitProdOperCondExt((P	*	case (int)
tion) node);	roduction) node);	* <exception< td=""><td>SyntaxConstants.PROD_L</td></exception<>	SyntaxConstants.PROD_L
case (int)	case (int)	<pre>cref='ParseException'&gt;</pre>	ITERALS2:
SyntaxConstants. PROD_0	SyntaxConstants. PROD_R	if the node analysis	CL:11D11:4 - 1 0/ 1
PER_COND:	EL_OP: return	<pre>* discovered errors</pre>	ChildProdLiterals2(nod e, child);
return	ExitProdRelOp((Product	*/	e, chira), break;
		,	
<pre>ExitProdOperCond((Prod uction) node);</pre>	ion) node);	public override	case (int)
ExitProdOperCond((Prod		<pre>public override void Child(Production</pre>	case (int) SyntaxConstants.PROD_G
ExitProdOperCond((Prod uction) node); case (int) SyntaxConstants.PROD_0	ion) node); case (int) SyntaxConstants.PROD_R	<pre>void Child(Production node, Node child) {</pre>	
<pre>ExitProdOperCond((Prod uction) node);     case (int)</pre>	ion) node); case (int)	void Child (Production	${\tt SyntaxConstants.PROD\_G}$

break;		case (int)	break;
case (int)	ChildProdArrayAid(node	SyntaxConstants.PROD R	case (int)
SyntaxConstants.PROD_D	, child);	ETURN:	SyntaxConstants.PROD_F
ECLARE:	break;		UNCTRET1:
	case (int)	ChildProdReturn(node,	
ChildProdDeclare(node,	SyntaxConstants.PROD E	child);	ChildProdFunctret1(nod
child);	LEM CHOICE:	break;	e, child);
break;	_	case (int)	break;
case (int)	ChildProdElemChoice(no	SyntaxConstants.PROD F	case (int)
SyntaxConstants.PROD D	de, child);	UNCTVOID:	SyntaxConstants.PROD F
ECLARE CHOICE:	break;		UNCTVOID1:
	case (int)	ChildProdFunctvoid(nod	
ChildProdDeclareChoice	SyntaxConstants.PROD E	e, child);	ChildProdFunctvoid1(no
(node, child);	LEMENT:	break;	de, child);
break;		case (int)	break;
case (int)	ChildProdElement(node,	SyntaxConstants.PROD_S	case (int)
SyntaxConstants.PROD_I	child);	TRUCT:	SyntaxConstants.PROD_S
NIT_CHOICE:	break;		TRUCT1:
	case (int)	ChildProdStruct(node,	
ChildProdInitChoice(no	SyntaxConstants.PROD_A	child);	ChildProdStruct1(node,
de, child);	DD_ELEM:	break;	child);
break;	_	case (int)	break;
case (int)	ChildProdAddElem(node,	SyntaxConstants.PROD_M	case (int)
SyntaxConstants.PROD A	child);	EM DEC:	SyntaxConstants.PROD C
DD ID:	break;	_	ONSTANT1:
_	case (int)	ChildProdMemDec(node,	
ChildProdAddId(node,	SyntaxConstants.PROD_M	child);	ChildProdConstant1(nod
child);	ELEM:	break;	e, child);
break;		case (int)	break;
case (int)	ChildProdMElem(node,	SyntaxConstants.PROD_I	case (int)
SyntaxConstants.PROD_N	child);	NIT_DEC:	SyntaxConstants.PROD_M
1:	break;		AIN:
	case (int)	ChildProdInitDec(node,	
ChildProdN1(node,	SyntaxConstants.PROD_M	child);	ChildProdMain(node,
child);	2_ELEM:	break;	child);
break;		case (int)	break;
case (int)	ChildProdM2Elem(node,	SyntaxConstants.PROD_I	case (int)
SyntaxConstants.PROD_N	child);	NIT_DEC_CHOICE:	SyntaxConstants.PROD_A
2:	break;		SSIGN_CHOICE:
	case (int)	ChildProdInitDecChoice	
ChildProdN2(node,	SyntaxConstants.PROD_F	(node, child);	ChildProdAssignChoice(
child);	UNCTRET:	break;	node, child);
break;		case (int)	break;
case (int)	ChildProdFunctret(node	SyntaxConstants.PROD_C	case (int)
SyntaxConstants.PROD_I	, child);	ONSTANT:	SyntaxConstants.PROD_A
NDEX:	break;		CCESS_ASSIGN_DTYPE:
	case (int)	ChildProdConstant(node	
ChildProdIndex(node,	SyntaxConstants.PROD_D	, child);	ChildProdAccessAssignD
child);	TYPE_A:	break;	type(node, child);
break;		case (int)	break;
case (int)	ChildProdDtypeA(node,	SyntaxConstants.PROD_L	case (int)
SyntaxConstants.PROD_S	child);	OCAL_CHOICE:	SyntaxConstants.PROD_A
MATH:	break;		SSIGN_VALUE_CHOICE:
	case (int)	ChildProdLocalChoice(n	
ChildProdSmath(node,	SyntaxConstants.PROD_E	ode, child);	ChildProdAssignValueCh
child);	XDTYPE_A:	break;	oice(node, child);
		case (int)	break;
break;			(. )
break; case (int)	ChildProdExdtypeA(node	SyntaxConstants.PROD_D	case (int)
	<pre>ChildProdExdtypeA(node , child);</pre>	SyntaxConstants.PROD_D ECLARE1:	case (int) SyntaxConstants.PROD_A
case (int)		_	
case (int) SyntaxConstants.PROD_A	, child);	_	SyntaxConstants.PROD_A

case (int) break; ChildProdAssigning(nod SyntaxConstants.PROD B ChildProdIfstatement(n case (int) ODY: SyntaxConstants.PROD F e, child); ode, child); break; OR STATE: break; case (int) ChildProdBody (node, case (int) ChildProdForState(node SyntaxConstants.PROD A child): SyntaxConstants.PROD E RRAY ID: LSEIF: break: . child): case (int) break; SyntaxConstants.PROD\_P ChildProdElseif(node, ChildProdArrayId(node, case (int) SyntaxConstants.PROD\_F child); RINT: child); ORSTATEMENT: break: break: case (int) ChildProdPrint(node, case (int) SyntaxConstants.PROD A child): ChildProdForstatement( SyntaxConstants.PROD E RRAY IDTAIL: node, child); LSEIFSTATEMENT: hreak. case (int) break: ChildProdArrayIdtail(n ChildProdElseifstateme SyntaxConstants. PROD P case (int) ode, child); OSTVAL: SyntaxConstants.PROD V nt(node, child); break; AL1: break: ChildProdPostval(node, case (int) case (int) ChildProdVall(node,  $SyntaxConstants.PROD\_A$ child);  ${\tt SyntaxConstants.\,PROD\_E}$ SSIGN SYM: child); LSE STATE: break: case (int) break; ChildProdElseState(nod ChildProdAssignSym(nod SyntaxConstants.PROD 0 case (int) e, child); UT: SyntaxConstants.PROD M e, child); NT COND: break; break: ChildProdOut(node, case (int) case (int) SyntaxConstants.PROD\_A ChildProdMntCond(node, SyntaxConstants.PROD E child); LSESTATEMENT: SSIGN VALUE: break: child): case (int) break: ChildProdAssignValue(n SyntaxConstants.PROD 0 case (int) ChildProdElsestatement SyntaxConstants.PROD M ode, child): UT C: (node, child): NT COND T: break; break; case (int) ChildProdOutC(node, case (int) SyntaxConstants.PROD C child); ChildProdMntCondT(node SyntaxConstants.PROD D ONVERT: OWHILE: break; , child); case (int) break; ChildProdConvert (node, SyntaxConstants.PROD S ChildProdDowhile(node, case (int) child); TRUCT\_C: SyntaxConstants.PROD\_M child); break: break: ChildProdStructC(node, case (int) case (int) SyntaxConstants.PROD F child): ChildProdMnt (node, SyntaxConstants.PROD D UNCT PARAM: child); OSTATEMENT: break: case (int) break: ChildProdFunctParam(no ChildProdDostatement(n SyntaxConstants.PROD\_C case (int) de, child); ONCAT LIT: SyntaxConstants.PROD I ode, child); break; FELSE: break; case (int) ChildProdConcatLit(nod case (int) ChildProdIfelse(node, SyntaxConstants.PROD F e, child); SyntaxConstants.PROD W UNCT IDPARAM: HILE STATE: break: child): case (int) break; ChildProdFunctIdparam( ChildProdWhileState(no SyntaxConstants.PROD\_S case (int) node, child); CAN: SyntaxConstants.PROD\_I de, child); FCONDITION: break; break: case (int) ChildProdScan(node, case (int) SyntaxConstants.PROD\_A child): ChildProdIfcondition(n SyntaxConstants.PROD W DDFUNCT IDPARAM: hreak. ode, child); HILESTATEMENT: case (int) break; ChildProdAddfunctIdpar SyntaxConstants.PROD E case (int) ChildProdWhilestatemen am(node, child): XT I: SyntaxConstants.PROD I t (node, child); break; FSTATEMENT: break; ChildProdExtI (node,

child);

case (int)	break;		* <param< th=""></param<>
SyntaxConstants. PROD S	case (int)	ChildProdOp1(node,	name='node'>the node
WITCH STATE:	SyntaxConstants. PROD_0	child);	being exited
"1101 <u>"</u> 51111 <u>5</u> 1	PER EXT S:	break;	*
ChildProdSwitchState(n		case (int)	* <returns>the</returns>
ode, child);	ChildProdOperExtS(node	SyntaxConstants.PROD L	node to add to the
break;	, child);	OG_OP:	parse tree, or
case (int)	break;		* null
SyntaxConstants.PROD C	case (int)	ChildProdLogOp(node,	if no parse tree
ASE STATE:	SyntaxConstants.PROD_0	child);	should be
	PER_EXT_REP:	break;	created
ChildProdCaseState(nod		case (int)	*
e, child);	ChildProdOperExtRep(no	SyntaxConstants.PROD_E	* <exception< td=""></exception<>
break;	de, child);	XT_LOG_OP:	<pre>cref='ParseException'&gt;</pre>
case (int)	break;		if the node analysis
SyntaxConstants.PROD_D	case (int)	ChildProdExtLogOp(node	* discovered
EF:	SyntaxConstants.PROD_0	, child);	errors
	PERAND:	break;	*/
ChildProdDef(node,		case (int)	public virtual
child);	ChildProdOperand(node,	SyntaxConstants.PROD_L	Node ExitMainN(Token
break;	child);	OG_OPER:	node) {
case (int)	break;		return node;
SyntaxConstants.PROD_C	case (int)	ChildProdLogOper(node,	}
ASESTATEMENT:	SyntaxConstants.PROD_S	child);	
	IM_MATH_OP:	break;	/**
ChildProdCasestatement		case (int)	* <summary>Called</summary>
(node, child);	ChildProdSimMathOp(nod	SyntaxConstants.PROD_E	when entering a parse
break;	e, child);	ND:	tree node.
case (int)	break;		*
SyntaxConstants.PROD_M	case (int)	ChildProdEnd(node,	* <param< td=""></param<>
ATH_OP:	SyntaxConstants.PROD_S	child);	name='node'>the node
	_MATH_EXT:	break;	being entered
ChildProdMathOp(node,		}	*
child);	ChildProdSMathExt(node	}	* <exception< td=""></exception<>
break;	, child);		cref='ParseException'>
case (int)	break;	/**	if the node analysis
SyntaxConstants. PROD_0	case (int)	* <summary>Called</summary>	* discovered
PER_COND:	SyntaxConstants.PROD_0	when entering a parse	errors
	DED COMP EVT.		
CL:11D: 10 - C - 1/- 1	PER_COND_EXT:	tree node.	*/
ChildProdOperCond(node		*	public virtual
, child);	ChildProdOperCondExt(n	* * <param< td=""><td><pre>public virtual void EnterPrintN(Token</pre></td></param<>	<pre>public virtual void EnterPrintN(Token</pre>
, child); break;	ChildProdOperCondExt(n ode, child);	*     * <param name="node"/> the node	<pre>public virtual void EnterPrintN(Token node) {</pre>
, child); break; case (int)	ChildProdOperCondExt(n ode, child); break;	*     * <param name="node"/> the node being entered	<pre>public virtual void EnterPrintN(Token</pre>
, child); break; case (int) SyntaxConstants.PROD_0	ChildProdOperCondExt(n ode, child); break; case (int)	*     * <param name="node"/> the node being entered *	<pre>public virtual void EnterPrintN(Token node) {     } }</pre>
, child); break; case (int)	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R	*     * <param name="node"/> the node being entered     *     * <exception< td=""><td><pre>public virtual void EnterPrintN(Token node) {      }      /**</pre></td></exception<>	<pre>public virtual void EnterPrintN(Token node) {      }      /**</pre>
, child); break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:	ChildProdOperCondExt(n ode, child); break; case (int)	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"></exception>	<pre>public virtual void EnterPrintN(Token node) {      }  /**      * <summary>Called</summary></pre>
, child); break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE: ChildProdOperCondChoic	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP:	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis</exception>	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse</summary></pre>
<pre>, child);</pre>	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node,	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered</exception>	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse tree node.</summary></pre>
<pre>break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE: ChildProdOperCondChoic e(node, child); break;</pre>	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child);	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception>	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse tree node.</summary>      *</pre>
<pre>break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child);     break; case (int)</pre>	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break;	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse tree node.</summary>      *      * <param< pre=""></param<></pre>
break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child); break; case (int) SyntaxConstants.PROD_O	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int)	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse tree node.</summary>      *      * <param name="node"/>the node</pre>
<pre>break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child);     break; case (int)</pre>	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int) SyntaxConstants.PROD_R	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse tree node.</summary>      *      * <param< pre=""></param<></pre>
break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child); break; case (int) SyntaxConstants.PROD_O PER_SYM:	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int)	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual	<pre>public virtual void EnterPrintN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>     *     * <param name="node"/>the node being exited     *</pre>
<pre>break; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child);</pre>	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int) SyntaxConstants.PROD_R ELOP_EXT:	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse tree node.</summary>      *      * <param name="node"/>the node being exited      *      * <returns>the</returns></pre>
<pre>break; case (int) SyntaxConstants.PROD_0 PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child);</pre>	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int) SyntaxConstants.PROD_R ELOP_EXT: ChildProdRelopExt(node	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token	<pre>public virtual void EnterPrintN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>     *     * <param name="node"/>the node being exited     *     * <returns>the node to add to the</returns></pre>
<pre>break; case (int) SyntaxConstants.PROD_0 PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child);</pre>	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int) SyntaxConstants.PROD_R ELOP_EXT: ChildProdRelopExt(node, child);	*     * * <param name="node"/> the node being entered     *     * * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token node) {     }  /**	<pre>public virtual void EnterPrintN(Token node) {      }       /**      * <summary>Called when exiting a parse tree node.</summary>      *      * <param name="node"/>the node being exited      *      * <returns>the</returns></pre>
preak; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child); break; case (int) SyntaxConstants.PROD_O PER_SYM:  ChildProdOperSym(node, child); break; case (int)	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int) SyntaxConstants.PROD_R ELOP_EXT: ChildProdRelopExt(node, child); break;	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token node) {     }  /**     * <summary>Called</summary>	<pre>public virtual void EnterPrintN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>     *     * <param name="node"/>the node being exited     *     * <returns>the node to add to the parse tree, or     * null</returns></pre>
preak; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child); break; case (int) SyntaxConstants.PROD_O PER_SYM:  ChildProdOperSym(node, child); break; case (int) SyntaxConstants.PROD_O	ChildProdOperCondExt(n ode, child);	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token node) {     }      /**     * <summary>Called when exiting a parse</summary>	<pre>public virtual void EnterPrintN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>     *     * <param name="node"/>the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree</returns></pre>
preak; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child); break; case (int) SyntaxConstants.PROD_O PER_SYM:  ChildProdOperSym(node, child); break; case (int)	ChildProdOperCondExt(n ode, child); break; case (int) SyntaxConstants.PROD_R EL_OP: ChildProdRelOp(node, child); break; case (int) SyntaxConstants.PROD_R ELOP_EXT: ChildProdRelopExt(node, child); break;	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token node) {     }  /**     * <summary>Called</summary>	<pre>public virtual void EnterPrintN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>     *     * <param name="node"/>the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree should be</returns></pre>
preak; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child); break; case (int) SyntaxConstants.PROD_O PER_SYM:  ChildProdOperSym(node, child); break; case (int) SyntaxConstants.PROD_O	ChildProdOperCondExt(n ode, child);	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>	<pre>public virtual void EnterPrintN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>     *     * <param name="node"/>the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree</returns></pre>
preak; case (int) SyntaxConstants.PROD_O PER_COND_CHOICE:  ChildProdOperCondChoic e(node, child); break; case (int) SyntaxConstants.PROD_O PER_SYM:  ChildProdOperSym(node, child); break; case (int) SyntaxConstants.PROD_O PER_EQ:	ChildProdOperCondExt(n ode, child);	*     * <param name="node"/> the node being entered     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void EnterMainN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>	<pre>public virtual void EnterPrintN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>     *     * <param name="node"/>the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree should be created</returns></pre>

* <exception< th=""><th>* <summary>Called</summary></th><th>*/</th><th>* <param< th=""></param<></th></exception<>	* <summary>Called</summary>	*/	* <param< th=""></param<>
cref='ParseException'>	when entering a parse	public virtual	name='node'>the node
if the node analysis	tree node.	void EnterReturn(Token	being exited
* discovered	*	node) {	*
errors	* <param< td=""><td>}</td><td>* <returns>the</returns></td></param<>	}	* <returns>the</returns>
*/	name='node'>the node		node to add to the
public virtual	being entered	/**	parse tree, or
Node ExitPrintN(Token	*	* <summary>Called</summary>	* null
node) {	* <exception< td=""><td>when exiting a parse</td><td>if no parse tree</td></exception<>	when exiting a parse	if no parse tree
return node;	cref='ParseException'>	tree node.	should be created
}	if the node analysis * discovered	* * <param< td=""><td>created</td></param<>	created
/**	errors	name='node'>the node	* <exception< td=""></exception<>
* <summary>Called</summary>	*/	being exited	cref='ParseException'>
when entering a parse	public virtual	*	if the node analysis
tree node.	void EnterConstN(Token	* <returns>the</returns>	* discovered
*	node) {	node to add to the	errors
* <param< td=""><td>}</td><td>parse tree, or</td><td>*/</td></param<>	}	parse tree, or	*/
name='node'>the node		* null	public virtual
being entered	/**	if no parse tree	Node ExitSwitchN(Token
*	* <summary>Called</summary>	should be	node) {
* <exception< td=""><td>when exiting a parse</td><td>created</td><td>return node;</td></exception<>	when exiting a parse	created	return node;
<pre>cref='ParseException'&gt;</pre>	tree node.	*	}
if the node analysis	*	* <exception< td=""><td>/state</td></exception<>	/state
<pre>* discovered errors</pre>	* <param name='node'&gt;the node</param 	cref='ParseException'> if the node analysis	/** * <summary>Called</summary>
*/	being exited	* discovered	when entering a parse
public virtual	*	errors	tree node.
void EnterScanN(Token	* <returns>the</returns>	*/	*
node) {	node to add to the	public virtual	* <param< td=""></param<>
}	parse tree, or	Node ExitReturn(Token	name='node'>the node
	* null	node) {	being entered
/**	if no parse tree	return node;	*
* <summary>Called</summary>	should be	}	* <exception< td=""></exception<>
when exiting a parse	created		<pre>cref='ParseException'&gt;</pre>
tree node.	*	/**	if the node analysis
*	* <exception< td=""><td>* <summary>Called</summary></td><td>* discovered</td></exception<>	* <summary>Called</summary>	* discovered
* <param name='node'&gt;the node</param 	<pre>cref='ParseException'&gt; if the node analysis</pre>	when entering a parse	errors */
being exited	* discovered	tree node.	*/ public virtual
*	errors	* <param< td=""><td>void EnterCaseN(Token</td></param<>	void EnterCaseN(Token
* <returns>the</returns>	*/	name='node'>the node	node) {
node to add to the	public virtual	being entered	}
parse tree, or	Node ExitConstN(Token	*	,
* null	node) {	* <exception< td=""><td>/**</td></exception<>	/**
if no parse tree	return node;	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>
should be	}	if the node analysis	when exiting a parse
created		* discovered	tree node.
*	/**	errors	*
* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td><td>* <param< td=""></param<></td></exception<>	* <summary>Called</summary>	*/	* <param< td=""></param<>
cref='ParseException'>	when entering a parse	public virtual	name='node'>the node
if the node analysis * discovered	tree node.	void EnterSwitchN(Token	being exited *
errors (/exception)	* <param< td=""><td>node) {</td><td>* <returns>the</returns></td></param<>	node) {	* <returns>the</returns>
*/	name='node'>the node	}	node to add to the
public virtual	being entered	,	parse tree, or
Node ExitScanN(Token	*	/**	* null
node) {	* <exception< td=""><td>* <summary>Called</summary></td><td>if no parse tree</td></exception<>	* <summary>Called</summary>	if no parse tree
return node;	cref='ParseException'>	when exiting a parse	should be
}	if the node analysis	tree node.	created
	* discovered	*	*
/**	errors		

* <exception< th=""><th>* <summary>Called</summary></th><th>*/</th><th>* <param< th=""></param<></th></exception<>	* <summary>Called</summary>	*/	* <param< th=""></param<>
<pre>cref='ParseException'&gt;</pre>	when entering a parse	public virtual	name='node'>the node
if the node analysis	tree node.	void EnterIf(Token	being exited
* discovered	*	node) {	*
errors	* <param< td=""><td>}</td><td>* <returns>the</returns></td></param<>	}	* <returns>the</returns>
*/	name='node'>the node		node to add to the
public virtual	being entered	/**	parse tree, or
Node ExitCaseN(Token	*	* <summary>Called</summary>	* null
node) {	* <exception< td=""><td>when exiting a parse</td><td>if no parse tree</td></exception<>	when exiting a parse	if no parse tree
return node;	cref='ParseException'>	tree node.	should be
}	if the node analysis	*	created
	* discovered	* <param< td=""><td>*</td></param<>	*
/**	errors	name='node'>the node	* <exception< td=""></exception<>
* <summary>Called</summary>	*/	being exited	<pre>cref='ParseException'&gt;</pre>
when entering a parse	public virtual	*	if the node analysis
tree node.	void EnterForN(Token	* <returns>the</returns>	* discovered
*	node) {	node to add to the	errors
* <param< td=""><td>}</td><td>parse tree, or</td><td>*/</td></param<>	}	parse tree, or	*/
name='node'>the node		* null	public virtual
being entered	/**	if no parse tree	Node ExitElseifN(Token
*	* <summary>Called</summary>	should be	node) {
* <exception< td=""><td>when exiting a parse</td><td>created</td><td>return node;</td></exception<>	when exiting a parse	created	return node;
cref='ParseException'>	tree node.	*	}
if the node analysis	*	* <exception< td=""><td></td></exception<>	
* discovered	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>/**</td></param<>	<pre>cref='ParseException'&gt;</pre>	/**
errors	name='node'>the node	if the node analysis	* <summary>Called</summary>
*/	being exited	* discovered	when entering a parse
public virtual	*	errors	tree node.
void EnterBreak (Token	* <returns>the</returns>	*/	*
node) {	node to add to the	public virtual	* <param< td=""></param<>
}	parse tree, or * null	Node ExitIf(Token	name='node'>the node
/ state	· Hall	node) {	being entered
/**	if no parse tree	return node;	*
* <summary>Called when exiting a parse</summary>	should be created	,	<pre>* <exception cref="ParseException"></exception></pre>
tree node.	*	/**	if the node analysis
tree node. // Summary/	7	\ 444b	11 the houe analysis
*	* (overntion	* (cummary)Callod	
* * ⟨naram	* <exception cref="ParseFycention"></exception>	* <summary>Called</summary>	* discovered
*  * <param name="node"/> the node	<pre>cref='ParseException'&gt;</pre>	when entering a parse	* discovered errors
name='node'>the node	<pre>cref='ParseException'&gt; if the node analysis</pre>	when entering a parse tree node.	<pre>* discovered errors */</pre>
	<pre>cref='ParseException'&gt; if the node analysis     * discovered</pre>	when entering a parse tree node.	* discovered errors */ public virtual
name='node'>the node being exited *	<pre>cref='ParseException'&gt; if the node analysis</pre>	when entering a parse tree node.  *  * <pre></pre>	* discovered errors */ public virtual void EnterElseN(Token
name='node'>the node being exited * * <returns>the</returns>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */</pre>	<pre>when entering a parse tree node.      *      * <param name="node"/>the node</pre>	* discovered errors */ public virtual
name='node'>the node being exited *	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors</pre>	when entering a parse tree node.  *  * <pre></pre>	<pre>* discovered errors     */     public virtual void EnterElseN(Token node) {</pre>
name='node'>the node being exited  *  * <returns>the node to add to the</returns>	<pre>cref='ParseException'&gt; if the node analysis   * discovered errors   */   public virtual</pre>	<pre>when entering a parse tree node.      *      * <param name="node"/>the node</pre>	<pre>* discovered errors     */     public virtual void EnterElseN(Token node) {</pre>
name='node'>the node being exited  *  * <returns>the node to add to the parse tree, or</returns>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token</pre>	<pre>when entering a parse tree node.     *     * <param name="node"/>the node being entered     *</pre>	<pre>* discovered errors     */     public virtual void EnterElseN(Token node) {     } }</pre>
name='node'>the node being exited  *  * <returns>the node to add to the parse tree, or  * null</returns>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {</pre>	when entering a parse tree node.  *  * <param name="node"/> the node being entered  *  * <exception< td=""><td><pre>* discovered errors     */     public virtual void EnterElseN(Token node) {     }     /***</pre></td></exception<>	<pre>* discovered errors     */     public virtual void EnterElseN(Token node) {     }     /***</pre>
name='node'>the node being exited  *  * <returns>the node to add to the parse tree, or  * null if no parse tree</returns>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {</pre>	when entering a parse tree node.  *  * <param name="node"/> the node being entered  *  * <exception cref="ParseException"></exception>	<pre>* discovered errors     */     public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called</summary></pre>
name='node'>the node being exited  *  * <returns>the node to add to the parse tree, or  * null if no parse tree should be</returns>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {</pre>	when entering a parse tree node.  *  * <param name="node"/> the node being entered  *  * <exception cref="ParseException"> if the node analysis</exception>	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }  /**     * <summary>Called when exiting a parse</summary>
name='node'>the node being exited  *  * <returns>the node to add to the parse tree, or  * null if no parse tree should be</returns>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {         return node;     }</pre>	when entering a parse tree node.  *  * <param name="node"/> the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered</exception>	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node.</summary>
name='node'>the node being exited  *  * <returns>the node to add to the parse tree, or  * null if no parse tree should be created</returns> *	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {         return node;     } /**</pre>	when entering a parse tree node.   *  * * <param name="node"/> the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception>	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. */summary&gt; *</summary>
name='node'>the node being exited  *  * <returns>the node to add to the parse tree, or  * null if no parse tree should be created</returns> *  * <exception< td=""><td><pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {         return node;     }      /**     * <summary>Called</summary></pre></td><td>when entering a parse tree node.   *  * * <param name="node"/>the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception>  */  public virtual void</td><td><pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <parame< pre=""></parame<></summary></td></exception<>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {         return node;     }      /**     * <summary>Called</summary></pre>	when entering a parse tree node.   *  * * <param name="node"/> the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */  public virtual void	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <parame< pre=""></parame<></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**     * <summary>Called when entering a parse tree node. </summary>     *</pre>	when entering a parse tree node.   *  * * <param name="node"/> the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */  public virtual void EnterElseifN(Token	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <pre>     * <param name="node"/>the node being exited *</pre></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param< pre=""></param<></pre>	when entering a parse tree node.   *  * * <param name="node"/> the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */  public virtual void	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <pre>     * <param name="node"/>the node being exited     *     * <returns>the</returns></pre></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param name="node"/>the node</pre>	when entering a parse tree node.   *  * * <param name="node"/> the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */  public virtual void EnterElseifN(Token	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * * param name='node'&gt;the node being exited     *     * <returns>the node to add to the</returns></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param name="node"/>the node being entered</pre>	<pre>when entering a parse tree node. </pre>	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <pre>param name='node'&gt;the node being exited</pre> * <pre>     * <returns>the node to add to the parse tree, or</returns></pre></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param name="node"/>the node being entered       *</pre>	when entering a parse tree node.   *  * * \param name='node' > the node being entered   *  * \exception cref='ParseException'> if the node analysis  * discovered errors   */  public virtual void EnterElseifN(Token node) { }  /**	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <param name="node"/>the node being exited     * <returns>the node to add to the parse tree, or     * null</returns></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param name="node"/>the node being entered       *       * <exception< pre=""></exception<></pre>	when entering a parse tree node.   *  * * \squaram name='node'>the node being entered  *  * \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * * param name='node'&gt;the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree</returns></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param name="node"/>the node being entered       *       * <exception cref="ParseException"></exception></pre>	when entering a parse tree node.   *  * * \param name='node'>the node being entered  *  * \exception cref='ParseException'> if the node analysis  * discovered errors  */  public virtual void EnterElseifN(Token node) {  }  /**  * <summary>Called when exiting a parse</summary>	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <param name="node"/>the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree should be</returns></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param name="node"/>the node being entered       *       * <exception cref="ParseException"> if the node analysis</exception></pre>	when entering a parse tree node.   *  * * \squaram name='node'>the node being entered  *  * * \exception cref='ParseException'> if the node analysis  * discovered errors  */  public virtual void EnterElseifN(Token node) {  }  /**  * \summary>Called when exiting a parse tree node.	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * * param name='node'&gt;the node being exited     * <returns>the node to add to the parse tree, or     * null if no parse tree should be created</returns></summary>
name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitForN(Token node) {        return node;     }      /**       * <summary>Called when entering a parse tree node. </summary>       *       * <param name="node"/>the node being entered       *       * <exception cref="ParseException"></exception></pre>	when entering a parse tree node.   *  * * \param name='node'>the node being entered  *  * \exception cref='ParseException'> if the node analysis  * discovered errors  */  public virtual void EnterElseifN(Token node) {  }  /**  * <summary>Called when exiting a parse</summary>	<pre>* discovered errors</pre> */ public virtual void EnterElseN(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. * <param name="node"/>the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree should be</returns></summary>

* <exception< th=""><th>* <summary>Called</summary></th><th>*/</th><th>* <param< th=""></param<></th></exception<>	* <summary>Called</summary>	*/	* <param< th=""></param<>
<pre>cref='ParseException'&gt;</pre>	when entering a parse	public virtual	name='node'>the node
if the node analysis	tree node.	void EnterVoid(Token	being exited
* discovered	*	node) {	*
errors	* <param< td=""><td>}</td><td>* <returns>the</returns></td></param<>	}	* <returns>the</returns>
*/	name='node'>the node		node to add to the
public virtual	being entered	/**	parse tree, or
Node ExitElseN(Token	*	* <summary>Called</summary>	* null
node) {	* <exception< td=""><td>when exiting a parse</td><td>if no parse tree</td></exception<>	when exiting a parse	if no parse tree
return node;	<pre>cref='ParseException'&gt;</pre>	tree node.	should be
}	if the node analysis	*	created
	* discovered	* <param< td=""><td>*</td></param<>	*
/**	errors	name='node'>the node	* <exception< td=""></exception<>
* <summary>Called</summary>	*/	being exited	cref='ParseException'>
when entering a parse	public virtual	*	if the node analysis
tree node.	void EnterWhileN(Token	* <returns>the</returns>	* discovered
*	node) {	node to add to the	errors
* <param< td=""><td>}</td><td>parse tree, or</td><td>*/</td></param<>	}	parse tree, or	*/
name='node'>the node	,	* null	public virtual
being entered	/**	if no parse tree	Node ExitGetch(Token
*	* <summary>Called</summary>	should be	node) {
* <exception< td=""><td>when exiting a parse</td><td>created</td><td>return node;</td></exception<>	when exiting a parse	created	return node;
cref='ParseException'>	tree node.	*	}
if the node analysis	*	* <exception< td=""><td>/</td></exception<>	/
* discovered	* <param< td=""><td>cref='ParseException'&gt;</td><td>/**</td></param<>	cref='ParseException'>	/**
errors */	name='node'>the node being exited	if the node analysis * discovered	* <summary>Called when entering a parse</summary>
≁/ public virtual	being exited\/param/ *	errors	tree node.
void EnterDo(Token	* <returns>the</returns>	*/	*
node) {	node to add to the	public virtual	* <param< td=""></param<>
houe) (	parse tree, or	Node ExitVoid(Token	name='node'>the node
J	* null	node) {	being entered
/**	if no parse tree	return node;	*
* <summary>Called</summary>	should be	}	* <exception< td=""></exception<>
when exiting a parse	created	•	cref='ParseException'>
tree node.	*	/**	if the node analysis
*	* <exception< td=""><td>* <summary>Called</summary></td><td>* discovered</td></exception<>	* <summary>Called</summary>	* discovered
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>when entering a parse</td><td>errors</td></param<>	<pre>cref='ParseException'&gt;</pre>	when entering a parse	errors
name='node'>the node	if the node analysis	tree node.	*/
being exited	* discovered	*	public virtual
*	errors	* <param< td=""><td>void</td></param<>	void
* <returns>the</returns>	*/	name='node'>the node	EnterStructN(Token
node to add to the	public virtual	being entered	node) {
parse tree, or	Node ExitWhileN(Token	*	}
* null	node) {	* <exception< td=""><td></td></exception<>	
if no parse tree	return node;	cref='ParseException'>	/**
should be	}	if the node analysis	* <summary>Called</summary>
created		* discovered	when exiting a parse
*	/**	errors	tree node.
* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td><td>*</td></exception<>	* <summary>Called</summary>	*/	*
cref='ParseException'>	when entering a parse	public virtual	* <param< td=""></param<>
if the node analysis	tree node.	void EnterGetch(Token	name='node'>the node
* discovered		node) {	being exited
errors */	* <param name="node"/> the node	ſ	* * <returns>the</returns>
*/ public virtual	being entered	/**	node to add to the
Node ExitDo(Token	being entered/param/	* <summary>Called</summary>	parse tree, or
node Exitho(Token node) {	* <exception< td=""><td>when exiting a parse</td><td>parse tree, or null</td></exception<>	when exiting a parse	parse tree, or null
return node;	cref='ParseException'>	tree node.	if no parse tree
}	if the node analysis	*	should be
,	* discovered		created
/**	errors		*
•			

* <exception< th=""><th>/**</th><th>* discovered</th><th>* <param< th=""></param<></th></exception<>	/**	* discovered	* <param< th=""></param<>
cref='ParseException'>	* <summary>Called</summary>	errors	name='node'>the node
if the node analysis	when entering a parse	*/	being exited
* discovered	tree node.	public virtual	*
errors	*	void EnterSgroot (Token	* <returns>the</returns>
*/	* <param< td=""><td>node) {</td><td>node to add to the</td></param<>	node) {	node to add to the
public virtual	name='node'>the node	}	parse tree, or
Node ExitStructN(Token	being entered	,	* null
node) {	*	/**	if no parse tree
return node;	* <exception< td=""><td>* <summary>Called</summary></td><td>should be</td></exception<>	* <summary>Called</summary>	should be
}	cref='ParseException'>	when exiting a parse	created
,	if the node analysis	tree node.	*
/**	* discovered	*	* <exception< td=""></exception<>
* <summary>Called</summary>	errors	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td></param<>	<pre>cref='ParseException'&gt;</pre>
when entering a parse	*/	name='node'>the node	if the node analysis
tree node.	public virtual	being exited	* discovered
*	void EnterClear(Token	*	errors
* <param< td=""><td>node) {</td><td>* <returns>the</returns></td><td>*/</td></param<>	node) {	* <returns>the</returns>	*/
name='node'>the node	}	node to add to the	public virtual
being entered	,	parse tree, or	Node ExitPlus(Token
*	/**	* null	node) {
* <exception< td=""><td>* <summary>Called</summary></td><td>if no parse tree</td><td>return node;</td></exception<>	* <summary>Called</summary>	if no parse tree	return node;
<pre>cref='ParseException'&gt;</pre>	when exiting a parse	should be	}
if the node analysis	tree node.	created	,
* discovered	*	*	/**
errors	* <param< td=""><td>* <exception< td=""><td>* <summary>Called</summary></td></exception<></td></param<>	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
*/	name='node'>the node	cref='ParseException'>	when entering a parse
public virtual	being exited	if the node analysis	tree node.
void	*	* discovered	*
EnterDefault(Token	* <returns>the</returns>	errors	* <param< td=""></param<>
node) {	node to add to the	*/	name='node'>the node
}	parse tree, or	public virtual	being entered
	* null	Node ExitSqroot(Token	*
/**	if no parse tree	node) {	* <exception< td=""></exception<>
* <summary>Called</summary>	should be	return node;	<pre>cref='ParseException'&gt;</pre>
when exiting a parse	created	}	if the node analysis
tree node.	*		* discovered
*	* <exception< td=""><td>/**</td><td>errors</td></exception<>	/**	errors
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>* <summary>Called</summary></td><td>*/</td></param<>	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	*/
name='node'>the node	if the node analysis	when entering a parse	public virtual
being exited	* discovered	tree node.	void EnterMinus(Token
*	errors	*	node) {
* <returns>the</returns>	*/	* <param< td=""><td>}</td></param<>	}
node to add to the	public virtual	name='node'>the node	
parse tree, or	Node ExitClear(Token	being entered	/**
* null	node) {	*	* <summary>Called</summary>
if no parse tree	return node;	* <exception< td=""><td>when exiting a parse</td></exception<>	when exiting a parse
should be	}	<pre>cref='ParseException'&gt;</pre>	tree node.
created		if the node analysis	*
*	/**	* discovered	* <param< td=""></param<>
* <exception< td=""><td>* <summary>Called</summary></td><td>errors</td><td>name='node'&gt;the node</td></exception<>	* <summary>Called</summary>	errors	name='node'>the node
cref='ParseException'>	when entering a parse	*/	being exited
if the node analysis	tree node.	public virtual	*
* discovered	*	void EnterPlus(Token	* <returns>the</returns>
errors	* <param< td=""><td>node) {</td><td>node to add to the</td></param<>	node) {	node to add to the
*/	name='node'>the node	}	parse tree, or
public virtual	being entered	,	* null
Node ExitDefault(Token	*	/**	if no parse tree
node) {	* <exception< td=""><td>* <summary>Called</summary></td><td>should be</td></exception<>	* <summary>Called</summary>	should be
return node;	cref='ParseException'>	when exiting a parse	created
}	if the node analysis	tree node.	*

* <exception< th=""><th>* <summary>Called</summary></th><th>*/</th><th>* <param< th=""></param<></th></exception<>	* <summary>Called</summary>	*/	* <param< th=""></param<>
cref='ParseException'>	when entering a parse	public virtual	name='node'>the node
if the node analysis	tree node.	void	being exited
* discovered	*	EnterModulus(Token	*
errors	* <param< td=""><td>node) {</td><td>* <returns>the</returns></td></param<>	node) {	* <returns>the</returns>
*/	name='node'>the node	}	node to add to the
public virtual	being entered		parse tree, or
Node ExitMinus(Token	*	/**	* null
node) {	* <exception< td=""><td>* <summary>Called</summary></td><td>if no parse tree</td></exception<>	* <summary>Called</summary>	if no parse tree
return node;	cref='ParseException'>	when exiting a parse	should be
}	if the node analysis	tree node.	created
,	* discovered	*	*
/**	errors	* <pre> * <pre> * <pre> * <pre> * </pre></pre></pre></pre>	* <exception< td=""></exception<>
* <summary>Called</summary>	*/	name='node'>the node	cref='ParseException'>
when entering a parse tree node.	public virtual void EnterDivide(Token	being exited *	if the node analysis * discovered
*	node) {	* <returns>the</returns>	errors
* <param< td=""><td>}</td><td>node to add to the</td><td>*/</td></param<>	}	node to add to the	*/
name='node'>the node	,	parse tree, or	public virtual
being entered	/**	* null	Node ExitEquals (Token
*	* <summary>Called</summary>	if no parse tree	node) {
* <exception< td=""><td>when exiting a parse</td><td>should be</td><td>return node;</td></exception<>	when exiting a parse	should be	return node;
<pre>cref='ParseException'&gt;</pre>	tree node.	created	}
if the node analysis	*	*	
* discovered	* <param< td=""><td>* <exception< td=""><td>/**</td></exception<></td></param<>	* <exception< td=""><td>/**</td></exception<>	/**
errors	name='node'>the node	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>
*/	being exited	if the node analysis	when entering a parse
public virtual	*	* discovered	tree node.
void EnterTimes(Token	* <returns>the</returns>	errors	*
node) {	node to add to the	*/	* <param< td=""></param<>
}	parse tree, or * null	public virtual	name='node'>the node
/**		Node ExitModulus(Token node) {	being entered
/** * <summary>Called</summary>	if no parse tree should be		* <exception< td=""></exception<>
when exiting a parse	created	return node;	cref='ParseException'>
tree node.	*	J	if the node analysis
*	* <exception< td=""><td>/**</td><td>* discovered</td></exception<>	/**	* discovered
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>* <summary>Called</summary></td><td>errors</td></param<>	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	errors
name='node'>the node	if the node analysis	when entering a parse	*/
being exited	* discovered	tree node.	public virtual
*	errors	*	void EnterSemic(Token
* <returns>the</returns>	*/	* <param< td=""><td>node) {</td></param<>	node) {
node to add to the	public virtual	name='node'>the node	}
parse tree, or	Node ExitDivide(Token	being entered	
* null	node) {	*	/**
if no parse tree	return node;	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
should be	}	cref='ParseException'>	when exiting a parse
created	/	if the node analysis	tree node.
*	/**	* discovered	*
* <exception< td=""><td>* <summary>Called</summary></td><td>errors */</td><td>* <param name="node"/>the node</td></exception<>	* <summary>Called</summary>	errors */	* <param name="node"/> the node
<pre>cref='ParseException'&gt; if the node analysis</pre>	when entering a parse tree node.	public virtual	being exited
* discovered	*	void EnterEquals(Token	*
errors	* <param< td=""><td>node) {</td><td>* <returns>the</returns></td></param<>	node) {	* <returns>the</returns>
*/	name='node'>the node	}	node to add to the
public virtual	being entered	•	parse tree, or
Node ExitTimes (Token	*	/**	* null
node) {	* <exception< td=""><td>* <summary>Called</summary></td><td>if no parse tree</td></exception<>	* <summary>Called</summary>	if no parse tree
return node;	<pre>cref='ParseException'&gt;</pre>	when exiting a parse	should be
}	if the node analysis	tree node.	created
	* discovered	*	*
/**	errors		

* <exception< th=""><th>* <summary>Called</summary></th><th>*/</th><th>* <param< th=""></param<></th></exception<>	* <summary>Called</summary>	*/	* <param< th=""></param<>
<pre>cref='ParseException'&gt;</pre>	when entering a parse	public virtual	name='node'>the node
if the node analysis	tree node.	void EnterAnd(Token	being exited
* discovered	*	node) {	*
errors	* <param< td=""><td>}</td><td>* <returns>the</returns></td></param<>	}	* <returns>the</returns>
*/	name='node'>the node	•	node to add to the
public virtual	being entered	/**	parse tree, or
Node ExitSemic (Token	*	* <summary>Called</summary>	* null
node) {	* <exception< td=""><td>when exiting a parse</td><td>if no parse tree</td></exception<>	when exiting a parse	if no parse tree
return node;	cref='ParseException'>	tree node.	should be
}	if the node analysis	*	created
	* discovered	* <param< td=""><td>*</td></param<>	*
/**	errors	name='node'>the node	* <exception< td=""></exception<>
* <summary>Called</summary>	*/	being exited	cref='ParseException'>
when entering a parse	public virtual	*	if the node analysis
tree node.	void EnterComma(Token	* <returns>the</returns>	* discovered
*	node) {	node to add to the	errors
* <param< td=""><td>}</td><td>parse tree, or</td><td>*/</td></param<>	}	parse tree, or	*/
name='node'>the node		* null	public virtual
being entered	/**	if no parse tree	Node ExitOr(Token
*	* <summary>Called</summary>	should be	node) {
* <exception< td=""><td>when exiting a parse</td><td>created</td><td>return node;</td></exception<>	when exiting a parse	created	return node;
<pre>cref='ParseException'&gt;</pre>	tree node.	*	}
if the node analysis	*	* <exception< td=""><td></td></exception<>	
* discovered	* <param< td=""><td>cref='ParseException'&gt;</td><td>/**</td></param<>	cref='ParseException'>	/**
errors	name='node'>the node	if the node analysis	* <summary>Called</summary>
*/	being exited	* discovered	when entering a parse
public virtual	*	errors	tree node.
void EnterDot(Token	* <returns>the</returns>	*/	*
node) {	node to add to the	public virtual	* <param< td=""></param<>
}	parse tree, or	Node ExitAnd(Token	name='node'>the node
	* null	node) {	being entered
/**	if no parse tree	return node;	*
* <summary>Called</summary>	should be	}	* <exception< td=""></exception<>
when exiting a parse	created		<pre>cref='ParseException'&gt;</pre>
tree node.	*	/**	if the node analysis
*	* <exception< td=""><td>* <summary>Called</summary></td><td>* discovered</td></exception<>	* <summary>Called</summary>	* discovered
* <param< td=""><td>cref='ParseException'&gt;</td><td>when entering a parse</td><td>errors</td></param<>	cref='ParseException'>	when entering a parse	errors
name='node'>the node	if the node analysis	tree node.	*/
being exited	* discovered	*	public virtual
*	errors	* <param< td=""><td>void EnterNot(Token</td></param<>	void EnterNot(Token
* <returns>the</returns>	*/	name='node'>the node	node) {
node to add to the	public virtual	being entered	}
parse tree, or	Node ExitComma(Token	*	
* null	node) {	* <exception< td=""><td>/**</td></exception<>	/**
if no parse tree	return node;	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>
should be			
Should be	}	if the node analysis	when exiting a parse
created	}	* discovered	when exiting a parse tree node.
	} /**	* discovered errors	
created * * <pre>*cexception</pre>	} /** * <summary>Called</summary>	<pre>* discovered errors */</pre>	tree node. * * *param
<pre>created      *      * <exception cref="ParseException"></exception></pre>	* <summary>Called when entering a parse</summary>	* discovered errors */ public virtual	<pre>tree node.     *     * <param name="node"/>the node</pre>
<pre>created     *     * <exception cref="ParseException"> if the node analysis</exception></pre>	* <summary>Called</summary>	* discovered errors     */     public virtual void EnterOr(Token	tree node. * * *param
<pre>created     *     * <exception cref="ParseException"> if the node analysis     * discovered</exception></pre>	* <summary>Called when entering a parse tree node.</summary> *	* discovered errors */ public virtual	<pre>tree node.</pre> *  * * param name='node'>the node being exited *
<pre>created     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception></pre>	* <summary>Called when entering a parse tree node.</summary> * * <param< td=""><td>* discovered errors     */     public virtual void EnterOr(Token</td><td><pre>tree node.       *      * <param name="node"/>the node being exited      *      * <returns>the</returns></pre></td></param<>	* discovered errors     */     public virtual void EnterOr(Token	<pre>tree node.       *      * <param name="node"/>the node being exited      *      * <returns>the</returns></pre>
<pre>created     *     * &lt; exception cref='ParseException'&gt; if the node analysis     * discovered errors     */</pre>	* <summary>Called when entering a parse tree node.</summary> *     * <param name="node"/> the node	* discovered errors     */     public virtual void EnterOr(Token node) { }	<pre>tree node.            *           * <param name="node"/>the node being exited           *           * <returns>the node to add to the</returns></pre>
<pre>created     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception>     */     public virtual</pre>	* <summary>Called when entering a parse tree node.</summary>	<pre>* discovered errors     */     public virtual void EnterOr(Token node) {     }     /**</pre>	<pre>tree node.            *           * <param name="node"/>the node being exited           *           * <returns>the node to add to the parse tree, or</returns></pre>
<pre>created     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception>     */     public virtual Node ExitDot(Token</pre>	<pre>* <summary>Called when entering a parse tree node.</summary></pre> *  * <param name="node"/> the node being entered *	<pre>* discovered errors     */     public virtual void EnterOr(Token node) {     }      /**     * <summary>Called</summary></pre>	<pre>tree node.       *      * <param name="node"/>the node being exited      *      * <returns>the node to add to the parse tree, or      * null</returns></pre>
<pre>created     *     * &lt; exception cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitDot(Token node) {</pre>	<pre>* <summary>Called when entering a parse tree node.</summary></pre> *  * <param name="node"/> the node being entered  *  * <exception< pre=""></exception<>	<pre>* discovered errors     */     public virtual void EnterOr(Token node) {     }      /**     * <summary>Called when exiting a parse</summary></pre>	<pre>tree node. </pre>
<pre>created     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception>     */     public virtual Node ExitDot(Token</pre>	* <summary>Called when entering a parse tree node. </summary>	<pre>* discovered errors     */     public virtual void EnterOr(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. </summary></pre>	<pre>tree node. </pre>
<pre>created     *     * &lt; exception cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitDot(Token node) {</pre>	* <summary>Called when entering a parse tree node.</summary>	<pre>* discovered errors     */     public virtual void EnterOr(Token node) {     }      /**     * <summary>Called when exiting a parse</summary></pre>	<pre>tree node. </pre>
<pre>created     *     * &lt; exception cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitDot(Token node) {</pre>	* <summary>Called when entering a parse tree node. </summary>	<pre>* discovered errors     */     public virtual void EnterOr(Token node) {     }      /**     * <summary>Called when exiting a parse tree node. </summary></pre>	<pre>tree node. </pre>

* <exception< th=""><th></th><th>* <exception< th=""><th>* <summary>Called</summary></th></exception<></th></exception<>		* <exception< th=""><th>* <summary>Called</summary></th></exception<>	* <summary>Called</summary>
cref='ParseException'>	/**	cref='ParseException'>	when exiting a parse
if the node analysis	* <summary>Called</summary>	if the node analysis	tree node.
* discovered	when entering a parse	* discovered	*
errors	tree node.	errors	* <param< td=""></param<>
*/	*	*/	name='node'>the node
public virtual	* <param< td=""><td>public virtual</td><td>being exited</td></param<>	public virtual	being exited
Node ExitNot(Token	name='node'>the node	void EnterPE(Token	*
node) {	being entered	node) {	* <returns>the</returns>
return node;	*	}	node to add to the
}	* <exception< td=""><td>/</td><td>parse tree, or</td></exception<>	/	parse tree, or
/stute	cref='ParseException'>	/**	* null
/** * <summary>Called</summary>	if the node analysis  * discovered	* <summary>Called when exiting a parse</summary>	if no parse tree should be
when entering a parse	errors	tree node.	created
tree node.	*/	*	*
*	public virtual	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
* <param< td=""><td>void</td><td>name='node'&gt;the node</td><td><pre>cref='ParseException'&gt;</pre></td></param<>	void	name='node'>the node	<pre>cref='ParseException'&gt;</pre>
name='node'>the node	EnterDecrement(Token	being exited	if the node analysis
being entered	node) {	*	* discovered
*	}	* <returns>the</returns>	errors
* <exception< td=""><td></td><td>node to add to the</td><td>*/</td></exception<>		node to add to the	*/
<pre>cref='ParseException'&gt;</pre>	/**	parse tree, or	public virtual
if the node analysis	* <summary>Called</summary>	* null	Node ExitME(Token
* discovered	when exiting a parse	if no parse tree	node) {
errors */	tree node.	should be	return node;
*/ public virtual	* * <param< td=""><td>created</td><td>}</td></param<>	created	}
void	name='node'>the node	* <exception< td=""><td>/**</td></exception<>	/**
EnterIncrement (Token	being exited	cref='ParseException'>	* <summary>Called</summary>
node) {	*	if the node analysis	when entering a parse
}	* <returns>the</returns>	* discovered	tree node.
	node to add to the	errors	*
/**	parse tree, or	*/	* <param< td=""></param<>
* <summary>Called</summary>	* null	public virtual	name='node'>the node
when exiting a parse	if no parse tree	Node ExitPE(Token	being entered
tree node.	should be	node) {	*
*	created	return node;	* <exception< td=""></exception<>
* <param< td=""><td>*</td><td>}</td><td><pre>cref='ParseException'&gt;</pre></td></param<>	*	}	<pre>cref='ParseException'&gt;</pre>
name='node'>the node being exited	<pre>* <exception cref="ParseException"></exception></pre>	/**	if the node analysis * discovered
*	if the node analysis	* <summary>Called</summary>	errors
* <returns>the</returns>	* discovered	when entering a parse	*/
node to add to the	errors	tree node.	public virtual
parse tree, or	*/	*	void EnterTE(Token
* null	public virtual	* <param< td=""><td>node) {</td></param<>	node) {
if no parse tree	Node	name='node'>the node	}
should be	ExitDecrement (Token	being entered	
created	node) {	*	/**
*	return node;	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
* <exception< td=""><td>}</td><td>cref='ParseException'&gt;</td><td>when exiting a parse</td></exception<>	}	cref='ParseException'>	when exiting a parse
cref='ParseException'>	/storte	if the node analysis	tree node.
if the node analysis * discovered	/** * <summary>Called</summary>	<pre>* discovered errors</pre>	* * <param< td=""></param<>
* discovered errors	* \summary/carred when entering a parse	errors*/	name='node'>the node
*/	tree node.	public virtual	being exited
public virtual	*	void EnterME(Token	*
Node	* <param< td=""><td>node) {</td><td>* <returns>the</returns></td></param<>	node) {	* <returns>the</returns>
ExitIncrement (Token	name='node'>the node	}	node to add to the
node) {	being entered		parse tree, or
return node;	*	/**	* null
}			if no parse tree

should be	}	* <exception< th=""><th>* <summary>Called</summary></th></exception<>	* <summary>Called</summary>
created	J	cref='ParseException'>	when exiting a parse
*	/**	if the node analysis	tree node.
* <exception< td=""><td>* <summary>Called</summary></td><td>* discovered</td><td>*</td></exception<>	* <summary>Called</summary>	* discovered	*
cref='ParseException'>	when entering a parse	errors	* <param< td=""></param<>
if the node analysis	tree node.	*/	name='node'>the node
* discovered	*	public virtual	being exited
errors	* <param< td=""><td>void</td><td>*</td></param<>	void	*
*/	name='node'>the node	EnterNewline (Token	* <returns>the</returns>
public virtual	being entered	node) {	node to add to the
Node ExitTE(Token	*	}	parse tree, or
node) {	* <exception< td=""><td>,</td><td>* null</td></exception<>	,	* null
return node;	<pre>cref='ParseException'&gt;</pre>	/**	if no parse tree
}	if the node analysis	* <summary>Called</summary>	should be
	* discovered	when exiting a parse	created
/**	errors	tree node.	*
* <summary>Called</summary>	*/	*	* <exception< td=""></exception<>
when entering a parse	public virtual	* <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
tree node.	void EnterModE(Token	name='node'>the node	if the node analysis
*	node) {	being exited	* discovered
* <param< td=""><td>}</td><td>*</td><td>errors</td></param<>	}	*	errors
name='node'>the node		* <returns>the</returns>	*/
being entered	/**	node to add to the	public virtual
*	* <summary>Called</summary>	parse tree, or	Node ExitNE(Token
* <exception< td=""><td>when exiting a parse</td><td>* null</td><td>node) {</td></exception<>	when exiting a parse	* null	node) {
<pre>cref='ParseException'&gt;</pre>	tree node.	if no parse tree	return node;
if the node analysis	*	should be	}
* discovered	* <param< td=""><td>created</td><td></td></param<>	created	
errors	name='node'>the node	*	/**
*/	being exited	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
public virtual	*	cref='ParseException'>	when entering a parse
void EnterDE(Token	* <returns>the</returns>	if the node analysis	tree node.
node) {	node to add to the	* discovered	*
}	parse tree, or	errors	* <param< td=""></param<>
	* null	*/	name='node'>the node
/**	if no parse tree	public virtual	being entered
* <summary>Called</summary>	should be	Node ExitNewline(Token	*
when exiting a parse	created	node) {	* <exception< td=""></exception<>
tree node.	*	return node;	cref='ParseException'>
*	* <exception< td=""><td>}</td><td>if the node analysis</td></exception<>	}	if the node analysis
* <param< td=""><td>cref='ParseException'&gt;</td><td></td><td>* discovered</td></param<>	cref='ParseException'>		* discovered
name='node'>the node	if the node analysis	/**	errors
being exited	* discovered	* <summary>Called</summary>	*/
*	errors	when entering a parse	public virtual
* <returns>the</returns>	*/	tree node.	void EnterOParen(Token
node to add to the	public virtual	*	node) {
parse tree, or	Node ExitModE(Token	* <pre>* <pre>* </pre></pre>	}
* null	node) {	name='node'>the node	/
if no parse tree	return node;	being entered	/**
should be	}	*	* <summary>Called</summary>
created	/state	<pre>* <exception cref="ParseException"></exception></pre>	when exiting a parse
* (avaantian	/**	•	tree node.
<pre>* <exception cref="ParseException"></exception></pre>	* <summary>Called when entering a parse</summary>	if the node analysis * discovered	
	wnen entering a parse tree node.		* <param name='node'&gt;the node</param 
<pre>if the node analysis   * discovered</pre>	tree node.	errors */	name= node >the node being exited
errors	* <param< td=""><td>public virtual</td><td>*</td></param<>	public virtual	*
errors\/exception/ */	name='node'>the node	void EnterNE(Token	*  * <returns>the</returns>
*/ public virtual	being entered	node) {	node to add to the
Node ExitDE(Token	being entered√param/ *	noue) (	parse tree, or
node ExitDE (Token node) {	**	J	parse tree, or null
return node;		/**	if no parse tree
Totalii iloue,		,	II no paroo oroc

should be	}	* <exception< th=""><th>* <summary>Called</summary></th></exception<>	* <summary>Called</summary>
created	,	cref='ParseException'>	when exiting a parse
*	/**	if the node analysis	tree node.
* <exception< td=""><td>* <summary>Called</summary></td><td>* discovered</td><td>*</td></exception<>	* <summary>Called</summary>	* discovered	*
<pre>cref='ParseException'&gt;</pre>	when entering a parse	errors	* <param< td=""></param<>
if the node analysis	tree node.	*/	name='node'>the node
* discovered	*	public virtual	being exited
errors	* <param< td=""><td>void EnterColon(Token</td><td>*</td></param<>	void EnterColon(Token	*
*/	name='node'>the node	node) {	* <returns>the</returns>
public virtual	being entered	}	node to add to the
Node ExitOParen(Token	*		parse tree, or
node) {	* <exception< td=""><td>/**</td><td>* null</td></exception<>	/**	* null
return node;	cref='ParseException'>	* <summary>Called</summary>	if no parse tree
}	if the node analysis	when exiting a parse	should be
	* discovered	tree node.	created
/**	errors	*	*
* <summary>Called</summary>	*/	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
when entering a parse	public virtual	name='node'>the node	cref='ParseException'>
tree node.	void EnterDQuote(Token	being exited	if the node analysis
*	node) {	*	* discovered
* <param< td=""><td>}</td><td>* <returns>the</returns></td><td>errors</td></param<>	}	* <returns>the</returns>	errors
name='node'>the node		node to add to the	*/
being entered	/**	parse tree, or	public virtual
*	* <summary>Called</summary>	* null	Node
* <exception< td=""><td>when exiting a parse</td><td>if no parse tree</td><td>ExitOBracket(Token</td></exception<>	when exiting a parse	if no parse tree	ExitOBracket(Token
cref='ParseException'>	tree node.	should be	node) {
if the node analysis	*	created	return node;
* discovered	* <param< td=""><td>*</td><td>}</td></param<>	*	}
errors	name='node'>the node	* <exception< td=""><td></td></exception<>	
*/	being exited	cref='ParseException'>	/**
public virtual	*	if the node analysis	* <summary>Called</summary>
void EnterCParen(Token	* <returns>the</returns>	* discovered	when entering a parse
node) {	node to add to the	errors	tree node.
}	parse tree, or	*/	*
	* null	public virtual	* <param< td=""></param<>
/**	if no parse tree	Node ExitColon(Token	name='node'>the node
* <summary>Called</summary>	should be	node) {	being entered
when exiting a parse	created	return node;	*
tree node.	*	}	* <exception< td=""></exception<>
*	* <exception< td=""><td></td><td><pre>cref='ParseException'&gt;</pre></td></exception<>		<pre>cref='ParseException'&gt;</pre>
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>/**</td><td>if the node analysis</td></param<>	<pre>cref='ParseException'&gt;</pre>	/**	if the node analysis
name='node'>the node	if the node analysis	* <summary>Called</summary>	* discovered
being exited	* discovered	when entering a parse	errors
*	errors	tree node.	*/
* <returns>the</returns>	*/	*	public virtual
node to add to the	public virtual	* <param< td=""><td>void</td></param<>	void
parse tree, or	Node ExitDQuote(Token	name='node'>the node	EnterCBracket (Token
* null	node) {	being entered	node) {
if no parse tree should be	return node;	*	}
created	,	<pre>* <exception cref="ParseException"></exception></pre>	/**
createu/returns/	/**	if the node analysis	,
•	,	* discovered	* <summary>Called</summary>
<pre>* <exception cref="ParseException"></exception></pre>	* <summary>Called when entering a parse</summary>	* discovered errors	when exiting a parse tree node.
if the node analysis	tree node.	errors*/	tree node. // Summary/
* discovered	tree node. summary/</td <td>*/ public virtual</td> <td>* <param< td=""></param<></td>	*/ public virtual	* <param< td=""></param<>
errors	* <param< td=""><td>void</td><td>name='node'&gt;the node</td></param<>	void	name='node'>the node
*/	name='node'>the node	EnterOBracket(Token	being exited
public virtual	being entered	node) {	*
Node ExitCParen(Token	being entered/param/ *	110ue/ ( }	*  * <returns>the</returns>
node) {	**	J	node to add to the
return node;		/**	parse tree, or
ictuin noue,		/ · · ·	parse tree, Or

* null	public virtual	* <param< th=""><th>public virtual</th></param<>	public virtual
if no parse tree	Node ExitGreater (Token	name='node'>the node	void EnterLessE(Token
should be	node) {	being entered	node) {
created	return node;	*	}
*	}	* <exception< td=""><td></td></exception<>	
* <exception< td=""><td></td><td><pre>cref='ParseException'&gt;</pre></td><td>/**</td></exception<>		<pre>cref='ParseException'&gt;</pre>	/**
<pre>cref='ParseException'&gt;</pre>	/**	if the node analysis	* <summary>Called</summary>
if the node analysis	* <summary>Called</summary>	* discovered	when exiting a parse
* discovered	when entering a parse	errors	tree node.
errors	tree node.	*/	*
*/	*	public virtual	* <param< td=""></param<>
public virtual	* <param< td=""><td>void</td><td>name='node'&gt;the node</td></param<>	void	name='node'>the node
Node	name='node'>the node	EnterGreaterE(Token	being exited
ExitCBracket (Token	being entered	node) {	*
node) {	*	}	* <returns>the</returns>
return node;	* <exception< td=""><td>/steste</td><td>node to add to the</td></exception<>	/steste	node to add to the
}	<pre>cref='ParseException'&gt; if the node analysis</pre>	/**	parse tree, or * null
/**	* discovered	* <summary>Called when exiting a parse</summary>	if no parse tree
* <summary>Called</summary>	errors	tree node.	should be
when entering a parse	*/	*	created
tree node.	public virtual	* <param< td=""><td>*</td></param<>	*
*	void EnterLess (Token	name='node'>the node	* <exception< td=""></exception<>
* <param< td=""><td>node) {</td><td>being exited</td><td><pre>cref='ParseException'&gt;</pre></td></param<>	node) {	being exited	<pre>cref='ParseException'&gt;</pre>
name='node'>the node	}	*	if the node analysis
being entered		* <returns>the</returns>	* discovered
*	/**	node to add to the	errors
* <exception< td=""><td>* <summary>Called</summary></td><td>parse tree, or</td><td>*/</td></exception<>	* <summary>Called</summary>	parse tree, or	*/
<pre>cref='ParseException'&gt;</pre>	when exiting a parse	* null	public virtual
if the node analysis	tree node.	if no parse tree	Node ExitLessE(Token
* discovered	*	should be	node) {
errors	* <param< td=""><td>created</td><td>return node;</td></param<>	created	return node;
*/	name='node'>the node	*	}
public virtual	being exited	* <exception< td=""><td></td></exception<>	
void	*	cref='ParseException'>	/**
EnterGreater (Token	* <returns>the</returns>	if the node analysis	* <summary>Called</summary>
node) {	node to add to the	<pre>* discovered errors</pre>	when entering a parse tree node.
J	parse tree, or * null	*/	tree node. // Summary/
/**	if no parse tree	public virtual	* <param< td=""></param<>
* <summary>Called</summary>	should be	Node	name='node'>the node
when exiting a parse	created	ExitGreaterE(Token	being entered
tree node.	*	node) {	*
*	* <exception< td=""><td>return node;</td><td>* <exception< td=""></exception<></td></exception<>	return node;	* <exception< td=""></exception<>
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>}</td><td><pre>cref='ParseException'&gt;</pre></td></param<>	<pre>cref='ParseException'&gt;</pre>	}	<pre>cref='ParseException'&gt;</pre>
name='node'>the node	if the node analysis		if the node analysis
being exited	* discovered	/**	* discovered
*	errors	* <summary>Called</summary>	errors
* <returns>the</returns>	*/	when entering a parse	*/
node to add to the	public virtual	tree node.	public virtual
parse tree, or	Node ExitLess (Token	*	void
* null	node) {	* <pre>* <pre>* </pre></pre>	EnterSObracket(Token
if no parse tree	return node;	name='node'>the node	node) {
should be created	ſ	being entered *	ſ
created\/returns/	/**	* * <exception< td=""><td>/**</td></exception<>	/**
* <exception< td=""><td>* <summary>Called</summary></td><td>cref='ParseException'&gt;</td><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>	cref='ParseException'>	* <summary>Called</summary>
cref='ParseException'>	when entering a parse	if the node analysis	when exiting a parse
if the node analysis	tree node.	* discovered	tree node.
* discovered	*	errors	*
errors		*/	

N. /	ate.		. did
* <param name="node"/> the node	* * <exception< td=""><td>/**</td><td>* discovered errors</td></exception<>	/**	* discovered errors
being exited	cref='ParseException'>	* <summary>Called</summary>	*/
*	if the node analysis	when entering a parse	public virtual
* <returns>the</returns>	* discovered	tree node.	void EnterHash(Token
node to add to the	errors	*	node) {
parse tree, or	*/	* <param< td=""><td>node) (</td></param<>	node) (
* null	public virtual	name='node'>the node	J
if no parse tree	Node	being entered	/**
should be	ExitSCbracket (Token	*	* <summary>Called</summary>
created	node) {	* <exception< td=""><td>when exiting a parse</td></exception<>	when exiting a parse
*	return node;	cref='ParseException'>	tree node.
* <exception< td=""><td>}</td><td>if the node analysis</td><td>*</td></exception<>	}	if the node analysis	*
cref='ParseException'>	,	* discovered	* <param< td=""></param<>
if the node analysis	/**	errors	name='node'>the node
* discovered	* <summary>Called</summary>	*/	being exited
errors	when entering a parse	public virtual	*
*/	tree node.	void EnterPower(Token	* <returns>the</returns>
public virtual	*	node) {	node to add to the
Node	* <param< td=""><td>}</td><td>parse tree, or</td></param<>	}	parse tree, or
ExitSObracket(Token	name='node'>the node		* null
node) {	being entered	/**	if no parse tree
return node;	*	* <summary>Called</summary>	should be
}	* <exception< td=""><td>when exiting a parse</td><td>created</td></exception<>	when exiting a parse	created
	<pre>cref='ParseException'&gt;</pre>	tree node.	*
/**	if the node analysis	*	* <exception< td=""></exception<>
* <summary>Called</summary>	* discovered	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td></param<>	<pre>cref='ParseException'&gt;</pre>
when entering a parse	errors	name='node'>the node	if the node analysis
tree node.	*/	being exited	* discovered
*	public virtual	*	errors
* <param< td=""><td>void EnterDollar(Token</td><td>* <returns>the</returns></td><td>*/</td></param<>	void EnterDollar(Token	* <returns>the</returns>	*/
name='node'>the node	node) {	node to add to the	public virtual
being entered	}	parse tree, or	Node ExitHash (Token
*		* null	node) {
* <exception< td=""><td>/**</td><td>if no parse tree</td><td>return node;</td></exception<>	/**	if no parse tree	return node;
<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	should be	}
if the node analysis	when exiting a parse	created	
* discovered	tree node.	*	/**
errors	*	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
*/	* <param< td=""><td>cref='ParseException'&gt;</td><td>when entering a parse</td></param<>	cref='ParseException'>	when entering a parse
public virtual	name='node'>the node	if the node analysis	tree node.
void	being exited	* discovered	*
EnterSCbracket (Token	*	errors	* <param< td=""></param<>
node) {	* <returns>the</returns>	*/	name='node'>the node
}	node to add to the	public virtual	being entered
/state	parse tree, or * null	Node ExitPower (Token	*
/**		node) {	* <exception< td=""></exception<>
* <summary>Called</summary>	if no parse tree should be	return node;	cref='ParseException'>
when exiting a parse tree node.	created	ſ	if the node analysis * discovered
*	*	/**	errors
* <param< td=""><td>* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td></exception<></td></param<>	* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td></exception<>	* <summary>Called</summary>	*/
name='node'>the node	cref='ParseException'>	when entering a parse	public virtual
being exited	if the node analysis	tree node.	void EnterNega(Token
*	* discovered	*	node) {
* <returns>the</returns>	errors	* <param< td=""><td>}</td></param<>	}
node to add to the	*/	name='node'>the node	j
parse tree, or	public virtual	being entered	/**
* null	Node ExitDollar(Token	*	* <summary>Called</summary>
if no parse tree	node) {	* <exception< td=""><td>when exiting a parse</td></exception<>	when exiting a parse
should be	return node;	cref='ParseException'>	tree node.
created	}	if the node analysis	*
		•	

. /			. /
* <param name='node'&gt;the node</param 	<pre>* <exception cref="ParseException"></exception></pre>	* <summary>Called</summary>	*/ public virtual
being exited	if the node analysis	when entering a parse tree node.	void EnterString(Token
*	* discovered	*	node) {
* <returns>the</returns>	errors	* <param< td=""><td>}</td></param<>	}
node to add to the	*/	name='node'>the node	,
parse tree, or	public virtual	being entered	/**
* null	Node ExitInt(Token	*	* <summary>Called</summary>
if no parse tree	node) {	* <exception< td=""><td>when exiting a parse</td></exception<>	when exiting a parse
should be	return node;	<pre>cref='ParseException'&gt;</pre>	tree node.
created	}	if the node analysis	*
*		* discovered	* <param< td=""></param<>
* <exception< td=""><td>/**</td><td>errors</td><td>name='node'&gt;the node</td></exception<>	/**	errors	name='node'>the node
cref='ParseException'>	* <summary>Called</summary>	*/	being exited
if the node analysis	when entering a parse	public virtual	*
<pre>* discovered errors</pre>	tree node.	<pre>void EnterFloat(Token node) {</pre>	* <returns>the node to add to the</returns>
*/	* <param< td=""><td>noue) (</td><td>parse tree, or</td></param<>	noue) (	parse tree, or
public virtual	name='node'>the node	J	* null
Node ExitNega (Token	being entered	/**	if no parse tree
node) {	*	* <summary>Called</summary>	should be
return node;	* <exception< td=""><td>when exiting a parse</td><td>created</td></exception<>	when exiting a parse	created
}	<pre>cref='ParseException'&gt;</pre>	tree node.	*
	if the node analysis	*	* <exception< td=""></exception<>
/**	* discovered	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td></param<>	<pre>cref='ParseException'&gt;</pre>
* <summary>Called</summary>	errors	name='node'>the node	if the node analysis
when entering a parse	*/	being exited	* discovered
tree node.	public virtual	*	errors
*	void EnterChar(Token	* <returns>the</returns>	*/
* <param name='node'&gt;the node</param 	node) {	node to add to the	public virtual Node ExitString(Token
being entered	l	parse tree, or * null	node) {
*	/**	if no parse tree	return node;
* <exception< td=""><td>* <summary>Called</summary></td><td>should be</td><td>}</td></exception<>	* <summary>Called</summary>	should be	}
<pre>cref='ParseException'&gt;</pre>	when exiting a parse	created	,
if the node analysis	tree node.	*	/**
* discovered	*	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
errors	* <param< td=""><td>cref='ParseException'&gt;</td><td>when entering a parse</td></param<>	cref='ParseException'>	when entering a parse
*/	name='node'>the node	if the node analysis	tree node.
public virtual	being exited	* discovered	*
void EnterInt (Token	*	errors	* <param< td=""></param<>
node) {	* <returns>the</returns>	*/	name='node'>the node
}	node to add to the	public virtual	being entered
/**	parse tree, or * null	Node ExitFloat(Token node) {	* * <exception< td=""></exception<>
* <summary>Called</summary>	if no parse tree	return node;	cref='ParseException'>
when exiting a parse	should be	}	if the node analysis
tree node.	created	,	* discovered
*	*	/**	errors
* <param< td=""><td>* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td></exception<></td></param<>	* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td></exception<>	* <summary>Called</summary>	*/
name='node'>the node	<pre>cref='ParseException'&gt;</pre>	when entering a parse	public virtual
being exited	if the node analysis	tree node.	void EnterBoolN(Token
*	* discovered	*	node) {
* <returns>the</returns>	errors	* <pre>* <pre><pre></pre></pre></pre>	}
node to add to the	*/	name='node'>the node	/·
parse tree, or	public virtual	being entered	/**
* null	Node ExitChar (Token	*	* <summary>Called</summary>
if no parse tree should be	node) { return node;	<pre>* <exception cref="ParseException"></exception></pre>	when exiting a parse tree node.
created	leturn noue,	if the node analysis	*
*	,	* discovered	•
	/**	errors	
		•	

No. /	w /	th. /	
* <param name='node'&gt;the node</param 	<pre>* <exception cref="ParseException"></exception></pre>	* <summary>Called</summary>	<pre>* discovered errors</pre>
being exited	if the node analysis	when entering a parse tree node.	*/
*	* discovered	tree node. // Summary/	public virtual
* <returns>the</returns>	errors	* <param< td=""><td>void EnterSChar(Token</td></param<>	void EnterSChar(Token
node to add to the	*/	name='node'>the node	node) {
parse tree, or	public virtual	being entered	loue) (
* null	Node ExitId(Token	*	J
if no parse tree	node) {	* <exception< td=""><td>/**</td></exception<>	/**
should be	return node;	cref='ParseException'>	* <summary>Called</summary>
created	}	if the node analysis	when exiting a parse
*	,	* discovered	tree node.
* <exception< td=""><td>/**</td><td>errors</td><td>*</td></exception<>	/**	errors	*
<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	*/	* <param< td=""></param<>
if the node analysis	when entering a parse	public virtual	name='node'>the node
* discovered	tree node.	void	being exited
errors	*	EnterDecimal(Token	*
*/	* <param< td=""><td>node) {</td><td>* <returns>the</returns></td></param<>	node) {	* <returns>the</returns>
public virtual	name='node'>the node	}	node to add to the
Node ExitBoolN(Token	being entered		parse tree, or
node) {	*	/**	* null
return node;	* <exception< td=""><td>* <summary>Called</summary></td><td>if no parse tree</td></exception<>	* <summary>Called</summary>	if no parse tree
}	<pre>cref='ParseException'&gt;</pre>	when exiting a parse	should be
	if the node analysis	tree node.	created
/**	* discovered	*	*
* <summary>Called</summary>	errors	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
when entering a parse	*/	name='node'>the node	cref='ParseException'>
tree node.	public virtual	being exited	if the node analysis
*	void EnterNum(Token	*	* discovered
* <param< td=""><td>node) {</td><td>* <returns>the</returns></td><td>errors</td></param<>	node) {	* <returns>the</returns>	errors
name='node'>the node	}	node to add to the	*/
being entered	,	parse tree, or	public virtual
*	/**	* null	Node ExitSChar(Token
* <exception< td=""><td>* <summary>Called</summary></td><td>if no parse tree</td><td>node) {</td></exception<>	* <summary>Called</summary>	if no parse tree	node) {
cref='ParseException'>	when exiting a parse	should be	return node;
if the node analysis	tree node.	created	}
* discovered	*	*	/**
errors */	* <param name='node'&gt;the node</param 	<pre>* <exception cref="ParseException"></exception></pre>	/** * <summary>Called</summary>
*/ public virtual	being exited	•	·
void EnterId(Token	*	if the node analysis * discovered	when entering a parse tree node.
node) {	* <returns>the</returns>	errors	*
}	node to add to the	*/	* <param< td=""></param<>
J	parse tree, or	public virtual	name='node'>the node
/**	* null	Node ExitDecimal (Token	being entered
* <summary>Called</summary>	if no parse tree	node) {	*
when exiting a parse	should be	return node;	* <exception< td=""></exception<>
tree node.	created	}	<pre>cref='ParseException'&gt;</pre>
*	*	,	if the node analysis
* <param< td=""><td>* <exception< td=""><td>/**</td><td>* discovered</td></exception<></td></param<>	* <exception< td=""><td>/**</td><td>* discovered</td></exception<>	/**	* discovered
name='node'>the node	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	errors
being exited	if the node analysis	when entering a parse	*/
*	* discovered	tree node.	public virtual
* <returns>the</returns>	errors	*	void EnterText(Token
node to add to the	*/	* <param< td=""><td>node) {</td></param<>	node) {
parse tree, or	public virtual	name='node'>the node	}
* null	Node ExitNum(Token	being entered	
if no parse tree	node) {	*	/**
should be	return node;	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
created	}	cref='ParseException'>	when exiting a parse
*	,	if the node analysis	tree node.
	/**		*

* <param< td=""><td>* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td></exception<></td></param<>	* <exception< td=""><td>* <summary>Called</summary></td><td>*/</td></exception<>	* <summary>Called</summary>	*/
name='node'>the node	cref='ParseException'>	when entering a parse	public virtual
being exited	if the node analysis	tree node.	void
*	* discovered	*	EnterFunctname (Token
* <returns>the</returns>	errors	* <param '.="" -'.="" 1="" 1<="" td=""/> <td>node) {</td>	node) {
node to add to the	*/	name='node'>the node	}
parse tree, or * null	public virtual Node ExitCom(Token	being entered *	/**
	node cxitcom(token node) {		,
if no parse tree should be	return node;	<pre>* <exception cref="ParseException"></exception></pre>	<pre>* <summary>Called when exiting a parse</summary></pre>
created	return node,	if the node analysis	tree node.
*	J	* discovered	*
* <exception< td=""><td>/**</td><td>errors</td><td>* <param< td=""></param<></td></exception<>	/**	errors	* <param< td=""></param<>
cref='ParseException'>	* <summary>Called</summary>	*/	name='node'>the node
if the node analysis	when entering a parse	public virtual	being exited
* discovered	tree node.	void EnterNo(Token	*
errors	*	node) {	* <returns>the</returns>
*/	* <param< td=""><td>}</td><td>node to add to the</td></param<>	}	node to add to the
public virtual	name='node'>the node	,	parse tree, or
Node ExitText (Token	being entered	/**	* null
node) {	*	* <summary>Called</summary>	if no parse tree
return node;	* <exception< td=""><td>when exiting a parse</td><td>should be</td></exception<>	when exiting a parse	should be
}	cref='ParseException'>	tree node.	created
,	if the node analysis	*	*
/**	* discovered	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
* <summary>Called</summary>	errors	name='node'>the node	cref='ParseException'>
when entering a parse	*/	being exited	if the node analysis
tree node.	public virtual	*	* discovered
*	void EnterYes(Token	* <returns>the</returns>	errors
* <param< td=""><td>node) {</td><td>node to add to the</td><td>*/</td></param<>	node) {	node to add to the	*/
name='node'>the node	}	parse tree, or	public virtual
being entered		* null	Node
*	/**	if no parse tree	ExitFunctname(Token
* <exception< td=""><td>* <summary>Called</summary></td><td>should be</td><td>node) {</td></exception<>	* <summary>Called</summary>	should be	node) {
<pre>cref='ParseException'&gt;</pre>	when exiting a parse	created	return node;
if the node analysis	tree node.	*	}
* discovered	*	* <exception< td=""><td></td></exception<>	
errors	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>/**</td></param<>	<pre>cref='ParseException'&gt;</pre>	/**
*/	name='node'>the node	if the node analysis	* <summary>Called</summary>
public virtual	being exited	* discovered	when entering a parse
void EnterCom(Token	*	errors	tree node.
node) {	* <returns>the</returns>	*/	*
}	node to add to the	public virtual	* <param< td=""></param<>
,	parse tree, or	Node ExitNo(Token	name='node'>the node
/**	* null	node) {	being entered
* <summary>Called</summary>	if no parse tree	return node;	*
when exiting a parse	should be	}	* <exception< td=""></exception<>
tree node.	created		cref='ParseException'>
*	*	/**	if the node analysis
* <param< td=""><td>* <exception< td=""><td>* <summary>Called</summary></td><td>* discovered</td></exception<></td></param<>	* <exception< td=""><td>* <summary>Called</summary></td><td>* discovered</td></exception<>	* <summary>Called</summary>	* discovered
name='node'>the node	cref='ParseException'>	when entering a parse	errors
being exited	if the node analysis	tree node.	*/
* * <returns>the</returns>	<pre>* discovered errors</pre>	* * <param< td=""><td>public virtual void</td></param<>	public virtual void
* <returns td="" the<=""><td>errors */</td><td>name='node'&gt;the node</td><td>Volu EnterStructname(Token</td></returns>	errors */	name='node'>the node	Volu EnterStructname(Token
parse tree, or	*/ public virtual	being entered	node) {
parse tree, or null	Node ExitYes(Token	*	Houe) (
if no parse tree	node Exities (lokeli node) {	* <exception< td=""><td>J</td></exception<>	J
should be	return node;	cref='ParseException'>	/**
created	}	if the node analysis	* <summary>Called</summary>
*	J	* discovered	when exiting a parse
•	/**	errors	tree node.
	*	// 01100P 01011/	,

*	should be	return node;	* <exception< th=""></exception<>
* <param< td=""><td>created</td><td>return node,</td><td>cref='ParseException'&gt;</td></param<>	created	return node,	cref='ParseException'>
name='node'>the node	created\/returns/	ſ	*
		/**	if the node analysis
being exited	* <exception< td=""><td>,</td><td>* discovered</td></exception<>	,	* discovered
*	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	errors
* <returns>the</returns>	if the node analysis	when entering a parse	*/
node to add to the	* discovered	tree node.	public virtual
parse tree, or	errors	*	void EnterS(Token
* null	*/	* <param< td=""><td>node) {</td></param<>	node) {
if no parse tree	public virtual	name='node'>the node	}
should be	Node	being entered	
created	ExitIdstruct(Token	*	/**
*	node) {	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
* <exception< td=""><td>return node;</td><td>cref='ParseException'&gt;</td><td>when exiting a parse</td></exception<>	return node;	cref='ParseException'>	when exiting a parse
cref='ParseException'>	}	if the node analysis	tree node.
if the node analysis		* discovered	*
* discovered	/**	errors	* <param< td=""></param<>
errors	* <summary>Called</summary>	*/	name='node'>the node
*/	when entering a parse	public virtual	being exited
public virtual	tree node.	void EnterD(Token	*
Node	*	node) {	* <returns>the</returns>
ExitStructname(Token	* <param< td=""><td>}</td><td>node to add to the</td></param<>	}	node to add to the
node) {	name='node'>the node		parse tree, or
return node;	being entered	/**	* null
}	*	* <summary>Called</summary>	if no parse tree
	* <exception< td=""><td>when exiting a parse</td><td>should be</td></exception<>	when exiting a parse	should be
/**	cref='ParseException'>	tree node.	created
* <summary>Called</summary>	if the node analysis	*	*
when entering a parse	* discovered	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
tree node.	errors	name='node'>the node	<pre>cref='ParseException'&gt;</pre>
*	*/	being exited	if the node analysis
* <param< td=""><td>public virtual</td><td>*</td><td>* discovered</td></param<>	public virtual	*	* discovered
name='node'>the node	void EnterF(Token	* <returns>the</returns>	errors
being entered	node) {	node to add to the	*/
*	}	parse tree, or	public virtual
* <exception< td=""><td></td><td>* null</td><td>Node ExitS(Token node)</td></exception<>		* null	Node ExitS(Token node)
cref='ParseException'>	/**	if no parse tree	{
if the node analysis	* <summary>Called</summary>	should be	return node;
* discovered	when exiting a parse	created	}
errors	tree node.	*	,
*/	*	* <exception< td=""><td>/**</td></exception<>	/**
public virtual	* <param< td=""><td>cref='ParseException'&gt;</td><td>* <summary>Called</summary></td></param<>	cref='ParseException'>	* <summary>Called</summary>
void	name='node'>the node	if the node analysis	when entering a parse
EnterIdstruct(Token	being exited	* discovered	tree node.
node) {	*	errors	*
}	* <returns>the</returns>	*/	* <param< td=""></param<>
,	node to add to the	public virtual	name='node'>the node
/**	parse tree, or	Node ExitD(Token node)	being entered
* <summary>Called</summary>	* null	{	*
when exiting a parse	if no parse tree	return node;	* <exception< td=""></exception<>
tree node.	should be	}	<pre>cref='ParseException'&gt;</pre>
*	created	,	if the node analysis
* <param< td=""><td>*</td><td>/**</td><td>* discovered</td></param<>	*	/**	* discovered
name='node'>the node	* <exception< td=""><td>* <summary>Called</summary></td><td>errors</td></exception<>	* <summary>Called</summary>	errors
being exited	cref='ParseException'>	when entering a parse	*/
*	if the node analysis	tree node.	public virtual
* <returns>the</returns>	* discovered	*	void EnterZero(Token
node to add to the	errors	* <param< td=""><td>node) {</td></param<>	node) {
parse tree, or	*/	name='node'>the node	}
* null	public virtual	being entered	J
if no parse tree	Node ExitF(Token node)	*	/**
ii no parse tree	f (lovell linde)	-p-	/ -mr
	ί		

di / \0.11.1	.1111		
* <summary>Called</summary>	should be	return node;	*
when exiting a parse	created	}	* <exception< td=""></exception<>
tree node.	*	(state	cref='ParseException'>
*	* <exception< td=""><td>/**</td><td>if the node analysis</td></exception<>	/**	if the node analysis
* <pre></pre>	cref='ParseException'>	* <summary>Called</summary>	* discovered
name='node'>the node	if the node analysis	when entering a parse	errors
being exited *	* discovered	tree node.	*/
	errors	*	public virtual
* <returns>the</returns>	*/	* <param name="node"/> the node	void EnterReverse(Token
node to add to the	public virtual Node ExitTochar(Token		node) {
parse tree, or * null		being entered *	node) (
11011	node) {	·	}
if no parse tree should be	return node;	<pre>* <exception cref="ParseException"></exception></pre>	/**
created	ſ		,
createu/returns/	/**	if the node analysis * discovered	* <summary>Called</summary>
* <exception< td=""><td>,</td><td>* discovered errors</td><td>when exiting a parse</td></exception<>	,	* discovered errors	when exiting a parse
cref='ParseException'>	* <summary>Called</summary>	*/	tree node.
if the node analysis	when entering a parse tree node.	public virtual	* <param< td=""></param<>
* discovered	*	void	name='node'>the node
errors	* <param< td=""><td>EnterContains (Token</td><td>being exited</td></param<>	EnterContains (Token	being exited
*/	name='node'>the node	node) {	being exited\/param/
public virtual	being entered	node) (	* <returns>the</returns>
Node ExitZero(Token	*	J	node to add to the
node Exitzero (Token node) {	* <exception< td=""><td>/**</td><td></td></exception<>	/**	
return node:	cref='ParseException'>	* <summary>Called</summary>	parse tree, or * null
leturn node,	if the node analysis	when exiting a parse	if no parse tree
J	* discovered	tree node.	should be
/**	errors	tree node. // Summary/	created
* <summary>Called</summary>	*/	* <param< td=""><td>*</td></param<>	*
when entering a parse	public virtual	name='node'>the node	* <exception< td=""></exception<>
tree node.	void	being exited	cref='ParseException'>
*	EnterLengthf (Token	*	if the node analysis
* <param< td=""><td>node) {</td><td>* <returns>the</returns></td><td>* discovered</td></param<>	node) {	* <returns>the</returns>	* discovered
name='node'>the node	node) (	node to add to the	errors
being entered	J	parse tree, or	*/
*	/**	* null	public virtual
* <exception< td=""><td>* <summary>Called</summary></td><td>if no parse tree</td><td>Node ExitReverse (Token</td></exception<>	* <summary>Called</summary>	if no parse tree	Node ExitReverse (Token
cref='ParseException'>	when exiting a parse	should be	node) {
if the node analysis	tree node.	created	return node;
* discovered	*	*	leturn node,
errors	* <param< td=""><td>* <exception< td=""><td>J</td></exception<></td></param<>	* <exception< td=""><td>J</td></exception<>	J
*/	name='node'>the node	cref='ParseException'>	/**
public virtual	being exited	if the node analysis	* <summary>Called</summary>
void EnterTochar(Token	*	* discovered	when entering a parse
node) {	* <returns>the</returns>	errors	tree node.
}	node to add to the	*/	*
J	parse tree, or	public virtual	* <param< td=""></param<>
/**	* null	Node	name='node'>the node
* <summary>Called</summary>	if no parse tree	ExitContains (Token	being entered
when exiting a parse	should be	node) {	*
tree node.	created	return node;	* <exception< td=""></exception<>
*	*	}	cref='ParseException'>
* <param< td=""><td>* <exception< td=""><td>,</td><td>if the node analysis</td></exception<></td></param<>	* <exception< td=""><td>,</td><td>if the node analysis</td></exception<>	,	if the node analysis
name='node'>the node	cref='ParseException'>	/**	* discovered
being exited	if the node analysis	* <summary>Called</summary>	errors
*	* discovered	when entering a parse	*/
* <returns>the</returns>	errors	tree node.	public virtual
node to add to the	*/	*	void
parse tree, or	public virtual	* <param< td=""><td>EnterProdStartProgram(</td></param<>	EnterProdStartProgram(
* null	Node ExitLengthf (Token	name='node'>the node	Production node) {
if no parse tree	node) {	being entered (/param)	}
II no parso tree	11040/	borns officerous // paramy	J

	* <summary>Called</summary>	*	ExitProdClear(Producti
/**	when entering a parse	* <exception< td=""><td>on node) {</td></exception<>	on node) {
* <summary>Called</summary>	tree node.	<pre>cref='ParseException'&gt;</pre>	return node;
when exiting a parse	*	if the node analysis	}
tree node.	* <param< td=""><td>* discovered</td><td></td></param<>	* discovered	
*	name='node'>the node	errors	/**
* <param< td=""><td>being entered</td><td>*/</td><td>* <summary>Called</summary></td></param<>	being entered	*/	* <summary>Called</summary>
name='node'>the node	*	public virtual	when adding a child to
being exited	* <exception< td=""><td>void</td><td>a parse tree</td></exception<>	void	a parse tree
*	<pre>cref='ParseException'&gt;</pre>	ChildProdProgram(Produ	* node.
* <returns>the</returns>	if the node analysis	ction node, Node	*
node to add to the	* discovered	child) {	* <param< td=""></param<>
parse tree, or	errors		name='node'>the parent
* null	*/	node.AddChild(child);	node
if no parse tree	public virtual	}	* <param< td=""></param<>
should be	void		name='child'>the child
created	EnterProdProgram(Produ	/**	node, or null
*	ction node) {	* <summary>Called</summary>	*
* <exception< td=""><td>}</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	}	when entering a parse	* <exception< td=""></exception<>
<pre>cref='ParseException'&gt;</pre>	,	tree node.	<pre>cref='ParseException'&gt;</pre>
if the node analysis	/**	*	if the node analysis
* discovered	* <summary>Called</summary>	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	when exiting a parse	name='node'>the node	errors
*/	tree node.	being entered	*/
public virtual	*	*	public virtual
Node	* <pre>* <pre>* </pre></pre>	* <exception< td=""><td>void</td></exception<>	void
ExitProdStartProgram(P	name='node'>the node	cref='ParseException'>	ChildProdClear(Product
roduction node) {	being exited *	if the node analysis	ion node, Node child)
return node;	•	* discovered	1
}	* <returns>the</returns>	errors */	AlaCh:11/-h:11/.
/**	node to add to the	*/ public virtual	node.AddChild(child);
* <summary>Called</summary>	parse tree, or * null	void	J
when adding a child to	if no parse tree	EnterProdClear(Product	/**
a parse tree	should be	ion node) {	* <summary>Called</summary>
* node.	created	}	when entering a parse
*	*	,	tree node.
* <param< td=""><td>* <exception< td=""><td>/**</td><td>*</td></exception<></td></param<>	* <exception< td=""><td>/**</td><td>*</td></exception<>	/**	*
name='node'>the parent	cref='ParseException'>	* <summary>Called</summary>	* <param< td=""></param<>
node	if the node analysis	when exiting a parse	name='node'>the node
* <param< td=""><td>* discovered</td><td>tree node. </td><td>being entered</td></param<>	* discovered	tree node.	being entered
name='child'>the child	errors	*	*
node, or null	*/	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
*	public virtual	name='node'>the node	cref='ParseException'>
* <exception< td=""><td>Node</td><td>being exited</td><td>if the node analysis</td></exception<>	Node	being exited	if the node analysis
<pre>cref='ParseException'&gt;</pre>	ExitProdProgram(Produc	*	* discovered
if the node analysis	tion node) {	* <returns>the</returns>	errors
* discovered	return node;	node to add to the	*/
errors	}	parse tree, or	public virtual
*/		* null	void
public virtual	/**	if no parse tree	EnterProdComments(Prod
void	* <summary>Called</summary>	should be	uction node) {
ChildProdStartProgram(	when adding a child to	created	}
Production node, Node	a parse tree	*	
child) {	* node.	* <exception< td=""><td>/**</td></exception<>	/**
	*	cref='ParseException'>	* <summary>Called</summary>
node.AddChild(child);	* <pre>* <pre> * <pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>	if the node analysis	when exiting a parse
}	name='node'>the parent	* discovered	tree node.
	node	errors	*
/**	* <pre>* <pre><pre></pre></pre></pre>	*/	* <param< td=""></param<>
	name='child'>the child	public virtual	name='node'>the node
	node, or null	Node	being exited

node, or null</param>

Node

being exited</param>

*	* discovered	tion node, Node child)	*
* <returns>the</returns>	errors	{	* <param< td=""></param<>
node to add to the	*/		name='node'>the parent
parse tree, or	public virtual	<pre>node. AddChild(child);</pre>	node
* null	void	}	* <param< td=""></param<>
if no parse tree	EnterProdNegate(Produc		name='child'>the child
should be	tion node) {	/**	node, or null
created	}	* <summary>Called</summary>	*
*		when entering a parse	* <exception< td=""></exception<>
* <exception< td=""><td>/**</td><td>tree node. </td><td>cref='ParseException'&gt;</td></exception<>	/**	tree node.	cref='ParseException'>
cref='ParseException'>	* <summary>Called</summary>	*	if the node analysis
if the node analysis	when exiting a parse	* <param< td=""><td>* discovered</td></param<>	* discovered
* discovered	tree node.	name='node'>the node	errors
errors	*	being entered	*/
*/	* <param< td=""><td>*</td><td>public virtual</td></param<>	*	public virtual
public virtual	name='node'>the node	* <exception< td=""><td>void</td></exception<>	void
Node	being exited	cref='ParseException'>	ChildProdDatatype(Prod
ExitProdComments(Produ	*	if the node analysis	uction node, Node
ction node) {	* <returns>the</returns>	* discovered	child) {
return node;	node to add to the	errors	
}	parse tree, or	*/	node.AddChild(child);
	* null	public virtual	}
/**	if no parse tree	void	,
* <summary>Called</summary>	should be	EnterProdDatatype (Prod	/**
when adding a child to	created	uction node) {	* <summary>Called</summary>
a parse tree	*	}	when entering a parse
* node.	* <exception< td=""><td>/</td><td>tree node.</td></exception<>	/	tree node.
*	<pre>cref='ParseException'&gt;</pre>	/**	*
* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td><td>* <param< td=""></param<></td></param<>	if the node analysis	* <summary>Called</summary>	* <param< td=""></param<>
name='node'>the parent	* discovered	when exiting a parse	name='node'>the node
node * <param< td=""><td>errors */</td><td>tree node.</td><td>being entered *</td></param<>	errors */	tree node.	being entered *
name='child'>the child	public virtual	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
node, or null	Node	name='node'>the node	cref='ParseException'>
noue, or nurry param/	Node ExitProdNegate(Product	being exited	if the node analysis
* <exception< td=""><td>ion node) {</td><td>*</td><td>* discovered</td></exception<>	ion node) {	*	* discovered
cref='ParseException'>	return node;	* <returns>the</returns>	errors
if the node analysis	}	node to add to the	*/
* discovered	,	parse tree, or	public virtual
errors	/**	* null	void
*/	* <summary>Called</summary>	if no parse tree	EnterProdLiterals(Prod
public virtual	when adding a child to	should be	uction node) {
void	a parse tree	created	}
ChildProdComments(Prod	* node.	*	
uction node, Node	*	* <exception< td=""><td>/**</td></exception<>	/**
child) {	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>* <summary>Called</summary></td></param<>	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>
	name='node'>the parent	if the node analysis	when exiting a parse
<pre>node. AddChild(child);</pre>	node	* discovered	tree node.
}	* <param< td=""><td>errors</td><td>*</td></param<>	errors	*
	name='child'>the child	*/	* <param< td=""></param<>
/**	node, or null	public virtual	name='node'>the node
* <summary>Called</summary>	*	Node	being exited
when entering a parse	* <exception< td=""><td>ExitProdDatatype (Produ</td><td>*</td></exception<>	ExitProdDatatype (Produ	*
tree node.	cref='ParseException'>	ction node) {	* <returns>the</returns>
*	if the node analysis	return node;	node to add to the
* <param< td=""><td>* discovered</td><td>}</td><td>parse tree, or</td></param<>	* discovered	}	parse tree, or
name='node'>the node	errors	/state	* null
being entered	*/	/**	if no parse tree
* / 0.000	public virtual	* <summary>Called</summary>	should be
* <exception< td=""><td>void</td><td>when adding a child to</td><td>created</td></exception<>	void	when adding a child to	created
<pre>cref='ParseException'&gt; if the node analysis</pre>	ChildProdNegate(Produc	a parse tree * node.	•
ii the houe analysis		• noue. // summary/	

* /avaantian	/**	*	* /oveentien
<pre>* <exception cref="ParseException"></exception></pre>	,		<pre>* <exception cref="ParseException"></exception></pre>
if the node analysis	* <summary>Called</summary>	* <param name="node"/> the node	if the node analysis
* discovered	when exiting a parse tree node.		* discovered
errors (/exception)	tree node. // Summary/	being entered	errors
errors\/exception/ */	*	* /	
,	* <param '.="" -'.="" 1="" 1<="" td=""/> <td>* <exception< td=""><td>*/</td></exception<></td>	* <exception< td=""><td>*/</td></exception<>	*/
public virtual	name='node'>the node	cref='ParseException'>	public virtual
Node	being exited	if the node analysis	void
ExitProdLiterals(Produ	*	* discovered	ChildProdGlobalDec(Pro
ction node) {	* <returns>the</returns>	errors */	duction node, Node child) {
return node;	node to add to the	*/ public virtual	Child) (
3	parse tree, or * null	•	1- A11CL:11(-L:11).
/**	* null if no parse tree	void EnterProdGlobalDec(Pro	node. AddChild(child);
,	•	•	l
* <summary>Called</summary>	should be	duction node) {	/ state
when adding a child to	created *	}	/**
a parse tree		/steale	* <summary>Called</summary>
* node.	<pre>* <exception cref="ParseException"></exception></pre>	/**	when entering a parse
*		* <summary>Called</summary>	tree node.
* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td><td>*</td></param<>	if the node analysis	when exiting a parse	*
name='node'>the parent	* discovered	tree node.	* <param< td=""></param<>
node	errors	*	name='node'>the node
* <param< td=""><td>*/</td><td>* <param< td=""><td>being entered</td></param<></td></param<>	*/	* <param< td=""><td>being entered</td></param<>	being entered
name='child'>the child	public virtual	name='node'>the node	*
node, or null	Node	being exited	* <exception< td=""></exception<>
*	ExitProdLiterals2(Prod	*	cref='ParseException'>
* <exception< td=""><td>uction node) {</td><td>* <returns>the</returns></td><td>if the node analysis</td></exception<>	uction node) {	* <returns>the</returns>	if the node analysis
<pre>cref='ParseException'&gt;</pre>	return node;	node to add to the	* discovered
if the node analysis	}	parse tree, or	errors
* discovered	,	* null	*/
errors	/**	if no parse tree	public virtual
*/	* <summary>Called</summary>	should be	void
public virtual	when adding a child to	created	EnterProdDeclare(Produ
void	a parse tree	*	ction node) {
ChildProdLiterals(Prod	* node.	* <exception< td=""><td>}</td></exception<>	}
uction node, Node	*	cref='ParseException'>	
child) {	* <param< td=""><td>if the node analysis</td><td>/**</td></param<>	if the node analysis	/**
	name='node'>the parent	* discovered	* <summary>Called</summary>
<pre>node. AddChild(child);</pre>	node	errors	when exiting a parse
}	* <param< td=""><td>*/</td><td>tree node.</td></param<>	*/	tree node.
	name='child'>the child	public virtual	*
/**	node, or null	Node	* <param< td=""></param<>
* <summary>Called</summary>	*	ExitProdGlobalDec(Prod	name='node'>the node
when entering a parse	* <exception< td=""><td>uction node) {</td><td>being exited</td></exception<>	uction node) {	being exited
tree node.	cref='ParseException'>	return node;	*
*	if the node analysis	}	* <returns>the</returns>
* <param< td=""><td>* discovered</td><td></td><td>node to add to the</td></param<>	* discovered		node to add to the
name='node'>the node	errors	/**	parse tree, or
being entered	*/	* <summary>Called</summary>	* null
*	public virtual	when adding a child to	if no parse tree
* <exception< td=""><td>void</td><td>a parse tree</td><td>should be</td></exception<>	void	a parse tree	should be
<pre>cref='ParseException'&gt;</pre>	ChildProdLiterals2(Pro	* node.	created
if the node analysis	duction node, Node	*	*
* discovered	child) {	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
errors		name='node'>the parent	<pre>cref='ParseException'&gt;</pre>
*/	<pre>node. AddChild(child);</pre>	node	if the node analysis
public virtual	}	* <param< td=""><td>* discovered</td></param<>	* discovered
void		name='child'>the child	errors
EnterProdLiterals2(Pro	/**	node, or null	*/
duction node) {	* <summary>Called</summary>	*	public virtual
}	when entering a parse		Node
	tree node.		

		. 1	1 (* 1 N 1
ExitProdDeclare(Production node) {	* * <returns>the</returns>	<pre>* discovered errors</pre>	oduction node, Node child) {
return node;	node to add to the	*/	ciiiu) (
return node,	parse tree, or	public virtual	node.AddChild(child);
,	* null	void	}
/**	if no parse tree	EnterProdInitChoice(Pr	,
* <summary>Called</summary>	should be	oduction node) {	/**
when adding a child to	created	}	* <summary>Called</summary>
a parse tree	*	,	when entering a parse
* node.	* <exception< td=""><td>/**</td><td>tree node. </td></exception<>	/**	tree node.
*	cref='ParseException'>	* <summary>Called</summary>	*
* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td><td>* <param< td=""></param<></td></param<>	if the node analysis	when exiting a parse	* <param< td=""></param<>
name='node'>the parent	* discovered	tree node.	name='node'>the node
node	errors	*	being entered
* <param< td=""><td>*/</td><td>* <param< td=""><td>*</td></param<></td></param<>	*/	* <param< td=""><td>*</td></param<>	*
name='child'>the child	public virtual	name='node'>the node	* <exception< td=""></exception<>
node, or null	Node	being exited	<pre>cref='ParseException'&gt;</pre>
*	ExitProdDeclareChoice(	*	if the node analysis
* <exception< td=""><td>Production node) {</td><td>* <returns>the</returns></td><td>* discovered</td></exception<>	Production node) {	* <returns>the</returns>	* discovered
cref='ParseException'>	return node;	node to add to the	errors
if the node analysis	}	parse tree, or	*/
* discovered		* null	public virtual
errors	/**	if no parse tree	void
*/	* <summary>Called</summary>	should be created	EnterProdAddId(Product
public virtual void	when adding a child to a parse tree	created <td>ion node) {</td>	ion node) {
ChildProdDeclare(Produ	* node. <td>* <exception< td=""><td>ſ</td></exception<></td>	* <exception< td=""><td>ſ</td></exception<>	ſ
ction node, Node	* Houe. \/ Summary/	cref='ParseException'>	/**
child) {	* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td></param<>	if the node analysis	* <summary>Called</summary>
CHIId) (	name='node'>the parent	* discovered	when exiting a parse
node.AddChild(child);	node	errors	tree node.
}	* <param< td=""><td>*/</td><td>*</td></param<>	*/	*
,	name='child'>the child	public virtual	* <param< td=""></param<>
/**	node, or null	Node	name='node'>the node
* <summary>Called</summary>	*	ExitProdInitChoice(Pro	being exited
when entering a parse	* <exception< td=""><td>duction node) {</td><td>*</td></exception<>	duction node) {	*
tree node.	<pre>cref='ParseException'&gt;</pre>	return node;	* <returns>the</returns>
*	if the node analysis	}	node to add to the
* <param< td=""><td>* discovered</td><td></td><td>parse tree, or</td></param<>	* discovered		parse tree, or
name='node'>the node	errors	/**	* null
being entered	*/	* <summary>Called</summary>	if no parse tree
*	public virtual	when adding a child to	should be
* <exception< td=""><td>void</td><td>a parse tree</td><td>created</td></exception<>	void	a parse tree	created
cref='ParseException'>	ChildProdDeclareChoice	* node.	*
if the node analysis	(Production node, Node	*	* <exception< td=""></exception<>
* discovered	child) {	* <pre></pre>	cref='ParseException'>
errors */	node.AddChild(child);	name='node'>the parent node	if the node analysis * discovered
public virtual	node. Addeniia (chiid),	node√param/ * <param< td=""><td>errors</td></param<>	errors
void	ſ	name='child'>the child	*/
EnterProdDeclareChoice	/**	node, or null	public virtual
(Production node) {	* <summary>Called</summary>	*	Node
}	when entering a parse	* <exception< td=""><td>ExitProdAddId(Producti</td></exception<>	ExitProdAddId(Producti
,	tree node.	<pre>cref='ParseException'&gt;</pre>	on node) {
/**	*	if the node analysis	return node;
* <summary>Called</summary>	* <param< td=""><td>* discovered</td><td>}</td></param<>	* discovered	}
when exiting a parse	name='node'>the node	errors	
tree node.	being entered	*/	/**
*	*	public virtual	* <summary>Called</summary>
* <param< td=""><td>* <exception< td=""><td>void</td><td>when adding a child to</td></exception<></td></param<>	* <exception< td=""><td>void</td><td>when adding a child to</td></exception<>	void	when adding a child to
name='node'>the node	<pre>cref='ParseException'&gt;</pre>	${\tt ChildProdInitChoice}({\tt Pr}$	a parse tree
being exited	if the node analysis		* node.

*	* <exception< th=""><th>* <summary>Called</summary></th><th>* <param< th=""></param<></th></exception<>	* <summary>Called</summary>	* <param< th=""></param<>
* <param< td=""><td>cref='ParseException'&gt;</td><td>when exiting a parse</td><td>name='node'&gt;the node</td></param<>	cref='ParseException'>	when exiting a parse	name='node'>the node
name='node'>the parent	if the node analysis	tree node.	being entered
node	* discovered	*	*
* <param< td=""><td>errors</td><td>* <param< td=""><td>* <exception< td=""></exception<></td></param<></td></param<>	errors	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
name='child'>the child	*/	name='node'>the node	<pre>cref='ParseException'&gt;</pre>
node, or null	public virtual	being exited	if the node analysis
*	Node	*	* discovered
* <exception< td=""><td>ExitProdN1 (Production</td><td>* <returns>the</returns></td><td>errors</td></exception<>	ExitProdN1 (Production	* <returns>the</returns>	errors
<pre>cref='ParseException'&gt;</pre>	node) {	node to add to the	*/
if the node analysis	return node;	parse tree, or	public virtual
* discovered	}	* null	void
errors */	/starts	if no parse tree	EnterProdIndex(Product
,	/**	should be	ion node) {
public virtual	* <summary>Called</summary>	created	}
void	when adding a child to	*	/state
ChildProdAddId(Product	a parse tree	* <exception< td=""><td>/**</td></exception<>	/**
ion node, Node child)	* node.	cref='ParseException'>	* <summary>Called</summary>
1		if the node analysis	when exiting a parse
AddCh:14(-h:14).	* <param< td=""><td>* discovered</td><td>tree node.</td></param<>	* discovered	tree node.
node. AddChild(child);	name='node'>the parent	errors	*
}	node	*/	* <param name="node"/> the node
/state	* <param name='child'&gt;the child</param 	public virtual	
/**		Node ExitProdN2(Production	being exited *
* <summary>Called</summary>	node, or null	node) {	* <returns>the</returns>
when entering a parse		return node:	
tree node.	<pre>* <exception cref="ParseException"></exception></pre>	return node;	node to add to the
	if the node analysis	J	parse tree, or * null
* <param name="node"/> the node	* discovered	/**	if no parse tree
being entered (/param)	errors	* <summary>Called</summary>	should be
*	*/	when adding a child to	created
* <exception< td=""><td>public virtual</td><td>a parse tree</td><td>*</td></exception<>	public virtual	a parse tree	*
cref='ParseException'>	void	* node.	* <exception< td=""></exception<>
if the node analysis	ChildProdN1(Production	* node. \/ Summary/	cref='ParseException'>
* discovered	node, Node child) {	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
errors	node, node entra) (	name='node'>the parent	* discovered
*/	node.AddChild(child);	node	errors
public virtual	}	* <pre>* <pre>*</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>	*/
void	J	name='child'>the child	public virtual
EnterProdN1 (Production	/**	node, or null	Node
node) {	* <summary>Called</summary>	*	ExitProdIndex(Producti
}	when entering a parse	* <exception< td=""><td>on node) {</td></exception<>	on node) {
,	tree node.	cref='ParseException'>	return node;
/**	*	if the node analysis	}
* <summary>Called</summary>	* <param< td=""><td>* discovered</td><td>,</td></param<>	* discovered	,
when exiting a parse	name='node'>the node	errors	/**
tree node.	being entered	*/	* <summary>Called</summary>
*	*	public virtual	when adding a child to
* <param< td=""><td>* <exception< td=""><td>void</td><td>a parse tree</td></exception<></td></param<>	* <exception< td=""><td>void</td><td>a parse tree</td></exception<>	void	a parse tree
name='node'>the node	cref='ParseException'>	ChildProdN2(Production	* node.
being exited	if the node analysis	node, Node child) {	*
*	* discovered	nede, nede eniig, (	* <param< td=""></param<>
* <returns>the</returns>	errors	node.AddChild(child);	name='node'>the parent
node to add to the	*/	}	node
parse tree, or	public virtual	,	* <pre>* <pre>*</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>
* null	void	/**	name='child'>the child
11411		* <summary>Called</summary>	node, or null
if no parse tree	Enterproduz (Production	· Sammary/Ourrou	noso, or nurry purally
if no parse tree	EnterProdN2(Production node) {		*
should be	node) {	when entering a parse	*

* discovered		* null	public virtual
errors	/**	if no parse tree	void
*/	* <summary>Called</summary>	should be	EnterProdElemChoice(Pr
public virtual	when adding a child to	created	oduction node) {
void	a parse tree	*	}
ChildProdIndex(Product	* node.	* <exception< td=""><td></td></exception<>	
ion node, Node child)	*	<pre>cref='ParseException'&gt;</pre>	/**
{	* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td></param<>	if the node analysis	* <summary>Called</summary>
	name='node'>the parent	* discovered	when exiting a parse
node.AddChild(child);	node	errors	tree node.
}	* <param< td=""><td>*/</td><td>*</td></param<>	*/	*
,	name='child'>the child	public virtual	* <param< td=""></param<>
/**	node, or null	Node	name='node'>the node
* <summary>Called</summary>	*	ExitProdArrayAid(Produ	being exited
when entering a parse	* <exception< td=""><td>ction node) {</td><td>*</td></exception<>	ction node) {	*
tree node.	cref='ParseException'>	return node;	* <returns>the</returns>
*	if the node analysis	}	node to add to the
* <param< td=""><td>* discovered</td><td>J</td><td>parse tree, or</td></param<>	* discovered	J	parse tree, or
name='node'>the node	errors	/**	* null
	*/	,	11011
being entered	,	* <summary>Called</summary>	if no parse tree
*	public virtual	when adding a child to	should be
* <exception< td=""><td>void</td><td>a parse tree</td><td>created</td></exception<>	void	a parse tree	created
cref='ParseException'>	ChildProdSmath(Product	* node.	*
if the node analysis	ion node, Node child)	*	* <exception< td=""></exception<>
* discovered	{	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td></param<>	<pre>cref='ParseException'&gt;</pre>
errors		name='node'>the parent	if the node analysis
*/	node.AddChild(child);	node	* discovered
public virtual	}	* <param< td=""><td>errors</td></param<>	errors
void		name='child'>the child	*/
EnterProdSmath(Product	/**	node, or null	public virtual
ion node) {	* <summary>Called</summary>	*	Node
}	when entering a parse	* <exception< td=""><td>ExitProdElemChoice(Pro</td></exception<>	ExitProdElemChoice(Pro
	tree node.	<pre>cref='ParseException'&gt;</pre>	duction node) {
/**	*	if the node analysis	return node;
* <summary>Called</summary>	* <param< td=""><td>* discovered</td><td>}</td></param<>	* discovered	}
when exiting a parse	name='node'>the node	errors	
tree node.	being entered	*/	/**
*	*	public virtual	* <summary>Called</summary>
* <param< td=""><td>* <exception< td=""><td>void</td><td>when adding a child to</td></exception<></td></param<>	* <exception< td=""><td>void</td><td>when adding a child to</td></exception<>	void	when adding a child to
name='node'>the node	cref='ParseException'>	ChildProdArrayAid(Prod	a parse tree
being exited	if the node analysis	uction node, Node	* node.
*	* discovered	child) {	*
* <returns>the</returns>	errors	onii (	* <param< td=""></param<>
node to add to the	*/	node.AddChild(child);	name='node'>the parent
parse tree, or	public virtual	}	node
* null	void	J	* <param< td=""></param<>
if no parse tree	EnterProdArrayAid(Prod	/**	name='child'>the child
should be	uction node) {	* <summary>Called</summary>	node, or null
	uction node) (	· ·	
created	,	when entering a parse	*
*	/	tree node.	* <exception< td=""></exception<>
* <exception< td=""><td>/**</td><td>*</td><td><pre>cref='ParseException'&gt;</pre></td></exception<>	/**	*	<pre>cref='ParseException'&gt;</pre>
cref='ParseException'>	* <summary>Called</summary>	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
if the node analysis	when exiting a parse	name='node'>the node	* discovered
* discovered	tree node.	being entered	errors
errors	*	*	*/
*/	* <param< td=""><td>* <exception< td=""><td>public virtual</td></exception<></td></param<>	* <exception< td=""><td>public virtual</td></exception<>	public virtual
public virtual	name='node'>the node	cref='ParseException'>	void
Node	being exited	if the node analysis	ChildProdElemChoice(Pr
ExitProdSmath(Producti	*	* discovered	oduction node, Node
on node) {	* <returns>the</returns>	errors	child) {
return node;	node to add to the	*/	
}	parse tree, or		<pre>node. AddChild(child);</pre>

}	* <param< th=""><th>*/</th><th>* <param< th=""></param<></th></param<>	*/	* <param< th=""></param<>
,	name='child'>the child	public virtual	name='node'>the node
/**	node, or null	Node	being exited
* <summary>Called</summary>	*	ExitProdAddElem(Produc	*
when entering a parse	* <exception< td=""><td>tion node) {</td><td>* <returns>the</returns></td></exception<>	tion node) {	* <returns>the</returns>
tree node.	cref='ParseException'>	return node;	node to add to the
* * <param< td=""><td>if the node analysis * discovered</td><td>}</td><td>parse tree, or * null</td></param<>	if the node analysis * discovered	}	parse tree, or * null
name='node'>the node	errors	/**	if no parse tree
being entered	*/	* <summary>Called</summary>	should be
*	public virtual	when adding a child to	created
* <exception< td=""><td>void</td><td>a parse tree</td><td>*</td></exception<>	void	a parse tree	*
<pre>cref='ParseException'&gt;</pre>	ChildProdElement(Produ	* node.	* <exception< td=""></exception<>
if the node analysis	ction node, Node	*	<pre>cref='ParseException'&gt;</pre>
* discovered	child) {	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
errors		name='node'>the parent	* discovered
*/	<pre>node. AddChild(child);</pre>	node	errors
public virtual	}	* <param< td=""><td>*/</td></param<>	*/
void		name='child'>the child	public virtual
EnterProdElement(Produ	/**	node, or null	Node
ction node) {	* <summary>Called</summary>	*	ExitProdMElem(Producti
}	when entering a parse	* <exception< td=""><td>on node) {</td></exception<>	on node) {
	tree node.	cref='ParseException'>	return node;
/**	*	if the node analysis	}
* <summary>Called</summary>	* <param< td=""><td>* discovered</td><td>,</td></param<>	* discovered	,
when exiting a parse	name='node'>the node	errors	/**
tree node.	being entered	*/	* <summary>Called</summary>
*	*	public virtual	when adding a child to
* <param name='node'&gt;the node</param 	<pre>* <exception cref="ParseException"></exception></pre>	void ChildProdAddElem(Produ	a parse tree
being exited	if the node analysis	ction node, Node	* node.
*	* discovered	child) {	* <param< td=""></param<>
* <returns>the</returns>	errors	child) (	name='node'>the parent
node to add to the	*/	node.AddChild(child);	node
parse tree, or	public virtual	}	* <param< td=""></param<>
* null	void		name='child'>the child
if no parse tree	EnterProdAddElem(Produ	/**	node, or null
should be	ction node) {	* <summary>Called</summary>	*
created	}	when entering a parse	* <exception< td=""></exception<>
*		tree node.	<pre>cref='ParseException'&gt;</pre>
* <exception< td=""><td>/**</td><td>*</td><td>if the node analysis</td></exception<>	/**	*	if the node analysis
cref='ParseException'>	* <summary>Called</summary>	* <param< td=""><td>* discovered</td></param<>	* discovered
if the node analysis	when exiting a parse	name='node'>the node	errors
* discovered	tree node.	being entered	*/
errors	*	*	public virtual
*/	* <param name="node"/> the node	<pre>* <exception cref="ParseException"></exception></pre>	void ChildProdMElem(Product
public virtual		*	•
Node ExitProdElement(Produc	being exited *	if the node analysis  * discovered	ion node, Node child)
tion node) {	* <returns>the</returns>	errors	l
return node;	node to add to the	*/	node.AddChild(child);
}	parse tree, or	public virtual	loue. Addentita (chitta),
,	* null	void	,
/**	if no parse tree	EnterProdMElem(Product	/**
* <summary>Called</summary>	should be	ion node) {	* <summary>Called</summary>
when adding a child to	created	}	when entering a parse
a parse tree	*		tree node.
* node.	* <exception< td=""><td>/**</td><td>*</td></exception<>	/**	*
*	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	* <param< td=""></param<>
* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td><td>name='node'&gt;the node</td></param<>	if the node analysis	when exiting a parse	name='node'>the node
name='node'>the parent	* discovered	tree node.	being entered
node	errors	*	*

* <exception< th=""><th>public virtual</th><th>* <summary>Called</summary></th><th>should be</th></exception<>	public virtual	* <summary>Called</summary>	should be
cref='ParseException'>	void	when adding a child to	created
if the node analysis	ChildProdM2Elem(Produc	a parse tree	*
* discovered	tion node, Node child)	* node.	* <exception< td=""></exception<>
errors	{	*	<pre>cref='ParseException'&gt;</pre>
*/		* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
public virtual	<pre>node. AddChild(child);</pre>	name='node'>the parent	* discovered
void	}	node	errors
EnterProdM2E1em(Produc		* <param< td=""><td>*/</td></param<>	*/
tion node) {	/**	name='child'>the child	public virtual
}	* <summary>Called</summary>	node, or null	Node
/	when entering a parse	*	ExitProdDtypeA(Product
/**	tree node.	<pre>* <exception cref="ParseException"></exception></pre>	ion node) {
* <summary>Called when exiting a parse</summary>	* <param< td=""><td>if the node analysis</td><td>return node;</td></param<>	if the node analysis	return node;
tree node.	name='node'>the node	* discovered	ſ
*	being entered (/param)	errors	/**
* <param< td=""><td>*</td><td>*/</td><td>* <summary>Called</summary></td></param<>	*	*/	* <summary>Called</summary>
name='node'>the node	* <exception< td=""><td>public virtual</td><td>when adding a child to</td></exception<>	public virtual	when adding a child to
being exited	<pre>cref='ParseException'&gt;</pre>	void	a parse tree
*	if the node analysis	ChildProdFunctret(Prod	* node.
* <returns>the</returns>	* discovered	uction node, Node	*
node to add to the	errors	child) {	* <param< td=""></param<>
parse tree, or	*/		name='node'>the parent
* null	public virtual	<pre>node.AddChild(child);</pre>	node
if no parse tree	void	}	* <param< td=""></param<>
should be	EnterProdFunctret(Prod		name='child'>the child
created	uction node) {	/**	node, or null
*	}	* <summary>Called</summary>	*
* <exception< td=""><td></td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>		when entering a parse	* <exception< td=""></exception<>
cref='ParseException'>	/**	tree node.	<pre>cref='ParseException'&gt;</pre>
if the node analysis	* <summary>Called</summary>	*	if the node analysis
* discovered	when exiting a parse	* <pre>* <pre>* <pre>* </pre></pre></pre>	* discovered
errors	tree node.	name='node'>the node	errors
*/	·	being entered *	*/ public virtual
public virtual Node	* <param name="node"/> the node	* <exception< td=""><td>void</td></exception<>	void
ExitProdM2Elem(Product	being exited	cref='ParseException'>	ChildProdDtypeA(Produc
ion node) {	*	if the node analysis	tion node, Node child)
return node;	* <returns>the</returns>	* discovered	{
}	node to add to the	errors	
,	parse tree, or	*/	node.AddChild(child);
/**	* null	public virtual	}
* <summary>Called</summary>	if no parse tree	void	
when adding a child to	should be	EnterProdDtypeA(Produc	/**
a parse tree	created	tion node) {	* <summary>Called</summary>
* node.	*	}	when entering a parse
*	* <exception< td=""><td></td><td>tree node.</td></exception<>		tree node.
* <param< td=""><td>cref='ParseException'&gt;</td><td>/**</td><td>*</td></param<>	cref='ParseException'>	/**	*
name='node'>the parent	if the node analysis	* <summary>Called</summary>	* <param< td=""></param<>
node	* discovered	when exiting a parse	name='node'>the node
* <param< td=""><td>errors</td><td>tree node.</td><td>being entered</td></param<>	errors	tree node.	being entered
name='child'>the child	*/	*	*
node, or null	public virtual	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
*	Node	name='node'>the node	cref='ParseException'>
* <exception< td=""><td>ExitProdFunctret(Produ</td><td>being exited</td><td>if the node analysis</td></exception<>	ExitProdFunctret(Produ	being exited	if the node analysis
cref='ParseException'>	ction node) {	* (roturns)the	* discovered
if the node analysis	return node;	* <returns>the</returns>	errors
<pre>* discovered errors</pre>	ſ	node to add to the	*/ public virtual
errors */	/**	parse tree, or * null	void virtual
**/	/ TT	if no parse tree	vOIu
		ii no parse tree	

EnterProdExdtypeA(Prod	/**	* <pre>* <pre>*</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>	*/
uction node) {	* <summary>Called</summary>	name='child'>the child	public virtual
}	when entering a parse	node, or null	Node
/**	tree node.	*	ExitProdFunctvoid(Production node) {
,	* <param< td=""><td><pre>* <exception cref="ParseException"></exception></pre></td><td></td></param<>	<pre>* <exception cref="ParseException"></exception></pre>	
* <summary>Called</summary>	name='node'>the node	if the node analysis	return node;
when exiting a parse tree node.	being entered	* discovered	l
*	being entered√/param/ *	errors	/**
* <param< td=""><td>* <exception< td=""><td>*/</td><td>* <summary>Called</summary></td></exception<></td></param<>	* <exception< td=""><td>*/</td><td>* <summary>Called</summary></td></exception<>	*/	* <summary>Called</summary>
name='node'>the node	cref='ParseException'>	public virtual	when adding a child to
being exited	if the node analysis	void	a parse tree
*	* discovered	ChildProdReturn(Produc	* node.
* <returns>the</returns>	errors	tion node, Node child)	* Houe, \/ Summary/
node to add to the	*/	{	* <param< td=""></param<>
parse tree, or	public virtual	·	name='node'>the parent
* null	void	node.AddChild(child);	node
if no parse tree	EnterProdReturn(Produc	}	* <pre>* <pre>* <pre>* </pre></pre></pre>
should be	tion node) {	,	name='child'>the child
created	}	/**	node, or null
*	,	* <summary>Called</summary>	*
* <exception< td=""><td>/**</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	/**	when entering a parse	* <exception< td=""></exception<>
cref='ParseException'>	* <summary>Called</summary>	tree node.	<pre>cref='ParseException'&gt;</pre>
if the node analysis	when exiting a parse	*	if the node analysis
* discovered	tree node.	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	*	name='node'>the node	errors
*/	* <param< td=""><td>being entered</td><td>*/</td></param<>	being entered	*/
public virtual	name='node'>the node	*	public virtual
Node	being exited	* <exception< td=""><td>void</td></exception<>	void
ExitProdExdtypeA(Produ	*	cref='ParseException'>	ChildProdFunctvoid(Pro
ction node) {	* <returns>the</returns>	if the node analysis	duction node, Node
return node;	node to add to the	* discovered	child) {
}	parse tree, or	errors	
	* null	*/	node.AddChild(child);
/**	if no parse tree	public virtual	}
* <summary>Called</summary>	should be	void	
when adding a child to	created	EnterProdFunctvoid(Pro	/**
a parse tree	*	duction node) {	* <summary>Called</summary>
* node.	* <exception< td=""><td>}</td><td>when entering a parse</td></exception<>	}	when entering a parse
*	cref='ParseException'>		tree node.
* <param< td=""><td>if the node analysis</td><td>/**</td><td>*</td></param<>	if the node analysis	/**	*
name='node'>the parent	* discovered	* <summary>Called</summary>	* <param< td=""></param<>
node	errors	when exiting a parse	name='node'>the node
* <param< td=""><td>*/</td><td>tree node.</td><td>being entered</td></param<>	*/	tree node.	being entered
name='child'>the child	public virtual	*	*
node, or null	Node	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
*	ExitProdReturn(Product	name='node'>the node	cref='ParseException'>
* <exception< td=""><td>ion node) {</td><td>being exited</td><td>if the node analysis</td></exception<>	ion node) {	being exited	if the node analysis
cref='ParseException'>	return node;	*	* discovered
if the node analysis	}	* <returns>the</returns>	errors
* discovered	/state	node to add to the	*/
errors */	/**	parse tree, or * null	public virtual void
*	* <summary>Called when adding a child to</summary>		EnterProdStruct(Produc
public virtual void	=	if no parse tree should be	tion node) {
voia ChildProdExdtypeA(Prod	a parse tree * node.	created	tion node) (
uction node, Node	* Hode. \/ Summary/	*	J
child) {	* <param< td=""><td>* <exception< td=""><td>/**</td></exception<></td></param<>	* <exception< td=""><td>/**</td></exception<>	/**
oniiu/ (	name='node'>the parent	cref='ParseException'>	* <summary>Called</summary>
node.AddChild(child);	node	if the node analysis	when exiting a parse
}	node vy paramy	* discovered	tree node.
,		errors	*
		,	

* <param< th=""><th>* <exception< th=""><th>public virtual</th><th>* <summary>Called</summary></th></exception<></th></param<>	* <exception< th=""><th>public virtual</th><th>* <summary>Called</summary></th></exception<>	public virtual	* <summary>Called</summary>
name='node'>the node	<pre>cref='ParseException'&gt;</pre>	void	when adding a child to
being exited	if the node analysis	ChildProdMemDec(Produc	a parse tree
*	* discovered	tion node, Node child)	* node.
* <returns>the</returns>	errors	{	*
node to add to the	*/	(	* <param< td=""></param<>
parse tree, or	public virtual	node.AddChild(child);	name='node'>the parent
* null	void	}	node
if no parse tree	EnterProdMemDec(Produc	,	* <param< td=""></param<>
should be	tion node) {	/**	name='child'>the child
created	}	* <summary>Called</summary>	node, or null
*	,	when entering a parse	*
* <exception< td=""><td>/**</td><td>tree node. </td><td>* <exception< td=""></exception<></td></exception<>	/**	tree node.	* <exception< td=""></exception<>
<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	*	<pre>cref='ParseException'&gt;</pre>
if the node analysis	when exiting a parse	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
* discovered	tree node.	name='node'>the node	* discovered
errors	*	being entered	errors
*/	* <param< td=""><td>*</td><td>*/</td></param<>	*	*/
public virtual	name='node'>the node	* <exception< td=""><td>public virtual</td></exception<>	public virtual
Node	being exited	cref='ParseException'>	void
ExitProdStruct(Product	*	if the node analysis	ChildProdInitDec(Produ
ion node) {	* <returns>the</returns>	* discovered	ction node, Node
return node;	node to add to the	errors	child) {
}	parse tree, or	*/	, ,
	* null	public virtual	<pre>node. AddChild(child);</pre>
/**	if no parse tree	void	}
* <summary>Called</summary>	should be	EnterProdInitDec(Produ	•
when adding a child to	created	ction node) {	/**
a parse tree	*	}	* <summary>Called</summary>
* node.	* <exception< td=""><td></td><td>when entering a parse</td></exception<>		when entering a parse
*	cref='ParseException'>	/**	tree node.
* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td><td>*</td></param<>	if the node analysis	* <summary>Called</summary>	*
name='node'>the parent	* discovered	when exiting a parse	* <param< td=""></param<>
node	errors	tree node.	name='node'>the node
* <param< td=""><td>*/</td><td>*</td><td>being entered</td></param<>	*/	*	being entered
name='child'>the child	public virtual	* <param< td=""><td>*</td></param<>	*
node, or null	Node	name='node'>the node	* <exception< td=""></exception<>
*	ExitProdMemDec(Product	being exited	cref='ParseException'>
* <exception< td=""><td>ion node) {</td><td>*</td><td>if the node analysis</td></exception<>	ion node) {	*	if the node analysis
cref='ParseException'>	return node;	* <returns>the</returns>	* discovered
if the node analysis	}	node to add to the	errors
* discovered		parse tree, or	*/
errors	/**	* null	public virtual
*/	* <summary>Called</summary>	if no parse tree	void
public virtual	when adding a child to	should be	${\tt EnterProdInitDecChoice}$
void	a parse tree	created	(Production node) {
ChildProdStruct(Produc	* node.	*	}
tion node, Node child)	*	* <exception< td=""><td></td></exception<>	
{	* <param< td=""><td>cref='ParseException'&gt;</td><td>/**</td></param<>	cref='ParseException'>	/**
	name='node'>the parent	if the node analysis	* <summary>Called</summary>
<pre>node. AddChild(child);</pre>	node	* discovered	when exiting a parse
}	* <param< td=""><td>errors</td><td>tree node.</td></param<>	errors	tree node.
	name='child'>the child	*/	*
/**	node, or null	public virtual	* <param< td=""></param<>
* <summary>Called</summary>	*	Node	name='node'>the node
when entering a parse	* <exception< td=""><td>ExitProdInitDec(Produc</td><td>being exited</td></exception<>	ExitProdInitDec(Produc	being exited
tree node.	<pre>cref='ParseException'&gt;</pre>	tion node) {	*
*	if the node analysis	return node;	* <returns>the</returns>
* <param< td=""><td>* discovered</td><td>}</td><td>node to add to the</td></param<>	* discovered	}	node to add to the
name='node'>the node	errors		parse tree, or
being entered	*/	/**	* null
*			if no parse tree

1 - 111	P. 4 . D. 1C . 4 . 4 (D. 1	/stude	ab. Z
should be created	EnterProdConstant(Production node) {	/** * <summary>Called</summary>	* <param name='child'&gt;the child</param 
*	uction node) {	when entering a parse	node, or null
* <exception< td=""><td>J</td><td>tree node. </td><td>*</td></exception<>	J	tree node.	*
<pre>cref='ParseException'&gt;</pre>	/**	*	* <exception< td=""></exception<>
if the node analysis	* <summary>Called</summary>	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td></param<>	<pre>cref='ParseException'&gt;</pre>
* discovered	when exiting a parse	name='node'>the node	if the node analysis
errors	tree node.	being entered	* discovered
*/	*	*	errors
public virtual	* <param< td=""><td>* <exception< td=""><td>*/</td></exception<></td></param<>	* <exception< td=""><td>*/</td></exception<>	*/
Node	name='node'>the node	<pre>cref='ParseException'&gt;</pre>	public virtual
ExitProdInitDecChoice(	being exited	if the node analysis	void
Production node) {	*	* discovered	ChildProdLocalChoice(P
return node;	* <returns>the</returns>	errors	roduction node, Node
}	node to add to the	*/	child) {
/stote	parse tree, or * null	public virtual	1- AlaCh:11(-h:11).
/** * <summary>Called</summary>	11011	void EnterProdLocalChoice(P	node.AddChild(child);
when adding a child to	if no parse tree should be	roduction node) {	J
a parse tree	created	}	/**
* node.	*	,	* <summary>Called</summary>
*	* <exception< td=""><td>/**</td><td>when entering a parse</td></exception<>	/**	when entering a parse
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>* <summary>Called</summary></td><td>tree node. </td></param<>	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	tree node.
name='node'>the parent	if the node analysis	when exiting a parse	*
node	* discovered	tree node.	* <param< td=""></param<>
* <param< td=""><td>errors</td><td>*</td><td>name='node'&gt;the node</td></param<>	errors	*	name='node'>the node
name='child'>the child	*/	* <param< td=""><td>being entered</td></param<>	being entered
node, or null	public virtual	name='node'>the node	*
*	Node	being exited	* <exception< td=""></exception<>
* <exception< td=""><td>ExitProdConstant (Produ</td><td>*</td><td><pre>cref='ParseException'&gt;</pre></td></exception<>	ExitProdConstant (Produ	*	<pre>cref='ParseException'&gt;</pre>
cref='ParseException'>	ction node) {	* <returns>the</returns>	if the node analysis
if the node analysis	return node;	node to add to the	* discovered
<pre>* discovered errors</pre>	}	parse tree, or * null	errors */
*/	/**	if no parse tree	public virtual
public virtual	* <summary>Called</summary>	should be	void
void	when adding a child to	created	EnterProdDeclare1(Prod
ChildProdInitDecChoice	a parse tree	*	uction node) {
(Production node, Node	* node.	* <exception< td=""><td>}</td></exception<>	}
child) {	*	cref='ParseException'>	
	* <param< td=""><td>if the node analysis</td><td>/**</td></param<>	if the node analysis	/**
<pre>node. AddChild(child);</pre>	name='node'>the parent	* discovered	* <summary>Called</summary>
}	node	errors	when exiting a parse
	* <param< td=""><td>*/</td><td>tree node.</td></param<>	*/	tree node.
/**	name='child'>the child	public virtual	*
* <summary>Called</summary>	node, or null	Node	* <param< td=""></param<>
when entering a parse	* * <exception< td=""><td>ExitProdLocalChoice(Production node) {</td><td>name='node'&gt;the node</td></exception<>	ExitProdLocalChoice(Production node) {	name='node'>the node
tree node.	cref='ParseException'>	return node;	being exited *
* <param< td=""><td>if the node analysis</td><td>return node,</td><td>* <returns>the</returns></td></param<>	if the node analysis	return node,	* <returns>the</returns>
name='node'>the node	* discovered	,	node to add to the
being entered	errors	/**	parse tree, or
*	*/	* <summary>Called</summary>	* null
* <exception< td=""><td>public virtual</td><td>when adding a child to</td><td>if no parse tree</td></exception<>	public virtual	when adding a child to	if no parse tree
<pre>cref='ParseException'&gt;</pre>	void	a parse tree	should be
if the node analysis	ChildProdConstant(Prod	* node.	created
* discovered	uction node, Node	*	*
errors	child) {	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
*/		name='node'>the parent	<pre>cref='ParseException'&gt;</pre>
public virtual	node.AddChild(child);	node	if the node analysis
void	}		* discovered
			errors

*/	* <param< th=""><th>* <exception< th=""><th>public virtual</th></exception<></th></param<>	* <exception< th=""><th>public virtual</th></exception<>	public virtual
public virtual	name='node'>the node	cref='ParseException'>	void
Node	being exited	if the node analysis	ChildProdFunctvoid1(Pr
ExitProdDeclare1(Produ	*	* discovered	oduction node, Node
ction node) {	* <returns>the</returns>	errors	child) {
return node;	node to add to the	*/	
}	parse tree, or	public virtual	<pre>node. AddChild(child);</pre>
	* null	void	}
/**	if no parse tree	EnterProdFunctvoid1(Pr	
* <summary>Called</summary>	should be	oduction node) {	/**
when adding a child to	created	}	* <summary>Called</summary>
a parse tree	*		when entering a parse
* node.	* <exception< td=""><td>/**</td><td>tree node.</td></exception<>	/**	tree node.
*	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	*
* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td><td>* <param< td=""></param<></td></param<>	if the node analysis	when exiting a parse	* <param< td=""></param<>
name='node'>the parent	* discovered	tree node.	name='node'>the node
node	errors */	*	being entered
* <param name='child'&gt;the child</param 	*/ public virtual	* <param name='node'&gt;the node</param 	* * <exception< td=""></exception<>
node, or null	Node	being exited	cref='ParseException'>
*	ExitProdFunctret1(Prod	*	if the node analysis
* <exception< td=""><td>uction node) {</td><td>* <returns>the</returns></td><td>* discovered</td></exception<>	uction node) {	* <returns>the</returns>	* discovered
cref='ParseException'>	return node;	node to add to the	errors
if the node analysis	}	parse tree, or	*/
* discovered	•	* null	public virtual
errors	/**	if no parse tree	void
*/	* <summary>Called</summary>	should be	EnterProdStruct1(Produ
public virtual	when adding a child to	created	ction node) {
void	a parse tree	*	}
ChildProdDeclare1(Prod	* node.	* <exception< td=""><td></td></exception<>	
uction node, Node	*	cref='ParseException'>	/**
child) {	* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td></param<>	if the node analysis	* <summary>Called</summary>
1 41101:11/1:11	name='node'>the parent	* discovered	when exiting a parse
node. AddChild(child);	node	errors	tree node.
}	* <param name="child"/> the child	*/	*
/**	name- child the child node, or null	public virtual Node	* <param name='node'&gt;the node</param 
* <summary>Called</summary>	*	ExitProdFunctvoid1(Pro	being exited
when entering a parse	* <exception< td=""><td>duction node) {</td><td>*</td></exception<>	duction node) {	*
tree node.	<pre>cref='ParseException'&gt;</pre>	return node:	* <returns>the</returns>
*	if the node analysis	}	node to add to the
* <param< td=""><td>* discovered</td><td></td><td>parse tree, or</td></param<>	* discovered		parse tree, or
name='node'>the node	errors	/**	* null
being entered	*/	* <summary>Called</summary>	if no parse tree
*	public virtual	when adding a child to	should be
* <exception< td=""><td>void</td><td>a parse tree</td><td>created</td></exception<>	void	a parse tree	created
cref='ParseException'>	ChildProdFunctret1(Pro	* node.	*
if the node analysis	duction node, Node	*	* <exception< td=""></exception<>
* discovered	child) {	* <pre>* <pre>* <pre></pre></pre></pre>	<pre>cref='ParseException'&gt;</pre>
errors	1 41101:11(1:11)	name='node'>the parent	if the node analysis
*/	node.AddChild(child);	node	* discovered
public virtual void	J	* <param name='child'&gt;the child</param 	errors */
EnterProdFunctret1(Pro	/**	node, or null	public virtual
duction node) {	* <summary>Called</summary>	node, or nurry param/	Node
}	when entering a parse	* <exception< td=""><td>ExitProdStruct1(Produc</td></exception<>	ExitProdStruct1(Produc
,	tree node.	cref='ParseException'>	tion node) {
/**	*	if the node analysis	return node;
* <summary>Called</summary>	* <param< td=""><td>* discovered</td><td>}</td></param<>	* discovered	}
when exiting a parse	name='node'>the node	errors	•
tree node.	being entered	*/	/**
*	*		

* <summary>Called</summary>	should be	EnterProdMain(Producti	* <summary>Called</summary>
when adding a child to	created	on node) {	when entering a parse
a parse tree	*	}	tree node.
* node.	<pre>* <exception cref="ParseException"></exception></pre>	/**	•
* (nonem	if the node analysis	* <summary>Called</summary>	* <param name="node"/> the node
* <param name='node'&gt;the parent</param 	* discovered	when exiting a parse	being entered
node	errors	tree node.	being entered/param/
* <param< td=""><td>*/</td><td>*</td><td>* <exception< td=""></exception<></td></param<>	*/	*	* <exception< td=""></exception<>
name='child'>the child	public virtual	* <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
node, or null	Node	name='node'>the node	if the node analysis
*	ExitProdConstant1(Prod	being exited	* discovered
* <exception< td=""><td>uction node) {</td><td>*</td><td>errors</td></exception<>	uction node) {	*	errors
cref='ParseException'>	return node;	* <returns>the</returns>	*/
if the node analysis	}	node to add to the	public virtual
* discovered	,	parse tree, or	void
errors	/**	* null	EnterProdAssignChoice(
*/	* <summary>Called</summary>	if no parse tree	Production node) {
public virtual	when adding a child to	should be	}
void	a parse tree	created	,
ChildProdStruct1(Produ	* node.	*	/**
ction node, Node	*	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
child) {	* <param< td=""><td>cref='ParseException'&gt;</td><td>when exiting a parse</td></param<>	cref='ParseException'>	when exiting a parse
	name='node'>the parent	if the node analysis	tree node.
node.AddChild(child);	node	* discovered	*
}	* <param< td=""><td>errors</td><td>* <param< td=""></param<></td></param<>	errors	* <param< td=""></param<>
	name='child'>the child	*/	name='node'>the node
/**	node, or null	public virtual	being exited
* <summary>Called</summary>	*	Node	*
when entering a parse	* <exception< td=""><td>ExitProdMain(Productio</td><td>* <returns>the</returns></td></exception<>	ExitProdMain(Productio	* <returns>the</returns>
tree node.	<pre>cref='ParseException'&gt;</pre>	n node) {	node to add to the
*	if the node analysis	return node;	parse tree, or
* <param< td=""><td>* discovered</td><td>}</td><td>* null</td></param<>	* discovered	}	* null
name='node'>the node	errors		if no parse tree
being entered	*/	/**	should be
*	public virtual	* <summary>Called</summary>	created
* <exception< td=""><td>void</td><td>when adding a child to</td><td>*</td></exception<>	void	when adding a child to	*
<pre>cref='ParseException'&gt;</pre>	ChildProdConstant1(Pro	a parse tree	* <exception< td=""></exception<>
if the node analysis	duction node, Node	* node.	cref='ParseException'>
* discovered	child) {	*	if the node analysis
errors		* <param< td=""><td>* discovered</td></param<>	* discovered
*/	<pre>node. AddChild(child);</pre>	name='node'>the parent	errors
public virtual	}	node	*/
void		* <param< td=""><td>public virtual</td></param<>	public virtual
EnterProdConstant1(Pro	/**	name='child'>the child	Node
duction node) {	* <summary>Called</summary>	node, or null	ExitProdAssignChoice(P
}	when entering a parse	*	roduction node) {
	tree node.	* <exception< td=""><td>return node;</td></exception<>	return node;
/**	*	cref='ParseException'>	}
* <summary>Called</summary>	* <param< td=""><td>if the node analysis</td><td></td></param<>	if the node analysis	
when exiting a parse	name='node'>the node	* discovered	/**
tree node.	being entered	errors	* <summary>Called</summary>
*	*	*/	when adding a child to
* <param< td=""><td>* <exception< td=""><td>public virtual</td><td>a parse tree</td></exception<></td></param<>	* <exception< td=""><td>public virtual</td><td>a parse tree</td></exception<>	public virtual	a parse tree
name='node'>the node	cref='ParseException'>	void	* node.
being exited	if the node analysis	ChildProdMain(Producti	*
*	* discovered	on node, Node child) {	* <param< td=""></param<>
* <returns>the</returns>	errors	1 41101 111/ 1 11	name='node'>the parent
node to add to the	*/	node.AddChild(child);	node
parse tree, or	public virtual	}	* <param -'="" 1:11'="" \<="" td=""/>
* null	void	/start-	name='child'>the child
if no parse tree		/**	node, or null

*	public virtual	* <param< th=""><th>* <exception< th=""></exception<></th></param<>	* <exception< th=""></exception<>
* <exception< td=""><td>Node</td><td>name='node'&gt;the node</td><td>cref='ParseException'&gt;</td></exception<>	Node	name='node'>the node	cref='ParseException'>
cref='ParseException'>	ExitProdAccessAssignDt	being exited	if the node analysis
if the node analysis	ype(Production node) {	*	* discovered
* discovered	return node;	* <returns>the</returns>	errors
errors	}	node to add to the	*/
*/	/state	parse tree, or	public virtual
public virtual	/**	* null	void
void	* <summary>Called</summary>	if no parse tree	EnterProdAssigning (Pro
ChildProdAssignChoice(	when adding a child to	should be	duction node) {
Production node, Node	a parse tree	created	}
child) {	* node.	*	/state
AddCh:1d(-h:1d).	*	* <exception< td=""><td>/**</td></exception<>	/**
node. AddChild(child);	* <pre></pre>	cref='ParseException'>	* <summary>Called</summary>
ĵ	name='node'>the parent	if the node analysis	when exiting a parse
/**	node * <param< td=""><td><pre>* discovered errors</pre></td><td>tree node.</td></param<>	<pre>* discovered errors</pre>	tree node.
* <summary>Called</summary>	name='child'>the child	*/	
when entering a parse	name- child /the child node, or null	public virtual	* <param name="node"/> the node
tree node.	*	Node	being exited
*	* <exception< td=""><td>ExitProdAssignValueCho</td><td>*</td></exception<>	ExitProdAssignValueCho	*
* <param< td=""><td>cref='ParseException'&gt;</td><td>ice(Production node) {</td><td>* <returns>the</returns></td></param<>	cref='ParseException'>	ice(Production node) {	* <returns>the</returns>
name='node'>the node	if the node analysis	return node;	node to add to the
being entered (/param)	* discovered	leturn node,	parse tree, or
*	errors	J	* null
* <exception< td=""><td>*/</td><td>/**</td><td>if no parse tree</td></exception<>	*/	/**	if no parse tree
cref='ParseException'>	public virtual	* <summary>Called</summary>	should be
if the node analysis	void	when adding a child to	created
* discovered	ChildProdAccessAssignD	a parse tree	*
errors	type (Production node,	* node.	* <exception< td=""></exception<>
*/	Node child) {	*	cref='ParseException'>
public virtual	node chila) (	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
void	<pre>node. AddChild(child);</pre>	name='node'>the parent	* discovered
EnterProdAccessAssignD	}	node	errors
type (Production node)	,	* <param< td=""><td>*/</td></param<>	*/
{	/**	name='child'>the child	public virtual
}	* <summary>Called</summary>	node, or null	Node
	when entering a parse	*	ExitProdAssigning(Prod
/**	tree node.	* <exception< td=""><td>uction node) {</td></exception<>	uction node) {
* <summary>Called</summary>	*	cref='ParseException'>	return node;
when exiting a parse	* <param< td=""><td>if the node analysis</td><td>}</td></param<>	if the node analysis	}
tree node.	name='node'>the node	* discovered	
*	being entered	errors	/**
* <param< td=""><td>*</td><td>*/</td><td>* <summary>Called</summary></td></param<>	*	*/	* <summary>Called</summary>
name='node'>the node	* <exception< td=""><td>public virtual</td><td>when adding a child to</td></exception<>	public virtual	when adding a child to
being exited	<pre>cref='ParseException'&gt;</pre>	void	a parse tree
*	if the node analysis	ChildProdAssignValueCh	* node.
* <returns>the</returns>	* discovered	oice(Production node,	*
node to add to the	errors	Node child) {	* <param< td=""></param<>
parse tree, or	*/		name='node'>the parent
* null	public virtual	<pre>node.AddChild(child);</pre>	node
if no parse tree	void	}	* <param< td=""></param<>
should be	EnterProdAssignValueCh		name='child'>the child
created	oice(Production node)	/**	node, or null
*	{	* <summary>Called</summary>	*
* <exception< td=""><td>}</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	}	when entering a parse	* <exception< td=""></exception<>
<pre>cref='ParseException'&gt;</pre>		tree node.	<pre>cref='ParseException'&gt;</pre>
if the node analysis	/**	*	if the node analysis
* discovered	* <summary>Called</summary>	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	when exiting a parse	name='node'>the node	errors
*/	tree node.	being entered	*/
	*	*	

public virtual	* <summary>Called</summary>	should be	EnterProdAssignSym(Pro
void	when adding a child to	created	duction node) {
ChildProdAssigning(Pro	a parse tree	*	}
duction node, Node	* node.	* <exception< td=""><td></td></exception<>	
child) {	*	<pre>cref='ParseException'&gt;</pre>	/**
	* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td></param<>	if the node analysis	* <summary>Called</summary>
node.AddChild(child);	name='node'>the parent	* discovered	when exiting a parse
}	node	errors	tree node.
	* <param< td=""><td>*/</td><td>*</td></param<>	*/	*
/**	name='child'>the child	public virtual	* <param< td=""></param<>
* <summary>Called</summary>	node, or null	Node	name='node'>the node
when entering a parse	*	ExitProdArrayIdtail(Pr	being exited
tree node.	* <exception< td=""><td>oduction node) {</td><td>*</td></exception<>	oduction node) {	*
*	<pre>cref='ParseException'&gt;</pre>	return node;	* <returns>the</returns>
* <param< td=""><td>if the node analysis</td><td>}</td><td>node to add to the</td></param<>	if the node analysis	}	node to add to the
name='node'>the node	* discovered		parse tree, or
being entered	errors	/**	* null
*	*/	* <summary>Called</summary>	if no parse tree
* <exception< td=""><td>public virtual</td><td>when adding a child to</td><td>should be</td></exception<>	public virtual	when adding a child to	should be
<pre>cref='ParseException'&gt;</pre>	void	a parse tree	created
if the node analysis	ChildProdArrayId(Produ	* node.	*
* discovered	ction node, Node	*	* <exception< td=""></exception<>
errors	child) {	* <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
*/		name='node'>the parent	if the node analysis
public virtual	node.AddChild(child);	node	* discovered
void	}	* <param< td=""><td>errors</td></param<>	errors
EnterProdArrayId(Produ		name='child'>the child	*/
ction node) {	/**	node, or null	public virtual
}	* <summary>Called</summary>	*	Node
	when entering a parse	* <exception< td=""><td>ExitProdAssignSym(Prod</td></exception<>	ExitProdAssignSym(Prod
/**	tree node.	<pre>cref='ParseException'&gt;</pre>	uction node) {
* <summary>Called</summary>	*	if the node analysis	return node;
when exiting a parse	* <param< td=""><td>* discovered</td><td>}</td></param<>	* discovered	}
tree node.	name='node'>the node	errors	,
*	being entered	*/	/**
* <param< td=""><td>*</td><td>public virtual</td><td>* <summary>Called</summary></td></param<>	*	public virtual	* <summary>Called</summary>
name='node'>the node	* <exception< td=""><td>void</td><td>when adding a child to</td></exception<>	void	when adding a child to
being exited	cref='ParseException'>	ChildProdArrayIdtail(P	a parse tree
*	if the node analysis	roduction node, Node	* node.
* <returns>the</returns>	* discovered	child) {	*
node to add to the	errors	. 1 411(1:11/1:11)	* <param< td=""></param<>
parse tree, or	*/	node.AddChild(child);	name='node'>the parent
* null	public virtual	}	node
if no parse tree should be	void EnterProdArrayIdtail(P	/**	* <param name="child"/> the child
created	roduction node) {	* <summary>Called</summary>	node, or null
*	roduction mode) (	when entering a parse	noue, or nurr√param/ *
* <exception< td=""><td>J</td><td>tree node. </td><td>* <exception< td=""></exception<></td></exception<>	J	tree node.	* <exception< td=""></exception<>
cref='ParseException'>	/**	*	cref='ParseException'>
if the node analysis	* <summary>Called</summary>	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
* discovered	when exiting a parse	name='node'>the node	* discovered
errors	tree node.	being entered <pre>/param&gt;</pre>	errors
*/	*	*	*/
public virtual	* <param< td=""><td>* <exception< td=""><td>public virtual</td></exception<></td></param<>	* <exception< td=""><td>public virtual</td></exception<>	public virtual
Node	name='node'>the node	cref='ParseException'>	void
ExitProdArrayId(Produc	being exited	if the node analysis	ChildProdAssignSym(Pro
tion node) {	*	* discovered	duction node, Node
return node;	* <returns>the</returns>	errors	child) {
}	node to add to the	*/	311147
,	parse tree, or	public virtual	node.AddChild(child);
/**	* null	void	}
•	if no parse tree		•
	=		

/**	* <param< th=""><th>*/</th><th>* <param< th=""></param<></th></param<>	*/	* <param< th=""></param<>
* <summary>Called</summary>	name='child'>the child	public virtual	name='node'>the node
when entering a parse	node, or null	Node	being exited
tree node.	*	ExitProdConvert(Produc	*
*	* <exception< td=""><td>tion node) {</td><td>* <returns>the</returns></td></exception<>	tion node) {	* <returns>the</returns>
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>return node;</td><td>node to add to the</td></param<>	<pre>cref='ParseException'&gt;</pre>	return node;	node to add to the
name='node'>the node	if the node analysis	}	parse tree, or
being entered	* discovered	/	* null
* * <exception< td=""><td>errors */</td><td>/** * <summary>Called</summary></td><td>if no parse tree should be</td></exception<>	errors */	/** * <summary>Called</summary>	if no parse tree should be
cref='ParseException'>	≁/ public virtual	when adding a child to	created
if the node analysis	void	a parse tree	*
* discovered	ChildProdAssignValue(P	* node.	* <exception< td=""></exception<>
errors	roduction node, Node	*	<pre>cref='ParseException'&gt;</pre>
*/	child) {	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
public virtual		name='node'>the parent	* discovered
void	<pre>node. AddChild(child);</pre>	node	errors
EnterProdAssignValue(P	}	* <param< td=""><td>*/</td></param<>	*/
roduction node) {		name='child'>the child	public virtual
}	/**	node, or null	Node
	* <summary>Called</summary>	*	ExitProdFunctParam(Pro
/**	when entering a parse	* <exception< td=""><td>duction node) {</td></exception<>	duction node) {
* <summary>Called</summary>	tree node.	cref='ParseException'>	return node;
when exiting a parse	*	if the node analysis	}
tree node.	* <param name="node"/> the node	<pre>* discovered errors</pre>	/**
* <param< td=""><td>being entered</td><td>*/</td><td>* <summary>Called</summary></td></param<>	being entered	*/	* <summary>Called</summary>
name='node'>the node	*	public virtual	when adding a child to
being exited	* <exception< td=""><td>void</td><td>a parse tree</td></exception<>	void	a parse tree
*	<pre>cref='ParseException'&gt;</pre>	ChildProdConvert(Produ	* node.
* <returns>the</returns>	if the node analysis	ction node, Node	*
node to add to the	* discovered	child) {	* <param< td=""></param<>
parse tree, or	errors		name='node'>the parent
* null	*/	<pre>node. AddChild(child);</pre>	node
if no parse tree	public virtual	}	* <param< td=""></param<>
should be	void		name='child'>the child
created	EnterProdConvert (Produ	/**	node, or null
*	ction node) {	* <summary>Called</summary>	*
* <exception< td=""><td>}</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	}	when entering a parse	* <exception< td=""></exception<>
<pre>cref='ParseException'&gt; if the node analysis</pre>	/**	tree node.	<pre>cref='ParseException'&gt; if the node analysis</pre>
* discovered	* <summary>Called</summary>	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	when exiting a parse	name='node'>the node	errors
*/	tree node.	being entered	*/
public virtual	*	*	public virtual
Node	* <param< td=""><td>* <exception< td=""><td>void</td></exception<></td></param<>	* <exception< td=""><td>void</td></exception<>	void
ExitProdAssignValue(Pr	name='node'>the node	<pre>cref='ParseException'&gt;</pre>	ChildProdFunctParam(Pr
oduction node) {	being exited	if the node analysis	oduction node, Node
return node;	*	* discovered	child) {
}	* <returns>the</returns>	errors	
	node to add to the	*/	node. AddChild(child);
/**	parse tree, or	public virtual	}
* <summary>Called when adding a child to</summary>	* null	void	/state
ŭ	if no parse tree	EnterProdFunctParam(Pr	/**
a parse tree * node.	should be created	oduction node) { }	* <summary>Called when entering a parse</summary>
* Hode. \/ Summary/	*	J	tree node.
* <param< td=""><td>* <exception< td=""><td>/**</td><td>*</td></exception<></td></param<>	* <exception< td=""><td>/**</td><td>*</td></exception<>	/**	*
name='node'>the parent	cref='ParseException'>	* <summary>Called</summary>	* <param< td=""></param<>
node	if the node analysis	when exiting a parse	name='node'>the node
	* discovered	tree node.	being entered
	errors	*	*

* <exception< td=""><td>public virtual</td><td>* <summary>Called</summary></td><td>should be</td></exception<>	public virtual	* <summary>Called</summary>	should be
<pre>cref='ParseException'&gt; if the node analysis</pre>	void ChildProdFunctIdparam(	when adding a child to	created
* discovered	Production node, Node	a parse tree * node.	* <exception< td=""></exception<>
errors	child) {	* Hode. (/ Summary/	cref='ParseException'>
*/	Chilid) (	∗ <param< td=""><td>if the node analysis</td></param<>	if the node analysis
public virtual	node.AddChild(child);	name='node'>the parent	* discovered
void	}	node	errors
EnterProdFunctIdparam(	•	* <param< td=""><td>*/</td></param<>	*/
Production node) {	/**	name='child'>the child	public virtual
}	* <summary>Called</summary>	node, or null	Node
	when entering a parse	*	ExitProdBody(Productio
/**	tree node.	* <exception< td=""><td>n node) {</td></exception<>	n node) {
* <summary>Called</summary>	*	<pre>cref='ParseException'&gt;</pre>	return node;
when exiting a parse	* <param< td=""><td>if the node analysis</td><td>}</td></param<>	if the node analysis	}
tree node.	name='node'>the node	* discovered	
*	being entered	errors	/**
* <param< td=""><td>*</td><td>*/</td><td>* <summary>Called</summary></td></param<>	*	*/	* <summary>Called</summary>
name='node'>the node	* <exception< td=""><td>public virtual</td><td>when adding a child to</td></exception<>	public virtual	when adding a child to
being exited	<pre>cref='ParseException'&gt;</pre>	void	a parse tree
*	if the node analysis	ChildProdAddfunctIdpar	* node.
* <returns>the</returns>	* discovered	am(Production node,	*
node to add to the	errors	Node child) {	* <param< td=""></param<>
parse tree, or  * null	*/		name='node'>the parent
if no parse tree	public virtual void	node.AddChild(child);	node * <param< td=""></param<>
should be	Volu EnterProdAddfunctIdpar	}	name='child'>the child
created	am(Production node) {	/**	node, or null
*	}	* <summary>Called</summary>	*
* <exception< td=""><td>,</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	,	when entering a parse	* <exception< td=""></exception<>
<pre>cref='ParseException'&gt;</pre>	/**	tree node.	<pre>cref='ParseException'&gt;</pre>
if the node analysis	* <summary>Called</summary>	*	if the node analysis
* discovered	when exiting a parse	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	tree node.	name='node'>the node	errors
*/	*	being entered	*/
public virtual	* <param< td=""><td>*</td><td>public virtual</td></param<>	*	public virtual
Node	name='node'>the node	* <exception< td=""><td>void</td></exception<>	void
ExitProdFunctIdparam(P	being exited	<pre>cref='ParseException'&gt;</pre>	ChildProdBody(Producti
roduction node) {	*	if the node analysis	on node, Node child) {
return node;	* <returns>the</returns>	* discovered	
}	node to add to the	errors	node. AddChild(child);
/	parse tree, or	*/	}
/**	* null	public virtual	/stell
* <summary>Called</summary>	if no parse tree should be	void	/**
when adding a child to	created	EnterProdBody(Producti on node) {	* <summary>Called</summary>
a parse tree * node.	*	on node) (	when entering a parse tree node.
* Hode. \/ Summary/	* <exception< td=""><td>J</td><td>*</td></exception<>	J	*
* <param< td=""><td>cref='ParseException'&gt;</td><td>/**</td><td>* <param< td=""></param<></td></param<>	cref='ParseException'>	/**	* <param< td=""></param<>
name='node'>the parent	if the node analysis	* <summary>Called</summary>	name='node'>the node
node	* discovered	when exiting a parse	being entered
* <param< td=""><td>errors</td><td>tree node. </td><td>*</td></param<>	errors	tree node.	*
name='child'>the child	*/	*	* <exception< td=""></exception<>
node, or null	public virtual	* <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
*	Node	name='node'>the node	if the node analysis
* <exception< td=""><td>ExitProdAddfunctIdpara</td><td>being exited</td><td>* discovered</td></exception<>	ExitProdAddfunctIdpara	being exited	* discovered
<pre>cref='ParseException'&gt;</pre>	m(Production node) {	*	errors
if the node analysis	return node;	* <returns>the</returns>	*/
* discovered	}	node to add to the	public virtual
errors		parse tree, or	void
*/	/**	* null	EnterProdPrint(Product
		if no parse tree	ion node) {

}	* <summary>Called</summary>	*	ExitProdOut(Production
	when entering a parse	* <exception< td=""><td>node) {</td></exception<>	node) {
/**	tree node.	cref='ParseException'>	return node;
* <summary>Called</summary>	*	if the node analysis	}
when exiting a parse	* <param< td=""><td>* discovered</td><td>,</td></param<>	* discovered	,
tree node.	name='node'>the node	errors	/**
*	being entered	*/	* <summary>Called</summary>
* <param< td=""><td>*</td><td>public virtual</td><td>when adding a child to</td></param<>	*	public virtual	when adding a child to
name='node'>the node	<pre>* <exception cref="ParseException"></exception></pre>	void	a parse tree
being exited	if the node analysis	ChildProdPostval(Produ ction node, Node	* node.
* <returns>the</returns>	* discovered	ction node, Node child) {	* /namam
node to add to the	errors	CHIId) (	* <param name="node"/> the parent
parse tree, or	*/	node.AddChild(child);	node
* null	public virtual	}	* <pre>* <pre>*</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>
if no parse tree	void	,	name='child'>the child
should be	EnterProdPostval(Produ	/**	node, or null
created	ction node) {	* <summary>Called</summary>	*
*	}	when entering a parse	* <exception< td=""></exception<>
* <exception< td=""><td></td><td>tree node. </td><td><pre>cref='ParseException'&gt;</pre></td></exception<>		tree node.	<pre>cref='ParseException'&gt;</pre>
<pre>cref='ParseException'&gt;</pre>	/**	*	if the node analysis
if the node analysis	* <summary>Called</summary>	* <param< td=""><td>* discovered</td></param<>	* discovered
* discovered	when exiting a parse	name='node'>the node	errors
errors	tree node.	being entered	*/
*/	*	*	public virtual
public virtual	* <param< td=""><td>* <exception< td=""><td>void</td></exception<></td></param<>	* <exception< td=""><td>void</td></exception<>	void
Node	name='node'>the node	cref='ParseException'>	ChildProdOut (Productio
ExitProdPrint(Producti	being exited	if the node analysis	n node, Node child) {
on node) {	*	* discovered	1 41101:11/1:11
return node;	* <returns>the</returns>	errors	node.AddChild(child);
}	node to add to the	*/	}
/**	parse tree, or * null	public virtual void	/**
* <summary>Called</summary>	if no parse tree	EnterProdOut(Productio	* <summary>Called</summary>
when adding a child to	should be	n node) {	when entering a parse
a parse tree	created	}	tree node.
* node.	*	,	*
*	* <exception< td=""><td>/**</td><td>* <param< td=""></param<></td></exception<>	/**	* <param< td=""></param<>
* * <param< td=""><td><pre>* <exception cref="ParseException"></exception></pre></td><td>/** * <summary>Called</summary></td><td>* <param name="node"/>the node</td></param<>	<pre>* <exception cref="ParseException"></exception></pre>	/** * <summary>Called</summary>	* <param name="node"/> the node
*		,	
* * <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>* <summary>Called</summary></td><td>name='node'&gt;the node</td></param<>	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	name='node'>the node
*     * <param name="node"/> the parent	<pre>cref='ParseException'&gt; if the node analysis</pre>	* <summary>Called when exiting a parse</summary>	name='node'>the node being entered  *  *exception
*     * <param name="node"/> the parent node	<pre>cref='ParseException'&gt; if the node analysis   * discovered</pre>	* <summary>Called when exiting a parse tree node. </summary> * * <pre> * </pre>	name='node'>the node being entered *
*     * <param name="node"/> the parent node     * <param< td=""><td><pre>cref='ParseException'&gt; if the node analysis   * discovered errors   */   public virtual</pre></td><td>* <summary>Called when exiting a parse tree node. </summary> *</td><td>name='node'&gt;the node being entered  *  * <exception cref="ParseException"> if the node analysis</exception></td></param<>	<pre>cref='ParseException'&gt; if the node analysis   * discovered errors   */   public virtual</pre>	* <summary>Called when exiting a parse tree node. </summary> *	name='node'>the node being entered  *  * <exception cref="ParseException"> if the node analysis</exception>
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     *	<pre>cref='ParseException'&gt; if the node analysis   * discovered errors   */   public virtual Node</pre>	* <summary>Called when exiting a parse tree node. </summary> * * <pre> * </pre>	name='node'>the node being entered  *  * <a href="mailto:kexception"> cref='ParseException"&gt; if the node analysis  * discovered</a>
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     *     * <exception< td=""><td><pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Produc</pre></td><td>* <summary>Called when exiting a parse tree node.</summary>  *  * <param name="node"/>the node being exited *</td><td><pre>name='node'&gt;the node being entered           *           *<exception cref="ParseException"> if the node analysis           * discovered errors</exception></pre></td></exception<>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Produc</pre>	* <summary>Called when exiting a parse tree node.</summary> *  * <param name="node"/> the node being exited *	<pre>name='node'&gt;the node being entered           *           *<exception cref="ParseException"> if the node analysis           * discovered errors</exception></pre>
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     *     * <exception cref="ParseException"></exception>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {</pre>	* <summary>Called when exiting a parse tree node.</summary>	<pre>name='node'&gt;the node being entered           *           *<exception cref="ParseException"> if the node analysis           * discovered errors</exception>           */</pre>
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     *     * <exception cref="ParseException"> if the node analysis</exception>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Produc</pre>	* <summary>Called when exiting a parse tree node.</summary>	name='node'>the node being entered  *  * <a href="mailto:*exception">*exception</a> cref='ParseException'> if the node analysis  * discovered errors errors public virtual
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     *     * <exception cref="ParseException"> if the node analysis     * discovered</exception>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {</pre>	* <summary>Called when exiting a parse tree node.</summary>	name='node'>the node being entered  *  * * * *cref='ParseException'> if the node analysis  * discovered errors  */  public virtual void
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }</pre>	* <summary>Called when exiting a parse tree node. </summary> *     * <param name="node"/> the node being exited     *     * <returns>the node to add to the parse tree, or     * null</returns>	name='node'>the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */ public virtual void EnterProdOutC(Producti
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }     /**</pre>	* <summary>Called when exiting a parse tree node. </summary>	name='node'>the node being entered  *  * * * *cref='ParseException'> if the node analysis  * discovered errors  */  public virtual void
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }      /**     * <summary>Called</summary></pre>	* <summary>Called when exiting a parse tree node. </summary>	name='node'>the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */ public virtual void EnterProdOutC(Producti
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual void	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }      /**     * <summary>Called when adding a child to</summary></pre>	* <summary>Called when exiting a parse tree node. </summary>	name='node'>the node being entered  *  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */ public virtual void EnterProdOutC(Producti
*     * <param name="node"/> the parent node     * <param name="child"/> the child node, or null     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception> */     public virtual	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }      /**     * <summary>Called</summary></pre>	* <summary>Called when exiting a parse tree node. </summary>	<pre>name='node'&gt;the node being entered</pre>
*     * \	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }      /**     * <summary>Called when adding a child to a parse tree</summary></pre>	* <summary>Called when exiting a parse tree node. </summary> *     * <param name="node"/> the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree should be created</returns> *	<pre>name='node'&gt;the node being entered</pre>
*     * \	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }      /**     * <summary>Called when adding a child to a parse tree</summary></pre>	* <summary>Called when exiting a parse tree node. </summary> *     * <param name="node"/> the node being exited     *     * <returns>the node to add to the parse tree, or     * null if no parse tree should be created</returns> *     * <exception< td=""><td><pre>name='node'&gt;the node being entered</pre></td></exception<>	<pre>name='node'&gt;the node being entered</pre>
*     * \	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {       return node;     }      /**       * <summary>Called when adding a child to a parse tree     * node.</summary>     *</pre>	* <summary>Called when exiting a parse tree node. </summary>	name='node'>the node being entered
*     * \	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors     */     public virtual Node ExitProdPostval(Production node) {         return node;     }      /**      * <summary>Called when adding a child to a parse tree     * node.</summary>     *     * <param< pre=""></param<></pre>	* <summary>Called when exiting a parse tree node. </summary>	name='node'>the node being entered
<pre>*     * <param name="node"/>the parent node     * <param name="child"/>the child node, or null     *     * <exception cref="ParseException"> if the node analysis     * discovered errors</exception>     */     public virtual void ChildProdPrint(Product ion node, Node child) { node.AddChild(child); }</pre>	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors</pre> errors errors errors /*/ public virtual Node ExitProdPostval (Production node) {     return node; }  /**     * <summary>Called when adding a child to a parse tree     * node. /summary&gt;     *  * <param name="node"/>the parent node     * <param< pre=""></param<></summary>	* <summary>Called when exiting a parse tree node. </summary>	name='node'>the node being entered
*     * \	<pre>cref='ParseException'&gt; if the node analysis     * discovered errors</pre> errors errors errors errors public virtual  Node ExitProdPostval (Production node) {     return node; }  /**     * <summary>Called when adding a child to a parse tree     * node. </summary> *     * <param name="node"/> the parent node	* <summary>Called when exiting a parse tree node. </summary>	name='node'>the node being entered

* <returns>the</returns>	*/		* <param< th=""></param<>
node to add to the	public virtual	node.AddChild(child);	name='node'>the parent
parse tree, or	void	}	node
* null	EnterProdStructC(Produ	J	* <param< td=""></param<>
if no parse tree	ction node) {	/**	name='child'>the child
should be	}	* <summary>Called</summary>	node, or null
created	J	when entering a parse	*
*	/**	tree node.	* <exception< td=""></exception<>
* <exception< td=""><td>* <summary>Called</summary></td><td>*</td><td>cref='ParseException'&gt;</td></exception<>	* <summary>Called</summary>	*	cref='ParseException'>
cref='ParseException'>	when exiting a parse	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
if the node analysis	tree node.	name='node'>the node	* discovered
* discovered	*	being entered	errors
errors	* <param< td=""><td>*</td><td>*/</td></param<>	*	*/
*/	name='node'>the node	* <exception< td=""><td>public virtual</td></exception<>	public virtual
public virtual	being exited	cref='ParseException'>	void
Node	*	if the node analysis	ChildProdConcatLit(Pro
ExitProdOutC(Productio	* <returns>the</returns>	* discovered	duction node, Node
n node) {	node to add to the	errors	child) {
return node;	parse tree, or	*/	child) (
leturn node,	* null	public virtual	node.AddChild(child);
J	if no parse tree	void	loue: Addeniii (chiiid),
/**	should be	EnterProdConcatLit(Pro	J
* <summary>Called</summary>	created	duction node) {	/**
when adding a child to	*	duction hode) {	* <summary>Called</summary>
a parse tree	* <exception< td=""><td>J</td><td>when entering a parse</td></exception<>	J	when entering a parse
* node.	cref='ParseException'>	/**	tree node.
* node. // Summary/	if the node analysis	* <summary>Called</summary>	tree node. // Summary/
* <param< td=""><td>* discovered</td><td>when exiting a parse</td><td>* <param< td=""></param<></td></param<>	* discovered	when exiting a parse	* <param< td=""></param<>
name='node'>the parent	errors	tree node.	name='node'>the node
node	*/	*	being entered (/param)
* <pre>* <pre>*</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>	public virtual	* <param< td=""><td>*</td></param<>	*
name='child'>the child	Node	name='node'>the node	* <exception< td=""></exception<>
node, or null	ExitProdStructC(Produc	being exited	cref='ParseException'>
*	tion node) {	being exited / param/	if the node analysis
* <exception< td=""><td>return node;</td><td>* <returns>the</returns></td><td>* discovered</td></exception<>	return node;	* <returns>the</returns>	* discovered
cref='ParseException'>	}	node to add to the	errors
if the node analysis	J	parse tree, or	*/
* discovered	/**	* null	public virtual
errors	* <summary>Called</summary>	if no parse tree	void
*/	when adding a child to	should be	EnterProdScan(Producti
public virtual	a parse tree	created	on node) {
void	* node.	*	}
ChildProdOutC(Producti	*	* <exception< td=""><td>,</td></exception<>	,
on node, Node child) {	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>/**</td></param<>	<pre>cref='ParseException'&gt;</pre>	/**
on nege, nege entra, (	name='node'>the parent	if the node analysis	* <summary>Called</summary>
node.AddChild(child);	node	* discovered	when exiting a parse
}	* <pre>* <pre>*</pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre></pre>	errors	tree node.
,	name='child'>the child	*/	*
/**	node, or null	public virtual	* <param< td=""></param<>
* <summary>Called</summary>	*	Node	name='node'>the node
when entering a parse	* <exception< td=""><td>ExitProdConcatLit(Prod</td><td>being exited</td></exception<>	ExitProdConcatLit(Prod	being exited
tree node.	<pre>cref='ParseException'&gt;</pre>	uction node) {	*
*	if the node analysis	return node;	* <returns>the</returns>
* <param< td=""><td>* discovered</td><td>}</td><td>node to add to the</td></param<>	* discovered	}	node to add to the
name='node'>the node	errors	,	parse tree, or
being entered	*/	/**	* null
*	public virtual	* <summary>Called</summary>	if no parse tree
* <exception< td=""><td>void</td><td>when adding a child to</td><td>should be</td></exception<>	void	when adding a child to	should be
cref='ParseException'>	ChildProdStructC(Produ	a parse tree	created
if the node analysis	ction node, Node	* node.	*
* discovered	child) {	*	
//	·		

 $\verb|errors| < | exception >$ 

de /	de /	de /	
<pre>* <exception cref="ParseException"></exception></pre>	* <summary>Called when exiting a parse</summary>	* <param name="node"/> the node	* discovered errors
if the node analysis	tree node.	being entered <pre>/param&gt;</pre>	*/
* discovered	*	*	public virtual
errors	* <param< td=""><td>* <exception< td=""><td>void</td></exception<></td></param<>	* <exception< td=""><td>void</td></exception<>	void
*/	name='node'>the node	<pre>cref='ParseException'&gt;</pre>	ChildProdForState(Prod
public virtual	being exited	if the node analysis	uction node, Node
Node	*	* discovered	child) {
ExitProdScan(Productio	* <returns>the</returns>	errors	
n node) {	node to add to the	*/	node.AddChild(child);
return node;	parse tree, or	public virtual	}
}	* null	void	/state
/**	if no parse tree should be	EnterProdForState(Prod uction node) {	/** * <summary>Called</summary>
* <summary>Called</summary>	created	detion hode) (	when entering a parse
when adding a child to	*	J	tree node.
a parse tree	* <exception< td=""><td>/**</td><td>*</td></exception<>	/**	*
* node.	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>	* <param< td=""></param<>
*	if the node analysis	when exiting a parse	name='node'>the node
* <param< td=""><td>* discovered</td><td>tree node.</td><td>being entered</td></param<>	* discovered	tree node.	being entered
name='node'>the parent	errors	*	*
node	*/	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
* <param< td=""><td>public virtual</td><td>name='node'&gt;the node</td><td><pre>cref='ParseException'&gt;</pre></td></param<>	public virtual	name='node'>the node	<pre>cref='ParseException'&gt;</pre>
name='child'>the child	Node	being exited	if the node analysis
node, or null	ExitProdExtI(Productio	*	* discovered
*	n node) {	* <returns>the</returns>	errors
<pre>* <exception cref="ParseException"></exception></pre>	return node;	node to add to the parse tree, or	*/ public virtual
if the node analysis	J	* null	void
* discovered	/**	if no parse tree	EnterProdForstatement(
errors	* <summary>Called</summary>	should be	Production node) {
*/	when adding a child to	created	}
public virtual	a parse tree	*	
void	* node.	* <exception< td=""><td>/**</td></exception<>	/**
ChildProdScan(Producti	*	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>
on node, Node child) {	* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td></param<>	if the node analysis	when exiting a parse
	name='node'>the parent	* discovered	tree node.
node.AddChild(child);	node	errors	*
}	* <param -'="" 1:11'<="" td=""/> <td>*/</td> <td>* <param< td=""></param<></td>	*/	* <param< td=""></param<>
/**	name='child'>the child node, or null	public virtual Node	name='node'>the node being exited
* <summary>Called</summary>	*	ExitProdForState(Produ	*
when entering a parse	* <exception< td=""><td>ction node) {</td><td>* <returns>the</returns></td></exception<>	ction node) {	* <returns>the</returns>
tree node.	<pre>cref='ParseException'&gt;</pre>	return node;	node to add to the
*	if the node analysis	}	parse tree, or
* <param< td=""><td>* discovered</td><td></td><td>* null</td></param<>	* discovered		* null
name='node'>the node	errors	/**	if no parse tree
being entered	*/	* <summary>Called</summary>	should be
*	public virtual	when adding a child to	created
* <exception< td=""><td>void</td><td>a parse tree</td><td>*</td></exception<>	void	a parse tree	*
<pre>cref='ParseException'&gt;</pre>	ChildProdExtI(Producti	* node.	* <exception< td=""></exception<>
if the node analysis	on node, Node child) {	*	cref='ParseException'>
* discovered errors	node.AddChild(child);	* <param name="node"/> the parent	if the node analysis * discovered
*/	loue. Addellita (clitta),	name- node /the parent node	* discovered errors
public virtual	,	* <param< td=""><td>*/</td></param<>	*/
void	/**	name='child'>the child	public virtual
EnterProdExtI(Producti	* <summary>Called</summary>	node, or null	Node
on node) {	when entering a parse	*	ExitProdForstatement(P
}	tree node.	* <exception< td=""><td>roduction node) {</td></exception<>	roduction node) {
	*	<pre>cref='ParseException'&gt;</pre>	return node;
/**		if the node analysis	}

	* null	EnterProdMntCond(Produ	/**
/**	if no parse tree	ction node) {	* <summary>Called</summary>
* <summary>Called</summary>	should be	}	when entering a parse
when adding a child to	created		tree node.
a parse tree	*	/**	*
* node.	* <exception< td=""><td>* <summary>Called</summary></td><td>* <param< td=""></param<></td></exception<>	* <summary>Called</summary>	* <param< td=""></param<>
*	<pre>cref='ParseException'&gt;</pre>	when exiting a parse	name='node'>the node
* <param< td=""><td>if the node analysis</td><td>tree node.</td><td>being entered</td></param<>	if the node analysis	tree node.	being entered
name='node'>the parent	* discovered	*	*
node	errors	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
* <param< td=""><td>*/</td><td>name='node'&gt;the node</td><td><pre>cref='ParseException'&gt;</pre></td></param<>	*/	name='node'>the node	<pre>cref='ParseException'&gt;</pre>
name='child'>the child	public virtual	being exited	if the node analysis
node, or null	Node	*	* discovered
*	ExitProdVall(Productio	* <returns>the</returns>	errors
* <exception< td=""><td>n node) {</td><td>node to add to the</td><td>*/</td></exception<>	n node) {	node to add to the	*/
<pre>cref='ParseException'&gt;</pre>	return node;	parse tree, or	public virtual
if the node analysis	}	* null	void
* discovered	,	if no parse tree	EnterProdMntCondT(Prod
errors	/**	should be	uction node) {
*/	* <summary>Called</summary>	created	}
public virtual	when adding a child to	*	,
void	a parse tree	* <exception< td=""><td>/**</td></exception<>	/**
ChildProdForstatement(	* node.	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>
Production node, Node	*	if the node analysis	when exiting a parse
child) {	* <param< td=""><td>* discovered</td><td>tree node.</td></param<>	* discovered	tree node.
AddCh:11/-h:11/.	name='node'>the parent	errors */	·
node. AddChild(child);	node * <param< td=""><td>*/ public virtual</td><td>* <param name='node'&gt;the node</param </td></param<>	*/ public virtual	* <param name='node'&gt;the node</param 
}	name='child'>the child	Node	being exited
/**	node, or null	ExitProdMntCond(Produc	*
* <summary>Called</summary>	*	tion node) {	* <returns>the</returns>
when entering a parse	* <exception< td=""><td>return node;</td><td>node to add to the</td></exception<>	return node;	node to add to the
tree node.	cref='ParseException'>	}	parse tree, or
*	if the node analysis	,	* null
* <param< td=""><td>* discovered</td><td>/**</td><td>if no parse tree</td></param<>	* discovered	/**	if no parse tree
name='node'>the node	errors	* <summary>Called</summary>	should be
being entered	*/	when adding a child to	created
*	public virtual	a parse tree	*
* <exception< td=""><td>void</td><td>* node.</td><td>* <exception< td=""></exception<></td></exception<>	void	* node.	* <exception< td=""></exception<>
<pre>cref='ParseException'&gt;</pre>	ChildProdVall(Producti	*	<pre>cref='ParseException'&gt;</pre>
if the node analysis	on node, Node child) {	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
* discovered		name='node'>the parent	* discovered
errors	<pre>node. AddChild(child);</pre>	node	errors
*/	}	* <param< td=""><td>*/</td></param<>	*/
public virtual		name='child'>the child	public virtual
void	/**	node, or null	Node
EnterProdVall(Producti	* <summary>Called</summary>	*	ExitProdMntCondT(Produ
on node) {	when entering a parse	* <exception< td=""><td>ction node) {</td></exception<>	ction node) {
}	tree node.	cref='ParseException'>	return node;
	*	if the node analysis	}
/**	* <pre>* <pre>* </pre></pre>	* discovered	/
* <summary>Called</summary>	name='node'>the node	errors	/**
when exiting a parse	being entered	*/	* <summary>Called</summary>
tree node.	* * <exception< td=""><td>public virtual void</td><td>when adding a child to</td></exception<>	public virtual void	when adding a child to
* * <param< td=""><td>* \exception cref='ParseException'&gt;</td><td>voia ChildProdMntCond(Produ</td><td>a parse tree * node.</td></param<>	* \exception cref='ParseException'>	voia ChildProdMntCond(Produ	a parse tree * node.
name='node'>the node	if the node analysis	ction node, Node	* node. \/ Summary/
being exited	* discovered	child) {	* <param< td=""></param<>
*	errors	OHITA/ (	name='node'>the parent
* <returns>the</returns>	*/	node.AddChild(child);	node
node to add to the	public virtual	}	noso , param
parse tree, or	void	,	
•			

. ,	. /		
* <param \+ba="" abild<="" nama="abild" td=""/> <td>*/</td> <td>* <pre>* <pre>param</pre></pre></td> <td>* <exception< td=""></exception<></td>	*/	* <pre>* <pre>param</pre></pre>	* <exception< td=""></exception<>
name='child'>the child node, or null	public virtual Node	name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis</pre>
node, or nully param/	ExitProdMnt(Production	*	* discovered
* <exception< td=""><td>node) {</td><td>* <returns>the</returns></td><td>errors</td></exception<>	node) {	* <returns>the</returns>	errors
cref='ParseException'>	return node;	node to add to the	*/
if the node analysis	}	parse tree, or	public virtual
* discovered	,	* null	void
errors	/**	if no parse tree	EnterProdIfcondition(P
*/	* <summary>Called</summary>	should be	roduction node) {
public virtual	when adding a child to	created	}
void	a parse tree	*	
ChildProdMntCondT(Prod	* node.	* <exception< td=""><td>/**</td></exception<>	/**
uction node, Node	*	cref='ParseException'>	* <summary>Called</summary>
child) {	* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td></param<>	if the node analysis	when exiting a parse
1 41101:11/1:11)	name='node'>the parent	* discovered	tree node.
node. AddChild(child);	node	errors	*
}	* <param name="child"/> the child	*/ public virtual	* <param name="node"/> the node
/**	node, or null	Node	being exited
* <summary>Called</summary>	*	ExitProdIfelse(Product	*
when entering a parse	* <exception< td=""><td>ion node) {</td><td>* <returns>the</returns></td></exception<>	ion node) {	* <returns>the</returns>
tree node.	<pre>cref='ParseException'&gt;</pre>	return node;	node to add to the
*	if the node analysis	}	parse tree, or
* <param< td=""><td>* discovered</td><td>•</td><td>* null</td></param<>	* discovered	•	* null
name='node'>the node	errors	/**	if no parse tree
being entered	*/	* <summary>Called</summary>	should be
*	public virtual	when adding a child to	created
* <exception< td=""><td>void</td><td>a parse tree</td><td>*</td></exception<>	void	a parse tree	*
cref='ParseException'>	ChildProdMnt(Productio	* node.	* <exception< td=""></exception<>
if the node analysis	n node, Node child) {	*	cref='ParseException'>
* discovered		* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
errors	node.AddChild(child);	name='node'>the parent	* discovered
*/	}	node	errors
public virtual	/**	* <param name='child'&gt;the child</param 	*/
void EnterProdMnt(Productio	* <summary>Called</summary>		public virtual Node
n node) {	when entering a parse	node, or null *	ExitProdIfcondition(Pr
n node/ (	tree node.	* <exception< td=""><td>oduction node) {</td></exception<>	oduction node) {
J	*	cref='ParseException'>	return node:
/**	* <param< td=""><td>if the node analysis</td><td>}</td></param<>	if the node analysis	}
* <summary>Called</summary>	name='node'>the node	* discovered	,
when exiting a parse	being entered	errors	/**
tree node.	*	*/	* <summary>Called</summary>
*	* <exception< td=""><td>public virtual</td><td>when adding a child to</td></exception<>	public virtual	when adding a child to
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>void</td><td>a parse tree</td></param<>	<pre>cref='ParseException'&gt;</pre>	void	a parse tree
name='node'>the node	if the node analysis	ChildProdIfelse(Produc	* node.
being exited	* discovered	tion node, Node child)	*
*	errors	{	* <param< td=""></param<>
* <returns>the</returns>	*/		name='node'>the parent
node to add to the	public virtual	node.AddChild(child);	node
parse tree, or	void	}	* <param< td=""></param<>
* null	EnterProdIfelse (Produc	/steste	name='child'>the child
if no parse tree should be	tion node) {	/** * <summary>Called</summary>	node, or null *
created	ſ	when entering a parse	* <exception< td=""></exception<>
created <td>/**</td> <td>tree node. </td> <td><pre>cref='ParseException'&gt;</pre></td>	/**	tree node.	<pre>cref='ParseException'&gt;</pre>
* <exception< td=""><td>* <summary>Called</summary></td><td>*</td><td>if the node analysis</td></exception<>	* <summary>Called</summary>	*	if the node analysis
cref='ParseException'>	when exiting a parse	* <param< td=""><td>* discovered</td></param<>	* discovered
if the node analysis	tree node.	name='node'>the node	errors
* discovered	*	being entered (/param)	*/
errors		*	,
•			

public virtual	* <summary>Called</summary>	should be	EnterProdElseifstateme
void	when adding a child to	created	nt(Production node) {
ChildProdIfcondition(P	a parse tree	*	}
roduction node, Node	* node.	* <exception< td=""><td></td></exception<>	
child) {	*	<pre>cref='ParseException'&gt;</pre>	/**
	* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td></param<>	if the node analysis	* <summary>Called</summary>
<pre>node. AddChild(child);</pre>	name='node'>the parent	* discovered	when exiting a parse
}	node	errors	tree node.
	* <param< td=""><td>*/</td><td>*</td></param<>	*/	*
/**	name='child'>the child	public virtual	* <param< td=""></param<>
* <summary>Called</summary>	node, or null	Node	name='node'>the node
when entering a parse	*	ExitProdElseif(Product	being exited
tree node.	* <exception< td=""><td>ion node) {</td><td>*</td></exception<>	ion node) {	*
*	cref='ParseException'>	return node;	* <returns>the</returns>
* <param< td=""><td>if the node analysis</td><td>}</td><td>node to add to the</td></param<>	if the node analysis	}	node to add to the
name='node'>the node	* discovered		parse tree, or
being entered	errors	/**	* null
*	*/	* <summary>Called</summary>	if no parse tree
* <exception< td=""><td>public virtual</td><td>when adding a child to</td><td>should be</td></exception<>	public virtual	when adding a child to	should be
cref='ParseException'>	void	a parse tree	created
if the node analysis	ChildProdIfstatement(P	* node.	*
* discovered	roduction node, Node	*	* <exception< td=""></exception<>
errors	child) {	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td></param<>	<pre>cref='ParseException'&gt;</pre>
*/	, (	name='node'>the parent	if the node analysis
public virtual	node.AddChild(child);	node	* discovered
void	}	* <param< td=""><td>errors</td></param<>	errors
EnterProdIfstatement(P	,	name='child'>the child	*/
roduction node) {	/**	node, or null	public virtual
}	* <summary>Called</summary>	*	Node
,	when entering a parse	* <exception< td=""><td>ExitProdElseifstatemen</td></exception<>	ExitProdElseifstatemen
/**	tree node.	<pre>cref='ParseException'&gt;</pre>	t (Production node) {
* <summary>Called</summary>	*	if the node analysis	return node;
when exiting a parse	* <param< td=""><td>* discovered</td><td>}</td></param<>	* discovered	}
tree node.	name='node'>the node	errors	,
*	being entered	*/	/**
* <param< td=""><td>*</td><td>public virtual</td><td>* <summary>Called</summary></td></param<>	*	public virtual	* <summary>Called</summary>
name='node'>the node	* <exception< td=""><td>void</td><td>when adding a child to</td></exception<>	void	when adding a child to
being exited	cref='ParseException'>	ChildProdElseif(Produc	a parse tree
*	if the node analysis	tion node, Node child)	* node.
* <returns>the</returns>	* discovered	{	* Hode. \/ Summary/
node to add to the	errors	(	* <param< td=""></param<>
parse tree, or	*/	node.AddChild(child);	name='node'>the parent
* null	public virtual	loue: Addentita (chitta) ,	node
if no parse tree	void	J	* <param< td=""></param<>
should be	EnterProdElseif(Produc	/**	name='child'>the child
created	tion node) {	* <summary>Called</summary>	node, or null
*	tion node) (	when entering a parse	node, or null√param/ *
	J		
* <exception< td=""><td>/state</td><td>tree node. </td><td>* <exception< td=""></exception<></td></exception<>	/state	tree node.	* <exception< td=""></exception<>
cref='ParseException'>	/**	*	cref='ParseException'>
if the node analysis	* <summary>Called</summary>	* <param name="node"/> the node	if the node analysis
* discovered	when exiting a parse		* discovered
errors	tree node.	being entered	errors
*/	*	*	*/
public virtual	* <pre>* <pre>* </pre></pre>	* <exception< td=""><td>public virtual</td></exception<>	public virtual
Node	name='node'>the node	cref='ParseException'>	void
ExitProdIfstatement (Pr	being exited	if the node analysis	ChildProdElseifstateme
oduction node) {	*	* discovered	nt (Production node,
return node;	* <returns>the</returns>	errors	Node child) {
}	node to add to the	*/	1 41101 11 / 1 11
/	parse tree, or	public virtual	node.AddChild(child);
/**	* null	void	}
	if no parse tree		

/**	* <param< th=""><th>*/</th><th>* <param< th=""></param<></th></param<>	*/	* <param< th=""></param<>
* <summary>Called</summary>	name='child'>the child	public virtual	name='node'>the node
when entering a parse	node, or null	Node	being exited
tree node.	*	ExitProdElsestatement(	*
*	* <exception< td=""><td>Production node) {</td><td>* <returns>the</returns></td></exception<>	Production node) {	* <returns>the</returns>
* <param< td=""><td>cref='ParseException'&gt;</td><td>return node;</td><td>node to add to the</td></param<>	cref='ParseException'>	return node;	node to add to the
name='node'>the node	if the node analysis	}	parse tree, or
being entered	* discovered	J	* null
*	errors	/**	if no parse tree
* <exception< td=""><td>*/</td><td>* <summary>Called</summary></td><td>should be</td></exception<>	*/	* <summary>Called</summary>	should be
cref='ParseException'>	public virtual	when adding a child to	created
if the node analysis	void	a parse tree	*
* discovered	ChildProdElseState(Pro	* node.	* <exception< td=""></exception<>
errors	duction node, Node	*	<pre>cref='ParseException'&gt;</pre>
*/	child) {	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
public virtual		name='node'>the parent	* discovered
void	<pre>node. AddChild(child);</pre>	node	errors
EnterProdElseState(Pro	}	* <param< td=""><td>*/</td></param<>	*/
duction node) {		name='child'>the child	public virtual
}	/**	node, or null	Node
	* <summary>Called</summary>	*	ExitProdDowhile(Produc
/**	when entering a parse	* <exception< td=""><td>tion node) {</td></exception<>	tion node) {
* <summary>Called</summary>	tree node.	<pre>cref='ParseException'&gt;</pre>	return node;
when exiting a parse	*	if the node analysis	}
tree node.	* <param< td=""><td>* discovered</td><td></td></param<>	* discovered	
*	name='node'>the node	errors	/**
* <param< td=""><td>being entered</td><td>*/</td><td>* <summary>Called</summary></td></param<>	being entered	*/	* <summary>Called</summary>
name='node'>the node	*	public virtual	when adding a child to
being exited	* <exception< td=""><td>void</td><td>a parse tree</td></exception<>	void	a parse tree
*	<pre>cref='ParseException'&gt;</pre>	ChildProdElsestatement	* node.
* <returns>the</returns>	if the node analysis	(Production node, Node	*
node to add to the	* discovered	child) {	* <param< td=""></param<>
parse tree, or * null	errors	. 1 411(1:11/ 1:11)	name='node'>the parent
· Hall	*/ public virtual	node.AddChild(child);	node
if no parse tree should be	void	ſ	* <param name="child"/> the child
created	EnterProdElsestatement	/**	node, or null
*	(Production node) {	* <summary>Called</summary>	*
* <exception< td=""><td>(Troduction hode) (</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	(Troduction hode) (	when entering a parse	* <exception< td=""></exception<>
cref='ParseException'>	J	tree node.	cref='ParseException'>
if the node analysis	/**	*	if the node analysis
* discovered	* <summary>Called</summary>	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	when exiting a parse	name='node'>the node	errors
*/	tree node.	being entered	*/
public virtual	*	*	public virtual
Node	* <param< td=""><td>* <exception< td=""><td>void</td></exception<></td></param<>	* <exception< td=""><td>void</td></exception<>	void
ExitProdElseState(Prod	name='node'>the node	<pre>cref='ParseException'&gt;</pre>	ChildProdDowhile(Produ
uction node) {	being exited	if the node analysis	ction node, Node
return node;	*	* discovered	child) {
}	* <returns>the</returns>	errors	
	node to add to the	*/	<pre>node. AddChild(child);</pre>
/**	parse tree, or	public virtual	}
* <summary>Called</summary>	* null	void	
when adding a child to	if no parse tree	EnterProdDowhile (Produ	/**
a parse tree	should be	ction node) {	* <summary>Called</summary>
* node.	created	}	when entering a parse
*	*	/state	tree node.
* <param< td=""><td>* <exception< td=""><td>/**</td><td>*</td></exception<></td></param<>	* <exception< td=""><td>/**</td><td>*</td></exception<>	/**	*
name='node'>the parent node	<pre>cref='ParseException'&gt; if the node analysis</pre>	* <summary>Called</summary>	* <param name='node'&gt;the node</param 
noue/param/	<pre>11 the node analysis     * discovered</pre>	when exiting a parse tree node.	
	* discovered errors	tree node. summary/</td <td>being entered *</td>	being entered *
	ellolg// evcehtloll/	**	T

* <exception< th=""><th>public virtual</th><th>* <summary>Called</summary></th><th>should be</th></exception<>	public virtual	* <summary>Called</summary>	should be
cref='ParseException'>	void	when adding a child to	created
if the node analysis	ChildProdDostatement(P	a parse tree	*
* discovered	roduction node, Node	* node.	* <exception< td=""></exception<>
errors	child) {	*	<pre>cref='ParseException'&gt;</pre>
*/	oniia, (	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
public virtual	node.AddChild(child);	name='node'>the parent	* discovered
void	}	node	errors
EnterProdDostatement(P	,	* <param< td=""><td>*/</td></param<>	*/
roduction node) {	/**	name='child'>the child	public virtual
}	* <summary>Called</summary>	node, or null	Node
•	when entering a parse	*	ExitProdWhilestatement
/**	tree node.	* <exception< td=""><td>(Production node) {</td></exception<>	(Production node) {
* <summary>Called</summary>	*	cref='ParseException'>	return node;
when exiting a parse	* <param< td=""><td>if the node analysis</td><td>}</td></param<>	if the node analysis	}
tree node.	name='node'>the node	* discovered	•
*	being entered	errors	/**
* <param< td=""><td>*</td><td>*/</td><td>* <summary>Called</summary></td></param<>	*	*/	* <summary>Called</summary>
name='node'>the node	* <exception< td=""><td>public virtual</td><td>when adding a child to</td></exception<>	public virtual	when adding a child to
being exited	cref='ParseException'>	void	a parse tree
*	if the node analysis	ChildProdWhileState(Pr	* node.
* <returns>the</returns>	* discovered	oduction node, Node	*
node to add to the	errors	child) {	* <param< td=""></param<>
parse tree, or	*/		name='node'>the parent
* null	public virtual	<pre>node. AddChild(child);</pre>	node
if no parse tree	void	}	* <param< td=""></param<>
should be	EnterProdWhileState(Pr	•	name='child'>the child
created	oduction node) {	/**	node, or null
*	}	* <summary>Called</summary>	*
* <exception< td=""><td>,</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	,	when entering a parse	* <exception< td=""></exception<>
cref='ParseException'>	/**	tree node.	cref='ParseException'>
if the node analysis	* <summary>Called</summary>	*	if the node analysis
* discovered	when exiting a parse	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	tree node.	name='node'>the node	errors
*/	*	being entered	*/
public virtual	* <param< td=""><td>*</td><td>public virtual</td></param<>	*	public virtual
Node	name='node'>the node	* <exception< td=""><td>void</td></exception<>	void
ExitProdDostatement(Pr	being exited	<pre>cref='ParseException'&gt;</pre>	ChildProdWhilestatemen
oduction node) {	*	if the node analysis	t(Production node,
return node;	* <returns>the</returns>	* discovered	Node child) {
}	node to add to the	errors	
	parse tree, or	*/	<pre>node.AddChild(child);</pre>
/**	* null	public virtual	}
* <summary>Called</summary>	if no parse tree	void	
when adding a child to	should be	EnterProdWhilestatemen	/**
a parse tree	created	t(Production node) {	* <summary>Called</summary>
* node.	*	}	when entering a parse
*	* <exception< td=""><td></td><td>tree node.</td></exception<>		tree node.
* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>/**</td><td>*</td></param<>	<pre>cref='ParseException'&gt;</pre>	/**	*
name='node'>the parent	if the node analysis	* <summary>Called</summary>	* <param< td=""></param<>
node	* discovered	when exiting a parse	name='node'>the node
* <param< td=""><td>errors</td><td>tree node.</td><td>being entered</td></param<>	errors	tree node.	being entered
name='child'>the child	*/	*	*
node, or null	public virtual	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
*	Node	name='node'>the node	<pre>cref='ParseException'&gt;</pre>
* <exception< td=""><td>ExitProdWhileState(Pro</td><td>being exited</td><td>if the node analysis</td></exception<>	ExitProdWhileState(Pro	being exited	if the node analysis
<pre>cref='ParseException'&gt;</pre>	duction node) {	*	* discovered
if the node analysis	return node;	* <returns>the</returns>	errors
* discovered	}	node to add to the	*/
errors		parse tree, or	public virtual
*/	/**	* null	void
		if no parse tree	

EnterProdSwitchState(P	/**	* <param< th=""><th>*/</th></param<>	*/
roduction node) {	* <summary>Called</summary>	name='child'>the child	public virtual
}	when entering a parse	node, or null	Node
,	tree node.	*	ExitProdDef(Production
/**	*	* <exception< td=""><td>node) {</td></exception<>	node) {
* <summary>Called</summary>	* <param< td=""><td>cref='ParseException'&gt;</td><td>return node;</td></param<>	cref='ParseException'>	return node;
when exiting a parse	name='node'>the node	if the node analysis	}
tree node.	being entered	* discovered	,
*	*	errors	/**
* <param< td=""><td>* <exception< td=""><td>*/</td><td>* <summary>Called</summary></td></exception<></td></param<>	* <exception< td=""><td>*/</td><td>* <summary>Called</summary></td></exception<>	*/	* <summary>Called</summary>
name='node'>the node	cref='ParseException'>	public virtual	when adding a child to
being exited	if the node analysis	void	a parse tree
*	* discovered	ChildProdCaseState(Pro	* node.
* <returns>the</returns>	errors	duction node, Node	*
node to add to the	*/	child) {	* <param< td=""></param<>
parse tree, or	public virtual		name='node'>the parent
* null	void	<pre>node.AddChild(child);</pre>	node
if no parse tree	EnterProdCaseState(Pro	}	* <param< td=""></param<>
should be	duction node) {		name='child'>the child
created	}	/**	node, or null
*		* <summary>Called</summary>	*
* <exception< td=""><td>/**</td><td>when entering a parse</td><td>* <exception< td=""></exception<></td></exception<>	/**	when entering a parse	* <exception< td=""></exception<>
cref='ParseException'>	* <summary>Called</summary>	tree node.	cref='ParseException'>
if the node analysis	when exiting a parse	*	if the node analysis
* discovered	tree node.	* <param< td=""><td>* discovered</td></param<>	* discovered
errors	*	name='node'>the node	errors
*/	* <param< td=""><td>being entered</td><td>*/</td></param<>	being entered	*/
public virtual	name='node'>the node	*	public virtual
Node	being exited	* <exception< td=""><td>void</td></exception<>	void
ExitProdSwitchState (Pr	*	<pre>cref='ParseException'&gt;</pre>	ChildProdDef(Productio
oduction node) {	* <returns>the</returns>	if the node analysis	n node, Node child) {
return node;	node to add to the	* discovered	1 41101:11(1:11)
}	parse tree, or * null	errors	node.AddChild(child);
/**	11011	*/	}
* <summary>Called</summary>	if no parse tree should be	public virtual void	/**
when adding a child to	created	EnterProdDef(Productio	* <summary>Called</summary>
a parse tree	*	n node) {	when entering a parse
* node.	* <exception< td=""><td>1 node) {</td><td>tree node. </td></exception<>	1 node) {	tree node.
*	cref='ParseException'>	J	*
* <param< td=""><td>if the node analysis</td><td>/**</td><td>* <param< td=""></param<></td></param<>	if the node analysis	/**	* <param< td=""></param<>
name='node'>the parent	* discovered	* <summary>Called</summary>	name='node'>the node
node	errors	when exiting a parse	being entered
* <pre>* <pre>* </pre></pre>	*/	tree node.	*
name='child'>the child	public virtual	*	* <exception< td=""></exception<>
node, or null	Node	* <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
*	ExitProdCaseState(Prod	name='node'>the node	if the node analysis
* <exception< td=""><td>uction node) {</td><td>being exited</td><td>* discovered</td></exception<>	uction node) {	being exited	* discovered
cref='ParseException'>	return node;	*	errors
if the node analysis	}	* <returns>the</returns>	*/
* discovered		node to add to the	public virtual
errors	/**	parse tree, or	void
*/	* <summary>Called</summary>	* null	EnterProdCasestatement
public virtual	when adding a child to	if no parse tree	(Production node) {
void	a parse tree	should be	}
ChildProdSwitchState(P	* node.	created	
roduction node, Node	*	*	/**
child) {	* <param< td=""><td>* <exception< td=""><td>* <summary>Called</summary></td></exception<></td></param<>	* <exception< td=""><td>* <summary>Called</summary></td></exception<>	* <summary>Called</summary>
node AddCL:11/-L:11	name='node'>the parent	cref='ParseException'>	when exiting a parse
node.AddChild(child);	node	if the node analysis	tree node.
Ĵ		<pre>* discovered errors</pre>	*
		errorg// exception/	

No. / 12 - 14 - 12			# /C-11-1
* <param name='node'&gt;the node</param 	<pre>* <exception cref="ParseException"></exception></pre>	public virtual void	* <summary>Called when adding a child to</summary>
being exited	if the node analysis	ChildProdMathOp(Produc	a parse tree
*	* discovered	tion node, Node child)	* node.
* <returns>the</returns>	errors	{	*
node to add to the	*/		* <param< td=""></param<>
parse tree, or	public virtual	<pre>node. AddChild(child);</pre>	name='node'>the parent
* null	void	}	node
if no parse tree	EnterProdMathOp(Produc		* <param< td=""></param<>
should be	tion node) {	/**	name='child'>the child
created	}	* <summary>Called</summary>	node, or null
*		when entering a parse	*
* <exception< td=""><td>/**</td><td>tree node.</td><td>* <exception< td=""></exception<></td></exception<>	/**	tree node.	* <exception< td=""></exception<>
cref='ParseException'>	* <summary>Called</summary>	*	cref='ParseException'>
if the node analysis	when exiting a parse	* <pre>* <pre>* </pre></pre>	if the node analysis
* discovered	tree node.	name='node'>the node	* discovered
errors */		being entered *	errors */
public virtual	* <param name="node"/> the node	* <exception< td=""><td>≁/ public virtual</td></exception<>	≁/ public virtual
Node	being exited	cref='ParseException'>	void
ExitProdCasestatement(	*	if the node analysis	ChildProdOperCond(Prod
Production node) {	* <returns>the</returns>	* discovered	uction node, Node
return node;	node to add to the	errors	child) {
}	parse tree, or	*/	
	* null	public virtual	<pre>node. AddChild(child);</pre>
/**	if no parse tree	void	}
* <summary>Called</summary>	should be	EnterProdOperCond(Prod	
when adding a child to	created	uction node) {	/**
a parse tree	*	}	* <summary>Called</summary>
* node.	* <exception< td=""><td></td><td>when entering a parse</td></exception<>		when entering a parse
*	cref='ParseException'>	/**	tree node.
* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td><td>*</td></param<>	if the node analysis	* <summary>Called</summary>	*
name='node'>the parent	* discovered	when exiting a parse	* <param< td=""></param<>
node	errors	tree node.	name='node'>the node
* <param< td=""><td>*/</td><td>*</td><td>being entered</td></param<>	*/	*	being entered
name='child'>the child	public virtual	* <pre> * <pre> * </pre></pre>	*
node, or null	Node ExitProdMathOp(Product	name='node'>the node being exited	<pre>* <exception cref="ParseException"></exception></pre>
* <exception< td=""><td>ion node) {</td><td>*</td><td>if the node analysis</td></exception<>	ion node) {	*	if the node analysis
cref='ParseException'>	return node;	* <returns>the</returns>	* discovered
if the node analysis	}	node to add to the	errors
* discovered	,	parse tree, or	*/
errors	/**	* null	public virtual
*/	* <summary>Called</summary>	if no parse tree	void
public virtual	when adding a child to	should be	EnterProdOperCondChoic
void	a parse tree	created	e(Production node) {
ChildProdCasestatement	* node.	*	}
(Production node, Node	*	* <exception< td=""><td></td></exception<>	
child) {	* <param< td=""><td><pre>cref='ParseException'&gt;</pre></td><td>/**</td></param<>	<pre>cref='ParseException'&gt;</pre>	/**
	name='node'>the parent	if the node analysis	* <summary>Called</summary>
node.AddChild(child);	node	* discovered	when exiting a parse
}	* <param< td=""><td>errors</td><td>tree node.</td></param<>	errors	tree node.
,	name='child'>the child	*/	*
/**	node, or null	public virtual	* <param< td=""></param<>
* <summary>Called</summary>	*	Node	name='node'>the node
when entering a parse	* <exception< td=""><td>ExitProdOperCond(Produ</td><td>being exited *</td></exception<>	ExitProdOperCond(Produ	being exited *
tree node.	cref='ParseException'>	ction node) {	*  * <returns>the</returns>
* * <param< td=""><td>if the node analysis * discovered</td><td>return node;</td><td>* <returns tne<br="">node to add to the</returns></td></param<>	if the node analysis * discovered	return node;	* <returns tne<br="">node to add to the</returns>
* \param name='node'>the node	* discovered errors	J	parse tree, or
being entered (/param)	*/	/**	* null
*	•,	,	if no parse tree
			F

1 111		/	
should be	EnterProdOperSym(Produ	/**	* <param< td=""></param<>
created *	ction node) {	* <summary>Called when entering a parse</summary>	name='child'>the child node, or null
* <exception< td=""><td>}</td><td>tree node. </td><td>node, or nully param/</td></exception<>	}	tree node.	node, or nully param/
cref='ParseException'>	/**	*	* <exception< td=""></exception<>
if the node analysis	* <summary>Called</summary>	* <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
* discovered	when exiting a parse	name='node'>the node	if the node analysis
errors	tree node.	being entered	* discovered
*/	*	*	errors
public virtual	* <param< td=""><td>* <exception< td=""><td>*/</td></exception<></td></param<>	* <exception< td=""><td>*/</td></exception<>	*/
Node	name='node'>the node	<pre>cref='ParseException'&gt;</pre>	public virtual
ExitProdOperCondChoice	being exited	if the node analysis	void
(Production node) {	*	* discovered	ChildProdOperEq(Produc
return node;	* <returns>the</returns>	errors	tion node, Node child)
}	node to add to the	*/	{
	parse tree, or	public virtual	1 41101:11/1:11
/**	* null	void	node.AddChild(child);
* <summary>Called</summary>	if no parse tree should be	EnterProdOperEq(Produc tion node) {	}
when adding a child to a parse tree	created	tion node) (	/**
* node.	*	J	* <summary>Called</summary>
* node. // Summary/	* <exception< td=""><td>/**</td><td>when entering a parse</td></exception<>	/**	when entering a parse
* <param< td=""><td>cref='ParseException'&gt;</td><td>* <summary>Called</summary></td><td>tree node. </td></param<>	cref='ParseException'>	* <summary>Called</summary>	tree node.
name='node'>the parent	if the node analysis	when exiting a parse	*
node	* discovered	tree node.	* <param< td=""></param<>
* <param< td=""><td>errors</td><td>*</td><td>name='node'&gt;the node</td></param<>	errors	*	name='node'>the node
name='child'>the child	*/	* <param< td=""><td>being entered</td></param<>	being entered
node, or null	public virtual	name='node'>the node	*
*	Node	being exited	* <exception< td=""></exception<>
* <exception< td=""><td>ExitProdOperSym(Produc</td><td>*</td><td><pre>cref='ParseException'&gt;</pre></td></exception<>	ExitProdOperSym(Produc	*	<pre>cref='ParseException'&gt;</pre>
<pre>cref='ParseException'&gt;</pre>	tion node) {	* <returns>the</returns>	if the node analysis
if the node analysis	return node;	node to add to the	* discovered
* discovered	}	parse tree, or	errors
errors	/starts	* null	*/
*/	/**	if no parse tree should be	public virtual void
public virtual void	* <summary>Called when adding a child to</summary>	created	EnterProdOperExtS(Prod
ChildProdOperCondChoic	a parse tree	*	uction node) {
e (Production node,	* node.	* <exception< td=""><td>letton node) (</td></exception<>	letton node) (
Node child) {	*	cref='ParseException'>	,
, (	* <param< td=""><td>if the node analysis</td><td>/**</td></param<>	if the node analysis	/**
node.AddChild(child);	name='node'>the parent	* discovered	* <summary>Called</summary>
}	node	errors	when exiting a parse
	* <param< td=""><td>*/</td><td>tree node.</td></param<>	*/	tree node.
/**	name='child'>the child	public virtual	*
* <summary>Called</summary>	node, or null	Node	* <param< td=""></param<>
when entering a parse	*	ExitProdOperEq(Product	name='node'>the node
tree node.	* <exception< td=""><td>ion node) {</td><td>being exited</td></exception<>	ion node) {	being exited
*	cref='ParseException'>	return node;	*
* <pre></pre>	if the node analysis	}	* <returns>the</returns>
name='node'>the node	* discovered	/**	node to add to the
being entered	errors */	* <summary>Called</summary>	parse tree, or * null
* <exception< td=""><td>public virtual</td><td>when adding a child to</td><td>if no parse tree</td></exception<>	public virtual	when adding a child to	if no parse tree
cref='ParseException'>	void	a parse tree	should be
if the node analysis	ChildProdOperSym(Produ	* node.	created
* discovered	ction node, Node	*	*
errors	child) {	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
*/		name='node'>the parent	cref='ParseException'>
public virtual	node.AddChild(child);	node	if the node analysis
void	}		* discovered
			errors

*/	* <param< th=""><th>* (evention</th><th>nublic virtual</th></param<>	* (evention	nublic virtual
≁/ public virtual	name='node'>the node	<pre>* <exception cref="ParseException"></exception></pre>	public virtual void
Node	being exited	if the node analysis	ChildProdOperand(Produ
ExitProdOperExtS(Produ	*	* discovered	ction node, Node
ction node) {	* <returns>the</returns>	errors	child) {
return node:	node to add to the	*/	31114)
}	parse tree, or	public virtual	node.AddChild(child);
	* null	void	}
/**	if no parse tree	EnterProdOperand(Produ	
* <summary>Called</summary>	should be	ction node) {	/**
when adding a child to	created	}	* <summary>Called</summary>
a parse tree	*		when entering a parse
* node.	* <exception< td=""><td>/**</td><td>tree node.</td></exception<>	/**	tree node.
*	cref='ParseException'>	* <summary>Called</summary>	*
* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td><td>* <param< td=""></param<></td></param<>	if the node analysis	when exiting a parse	* <param< td=""></param<>
name='node'>the parent	* discovered	tree node.	name='node'>the node
node	errors	*	being entered
* <param< td=""><td>*/</td><td>* <pre> * <pre></pre></pre></td><td>*</td></param<>	*/	* <pre> * <pre></pre></pre>	*
name='child'>the child	public virtual	name='node'>the node	* <exception< td=""></exception<>
node, or null	Node	being exited	cref='ParseException'>
•	ExitProdOperExtRep(Pro	* * <returns>the</returns>	if the node analysis * discovered
<pre>* <exception cref="ParseException"></exception></pre>	<pre>duction node) {     return node;</pre>	node to add to the	* alscovered errors
if the node analysis	return node,	parse tree, or	*/
* discovered	J	* null	public virtual
errors	/**	if no parse tree	void
*/	* <summary>Called</summary>	should be	EnterProdSimMathOp(Pro
public virtual	when adding a child to	created	duction node) {
void	a parse tree	*	}
ChildProdOperExtS(Prod	* node.	* <exception< td=""><td>•</td></exception<>	•
uction node, Node	*	<pre>cref='ParseException'&gt;</pre>	/**
child) {	* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td></param<>	if the node analysis	* <summary>Called</summary>
	name='node'>the parent	* discovered	when exiting a parse
<pre>node. AddChild(child);</pre>	node	errors	tree node.
}	* <param< td=""><td>*/</td><td>*</td></param<>	*/	*
	name='child'>the child	public virtual	* <param< td=""></param<>
/**	node, or null	Node	name='node'>the node
* <summary>Called</summary>	*	ExitProdOperand (Produc	being exited
when entering a parse	* <exception< td=""><td>tion node) {</td><td>*</td></exception<>	tion node) {	*
tree node.	cref='ParseException'>	return node;	* <returns>the</returns>
*	if the node analysis	}	node to add to the
* <param< td=""><td>* discovered</td><td>/</td><td>parse tree, or</td></param<>	* discovered	/	parse tree, or
name='node'>the node	errors */	/** * <summary>Called</summary>	* null
being entered	*/ public virtual	* \summary/carred when adding a child to	if no parse tree should be
* <exception< td=""><td>void</td><td>a parse tree</td><td>created</td></exception<>	void	a parse tree	created
cref='ParseException'>	ChildProdOperExtRep(Pr	* node.	*
if the node analysis	oduction node, Node	*	* <exception< td=""></exception<>
* discovered	child) {	∗ <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
errors	chila) (	name='node'>the parent	if the node analysis
*/	node.AddChild(child);	node	* discovered
public virtual	}	* <param< td=""><td>errors</td></param<>	errors
void	•	name='child'>the child	*/
EnterProdOperExtRep(Pr	/**	node, or null	public virtual
oduction node) {	* <summary>Called</summary>	*	Node
}	when entering a parse	* <exception< td=""><td>ExitProdSimMathOp(Prod</td></exception<>	ExitProdSimMathOp(Prod
	tree node.	<pre>cref='ParseException'&gt;</pre>	uction node) {
/**	*	if the node analysis	return node;
* <summary>Called</summary>	* <param< td=""><td>* discovered</td><td>}</td></param<>	* discovered	}
when exiting a parse	name='node'>the node	errors	
tree node.	being entered	*/	/**
*	*		

	1111	E. t D 10 C IE . t / D	/state
<pre>* <summary>Called when adding a child to</summary></pre>	should be created	EnterProdOperCondExt(P roduction node) {	/** * <summary>Called</summary>
a parse tree	created/returns/	roduction hode) (	when entering a parse
* node.	* <exception< td=""><td>J</td><td>tree node. </td></exception<>	J	tree node.
* 110dC. \/ 3dmind1 y/	cref='ParseException'>	/**	*
* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td><td>* <param< td=""></param<></td></param<>	if the node analysis	* <summary>Called</summary>	* <param< td=""></param<>
name='node'>the parent	* discovered	when exiting a parse	name='node'>the node
node	errors	tree node.	being entered
* <pre>* <pre>* </pre></pre>	*/	*	*
name='child'>the child	public virtual	* <param< td=""><td>* <exception< td=""></exception<></td></param<>	* <exception< td=""></exception<>
node, or null	Node	name='node'>the node	cref='ParseException'>
*	ExitProdSMathExt(Produ	being exited	if the node analysis
* <exception< td=""><td>ction node) {</td><td>*</td><td>* discovered</td></exception<>	ction node) {	*	* discovered
<pre>cref='ParseException'&gt;</pre>	return node;	* <returns>the</returns>	errors
if the node analysis	}	node to add to the	*/
* discovered		parse tree, or	public virtual
errors	/**	* null	void
*/	* <summary>Called</summary>	if no parse tree	EnterProdRelOp(Product
public virtual	when adding a child to	should be	ion node) {
void	a parse tree	created	}
ChildProdSimMathOp(Pro	* node.	*	
duction node, Node	*	* <exception< td=""><td>/**</td></exception<>	/**
child) {	* <param< td=""><td><pre>cref='ParseException'&gt; if the node analysis</pre></td><td>* <summary>Called</summary></td></param<>	<pre>cref='ParseException'&gt; if the node analysis</pre>	* <summary>Called</summary>
node.AddChild(child);	name='node'>the parent node	* discovered	when exiting a parse tree node.
loue. Addentita (chitta),	noue√param * <param< td=""><td>errors</td><td>*</td></param<>	errors	*
J	name='child'>the child	*/	* <param< td=""></param<>
/**	node, or null	public virtual	name='node'>the node
* <summary>Called</summary>	*	Node	being exited
when entering a parse	* <exception< td=""><td>ExitProdOperCondExt(Pr</td><td>*</td></exception<>	ExitProdOperCondExt(Pr	*
tree node.	cref='ParseException'>	oduction node) {	* <returns>the</returns>
*	if the node analysis	return node;	node to add to the
* <param< td=""><td>* discovered</td><td>}</td><td>parse tree, or</td></param<>	* discovered	}	parse tree, or
name='node'>the node	errors		* null
being entered	*/	/**	if no parse tree
*	public virtual	* <summary>Called</summary>	should be
* <exception< td=""><td>void</td><td>when adding a child to</td><td>created</td></exception<>	void	when adding a child to	created
cref='ParseException'>	ChildProdSMathExt(Prod	a parse tree	*
if the node analysis	uction node, Node	* node.	* <exception< td=""></exception<>
* discovered errors	child) {	* * <param< td=""><td><pre>cref='ParseException'&gt; if the node analysis</pre></td></param<>	<pre>cref='ParseException'&gt; if the node analysis</pre>
*/	node.AddChild(child);	name='node'>the parent	* discovered
public virtual	node. Addentia (chiid),	name- node /the parent node	errors
void	J	noue√param * <param< td=""><td>*/</td></param<>	*/
EnterProdSMathExt(Prod	/**	name='child'>the child	public virtual
uction node) {	* <summary>Called</summary>	node, or null	Node
}	when entering a parse	*	ExitProdRelOp(Producti
	tree node.	* <exception< td=""><td>on node) {</td></exception<>	on node) {
/**	*	cref='ParseException'>	return node;
* <summary>Called</summary>	* <param< td=""><td>if the node analysis</td><td>}</td></param<>	if the node analysis	}
when exiting a parse	name='node'>the node	* discovered	
tree node.	being entered	errors	/**
*	*	*/	* <summary>Called</summary>
* <param< td=""><td>* <exception< td=""><td>public virtual</td><td>when adding a child to</td></exception<></td></param<>	* <exception< td=""><td>public virtual</td><td>when adding a child to</td></exception<>	public virtual	when adding a child to
name='node'>the node	cref='ParseException'>	void	a parse tree
being exited	if the node analysis	ChildProdOperCondExt(P	* node.
* * <returns>the</returns>	* discovered	roduction node, Node child) {	* \( \text{narar} \)
node to add to the	errors */	CIIIIu) (	* <param name="node"/> the parent
parse tree, or	public virtual	node.AddChild(child);	node
* null	void	}	1000 , param
if no parse tree		•	

public virtual Node ExitProdRelopExt(Produ	name='node'>the node being exited	<pre>cref='ParseException'&gt; if the node analysis</pre>
	being exited	if the node analysis
ExitProdRelopExt(Produ		
* - /	*	* discovered
ction node) {	* <returns>the</returns>	errors
return node;	node to add to the	*/
}	parse tree, or	public virtual
	* null	void
/**	if no parse tree	EnterProdLogOp(Product
* <summary>Called</summary>	should be	ion node) {
when adding a child to	created	}
a parse tree	*	
* node.	* <exception< td=""><td>/**</td></exception<>	/**
*	<pre>cref='ParseException'&gt;</pre>	* <summary>Called</summary>
* <param< td=""><td>if the node analysis</td><td>when exiting a parse</td></param<>	if the node analysis	when exiting a parse
name='node'>the parent	* discovered	tree node.
node	errors	*
* <param< td=""><td>*/</td><td>* <param< td=""></param<></td></param<>	*/	* <param< td=""></param<>
name='child'>the child	public virtual	name='node'>the node
node, or null	Node	being exited
*	ExitProdOp1(Production	*
* <exception< td=""><td>node) {</td><td>* <returns>the</returns></td></exception<>	node) {	* <returns>the</returns>
<pre>cref='ParseException'&gt;</pre>	return node;	node to add to the
if the node analysis	}	parse tree, or
* discovered		* null
errors	/**	if no parse tree
*/	* <summary>Called</summary>	should be
public virtual	when adding a child to	created
void	a parse tree	*
ChildProdRelopExt(Prod	* node.	* <exception< td=""></exception<>
uction node, Node	*	cref='ParseException'>
child) {	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
	name='node'>the parent	* discovered
node.AddChild(child);		errors
}	* <param< td=""><td>*/</td></param<>	*/
	name='child'>the child	public virtual
/**	node, or null	Node
* <summary>Called</summary>	*	ExitProdLogOp(Producti
	* <exception< td=""><td>on node) {</td></exception<>	on node) {
	cref='ParseException'>	return node;
*	•	}
* <param< td=""><td>* discovered</td><td>•</td></param<>	* discovered	•
_	errors	/**
	*/	* <summary>Called</summary>
*	public virtual	when adding a child to
* <exception< td=""><td>=</td><td>a parse tree</td></exception<>	=	a parse tree
= = = = = = = = = = = = = = = = = = = =		* node.
		*
-	n nege, nege entre, (	* <param< td=""></param<>
	node.AddChild(child):	name='node'>the parent
_	}	node
,	,	* <param< td=""></param<>
_	/**	name='child'>the child
	,	node, or null
		*
}		* <exception< td=""></exception<>
j	•	cref='ParseException'>
/**		if the node analysis
* <summary>Called</summary>	name='node'>the node	* discovered
when exiting a parco	heing entared(/naram)	errore(/avcantion)
when exiting a parse tree node.	being entered *	errors */
	/**  * <summary>Called when adding a child to a parse tree  * node. </summary> * <param name="node"/> the parent node  * <param name="child"/> the child node, or null  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */  public virtual void ChildProdRelopExt(Prod uction node, Node child) {  node. AddChild(child); }  /**  * <summary>Called when entering a parse tree node. </summary> * <param name="node"/> the node being entered  * <exception cref="ParseException"> if the node analysis  * discovered errors</exception> */  public virtual void EnterProdOpl(Productio n node) { }  /**	parse tree, or

public virtual	* <summary>Called</summary>	should be	EnterProdEnd(Productio
void	when adding a child to	created	n node) {
ChildProdLogOp(Product	a parse tree	*	}
ion node, Node child)	* node.	* <exception< td=""><td></td></exception<>	
{	*	cref='ParseException'>	/**
	* <param< td=""><td>if the node analysis</td><td>* <summary>Called</summary></td></param<>	if the node analysis	* <summary>Called</summary>
<pre>node. AddChild(child);</pre>	name='node'>the parent	* discovered	when exiting a parse
}	node	errors	tree node.
	* <param< td=""><td>*/</td><td>*</td></param<>	*/	*
/**	name='child'>the child	public virtual	* <param< td=""></param<>
* <summary>Called</summary>	node, or null	Node	name='node'>the node
when entering a parse	*	ExitProdLogOper(Produc	being exited
tree node.	* <exception< td=""><td>tion node) {</td><td>*</td></exception<>	tion node) {	*
*	<pre>cref='ParseException'&gt;</pre>	return node;	* <returns>the</returns>
* <param< td=""><td>if the node analysis</td><td>}</td><td>node to add to the</td></param<>	if the node analysis	}	node to add to the
name='node'>the node	* discovered		parse tree, or
being entered	errors	/**	* null
*	*/	* <summary>Called</summary>	if no parse tree
* <exception< td=""><td>public virtual</td><td>when adding a child to</td><td>should be</td></exception<>	public virtual	when adding a child to	should be
cref='ParseException'>	void	a parse tree	created
if the node analysis	ChildProdExtLogOp(Prod	* node.	*
* discovered	uction node, Node	*	* <exception< td=""></exception<>
errors	child) {	* <param< td=""><td>cref='ParseException'&gt;</td></param<>	cref='ParseException'>
*/		name='node'>the parent	if the node analysis
public virtual	node. AddChild(child);	node	* discovered
void	}	* <param< td=""><td>errors</td></param<>	errors
EnterProdExtLogOp (Prod	,	name='child'>the child	*/
uction node) {	/**	node, or null	public virtual
}	* <summary>Called</summary>	*	Node
	when entering a parse	* <exception< td=""><td>ExitProdEnd(Production</td></exception<>	ExitProdEnd(Production
/**	tree node.	cref='ParseException'>	node) {
* <summary>Called</summary>	*	if the node analysis	return node;
when exiting a parse	* <pre>* <pre>* <pre>* </pre></pre></pre>	* discovered	}
tree node.	name='node'>the node	errors	/state
*	being entered	*/	/**
* <param< td=""><td>*</td><td>public virtual</td><td>* <summary>Called</summary></td></param<>	*	public virtual	* <summary>Called</summary>
name='node'>the node being exited	<pre>* <exception cref="ParseException"></exception></pre>	void ChildProdLogOper(Produ	when adding a child to
*		ction node, Node	a parse tree * node.
* <returns>the</returns>	if the node analysis * discovered	child) {	* node. \/ summary/
node to add to the	errors	CIIIIu) (	* <param< td=""></param<>
parse tree, or	*/	node.AddChild(child);	name='node'>the parent
* null	public virtual	node. Addentita (chitta),	node
if no parse tree	void	J	* <param< td=""></param<>
should be	EnterProdLogOper(Produ	/**	name='child'>the child
created	ction node) {	* <summary>Called</summary>	node, or null
*	}	when entering a parse	*
* <exception< td=""><td>,</td><td>tree node. </td><td>* <exception< td=""></exception<></td></exception<>	,	tree node.	* <exception< td=""></exception<>
cref='ParseException'>	/**	*	<pre>cref='ParseException'&gt;</pre>
if the node analysis	* <summary>Called</summary>	* <param< td=""><td>if the node analysis</td></param<>	if the node analysis
* discovered	when exiting a parse	name='node'>the node	* discovered
errors	tree node.	being entered	errors
*/	*	*	*/
public virtual	* <param< td=""><td>* <exception< td=""><td></td></exception<></td></param<>	* <exception< td=""><td></td></exception<>	
Node	name='node'>the node	<pre>cref='ParseException'&gt;</pre>	public virtual void
ExitProdExtLogOp(Produ	being exited	if the node analysis	ChildProdEnd(Productio
ction node) {	*	* discovered	n node, Node child) {
return node;	* <returns>the</returns>	errors	
}	node to add to the	*/	<pre>node. AddChild(child);</pre>
	parse tree, or	public virtual	}
/**	* null	void	}
	if no parse tree		

Syntax Analyzer:	INT = 1058,	PROD_ELEMENT =	PROD_FUNCT_PARAM =
SyntaxConstants.cs	CHAR = 1059,	2020,	2050,
	FLOAT = 1060,	PROD_ADD_ELEM =	PROD_FUNCT_IDPARAM
<pre>public enum SyntaxConstants {</pre>	STRING = 1061, BOOL N = 1062,	2021, PROD M ELEM =	= 2051,
MAIN N = 1001,	ID = 1063,	2022,	PROD ADDFUNCT IDPARAM
PRINT N = 1001,	NUM = 1064,	PROD M2 ELEM =	= 2052,
SCAN N = 1003,	DECIMAL = 1065,	2023,	PROD BODY = 2053,
CONST N = $1004$ ,	S CHAR = 1066,	PROD FUNCTRET =	$PROD_PRINT = 2054,$
RETURN = 1005,	-TEXT = 1067,	2024,	PROD POSTVAL =
$SWITCH_N = 1006$ ,	COM = 1068,	PROD_DTYPE_A =	2055,
$CASE_N = 1007$ ,	YES = 1069,	2025,	$PROD_OUT = 2056,$
BREAK = 1008,	NO = 1070,	PROD_EXDTYPE_A =	$PROD\_OUT\_C = 2057,$
$FOR_N = 1009,$	FUNCTNAME = 1071,	2026,	PROD_STRUCT_C =
IF = 1010,	STRUCTNAME = 1072,	PROD_RETURN =	2058,
ELSEIF_N = 1011,	IDSTRUCT = 1073,	2027,	PROD_CONCAT_LIT =
ELSE_N = 1012,	F = 1074,	PROD_FUNCTVOID = 2028,	2059,
DO = 1013, WHILE N = 1014,	D = 1075, $S = 1076,$	PROD STRUCT =	$PROD\_SCAN = 2060,$ $PROD\_EXT\_I = 2061,$
VOID = 1015,	ZERO = 1077,	2029,	PROD FOR STATE =
GETCH = 1016,	SPACE = $1078$ ,	PROD MEM DEC =	2062,
STRUCT $N = 1017$ ,	N LINE = 1079,	2030,	PROD FORSTATEMENT
DEFAULT = 1018,	WHITESPACE = 1080,	PROD_INIT_DEC =	= 2063,
CLEAR = 1019,	TOCHAR = 1081,	2031,	$PROD_VAL1 = 2064,$
SQROOT = 1020,	LENGTHF = $1082$ ,		PROD_MNT_COND =
PLUS = 1021,	CONTAINS = 1083,	PROD_INIT_DEC_CHOICE =	2065,
MINUS = 1022,	REVERSE = 1084,	2032,	PROD_MNT_COND_T =
TIMES = 1023,	PROD_START_PROGRAM	PROD_CONSTANT =	2066,
DIVIDE = $1024$ ,	= 2001,	2033,	PROD_MNT = 2067,
MODULUS = 1025, EQUALS = 1026,	PROD_PROGRAM = 2002,	PROD_LOCAL_CHOICE = 2034,	PROD_IFELSE = 2068,
SEMIC = 1027,	PROD CLEAR = 2003,	PROD DECLARE1 =	PROD IFCONDITION =
DOT = 1028,	PROD COMMENTS =	2035,	2069,
COMMA = 1029,	2004,	PROD FUNCTRET1 =	PROD IFSTATEMENT =
AND = 1030,	PROD_NEGATE =	2036,	2070,
OR = 1031,	2005,	PROD_FUNCTVOID1 =	PROD_ELSEIF =
NOT = 1032,	PROD_DATATYPE =	2037,	2071,
INCREMENT = 1033,	2006,	PROD_STRUCT1 =	
DECREMENT = 1034,	PROD_LITERALS =	2038,	PROD_ELSEIFSTATEMENT =
P_E = 1035, M E = 1036,	2007, PROD LITERALS2 =	PROD_CONSTANT1 = 2039,	2072, PROD ELSE STATE =
$M_E = 1030,$ $T E = 1037,$	2008,	PROD MAIN = 2040,	2073,
D E = 1038,	PROD_GLOBAL_DEC =	PROD_ASSIGN_CHOICE	PROD ELSESTATEMENT
MOD E = 1039,	2009,	= 2041,	= 2074,
$\overline{NEWLINE} = 1040,$	PROD_DECLARE =		PROD_DOWHILE =
$N_{E} = 1041,$	2010,	PROD_ACCESS_ASSIGN_DTY	2075,
$O_PAREN = 1042,$		PE = 2042,	PROD_DOSTATEMENT =
$C_{PAREN} = 1043,$	PROD_DECLARE_CHOICE =		2076,
D_QUOTE = 1044,	2011,	PROD_ASSIGN_VALUE_CHOI	PROD_WHILE_STATE =
COLON = 1045, $O PRACKET = 1046$	PROD_INIT_CHOICE =	CE = 2043,	2077,
O_BRACKET = 1046, C BRACKET = 1047,	2012, PROD ADD ID =	PROD_ASSIGNING = 2044,	PROD WHILESTATEMENT =
$C_{\text{DRACKET}} = 1047,$ $GREATER = 1048,$	2013,	PROD ARRAY ID =	2078,
LESS = 1049,	PROD N1 = 2014,	2045,	PROD SWITCH STATE
GREATER_E = $1050$ ,	$PROD_N2 = 2015,$	PROD_ARRAY_IDTAIL	= 2079,
LESS_E = 1051,	$PROD_INDEX = 2016,$	= 2046,	PROD_CASE_STATE =
$S_OBRACKET = 1052,$	$PROD\_SMATH = 2017,$	PROD_ASSIGN_SYM =	2080,
$S_{CBRACKET} = 1053,$	PROD_ARRAY_AID =	2047,	$PROD\_DEF = 2081,$
DOLLAR = 1054,	2018,	PROD_ASSIGN_VALUE	PROD_CASESTATEMENT
POWER = 1055,	PROD_ELEM_CHOICE =	= 2048,	= 2082,
HASH = 1056, NEGA = 1057,	2019,	PROD_CONVERT = 2049,	PROD_MATH_OP = 2083,
NEGA - 1007,		۵0±3,	۵۵۵۰,

PROD OPER COND =	List <node></node>		int line =
2084,	prevparent = new	productions.Add(node);	1;
2001,	List <node>();</node>	productions. Add (node),	int
PROD OPER COND CHOICE	List(Node)	<pre>production += "Enter:</pre>	linejump = 0;
= 2085,	productions = new	" + name + " Parent: "	foreach
PROD OPER SYM =	List <node>();</node>	+ currparent. GetName()	
			(var t in tokens)
2086,	public	+ "\n";	1
PROD_OPER_EQ =	List <string> SET = new</string>	}	if
2087,	List <string>();</string>	1	(t.getLines() != line)
PROD_OPER_EXT_S =	public	}	1
2088,	List <string></string>	public	
PROD_OPER_EXT_REP	PRODUCTION = new	override Node	linejump =
= 2089,	List <string>();</string>	Exit(Node node)	t.getLines() - line;
PROD_OPERAND =		{	
2090,	public	if	line = t.getLines();
$PROD\_SIM\_MATH\_OP =$	override void	(currparent == node)	
2091,	Enter(Node node)	{	for (int $i = 0$ ; $i <$
$PROD\_S\_MATH\_EXT =$	{		linejump; i++)
2092,	string	currparent =	{
PROD_OPER_COND_EXT	name = node.GetName();	prevparent[prevparent.	
= 2093,	if	Count - 1];	tokenstream += "\n";
PROD_REL_OP =	(name.Contains("Prod_"		}
2094,	))	prevparent.RemoveAt(pr	}
PROD_RELOP_EXT =	{	evparent.Count - 1);	
2095,		}	tokenstream +=
PROD OP1 = 2096,	node. SetParent (currpar	return	t.getTokens() + " ";
PROD LOG OP =	ent);	node;	}
2097,	name =	}	,
PROD EXT LOG OP =	name. Substring(5);	•	tokenstream =
2098,		public	tokenstream.TrimEnd();
PROD LOG OPER =	if	override Node	tenensulation in the contract of the contract
2099,	(currparent != null)	Analyze (Node node)	Parser p;
PROD END = 2100	(callparent : mall)	{	p =
		C C	P
FROD_END - 2100 }	(	return	CreateParser(tokenstre
FROD_END	production += "Enter:	return hase Analyze(node):	CreateParser(tokenstre
}	production += "Enter:	return base.Analyze(node); }	<pre>CreateParser(tokenstre am);</pre>
<pre>} Syntax Analyzer:</pre>	<" + name + "> Parent:		am);
}	<" + name + "> Parent: " +	<pre>base. Analyze (node); }</pre>	
Syntax Analyzer: SyntaxInitializer.cs	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() +</pre>	<pre>base.Analyze(node); } public</pre>	am); try {
Syntax Analyzer: SyntaxInitializer.cs using System;	<" + name + "> Parent: " +	<pre>base.Analyze(node); }  public override Node</pre>	am); try { Node
Syntax Analyzer: SyntaxInitializer.cs using System; using System. I0;	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";</pre>	<pre>base. Analyze (node); }  public override Node Analyze (Node node,</pre>	am); try {
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() +</pre>	base.Analyze(node);  }  public override Node Analyze(Node node, ParserLogException	am);  try {  Node parse = p. Parse();
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node); }</pre>	base. Analyze (node);    public   override Node   Analyze (Node node,   ParserLogException	am);  try {
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric;	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log) {</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node); }</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric; using Core. Library;	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node); } else {</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node,</pre>	am);  try { Node parse = p.Parse();  Fail("parsing succeeded");  result = "Syntax Analyzer
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric;	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node);</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric; using Core. Library; using TokenLibrary;	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node); } else {</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node,</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded"; }</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric; using Core. Library; using TokenLibrary; namespace	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node);</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node, log);     } }</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded"; } catch</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric; using Core. Library; using TokenLibrary;	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node);</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node, log);     }     public</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationExcepti</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric; using Core. Library; using TokenLibrary; namespace Syntax_Analyzer {	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n"; productions.Add(node);</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node, log);     }     public ErrorClass errors =</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded"; } catch</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System.IO; using System.Collections.Gen eric; using Core.Library; using TokenLibrary; namespace Syntax_Analyzer {    public class	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node, log);     }      public ErrorClass errors = new ErrorClass();</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationExcepti</pre>
Syntax Analyzer: SyntaxInitializer.cs using System; using System. IO; using System. Collections. Gen eric; using Core. Library; using TokenLibrary; namespace Syntax_Analyzer {    public class SyntaxInitializer:	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node, log);     }      public ErrorClass errors = new ErrorClass();     public string</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationException e)</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System.IO; using System.Collections.Gen eric; using Core.Library;  using TokenLibrary;  namespace Syntax_Analyzer {    public class	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationException e) { Fail(e.Message);</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System.IO; using System.Collections.Gen eric; using Core.Library;  using TokenLibrary;  namespace Syntax_Analyzer {    public class SyntaxInitializer: SyntaxAnalyzer {	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);      public override Node Analyze (Node node, ParserLogException log)     {         return base. Analyze (node, log);     }      public ErrorClass errors = new ErrorClass();     public string</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationException e)  { Fail(e.Message);     result</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System. IO; using System. Collections. Gen eric; using Core. Library;  using TokenLibrary;  namespace Syntax_Analyzer {    public class SyntaxInitializer: SyntaxAnalyzer {    public string	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationException e) { Fail(e.Message);</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System. IO; using System. Collections. Gen eric; using Core. Library;  using TokenLibrary;  namespace Syntax_Analyzer {    public class SyntaxInitializer: SyntaxAnalyzer    qublic string production = "";	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationExcepti on e)  { Fail(e.Message);     result = e.Message; }</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System. IO; using System. Collections. Gen eric; using Core. Library;  using TokenLibrary;  namespace Syntax_Analyzer {    public class SyntaxInitializer: SyntaxAnalyzer    qublic string production = "";    public string	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationExcepti on e)  { Fail(e.Message);     result = e.Message;     }     catch</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System. IO; using System. Collections. Gen eric; using Core. Library;  using TokenLibrary;  namespace Syntax_Analyzer {     public class SyntaxInitializer: SyntaxAnalyzer     {         public string production = "";         public string recursiveprod = "";	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationExcepti on e)  { Fail(e.Message);     result = e.Message; }</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System.IO; using System.Collections.Gen eric; using Core.Library;  using TokenLibrary;  namespace Syntax_Analyzer {     public class SyntaxInitializer: SyntaxAnalyzer     {         public string production = "";         public string recursiveprod = "";         Node	<pre><!-- + name + "--> Parent:     " +     currparent.GetName() +     "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded"; }     catch (ParserCreationExcepti on e)  { Fail(e.Message);     result = e.Message; } catch</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System. IO; using System. Collections. Gen eric; using Core. Library;  using TokenLibrary;  namespace Syntax_Analyzer {     public class SyntaxInitializer: SyntaxAnalyzer     {         public string production = "";         public string recursiveprod = "";	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>public     override Node Analyze(Node node, ParserLogException log)     {         return base. Analyze(node, log);     }     public ErrorClass errors =     new ErrorClass();         public string Start(List<tokensclass> tokens)     {             //Boolean isDone = false;             string tokenstream = "";             string</tokensclass></pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded"; }     catch (ParserCreationExcepti on e)  { Fail(e.Message);     result = e.Message; } catch</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System.IO; using System.Collections.Gen eric; using Core.Library;  using TokenLibrary;  namespace Syntax_Analyzer {     public class SyntaxInitializer: SyntaxAnalyzer     {         public string production = "";         public string recursiveprod = "";         Node	<pre><!-- + name + "--> Parent:     " +     currparent.GetName() +     "\n";  productions.Add(node);</pre>	<pre>base. Analyze (node);</pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded";     }     catch (ParserCreationException e)  { Fail(e.Message);     result = e.Message;     }     catch (ParserLogException e)</pre>
Syntax Analyzer: SyntaxInitializer.cs  using System; using System.IO; using System.Collections.Gen eric; using Core.Library;  using TokenLibrary;  namespace Syntax_Analyzer {     public class SyntaxInitializer: SyntaxAnalyzer     {         public string production = "";         public string recursiveprod = "";         Node	<pre>&lt;" + name + "&gt; Parent: " + currparent.GetName() + "\n";  productions.Add(node);</pre>	<pre>public     override Node Analyze(Node node, ParserLogException log)     {         return base. Analyze(node, log);     }     public ErrorClass errors =     new ErrorClass();         public string Start(List<tokensclass> tokens)     {             //Boolean isDone = false;             string tokenstream = "";             string</tokensclass></pre>	<pre>am);  try {     Node parse = p.Parse();  Fail("parsing succeeded");     result = "Syntax Analyzer Succeeded"; }     catch (ParserCreationExcepti on e)  { Fail(e.Message);     result = e.Message; } catch</pre>

<pre>p. GetAllProductionCode ();</pre>	// message += item + ",	//}	<pre>node.GetName().ToLower ();</pre>
	<b>"</b> ;		
<pre>PredictSets ps = new PredictSets();</pre>	//	message += ".";	<pre>currentparent = node.GetParent().GetNa</pre>
string	//}		me():
message = "Expected:	if	amana satEmanMassass	me(),
message - Expected.		errors.setErrorMessage	
,	<pre>(message == "Expected:</pre>	(message);	currentparent =
.0.1 ( 0 .	")	, T. ( C , F	currentparent.ToLower(
errors.setColumn(e.Get	{	errors.setType(e.GetEr	);
Error(0).Column);		ror(0). Type. ToString()	
	string errormessage =	);	delete = true;
errors.setLines(e.GetE	e.GetError(0).ErrorMes	result	
rror(0).Line);	sage;	= e.Message;	if
int	if		(currentparent.Contain
ctr =	(errormessage.Contains	}	s("prod_"))
<pre>GetSyntaxTable(codes);</pre>	("unexpected token"))		{
	{	recursiveprod =	
//isDone = true;		p.GetRecursiveProducti	currentparent = "<" +
	errormessage = "";	on();	currentparent.Substrin
if	,	,	g(5) + ">";
(codes. Count - 1) =	foreach (var item in	GetSyntaxTable(p.GetAl	}
ctr)	e. GetError(0). Details)	1ProductionCode());	if
( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( ( (	e. detEilor (0). Details)	return	(nodename.Contains("pr
l	ſ		
:	1	result;	od_"))
<pre>int code = codes[ctr];</pre>		}	1
	errormessage += item +		1 " ' ' '
message +=	", ";	private int	nodename = "<" +
ps.GetPredictSet(code)		GetSyntaxTable(List <in< td=""><td>nodename.Substring(5)</td></in<>	nodename.Substring(5)
;	}	t> code)	+ ">";
}	}	{	}
else	if	Node node	
{	(errormessage ==	= nu11;	
{	<pre>(errormessage == "unexpected end of</pre>	= null; Boolean	PRODUCTION.Add(current
<pre>int code = codes[ctr-</pre>	_	•	PRODUCTION.Add(current parent);
<pre>int code = codes[ctr- 1];</pre>	"unexpected end of	Boolean	· · · · · · · · · · · · · · · · · · ·
	"unexpected end of	Boolean delete = true;	· · · · · · · · · · · · · · · · · · ·
	"unexpected end of file")	Boolean delete = true; string	parent);
1];	"unexpected end of file") errormessage =	Boolean delete = true; string recprod =	parent);
1]; message +=	"unexpected end of file") errormessage =	Boolean  delete = true;     string  recprod =  recursiveprod;     int ctr =	<pre>parent);  SET. Add (nodename); }</pre>
1]; message +=	<pre>"unexpected end of file") errormessage = "\".\"";</pre>	Boolean  delete = true;     string  recprod =  recursiveprod;     int ctr = -1, count = 1,	<pre>parent);  SET. Add (nodename); }</pre>
<pre>1]; message += ps.GetPredictSet(code) ; }</pre>	<pre>"unexpected end of file") errormessage = "\".\""; message +=</pre>	Boolean delete = true; string recprod = recursiveprod; int ctr = -1, count = 1, prodcode = 0;	<pre>parent);  SET. Add (nodename); }</pre>
<pre>1]; message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file") errormessage = "\".\"";</pre>	Boolean delete = true; string recprod = recursiveprod; int ctr = -1, count = 1, prodcode = 0; string	<pre>parent);  SET. Add (nodename);</pre>
<pre>1]; message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file") errormessage = "\".\""; message +=</pre>	Boolean delete = true; string recprod = recursiveprod; int ctr = -1, count = 1, prodcode = 0; string currentparent = "";	<pre>parent);  SET. Add (nodename); } else { string name =</pre>
<pre>1]; message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =   "\".\"";  message += errormessage; }</pre>	Boolean delete = true; string recprod = recursiveprod; int ctr = -1, count = 1, prodcode = 0; string currentparent = ""; while	<pre>parent);  SET. Add (nodename);</pre>
<pre>1]; message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =   "\".\"";  message += errormessage; }  //if</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>1]; message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =   "\".\"";  message += errormessage; }  //if (message == "Expected:</pre>	Boolean delete = true; string recprod = recursiveprod; int ctr = -1, count = 1, prodcode = 0; string currentparent = ""; while	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =   "\".\"";  message +=   errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =    "\".\"";  message +=    errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =    "\".\"";  message +=    errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps. GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =    "\".\"";  message += errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =    "\".\"";  message += errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps. GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =    "\".\"";  message +=    errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =    "\".\"";  message += errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add(nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =     "\".\"";  message +=     errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =     "\".\"";  message +=     errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>nessage += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =     "\".\"";  message +=     errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =     "\".\"";  message +=     errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
message += ps.GetPredictSet(code) ;	<pre>"unexpected end of file")  errormessage =     "\".\"";  message +=     errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
<pre>message += ps.GetPredictSet(code) ;</pre>	<pre>"unexpected end of file")  errormessage =     "\".\"";  message +=     errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
message += ps.GetPredictSet(code) ;	<pre>"unexpected end of file")  errormessage =    "\".\"";  message +=    errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
message += ps.GetPredictSet(code) ;	<pre>"unexpected end of file")  errormessage =    "\".\"";  message +=    errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);</pre>
message += ps.GetPredictSet(code) ;	<pre>"unexpected end of file")  errormessage =    "\".\"";  message += errormessage;</pre>	Boolean  delete = true;	<pre>parent);  SET. Add (nodename);  else {  string name = Enum. GetName (typeof (Sy ntaxConstants), prodcode);  name = name. ToLower();     if     (name. Contains ("prod_" ))  {  name = "&lt;" + name. Substring(5) + "&gt;"; }  if     (PRODUCTION. Count !=</pre>

	}		
currentparent.ToLower(	j	private enum	
);		SynteticPatterns {	<pre>CreatePatterns();</pre>
	}	}	}
if	return		
(currentparent.Contain	(ctr + 1);	/**	/**
s("prod_"))	}	*	*
		<pre><summary>Creates a new</summary></pre>	<pre><summary>Creates a new</summary></pre>
{	private Parser	parser with a default	tokenizer for this
	CreateParser(string	analyzer.	parser. Can be
currentparent = "<" +	input)	*	overridden
currentparent.Substrin	{	* <param< td=""><td>* by a subclass</td></param<>	* by a subclass
g(5) + ">";	Parser	name='input'>the input	to provide a custom
1	parser = null;	stream to read	implementation.
}	try	from *	ry>
PRODUCTION. Add (current	nargor	* <exception< td=""><td>* <param< td=""></param<></td></exception<>	* <param< td=""></param<>
parent);	parser = new SyntaxParser(new	cref='ParserCreationEx	name='input'>the input
parent),	StringReader(input),	ception'>if the parser	stream to read
SET. Add(name);	this);	* couldn't be	from
bbi. Itaa (Itame) ,	onis,	initialized	*
PRODUCTION. Add (name):	parser. Prepare();	correctly	* <returns>the</returns>
,	F,	*/	tokenizer
SET. Add (" λ");	}	public	created
	catch	SyntaxParser(TextReade	*
<pre>delete = false;</pre>	(ParserCreationExcepti	r input)	* <exception< td=""></exception<>
}	on e)	: base(input)	cref='ParserCreationEx
	{	{	ception'>if the
else			tokenizer
{	Fail(e.Message);		* couldn't be
PROPUGITAN A 11/1/	}	CreatePatterns();	initialized
PRODUCTION. Add (" <progr< td=""><td>return</td><td>}</td><td>correctly</td></progr<>	return	}	correctly
am>");	parser;	/state	*/
SET. Add(name);	}	/**	protected override Tokenizer
SEI. Add (Halle),	protected void	<pre><summary>Creates a new</summary></pre>	NewTokenizer(TextReade
PRODUCTION. Add(name);	Fail (string message)	parser.	r input) {
The section has traine,	{	*	return new
SET. Add (" λ");	if	* <param< td=""><td>SyntaxTokenizer(input)</td></param<>	SyntaxTokenizer(input)
	(message != "parsing	name='input'>the input	;
delete = false;	succeeded")	stream to read	}
}	throw	from	
}	new	*	/**
	<pre>Exception(message);</pre>	* <param< td=""><td>*</td></param<>	*
if	}	name='analyzer'>the	<summary>Initializes</summary>
(count != 1 && delete)		analyzer to parse	the parser by creating
{		with	all the production
D 1 /0	,	*	*
productions.RemoveAt(0	}	* <exception< td=""><td>patterns.</td></exception<>	patterns.
);	}	<pre>cref='ParserCreationEx ception'&gt;if the parser</pre>	* * <exception< td=""></exception<>
else	Syntax Analyzan	* couldn't be	cref='ParserCreationEx
if (delete)	Syntax Analyzer: SyntaxParser.cs	initialized	ception'>if the parser
{	Sylivani ai Boi. OS	correctly	* couldn't be
ι	using System.IO;	*/	initialized
productions.RemoveAt(0		public	correctly
);	using Core.Library;	SyntaxParser(TextReade	*/
	·	r input,	private void
productions.RemoveAt(0	public class	SyntaxAnalyzer	CreatePatterns() {
);	SyntaxParser :	analyzer)	-
	RecursiveDescentParser	: base(input,	ProductionPattern
count = 0;	{	analyzer) {	pattern;

	,	SuntayConstants DDOD N	
ProductionPatternAlter	SyntaxConstants.PROD_M	SyntaxConstants.PROD_N EGATE,	pattern.AddAlternative
native alt;	AIN, 1, 1);	Lonie,	(alt);
native art,	1111, 1, 1,	"Prod Negate");	alt = new
pattern = new	alt.AddProduction((int	alt = new	ProductionPatternAlter
ProductionPattern((int	)	ProductionPatternAlter	<pre>native();</pre>
)	SyntaxConstants.PROD_B	native();	
SyntaxConstants.PROD_S	ODY, 0, 1);		alt.AddToken((int)
TART_PROGRAM,		alt.AddToken((int)	SyntaxConstants.BOOL_N
	pattern.AddAlternative	SyntaxConstants.NEGA,	, 1, 1);
"Prod_StartProgram");	(alt);	1, 1);	
alt = new			pattern.AddAlternative
ProductionPatternAlter	AddPattern(pattern);	pattern. AddAlternative	(alt);
native();		(alt);	
	pattern = new		AddPattern(pattern);
alt.AddProduction((int	ProductionPattern((int	AddPattern(pattern);	
)	)		pattern = new
SyntaxConstants. PROD_C	SyntaxConstants.PROD_C	pattern = new	ProductionPattern((int
OMMENTS, 0, 1);	LEAR,	ProductionPattern((int	)
-14 AddD 1	"D	)	SyntaxConstants.PROD_L
alt.AddProduction((int	"Prod_Clear");	SyntaxConstants.PROD_D	ITERALS,
) StCtt DDOD D	alt = new	ATATYPE,	"Dr. d I :+1-") .
SyntaxConstants.PROD_P ROGRAM, 1, 1);	<pre>ProductionPatternAlter native();</pre>	"Prod_datatype");	"Prod_Literals"); alt = new
NOGRAM, 1, 1),	native(),	alt = new	ProductionPatternAlter
alt.AddProduction((int	alt.AddToken((int)	ProductionPatternAlter	native();
)	SyntaxConstants. CLEAR,	native();	native(),
SyntaxConstants. PROD C	1, 1);	native (),	alt.AddProduction((int
OMMENTS, 0, 1);	1, 1/,	alt.AddToken((int)	)
Cimilarities, C, 17,	alt.AddToken((int)	SyntaxConstants. INT,	SyntaxConstants.PROD N
alt.AddProduction((int	SyntaxConstants. SEMIC,	1, 1);	EGATE, 0, 1);
)	1, 1);	1, 1,	Beili2, 0, 1,,
SyntaxConstants.PROD E	-, -, ,	pattern.AddAlternative	alt.AddToken((int)
ND, 1, 1);	pattern.AddAlternative	(alt);	SyntaxConstants.NUM,
, , , ,	(alt);	alt = new	1, 1);
alt.AddProduction((int		ProductionPatternAlter	
)	AddPattern(pattern);	native();	pattern.AddAlternative
SyntaxConstants.PROD_C			(alt);
OMMENTS, 0, 1);	pattern = new	alt.AddToken((int)	alt = new
	ProductionPattern((int	SyntaxConstants.FLOAT,	ProductionPatternAlter
pattern.AddAlternative	)	1, 1);	native();
(alt);	SyntaxConstants.PROD_C		
	OMMENTS,	pattern.AddAlternative	alt.AddToken((int)
AddPattern(pattern);		(alt);	SyntaxConstants.DECIMA
	"Prod_comments");	alt = new	L, 1, 1);
pattern = new	alt = new	ProductionPatternAlter	
ProductionPattern((int	ProductionPatternAlter	native();	pattern.AddAlternative
)	native();		(alt);
SyntaxConstants.PROD_P	1. 4110 1 (// .)	alt.AddToken((int)	alt = new
ROGRAM,	alt.AddToken((int)	SyntaxConstants. STRING	ProductionPatternAlter
"D 1 ")	SyntaxConstants.COM,	, 1, 1);	native();
"Prod_program");	1, 1);		-1+ AllT-1((:-+)
alt = new		pattern.AddAlternative	alt.AddToken((int)
ProductionPatternAlter native();	pattern.AddAlternative (alt);	(alt); alt = new	SyntaxConstants.TEXT, 1, 1);
native(),	(a1t),	art = new ProductionPatternAlter	1, 1/,
alt.AddProduction((int	AddPattern(pattern);	native();	pattern.AddAlternative
)	nuur arrern (parrern),	Hative (/,	(alt);
SyntaxConstants.PROD_G	pattern = new	alt.AddToken((int)	alt = new
LOBAL DEC, 0, 1);	ProductionPattern((int	SyntaxConstants. CHAR,	ProductionPatternAlter
1001III_DD0, 0, 1/,	)	1, 1);	native();
alt.AddProduction((int	,	±, ±/,	11401.0(/,
31 5. Man 1 0 4 4 0 1 1 0 11 ( 1 11 t			

	-14 =		
alt.AddToken((int)	alt = new ProductionPatternAlter	alt.AddToken((int)	alt.AddProduction((int
SyntaxConstants. S CHAR	native();	SyntaxConstants. ID, 1,	)
, 1, 1);		1);	SyntaxConstants.PROD_C
	alt.AddToken((int)		OMMENTS, 0, 1);
pattern.AddAlternative	SyntaxConstants.TEXT,	alt.AddProduction((int	
(alt);	1, 1);	)	pattern.AddAlternative
alt = new	A 1141	SyntaxConstants. PROD_F	(alt);
<pre>ProductionPatternAlter native();</pre>	<pre>pattern.AddAlternative (alt);</pre>	UNCTVOID, 1, 1);	AddPattern(pattern);
native(),	alt = new	alt.AddProduction((int	Addi attern (pattern),
alt.AddToken((int)	ProductionPatternAlter	)	pattern = new
SyntaxConstants. YES,	native();	SyntaxConstants.PROD C	ProductionPattern((int
1, 1);		OMMENTS, 0, 1);	)
	alt.AddToken((int)		SyntaxConstants.PROD_D
pattern. AddAlternative	SyntaxConstants. S_CHAR	pattern.AddAlternative	ECLARE,
(alt);	, 1, 1);	(alt);	"D 1 D 1 ")
alt = new ProductionPatternAlter		alt = new ProductionPatternAlter	"Prod_Declare"); alt = new
native();	<pre>pattern.AddAlternative (alt);</pre>	native();	ProductionPatternAlter
native(),	(a1t),	native(),	native();
alt.AddToken((int)	AddPattern(pattern);	alt.AddToken((int)	,
SyntaxConstants.NO, 1,	4	SyntaxConstants.STRUCT	alt.AddProduction((int
1);	pattern = new	_N, 1, 1);	)
	ProductionPattern((int		SyntaxConstants.PROD_D
pattern. AddAlternative	)	alt.AddToken((int)	ECLARE_CHOICE, 0, 1);
(alt);	SyntaxConstants.PROD_G	SyntaxConstants. ID, 1,	-1+ AllT-1((:-+)
AddPattern(pattern);	LOBAL_DEC,	1);	alt.AddToken((int) SyntaxConstants.SEMIC,
Addi attern (pattern),	"Prod globalDec");	alt.AddProduction((int	1, 1);
pattern = new	alt = new	)	1, 1,
ProductionPattern((int	ProductionPatternAlter	SyntaxConstants.PROD_S	alt.AddProduction((int
)	native();	TRUCT, 1, 1);	)
SyntaxConstants.PROD_L			SyntaxConstants. PROD_G
ITERALS2,	alt.AddProduction((int	alt.AddProduction((int	LOBAL_DEC, 0, 1);
"Prod Literals2");	) SyntaxConstants.PROD D	) SyntaxConstants.PROD C	alt.AddProduction((int
alt = new	ATATYPE, 1, 1);	OMMENTS, 0, 1);	)
ProductionPatternAlter		3.mail. (15), (1),	SyntaxConstants.PROD C
native();	alt.AddToken((int)	pattern.AddAlternative	LEAR, 0, 1);
	SyntaxConstants.ID, 1,	(alt);	
alt.AddProduction((int	1);	alt = new	pattern.AddAlternative
)		ProductionPatternAlter	(alt);
SyntaxConstants. PROD_N	alt.AddProduction((int	native();	alt = new
EGATE, 0, 1);	) SyntaxConstants.PROD D	alt.AddToken((int)	<pre>ProductionPatternAlter native();</pre>
alt.AddToken((int)	ECLARE, 1, 1);	SyntaxConstants. CONST_	native(),
SyntaxConstants. NUM,	2021m2, 1, 1, 1,	N, 1, 1);	alt.AddProduction((int
1, 1);	alt.AddProduction((int	, , , ,	)
	)	alt.AddProduction((int	SyntaxConstants.PROD_F
pattern. AddAlternative	SyntaxConstants.PROD_C	)	UNCTRET, 1, 1);
(alt);	OMMENTS, 0, 1);	SyntaxConstants.PROD_D	1. 4110 1 //
alt = new ProductionPatternAlter		ATATYPE, 1, 1);	alt.AddProduction((int )
native();	pattern.AddAlternative (alt);	alt.AddToken((int)	SyntaxConstants.PROD G
nacive (/ ,	alt = new	SyntaxConstants. ID, 1,	LOBAL DEC, 0, 1);
alt.AddToken((int)	ProductionPatternAlter	1);	, ,
SyntaxConstants. DECIMA	native();		alt.AddProduction((int
L, 1, 1);		alt.AddProduction((int	)
	alt.AddToken((int)	)	SyntaxConstants.PROD_C
pattern. AddAlternative	SyntaxConstants. VOID,	SyntaxConstants. PROD_C	LEAR, 0, 1);
(alt);	1, 1);	ONSTANT, 1, 1);	

		)	pattern = new
<pre>pattern.AddAlternative (alt);</pre>	<pre>alt.AddProduction((int )</pre>	SyntaxConstants.PROD_N 1,	<pre>ProductionPattern((int )</pre>
(410),	SyntaxConstants.PROD I	1,	SyntaxConstants.PROD I
AddPattern(pattern);	NIT_CHOICE, 0, 1);	"Prod_N1"); alt = new	NDEX,
pattern = new	pattern.AddAlternative	ProductionPatternAlter	"Prod_index");
ProductionPattern((int	(alt);	<pre>native();</pre>	alt = new
)	alt = new		ProductionPatternAlter
SyntaxConstants.PROD_D	ProductionPatternAlter	alt.AddToken((int)	<pre>native();</pre>
ECLARE_CHOICE,	native();	SyntaxConstants.S_OBRACKET, 1, 1);	alt.AddToken((int)
"Prod DeclareChoice");	alt.AddToken((int)	3111, 1, 1,	SyntaxConstants. NUM,
alt = new	SyntaxConstants. EQUALS	alt.AddProduction((int	1, 1);
ProductionPatternAlter	, 1, 1);	)	
native();		SyntaxConstants.PROD_I	alt.AddProduction((int
	alt.AddProduction((int	NDEX, 1, 1);	)
alt.AddProduction((int	)	1. 11171 . ((:)	SyntaxConstants. PROD_S
SyntaxConstants.PROD I	SyntaxConstants.PROD_L ITERALS, 1, 1);	alt.AddToken((int) SyntaxConstants.S_CBRA	MATH, 0, 1);
NIT_CHOICE, 1, 1);	HERALS, 1, 1),	CKET, 1, 1);	pattern.AddAlternative
MII_CHOICE, I, I),	alt.AddProduction((int	ORL1, 1, 1/,	(alt);
pattern. AddAlternative	)	alt.AddProduction((int	alt = new
(alt);	SyntaxConstants.PROD_A	)	ProductionPatternAlter
alt = new	DD_ID, 0, 1);	SyntaxConstants.PROD_N	<pre>native();</pre>
ProductionPatternAlter		2, 0, 1);	
native();	pattern.AddAlternative		alt.AddToken((int)
1	(alt);	pattern.AddAlternative	SyntaxConstants. ID, 1,
alt.AddProduction((int	AddDattam (nattam)	(alt);	1);
SyntaxConstants. PROD N	AddPattern(pattern);	AddPattern(pattern);	alt.AddProduction((int
1, 1, 1);	pattern = new	Addi attern (pattern),	)
_, _, _,	ProductionPattern((int	pattern = new	SyntaxConstants.PROD S
alt.AddProduction((int	)	ProductionPattern((int	MATH, 0, 1);
)	SyntaxConstants.PROD_A	)	
SyntaxConstants. PROD_A	DD_ID,	SyntaxConstants.PROD_N	pattern. AddAlternative
RRAY_AID, 0, 1);	"D 1 1170"	2,	(alt);
A 1 1 A 1 1 A 1 A 1 A 1 A 1 A 1 A 1 A 1	"Prod_addID");	"D. 1 NO")	A 1 ID 44 ( - 44 )
<pre>pattern. AddAlternative (alt);</pre>	alt = new ProductionPatternAlter	"Prod_N2"); alt = new	AddPattern(pattern);
(a1t),	native();	ProductionPatternAlter	pattern = new
AddPattern(pattern);	native (),	native();	ProductionPattern((int
mad accorn (paccorn, ,	alt.AddToken((int)	11401.0 (/ ,	)
pattern = new	SyntaxConstants.COMMA,	alt.AddToken((int)	SyntaxConstants.PROD_S
ProductionPattern((int	1, 1);	SyntaxConstants.S_OBRA	MATH,
)		CKET, 1, 1);	
SyntaxConstants.PROD_I	alt.AddToken((int)		"Prod_Smath");
NIT_CHOICE,	SyntaxConstants. ID, 1,	alt.AddProduction((int	alt = new
"Prod InitChoice");	1);	) SyntaxConstants.PROD I	<pre>ProductionPatternAlter native();</pre>
alt = new	alt.AddProduction((int	NDEX, 1, 1):	native(),
ProductionPatternAlter	)	NDEA, 1, 1/,	alt.AddProduction((int
native();	SyntaxConstants.PROD I	alt.AddToken((int)	)
<del></del>	NIT_CHOICE, 0, 1);	SyntaxConstants. S_CBRA	SyntaxConstants.PROD_0
alt.AddToken((int)		CKET, 1, 1);	PER_SYM, 1, 1);
SyntaxConstants.COMMA,	pattern.AddAlternative		
1, 1);	(alt);	pattern.AddAlternative (alt);	alt.AddProduction((int
alt.AddToken((int)	AddPattern(pattern);	(αι υ,	SyntaxConstants.PROD I
SyntaxConstants. ID, 1,	man account (particin),	AddPattern(pattern);	NDEX, 1, 1);
1);	pattern = new	*	
	ProductionPattern((int		

ProductionPattern((int

pattern.AddAlternative (alt);	<pre>SyntaxConstants.0_BRAC KET, 1, 1);</pre>	alt.AddProduction((int	SyntaxConstants.PROD_M _ELEM, 1, 1);
AddPattern(pattern);	<pre>alt.AddProduction((int )</pre>	SyntaxConstants.PROD_E LEMENT, 1, 1);	<pre>pattern.AddAlternative (alt);</pre>
pattern = new ProductionPattern((int	SyntaxConstants.PROD_E LEMENT, 1, 1);	pattern.AddAlternative (alt);	AddPattern(pattern);
) SyntaxConstants.PROD_A RRAY_AID,	<pre>alt.AddToken((int) SyntaxConstants.C_BRAC KET, 1, 1);</pre>	AddPattern(pattern);	<pre>pattern = new ProductionPattern((int )</pre>
"Prod_arrayAID"); alt = new	alt.AddProduction((int)	pattern = new ProductionPattern((int )	SyntaxConstants.PROD_FUNCTRET,
<pre>ProductionPatternAlter native();</pre>	SyntaxConstants.PROD_M _ELEM, 1, 1);	SyntaxConstants.PROD_M _ELEM,	"Prod_functret"); alt = new ProductionPatternAlter
alt.AddToken((int) SyntaxConstants.EQUALS	<pre>pattern. AddAlternative (alt);</pre>	"Prod_M_Elem"); alt = new	native();
<pre>, 1, 1); alt.AddToken((int)</pre>	AddPattern(pattern);	<pre>ProductionPatternAlter native();</pre>	alt.AddToken((int) SyntaxConstants.O_PARE N, 1, 1);
SyntaxConstants.O_BRAC KET, 1, 1);	<pre>pattern = new ProductionPattern((int))</pre>	alt.AddToken((int) SyntaxConstants.COMMA,	alt.AddProduction((int
<pre>alt.AddProduction((int )</pre>	SyntaxConstants.PROD_E LEMENT,	1, 1); alt.AddToken((int)	SyntaxConstants.PROD_D TYPE_A, 0, 1);
SyntaxConstants.PROD_E LEM_CHOICE, 1, 1);	"Prod_Element"); alt = new	SyntaxConstants.O_BRAC KET, 1, 1);	alt.AddToken((int) SyntaxConstants.C_PARE
<pre>alt.AddToken((int) SyntaxConstants.C_BRAC KET, 1, 1);</pre>	<pre>ProductionPatternAlter native();</pre>	alt.AddProduction((int ) SyntaxConstants.PROD E	N, 1, 1); alt.AddToken((int)
pattern.AddAlternative	alt.AddProduction((int))	LEMENT, 1, 1);	SyntaxConstants. O_BRAC KET, 1, 1);
<pre>(alt); AddPattern(pattern);</pre>	SyntaxConstants.PROD_L ITERALS2, 1, 1);	alt.AddToken((int) SyntaxConstants.C_BRAC KET, 1, 1);	<pre>alt.AddProduction((int )</pre>
pattern = new ProductionPattern((int	alt.AddProduction((int ) SyntaxConstants.PROD A	alt.AddProduction((int	SyntaxConstants.PROD_BODY, 0, 1);
) SyntaxConstants. PROD_E LEM CHOICE,	DD_ELEM, 0, 1);	SyntaxConstants.PROD_M 2_ELEM, 0, 1);	<pre>alt.AddToken((int) SyntaxConstants.RETURN , 1, 1);</pre>
<pre>"Prod_ElemChoice");</pre>	pattern.AddAlternative (alt);	<pre>pattern.AddAlternative (alt);</pre>	alt.AddToken((int)
<pre>alt = new ProductionPatternAlter native();</pre>	AddPattern(pattern);  pattern = new	AddPattern(pattern);	SyntaxConstants.O_PAREN, 1, 1);
alt.AddProduction((int	ProductionPattern((int ) SyntaxConstants.PROD A	<pre>pattern = new ProductionPattern((int )</pre>	<pre>alt.AddProduction((int ) SyntaxConstants.PROD R</pre>
SyntaxConstants.PROD_E LEMENT, 1, 1);	DD_ELEM,	SyntaxConstants.PROD_M 2_ELEM,	ETURN, 0, 1);
<pre>pattern.AddAlternative (alt);</pre>	"Prod_addElem"); alt = new ProductionPatternAlter	"Prod_M2_Elem"); alt = new	<pre>alt.AddToken((int) SyntaxConstants.C_PARE N, 1, 1);</pre>
<pre>alt = new ProductionPatternAlter native();</pre>	<pre>native(); alt.AddToken((int)</pre>	<pre>ProductionPatternAlter native();</pre>	alt.AddToken((int) SyntaxConstants.SEMIC,
alt.AddToken((int)	SyntaxConstants.COMMA, 1, 1);	<pre>alt.AddProduction((int )</pre>	1, 1);

	SyntaxConstants.COMMA,		)
alt.AddToken((int) SyntaxConstants.C BRAC	1, 1);	<pre>alt.AddProduction((int )</pre>	SyntaxConstants.PROD_B ODY, 0, 1);
KET, 1, 1);	<pre>alt.AddProduction((int )</pre>	SyntaxConstants.PROD_R ETURN, 1, 1);	alt.AddToken((int)
alt.AddProduction((int	SyntaxConstants.PROD_D TYPE A, 0, 1);	alt.AddToken((int)	SyntaxConstants. C_BRAC KET, 1, 1);
SyntaxConstants.PROD_G	111 E_N, 0, 1/,	SyntaxConstants. C_PARE	ND1, 1, 1/,
LOBAL_DEC, 0, 1);	<pre>pattern. AddAlternative (alt);</pre>	N, 1, 1);	<pre>alt.AddProduction((int )</pre>
<pre>pattern. AddAlternative (alt);</pre>	AddPattern(pattern);	pattern.AddAlternative (alt);	SyntaxConstants.PROD_G LOBAL_DEC, 0, 1);
		alt = new	
AddPattern(pattern);	<pre>pattern = new ProductionPattern((int )</pre>	<pre>ProductionPatternAlter native();</pre>	<pre>pattern.AddAlternative (alt);</pre>
<pre>pattern = new ProductionPattern((int</pre>	SyntaxConstants.PROD R	alt.AddToken((int)	AddPattern(pattern);
)	ETURN,	SyntaxConstants. ID, 1,	Ţ,,
SyntaxConstants.PROD_D	<b>"</b> — <b>"</b>	1);	pattern = new
TYPE_A,	"Prod_return"); alt = new	alt.AddToken((int)	ProductionPattern((int
"Prod dtypeA");	ProductionPatternAlter	SyntaxConstants. DOT,	SyntaxConstants.PROD S
alt = new ProductionPatternAlter	native();	1, 1);	TRUCT,
native();	alt.AddProduction((int	alt.AddToken((int)	"Prod struct");
	)	SyntaxConstants.ID, 1,	alt = new
<pre>alt.AddProduction((int )</pre>	<pre>SyntaxConstants.PROD_L ITERALS, 1, 1);</pre>	1);	ProductionPatternAlter native();
SyntaxConstants.PROD_D		pattern.AddAlternative	
ATATYPE, 0, 1);	pattern.AddAlternative (alt);	(alt);	alt.AddToken((int) SyntaxConstants.O_BRAC
alt.AddToken((int)	alt = new	AddPattern(pattern);	KET, 1, 1);
<pre>SyntaxConstants.ID, 1, 1);</pre>	<pre>ProductionPatternAlter native();</pre>	pattern = new	alt.AddProduction((int
alt.AddProduction((int	alt.AddProduction((int	ProductionPattern((int	) SyntaxConstants.PROD M
)	)	SyntaxConstants.PROD F	EM DEC, 0, 1);
${\tt SyntaxConstants.PROD\_N}$	${\tt SyntaxConstants.PROD\_N}$	UNCTVOID,	
1, 0, 1);	EGATE, 0, 1);	"D 1 C	alt. AddToken((int)
alt.AddProduction((int	alt.AddToken((int)	"Prod_functvoid"); alt = new	SyntaxConstants.C_BRAC KET, 1, 1);
)	SyntaxConstants. ID, 1,	ProductionPatternAlter	
SyntaxConstants.PROD_E XDTYPE_A, 0, 1);	1);	native();	<pre>alt.AddToken((int) SyntaxConstants.SEMIC,</pre>
pattern.AddAlternative	alt.AddProduction((int	alt.AddToken((int)	1, 1);
(alt);	SyntaxConstants.PROD_0	SyntaxConstants.O_PAREN, 1, 1);	alt.AddProduction((int
AddPattern(pattern);	UT_C, 0, 1);	alt.AddProduction((int	SyntaxConstants.PROD_G
pattern = new	<pre>pattern.AddAlternative (alt);</pre>	) SyntaxConstants.PROD D	LOBAL_DEC, 0, 1);
ProductionPattern((int	alt = new	TYPE_A, 0, 1);	alt.AddProduction((int
)	${\tt ProductionPatternAlter}$		)
SyntaxConstants.PROD_E XDTYPE_A,	native();	alt.AddToken((int) SyntaxConstants.C_PARE	SyntaxConstants.PROD_C LEAR, 0, 1);
"D. 1 EV1, 4"\	alt.AddToken((int)	N, 1, 1);	
"Prod_EXdtypeA"); alt = new	SyntaxConstants.SQROOT , 1, 1);	alt.AddToken((int)	<pre>pattern.AddAlternative (alt);</pre>
ProductionPatternAlter	, 1, 1/,	SyntaxConstants. O_BRAC	(u10),
native();	alt.AddToken((int) SyntaxConstants.O_PARE	KET, 1, 1);	AddPattern(pattern);
alt.AddToken((int)	N, 1, 1);	alt.AddProduction((int	

pattern = new	)		
ProductionPattern((int	SyntaxConstants.PROD N	alt.AddProduction((int	alt.AddToken((int)
)	1, 1, 1);	)	SyntaxConstants. VOID,
SyntaxConstants. PROD M	1, 1, 1/,	SyntaxConstants.PROD G	1, 1);
EM DEC.	pattern.AddAlternative	LOBAL DEC, 0, 1);	1, 1/,
EM_DEC,		LUDAL_DEC, 0, 1);	1. 411771 (/: .)
"D 1 D ")	(alt);	1	alt.AddToken((int)
"Prod_memDec");		alt.AddProduction((int	SyntaxConstants. ID, 1,
alt = new	AddPattern(pattern);	)	1);
ProductionPatternAlter		SyntaxConstants.PROD_C	
native();	pattern = new	OMMENTS, 0, 1);	alt.AddProduction((int
	ProductionPattern((int		)
alt.AddProduction((int	)	alt.AddProduction((int	SyntaxConstants.PROD_F
)	SyntaxConstants.PROD I	)	UNCTVOID1, 1, 1);
SyntaxConstants.PROD D	NIT DEC CHOICE,	SyntaxConstants.PROD C	
ATATYPE, 1, 1);	,	LEAR, 0, 1);	alt.AddProduction((int
1, 1,	"Prod initDecChoice");	Elint, o, 1/,	)
alt.AddToken((int)	alt = new	pattern.AddAlternative	SyntaxConstants. PROD C
		=	_
SyntaxConstants. ID, 1,	ProductionPatternAlter	(alt);	OMMENTS, 0, 1);
1);	native();		
		AddPattern(pattern);	alt.AddProduction((int
alt.AddProduction((int	alt.AddToken((int)		)
)	SyntaxConstants.COMMA,	pattern = new	SyntaxConstants.PROD_C
SyntaxConstants.PROD I	1, 1);	ProductionPattern((int	LEAR, 0, 1);
NIT DEC, 0, 1);		)	
_ , , , ,	alt.AddToken((int)	SyntaxConstants.PROD L	pattern.AddAlternative
alt.AddToken((int)	SyntaxConstants. ID, 1,	OCAL CHOICE,	(alt);
SyntaxConstants. SEMIC,	1);	oone_onorce,	alt = new
	1),	"D 1 I 1Ch-:").	
1, 1);	1. 1175	"Prod_LocalChoice");	ProductionPatternAlter
	alt.AddProduction((int	alt = new	native();
alt.AddProduction((int	)	ProductionPatternAlter	
)	SyntaxConstants.PROD_I	native();	alt.AddToken((int)
SyntaxConstants.PROD_M	<pre>NIT_DEC_CHOICE, 0, 1);</pre>		SyntaxConstants.STRUCT
EM_DEC, 0, 1);		alt.AddProduction((int	_N, 1, 1);
	pattern.AddAlternative	)	
pattern. AddAlternative	(alt);	SyntaxConstants.PROD D	alt.AddToken((int)
(alt);	(	ATATYPE, 1, 1);	SyntaxConstants. ID, 1,
(410),	AddPattern(pattern);	MIMITE, 1, 1/,	1);
AddPattern(pattern);	Addi attern (pattern),	alt.AddToken((int)	1),
Addi attern (pattern),			14 A 11D: 1: 4: - (/:-4
	pattern = new	SyntaxConstants. ID, 1,	alt.AddProduction((int
pattern = new	ProductionPattern((int	1);	)
ProductionPattern((int	)		SyntaxConstants.PROD_S
)	SyntaxConstants.PROD_C	alt.AddProduction((int	TRUCT1, 1, 1);
SyntaxConstants.PROD_I	ONSTANT,	)	
NIT_DEC,		SyntaxConstants.PROD D	alt.AddProduction((int
	"Prod constant");	ECLARE1, 1, 1);	)
"Prod initDec");	alt = new	, _, _,,	SyntaxConstants.PROD C
alt = new	ProductionPatternAlter	alt.AddProduction((int	OMMENTS, 0, 1);
ProductionPatternAlter		art. Addi rodde troii ((Tiit	OMMENTS, O, 1/,
	native();	)	1. 110 1 //
native();	1	SyntaxConstants. PROD_C	alt.AddProduction((int
	alt.AddToken((int)	OMMENTS, 0, 1);	)
alt.AddProduction((int	SyntaxConstants. EQUALS		SyntaxConstants.PROD_C
)	, 1, 1);	alt.AddProduction((int	LEAR, 0, 1);
SyntaxConstants.PROD_I		)	
NIT DEC CHOICE, 1, 1);	alt.AddProduction((int	SyntaxConstants.PROD C	pattern.AddAlternative
, , , , ,	)	LEAR, 0, 1);	(alt);
pattern. AddAlternative	SyntaxConstants.PROD L	, -, -, ,	alt = new
(alt);	. –	nattorn AddAltornative	ProductionPatternAlter
	ITERALS, 1, 1);	pattern. AddAlternative	
alt = new	1. 4110 1 //	(alt);	native();
ProductionPatternAlter	alt.AddToken((int)	alt = new	
native();	SyntaxConstants.SEMIC,	ProductionPatternAlter	alt.AddToken((int)
	1, 1);	native();	SyntaxConstants.CONST_
alt.AddProduction((int			N, 1, 1);

	)		pattern = new
alt.AddToken((int)	SyntaxConstants. PROD_L	alt.AddToken((int)	ProductionPattern((int
SyntaxConstants.ID, 1,	OCAL_CHOICE, 0, 1);	SyntaxConstants.C_BRAC	)
1);		KET, 1, 1);	SyntaxConstants.PROD_S
-1+ AddDd+:((:-+	pattern. AddAlternative	-1+ AddDd+:((:-+	TRUCT1,
alt.AddProduction((int	(alt);	alt.AddProduction((int	"Prod struct1");
SyntaxConstants.PROD_C	AddPattern(pattern);	SyntaxConstants.PROD L	alt = new
ONSTANT1, 1, 1);	•	OCAL_CHOICE, 1, 1);	ProductionPatternAlter
	pattern = new		<pre>native();</pre>
alt.AddProduction((int	ProductionPattern((int	pattern. AddAlternative	1. 110 1 //: .)
) SyntaxConstants.PROD C	) SyntaxConstants.PROD_F	(alt);	alt.AddToken((int) SyntaxConstants.O BRAC
OMMENTS, 0, 1);	UNCTRET1,	AddPattern(pattern);	KET, 1, 1);
, ,	,	4	, , , ,
alt.AddProduction((int	"Prod_functret1");	pattern = new	alt.AddProduction((int
)	alt = new	ProductionPattern((int	)
SyntaxConstants.PROD_C LEAR, 0, 1);	ProductionPatternAlter native();	) SyntaxConstants.PROD_F	SyntaxConstants.PROD_M EM DEC, 0, 1);
LEAR, 0, 1),	mative(),	UNCTVOID1,	EM_DEC, 0, 1),
pattern.AddAlternative	alt.AddToken((int)	cherroldi,	alt.AddToken((int)
(alt);	SyntaxConstants.O_PARE	"Prod_functvoid1");	SyntaxConstants.C_BRAC
	N, 1, 1);	alt = new	KET, 1, 1);
AddPattern(pattern);	14 A 1 ID. 1. 4 1 . (/1.4	ProductionPatternAlter	1. 1171 . ((:)
pattern = new	alt.AddProduction((int	native();	alt.AddToken((int) SyntaxConstants.SEMIC,
ProductionPattern((int	SyntaxConstants.PROD D	alt.AddToken((int)	1, 1);
)	TYPE_A, 0, 1);	SyntaxConstants.O_PARE	
SyntaxConstants.PROD_D		N, 1, 1);	alt.AddProduction((int
ECLARE1,	alt.AddToken((int)	1. 4110 1 //	)
"Prod Declarel");	SyntaxConstants.C_PARE N, 1, 1);	alt.AddProduction((int	SyntaxConstants.PROD_LOCAL_CHOICE, 0, 1);
alt = new	Ν, 1, 1/,	SyntaxConstants.PROD D	OCAL_CHOICE, 0, 1/,
ProductionPatternAlter	alt.AddToken((int)	TYPE_A, 0, 1);	pattern.AddAlternative
<pre>native();</pre>	SyntaxConstants.O_BRAC		(alt);
1. 4110 1 //	KET, 1, 1);	alt.AddToken((int)	4.115 ( )
alt.AddProduction((int	alt.AddProduction((int	SyntaxConstants.C_PARE N, 1, 1);	AddPattern(pattern);
SyntaxConstants.PROD D	)	N, 1, 1/,	pattern = new
ECLARE_CHOICE, 0, 1);	SyntaxConstants. PROD_B	alt.AddToken((int)	ProductionPattern((int
	ODY, 0, 1);	SyntaxConstants.O_BRAC	)
alt.AddToken((int)		KET, 1, 1);	SyntaxConstants.PROD_C
SyntaxConstants. SEMIC,	alt.AddToken((int) SyntaxConstants.RETURN	alt.AddProduction((int	ONSTANT1,
1, 1);	, 1, 1);	art. Addrioduction((int	"Prod_constant1");
alt.AddProduction((int	, 1, 1,	SyntaxConstants.PROD_B	alt = new
)	alt.AddToken((int)	ODY, 0, 1);	ProductionPatternAlter
SyntaxConstants.PROD_L	SyntaxConstants.O_PARE		native();
OCAL_CHOICE, 0, 1);	N, 1, 1);	alt. AddToken((int)	alt.AddToken((int)
pattern.AddAlternative	alt.AddProduction((int	SyntaxConstants.C_BRAC KET, 1, 1);	SyntaxConstants. EQUALS
(alt);	)	REI, I, I),	, 1, 1);
alt = new	SyntaxConstants. PROD_R	alt.AddProduction((int	
ProductionPatternAlter	ETURN, 0, 1);	)	alt.AddProduction((int
<pre>native();</pre>	ol+ AddT-1(/:)	SyntaxConstants.PROD_L	) SyntoyCt DDOD I
alt.AddProduction((int	alt.AddToken((int) SyntaxConstants.C PARE	OCAL_CHOICE, 0, 1);	SyntaxConstants.PROD_L ITERALS, 1, 1);
)	N, 1, 1);	pattern.AddAlternative	11LIMILD, 1, 1/,
SyntaxConstants.PROD_F		(alt);	alt.AddToken((int)
UNCTRET1, 1, 1);	alt.AddToken((int)		SyntaxConstants. SEMIC,
-14 AddDdd(/t	SyntaxConstants.SEMIC,	AddPattern(pattern);	1, 1);
alt.AddProduction((int	1, 1);		

alt.AddProduction((int	alt = new ProductionPatternAlter native();	pattern.AddAlternative (alt):	SyntaxConstants.SEMIC, 1, 1);
SyntaxConstants.PROD_L	native(),	(a11),	alt.AddProduction((int
OCAL_CHOICE, 0, 1);	alt.AddProduction((int	AddPattern(pattern);	)
	)		SyntaxConstants.PROD_A
alt.AddProduction((int	SyntaxConstants. PROD_A	pattern = new	SSIGN_CHOICE, 0, 1);
) SyntaxConstants.PROD C	CCESS_ASSIGN_DTYPE, 1, 1);	ProductionPattern((int	pattern.AddAlternative
OMMENTS, 0, 1);	1),	SyntaxConstants.PROD_A	(alt);
, ,	alt.AddProduction((int	CCESS_ASSIGN_DTYPE,	, , ,
alt.AddProduction((int	)		AddPattern(pattern);
)	SyntaxConstants.PROD_B	"Prod_AccessAssignDtyp	
SyntaxConstants.PROD_C LEAR, 0, 1);	ODY, 0, 1);	e"); alt = new	pattern = new ProductionPattern((int
LEAR, U, 1),	pattern.AddAlternative	ProductionPatternAlter	)
pattern. AddAlternative	(alt);	native();	SyntaxConstants.PROD_A
(alt);	alt = new		SSIGNING,
	ProductionPatternAlter	alt.AddToken((int)	
AddPattern(pattern);	native();	<pre>SyntaxConstants.ID, 1, 1);</pre>	"Prod_assigning"); alt = new
pattern = new	alt.AddProduction((int		ProductionPatternAlter
ProductionPattern((int	)	alt.AddProduction((int	native();
) SyntaxConstants.PROD M	SyntaxConstants.PROD_M NT COND T, 1, 1);	) SyntaxConstants.PROD A	alt.AddToken((int)
AIN,	N1_COND_1, 1, 1),	RRAY ID, 0, 1);	SyntaxConstants. EQUALS
,	alt.AddToken((int)		, 1, 1);
"Prod_main");	SyntaxConstants. SEMIC,	alt.AddProduction((int	
alt = new	1, 1);	)	pattern. AddAlternative
<pre>ProductionPatternAlter native();</pre>	alt.AddProduction((int	SyntaxConstants.PROD_A	(alt); alt = new
native(),	)	SSIGN_VALUE_CHOICE, 1, 1);	ProductionPatternAlter
alt.AddToken((int)	SyntaxConstants.PROD B	17,	native();
SyntaxConstants.MAIN_N	ODY, 0, 1);	pattern.AddAlternative	
, 1, 1);		(alt);	alt.AddProduction((int
alt.AddToken((int)	<pre>pattern.AddAlternative (alt);</pre>	AddPattern(pattern);	) SyntaxConstants.PROD A
SyntaxConstants. O PARE	alt = new	Addi attern (pattern),	SSIGN_SYM, 1, 1);
N, 1, 1);	ProductionPatternAlter	pattern = new	<u>-</u> , -, -, -, -,
	<pre>native();</pre>	ProductionPattern((int	alt.AddProduction((int
alt.AddToken((int)	1 110 1 ((1)	)	)
SyntaxConstants.C_PARE	alt.AddToken((int)	SyntaxConstants.PROD_A SSIGN_VALUE_CHOICE,	SyntaxConstants.PROD_M
N, 1, 1);	SyntaxConstants.REVERS E, 1, 1);	SSIGN_VALUE_CHOICE,	ATH_OP, 1, 1);
alt.AddToken((int)	_, _, _,,	"Prod_assignValueChoic	alt.AddToken((int)
SyntaxConstants.O_BRAC	alt.AddToken((int)	e");	SyntaxConstants. SEMIC,
KET, 1, 1);	SyntaxConstants.O_PARE	alt = new	1, 1);
pattern.AddAlternative	N, 1, 1);	<pre>ProductionPatternAlter native();</pre>	nottown AddAltownotive
(alt);	alt.AddProduction((int	native(),	pattern.AddAlternative (alt);
(	)	alt.AddProduction((int	alt = new
AddPattern(pattern);	SyntaxConstants.PROD_R	)	ProductionPatternAlter
	ETURN, 1, 1);	SyntaxConstants. PROD_A	native();
pattern = new	alt.AddToken((int)	SSIGNING, 1, 1);	alt.AddToken((int)
ProductionPattern((int )	alt. Addloken((int) SyntaxConstants. C_PARE	alt.AddProduction((int	alt. Addloken((int) SyntaxConstants. DOT,
SyntaxConstants.PROD_A	N, 1, 1);	)	1, 1);
SSIGN_CHOICE,		SyntaxConstants.PROD_A	
<i>n</i> =	alt.AddToken((int)	SSIGN_VALUE, 0, 1);	alt.AddToken((int)
"Prod_assignChoice");	SyntaxConstants.SEMIC,	ol+ AddT-1(/:/\	SyntaxConstants. ID, 1,
	1, 1);	alt.AddToken((int)	1);

		SyntaxConstants.EQUALS	alt = new
alt.AddToken((int)	alt.AddToken((int)	, 1, 1);	ProductionPatternAlter
SyntaxConstants. EQUALS	SyntaxConstants.S_CBRA		<pre>native();</pre>
, 1, 1);	CKET, 1, 1);	pattern.AddAlternative	
		(alt);	alt.AddToken((int)
pattern.AddAlternative	alt.AddProduction((int		SyntaxConstants.CONTAI
(alt);	)	AddPattern(pattern);	NS, 1, 1);
alt = new	SyntaxConstants.PROD_A		
ProductionPatternAlter	RRAY_IDTAIL, 0, 1);	pattern = new	alt.AddToken((int)
native();		ProductionPattern((int	SyntaxConstants.O_PARE
	pattern.AddAlternative	)	N, 1, 1);
alt.AddProduction((int	(alt);	SyntaxConstants.PROD_A	
)		SSIGN_VALUE,	alt.AddProduction((int
SyntaxConstants.PROD_M	AddPattern(pattern);		)
NT, 1, 1);		"Prod_assignValue");	SyntaxConstants.PROD_R
	pattern = new	alt = new	ETURN, 1, 1);
pattern. AddAlternative	ProductionPattern((int	ProductionPatternAlter	
(alt);	)	native();	alt.AddToken((int)
alt = new	SyntaxConstants.PROD_A	1	SyntaxConstants.C_PARE
ProductionPatternAlter	RRAY_IDTAIL,	alt.AddProduction((int	N, 1, 1);
native();	"D 1 A TD" 11"\	)	, , , , , , , , , , , , , , , , , , ,
1. 110 1 //: .	"Prod_ArrayIDTail");	SyntaxConstants.PROD_M	pattern. AddAlternative
alt.AddProduction((int	alt = new	ATH_OP, 1, 1);	(alt);
)	ProductionPatternAlter	1, 4110 1 //: ,	4.1.ID
SyntaxConstants. PROD_F	native();	alt.AddProduction((int	AddPattern(pattern);
UNCT_PARAM, 1, 1);	alt AddTakan((;nt)	) SyntaxConstants.PROD F	nottonn = now
pattern. AddAlternative	alt.AddToken((int) SyntaxConstants.S_OBRA	UNCT PARAM, 0, 1);	<pre>pattern = new ProductionPattern((int</pre>
(alt);	CKET, 1, 1);	UNCI_IANAM, 0, 1),	)
alt = new	CRE1, 1, 1),	pattern.AddAlternative	SyntaxConstants.PROD F
ProductionPatternAlter	alt.AddProduction((int	(alt);	UNCT PARAM,
native();	)	(411),	ower_i mam,
native (),	SyntaxConstants.PROD_I	AddPattern(pattern);	"Prod functParam");
alt.AddToken((int)	NDEX, 1, 1);	naar accorn (paccorn) ,	alt = new
SyntaxConstants. ID, 1,	1, 1, 1,	pattern = new	ProductionPatternAlter
1);	alt.AddToken((int)	ProductionPattern((int	<pre>native();</pre>
	SyntaxConstants.S CBRA	)	
pattern.AddAlternative	CKET, 1, 1);	SyntaxConstants.PROD_C	alt.AddToken((int)
(alt);		ONVERT,	SyntaxConstants.O_PARE
	pattern.AddAlternative		N, 1, 1);
AddPattern(pattern);	(alt);	"Prod_Convert");	
		alt = new	alt.AddProduction((int
pattern = new	AddPattern(pattern);	ProductionPatternAlter	)
ProductionPattern((int		native();	SyntaxConstants.PROD_F
)	pattern = new		UNCT_IDPARAM, 0, 1);
SyntaxConstants.PROD_A	ProductionPattern((int	alt.AddToken((int)	
RRAY_ID,	)	SyntaxConstants.TOCHAR	alt.AddToken((int)
	SyntaxConstants.PROD_A	, 1, 1);	SyntaxConstants.C_PARE
"Prod_ArrayID");	SSIGN_SYM,		N, 1, 1);
alt = new		pattern. AddAlternative	
ProductionPatternAlter	"Prod_AssignSym");	(alt);	pattern.AddAlternative
native();	alt = new	alt = new	(alt);
1. 117 1 ((* .)	ProductionPatternAlter	ProductionPatternAlter	alt = new
alt.AddToken((int)	native();	native();	ProductionPatternAlter
SyntaxConstants. S_OBRA	ol+ AddDmodu-+://:-	ol+ AddToker(/:-+)	native();
CKET, 1, 1);	alt.AddProduction((int	alt.AddToken((int)	olt AddDnoductica (//:
alt.AddProduction((int	) SyntaxConstants.PROD 0	SyntaxConstants.LENGTH F, 1, 1);	alt.AddProduction((int
)	PER SYM, 1, 1);	Γ, 1, 1/,	SyntaxConstants.PROD 0
SyntaxConstants.PROD I	IEN_SIM, 1, 1/,	pattern.AddAlternative	PER SYM, 1, 1);
NDEX, 1, 1);	alt.AddToken((int)	(alt);	1 DR_OIM, 1, 1/,
10DA, 1, 1/,	art. Madroken ( (1111)	(αι υ/ ,	alt.AddProduction((int
			aro. Man round tron (till

) SyntaxConstants.PROD_0 PERAND, 1, 1);	alt.AddProduction((int))	) SyntaxConstants.PROD_F OR_STATE, 1, 1);	<pre>alt = new ProductionPatternAlter native();</pre>
alt.AddProduction((int	SyntaxConstants.PROD_A DDFUNCT_IDPARAM, 0, 1);	<pre>pattern.AddAlternative (alt);</pre>	<pre>alt.AddProduction((int )</pre>
SyntaxConstants.PROD_F UNCT_PARAM, 0, 1);	pattern.AddAlternative	<pre>alt = new ProductionPatternAlter</pre>	SyntaxConstants.PROD_C OMMENTS, 1, 1);
pattern. AddAlternative	(alt);	native();	alt.AddProduction((int
(alt);	AddPattern(pattern);	alt.AddProduction((int )	SyntaxConstants. PROD_B
AddPattern(pattern);	<pre>pattern = new ProductionPattern((int</pre>	<pre>SyntaxConstants.PROD_I FELSE, 1, 1);</pre>	ODY, 0, 1);
<pre>pattern = new ProductionPattern((int )</pre>	) SyntaxConstants.PROD_B	pattern. AddAlternative	<pre>pattern.AddAlternative   (alt);</pre>
)	ODY,	(alt);	
SyntaxConstants.PROD_F UNCT_IDPARAM,	"Prod_body"); alt = new	<pre>alt = new ProductionPatternAlter native();</pre>	<pre>ProductionPatternAlter native();</pre>
"Prod_functIDParam"); alt = new	ProductionPatternAlter native();	alt.AddProduction((int	<pre>alt.AddProduction((int )</pre>
ProductionPatternAlter	native () ,	)	SyntaxConstants.PROD C
native();	<pre>alt.AddProduction((int )</pre>	SyntaxConstants.PROD_D  OWHILE, 1, 1);	LEAR, 1, 1);
alt.AddProduction((int	SyntaxConstants.PROD L		alt.AddProduction((int
)	OCAL_CHOICE, 1, 1);	pattern.AddAlternative	)
SyntaxConstants. PROD 0	ocid_chored, 1, 1,	(alt);	SyntaxConstants.PROD_B
_	-1+ AddDd+i(/i-+		ODY, $0, 1$ ;
PERAND, 1, 1);	alt.AddProduction((int)	alt = new ProductionPatternAlter	, ,
alt.AddProduction((int )	SyntaxConstants.PROD_B ODY, 0, 1);	native();	<pre>pattern.AddAlternative (alt);</pre>
SyntaxConstants.PROD_A		alt.AddProduction((int	alt = new
DDFUNCT_IDPARAM, 0, 1);	<pre>pattern.AddAlternative (alt);</pre>	) SyntaxConstants.PROD_W	<pre>ProductionPatternAlter native();</pre>
	alt = new	HILE_STATE, 1, 1);	
pattern.AddAlternative	ProductionPatternAlter		alt.AddToken((int)
(alt);	native();	<pre>pattern.AddAlternative (alt);</pre>	SyntaxConstants.BREAK, 1, 1);
AddPattern(pattern);	alt.AddProduction((int )	alt = new ProductionPatternAlter	alt.AddToken((int)
<pre>pattern = new ProductionPattern((int</pre>	SyntaxConstants.PROD_P	<pre>native();</pre>	SyntaxConstants.O_PARE
)	RINT, 1, 1);	alt.AddProduction((int	N, 1, 1);
SyntaxConstants.PROD A	pattern. AddAlternative	)	alt.AddToken((int)
DDFUNCT IDPARAM,	(alt);	SyntaxConstants.PROD S	SyntaxConstants.C PARE
,	alt = new	WITCH STATE, 1, 1);	N, 1, 1);
"Prod_addfunctIDParam"	ProductionPatternAlter	,,,,, ,	, -, -, -,
); alt = new	native();	<pre>pattern.AddAlternative (alt);</pre>	<pre>alt.AddToken((int) SyntaxConstants.SEMIC,</pre>
ProductionPatternAlter	alt.AddProduction((int	alt = new	1, 1);
native();	)	ProductionPatternAlter	
alt.AddToken((int)	<pre>SyntaxConstants.PROD_S CAN, 1, 1);</pre>	native();	<pre>pattern.AddAlternative (alt);</pre>
SyntaxConstants.COMMA,		alt.AddProduction((int	
1, 1);	<pre>pattern.AddAlternative (alt);</pre>	) SyntaxConstants.PROD_A	AddPattern(pattern);
alt.AddProduction((int	alt = new	SSIGN_CHOICE, 1, 1);	pattern = new
)	ProductionPatternAlter		ProductionPattern((int
SyntaxConstants.PROD_0 PERAND, 1, 1);	native();	pattern.AddAlternative (alt);	) SyntaxConstants.PROD P
, -, -/, 1	alt.AddProduction((int	·,	RINT,

"5 , , , ")		DDFUNCT_IDPARAM, O,	1 115 1 (//
"Prod_print");	pattern = new	1);	alt.AddToken((int)
alt = new	ProductionPattern((int	-1+ AddT-1(/:)	SyntaxConstants. PLUS,
ProductionPatternAlter native();	SyntaxConstants.PROD 0	alt.AddToken((int) SyntaxConstants.C PARE	1, 1);
native(),	UT,	N, 1, 1);	alt.AddProduction((int
alt.AddToken((int)	01,	N, 1, 1),	)
SyntaxConstants. PRINT_	"Prod Out");	pattern.AddAlternative	SyntaxConstants.PROD 0
N, 1, 1);	alt = new	(alt);	UT, 1, 1);
, , , ,	ProductionPatternAlter	alt = new	, ,
alt.AddToken((int)	native();	ProductionPatternAlter	alt.AddProduction((int
SyntaxConstants.O_PARE		native();	)
N, 1, 1);	alt.AddProduction((int		SyntaxConstants.PROD_C
	)	alt.AddToken((int)	ONCAT_LIT, $0, 1)$ ;
alt.AddProduction((int	SyntaxConstants.PROD_R	SyntaxConstants.DOT,	
)	ETURN, 1, 1);	1, 1);	pattern.AddAlternative
SyntaxConstants.PROD_P		1	(alt);
OSTVAL, 1, 1);	pattern. AddAlternative	alt.AddProduction((int	A LID
1	(alt);	)	AddPattern(pattern);
alt.AddToken((int)	AddPattern(pattern);	SyntaxConstants.PROD_C	
SyntaxConstants.C_PARE N, 1, 1);	Addrattern (pattern);	ONVERT, 1, 1);	pattern = new ProductionPattern((int
N, 1, 1),	pattern = new	pattern.AddAlternative	)
alt.AddToken((int)	ProductionPattern((int	(alt):	SyntaxConstants.PROD S
SyntaxConstants. SEMIC,	)	(410),	CAN,
1, 1);	SyntaxConstants.PROD 0	AddPattern(pattern);	orn.,
-, -, ,	UT C,	, , , , , , , , , , , , , , , , , , ,	"Prod scan");
alt.AddProduction((int	_ ,	pattern = new	alt = new
)	"Prod_OutC");	ProductionPattern((int	ProductionPatternAlter
SyntaxConstants.PROD_B	alt = new	)	native();
ODY, 0, 1);	ProductionPatternAlter	SyntaxConstants.PROD_S	
	native();	TRUCT_C,	alt.AddToken((int)
pattern.AddAlternative			SyntaxConstants.SCAN_N
(alt);	alt.AddProduction((int	"Prod_structC");	, 1, 1);
	)	alt = new	1
AddPattern(pattern);	SyntaxConstants. PROD_A	ProductionPatternAlter	alt.AddToken((int)
nottown = now	RRAY_ID, 1, 1);	native();	SyntaxConstants.O_PARE
pattern = new ProductionPattern((int	alt.AddProduction((int	alt.AddToken((int)	N, 1, 1);
)	)	SyntaxConstants. DOT,	alt.AddToken((int)
SyntaxConstants.PROD_P	SyntaxConstants.PROD S	1, 1);	SyntaxConstants. HASH,
OSTVAL.	TRUCT_C, 0, 1);	1, 1/,	1, 1);
oornie,	11001_0, 0, 1/,	alt.AddToken((int)	1, 1,
"Prod postval");	pattern.AddAlternative	SyntaxConstants. ID, 1,	alt.AddProduction((int
alt = new	(alt);	1);	)
ProductionPatternAlter	alt = new		SyntaxConstants.PROD_R
<pre>native();</pre>	ProductionPatternAlter	pattern.AddAlternative	ETURN, 1, 1);
	native();	(alt);	
alt.AddProduction((int			alt.AddProduction((int
)	alt.AddToken((int)	AddPattern(pattern);	)
SyntaxConstants.PROD_0	SyntaxConstants.O_PARE		SyntaxConstants.PROD_E
UT, 1, 1);	N, 1, 1);	pattern = new	$XT_{I}$ , 0, 1);
14 A 1 ID. 15 4 1 1 // 15 4	14 A 1 ID. 1. 4 1 4 (/1.4	ProductionPattern((int	14 4117 1 . ((:)
alt.AddProduction((int	alt.AddProduction((int	SyntayCanatanta DDOD C	alt.AddToken((int) SyntaxConstants.C PARE
SyntaxConstants.PROD C	SyntaxConstants.PROD 0	SyntaxConstants.PROD_C ONCAT LIT,	N, 1, 1);
ONCAT LIT, 0, 1);	PERAND, 1, 1);	ONOMI_LII,	11, 1, 1/,
Onomi_Lii, 0, 1/,	11/11/11/1/,	"Prod_ConcatLit");	alt.AddToken((int)
pattern.AddAlternative	alt.AddProduction((int	alt = new	SyntaxConstants. SEMIC,
(alt);	)	ProductionPatternAlter	1, 1);
//	SyntaxConstants.PROD A	native();	-, -, ,
AddPattern(pattern);		<del>" '</del>	alt.AddProduction((int

SyntaxConstants. PROD_B	alt.AddToken((int)	alt.AddProduction((int	pattern.AddAlternative
ODY, 0, 1);	<pre>SyntaxConstants. ID, 1, 1);</pre>	) SyntaxConstants.PROD_B	(alt);
pattern.AddAlternative (alt);	alt.AddToken((int)	ODY, 0, 1);	AddPattern(pattern);
	SyntaxConstants. EQUALS	pattern. AddAlternative	pattern = new
AddPattern(pattern);	, 1, 1);	(alt);	ProductionPattern((int)
<pre>pattern = new ProductionPattern((int</pre>	alt.AddProduction((int)	AddPattern(pattern);	SyntaxConstants.PROD_M NT_COND,
)	SyntaxConstants.PROD_V	pattern = new	
SyntaxConstants.PROD_E XT_I,	AL1, 1, 1);	ProductionPattern((int )	"Prod_mntCond"); alt = new
	alt.AddToken((int)	SyntaxConstants.PROD_F	ProductionPatternAlter
"Prod_ExtI"); alt = new	<pre>SyntaxConstants.SEMIC, 1, 1);</pre>	ORSTATEMENT,	<pre>native();</pre>
ProductionPatternAlter		"Prod_forstatement");	alt.AddToken((int)
<pre>native();</pre>	alt.AddToken((int) SyntaxConstants.ID, 1,	alt = new ProductionPatternAlter	SyntaxConstants. ID, 1, 1);
alt.AddToken((int)	1);	native();	
SyntaxConstants.COMMA,		1	alt.AddProduction((int
1, 1);	alt.AddProduction((int )	alt.AddProduction((int )	) SyntaxConstants.PROD_M
alt.AddToken((int) SyntaxConstants.HASH,	SyntaxConstants.PROD_A RRAY_ID, 0, 1);	SyntaxConstants.PROD_B ODY, 1, 1);	NT, 1, 1);
1, 1);	alt.AddProduction((int	pattern.AddAlternative	<pre>pattern. AddAlternative   (alt);</pre>
alt.AddProduction((int	)	(alt);	alt = new
SyntaxConstants.PROD_R ETURN, 1, 1);	SyntaxConstants.PROD_0 P1, 1, 1);	AddPattern(pattern);	<pre>ProductionPatternAlter native();</pre>
alt. AddProduction((int	<pre>alt.AddProduction((int )</pre>	pattern = new ProductionPattern((int	<pre>alt.AddProduction((int )</pre>
)	SyntaxConstants.PROD V	)	SyntaxConstants.PROD M
SyntaxConstants.PROD_E XT_I, 0, 1);	AL1, 1, 1);	SyntaxConstants.PROD_V AL1,	NT_COND_T, 1, 1);
	alt.AddToken((int)		pattern.AddAlternative
<pre>pattern.AddAlternative (alt);</pre>	SyntaxConstants.SEMIC, 1, 1);	"Prod_val1"); alt = new	(alt);
AddPattern(pattern);	alt.AddProduction((int	<pre>ProductionPatternAlter native();</pre>	AddPattern(pattern);
nadrattern (pattern),	)	native (),	pattern = new
<pre>pattern = new ProductionPattern((int</pre>	SyntaxConstants.PROD_M NT COND, 1, 1);	alt.AddToken((int) SyntaxConstants.NUM,	<pre>ProductionPattern((int )</pre>
)		1, 1);	SyntaxConstants.PROD_M
SyntaxConstants.PROD_F OR_STATE,	alt.AddToken((int) SyntaxConstants.C_PARE	pattern.AddAlternative	NT_COND_T,
	N, 1, 1);	(alt);	"Prod_mntCondT");
"Prod_for_state"); alt = new	alt.AddToken((int)	alt = new	alt = new
ProductionPatternAlter	SyntaxConstants.O_BRAC KET, 1, 1);	<pre>ProductionPatternAlter native();</pre>	<pre>ProductionPatternAlter native();</pre>
	, -, -, ,	alt.AddToken((int)	alt.AddProduction((int
alt.AddToken((int)	alt.AddProduction((int	SyntaxConstants. ID, 1,	)
SyntaxConstants. FOR_N,	)	1);	SyntaxConstants.PROD_M
1, 1);	<pre>SyntaxConstants.PROD_F ORSTATEMENT, 0, 1);</pre>	alt.AddProduction((int	NT, 1, 1);
alt.AddToken((int)	1. 41.00	)	alt.AddToken((int)
SyntaxConstants.O_PARE N, 1, 1);	alt.AddToken((int) SyntaxConstants.C_BRAC	SyntaxConstants.PROD_A RRAY ID, 0, 1);	SyntaxConstants. ID, 1, 1);
11, 1, 1/,	KET, 1, 1);	$\text{mati}_{1}$ $\nu$ , $\forall$ , $1$ /,	1/,

pattern = new   ProductionPattern((int   )   )	pattern.AddAlternative (alt);	<pre>SyntaxConstants.C_PARE N, 1, 1);</pre>	pattern.AddAlternative (alt);	<pre>alt = new ProductionPatternAlter native();</pre>
ProductionPattern((int	- -	SyntaxConstants.O_BRAC	-	SyntaxConstants.ELSEIF
SyntaxConstants.PROD_M   SyntaxConstants.PROD_I   SyntaxConstants.PROD_I   PSTATEMENT, 0, 1);   PSTATEMENT, N, 1, 1);   SyntaxConstants.PROD_I   N, 1, 1);   SyntaxConstants.PROD_I   N, 1, 1);   SyntaxConstants.PROD_I   N, 1, 1);   SyntaxConstants.PROD_I   SyntaxConstan		alt.AddProduction((int	÷	
TroductionPatternAlter   NET.   1);   native ();   nati		-		SyntaxConstants.O_PARE
ProductionPatternAlter	= * *			<pre>alt.AddProduction((int )</pre>
alt. AddToken((int)   )	ProductionPatternAlter		${\tt ProductionPatternAlter}$	SyntaxConstants.PROD_I FCONDITION, 1, 1);
SyntaxConstants. INCREM   ENT. 1. 1);		alt.AddProduction((int		
DOY, 1, 1);	SyntaxConstants. INCREM		)	SyntaxConstants.C_PARE
Dattern. AddAlternative		, _, _,,		, -, -, ,
ProductionPatternAlter native(); alt. AddProduction((int native(); alt. AddProduction((int) alt. AddProduction((int) )) syntaxConstants. DECREM SyntaxConstants. PROD_B ENT. 1, 1); SyntaxConstants. PROD_B pattern. AddAlternative (alt); (alt); alt. AddProduction((int) SyntaxConstants. Decrem pattern = new pattern = new pattern = new pattern = new productionPattern((int) ) syntaxConstants. PROD_B pattern. AddAlternative (alt); (	(alt);	)	_	SyntaxConstants.O_BRAC
AddToken((int)   SyntaxConstants.DECRM	ProductionPatternAlter	_	<pre>alt = new ProductionPatternAlter</pre>	alt.AddProduction((int
SyntaxConstants.DECREM ODY, 0, 1); SyntaxConstants.RETURN , 1, 1); ODY, 0, 1); SyntaxConstants.RETURN , 1, 1); alt. AddToken((int) SyntaxConstants.C_BRAC (alt); alt. AddToken((int) SyntaxConstants.C_BRAC (alt); alt. AddToken((int) SyntaxConstants.C_PRAE (alt); alt. AddProduction((int SyntaxConstants.D_PARE AddPattern(pattern); N. 1, 1); alt. AddProduction((int SyntaxConstants.PROD_I pattern = new pattern = new alt. AddProduction((int SyntaxConstants.PROD_I SyntaxConstants.PROD_R ETURN, 0, 1); pattern. AddAlternative (alt); alt. AddToken((int) SyntaxConstants.PROD_I pattern.AddAlternative (alt); alt. AddToken((int) SyntaxConstants.PROD_I pattern.AddAlternative (int) syntaxConstants.O_PARE pattern.AddAlternative (int) syntaxConstants.PROD_I syntaxConstants.PROD_E syntaxCons		alt.AddProduction((int	native();	)
pattern. AddAlternative (alt); (alt); (alt); (alt); (alt) alt. AddToken((int) SyntaxConstants. C_BRAC (int) SyntaxConstants. O_PARE  AddPattern(pattern); AddPattern(pattern);  pattern = new pattern = new pattern = new pattern = new ProductionPattern((int ) SyntaxConstants. PROD_I SyntaxConstants. PROD_I FELSE, FCONDITION,  "Prod_ifelse"); alt = new productionPatternAlter native(); alt. AddToken((int) SyntaxConstants. C_PARE native(); SyntaxConstants. SEMIC, alt. AddToken((int) SyntaxConstants. FROD_I SyntaxConstants. FROD_I SyntaxConstants. FROD_I ProductionPatternAlter native(); syntaxConstants. SEMIC, alt. AddToken((int) SyntaxConstants. FROD_I SyntaxConstants. F, 1, 1); SyntaxConstants. F, 1, 1); SyntaxConstants. F, 1, 1); SyntaxConstants. PROD_R EL_OP, 1, 1); (alt);  pattern. AddAlternative (alt);  prod_elseifstatement' ); SyntaxConstants. O_PARE N, 1, 1); AddProduction((int) SyntaxConstants. O_PARE N, 1, 1);  alt = new native();  pattern. AddAlternative (alt);  prod_elseifstatement' ); SyntaxConstants. PROD_I SyntaxCo	SyntaxConstants.DECREM			SyntaxConstants.PROD_E LSEIFSTATEMENT, 0, 1);
(alt);(alt);alt.AddToken((int)KET, 1, 1);AddPattern(pattern);AddPattern(pattern);N, 1, 1);alt.AddProduction((int)pattern = newpattern = newalt.AddProduction((int)SyntaxConstants.PROD_IProductionPattern((int)ProductionPattern((int))LSEIF, 0, 1);)SyntaxConstants.PROD_IETURN, 0, 1);pattern.AddAlternative (alt);FELSE,FCONDITION,alt.AddToken((int)"Prod_ifelse");"Prod_ifcondition");SyntaxConstants.C_PAREAddPattern(pattern);alt = newnative();syntaxConstants.C_PAREAddPattern(pattern);ProductionPatternAlterpatternpattern = newnative();alt.AddToken((int)ProductionPattern((int)SyntaxConstants.IF, 1, 1);syntaxConstants.SEMIC, 1, 1);LSEIFSTATEMENT, 1, 1);SyntaxConstants.IF, 1, 2)pattern.AddAlternative (alt);pattern.AddAlternative (alt);SyntaxConstants.O_PARE (alt);pattern.AddAlternative (alt);"Prod_elseifstatement"N, 1, 1);alt = newpattern = newN, 1, 1);alt = newpattern = newalt.AddProduction((int)ProductionPatternAlter (int)SyntaxConstants.OPARE (alt);pattern = newnative();syntaxConstants.PROD_ESyntaxConstants.PROD_ISyntaxConstants.PROD_ESyntaxConstants.PROD_ISyntaxConstants.PROD_ESyntaxConstants.PROD_IODY, 1, 1);alt.AddToken((int)SyntaxConstants.PROD_E			, 1, 1);	alt.AddToken((int)
AddPattern(pattern);  Pattern = new pattern. AddProduction((int	_	_		SyntaxConstants.C_BRAC KET, 1, 1);
ProductionPattern((int ) SyntaxConstants.PROD_R   SyntaxConstants.PROD_R   SyntaxConstants.PROD_I   ETURN, 0, 1);   pattern.AddAlternative (alt);   (alt);   alt. AddToken((int)   ProductionPattern(alternative (int));   alt. AddToken((int));   alt. AddProduction((int));	AddPattern(pattern);	AddPattern(pattern);		alt.AddProduction((int
SyntaxConstants.PROD_I SyntaxConstants.PROD_I FELSE, FCONDITION, alt. AddToken((int)  "Prod_ifelse"); alt = new alt = new native(); alt. AddToken((int)  SyntaxConstants.CPARE native(); alt. AddToken((int)  SyntaxConstants.CPARE native(); alt. AddToken((int)  SyntaxConstants.SEMIC, alt. AddToken((int)  SyntaxConstants.SEMIC, l)  alt. AddToken((int)  SyntaxConstants.IF, l, l); SyntaxConstants.PROD_R  EL_OP, l, l); alt. AddToken((int)  SyntaxConstants.OPARE N, l, l);  alt. AddToken((int)  SyntaxConstants.OPARE N, l, l);  alt. AddToken((int)  SyntaxConstants.OPARE N, l, l);  alt = new pattern = new native(); alt = new pattern (pattern); alt = new pattern = new native();  syntaxConstants.OPARE N, l, l);  alt = new pattern = new pattern = new native();  SyntaxConstants.OPARE N, l, l); alt = new pattern = new pattern = new native();  SyntaxConstants.PROD_I  FCONDITION, l, l); alt. AddProduction((int) SyntaxConstants.PROD_E )  SyntaxConstants.PROD_E )  alt. AddProduction((int) SyntaxConstants.PROD_E )  AddPattern(pattern); alt = new pattern = new native(); SyntaxConstants.PROD_E )  SyntaxConstants.PROD_E )  SyntaxConstants.PROD_E )  AddProduction((int) SyntaxCon			alt.AddProduction((int	SyntaxConstants.PROD_E
SyntaxConstants.PROD_I FELSE, FCONDITION, alt. AddToken((int)  "Prod_ifelse"); alt = new ProductionPatternAlter native(); alt. AddToken((int)  alt. AddToken((int)  "Prod_ifelse"); alt = new ProductionPatternAlter native(); alt. AddToken((int)  alt. AddToken((int)  alt. AddToken((int)  syntaxConstants.SEMIC,  alt. AddToken((int)  syntaxConstants. IF, 1,  l);  SyntaxConstants. IF, 1,  l);  SyntaxConstants. PROD_R EL_OP, 1, 1);  alt. AddToken((int)  SyntaxConstants.O_PARE N, 1, 1);  alt. AddToken((int)  SyntaxConstants.O_PARE N, 1, 1);  alt. AddProduction((int)  SyntaxConstants.O_PARE Dattern. AddAlternative AddPattern(pattern); alt = new Dattern Dattern = new Dattern =	rroductionPattern((int	rroductionPattern((int	SymtoxConstants DDOD D	LSEIF, U, I);
"Prod_ifelse");			_	<pre>pattern.AddAlternative (alt);</pre>
alt = new alt = new N, 1, 1);  ProductionPatternAlter ProductionPatternAlter pattern = new native(); alt. AddToken((int) ProductionPattern((int SyntaxConstants. SEMIC, )  alt. AddToken((int) alt. AddProduction((int 1, 1); SyntaxConstants. PROD_E SyntaxConstants. IF, 1, )  1); SyntaxConstants. IF, 1, )  alt. AddToken((int) SyntaxConstants. PROD_R pattern. AddAlternative EL_OP, 1, 1); (alt); "Prod_elseifstatement'); syntaxConstants. O_PARE pattern. AddAlternative AddPattern(pattern); alt = new pattern = new pattern = new pattern = new native();  alt. AddProduction((int ProductionPatternAlter ProductionPattern((int native(); SyntaxConstants. PROD_E			alt.AddToken((int)	
<pre>native();</pre>	alt = new	alt = new	-	
alt. AddProduction((int 1, 1); SyntaxConstants. PROD_E LSEIFSTATEMENT, 1); SyntaxConstants. PROD_R pattern. AddAlternative EL_OP, 1, 1); (alt); "Prod_elseifstatement' ); SyntaxConstants. O_PARE pattern. AddAlternative AddPattern(pattern); alt = new pattern = new native(); ProductionPatternAlter productionPattern((int 1, 2)); alt. AddProduction((int 2, 2)); SyntaxConstants. PROD_I ProductionPatternAlter ProductionPattern((int 3, 2)); SyntaxConstants. PROD_I S				<pre>pattern = new ProductionPattern((int )</pre>
1); SyntaxConstants.PROD_R pattern.AddAlternative EL_OP, 1, 1); (alt); "Prod_elseifstatement' ); SyntaxConstants.O_PARE pattern.AddAlternative AddPattern(pattern); alt = new ProductionPatternAlter alt = new pattern = new native(); alt.AddProduction((int ProductionPatternAlter ProductionPattern((int native(); ) alt.AddProduction((int SyntaxConstants.PROD_E )	alt.AddToken((int)	alt.AddProduction((int		SyntaxConstants.PROD_E
alt. AddToken((int)  SyntaxConstants. O_PARE N, 1, 1);  (alt);  alt = new pattern = new pattern = new productionPatternAlter native();  SyntaxConstants. PROD_I  FCONDITION, 1, 1);  alt. AddProduction((int) SyntaxConstants. PROD_L  ProductionPattern((int) D SyntaxConstants. PROD_E D SyntaxConstants. PROD		) SyntaxConstants.PROD_R	pattern.AddAlternative	LSEIFSTATEMENT,
SyntaxConstants.O_PARE pattern.AddAlternative AddPattern(pattern); alt = new ProductionPatternAlter alt = new pattern = new native();  alt.AddProduction((int ProductionPatternAlter ProductionPattern((int native(); ) alt.AddProduction((int SyntaxConstants.PROD_I SyntaxConstants.PROD_E )  FCONDITION, 1, 1); alt.AddProduction((int LSEIF, SyntaxConstants.PROD_F )  alt.AddToken((int) SyntaxConstants.PROD_L "Prod_elseif");	alt.AddToken((int)	EL_OP, 1, 1);	(alt);	"Prod_elseifstatement" ):
alt.AddProduction((int	SyntaxConstants.O_PARE	-	AddPattern(pattern);	
native();  SyntaxConstants.PROD_I  FCONDITION, 1, 1);  alt.AddProduction((int LSEIF, SyntaxConstants.PROD_E)  ODY, 1, 1);  alt.AddToken((int) SyntaxConstants.PROD_L "Prod_elseif");	alt.AddProduction((int			<pre>native();</pre>
FCONDITION, 1, 1); alt.AddProduction((int LSEIF, SyntaxConstants.PROD_F ODY, 1, 1); alt.AddToken((int) SyntaxConstants.PROD_L "Prod_elseif");	)		)	alt.AddProduction((int
alt.AddToken((int) SyntaxConstants.PROD_L "Prod_elseif");	_	alt.AddProduction((int	-	) SyntaxConstants.PROD_B
0G_0P, 1, 1);	alt.AddToken((int)	SyntaxConstants.PROD_L OG_OP, 1, 1);	"Prod_elseif");	Oν1, 1, 1/;

		alt = new	
<pre>pattern. AddAlternative (alt);</pre>	<pre>pattern.AddAlternative (alt);</pre>	<pre>ProductionPatternAlter native();</pre>	alt.AddProduction((int)
<pre>alt = new ProductionPatternAlter native();</pre>	AddPattern(pattern);	alt.AddToken((int) SyntaxConstants.DO, 1,	SyntaxConstants.PROD_B ODY, 1, 1);
alt.AddToken((int)	<pre>pattern = new ProductionPattern((int</pre>	1);	<pre>pattern.AddAlternative (alt);</pre>
SyntaxConstants.RETURN , 1, 1);	) SyntaxConstants.PROD_E LSESTATEMENT,	alt.AddToken((int) SyntaxConstants.O_BRAC KET, 1, 1);	AddPattern(pattern);
<pre>alt.AddToken((int) SyntaxConstants.O_PARE N, 1, 1);</pre>	"Prod_elsestatement"); alt = new	alt.AddProduction((int)	<pre>pattern = new ProductionPattern((int )</pre>
alt.AddProduction((int	<pre>ProductionPatternAlter native();</pre>	SyntaxConstants.PROD_D OSTATEMENT, 0, 1);	SyntaxConstants.PROD_W HILE_STATE,
SyntaxConstants.PROD_R ETURN, 0, 1);	<pre>alt.AddProduction((int ) SyntaxConstants.PROD_B</pre>	<pre>alt.AddToken((int) SyntaxConstants.C_BRAC KET, 1, 1);</pre>	"Prod_while_state"); alt = new ProductionPatternAlter
alt.AddToken((int) SyntaxConstants.C_PARE N, 1, 1);	ODY, 1, 1); pattern.AddAlternative	alt.AddToken((int) SyntaxConstants.WHILE_	<pre>native(); alt.AddToken((int)</pre>
alt.AddToken((int) SyntaxConstants.SEMIC,	(alt); alt = new ProductionPatternAlter	N, 1, 1);  alt. AddToken((int)	SyntaxConstants.WHILE_ N, 1, 1);
1, 1); pattern.AddAlternative	<pre>native(); alt.AddToken((int)</pre>	<pre>SyntaxConstants.0_PARE N, 1, 1);</pre>	<pre>alt.AddToken((int) SyntaxConstants.O_PARE N, 1, 1);</pre>
(alt);	SyntaxConstants.RETURN , 1, 1);	alt.AddProduction((int))	alt.AddProduction((int
AddPattern(pattern);  pattern = new	alt.AddToken((int) SyntaxConstants.O PARE	SyntaxConstants.PROD_I FCONDITION, 1, 1);	) SyntaxConstants.PROD_I FCONDITION, 1, 1);
ProductionPattern((int)	N, 1, 1);	alt.AddToken((int) SyntaxConstants.C_PARE	alt.AddToken((int)
SyntaxConstants.PROD_E LSE_STATE,	<pre>alt.AddProduction((int ) SyntaxConstants.PROD R</pre>	N, 1, 1); alt.AddToken((int)	SyntaxConstants.C_PARE N, 1, 1);
"Prod_else_state"); alt = new	ETURN, 0, 1);	<pre>SyntaxConstants.SEMIC, 1, 1);</pre>	alt.AddToken((int) SyntaxConstants.O_BRAC
<pre>ProductionPatternAlter native();</pre>	<pre>alt.AddToken((int) SyntaxConstants.C_PARE N, 1, 1);</pre>	<pre>alt.AddProduction((int )</pre>	<pre>KET, 1, 1); alt.AddProduction((int</pre>
<pre>alt.AddToken((int) SyntaxConstants.ELSE_N , 1, 1);</pre>	<pre>alt.AddToken((int) SyntaxConstants.SEMIC, 1, 1);</pre>	SyntaxConstants.PROD_B ODY, 0, 1); pattern.AddAlternative	) SyntaxConstants.PROD_W HILESTATEMENT, 0, 1);
<pre>alt.AddToken((int) SyntaxConstants.O_BRAC KET, 1, 1);</pre>	pattern.AddAlternative (alt);	(alt); AddPattern(pattern);	<pre>alt.AddToken((int) SyntaxConstants.C_BRAC KET, 1, 1);</pre>
<pre>alt.AddProduction((int )</pre>	AddPattern(pattern);	pattern = new ProductionPattern((int	<pre>alt.AddProduction((int )</pre>
SyntaxConstants.PROD_E LSESTATEMENT, 0, 1);	<pre>pattern = new ProductionPattern((int )</pre>	) SyntaxConstants.PROD_D OSTATEMENT,	SyntaxConstants.PROD_B ODY, 0, 1);
alt.AddToken((int) SyntaxConstants.C_BRAC KET, 1, 1);	SyntaxConstants.PROD_D OWHILE,	"Prod_dostatement");  alt = new	<pre>pattern.AddAlternative (alt);</pre>
ND1, 1, 1/,	"Prod_dowhile");	ProductionPatternAlter native();	AddPattern(pattern);

pattern = new		SyntaxConstants.PROD_C	alt = new
ProductionPattern((int	alt.AddToken((int)	ASE_STATE, 0, 1);	ProductionPatternAlter
)	SyntaxConstants.C_BRAC		native();
SyntaxConstants. PROD_W	KET, 1, 1);	pattern.AddAlternative	1. ALID 1 //
HILESTATEMENT,	1, 4110 1 //: ,	(alt);	alt.AddProduction((int
"D. 1 1:1 ( , , , ")	alt.AddProduction((int	A 1 ID + + + + + + + + + + + + + + + + + +	Control Control DDOD D
"Prod_whilestatement")	SyntayConstants DDOD D	AddPattern(pattern);	SyntaxConstants.PROD_B ODY, 1, 1);
; alt = new	SyntaxConstants.PROD_B ODY, 0, 1);	nottown = now	ODY, 1, 1);
ProductionPatternAlter	OD1, O, 1),	<pre>pattern = new ProductionPattern((int</pre>	pattern.AddAlternative
native();	pattern.AddAlternative	)	(alt);
native(),	(alt);	SyntaxConstants.PROD D	(a1t),
alt.AddProduction((int	(a1t),	EF.	AddPattern(pattern);
)	AddPattern(pattern);	Er,	Addi attern (pattern),
SyntaxConstants. PROD B	nadi attern (pattern),	"Prod_def");	pattern = new
ODY, 1, 1);	pattern = new	alt = new	ProductionPattern((int
OD1, 1, 1),	ProductionPattern((int	ProductionPatternAlter	)
pattern. AddAlternative	)	native();	SyntaxConstants.PROD_M
(alt);	SyntaxConstants. PROD_C	nacive (),	ATH OP,
(610)	ASE STATE,	alt.AddToken((int)	,
AddPattern(pattern);	1102_5 11112,	SyntaxConstants. DEFAUL	"Prod MathOp");
man attern (pattern),	"Prod_case_state");	T, 1, 1);	alt = new
pattern = new	alt = new	1, 1, 1,	ProductionPatternAlter
ProductionPattern((int	ProductionPatternAlter	alt.AddToken((int)	native():
)	native();	SyntaxConstants.COLON,	· ,
SyntaxConstants.PROD S	,	1, 1);	alt.AddProduction((int
WITCH_STATE,	alt.AddToken((int)		)
	SyntaxConstants.CASE_N	alt.AddProduction((int	SyntaxConstants.PROD_0
"Prod_switch_state");	, 1, 1);	)	PER_COND, 1, 1);
alt = new		SyntaxConstants.PROD_C	
ProductionPatternAlter	alt.AddProduction((int	ASESTATEMENT, 0, 1);	pattern.AddAlternative
<pre>native();</pre>	)		(alt);
	SyntaxConstants.PROD_L	alt.AddToken((int)	
alt.AddToken((int)	ITERALS, 1, 1);	SyntaxConstants.BREAK,	AddPattern(pattern);
SyntaxConstants.SWITCH		1, 1);	
_N, 1, 1);	alt.AddToken((int)		pattern = new
	SyntaxConstants.COLON,	alt.AddToken((int)	ProductionPattern((int
alt.AddToken((int)	1, 1);	SyntaxConstants.O_PARE	)
SyntaxConstants.O_PARE		N, 1, 1);	SyntaxConstants.PROD_0
N, 1, 1);	alt.AddProduction((int		PER_COND,
	)	alt.AddToken((int)	
alt.AddToken((int)	SyntaxConstants.PROD_C	SyntaxConstants.C_PARE	"Prod_operCond");
SyntaxConstants. ID, 1,	ASESTATEMENT, 0, 1);	N, 1, 1);	alt = new
1);			ProductionPatternAlter
	alt.AddToken((int)	alt.AddToken((int)	native();
alt.AddToken((int)	SyntaxConstants. BREAK,	SyntaxConstants. SEMIC,	1
SyntaxConstants.C_PARE	1, 1);	1, 1);	alt.AddToken((int)
N, 1, 1);			SyntaxConstants.O_PARE
1. 411771 (//)	alt.AddToken((int)	pattern. AddAlternative	N, 1, 1);
alt.AddToken((int)	SyntaxConstants.O_PARE	(alt);	1. 110 1 //: .
SyntaxConstants.O_BRAC	N, 1, 1);	A 11D	alt.AddProduction((int
KET, 1, 1);	1. 11171 . ((:)	AddPattern(pattern);	Control Control DDOD O
-1+ AddDd+:(/:-+	alt.AddToken((int)		SyntaxConstants.PROD_0
alt.AddProduction((int	SyntaxConstants.C_PARE	<pre>pattern = new ProductionPattern((int</pre>	PERAND, 1, 1);
) SyntaxConstants.PROD C	N, 1, 1);	rroductionPattern((int	alt.AddProduction((int
-	alt AddTakan((int)	SymtoxConstants DDOD C	art. Addrioduction((int
ASE_STATE, 1, 1);	alt.AddToken((int) SyntaxConstants.SEMIC,	SyntaxConstants.PROD_C ASESTATEMENT,	SyntayConstants DDOD O
alt.AddProduction((int	SyntaxConstants. SEMIC, 1, 1);	ASESTATEMENT,	SyntaxConstants.PROD_0 PER SYM, 1, 1);
art. Addi roduction ((Int.)	1, 1/,	"Prod casestatement");	IEN_SIM, 1, 1/,
SyntaxConstants. PROD D	alt.AddProduction((int	iiou_casestatement /,	alt.AddProduction((int
EF, 0, 1);	)		)
, v, 1/,	,		,

SyntaxConstants.PROD_0 PER EXT S, 1, 1);	pattern = new ProductionPattern((int	<pre>SyntaxConstants.POWER, 1, 1);</pre>	pattern.AddAlternative
1LK_LK1_0, 1, 1/,	)	1, 1/,	(alt);
alt.AddToken((int) SyntaxConstants.C PARE	SyntaxConstants.PROD_0 PER_SYM,	pattern.AddAlternative (alt):	<pre>alt = new ProductionPatternAlter</pre>
N, 1, 1);		alt = new	native();
alt.AddProduction((int	"Prod_operSym"); alt = new	<pre>ProductionPatternAlter native();</pre>	alt.AddToken((int)
)	ProductionPatternAlter	mative(),	SyntaxConstants. MOD E,
SyntaxConstants.PROD_0	<pre>native();</pre>	alt.AddToken((int)	1, 1);
PER_COND_EXT, 0, 1);	alt.AddToken((int)	SyntaxConstants.DOT,	
pattern. AddAlternative	SyntaxConstants. PLUS.	1, 1);	<pre>pattern.AddAlternative (alt);</pre>
(alt);	1, 1);	pattern.AddAlternative	alt = new
alt = new		(alt);	ProductionPatternAlter
ProductionPatternAlter native();	<pre>pattern.AddAlternative (alt);</pre>	AddPattern(pattern);	native();
1401.0(),	alt = new	naar accern (paccern, ,	alt.AddToken((int)
alt.AddProduction((int	ProductionPatternAlter	pattern = new	SyntaxConstants. EQUALS
) SyntaxConstants. PROD_0	native();	ProductionPattern((int	, 1, 1);
PERAND, 1, 1);	alt.AddToken((int)	SyntaxConstants.PROD_0	pattern.AddAlternative
	SyntaxConstants. MINUS,	PER_EQ,	(alt);
alt.AddProduction((int )	1, 1);	"Prod_operEq");	AddPattern (pattern);
SyntaxConstants.PROD_0	pattern.AddAlternative	alt = new	nuar actern (pactern),
PER_COND_CHOICE, 0,	(alt);	ProductionPatternAlter	pattern = new
1);	alt = new ProductionPatternAlter	native();	ProductionPattern((int
pattern.AddAlternative	native();	alt.AddToken((int)	SyntaxConstants.PROD_0
(alt);		SyntaxConstants.P_E,	PER_EXT_S,
AddPattern(pattern);	alt.AddToken((int) SyntaxConstants.TIMES,	1, 1);	"Prod_operExt_s");
Addi attern (pattern),	1, 1);	pattern.AddAlternative	alt = new
pattern = new		(alt);	ProductionPatternAlter
ProductionPattern((int	<pre>pattern.AddAlternative (alt);</pre>	alt = new ProductionPatternAlter	<pre>native();</pre>
SyntaxConstants. PROD 0	alt = new	native();	alt.AddProduction((int
PER_COND_CHOICE,	${\tt ProductionPatternAlter}$		)
"Prod operCondChoice")	native();	alt. AddToken((int)	SyntaxConstants. PROD_0
:	alt.AddToken((int)	SyntaxConstants.M_E, 1, 1);	PERAND, 1, 1);
alt = new	SyntaxConstants.DIVIDE		alt.AddProduction((int
ProductionPatternAlter	, 1, 1);	pattern.AddAlternative	)
native();	pattern.AddAlternative	(alt); alt = new	SyntaxConstants.PROD_S MATH EXT, 0, 1);
alt.AddProduction((int	(alt);	ProductionPatternAlter	, -, -, -, -, -, -, -, -, -, -, -, -, -,
)	alt = new	native();	pattern. AddAlternative
SyntaxConstants.PROD_O PER SYM, 1, 1);	<pre>ProductionPatternAlter native();</pre>	alt.AddToken((int)	(alt); alt = new
1211_3111, 1, 1,	1401.00,	SyntaxConstants. T_E,	ProductionPatternAlter
alt.AddProduction((int	alt.AddToken((int)	1, 1);	<pre>native();</pre>
) SyntaxConstants.PROD 0	SyntaxConstants.MODULU S, 1, 1);	pattern.AddAlternative	alt.AddToken((int)
PER_EXT_S, 1, 1);	5, 1, 1/,	(alt);	SyntaxConstants. O_PARE
	pattern. AddAlternative	alt = new	N, 1, 1);
<pre>pattern. AddAlternative (alt);</pre>	(alt); alt = new	<pre>ProductionPatternAlter native();</pre>	alt.AddProduction((int
(u± t/ ,	ProductionPatternAlter	native (/,	)
AddPattern(pattern);	<pre>native();</pre>	alt.AddToken((int)	SyntaxConstants. PROD_S
	alt.AddToken((int)	SyntaxConstants.D_E,	IM_MATH_OP, 1, 1);
	art. Audroken ((int)	1, 1);	

	pattern = new		
alt.AddToken((int) SyntaxConstants.C_PARE N, 1, 1);	ProductionPattern((int ) SyntaxConstants.PROD_S	"Prod_operCondExt"); alt = new ProductionPatternAlter	<pre>alt.AddProduction((int ) SyntaxConstants.PROD_0</pre>
1. 4110 1 //	IM_MATH_OP,	<pre>native();</pre>	PERAND, 1, 1);
<pre>alt.AddProduction((int ) SyntaxConstants.PROD 0</pre>	"Prod_simMathOp"); alt = new	alt.AddProduction((int	<pre>alt.AddProduction((int )</pre>
PER_EXT_REP, 0, 1);	ProductionPatternAlter native();	SyntaxConstants.PROD_O PER_SYM, 1, 1);	SyntaxConstants.PROD_R ELOP_EXT, 0, 1);
<pre>pattern. AddAlternative (alt);</pre>	alt.AddProduction((int	alt.AddProduction((int	pattern.AddAlternative
AddPattern(pattern);	SyntaxConstants.PROD_0 PERAND, 1, 1);	SyntaxConstants.PROD_O PER_EXT_S, 1, 1);	AddPattern(pattern);
<pre>pattern = new ProductionPattern((int )</pre>	<pre>alt.AddProduction((int )</pre>	<pre>pattern.AddAlternative (alt);</pre>	pattern = new ProductionPattern((int
SyntaxConstants.PROD_0 PER_EXT_REP,	<pre>SyntaxConstants.PROD_S _MATH_EXT, 0, 1);</pre>	AddPattern(pattern);	) SyntaxConstants.PROD_0 P1,
"Prod_operExt_rep"); alt = new	<pre>pattern.AddAlternative (alt);</pre>	<pre>pattern = new ProductionPattern((int))</pre>	"Prod_op1");
<pre>ProductionPatternAlter native();</pre>	AddPattern(pattern);	SyntaxConstants.PROD_R EL_OP,	<pre>alt = new ProductionPatternAlter native();</pre>
<pre>alt.AddProduction((int ) SyntaxConstants.PROD 0</pre>	<pre>pattern = new ProductionPattern((int )</pre>	"Prod_RelOp"); alt = new	alt.AddToken((int) SyntaxConstants.N_E,
PER_SYM, 1, 1);	SyntaxConstants.PROD_S _MATH_EXT,	ProductionPatternAlter native();	1, 1);
<pre>alt.AddProduction((int ) SyntaxConstants.PROD 0</pre>	"Prod_S_MathExt"); alt = new	alt.AddProduction((int	<pre>pattern.AddAlternative (alt);     alt = new</pre>
PER_EXT_S, 1, 1);	ProductionPatternAlter native();	SyntaxConstants.PROD_0 PERAND, 1, 1);	ProductionPatternAlter
<pre>pattern. AddAlternative (alt);</pre>	alt.AddProduction((int	alt.AddProduction((int	alt.AddToken((int) SyntaxConstants.GREATE
AddPattern(pattern);	SyntaxConstants.PROD_O PER_SYM, 1, 1);	SyntaxConstants.PROD_R ELOP_EXT, 0, 1);	R, 1, 1);
<pre>pattern = new ProductionPattern((int )</pre>	<pre>alt.AddProduction((int )</pre>	<pre>pattern.AddAlternative (alt);</pre>	<pre>pattern.AddAlternative (alt);</pre>
SyntaxConstants.PROD_0 PERAND,	SyntaxConstants.PROD_0 PERAND, 1, 1);	AddPattern(pattern);	<pre>ProductionPatternAlter native();</pre>
"Prod_operand"); alt = new	<pre>alt.AddProduction((int )</pre>	<pre>pattern = new ProductionPattern((int</pre>	alt.AddToken((int) SyntaxConstants.LESS,
<pre>ProductionPatternAlter native();</pre>	SyntaxConstants.PROD_S _MATH_EXT, 0, 1);	) SyntaxConstants.PROD_R ELOP EXT,	1, 1); pattern.AddAlternative
<pre>alt.AddProduction((int )</pre>	<pre>pattern.AddAlternative (alt);</pre>	<pre>"Prod_RelopExt");</pre>	(alt); alt = new
SyntaxConstants.PROD_R ETURN, 1, 1);	AddPattern(pattern);	<pre>alt = new ProductionPatternAlter native();</pre>	<pre>ProductionPatternAlter native();</pre>
<pre>pattern.AddAlternative (alt);</pre>	<pre>pattern = new ProductionPattern((int )</pre>	alt.AddProduction((int	<pre>alt.AddToken((int) SyntaxConstants.GREATE R E, 1, 1);</pre>
AddPattern(pattern);	SyntaxConstants.PROD_O PER_COND_EXT,	SyntaxConstants.PROD_0 P1, 1, 1);	N_L, 1, 1/,

		SyntaxConstants.OR, 1,	
<pre>pattern.AddAlternative (alt);</pre>	alt.AddToken((int) SyntaxConstants.O PARE	1);	public class SyntaxTokenizer :
alt = new	N, 1, 1);	pattern.AddAlternative	Tokenizer {
ProductionPatternAlter		(alt);	
<pre>native();</pre>	alt.AddProduction((int	alt = new ProductionPatternAlter	public
alt.AddToken((int)	SyntaxConstants.PROD_R	native();	SyntaxTokenizer(TextRe ader input)
SyntaxConstants. LESS_E	EL_OP, 1, 1);		: base(input,
, 1, 1);		alt.AddToken((int)	false) {
	alt.AddToken((int)	SyntaxConstants. AND,	
<pre>pattern.AddAlternative (alt);</pre>	SyntaxConstants.C_PARE N, 1, 1);	1, 1);	<pre>CreatePatterns();</pre>
alt = new	N, 1, 1),	pattern.AddAlternative	creater atterns (),
ProductionPatternAlter	alt.AddProduction((int	(alt);	,
<pre>native();</pre>	)		/**
1. 4117 1 (/: .)	SyntaxConstants. PROD_E	AddPattern(pattern);	*
alt.AddToken((int) SyntaxConstants.EQUALS	XT_LOG_OP, 0, 1);	pattern = new	<summary>Initializes the tokenizer by</summary>
, 1, 1);	pattern.AddAlternative	ProductionPattern((int	creating all the token
	(alt);	)	*
alt.AddToken((int)		SyntaxConstants.PROD_E	patterns.
SyntaxConstants. EQUALS	AddPattern(pattern);	ND,	*
, 1, 1);	pattern = new	"Prod end");	* <exception cref='ParserCreationEx</exception 
pattern.AddAlternative	ProductionPattern((int	alt = new	ception'>if the
(alt);	)	${\tt ProductionPatternAlter}$	tokenizer
alt = new	SyntaxConstants.PROD_E	native();	* couldn't be
ProductionPatternAlter native();	XT_LOG_OP,	alt.AddToken((int)	initialized correctly
native (),	"Prod_ExtLogOp");	SyntaxConstants. C BRAC	*/
alt.AddProduction((int	alt = new	KET, 1, 1);	private void
)	ProductionPatternAlter		CreatePatterns() {
SyntaxConstants.PROD_0 PER EQ, 1, 1);	native();	alt.AddToken((int) SyntaxConstants.GETCH,	TokenPattern
1 EN_EQ, 1, 1),	alt.AddProduction((int	1, 1);	pattern;
pattern.AddAlternative	)	, , ,	pattern = new
(alt);	SyntaxConstants.PROD_L	alt.AddToken((int)	TokenPattern((int)
alt = new	OG_OPER, 1, 1);	SyntaxConstants. O_PARE	SyntaxConstants.MAIN_N
<pre>ProductionPatternAlter native();</pre>	alt.AddProduction((int	N, 1, 1);	,
nacive (),	)	alt.AddToken((int)	"MAIN_N",
alt.AddToken((int)	SyntaxConstants.PROD_L	SyntaxConstants.C_PARE	
SyntaxConstants.MODULU	OG_OP, 1, 1);	N, 1, 1);	TokenPattern. PatternTy
S, 1, 1);	pattern.AddAlternative	alt.AddToken((int)	pe. STRING,
pattern.AddAlternative	(alt);	SyntaxConstants. SEMIC,	"PrimaryMission");
(alt);		1, 1);	
(110	AddPattern(pattern);		AddPattern(pattern);
AddPattern(pattern);	pottom = nov	pattern.AddAlternative (alt);	nottonn = now
pattern = new	pattern = new ProductionPattern((int	(a1t),	<pre>pattern = new TokenPattern((int)</pre>
ProductionPattern((int	)	AddPattern(pattern);	SyntaxConstants. PRINT_
)	${\tt SyntaxConstants.PROD\_L}$	}	N,
SyntaxConstants.PROD_L	OG_OPER,	}	"DDING N"
0G_0P,	"Prod LogOper");	Syntax Analyzer:	"PRINT_N",
"Prod_LogOp");	alt = new	Syntax Analyzer. SyntaxTokenizer.cs	TokenPattern.PatternTy
alt = new	ProductionPatternAlter	-	pe. STRING,
ProductionPatternAlter	<pre>native();</pre>	using System.IO;	" "
native();	alt.AddToken((int)	using Core.Library;	"post");
	art. Audrokell (Tilt)	using core. Library,	

	SyntaxConstants.CASE N		
AddPattern(pattern);	,	AddPattern(pattern);	TokenPattern.PatternTy pe.STRING,
<pre>pattern = new TokenPattern((int)</pre>	"CASE_N",	<pre>pattern = new TokenPattern((int)</pre>	"deploy");
SyntaxConstants.SCAN_N	TokenPattern.PatternTy pe.STRING,	SyntaxConstants.ELSE_N	AddPattern(pattern);
"SCAN_N",	"operation");	"ELSE_N",	pattern = new TokenPattern((int)
TokenPattern.PatternTy pe.STRING,	AddPattern(pattern);	TokenPattern.PatternTy pe.STRING,	SyntaxConstants. STRUCT _N,
"capture");	pattern = new TokenPattern((int) SyntaxConstants.BREAK,	"order");	"STRUCT_N",
AddPattern(pattern);	"BREAK",	AddPattern(pattern);	TokenPattern.PatternTy pe.STRING,
pattern = new		pattern = new	
TokenPattern((int) SyntaxConstants.CONST_	TokenPattern.PatternTy pe.STRING,	TokenPattern((int) SyntaxConstants.DO,	"struct");
N,	<i>"</i> - <i>"</i>	"70"	AddPattern(pattern);
"CONST N",	"abort");	"D0",	pattern = new
CONST_N ,	AddPattern(pattern);	TokenPattern.PatternTy	TokenPattern((int)
TokenPattern.PatternTy	madi accord (paccord)	pe. STRING,	SyntaxConstants. DEFAUL
pe. STRING,	pattern = new		Τ,
// 1.7/\)	TokenPattern((int)	"go");	"DDD 111 0"
"hold");	SyntaxConstants.FOR_N,	AddDattarn (nattarn)	"DEFAULT",
AddPattern(pattern);	"FOR_N",	AddPattern(pattern);	TokenPattern.PatternTy
nadi de com (particin),	i on_iv ,	pattern = new	pe. STRING,
pattern = new	TokenPattern.PatternTy	TokenPattern((int)	•
TokenPattern((int)	pe. STRING,	${\tt SyntaxConstants.WHILE}\_$	"action");
SyntaxConstants.RETURN	<b>".</b> "\	Ν,	
,	"inquire");	"WHILE N",	AddPattern(pattern);
"RETURN",	AddPattern(pattern);	WHILE_N ,	pattern = new
RETORIV,	ndar at term (pattern),	TokenPattern.PatternTy	TokenPattern((int)
TokenPattern.PatternTy	pattern = new	pe. STRING,	SyntaxConstants.CLEAR,
pe. STRING,	TokenPattern((int)		
	SyntaxConstants. IF,	"phase");	"CLEAR",
"backup");	"TP"	AllDari	T. 1. D D T.
AddPattern(pattern);	"IF",	AddPattern(pattern);	TokenPattern.PatternTy pe.STRING,
, , , , , , , , , , , , , , , , , , ,	TokenPattern.PatternTy	pattern = new	F,
pattern = new	pe.STRING,	TokenPattern((int)	"commence");
TokenPattern((int)		SyntaxConstants. VOID,	
SyntaxConstants.SWITCH	"inorder");	"NOTP"	AddPattern(pattern);
_N,	AddPattern(pattern);	"VOID",	pattern = new
"SWITCH_N",	Addi attern (pattern),	TokenPattern.PatternTy	TokenPattern((int)
Switch_iv,	pattern = new	pe. STRING,	SyntaxConstants. SQROOT
TokenPattern.PatternTy	TokenPattern((int)	,	,
pe. STRING,	SyntaxConstants.ELSEIF	"miss");	
" . ")	_N,	11ID	"SQROOT",
"campaign");	"DISCIE N"	AddPattern(pattern);	TokonDottom Dottom
AddPattern(pattern);	"ELSEIF_N",	pattern = new	TokenPattern.PatternTy pe.STRING,
naar accorn (pactern),	TokenPattern.PatternTy	TokenPattern((int)	po. otkino,
<pre>pattern = new TokenPattern((int)</pre>	pe. STRING,	SyntaxConstants. GETCH,	"sqrt");
\\/	"otherorder");	"GETCH",	AddPattern(pattern);

		pattern = new	
<pre>pattern = new TokenPattern((int)</pre>	TokenPattern.PatternTy pe.STRING,	TokenPattern((int) SyntaxConstants.AND,	″″);
SyntaxConstants.PLUS,	″%″);	"AND",	AddPattern(pattern);
"PLUS",	•	,	pattern = new
TokenPattern.PatternTy	AddPattern(pattern);	TokenPattern.PatternTy pe.STRING,	TokenPattern((int) SyntaxConstants.P_E,
pe. STRING,	pattern = new TokenPattern((int)	″&″);	"P_E",
"+");	SyntaxConstants.EQUALS	AddPattern(pattern);	TokenPattern.PatternTy
AddPattern(pattern);	"FOULL C"	-	pe.STRING,
pottown = pow	"EQUALS",	pattern = new TokenPattern((int)	"+=");
<pre>pattern = new TokenPattern((int)</pre>	TokenPattern.PatternTy	SyntaxConstants.OR,	<del></del> ),
SyntaxConstants. MINUS,	pe. STRING,		AddPattern(pattern);
"""""""""""""""""""""""""""""""""""""""	" "	"OR",	
"MINUS",	″=″) ;	m.1 . D	pattern = new
TokenPattern.PatternTy	AddPattern(pattern);	TokenPattern.PatternTy pe.STRING,	TokenPattern((int) SyntaxConstants.M_E,
	pattern = new	"  ");	"M_E",
″-″);	TokenPattern((int) SyntaxConstants.SEMIC,	AddPattern(pattern);	TokenPattern.PatternTy
AddPattern(pattern);	"SEMIC",	pattern = new	pe.STRING,
pattern = new	GENITO ,	TokenPattern((int)	″-=″) :
TokenPattern((int)	TokenPattern.PatternTy	SyntaxConstants. NOT,	, ,
SyntaxConstants. TIMES,	pe. STRING,	"NOT",	AddPattern(pattern);
"TIMES",	";");	NOI,	pattern = new
TokenPattern. PatternTy	AddPattern(pattern);	TokenPattern. PatternTy pe. STRING,	TokenPattern((int) SyntaxConstants.T E,
pe. STRING,	-		
″*″) ;	pattern = new TokenPattern((int)	"!");	"T_E",
AddPattern(pattern);	SyntaxConstants.DOT,	AddPattern (pattern);	TokenPattern.PatternTy pe.STRING,
pattern = new	"DOT",	<pre>pattern = new TokenPattern((int)</pre>	″*=″) ;
TokenPattern((int)	TokenPattern.PatternTy	SyntaxConstants. INCREM	~- ),
SyntaxConstants. DIVIDE	pe. STRING,	ENT,	AddPattern(pattern);
,	". ") ;	"INCREMENT",	pattern = new
"DIVIDE",	AddPattern(pattern);	TokenPattern.PatternTy	TokenPattern((int) SyntaxConstants.D E,
TokenPattern.PatternTy	-	pe. STRING,	
pe. STRING,	pattern = new TokenPattern((int)	"++");	″D_E″,
"/");	SyntaxConstants.COMMA,	AddPattern(pattern);	TokenPattern.PatternTy pe.STRING,
AddPattern(pattern);	"COMMA",	pattern = new	"/=");
pattern = new	TokenPattern.PatternTy	TokenPattern((int)	, , ,
TokenPattern((int) SyntaxConstants.MODULU	pe. STRING,	SyntaxConstants.DECREM ENT,	AddPattern(pattern);
S,	", ") ;	,	pattern = new
"MODULUS",	AddPattern(pattern);	"DECREMENT",	TokenPattern((int) SyntaxConstants.MOD_E,
		TokenPattern.PatternTy pe.STRING,	"Mod_E",

TokenPattern.PatternTy	<pre>pattern = new TokenPattern((int)</pre>	TokenPattern.PatternTy	<pre>pattern = new TokenPattern((int)</pre>
″%=″);	SyntaxConstants.D_QUOT E,	"<");	SyntaxConstants.S_CBRACKET,
AddPattern(pattern);	"D_QUOTE",	AddPattern(pattern);	"S_CBRACKET",
<pre>pattern = new TokenPattern((int) SyntaxConstants.NEWLIN</pre>	TokenPattern.PatternTy pe.REGEXP,	<pre>pattern = new TokenPattern((int) SyntaxConstants.LESS,</pre>	TokenPattern.PatternTy pe.STRING,
E,	"[\"]");	"LESS",	"]");
"NEWLINE",	AddPattern(pattern);	TokenPattern.PatternTy	AddPattern(pattern);
TokenPattern.PatternTy pe.STRING,	<pre>pattern = new TokenPattern((int) SyntaxConstants.COLON,</pre>	pe. STRING, ">");	pattern = new TokenPattern((int) SyntaxConstants.DOLLAR
"\\n");	"COLON",	AddPattern(pattern);	,
AddPattern(pattern);	TokenPattern.PatternTy	pattern = new	"DOLLAR",
<pre>pattern = new TokenPattern((int) SyntaxConstants.N_E,</pre>	pe. STRING, ":");	TokenPattern((int) SyntaxConstants.GREATE R_E,	TokenPattern.PatternTy pe.STRING,
"N_E",	AddPattern(pattern);	"GREATER_E",	"\$"); AddPattern(pattern);
TokenPattern.PatternTy pe.STRING,	pattern = new TokenPattern((int)	TokenPattern.PatternTy pe.STRING,	pattern = new
"!=");	SyntaxConstants.O_BRAC KET,	"<=");	TokenPattern((int) SyntaxConstants.POWER,
AddPattern(pattern);	"O_BRACKET",	AddPattern(pattern);	"POWER",
<pre>pattern = new TokenPattern((int) SyntaxConstants.0 PARE</pre>	TokenPattern.PatternTy pe.STRING,	pattern = new TokenPattern((int) SyntaxConstants.LESS_E	TokenPattern.PatternTy pe.STRING,
N,	"{");	,	"^");
"O_PAREN",	AddPattern(pattern);	"LESS_E",	AddPattern(pattern);
TokenPattern.PatternTy pe.STRING,	pattern = new TokenPattern((int) SyntaxConstants.C_BRAC	TokenPattern.PatternTy pe. STRING,	pattern = new TokenPattern((int) SyntaxConstants.HASH,
"(");	KET,	″>=″) ;	"HASH",
AddPattern(pattern);	"C_BRACKET",	AddPattern(pattern);	TokenPattern.PatternTy
pattern = new TokenPattern((int) SyntaxConstants.C PARE	TokenPattern.PatternTy pe.STRING,	pattern = new TokenPattern((int) SyntaxConstants.S OBRA	pe.STRING, "#");
N,	<i>"</i> }");	CKET,	AddPattern(pattern);
"C_PAREN",	AddPattern(pattern);	"S_OBRACKET",	pattern = new
TokenPattern. PatternTy pe. STRING,	pattern = new TokenPattern((int) SyntaxConstants.GREATE	TokenPattern.PatternTy pe.STRING,	TokenPattern((int) SyntaxConstants.NEGA,
")");	R,	"[");	"NEGA",

AddPattern(pattern);

AddPattern(pattern);

"GREATER",

	pattern = new		SyntaxConstants.FUNCTN
TokenPattern.PatternTy pe.STRING,	TokenPattern((int) SyntaxConstants.BOOL_N	TokenPattern.PatternTy pe.STRING,	AME,
″~″) ;	,	"Charlit");	"FUNCTNAME",
),	"BOOL_N",	Charift ),	TokenPattern.PatternTy
AddPattern(pattern);	TokenPattern. PatternTy	AddPattern(pattern);	pe. STRING,
<pre>pattern = new TokenPattern((int)</pre>	pe. STRING,	<pre>pattern = new TokenPattern((int)</pre>	"functName");
SyntaxConstants. INT,	"response");	SyntaxConstants. TEXT,	AddPattern(pattern);
"INT",	AddPattern(pattern);	"TEXT",	pattern = new TokenPattern((int)
TokenPattern.PatternTy	pattern = new	TokenPattern.PatternTy	SyntaxConstants. STRUCT
pe. STRING,	TokenPattern((int) SyntaxConstants. ID,	pe. STRING,	NAME,
"unit");	Syntaxoons tantes. 12,	"Stringlit");	"STRUCTNAME",
	"ID",	-	
AddPattern(pattern);	TokenPattern.PatternTy	AddPattern(pattern);	TokenPattern.PatternTy
pattern = new	pe. STRING,	pattern = new	P,
TokenPattern((int)	pot sixino,	TokenPattern((int)	"structname");
SyntaxConstants.CHAR,	"id");	SyntaxConstants.COM,	•
- ,	/,	- ,,	AddPattern(pattern);
"CHAR",	AddPattern(pattern);	"COM",	, , , , , , , , , , , , , , , , , , ,
,	4	,	pattern = new
TokenPattern.PatternTy	pattern = new	TokenPattern.PatternTy	TokenPattern((int)
pe. STRING,	TokenPattern((int)	pe. STRING,	SyntaxConstants. IDSTRU
F	SyntaxConstants. NUM,	F,	CT,
"joe");	-,,	"comment");	,
<u> </u>	"NUM",	, ,	"IDSTRUCT",
AddPattern(pattern);		AddPattern(pattern);	
-	TokenPattern.PatternTy	-	TokenPattern.PatternTy
pattern = new	pe. STRING,	pattern = new	pe. STRING,
TokenPattern((int)		TokenPattern((int)	
SyntaxConstants.FLOAT,	"Numlit");	SyntaxConstants.YES,	"idStruct");
"FLOAT",	AddPattern(pattern);	"YES",	AddPattern(pattern);
	- -		-
TokenPattern. PatternTy	pattern = new	TokenPattern. PatternTy	pattern = new
pe. STRING,	TokenPattern((int)	pe. STRING,	TokenPattern((int)
# * · · · # \	SyntaxConstants.DECIMA	" · · · · · · · · · · · · · · · ·	SyntaxConstants.F,
"digit");	L,	"AFFIRMATIVE");	""
A 1 1D	"DECIMAL"	A 1 ID. 44 ( 44 ) .	"F",
AddPattern(pattern);	"DECIMAL",	AddPattern(pattern);	T-1D-++ D-++T
nottonn = now	TokonPottorn PottornT-	nottown = now	TokenPattern. PatternTy
pattern = new	TokenPattern. PatternTy	pattern = new	pe. STRING,
TokenPattern((int) SyntaxConstants.STRING	pe. STRING,	TokenPattern((int) SyntaxConstants.NO,	"f");
SyntaxConstants. SIRING	"Declit");	Syntaxconstants. No,	1 );
,	Deciit );	"NO",	AddPattern(pattern);
"STRING",	AddPattern(pattern);	NO ,	Addi attern (pattern),
orkino ,	nadi determ (parterm),	TokenPattern.PatternTy	pattern = new
TokenPattern.PatternTy	pattern = new	pe. STRING,	TokenPattern((int)
pe. STRING,	TokenPattern((int)	por outling,	SyntaxConstants. D,
F 0. 0.11.1.0)	SyntaxConstants. S CHAR	"NEGATIVE");	Symposius van vos. D,
"company");		nonii , ,	″D″,
company,	,	AddPattern(pattern);	ν,
AddPattern(pattern);	"S_CHAR",	naar accorn (paccorn),	TokenPattern.PatternTy
partolii,	~	pattern = new	pe. STRING,
		TokenPattern((int)	
		/	

	nottorn = now		* various child()
″d″);	pattern = new TokenPattern((int) SyntaxConstants.WHITES	"REVERSE",	calls are made, they will be made from left
AddPattern(pattern);	PACE,	TokenPattern.PatternTy pe.STRING,	to  * right as child
pattern = new	"WHITESPACE",	pe. orkino,	nodes are added (to
TokenPattern((int)	,	"Swap");	the right).
SyntaxConstants.S,	TokenPattern.PatternTy		*/
"0"	pe. REGEXP,	AddPattern (pattern);	public class
"S",	"[ \\t\\n\\r]+");	}	Analyzer {
TokenPattern.PatternTy	pattern. Ignore	,	/**
pe. STRING,	= true;	Core. Library	* Creates a
			new parse tree
"s");	AddPattern(pattern);		analyzer.
AddPattern(pattern);	pattern = new	using System.Collections;	*/ public
nadi attern (pattern),	TokenPattern((int)	System. Corrections,	Analyzer() {
pattern = new	SyntaxConstants. TOCHAR	namespace Core.Library	}
TokenPattern((int)	,	{	
SyntaxConstants.ZERO,	"TOOLAD"	/	/**
"ZERO",	"TOCHAR",	/** * A parse tree	* Resets this analyzer when the
ZERO ,	TokenPattern.PatternTy	analyzer. This class	parser is reset for
TokenPattern.PatternTy	pe. STRING,	provides callback	another
pe. STRING,		methods that	* input
"Zero");	"ToJoeRange");	* may be used	stream. The default
zero );	AddPattern(pattern);	either during parsing, or for a parse tree	implementation of this method does
AddPattern(pattern);	nadi di terii (patterii),	traversal.	* nothing.
-	pattern = new	* This class	*
pattern = new	TokenPattern((int)	should be subclassed	*
TokenPattern((int) SyntaxConstants.SPACE,	SyntaxConstants.LENGTH	to provide adequate handling of the	*/
Syntaxconstants. SPACE,	F,	* parse tree	<pre>public virtual void Reset() {</pre>
"SPACE",	"LENGTHF",	nodes.	// Default
		*	implementation does
TokenPattern. PatternTy	TokenPattern. PatternTy	* The general	nothing
pe. STRING,	pe. STRING,	contract for the analyzer class does	}
" ");	"Extent");	not guarantee a	/**
pattern. Ignore	, ,	* strict call	* Analyzes a
= true;	AddPattern(pattern);	order for the callback	parse tree node by
AddDa++ar-(++)		methods. Depending on	traversing all it's
AddPattern(pattern);	pattern = new TokenPattern((int)	the type * of parser, the	child nodes. * The tree
pattern = new	SyntaxConstants. CONTAI	enter() and exit()	traversal is depth-
TokenPattern((int)	NS,	methods for production	first, and the
SyntaxConstants.N_LINE	"CONTATNO"	nodes can	appropriate
,	"CONTAINS",	* be called either in a top-down	* callback methods will be
"N LINE",	TokenPattern.PatternTy	or a bottom-up	called. If the node is
·,	pe. STRING,	fashion. The only	a production
TokenPattern.PatternTy		* guarantee	* node, a new
pe. STRING,	"Carry");	provided by this API,	production node will
"\\n");	AddPattern(pattern);	is that the calls for any given	be created and children will
pattern. Ignore	naai attern (pattern),	* node will	* be added by
= true;	pattern = new	always be in the order	recursively processing
	TokenPattern((int)	enter(), child(), and	the children of the
AddPattern(pattern);	SyntaxConstants. REVERS	exit(). If	* specified
	Е,		production node. This

.1.1.			
method is used to	method is used to	res = Exit(prod);	* * @return the
process a * parse tree	process a * parse tree	res - Exit (prod),	new production node
after creation.	after creation.	return res;	new production node *
*	*	}	*/
* @param node	* @param node	catch (ParseException	public virtual
the parse tree node to	the parse tree node to	e) {	Production
process	process	if	NewProduction(Producti
*	* @param log	(errorCount ==	onPattern pattern) {
* @return the	the parser error log	log.Count) {	return new
resulting parse tree	*		Production(pattern);
node	* @return the	log. AddError(e);	}
*	resulting parse tree	}	
* @throws	node	}	/**
ParserLogException if	*/	}	* Called when
the node analysis	public virtual	else {	entering a parse tree
discovered	Node Analyze(Node		node. By default this
*	node,	node.Values.Clear();	method
errors	ParserLogException	try {	* does
*/	log) {		nothing. A subclass
public virtual	Production	Enter (node);	can override this
Node Analyze(Node	prod;	}	method to handle
node) {	int	catch (ParseException	* each node
	errorCount;	e) {	separately.
ParserLogException	N 1	1 4115 ()	*
log = new	Node res =	log. AddError(e);	* @param node
<pre>ParserLogException();</pre>	null;	}	the node being entered *
nodo -	errorCount	try {	•
<pre>node = Analyze(node, log);</pre>	= log.Count; if (node	res = Exit(node);	* @throws
if	is Production) {	ies - Exit (node),	ParseException if the node analysis
(log. Count > 0) {	prod =	return res;	discovered errors
throw	(Production) node;	}	*/
log;	prod =	catch (ParseException	public virtual
}	NewProduction(prod.Pat	e) {	void Enter (Node node)
return	tern);	if	{
node;	try {	(errorCount ==	}
}	•	log. Count) {	
	Enter(prod);		/**
/**	}	log. AddError(e);	* Called when
* Analyzes a	catch (ParseException	}	exiting a parse tree
parse tree node by	e) {	}	node. By default this
traversing all it's		}	method
child nodes.	log.AddError(e);	return	* returns the
* The tree	}	null;	node. A subclass can
traversal is depth-	for	}	override this method
first, and the	(int i = 0; i <	,	to handle
appropriate	node.Count; i++) {	/**	* each node
* callback		* Factory	separately. If no
methods will be	try {	method to create a new	parse tree should be
called. If the node is	01:11/	production node. This	created, this
a production	Child(prod,	method	* method
* node, a new	Analyze(node[i],	* can be	should return null.
production node will	log));	overridden to provide	* * @param node
be created and children will	catch (ParseException	other production implementations	the node being exited
* be added by	e) {	* than the	the hode being exited
recursively processing	C/ (	default one.	* @return the
the children of the	log.AddError(e);	*	node to add to the
* specified	10g. naubi101 (c) ,	* @param	parse tree, or
production node. This	}	pattern the	r
1	try {	production pattern	
	, ,	*	

		1	
*	* exception	}	
null if no parse tree	with the internal	/ state	return child;
should be created *	error type. *	/** * Returns the	}
* @throws	•	* Returns the	} *h*
	* @param node		throw new
ParseException if the	the parent node	specified id. If the	ParseException(
node analysis discovered errors	* @param pos the child position	node is * null, or no	ParseException.ErrorTy
*/	the chird position *	child with the	pe. INTERNAL,
public virtual	* @return the	specified id could be	"node
Node Exit(Node node) {	child node	found, this	" + node. Name + "
return	*	* method will	has no child with id "
node;	* @throws	throw a parse	+ id,
Noue,	ParseException if	exception with the	Tu,
J	either the node or the	internal error	node.StartLine,
/**	child node	* type.	node, StartErne,
* Called when	*	* type:	node.StartColumn);
adding a child to a	was null	* @param node	}
parse tree node. By	*/	the parent node	J
default	protected Node	* @param id	/**
* this method	GetChildAt (Node node,	the child node id	* Returns the
adds the child to the	int pos) {	*	node value at the
production node. A	Node	* @return the	specified position. If
subclass	child;	child node	either
* can	oniia,	*	* the node or
override this method	if (node	* @throws	the value is null,
to handle each node	== null) {	ParseException if the	this method will throw
separately. Note	throw	node was null, or a	a parse
* that the	new ParseException(	child node	* exception
child node may be null		*	with the internal
if the corresponding	ParseException. ErrorTy	couldn't be found	error type.
exit()	pe. INTERNAL,	*/	*
* method	-	protected Node	* @param node
returned null.	"attempt to read	GetChildWithId(Node	the parse tree node
*	'null' parse tree	node, int id) {	* @param pos
* @param node	node",	Node	the child position
the parent node	_	child;	*
* @param	1,		* @return the
child the	_	if (node	value object
child node, or null	1);	== nu11) {	*
*	}	throw	* @throws
* @throws	child =	new ParseException(	ParseException if
ParseException if the	node[pos];		either the node or the
node analysis	if (child	ParseException.ErrorTy	value was null
discovered errors	== null) {	pe. INTERNAL,	*/
*/	throw		protected
public virtual	new ParseException(	"attempt to read	object GetValue(Node
void Child(Production		'null' parse tree	node, int pos) {
node, Node child) {	ParseException.ErrorTy	node",	object
	pe. INTERNAL,	_	value;
node.AddChild(child);		1,	
}	"node'" + node.Name +	_	if (node
,	$^{\prime\prime\prime}$ has no child at $^{\prime\prime}$ +	1);	== null) {
/**		}	throw
* Returns a	"position" + pos,	for (int i	new ParseException(
child at the specified		= 0; i < node.Count;	
position. If either	node.StartLine,	i++) {	ParseException. ErrorTy
the node	1 0 2 3	child	pe. INTERNAL,
* or the	node.StartColumn);	= node[i];	"
child node is null,	}	if	"attempt to read
this method will throw	return	(child != null &&	'null' parse tree
a parse	child;	child.Id == id) {	node",

_	protected int	protected	values
1,	GetIntValue(Node node,	string	= child. Values;
_	int pos) {	GetStringValue(Node	if
1);	object	node, int pos) {	(values != null) {
}	value;	object	(varues : narr) (
value =	varae,	value;	result. AddRange(values
node. Values[pos];	value =	varae,	);
if (value	GetValue(node, pos);	value =	}
== null) {	if (value	GetValue(node, pos);	}
throw	is int) {	if (value	return
new ParseException(	return	is string) {	result;
•	(int) value;	return	}
ParseException.ErrorTy	} else {	(string) value;	}
pe. INTERNAL,	throw	} else {	}
	new ParseException(	throw	
"node '" + node.Name +		new ParseException(	/*
"' has no value at " +	ParseException.ErrorTy		* LookAheadSet.cs
	pe. INTERNAL,	ParseException.ErrorTy	*/
"position" + pos,		pe. INTERNAL,	
	"node'" + node.Name +		using
node.StartLine,	"' has no integer	"node '" + node.Name +	System. Collections;
	value " +	"' has no string value	using System. Text;
node.StartColumn);		<b>"</b> +	
}	"at position" + pos,		namespace Core.Library
return		"at position" + pos,	{
value;	node.StartLine,		,
}		node.StartLine,	/**
	node.StartColumn);	1 0	* A token look-
/**	}	node.StartColumn);	ahead set. This class
* Returns the	}	}	contains a set of
node integer value at		}	token id
	/.tt.		A11
the specified	/**	/state	* sequences. All
the specified position. If	* Returns the	/**	sequences in the set
the specified position. If * either the	* Returns the node string value at	* Returns all	sequences in the set are limited in length,
the specified position. If     * either the node is null, or the	* Returns the node string value at the specified	* Returns all the node values for	sequences in the set are limited in length, so
the specified position. If     * either the node is null, or the value is not an	* Returns the node string value at the specified position. If	* Returns all the node values for all child nodes.	sequences in the set are limited in length, so * that no single
the specified position. If     * either the node is null, or the value is not an instance of	* Returns the node string value at the specified position. If * either the	* Returns all the node values for all child nodes. *	sequences in the set are limited in length, so * that no single sequence is longer
the specified position. If     * either the node is null, or the value is not an instance of     * the Integer	* Returns the node string value at the specified position. If	* Returns all the node values for all child nodes.  *  * @param node	sequences in the set are limited in length, so  * that no single sequence is longer than a maximum value.
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method	* Returns the node string value at the specified position. If	* Returns all the node values for all child nodes. *	sequences in the set are limited in length, so  * that no single sequence is longer than a maximum value. This
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse	* Returns the node string value at the specified position. If	* Returns all the node values for all child nodes.  *  * @param node the parse tree node *	sequences in the set are limited in length, so  * that no single sequence is longer than a maximum value.
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String	* Returns all the node values for all child nodes.  * @param node the parse tree node	sequences in the set are limited in length, so         * that no single sequence is longer than a maximum value. This         * class also filters out
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String class, this method	* Returns all the node values for all child nodes.  * * @param node the parse tree node  * * @return a list with all the	sequences in the set are limited in length, so     * that no single sequence is longer than a maximum value. This     * class also filters out duplicates. Each token
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String	* Returns all the node values for all child nodes.  * * @param node the parse tree node  * * @return a	sequences in the set are limited in length, so     * that no single sequence is longer than a maximum value. This     * class also filters out
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String class, this method will throw a parse	* Returns all the node values for all child nodes.  * * @param node the parse tree node  * * @return a list with all the child node values	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String class, this method will throw a parse exception	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String class, this method will throw a parse exception     * with the	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String class, this method will throw a parse exception     * with the internal error type.	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */ protected	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos	* Returns the node string value at the specified position. If	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */ protected ArrayList	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position	* Returns the node string value at the specified position. If     * either the node is null, or the value is not an instance of     * the String class, this method will throw a parse exception     * with the internal error type.     *     * @param node	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */ protected ArrayList GetChildValues(Node node) {  ArrayList	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *	* Returns the node string value at the specified position. If	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */ protected ArrayList GetChildValues(Node node) {  ArrayList result = new	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *  * @return the	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node	sequences in the set are limited in length, so
the specified position. If	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;	sequences in the set are limited in length, so
the specified position. If	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;  ArrayList	sequences in the set are limited in length, so
the specified position. If	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;	sequences in the set are limited in length, so
the specified position. If	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws ParseException if	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;  ArrayList values;	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws ParseException if either the node was null, or the  * value wasn't an	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws ParseException if either the node was	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;  ArrayList values;	sequences in the set are limited in length, so
the specified position. If	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws ParseException if either the node was null, or the	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;  ArrayList values;  for (int i = 0; i < node.Count;	sequences in the set are limited in length, so
the specified position. If  * either the node is null, or the value is not an instance of  * the Integer class, this method will throw a parse exception  * with the internal error type.  *  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws ParseException if either the node was null, or the  * value wasn't an	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws ParseException if either the node was null, or the  *	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;  ArrayList values;  for (int i = 0; i < node.Count; i++) {	sequences in the set are limited in length, so  * that no single sequence is longer than a maximum value. This  * class also filters out duplicates. Each token sequence also  * contains a repeat flag, allowing the look-ahead set to contain  * information about possible infinite repetitions of certain  * sequences. That information is important when conflicts arise  * between two look-ahead sets, as such a conflict cannot be  * resolved if the
the specified position. If	* Returns the node string value at the specified position. If  * either the node is null, or the value is not an instance of  * the String class, this method will throw a parse exception  * with the internal error type.  * @param node the parse tree node  * @param pos the child position  *  * @return the value object  *  * @throws ParseException if either the node was null, or the	* Returns all the node values for all child nodes.  *  * @param node the parse tree node  *  * @return a list with all the child node values  *  */  protected ArrayList GetChildValues(Node node) {  ArrayList result = new ArrayList();  Node child;  ArrayList values;  for (int i = 0; i < node.Count;	sequences in the set are limited in length, so

* cause infinite	maximum token sequence		ArrayList
loop).	length	min = seq. Length();	list = new
*	* @param set	}	ArrayList();
	the look-ahead set to	}	int[]
* @version 1.1	сору	return	result;
*/	*/	$(\min < 0) ? 0 : \min;$	object
internal class	public	}	token;
LookAheadSet {	LookAheadSet(int	,	int
	maxLength,	/**	i;
/**	LookAheadSet set)	* Returns the	
* The set of	:	length of the longest	for (i =
token look-ahead	this(maxLength) {	token sequence in this	0; i < elements.Count;
sequences. Each		* set. This	i++) {
sequence in		method will return	token
* turn is	AddAll(set);	zero (0) if the set is	= ((Sequence)
represented by an	}	empty.	elements[i]).GetToken(
ArrayList with		*	0);
Integers for the	/**	* @return the	if
* token id:s.	* Returns the	length of the longest	(token != null &&
*/	size of this look-	token sequence	!list.Contains(token))
private	ahead set.	*/	{
ArrayList elements =	*	public int	
new ArrayList();	* @return the	GetMaxLength() {	list.Add(token);
	number of token	Sequence	}
/**	sequences in the set	seq;	}
* The maximum	*/	int	result =
length of any look-	public int	$\max = 0;$	<pre>new int[list.Count];</pre>
ahead sequence.	Size() {		for $(i =$
*/	return	for (int i	0; i < list.Count;
private int	elements.Count;	= 0; i <	i++) {
maxLength;	}	elements.Count; i++) {	
maxLength;	}	elements.Count; i++) {     seq =	result[i] = (int)
maxLength; /**	} /**		result[i] = (int) list[i];
	} /** * Returns the	seq =	
/**	,	seq = (Sequence)	
/** * Creates a	* Returns the	<pre>seq = (Sequence) elements[i];</pre>	list[i]; }
/** * Creates a new look-ahead set	* Returns the length of the shortest	<pre>seq = (Sequence) elements[i]; if</pre>	list[i];
/**  * Creates a  new look-ahead set  with the specified	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i]; if</pre>	list[i];
/**  * Creates a  new look-ahead set  with the specified  maximum	* Returns the length of the shortest token sequence in this * set. This	<pre>seq = (Sequence) elements[i];</pre>	list[i];
/**     * Creates a new look-ahead set with the specified maximum     * length.	* Returns the length of the shortest token sequence in this * set. This method will return	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**     * Creates a new look-ahead set with the specified maximum     * length.     *	* Returns the length of the shortest token sequence in this * set. This method will return zero (0) if the set is	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**     * Creates a new look-ahead set with the specified maximum     * length.     *     * @param	* Returns the length of the shortest token sequence in this * set. This method will return zero (0) if the set is empty.	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**     * Creates a new look-ahead set with the specified maximum     * length.     *     * @param maxLength the	* Returns the length of the shortest token sequence in this     * set. This method will return zero (0) if the set is empty.     *     * @return the	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     } } return</pre>	<pre>list[i];</pre>
/**     * Creates a new look-ahead set with the specified maximum     * length.     *     * @param maxLength the maximum token sequence	* Returns the length of the shortest token sequence in this * set. This method will return zero (0) if the set is empty.	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     } } return</pre>	<pre>list[i];</pre>
/**     * Creates a new look-ahead set with the specified maximum     * length.     *     * @param maxLength the maximum token sequence length	* Returns the length of the shortest token sequence in this     * set. This method will return zero (0) if the set is empty.     *     * @return the length of the shortest	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     } } return</pre>	<pre>list[i];</pre>
/**     * Creates a new look-ahead set with the specified maximum     * length.     *     * @param maxLength the maximum token sequence length     */	* Returns the length of the shortest token sequence in this     * set. This method will return zero (0) if the set is empty.     *     * @return the length of the shortest token sequence	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     }     return max; }</pre>	<pre>list[i];</pre>
/**     * Creates a new look-ahead set with the specified maximum     * length.     *     * @param maxLength the maximum token sequence length     */     public	* Returns the length of the shortest token sequence in this     * set. This method will return zero (0) if the set is empty.     *     * @return the length of the shortest token sequence     */	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  *  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int	* Returns the length of the shortest token sequence in this     * set. This method will return zero (0) if the set is empty.     *     * @return the length of the shortest token sequence     */     public int	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  *  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     }     return  max; }  /**     * Returns a list of the initial</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  *  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     }     return  max; }  /**     * Returns a list of the initial token id:s in this</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  *  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     }     return  max; }  /**     * Returns a list of the initial token id:s in this look-ahead     * set. The</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  *  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     }     return  max; }  /**     * Returns a list of the initial token id:s in this look-ahead     * set. The list returned will not</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  *  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  *  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];     if (seq.Length() &gt; max) {  max = seq.Length();     }     return  max; }  /**     * Returns a list of the initial token id:s in this look-ahead     * set. The list returned will not</pre>	list[i];
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**  * Creates a	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];</pre>	list[i];  return  result;  /**  * Checks if  this look-ahead set  contains a repetitive  token  * sequence.  *  * @return  true if at least one  token sequence is  repetitive, or  *  false otherwise  */  public bool  IsRepetitive() {
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**  * Creates a  duplicate look-ahead	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];</pre>	list[i];  return  result;  /**  * Checks if  this look-ahead set  contains a repetitive  token  * sequence.  *  * @return  true if at least one  token sequence is  repetitive, or  *  false otherwise  */  public bool  IsRepetitive() {  Sequence
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**  * Creates a  duplicate look-ahead  set, possibly with a	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];</pre>	list[i];  return  result;  /**  * Checks if  this look-ahead set  contains a repetitive  token  * sequence.  *  * @return  true if at least one  token sequence is  repetitive, or  *  false otherwise  */  public bool  IsRepetitive() {
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**  * Creates a  duplicate look-ahead  set, possibly with a  * different	* Returns the length of the shortest token sequence in this	seq = (Sequence) elements[i];  if (seq.Length() > max) {  max = seq.Length(); } return  max; }  /**  * Returns a list of the initial token id:s in this look-ahead  * set. The list returned will not contain any duplicates.  *  * @return a list of the inital token id:s in this	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**  * Creates a  duplicate look-ahead  set, possibly with a  * different  maximum length.	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  * @param  maxLength the  maximum token sequence  length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**  * Creates a  duplicate look-ahead  set, possibly with a  * different  maximum length.  *	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>
/**  * Creates a  new look-ahead set  with the specified  maximum  * length.  * @param  maxLength the  maximum token sequence length  */  public  LookAheadSet(int  maxLength) {  this.maxLength =  maxLength;  }  /**  * Creates a  duplicate look-ahead  set, possibly with a  * different  maximum length.	* Returns the length of the shortest token sequence in this	<pre>seq = (Sequence) elements[i];</pre>	<pre>list[i];</pre>

seq =	* @param	token sequence that	if
(Sequence)	parser the	overlaps, or	(seq.StartsWith(elem)
elements[i];	parser to check	*	
if	* @param	false otherwise	elem.StartsWith(seq))
<pre>(seq.IsRepetitive()) {</pre>	length the	*/	{
	maximum number of	public bool	
return true;	tokens to check	IsOverlap(LookAheadSet	return true;
}	*	set) {	}
}	* @return	for (int i	}
return	true if the next	= 0; i <	return
false;	tokens are in the set,	elements.Count; i++) {	false;
}	or	if	}
	*	(set.IsOverlap((Sequen	
/**	false otherwise	ce) elements[i])) {	/**
* Checks if	*/		* Checks if
the next token(s) in	public bool	return true;	the specified token
the parser match any	IsNext(Parser parser,	}	sequence is present in
token	int length) {	}	the
* sequence in	Sequence	return	* set.
this set.	seq;	false;	*
*	,	}	* @param elem
* @param	for (int i	,	the token sequence to
parser the	= 0; i <	/**	check
parser to check	elements.Count; i++) {	* Checks if a	*
*	seq =	token sequence is	* @return
* @return	(Sequence)	overlapping. An	true if the sequence
true if the next	elements[i];	overlapping token	is present in this
tokens are in the set,	if	* sequence is	set, or
or	(seq. IsNext (parser,	a token sequence that	*
*	length)) {	is identical to	false otherwise */
false otherwise  */	roturn truo:	another	*/ private bool
public bool	return true;	* sequence, but for the length.	Contains (Sequence
IsNext (Parser parser)	}	I.e. one of the two	elem) {
{	return	sequences may	return
Sequence	false;	* be longer	FindSequence(elem) !=
seq;	}	than the other.	null;
seq,	j	*	}
for (int i	/**	* @param seq	,
= 0; i <	* Checks if	the token sequence to	/**
elements.Count; i++) {	another look-ahead set	check	* Checks if
seq =	has an overlapping	*	some token sequence is
(Sequence)	token	* @return	present in both this
elements[i];	* sequence.	true if there is some	set
if	An overlapping token	token sequence that	* and a
(seq. IsNext(parser)) {	sequence is a token	overlaps, or	specified one.
	sequence	*	*
return true;	* that is	false otherwise	* @param set
}	identical to another	*/	the look-ahead set to
}	sequence, but for the	private bool	compare with
return	length.	IsOverlap(Sequence	*
false;	* I.e. one of	seq) {	* @return
}	the two sequences may	Sequence	true if the look-ahead
	be longer than the	elem;	sets intersect, or
/**	other.		*
* Checks if	*	for (int i	false otherwise
the next token(s) in	* @param set	= 0; i <	*/
the parser match any	the look-ahead set to	elements.Count; i++) {	public bool
token	check	elem =	Intersects (LookAheadSe
* sequence in	*	(Sequence)	t set) {
this set.	* @return	elements[i];	
*	true if there is some		

for (int i	* sequence	for (int i	1
= 0; i <	will be added instead.	= 0; i <	,
•		•	ı
elements.Count; i++) {	*	set.elements.Count;	/
if	* @param seq	i++) {	/**
(set. Contains ((Sequenc	the token sequence to	111//0	* Creates a
e) elements[i])) {	add	Add((Sequence)	new look-ahead set
	*/	set.elements[i]);	that is the result of
return true;	private void	}	reading .
}	Add (Sequence seq) {	}	* the
}	if	,	specified token. The
return	(seq. Length() >	/**	new look-ahead set
false;	maxLength) {	* Adds an	will contain
}	seq =	empty token sequence	* the rest of
	new	to this set. The	all the token
/**	Sequence (maxLength,	sequence will	sequences that started
* Finds an	seq);	* only be	with the
identical token	}	added if it is not	* specified
sequence if present in	if	already in the set.	token.
the set.	(!Contains(seq)) {	*/	*
*		public void	* @param
* @param elem	elements.Add(seq);	AddEmpty() {	token the
the token sequence to	}	Add (new	token to read
search for	}	Sequence());	*
*		}	* @return a
* @return an	/**		new look-ahead set
identical the token	* Adds a new	/**	containing the
sequence if found, or	token sequence with a	* Removes a	remaining tokens
*	single token to this	token sequence from	*/
null if not found	set.	this set.	public
*/	* The	*	LookAheadSet
private	sequence will only be	* @param seq	CreateNextSet(int
Sequence	added if it is not	the token sequence to	token) {
FindSequence (Sequence	already in the	remove	
elem) {	* set.	*/	LookAheadSet result =
for (int i	*	private void	new
= 0; i <	* @param	Remove(Sequence seq) {	LookAheadSet(maxLength
elements.Count; i++) {	token the		- 1);
if	token to add	elements.Remove(seq);	Sequence
(elements[i]. Equals(el	*/	}	seq;
em)) {	public void		object
	Add(int token) {	/**	value;
return (Sequence)	Add (new	* Removes all	,
elements[i];	Sequence(false,	the token sequences	for (int i
}	token));	from a specified set.	= 0; i <
}	}	Only	elements.Count; i++) {
return	,	* sequences	seq =
null;	/**	already in this set	(Sequence)
}	* Adds all	will be removed.	elements[i];
,	the token sequences	*	value
/**	from a specified set.	* @param set	= seq. GetToken(0);
* Adds a	Only	the set to remove from	if
token sequence to this	* sequences	*/	(value != null &&
set. The sequence will	not already in this	public void	token == (int) value)
only	set will be added.	RemoveAll(LookAheadSet	{
* be added if	set will be added.	set) {	· ·
it is not already in	* @param set	for (int i	result. Add (seg. Subsequ
the set. Also, if the	the set to add from	= 0; i <	ence(1));
	the set to add 110m */		ence (1//,
* sequence is	*/ public void	set.elements.Count;	\ \
longer than the	public void AddAll(LookAheadSet	i++) {	, noturn
allowed maximum, a	set) {	Remove((Sequence)	return
truncated	Set/ (	set.elements[i]);	result;
		Set. elements[1]/,	J

	return	// Create	CreateOverlaps(LookAhe
/**	result;	combinations	adSet set) {
* Creates a	}	for (int i	
new look-ahead set		= 0; i <	LookAheadSet result =
that is the	/**	elements.Count; i++) {	new
intersection of	* Creates a	first	LookAheadSet(maxLength
* this set	new look-ahead set	= (Sequence)	);
with another set. The	that is the	elements[i];	Sequence
token sequences in the	combination of	if	seq;
net	* this set	(first.Length() >=	
* set will	with another set. The	maxLength) {	for (int i
only have the repeat	combination is created		= 0; i <
flag set if it was set	by	result.Add(first);	elements.Count; i++) {
in	* creating	} else	seq =
* both the	new token sequences	if (first.Length() <=	(Sequence)
identical token	that consist of	0) {	elements[i];
sequences.	appending all * elements	result.AddAll(set);	(set. IsOverlap(seg)) {
* @param set	from the specified set	result. AddAll(Set), } else	(set. Isoverlap(seq)) (
the set to intersect	onto all elements in	} else	result.Add(seg);
with	this	(	resurt. Add (seq),
*	* set. This	for (int j = 0; j <	}
* @return a	is sometimes referred	set. elements. Count;	return
new look-ahead set	to as the cartesian	j++) {	result;
containing the	* product.	5 7 (	}
intersection	*	second = (Sequence)	,
*/	* @param set	set.elements[j];	/**
public	the set to combine	-0	* Creates a
LookAheadSet	with	result.Add(first.Conca	new look-ahead set
CreateIntersection(Loo	*	t(maxLength, second));	filter. The filter
kAheadSet set) {	* @return a	}	will contain
	new look-ahead set	}	* all
LookAheadSet result =	containing the	}	sequences from this
new	combination	return	set, possibly left
LookAheadSet(maxLength	*/	result;	trimmed by each one
);	public	}	* of the
Sequence	LookAheadSet	,	sequences in the
seq1;	CreateCombination(Look	/**	specified set.
Sequence	AheadSet set) {	* Creates a	*
seq2;	1 141 10	new look-ahead set	* @param set
6 (:	LookAheadSet result =	with overlaps from	the look-ahead set to
for (int i	new	another. All	trim with
= 0; i <	LookAheadSet(maxLength	* token	*
elements.Count; i++) {	); Sequence	sequences in this set	* @return a new look-ahead set
seq1 = (Sequence)	first;	that overlaps with the other set	filter
elements[i];	Sequence	* will be	*/
seq2 =	second;	added to the new look-	public
set. FindSequence (seq1)	Second,	ahead set.	LookAheadSet
:	// Handle	*	CreateFilter(LookAhead
if	special cases	* @param set	Set set) {
(seq2 != null &&	if	the look-ahead set to	, ,
seq1. IsRepetitive()) {	(this.Size() <= 0) {	check with	LookAheadSet result =
	return	*	new
	set;	* @return a	LookAheadSet(maxLength
result.Add(seq2);			_
result.Add(seq2); } else	} else if	new look-ahead set	);
		new look-ahead set containing the	); Sequence
} else	} else if		
} else	} else if (set.Size() <= 0) {	containing the	Sequence
} else if (seq2 != null) {	<pre>} else if (set.Size() &lt;= 0) {     return</pre>	containing the overlaps	Sequence first;
} else if (seq2 != null) {	<pre>} else if (set.Size() &lt;= 0) {           return this;</pre>	containing the overlaps */	Sequence first; Sequence

// Handle			
	for (int i	for (int i	* Creates
special cases	= 0; i <	= 0; i <	a new empty token
if	elements.Count; i++) {	elements.Count; i++) {	sequence. The repeat
(this. Size () <= 0	seq =	seq =	flag
set.Size() <= 0) {	(Sequence)	(Sequence)	* will be
return	elements[i];	elements[i];	set to false.
this;	if		*/
}	(seq.IsRepetitive()) {	<pre>buffer.Append("\n ");</pre>	public
			Sequence() {
// Create	result.Add(seq);	buffer.Append(seq.ToSt	
combinations	} else	ring(tokenizer));	this.repeat = false;
for (int i	{	}	
= 0; i <			this.tokens = new
elements.Count; i++) {	result.Add(new	buffer. Append $("\n}")$ ;	ArrayList(0);
first	Sequence(true, seq));	return	}
= (Sequence)	}	<pre>buffer.ToString();</pre>	
elements[i];	}	}	/**
for	return		* Creates
(int j = 0; j <	result;		a new token sequence
set.elements.Count;	}	/**	with a single token.
j++) {		* A token	*
	/**	sequence. This class	* @param
second = (Sequence)	* Returns a	contains a list of	repeat the
set.elements[j];	string representation	token ids.	repeat flag value
if	of this object.	* It is	* @param
(first.StartsWith(seco	*	immutable after	token the
nd)) {	* @return a	creation, meaning that	token to add
	string representation	no changes	*/
result.Add(first.Subse	of this object	* will be	public
quence(second.Length()	*/	made to an instance	Sequence(bool repeat,
));	public	after creation.	int token) {
}	override string	*	, ,
}	ToString() {		this.repeat = false;
}	return	* @version	,
return	ToString(null);	1.0	this.tokens = new
result;	}	*/	ArrayList(1);
}	•	private class	
,	/**	Sequence {	this.tokens.Add(token)
	* Returns a		, ,
/**	↑ Returns a		:
,		/**	;
* Creates a	string representation	,	;
* Creates a new identical look-		* The	; } /**
* Creates a new identical look- ahead set, except for	string representation of this object.	* The repeat flag. If this	} /**
* Creates a new identical look- ahead set, except for the	string representation of this object.  * * @param	* The	} /** * Creates
* Creates a new identical look- ahead set, except for the	string representation of this object.  * * @param tokenizer the	* The repeat flag. If this flag is set, the token *	/**  * Creates a new token sequence
* Creates a new identical look- ahead set, except for the     * repeat flag being set in each	string representation of this object.  * * @param tokenizer the tokenizer containing	* The repeat flag. If this flag is set, the token	/**  * Creates a new token sequence that is a duplicate of
* Creates a new identical look- ahead set, except for the	string representation of this object.  * * @param tokenizer the	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated	/**  * Creates a new token sequence that is a duplicate of * another
* Creates a new identical look— ahead set, except for the     * repeat flag being set in each token sequence.     *	string representation of this object.  *	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a
* Creates a new identical look— ahead set, except for the	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of
* Creates a new identical look— ahead set, except for the	string representation of this object.  *	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will
* Creates a new identical look— ahead set, except for the	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be
* Creates a new identical look— ahead set, except for the	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;	/**  * Creates  a new token sequence  that is a duplicate of  * another  sequence. Only a  limited number of  tokens will  * be  copied however. The
* Creates a new identical look— ahead set, except for the	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */ public string	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the
* Creates a new identical look— ahead set, except for the	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */ public string ToString(Tokenizer	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original
* Creates a new identical look— ahead set, except for the	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */ public string	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The list of token ids in	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be
* Creates a new identical look— ahead set, except for the	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */  public string ToString(Tokenizer tokenizer) {	* The repeat flag. If this flag is set, the token * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The list of token ids in this sequence.	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be kept intact.
* Creates a  new identical look— ahead set, except for the  * repeat flag being set in each token sequence.  * @return a  new repetitive look— ahead set  */ public LookAheadSet CreateRepetitive() {  LookAheadSet result =	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */  public string ToString(Tokenizer tokenizer) { StringBuilder buffer	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**      * The list of token ids in this sequence.  */	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be kept intact.
* Creates a  new identical look— ahead set, except for the  * repeat flag being set in each token sequence.  *  * @return a  new repetitive look— ahead set  */ public LookAheadSet CreateRepetitive() {  LookAheadSet result = new	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */  public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder();	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The list of token ids in this sequence.  */ private	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be kept intact.  *  * @param
* Creates a  new identical look— ahead set, except for the  * repeat flag being set in each token sequence.  *  * @return a  new repetitive look— ahead set  */ public LookAheadSet CreateRepetitive() {  LookAheadSet result = new LookAheadSet(maxLength	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder(); Sequence	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The list of token ids in this sequence.  */ private ArrayList tokens =	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be kept intact.  *  @param length  the
* Creates a new identical look— ahead set, except for the  * repeat flag being set in each token sequence.  *  * @return a new repetitive look— ahead set  */ public LookAheadSet CreateRepetitive() {  LookAheadSet result = new LookAheadSet (maxLength );	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */  public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder();	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The list of token ids in this sequence.  */ private	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be kept intact.  * @param length the maximum number of
* Creates a  new identical look— ahead set, except for the  * repeat flag being set in each token sequence.  * * @return a  new repetitive look— ahead set  */ public LookAheadSet CreateRepetitive() {  LookAheadSet result =  new LookAheadSet (maxLength ); Sequence	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder(); Sequence	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The list of token ids in this sequence.  */ private ArrayList tokens = null;	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be kept intact.  *  @param length  the
* Creates a new identical look— ahead set, except for the  * repeat flag being set in each token sequence.  *  * @return a new repetitive look— ahead set  */ public LookAheadSet CreateRepetitive() {  LookAheadSet result = new LookAheadSet (maxLength );	string representation of this object.  *  * @param tokenizer the tokenizer containing the tokens  * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder(); Sequence	* The repeat flag. If this flag is set, the token  * sequence or some part of it may be repeated infinitely.  */ private bool repeat = false;  /**  * The list of token ids in this sequence.  */ private ArrayList tokens =	/**  * Creates a new token sequence that is a duplicate of  * another sequence. Only a limited number of tokens will  * be copied however. The repeat flag from the original  * will be kept intact.  * @param length the maximum number of

* Ananan	* Anatum	*	1
* @param seq the	* @return the number of tokens	false otherwise	}
sequence to copy	in the sequence	*/	return
*/	*/	public	true;
public	public int	override bool	}
Sequence (int length,	Length() {	Equals(object obj) {	,
Sequence seq) {	return	if	/**
	tokens.Count;	(obj is Sequence) {	* Returns
this.repeat =	}		a hash code for this
seq.repeat;		return	object.
	/**	Equals((Sequence)	*
this. tokens = new	* Returns	obj);	* @return
ArrayList(length);	a token at a specified	} else	a hash code for this
if	position in the	{	object
(seq.Length() <	sequence.		*/
length) {	*	return false;	public
	* @param	}	override int
length = seq. Length();	pos the	}	GetHashCode() {
}	sequence position	,	return
for	*	/**	tokens. Count. GetHashCo
(int i = 0; i <	* @return	* Checks	de();
length; i++) {	the token id found, or	if this sequence is	}
4 1 · · · A11/· · · · · · · · · · · · · ·	null	equal to another	/stute
tokens. Add (seq. tokens[	*/ public	sequence.	/**
i]);	object GetToken(int	* Only sequences with the	* Checks
}	pos) {	same tokens in the	if this token sequence starts with the tokens
J	if	same order	from
/**	(pos >= 0 && pos <	* will be	* another
* Creates	tokens. Count) {	considered equal. The	sequence. If the other
a new token sequence	tokens. count) (	repeat flag will be	sequence is longer
that is a duplicate of	return tokens[pos];	*	than this
* another	} else	disregarded.	*
sequence. The new	{	*	sequence, this method
value of the repeat	•	* @param	will always return
flag will	return null;	seq the	false.
* be used	}	sequence to compare	*
however.	}	with	* @param
*		*	seq the
* @param	/**	* @return	token sequence to
repeat the new	* Checks	true if the sequences	check
repeat flag value	if this sequence is	are equal, or	*
* @param	equal to another	*	* @return
seq the	object.	false otherwise	true if this sequence
sequence to copy	* Only	*/	starts with the other,
*/	token sequences with	public	or
public	the same tokens in the	bool Equals (Sequence	* 
Sequence (bool repeat,	same * order	seq) { if	false otherwise */
Sequence seq) {	will be considered	(tokens.Count !=	public
this.repeat = repeat;	equal. The repeat flag	seq. tokens. Count) {	bool
this, repeat repeat,	will be	seq. tokens. county	StartsWith(Sequence
this.tokens =	*	return false;	seq) {
seq. tokens;	disregarded.	}	if
}	*	for	(Length() <
·	* @param	(int i = 0; i <	seq. Length()) {
/**	obj the	tokens.Count; i++) {	
* Returns	object to compare with	if	return false;
the length of the	*	(!tokens[i].Equals(seq	}
token sequence.	* @return	.tokens[i])) {	for
*	true if the objects		(int $i = 0$ ; $i <$
	are equal, or	return false;	

seq.tokens.Count; i++)	for		
{	(int $i = 0$ ; $i <$	token =	buffer. Append (tokens. T
if	tokens.Count; i++) {	<pre>parser.PeekToken(i);</pre>	oString());
(!tokens[i].Equals(seq	id	if	} else
.tokens[i])) {	= (int) tokens[i];	(token == null	{
		token.Id != id) {	
return false;	token =		<pre>buffer. Append("[");</pre>
}	<pre>parser.PeekToken(i);</pre>	return false;	
}	if	}	for (int $i = 0$ ; $i <$
return	(token == null	}	tokens.Count; i++) {
true;	token.Id != id) {	return	
}		true;	<pre>id = (int) tokens[i];</pre>
/**	return false;	}	-4
* Checks	}	/**	str = tokenizer.GetPatternDe
if this token sequence	return	* Returns	scription(id);
is repetitive. A	true;	a string	scription(id),
repetitive	}	representation of this	if (i > 0) {
* token	J	object.	11 (1 / 0) (
sequence is one with	/**	*	<pre>buffer. Append(" ");</pre>
the repeat flag set.	* Checks	* @return	salion appena ( ),
*	if the next token(s)	a string	}
* @return	in the parser matches	representation of this	,
true if this token	this	object	<pre>buffer.Append(str);</pre>
sequence is	* token	*/	}
repetitive, or	sequence.	public	
*	*	override string	<pre>buffer. Append("]");</pre>
false otherwise	* @param	ToString() {	}
*/	parser the	return	if
public	parser to check	ToString(null);	(repeat) {
<pre>bool IsRepetitive() {</pre>	* @param	}	
return	length the	,	<pre>buffer. Append(" *");</pre>
repeat;	maximum number of	/**	}
}	tokens to check	* Returns	return
/state	*	a string	<pre>buffer. ToString();</pre>
/** * Checks	* @return	representation of this	,
* Checks if the next token(s)	true if the next tokens are in the	object. *	/**
			,
		y (dhotam	* Crostos
in the parser matches	sequence, or	* @param	* Creates
this	*	tokenizer the	a new token sequence
this * token	* false otherwise	tokenizer the tokenizer containing	a new token sequence that is the
this	* false otherwise */	tokenizer the	a new token sequence that is the concatenation
this  * token sequence.  *	* false otherwise */ public	tokenizer the tokenizer containing the tokens	a new token sequence that is the concatenation * of this
this  * token sequence.  *  * @param	* false otherwise */ public bool IsNext(Parser	tokenizer the tokenizer containing the tokens  * # #@return	a new token sequence that is the concatenation * of this sequence and another.
this  * token  sequence.  *  * @param  parser the	* false otherwise */ public	tokenizer the tokenizer containing the tokens  *	a new token sequence that is the concatenation * of this
this  * token sequence.  *  * @param	* false otherwise */ public bool IsNext(Parser parser, int length) {	tokenizer the tokenizer containing the tokens  * # #@return	a new token sequence that is the concatenation * of this sequence and another. A maximum length for
this  * token  sequence.  *  * @param  parser the  parser to check	* false otherwise */ public bool IsNext(Parser parser, int length) { Token	tokenizer the tokenizer containing the tokens  *	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the
this  * token  sequence.  *  * @param  parser the  parser to check  *	* false otherwise */ public bool IsNext(Parser parser, int length) {	tokenizer the tokenizer containing the tokens  *	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return	* false otherwise */ public bool IsNext(Parser parser, int length) {	tokenizer the tokenizer containing the tokens  * * @return a string representation of this object  */	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next	* false otherwise */ public bool IsNext(Parser parser, int length) {	tokenizer the tokenizer containing the tokens  * * @return a string representation of this object  */ public	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the	* false otherwise	tokenizer the tokenizer containing the tokens  * * @return a string representation of this object  */ public string	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  *
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the	<pre># false otherwise</pre>	tokenizer the tokenizer containing the tokens  * * @return a string representation of this object  */ public string ToString(Tokenizer	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * #@param
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *	<pre># false otherwise     */     public bool IsNext(Parser parser, int length) {</pre>	tokenizer the tokenizer containing the tokens  * * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *  false otherwise	<pre># false otherwise     */     public bool IsNext(Parser parser, int length) {</pre>	tokenizer the tokenizer containing the tokens  * * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the maximum length of the
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *  false otherwise  */	<pre># false otherwise     */     public bool IsNext(Parser parser, int length) {</pre>	tokenizer the tokenizer containing the tokens  * * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the maximum length of the result
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *  false otherwise  */  public	<pre># false otherwise     */     public bool IsNext(Parser parser, int length) {</pre>	tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder();	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the maximum length of the result  * @param
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *  false otherwise  */  public  bool IsNext(Parser	<pre># false otherwise     */     public bool IsNext(Parser parser, int length) {</pre>	tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder(); string	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the maximum length of the result  * @param seq the
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *  false otherwise  */  public  bool IsNext(Parser  parser) {	<pre># false otherwise     */     public bool IsNext(Parser parser, int length) {</pre>	tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */ public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder(); string str;	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the maximum length of the result  * @param seq the other sequence
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *  false otherwise  */  public  bool IsNext(Parser  parser) {  Token  token;  int	<pre>false otherwise     */     public bool IsNext(Parser parser, int length) {         Token token;         int id;      if (length &gt; tokens.Count) {  length = tokens.Count;         }         for (int i = 0; i &lt; length; i++) {     id</pre>	tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */  public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder();  string str;  int	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the maximum length of the result  * @param seq the other sequence  *
this  * token  sequence.  *  * @param  parser the  parser to check  *  * @return  true if the next  tokens are in the  sequence, or  *  false otherwise  */  public  bool IsNext(Parser  parser) {  Token  token;	<pre># false otherwise</pre>	tokenizer the tokenizer containing the tokens  *  * @return a string representation of this object  */  public string ToString(Tokenizer tokenizer) {  StringBuilder buffer = new StringBuilder();  string str;  int	a new token sequence that is the concatenation  * of this sequence and another. A maximum length for the  * new sequence is also specified.  * @param length the maximum length of the result  * @param seq the other sequence  * * @return

nuhlia	while	* Checks if	l
public Sequence Concat(int	(start > 0 &&	this node is hidden,	,
length, Sequence seq)	res. tokens. Count > 0)	i.e. if it should not	/**
{	{	be	* The node
		* visible	name property (read-
Sequence res = new	res. tokens. RemoveAt(0)	outside the parser.	only).
Sequence (length,	:	*	*
this);	,	* @return	*
	start;	true if the node	*/
if	}	should be hidden, or	public
(seq.repeat) {	return	*	abstract string Name {
	res;	false otherwise	get;
res.repeat = true;	}	*/	}
}	}	internal	
length	}	virtual bool	/**
-= this.Length();	}	IsHidden() {	* Returns the
if	,	return	node name.
(length >	/*	false;	*
seq.Length()) {	* Node. cs	}	* @return the
res. tokens. AddRange(se	*/	/**	node name *
q. tokens);	using	* The node	* @see #Name
q. tokens),	System. Collections;	type id property	* esee #Name
{	using System. IO;	(read-only). This	* @deprecated
(	using bystem. 10,	value is set as	Use the Name property
for (int $i = 0$ ; $i <$	namespace Core.Library	* a unique	instead.
length; i++) {	{	identifier for each	*/
	•	type of node, in order	public virtual
res. tokens. Add (seq. tok	/**	to	string GetName() {
ens[i]);	* An abstract	* simplify	return
}	parse tree node. This	later identification.	Name;
}	class is inherited by	*	}
return	all	*	
res;	* nodes in the	*/	/**
}	parse tree, i.e. by	public	* The line
/stude	the token and	abstract int Id {	number property of the
/** * Creates	production * classes.	get;	first character in this
a new token sequence	* Classes.	J	* node (read-
that is a subsequence	ጥ	/**	only). If the node has
of	*	* Returns the	child elements, this
* this	*/	node type id. This	* value will
one.	public abstract	value is set as a	be fetched from the
*	class Node {	unique	first child.
* @param		* identifier	*
start the	/**	for each type of node,	*
subsequence start	* The parent	in order to simplify	*/
position	node.	* later	public virtual
*	*/	identification.	<pre>int StartLine {</pre>
* @return	private Node	*	get {
the new token	parent = null;	* @return the	int
subsequence	/	node type id	line;
*/	/**	*	
public	* The	* @see #Id	for
Sequence (int. start)	computed node values. */	* # ##################################	(int i = 0; i < Count; i++) {
Subsequence(int start)	,	* @deprecated	1**/ (
l	private ArrayList values =	Use the Id property instead.	line =
Sequence res = new	null;	instead. */	this[i].StartLine;
Sequence (Length(),	nurr,	public virtual	if
this);	/**	int GetId() {	(line >= 0) {
, ,	,	return Id;	· / •/ (
		*	

	for	*	* node (read-
return line;	(int i = 0; i < Count;	*/	only). If the node has
}	i++) {	public virtual	child elements, this
}		int EndLine {	* value will
return	col =	get {	be fetched from the
-1;	this[i].StartColumn;	int	last child.
1,	if	line:	*
}	$(co1 \ge 0)$ {	Time,	*
J	(601 / 0) (	for	*/
/**	return col;	(int i = Count - 1: i	public virtual
* The line	return cor,	>= 0; i) {	int EndColumn {
	J	/- U, I <sup></sup> ) (	
number of the first	}	1:	get {
character in this	return	line =	int
node. If the	-1;	this[i].EndLine;	col;
* node has	}	if	c
child elements, this	}	$(line \ge 0)$ {	for
value will be fetched			(int i = Count - 1; i
from	/**	return line;	>= 0; i) {
* the first	* The column	}	-
child.	number of the first	}	col =
*	character in this	return	this[i].EndColumn;
* @return the	node. If	-1;	if
line number of the	* the node	}	$(co1 \ge 0)$ {
first character, or	has child elements,	}	
* -1	this value will be		return col;
if not applicable	fetched	/**	}
*	* from the	* The line	}
* @see	first child.	number of the last	return
#StartLine	*	character in this	-1;
*	* @return the	node. If the	}
* @deprecated	column number of the	* node has	}
Use the StartLine	first token character,	child elements, this	
property instead.	or	value will be fetched	/**
*/	* -1	from	* The column
public virtual	if not applicable	* the last	number of the last
int GetStartLine() {	*	child.	character in this
return	* @see	*	node. If
StartLine;	#StartColumn	* @return the	* the node
}	*	line number of the	has child elements,
	* @deprecated	last token character,	this value will be
/**	Use the StartColumn	or	fetched
* The column	property instead.	* -1	* from the
number property of the	*/	if not applicable	last child.
first character in	public virtual	*	*
this	int GetStartColumn() {	* @see	* @return the
* node (read-	return	#EndLine	column number of the
only). If the node has	StartColumn;	*	last token character,
child elements, this	}	* @deprecated	or
* value will	,	Use the EndLine	* -1
be fetched from the	/**	property instead.	if not applicable
first child.	* The line	*/	*
*	number property of the	public virtual	* @see
*	last character in this	int GetEndLine() {	#EndColumn
*/	node	return	**
public virtual	* (read-	EndLine;	* @deprecated
int StartColumn {	only). If the node has	liabine,	Use the EndColumn
get {	child elements, this	J	property instead.
int	value	/**	*/
col;	varue * will be	* The column	≁/ public virtual
001,	fetched from the last	number property of the	int GetEndColumn() {
	child.	last character in this	return
	*	Tast Character III this	EndColumn;
	**		Endoording,

1	*/	*	* value array
J	public virtual	* @return the	list is allowed and is
/**	int Count {	child node found, or	immediately reflected
* The parent	get {	*	* through the
node property (read-	return	null if index out of	various value reading
only).	0;	bounds	and manipulation
*	}	*	methods.
*	}	*	*
*/	J	*/	*
public Node	/**	public virtual	*/
Parent {	* Returns the	Node this[int index] {	public
get {	number of child nodes.	get {	ArrayList Values {
return	*	return	get {
parent;	* @return the	null;	if
}	number of child nodes	}	(values == null) {
set {	*	}	
•	* @deprecated	•	values = new
this.parent = value;	Use the Count property	/**	ArrayList();
}	instead.	* Returns the	}
}	*/	child node with the	return
	public virtual	specified index.	values;
/**	int GetChildCount() {	*	}
* Returns the	return	* @param	set {
parent node.	Count;	index the	
*	}	child index, 0 <=	this.values = value;
* @return the		index < count	}
parent parse tree node	/**	*	}
*	* Returns the	* @return the	
* @see	number of descendant	child node found, or	/**
#Parent	nodes.	*	* Returns the
*	*	null if index out of	number of computed
* @deprecated	* @return the	bounds	values associated with
Use the Parent	number of descendant	*	this
property instead.	nodes	* @deprecated	* node. Any
*/	*	Use the class indexer	number of values can
public Node	*	instead.	be associated with a
GetParent() {	*/	*/	node
return	public int	public virtual	* through
Parent;	GetDescendantCount() {	Node GetChildAt(int	calls to AddValue().
}	int count	index) {	*
	= 0;	return	* @return the
/**	,	this[index];	number of values
* Sets the	for (int i	}	associated with this
parent node.	$= 0; i < Count; i++) {$	,	node
*	count	/**	*
* @param	+= 1 +	* The node	* @see
parent the new	this[i].GetDescendantC	values property. This	#Values
parent node	ount();	property provides	*
*/	}	direct	* @deprecated
public void	return	* access to	Use the Values and
SetParent (Node parent)	count;	the list of computed	Values. Count
Donont -	}	values associated with	properties
Parent =	/**	this * node during	instead.
parent;	* The child	* node during analysis. Note that	*/
J	node index (read-	setting this property	*/ public int
/**	only).	to	GetValueCount() {
* The child	oniy). *	* null will	if (values
node count property	* @param	remove all node	== null) {
(read-only).	index the	values. Any operation	null) (
(read only).	child index, 0 <=	on the	0;
*	index < Count	on the	} else {
	india . Count		, 6136 (

	*	* @param	public void
return values.Count;	* @see	values the	PrintTo(TextWriter
}	#Values	vector with node	output) {
, }	**************************************	values	output) (
J	* @deprecated	varues *	PrintTo(output, "");
/**	•		TTTTTTO (Output, ),
,	Use the Values	* @see	, , Pl 1 ()
* Returns a	property instead. Note	#Values	output. Flush();
computed value of this	that the	*	}
node, if previously	* Values	* @deprecated	/
set. A	property will never be	Use the Values	/**
* value may	null, but possibly	property and the	* Prints this
be used for storing	empty.	Values. AddRange	node and all subnodes
intermediate results	*/	* method	to the specified
in the	public	instead.	output
* parse tree	ArrayList	*/	* stream.
during analysis.	GetAllValues() {	public void	*
*	return	AddValues(ArrayList	* @param
* @param pos	values;	values) {	output the
the value position, 0	}	if (values	output stream to use
<= pos < count		!= null) {	* @param
*	/**		indent the
* @return the	* Adds a	Values. AddRange(values	indentation string
computed node value,	computed value to this	);	*/
or	node. The computed	}	private void
*	value may	}	PrintTo(TextWriter
null if not set	* be used for	,	output, string indent)
*	storing intermediate	/**	{
* @see	results in the parse	* Removes all	C
#Values	tree	computed values stored	output.WriteLine(inden
*	* during	in this node.	t + ToString());
* @deprecated		*	indent =
Use the Values	analysis. *	* @see	indent + " ";
_		∜ ⊌see #Values	
property and it's	* @param		for (int i
array indexer	value the	*	= 0; i < Count; i++) {
*	node value	* @deprecated	.1. [.] p
instead.	*	Use the Values	this[i].PrintTo(output
*/	* @see	property and the	, indent);
public object	#Values	Values. Clear	}
GetValue(int pos) {	*	* method	}
return	* @deprecated	instead. Alternatively	}
			i i
Values[pos];	Use the Values	the Values property	}
<pre>values[pos]; }</pre>	property and the	can	}
values[pos]; }			} /*
/**	property and the	can  * be set to null.	} /* * ParseException.cs
}	property and the Values.Add	can * be set	} /*     * ParseException.cs     */
} /**	property and the Values.Add * method	can  * be set to null.	
/**	property and the Values.Add  * method instead.	can * be set to null. */	
/**  * Returns the list with all the	property and the Values.Add  * method  instead.  */	can  * be set  to null.  */  public void	*/
/**  * Returns the list with all the computed values for	property and the Values.Add  * method  instead.  */  public void	can  * be set  to null.  */  public void  RemoveAllValues() {	*/ using System;
/**  * Returns the list with all the computed values for this	property and the Values.Add  * method  instead.  */  public void	can  * be set  to null.  */  public void  RemoveAllValues() {  values =	*/ using System; using
/**  * Returns the list with all the computed values for this  * node. Note	property and the Values.Add  * method  instead.  */  public void  AddValue(object value) {	can  * be set  to null.  */  public void  RemoveAllValues() {  values =	*/ using System; using System. Collections;
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a	property and the Values.Add  * method  instead.  */  public void AddValue(object value) {  if (value	can  * be set  to null.  */  public void  RemoveAllValues() {  values =	*/ using System; using System. Collections;
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will	property and the Values.Add  * method  instead.  */  public void AddValue(object value) {  if (value	can  * be set  to null.  */  public void  RemoveAllValues() {  values =  null;  }	*/ using System; using System. Collections; using System. Text;
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node	<pre>property and the Values.Add     * method instead.     */     public void AddValue(object value) {         if (value != null) {</pre>	<pre>can</pre>	*/ using System; using System. Collections; using System. Text;
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node (as it is the same	<pre>property and the Values.Add</pre>	<pre>can</pre>	*/ using System; using System. Collections; using System. Text;
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node	<pre>property and the Values.Add</pre>	<pre>can</pre>	*/ using System; using System.Collections; using System.Text; namespace Core.Library { /**
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node (as it is the same object).  *	<pre>property and the Values.Add</pre>	<pre>can</pre>	*/ using System; using System.Collections; using System.Text;  namespace Core.Library {  /**      * A parse
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node (as it is the same object).  *  * @return a	<pre>property and the Values. Add</pre>	<pre>can</pre>	*/ using System; using System.Collections; using System.Text;  namespace Core.Library {  /**     * A parse exception.
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node (as it is the same object).  * @return a list with all values,	<pre>property and the Values. Add</pre>	<pre>can</pre>	*/ using System; using System.Collections; using System.Text;  namespace Core.Library {  /**      * A parse
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node (as it is the same object).  *  * Creturn a list with all values, or	<pre>property and the Values. Add</pre>	<pre>can</pre>	*/ using System; using System.Collections; using System.Text;  namespace Core.Library {  /**     * A parse exception.     *
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node (as it is the same object).  *  * @return a list with all values, or	property and the Values. Add  * method  instead.  */  public void  AddValue(object value) {     if (value != null) {  Values. Add(value);  } }  /**  * Adds a set of computed values to this node.	<pre>can</pre>	<pre>*/ using System; using System.Collections; using System.Text;  namespace Core.Library {     /**     * A parse exception.     *     *</pre>
/**  * Returns the list with all the computed values for this  * node. Note that the list is not a copy, so changes will  * affect the values in this node (as it is the same object).  *  * Creturn a list with all values, or	<pre>property and the Values. Add</pre>	<pre>can</pre>	*/ using System; using System.Collections; using System.Text;  namespace Core.Library {  /**     * A parse exception.     *

<pre>public class ParseException :</pre>	type is used when another	information. This variable is only	information array, which is
•	* token	variable is only * used for	* only used
Exception {	* token than the expected one		for expected token
/**	is encountered.	unexpected token	errors. The list then
* The error	*/	errors. */	contains
	*/	private	contains *
type enumeration. */	UNEXPECTED TOKEN,	ArrayList details;	descriptions of the
public enum	UNEXI ECTED_TOKEN,	AllayList details,	expected tokens.
ErrorType {	/**	/**	expected tokens.
Effortype	* The	* The line	* @param type
/**	invalid token error	number.	the parse error type
* The	type is used when a	*/	* @param info
internal error type is	token	private int	the additional
only used to signal an	* pattern	line;	information
error	with an error message	Time,	* @param
* that is	is matched. The	/**	details the
a result of a bug in	*	* The column	additional detailed
the parser or	additional information	number.	information
tokenizer	provided should	*/	* @param line
* code.	contain the	private int	the line number, or -1
*/	* error	column:	for unknown
INTERNAL,	message.	Column,	* @param
INTERNAL,	*/	/**	column the
/**	**/	* Creates a	column number, or -1
* The I/O	INVALID TOKEN,	new parse exception.	for unknown
error type is used for	INVALID_TOKEN,	new parse exception.	*/
stream I/O errors.	/**	* @param type	public
*/	* The	the parse error type	ParseException(ErrorTy
10,	analysis error type is	* @param info	pe type,
10,	used when an error is	the additional	pe type,
/**	used when an error is	information	string info,
* The	encountered in the	* @param line	Stilling linio,
unexpected end of file	analysis. The	the line number, or -1	ArrayList details,
error type is used	additional information	for unknown	AllayList details,
when end	*	* @param	int line,
* of file	provided should	column the	int line,
is encountered instead	contain the error	column number, or -1	int column) {
of a valid token.	message.	for unknown	int cordinity (
*/	*/	*/	this.type
•••	ANALYSIS	public	= type;
UNEXPECTED EOF,	}	ParseException(ErrorTy	this.info
ONEM ECTED_ECT,	J	pe type,	= info;
/**	/**	ре суре,	inio,
* The	* The error	string info,	this.details =
unexpected character	type.	string into,	details;
error type is used	*/	int line,	this.line
when a	private	ine iine,	= line;
*	ErrorType type;	int column)	Tine,
character is read that	Effortype type,	·	this.column = column;
isn't handled by one	/**	this(type, info, null,	}
of the	* The	line, column) {	J
* token	additional information	ine, column, (	/**
patterns.	string.	J	* The error
*/	*/	/**	type property (read-
**/	private string	* Creates a	only).
UNEXPECTED CHAR,	info;	new parse exception.	*
oneni eoted_char,	IIIO,	This constructor is	*
/**	/**		*/
/** * The	/** * The	only * used to	*/ public
unexpected token error	* The additional details	* used to supply the detailed	public ErrorType Type {
anexpected token error	auditional uetalis	suppry the detailed	get {
			get (

return	/**	/**	public int
type;	* The	* Returns the	GetColumn() {
}	additional detailed	line number where the	return
}	error information	error occured.	column;
,	property	*	}
/**	* (read-	* @return the	
* Returns the	only).	line number of the	/**
error type.	*	error, or	* The message
*	*	* -1	property (read-only).
* @return the	*/	if unknown	This property contains
error type	public	*	* the
*	ArrayList Details {	* @see #Line	detailed exception
* @see #Type	get {	*	error message,
*	return	* @deprecated	including line and
* @deprecated	new	Use the Line property	* column
Use the Type property	ArrayList(details);	instead.	numbers when
instead.	}	*/	available.
*/	}	public int	*
public	/dul-	GetLine() {	* @see
ErrorType	/**	return	#ErrorMessage
GetErrorType() {	* Returns the	Line;	*/
return	additional detailed	}	public
Type;	error information.	/state	override string
}	* * @return the	/**	Message {
/**	* @return the additional detailed	* The column number property (read-	get {
/ <b>**</b> * The	error information	only). This is the	StringBuilder buffer
additional error	error information *	column	= new StringBuilder();
information property	* @see	* number	- new Stillighallael(),
(read-only).	#Details	where the error	// Add
(read only).	#Details *	occured, or -1 if	error description
*	* @deprecated	unknown.	error description
*/	Use the Details	*	buffer. Append (ErrorMes
public string	property instead.	*	sage);
Info {	*/	*/	Sage/,
get {	public	public int	// Add
return	ArrayList GetDetails()	Column {	line and column
info;	{	get {	if
}	return	return	(line > 0 && column >
}	Details:	column;	0) {
,	}	}	-, (
/**	·	}	buffer.Append(", on
* Returns the	/**	•	line: ");
additional error	* The line	/**	
information.	number property (read-	* Returns the	<pre>buffer.Append(line);</pre>
*	only). This is the	column number where	
* @return the	line	the error occured.	buffer.Append("
additional error	* number	*	column: ");
information	where the error	* @return the	
*	occured, or -1 if	column number of the	buffer.Append(column);
* @see #Info	unknown.	error, or	}
*	*	* -1	
* @deprecated	*	if unknown	return
Use the Info property	*/	*	<pre>buffer.ToString();</pre>
instead.	public int	* @see	}
*/	Line {	#Column	}
public string	get {	*	
GetInfo() {	return	* @deprecated	/**
return	line;	Use the Column	* Returns a
Info;	}	property instead.	default error message.
}	}	*/	*

4 Aroturn o	0000		if (i
* @return a default error message	case ErrorType.UNEXPECTED C	<pre>buffer. Append(info);</pre>	> 0) {
*	HAR:	}	, , ,
* @see		•	<pre>buffer.Append(", ");</pre>
#Message	buffer.Append("unexpec	break;	if
*	ted character '");	}	(i + 1 ==
* @deprecated	1 00 1 1/1 0		details.Count) {
Use the Message	<pre>buffer.Append(info);</pre>	return	1
property instead. */	<pre>buffer. Append("'");</pre>	<pre>buffer.ToString(); }</pre>	<pre>buffer. Append ("or "); }</pre>
public string	barrer, appeara ( ),	}	}
GetMessage() {	break;	,	,
return	case	/**	buffer.Append(details[
Message;	ErrorType.UNEXPECTED_T	* Returns the	i]);
}	OKEN:	error message. This	}
	1 00 1///	message will contain	
/**	buffer. Append ("unexpec	all the	return
* The error message property	ted token ");	<pre>* information available, except for</pre>	buffer.ToString();
(read-only). This	<pre>buffer. Append(info);</pre>	the line and column	}
property	if	number	}
* contains	(details != null) {	*	•
all the information		information.	/*
available, except for	buffer.Append(",	*	* Parser.cs
the line	expected ");	* @return the	*
* and column		error message	*/
number information. *	if (details.Count > 1)	* * @see	using System:
* @see	l	#ErrorMessage	using System; using
#Message	buffer.Append("one of	**************************************	System. Collections;
*	");	* @deprecated	using
*		Use the ErrorMessage	System. Collections. Gen
*/	}	property instead.	eric;
public string		*/	using System.IO;
ErrorMessage {	buffer. Append (GetMessa	public string	using System.Text;
get {	geDetails());	GetErrorMessage() {	
StringBuilder buffer	,	return ErrorMessage;	namespace Core.Library
= new StringBuilder();	break;	liformessage,	(
new seringsariaer () ,	case	,	/**
// Add	ErrorType.INVALID_TOKE	/**	* A base parser
type and info	N:	* Returns a	class. This class
switch		string containing all	provides the standard
(type) {	<pre>buffer.Append(info);</pre>	the detailed	parser
case	hara a la c	information in	* interface, as
ErrorType. IO:	break; case	* a list. The elements are separated	well as token handling.
buffer. Append ("I/O	ErrorType. ANALYSIS:	with a comma.	*
error: ");	Bilolly pot manifeld.	*	
	<pre>buffer.Append(info);</pre>	* @return the	
<pre>buffer.Append(info);</pre>		detailed information	*/
	break;	string	public abstract
break;		*/	class Parser {
case	default:	private string	/.tr.
ErrorType.UNEXPECTED_E OF:	buffer.Append("interna	GetMessageDetails() {	/** * The parser
01.	l error");	StringBuilder buffer	* The parser initialization flag.
buffer. Append ("unexpec	if	= new StringBuilder();	*/
ted end of file");	(info != null) {		private bool
• •		for (int i	initialized = false;
break;	<pre>buffer.Append(": ");</pre>	= 0; i <	
		details.Count; i++) {	/**

* The	}	* count is	* Creates a
production output out	J	higher than zero (0),	new parser.
of	/**	this log will be	*
RecursiveDescentParser	* The	thrown as the	* @param
Recui si vedescenti di sei	tokenizer to use.	* result from	input the
*/	*/	the parse() method.	input stream to read
,	,	the parse() method.  */	*
public	private	,	from
SyntaxProductions	Tokenizer tokenizer;	private	* @param
production = new	/	ParserLogException	analyzer the
<pre>SyntaxProductions();</pre>	/**	errorLog = new	analyzer callback to
,	* The	<pre>ParserLogException();</pre>	use
/**	analyzer to use for		*
* Get the	callbacks.	/**	* @throws
Production set of	*/	* The error	ParserCreationExceptio
production.	private	recovery counter. This	n if the tokenizer
*/	Analyzer analyzer;	counter is initially	couldn't be
public string		set to a	*
${\tt GetRecursiveProduction}$	/**	* negative	initialized correctly
()	* The list of	value to indicate that	*
{	production patterns.	no error requiring	*
return	*/	recovery	*/
("Enter:	private	* has been	internal
<startprogram>\n" +</startprogram>	ArrayList patterns =	encountered. When a	Parser(TextReader
production.GetRecursiv	new ArrayList();	parse error is found,	input, Analyzer
eProductions());	,, ,	the counter	analyzer) {
}	/**	* is set to	anaryzer, (
,	* The map	three (3), and is then	this.tokenizer =
public int	with production	decreased by one for	NewTokenizer(input);
GetLastProductionCode(	patterns and their	each	newrokenizer (input),
)	id:s. This map	* correctly	this.analyzer =
,	* contains	read token until it	(analyzer == null) ?
1			
return	the production	reaches zero (0).	NewAnalyzer() :
production.GetLastProd	patterns indexed by	*/	analyzer;
uctionCode();	their id:s.	private int	}
}	*/	errorRecovery = -1;	,
	private	,	/**
public string	Hashtable patternIds =	/**	* Creates a
GetLastProductionState	new Hashtable();	* Creates a	new parser.
()		new parser.	*
{	/**	*	* @param
return	* The list of	* @param	tokenizer the
production.GetLastProd	buffered tokens. This	input the	tokenizer to use
uctionState();	list will contain	input stream to read	*/
}	tokens that	from	internal
	* have been	*	Parser(Tokenizer
public	read from the	* @throws	tokenizer) :
List <string></string>	tokenizer, but not yet	ParserCreationExceptio	this(tokenizer, null)
GetAllProductionState(	consumed.	n if the tokenizer	{
)	*/	couldn't be	}
{	private	*	
return	ArrayList tokens = new	initialized correctly	/**
production.GetAllProdu	ArrayList();	*	* Creates a
ctionState();	, ,	*	new parser.
}	/**	*/	*
,	* The error	internal	* @param
public	log. All parse errors	Parser (TextReader	tokenizer the
List (int)	will be added to this	input) : this(input,	tokenizer to use
GetAllProductionCode()	log as	null) {	* @param
Sc talli roude troncode ()	* the parser	nuii) (	
l matum		l	analyzer the analyzer callback to
return	attempts to recover	/ state	•
production.GetAllProdu	from the error. If the	/**	use */
ctionCode();	error		*/

:	wl1 4-		ale.
internal Parser(Tokenizer	* subclass to provide a custom	/**	* * @param
tokenizer, Analyzer	implementation.	* Returns the	initialized the new
analyzer) {	*	tokenizer in use by	initialized flag
	* @return the	this parser.	*/
this.tokenizer =	analyzer created	*	internal void
tokenizer;	*	* @return the	SetInitialized(bool
	*	tokenizer in use by	initialized) {
this.analyzer =	*/	this parser	
(analyzer == null) ?	protected	*	this.initialized =
NewAnalyzer() :	virtual Analyzer	*	initialized;
analyzer;	NewAnalyzer() {	*	}
}	// TODO:	* @see	
	This method should	#Tokenizer	/**
/**	really be abstract,	*	* Adds a new
* Creates a	but it isn't in this	* @deprecated	production pattern to
new tokenizer for this	yangian dua ta	Use the Tokenizer	the parser. The first
parser. Can be overridden by	version due to backwards	property instead. */	pattern * added is
* a subclass	compatibility	public	assumed to be the
to provide a custom	requirements.	Tokenizer	starting point in the
implementation.	return new	GetTokenizer() {	grammar. The
*	Analyzer();	return	* patterns
* @param in	}	Tokenizer;	added may be validated
the input stream to		}	to some extent.
read from	/**		*
*	* The	/**	* @param
* @return the	tokenizer property	* Returns the	pattern the
tokenizer created	(read-only). This	analyzer in use by	pattern to add
*	property contains	this parser.	*
* @throws	* the	*	* @throws
ParserCreationExceptio	tokenizer in use by	* @return the	ParserCreationExceptio
n if the tokenizer	this parser.	analyzer in use by	n if the pattern
couldn't be	*	this parser *	couldn't be
initialized correctly	*/	*	added correctly to the
*	public	*	parser
*	Tokenizer Tokenizer {	* @see	*/
*/	get {	#Analyzer	public virtual
protected	return	*	void
virtual Tokenizer	tokenizer;	* @deprecated	AddPattern(ProductionP
NewTokenizer(TextReade	}	Use the Analyzer	attern pattern) {
r input) {	}	property instead.	if
// TODO:		*/	(pattern.Count <= 0) {
This method should	/**	public	throw
really be abstract,	* The	Analyzer GetAnalyzer()	new
but it isn't in this	analyzer property	{	ParserCreationExceptio
	(read-only). This	return	n (
version due to	property contains	Analyzer;	December 11 Page 11
backwards	* the	}	ParserCreationException. ErrorType. INVALID_PR
compatibility requirements.	analyzer in use by this parser.	/**	ODUCTION,
return new	this parser.	* Sets the	ODUCTION,
Tokenizer(input);	*	parser initialized	pattern. Name,
}	*/	flag. Normally this	paccolli, Lamo,
,	public	flag is set by	"no production
/**	Analyzer Analyzer {	* the	alternatives are
* Creates a	get {	prepare() method, but	present (must have at
new analyzer for this	return	this method allows	" +
parser. Can be	analyzer;	further	
overridden by a	}	*	"least one)");
	}	modifications to it.	}

if	n. ErrorType. INVALID PA	alternative for	* @param elem
(patternIds.ContainsKe	RSER,	completeness.	the production pattern
y (pattern. Id)) {	KOLK,	* If some	element to check
throw	"no production	element in the	*
	_		•
new	patterns have been	alternative referenced	* @throws
ParserCreationExceptio	added");	a production	ParserCreationExceptio
n (	}	* pattern not	n if the element
	for (int i	added to this parser,	referenced a
ParserCreationExceptio	= 0; i <	a parser creation	*
n.ErrorType.INVALID_PR	patterns.Count; $i++$ ) {	* exception	pattern not added to
ODUCTION,		will be thrown.	this parser
	CheckPattern((Producti	*	*/
pattern. Name,	onPattern)	* @param name	private void
•	patterns[i]);	the name of the	CheckElement(string
"another pattern with	}	pattern being checked	name,
the same id (" +	,	* @param alt	name,
pattern. Id +	SetInitialized(true);	*	ProductionPatternEleme
pattern. 10 + ")	Setilitiaiized(tide),	the production pattern	
,	}	alternative	nt elem) {
has already been		*	
added");	/**	* @throws	if
}	* Checks a	ParserCreationExceptio	(elem.IsProduction()
	production pattern for	n if the alternative	&& GetPattern(elem.Id)
patterns.Add(pattern);	completeness. If some	*	== null) {
	rule	referenced a pattern	throw
patternIds.Add(pattern	* in the	not added to this	new
. Id, pattern);	pattern referenced an	parser	ParserCreationExceptio
, , ,	production pattern not	*/	n (
SetInitialized(false);	added	private void	II (
}	* to this	CheckAlternative(strin	ParserCreationExceptio
J			= = = = = = = = = = = = = = = = = = =
/	parser, a parser	g name,	n. ErrorType. INVALID_PR
/**	creation exception	D 1 D 41.	ODUCTION,
* Initializes	will be thrown.	ProductionPatternAlter	
the parser. All the	*	native alt) {	name,
added production	* @param		_
patterns will	pattern the	for (int i	"an undefined
* be analyzed	production pattern to	= 0; i < alt.Count;	production pattern id
for ambiguities and	check	i++) {	(" + elem.Id +
errors. This method	*		")
also	* @throws	CheckElement(name,	is referenced");
* initializes	ParserCreationExceptio	alt[i]);	}
internal data	n if the pattern	}	}
structures used during	referenced a	}	,
the parsing.	*	,	/**
*	pattern not added to	/**	* Resets this
* @throws	this parser	* Checks a	parser for usage with
ParserCreationExceptio	this parser */		
	,	production pattern	another input stream.
n if the parser	private void	element for	The
couldn't be	CheckPattern(Productio	completeness. If	* associated
*	nPattern pattern) {	* the element	tokenizer and analyzer
initialized correctly	for (int i	references a	will also be reset.
*/	= 0; i <	production pattern not	This
public virtual	pattern.Count; i++) {	added to	* method will
void Prepare() {		* this	clear all the internal
if	CheckAlternative(patte	parser, a parser	state and the error
(patterns.Count <= 0)	rn.Name, pattern[i]);	creation exception	log in
{	}	will be thrown.	* the parser.
throw	}	*	It is normally called
new	j	* @param name	in order to reuse a
	/**	•	
ParserCreationExceptio	,	the name of the	parser
n (	* Checks a	pattern being checked	* and
ParserCreationExceptio	production pattern		tokenizer pair with
rarsertreationPycentio			

 ${\tt ParserCreationExceptio}$ 

14:-1- :4	at.	at.	:
multiple input	*	*	input couldn't be
streams, thereby	* @see	* @see	parsed
* avoiding	Tokenizer#Reset *	#Prepare	*
the cost of re-	·	* @see #Reset	correctly */
analyzing the grammar	* @since 1.6	* @see	,
structures.	*/	Tokenizer#Reset	protected
*	public void	*/	abstract Node
* @param	Reset (TextReader	public Node	ParseStart();
input the new	input, Analyzer	Parse() {	/state
input stream to read	analyzer) {	Node root	/ <b>*</b> *
*	11: 1 : D (/:	= null;	* Factory
* @see	this.tokenizer.Reset(i	//	method to create a new
Tokenizer#Reset	nput);	//	production node. This
* @see	.1. 1	Initialize parser	method
Analyzer#Reset	this.analyzer =	if	* can be
*	analyzer;	(!initialized) {	overridden to provide
*	}	- 0	other production
*/	,	Prepare();	implementations
public void	/**	}	* than the
Reset(TextReader	* Parses the		default one.
input) {	token stream and	this.tokens.Clear();	*
	returns a parse tree.		* @param
this.tokenizer.Reset(i	This	this.errorLog = new	pattern the
nput);	* method will	<pre>ParserLogException();</pre>	production pattern
	call Prepare() if not		*
this.analyzer.Reset();	previously called. It	this.errorRecovery = -	* @return the
}	* will also	1;	new production node
	call the Reset()		*
/**	method, to make sure	// Parse	*
* Resets this	that only	input	*/
parser for usage with	* the	try {	protected
another input stream.	Tokenizer.Reset()	root =	virtual Production
The	method must be	ParseStart();	NewProduction(Producti
* associated	explicitly called in	} catch	onPattern pattern) {
tokenizer will also be	* order to	(ParseException e) {	return
reset and the analyzer	reuse a parser for		analyzer.NewProduction
* replaced.	multiple input	AddError(e, true);	(pattern);
This method will clear	streams. In case	}	}
all the internal state	* of a parse		
and	error, the parser will	// Check	/**
* the error	attempt to recover and	for errors	* Adds an
log in the parser. It	* throw all	if	error to the error
is normally called in	the errors found in a	(errorLog.Count > 0) {	log. If the parser is
order	parser log exception	throw	in error
* to reuse a	in the	errorLog;	* recovery
parser and tokenizer	* end.	}	mode, the error will
pair with multiple	*		not be added to the
input	* @return the	return	log. If the
* streams,	parse tree	root;	* recovery
thereby avoiding the	*	}	flag is set, this
cost of re-analyzing	* @throws		method will set the
the	ParserCreationExceptio	/**	error recovery
* grammar	n if the parser	* Parses the	* counter
structures.	couldn't be	token stream and	thus enter error
*	*	returns a parse tree.	recovery mode. Only
* @param	initialized correctly	*	lexical or
input the new	* @throws	* @return the	* syntactical
input stream to read	ParserLogException if	parse tree	errors require
* @param	the input couldn't be	*	recovery, so this flag
analyzer the new	parsed	* @throws	shouldn't be
analyzer callback to	*	ParseException if the	* set
use	correctly		otherwise.
	ž		

*	if	}	*
* @param e	(patterns.Count <= 0)	,	* @param node
the error to add	{	/**	the parent parse tree
* @param	return	* Handles the	node
recovery the	null;	parser leaving a	* @param
recover flag	} else {	production. This	child the
*/	return	method calls the	child parse tree node,
internal void	(ProductionPattern)	* appropriate	or null
AddError(ParseExceptio	patterns[0];	analyzer callback if	*/
n e, bool recovery) {	}	the node is not	internal void
if	}	hidden, and	AddNode(Production
(errorRecovery <= 0) {		* returns the	node, Node child) {
	/**	result. Note that this	if
errorLog.AddError(e);	* Returns the	method will not call	(errorRecovery >= 0) {
}	ordered set of	any	// Do
if	production patterns.	* callback if	nothing
(recovery) {	*	an error requiring	} else if
	* @return the	recovery has ocurred.	(node.IsHidden()) {
errorRecovery = 3;	ordered set of	*	
}	production patterns	* @param node	node.AddChild(child);
}	*/	the parse tree node	} else if
,	internal	*	(child != null &&
/**	ICollection	* @return the	child.IsHidden()) {
* Returns the	GetPatterns() {	parse tree node, or	for
production pattern	return	*	(int i = 0; i <
with the specified id.	patterns;	null if no parse tree	child.Count; i++) {
*	}	should be created	A
* @param id	/steate	*/ internal Node	AddNode (node,
the production pattern	/** * Handles the	ExitNode (Node node) {	child[i]);
1d *		if	} else {
* @return the	parser entering a production. This	(!node. IsHidden() &&	try {
production pattern	method calls the	errorRecovery < 0) {	tī y (
found, or	* appropriate	try {	analyzer.Child(node,
*	analyzer callback if	tī y (	child);
null if non-existent	the node is not	return	}
*/	hidden. Note	analyzer.Exit(node);	catch (ParseException
internal	* that this	anaryzer. Exit (node);	e) {
ProductionPattern	method will not call	catch (ParseException	5, (
GetPattern(int id) {	any callback if an	e) {	AddError(e, false);
return	error	-, (	}
(ProductionPattern)	* requiring	AddError(e, false);	}
patternIds[id];	recovery has ocurred.	}	}
}	*	}	
	* @param node	return	/**
/**	the parse tree node	node;	* Reads and
* Returns the	*/	}	consumes the next
production pattern for	internal void		token in the queue. If
the starting	EnterNode(Node node) {	/**	no token
production.	if	* Handles the	* was
*	(!node.IsHidden() &&	parser adding a child	available for
* @return the	errorRecovery < 0) {	node to a production.	consumation, a parse
start production	try {	This	error will be
pattern, or		* method	* thrown.
*	analyzer.Enter(node);	calls the appropriate	*
null if no patterns	}	analyzer callback.	* @return the
have been added	catch (ParseException	Note that this	token consumed
*/	e) {	* method will	*
internal	AddEnter ( C. 1 )	not call any callback	* @throws
ProductionPattern	AddError(e, false);	if an error requiring	ParseException if the
GetStartPattern() {	}	* recovery has ocurred.	input stream couldn't be read or
	J	nas oculteu.	ne read Or

*	internal Token	*/	((ProductionPattern)
parsed correctly	NextToken(int id) {	internal Token	patterns[i]));
*/	Token	<pre>PeekToken(int steps) {</pre>	1 2 3,7,7
internal Token	<pre>token = NextToken();</pre>	Token	buffer. Append (" $\n$ ");
NextToken() {	ArrayList	token;	}
Token	list;		return
token = PeekToken(0);	: 6	while	<pre>buffer. ToString();</pre>
if (token	if (token.Id == id) {	(steps >= tokens.Count) {	}
!= null) {	if	tokens. Count) { try {	/**
. 11411)	(errorRecovery > 0) {	01,7	* Returns a
tokens.RemoveAt(0);	•	token =	string representation
return	errorRecovery;	<pre>tokenizer.Next();</pre>	of a production
token;	}	if	pattern.
} else {	return	$(token == null) {$	*
throw	token; } else {	noturn null.	* @param prod
new ParseException(	list =	return null;	the production pattern
ParseException.ErrorTy	new ArrayList(1);	else {	* @return a
pe. UNEXPECTED EOF,		(	detailed string
	list.Add(tokenizer.Get	tokens.Add(token);	representation of the
null,	PatternDescription(id)	}	pattern
	);	}	*/
tokenizer.GetCurrentLi	throw	catch (ParseException	private string
ne(),	new ParseException(	e) {	ToString(ProductionPat tern prod) {
tokenizer.GetCurrentCo	ParseException.ErrorTy	AddError(e, true);	tern prod) {
lumn());	pe. UNEXPECTED TOKEN,	AddETIOT(c, true);	StringBuilder buffer
}	_ /	}	= new StringBuilder();
}	<pre>token.ToShortString(),</pre>	return	
		<pre>(Token) tokens[steps];</pre>	StringBuilder indent
/**	list,	}	= new StringBuilder();
* Reads and		}	
* Reads and consumes the next	list, token.StartLine,	/**  * Poturns a	LookAheadSet set;
* Reads and consumes the next token in the queue. If	token.StartLine,	* Returns a	LookAheadSet set; int
* Reads and consumes the next		* Returns a string representation	LookAheadSet set;
* Reads and consumes the next token in the queue. If no token was	token.StartLine,	* Returns a	LookAheadSet set; int
* Reads and consumes the next token in the queue. If no token was * available	token.StartLine,	* Returns a string representation of this parser. The	LookAheadSet set; int
* Reads and consumes the next token in the queue. If no token was	<pre>token. StartLine, token. StartColumn); } /**</pre>	* Returns a string representation of this parser. The string will	LookAheadSet set; int i;
* Reads and consumes the next token in the queue. If no token was	<pre>token. StartLine, token. StartColumn); }  /**</pre>	* Returns a string representation of this parser. The string will  * contain all the production definitions and	LookAheadSet set; int i; buffer.Append(prod.Nam e);
* Reads and consumes the next token in the queue. If no token was	<pre>token.StartLine, token.StartColumn); }  /**      * Returns a token from the queue.</pre>	* Returns a string representation of this parser. The string will	LookAheadSet set; int i; buffer.Append(prod.Nam
* Reads and consumes the next token in the queue. If no token was	<pre>token.StartLine,  token.StartColumn);      } }  /**      * Returns a token from the queue. This method is used to</pre>	* Returns a string representation of this parser. The string will	LookAheadSet set; int i; buffer.Append(prod.Nam e); buffer.Append(" (");
* Reads and consumes the next token in the queue. If no token was	<pre>token.StartLine,  token.StartColumn);      }  /**      * Returns a token from the queue. This method is used to check</pre>	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional	LookAheadSet set; int i; buffer.Append(prod.Nam e);
* Reads and consumes the next token in the queue. If no token was	<pre>token.StartLine,  token.StartColumn); }  /**  * Returns a token from the queue. This method is used to</pre>	* Returns a string representation of this parser. The string will     * contain all the production definitions and various additional     * information.	LookAheadSet set; int i; buffer.Append(prod.Nam e); buffer.Append(" (");
* Reads and consumes the next token in the queue. If no token was	<pre>token.StartLine,  token.StartColumn);      }  /**      * Returns a token from the queue. This method is used to check      * coming</pre>	* Returns a string representation of this parser. The string will     * contain all the production definitions and various additional     * information. *	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") ");
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a  token from the queue.  This method is used to check  * coming  tokens before they have been consumed.  Any number of	* Returns a string representation of this parser. The string will	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") "); for (i =
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a  token from the queue.  This method is used to check  * coming  tokens before they have been consumed.  Any number of  * tokens	* Returns a string representation of this parser. The string will	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") "); for (i = 0; i < buffer.Length;
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be	* Returns a string representation of this parser. The string will	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") "); for (i =
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.	* Returns a string representation of this parser. The string will	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") ");  for (i = 0; i < buffer.Length; i++) {
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") "); for (i = 0; i < buffer.Length;
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.	* Returns a string representation of this parser. The string will	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") ");  for (i = 0; i < buffer.Length; i++) {
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *  * @param	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string	LookAheadSet set; int i;  buffer.Append(prod.Nam e);  buffer.Append(" (");  buffer.Append(prod.Id);  buffer.Append(") ");  for (i = 0; i < buffer.Length; i++) {
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *  * @param steps the	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string ToString() {	LookAheadSet set; int i;  buffer. Append (prod. Nam e);  buffer. Append (" (");  buffer. Append (prod. Id);  buffer. Append (") ");  for (i = 0; i < buffer. Length; i++) {  indent. Append (" ");  buffer. Append (" ");
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *  * @param steps the token queue number, zero (0) for first  *	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string ToString() {  StringBuilder buffer = new StringBuilder();	LookAheadSet set; int i;  buffer. Append (prod. Nam e);  buffer. Append (" (");  buffer. Append (prod. Id);  buffer. Append (" ");  for (i = 0; i < buffer. Length; i++) {  indent. Append (" ");  buffer. Append (" ");
* Reads and consumes the next token in the queue. If no token was	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *  * @param steps the token queue number, zero (0) for first  *  * @return the	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string ToString() {  StringBuilder buffer = new StringBuilder();  for (int i	LookAheadSet set; int i;  buffer. Append (prod. Nam e);  buffer. Append (" (");  buffer. Append (prod. Id);  buffer. Append (prod. Id);  for (i = 0; i < buffer. Length; i++) {  indent. Append (" ");  buffer. Append (" ");  for (i =
* Reads and consumes the next token in the queue. If no token was  * available for consumation, a parse error will be thrown. A  * parse error will also be thrown if the token id didn't match  * the specified one.  * @param id the expected token id  * @ Teturn the token consumed  * @ Throws ParseException if the input stream couldn't be parsed  * correctly, or if the	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *  * @param steps the token queue number, zero (0) for first  *  * @return the token in the queue, or	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string ToString() {  StringBuilder buffer = new StringBuilder();  for (int i = 0; i <	LookAheadSet set; int i;  buffer. Append (prod. Nam e);  buffer. Append (" (");  buffer. Append (prod. Id);  buffer. Append (prod. Id);  for (i = 0; i < buffer. Length; i++) {  indent. Append (" ");  buffer. Append (" ");  indent. Append (" ");  for (i = 0; i < prod. Count;
* Reads and consumes the next token in the queue. If no token was  * available for consumation, a parse error will be thrown. A  * parse error will also be thrown if the token id didn't match  * the specified one.  *  * @param id the expected token id  *  * @return the token consumed  *  * @throws ParseException if the input stream couldn't be parsed  * correctly, or if the token wasn't expected	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *  * @param steps the token queue number, zero (0) for first  *  * @return the token in the queue, or	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string ToString() {  StringBuilder buffer = new StringBuilder();  for (int i	LookAheadSet set; int i;  buffer. Append (prod. Nam e);  buffer. Append (" (");  buffer. Append (prod. Id);  buffer. Append (prod. Id);  buffer. Append (" ");  for (i = 0; i < buffer. Length; i++) {  indent. Append (" ");  buffer. Append (" ");  indent. Append (" ");  for (i = 0; i < prod. Count; i++) {
* Reads and consumes the next token in the queue. If no token was  * available for consumation, a parse error will be thrown. A  * parse error will also be thrown if the token id didn't match  * the specified one.  * @param id the expected token id  * @ Teturn the token consumed  * @ Throws ParseException if the input stream couldn't be parsed  * correctly, or if the	token. StartLine,  token. StartColumn);  }  /**  * Returns a token from the queue. This method is used to check  * coming tokens before they have been consumed. Any number of  * tokens forward can be checked.  *  * @param steps the token queue number, zero (0) for first  *  * @return the token in the queue, or	* Returns a string representation of this parser. The string will  * contain all the production definitions and various additional  * information.  *  * @return a detailed string representation of this parser  */ public override string ToString() {  StringBuilder buffer = new StringBuilder();  for (int i = 0; i <	LookAheadSet set; int i;  buffer. Append (prod. Nam e);  buffer. Append (" (");  buffer. Append (prod. Id);  buffer. Append (prod. Id);  for (i = 0; i < buffer. Length; i++) {  indent. Append (" ");  buffer. Append (" ");  indent. Append (" ");  for (i = 0; i < prod. Count;

		if (min ==	*
<pre>buffer.Append(indent);</pre>	StringBuilder buffer	0 && max == 1) {	ParserCreationExceptio
}	<pre>= new StringBuilder();</pre>		n. cs
		<pre>buffer. Append("]");</pre>	*/
buffer. Append (ToString	for (int i	} else if	
(prod[i]));	= 0; i < alt.Count;	(min == 0 && max ==	using System;
1 CC A 1/"\ "\	i++) {	Int32.MaxValue) {	using
buffer. Append $("\n")$ ;	if (i	1 C.C A 1 ("")	System. Collections;
for (i =	> 0) {	<pre>buffer.Append("*"); } else if</pre>	using System.Text;
0; i < prod. Count;	<pre>buffer.Append(" ");</pre>	(min == 1 && max ==	namespace Core.Library
i++) {	buffer. Append ( ),	Int32. MaxValue) {	{
set =	J	intoz. maxvaracy	
prod[i].LookAhead;	buffer. Append (ToString	<pre>buffer. Append("+");</pre>	/**
if	(alt[i]));	} else if	* A parser
(set.GetMaxLength() >	}	(min != 1    max != 1)	creation exception.
1) {	return	{	This exception is used
	<pre>buffer.ToString();</pre>		for signalling
buffer.Append("Using	}	<pre>buffer. Append("{");</pre>	* an error in the
") ;			token or production
1 00 1/ 00 1/	/**	buffer.Append(min);	patterns, making it
buffer. Append (set. GetM	* Returns a	1	impossible
<pre>axLength());</pre>	string representation	<pre>buffer. Append(", ");</pre>	* to create a
buffer.Append(" token	of a production pattern	buffer.Append(max);	working parser or tokenizer.
look-ahead for	* element.	buller. Append (max),	*
alternative ");	*	<pre>buffer. Append(")");</pre>	·
,,	* @param elem	}	*
<pre>buffer.Append(i + 1);</pre>	the production pattern	return	*/
	element	<pre>buffer.ToString();</pre>	public class
<pre>buffer.Append(": ");</pre>	*	}	ParserCreationExceptio
	* @return a		n : Exception {
buffer.Append(set.ToSt	detailed string	/**	
ring(tokenizer));	representation of the	* Returns a	/**
1 00 1 1/11 11	element	token description for	* The error
buffer. Append $("\n")$ ;	*/	a specified token.	type enumeration.
}	private string ToString(ProductionPat	* * @param	*/ public enum
return	ternElement elem) {	token the	ErrorType {
buffer. ToString();	ternbrement erem/ (	token to describe	Effortype
}	StringBuilder buffer	*	/**
,	= new StringBuilder();	* @return the	* The
/**	int	token description	internal error type is
* Returns a	<pre>min = elem.MinCount;</pre>	*/	only used to signal an
string representation	int	internal	* error
of a production	<pre>max = elem.MaxCount;</pre>	string	that is a result of a
pattern		GetTokenDescription(in	bug in the parser or
*	if (min ==	t token) {	*
alternative.	0 && max == 1) {	if	tokenizer code.
*	1 · CC · A · · 1(" [") ·	(tokenizer == null) {	*/
* @param alt	buffer. Append("[");	return "":	INTERNAL,
the production pattern alternative	$\mathbf{if}$	, } else {	/**
*	(elem.IsToken()) {	return	* The
* @return a	(OTOM: ISTOROH (//)	tokenizer.GetPatternDe	invalid parser error
detailed string	buffer.Append(GetToken	scription(token);	type is used when the
representation of the	Description (elem. Id));	}	parser
alternative	} else {	}	* as such
*/		}	is invalid. This error
private string	buffer.Append(GetPatte	}	is typically caused by
ToString(ProductionPat	rn(elem.Id).Name);		
ternAlternative alt) {	}	/*	

* using a		*	public
· ·	/**		
parser without any	,	* @param type	ParserCreationExceptio
patterns.	* The	the parse error type	n(ErrorType type,
*/	inherent ambiguity	* @param info	C
	error type is used	the additional error	String name,
INVALID_PARSER,	when the set	information	
	* of	*/	String info,
/**	production patterns	public	
* The	(i.e. the grammar)	ParserCreationExceptio	ArrayList details) {
invalid token error	contains	n(ErrorType type,	
type is used when a	*		this. type
token	ambiguities that	String info)	= type;
* pattern	cannot be resolved.	:	this.name
is erroneous. This	*/	this(type, null, info)	= name;
error is typically		{	this, info
caused	INHERENT AMBIGUITY	}	= info;
* by an	}	,	, , , , , , , , , , , , , , , , , , ,
invalid pattern type	,	/**	this.details =
or an erroneous	/**	* Creates a	details;
regular	* The error	new parser creation	l l
1 egu1a1 *		•	J
	type.	exception.	/**
expression.	*/	*	,
*/	private	* @param type	* The error
T T	ErrorType type;	the parse error type	type property (read-
INVALID_TOKEN,	,	* @param name	only).
	/**	the token or	*
/**	* The token	production pattern	*
* The	or production pattern	name	*/
invalid production	name. This variable is	* @param info	public
error type is used	only	the additional error	ErrorType Type {
when a	* set for	information	get {
*	some error types.	*/	return
production pattern is	*/	public	type;
		D 0 11 D 11	1
erroneous. This error	private string	ParserCreationExceptio	}
erroneous. This error is	private string name;	n(ErrorType type,	}
			}
is *		n(ErrorType type,	} /**
is * typically caused by	name;		/**  * Returns the
is  * typically caused by referencing undeclared	name; /** * The	n(ErrorType type, String name,	* Returns the
is  * typically caused by referencing undeclared productions,	name; /**  * The additional error	n(ErrorType type,	,
<pre>is           * typically caused by referencing undeclared productions,           * or</pre>	name; /**  * The additional error information string.	n (ErrorType type,  String name,  String info)	* Returns the error type.
<pre></pre>	name;  /**  * The additional error information string. This variable is only	n(ErrorType type,  String name,  String info)  : this(type, name, info,	* Returns the error type.  *  * @return the
typically caused by referencing undeclared productions,  * or violating some other production pattern	name;  /**  * The additional error information string. This variable is only  * set for	n (ErrorType type,  String name,  String info)	* Returns the error type.
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.	name;  /**  * The additional error information string. This variable is only  * set for some error types.	n(ErrorType type,  String name,  String info)  : this(type, name, info,	* Returns the error type.  * @return the error type  *
typically caused by referencing undeclared productions,  * or violating some other production pattern	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */	<pre>n(ErrorType type,  String name,  String info)      :   this(type, name, info, null) {      } }</pre>	* Returns the error type.  *  * @return the
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string	<pre>n(ErrorType type,  String name,  String info)      :   this(type, name, info, null) {       }       /**</pre>	* Returns the error type.  * @return the error type  * @see #Type  *
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */	n(ErrorType type,  String name,  String info) : this(type, name, info, null) { }  /**  * Creates a	* Returns the error type.  *
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */ INVALID_PRODUCTION,	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;	n(ErrorType type,  String name,  String info) : this(type, name, info, null) { }  /**     * Creates a new parser creation	* Returns the error type.  *
<pre>typically caused by referencing undeclared productions,     * or violating some other production pattern constraint.     */ INVALID_PRODUCTION, /**</pre>	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;	n(ErrorType type,  String name,  String info) : this(type, name, info, null) { }  /**  * Creates a new parser creation exception.	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.
<pre>typically caused by referencing undeclared productions,     * or violating some other production pattern constraint.     */ INVALID_PRODUCTION,     /**     * The</pre>	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;  /**  * The error	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {       }      /**     * Creates a new parser creation exception.     *	* Returns the error type.  *
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error	name;  /**  * The  additional error  information string.  This variable is only  * set for  some error types.  */  private string  info;  /**  * The error  details list. This	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {       }  /**       * Creates a new parser creation exception.       *       * @param type	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;  /**  * The error details list. This variable is only set	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {       }        /**       * Creates a new parser creation exception.       *       * @param type the parse error type	* Returns the error type.  *  * @return the error type  *  * @see #Type  *  * @deprecated Use the Type property instead.  */ public ErrorType
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;  /**  * The error details list. This variable is only set for some	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  *  * @return the error type  *  * @see #Type  *  * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() {
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;  /**  * The error details list. This variable is only set	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  *  * @return the error type  *  * @see #Type  *  * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() { return
typically caused by referencing undeclared productions,	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;  /**  * The error details list. This variable is only set for some  * error types.	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  *  * @return the error type  *  * @see #Type  *  * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() {
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop has been detected in the grammar. One of	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */  private string info;  /**  * The error details list. This variable is only set for some  * error types.  */	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() { return
typically caused by referencing undeclared productions,	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */ private string info;  /**  * The error details list. This variable is only set for some  * error types.	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() { return
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop has been detected in the grammar. One of	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */  private string info;  /**  * The error details list. This variable is only set for some  * error types.  */	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() { return
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop has been detected in the grammar. One of the	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */  private string info;  /**  * The error details list. This variable is only set for some  * error types.  */  private	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() {     return Type; }
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop has been detected in the grammar. One of the  *	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */  private string info;  /**  * The error details list. This variable is only set for some  * error types.  */  private	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() {     return Type; }  /**
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop has been detected in the grammar. One of the  * productions in the	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */  private string info;  /**  * The error details list. This variable is only set for some  * error types.  */  private ArrayList details;	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() {     return Type; }  /**  * The token
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop has been detected in the grammar. One of the  * productions in the loop will be reported.	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */  private string info;  /**  * The error details list. This variable is only set for some  * error types.  */  private ArrayList details;	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() {     return Type; }  /**  * The token or production name
typically caused by referencing undeclared productions,  * or violating some other production pattern constraint.  */  INVALID_PRODUCTION,  /**  * The infinite loop error type is used when an infinite  * loop has been detected in the grammar. One of the  * productions in the loop will be reported.	name;  /**  * The additional error information string. This variable is only  * set for some error types.  */  private string info;  /**  * The error details list. This variable is only set for some  * error types.  */  private ArrayList details;  /**  * Creates a	n(ErrorType type,  String name,  String info) : this(type, name, info, null) {	* Returns the error type.  * * @return the error type  * * @see #Type  * * @deprecated Use the Type property instead.  */ public ErrorType GetErrorType() {     return Type; }  /**  * The token or production name property (read-only).

*/	public string	* @see	
public string	GetInfo() {	#Details	buffer.Append(name);
Name {	return	*	
get {	Info;	* @deprecated	buffer.Append("' is
return	}	Use the Details	invalid, as ");
name;		property instead.	
}	/**	*/	<pre>buffer. Append(info);</pre>
}	* The	public string	
	detailed error	<pre>GetDetails() {</pre>	break;
/**	information property	return	case
* Returns the	(read-only).	Details;	ErrorType.INFINITE_L00
token or production	*	}	P:
name.	*	,	
*	*/	/**	buffer. Append ("infinit
* @return the	public string	* The message	e loop found in
token or production	Details {	property (read-only).	production pattern
name	get {	This property contains	'");
*	Ctaria Duilden buffer	* the	h
* @see #Name *	StringBuilder buffer = new StringBuilder();	detailed exception	buffer.Append(name);
*  * @deprecated	- new Stringbullder();	error message. */	<pre>buffer. Append("'");</pre>
•	if	*/ public	buller. Append ( ),
Use the Name property instead.	(details == null) {	override string	break;
instead.	(details hull) (	Message {	case
public string	return null;	get {	ErrorType. INHERENT AMB
GetName() {	}	ger	IGUITY:
return	for	StringBuilder buffer	100111.
Name;	(int $i = 0$ ; $i <$	= new StringBuilder();	buffer. Append ("inheren
}	details.Count; i++) {	ne " seringsarraer (, ,	t ambiguity in
,	if	switch	production '");
/**	(i > 0) {	(type) {	•
* The		case	buffer. Append (name);
additional error	<pre>buffer.Append(", ");</pre>	ErrorType. INVALID PARS	
information property		ER:	<pre>buffer. Append("'");</pre>
(read-only).	if (i + 1 ==		if
*	details.Count) {	buffer.Append("parser	(info != null) {
*		is invalid, as ");	
*/	buffer.Append("and ");		buffer. Append (" $"$ );
public string		<pre>buffer. Append(info);</pre>	
Info {	}		<pre>buffer. Append(info);</pre>
get {	}	break;	}
return		case	if
info;	buffer.Append(details[	ErrorType. INVALID_TOKE	(details != null) {
}	i]);	N:	1 00 1 1/1
}	}	1 00 1 1/": 1	buffer. Append ("
/stests		<pre>buffer. Append("token '");</pre>	starting with ");
/** * Returns the	return	);	: £ (1-t-:1- Ct \ 1)
additional error	<pre>buffer. ToString();</pre>	buffer. Append (name);	if (details.Count > 1)
information.	}	burrer. Append (name),	l
*	J	buffer.Append("' is	buffer.Append("tokens
* @return the	/**	invalid, as ");	");
additional error	* Returns the	invalia, as /,	<i>)</i> ,
information	detailed error	<pre>buffer. Append(info);</pre>	} else {
*	information as a	sallatinppona (Into),	, 5155 (
* @see #Info	string	break;	buffer. Append ("token
*	*	case	");
* @deprecated	* @return the	ErrorType.INVALID_PROD	• •
Use the Info property	detailed error	UCTION:	}
instead.	information		•
*/	*	buffer.Append("product	buffer.Append(Details)
		ion '");	;

}	* parse errors	/**	}
	encountered while	* The error	
break;	parsing.	count property (read-	/**
	*	only).	* Returns a
default:		*	specific error from
	*	*	the log.
buffer.Append("interna	* @since 1.1	*/	*
l error");	*/	public int	* @param
	public class	Count {	index the
break;	ParserLogException:	get {	error index, 0 <=
}	Exception {	return	index < count
return		errors.Count;	*
<pre>buffer.ToString();</pre>	/**	}	* @return the
}	* The list of	}	parse error requested
}	errors found.		*
	*/	/**	* @deprecated
/**	private	* Returns the	Use the class indexer
* Returns the	ArrayList errors = new	number of errors in	instead.
error message. This	ArrayList();	this log.	*/
message will contain		*	public
all the	/**	* @return the	ParseException
* information	* Creates a	number of errors in	GetError(int index) {
available.	new empty parser log	this log	return
*	exception.	*	this[index];
* @return the	*/	* @see #Count	}
error message	public	*	
*	ParserLogException() {	* @deprecated	/**
* @see	}	Use the Count property	* Adds a
#Message		instead.	parse error to the
*	/**	*/	log.
* @deprecated	* The message	public int	*
Use the Message	property (read-only).	GetErrorCount() {	* @param e
property instead.	This property contains	return	the parse error to add
*/	* the	Count;	*/
public string	detailed exception	}	public void
GetMessage() {	error message.	/state	AddError(ParseExceptio
return	*/ public	/**	n e) {
Message;		* The error index (read-only).	amana Add(a).
}	override string Message {	This index contains	errors. Add (e);
}	message { get{	all the	ſ
J	get(	* errors in	/**
/*	StringBuilder buffer	this error log.	* Returns the
*	= new StringBuilder();	*	detailed error
ParserLogException.cs	new seringsdrider () ,	* @param	message. This message
*/	for	index the	will contain
,	(int $i = 0$ ; $i < Count$ ;	error index, 0 <=	* the error
using System;	i++) {	index < Count	messages from all
using	if	*	errors in this log,
System. Collections;	$(i > 0)$ {	* @return the	separated by
using System.Text;		parse error requested	* a newline.
	<pre>buffer.Append("\n");</pre>	*	*
namespace Core.Library	}	*	* @return the
{		*/	detailed error message
	buffer.Append(this[i].	public	*
/**	Message);	ParseException	* @see
* A parser log	}	this[int index] {	#Message
exception. This class	return	get {	*
contains a list of all	<pre>buffer.ToString();</pre>	return	* @deprecated
the	}	(ParseException)	Use the Message
	}	errors[index];	property instead.
		}	*/

public string	++,, (, ), abort,	string unitAID	string
GetMessage() {	}, \$";	= "=, unit, digit,	functParam = "(, ;, +,
return	string	company, joe,	-, *, /, %, ^";
Message;	localdec = "=, ,, ;";	response, ,, ;, {, ),	string
}	string	} ";	functIDParam =
}	UnitaddID = "=, ,, ;";	string	"Numlit, Declit,
}	string	unitAIDTWO = "=, ,, ;,	Stringlit, Charlit,
	UnitEXinit = ",, ;";	unit, digit, company,	id, AFFIRMATIVE,
using System;	string main =	joe, response, =, )";	NEGATIVE, )";
using	"PrimaryMission";	string	string
System. Collections. Gen	string	unitElem = "Numlit,	addfunctIDParam = ",,
eric;	globalDec = "unit,	Declit, Stringlit,	)";
using System.Ling;	digit, company, joe,	Charlit, id,	string funct =
using System. Text;	response, struct,	AFFIRMATIVE,	"unit, digit, company,
using bystem. Text,	PrimaryMission, unit,	NEGATIVE";	joe, response, miss";
System. Threading. Tasks	digit, company, joe,	string EXTelem	string
		= "Numlit, Declit,	functReturn = "unit,
,	response, struct,		,
0 1.1	comment";	Stringlit, Charlit,	digit, company, joe,
namespace Core.Library	string	id, AFFIRMATIVE,	response";
11: 1	localdecChoice = ",	NEGATIVE";	string
public class	unit, digit, company,	string	<pre>functVoid = "miss";</pre>
PredictSets	joe, response,	EXTelemChoice = ",,	string dtypeA
{	struct,";	} ";	= "unit, digit,
string program	string	string	response, id, )";
= "hold, unit, digit,	<pre>decChoice = "unit,</pre>	unitElemTwo = "{";	string
company, joe,	digit, company, joe,	string	EXdtypeA = ",, )";
response, struct,	response, ), ,";	<pre>ElemTwoLit = ",, },</pre>	string dtypef
PrimaryMission";	string	=";	= "unit, digit,
string	<pre>globalChoice = "=, ,,</pre>	string	response, id, backup,
comments = "comment,	(, ";	<pre>ElemTwoTail = ",, }";</pre>	++,, }, backup";
hold, PrimaryMission,	string	string	string ExID =
unit, digit, company,	BodyChoice = "=, ,";	assignChoice = "id,	", ";
joe, response, struct,	string	++,, }, backup,	string
}, post, inquire, go,	varUnitBody = "=, ,";	abort, \$, comment,	arrIndex = "";
campaign, comment,	string	post, inquire, go,	string
capture, phase,	functReturnBody = "(";	campaign, capture,	struct_U = "struct";
inorder, id, ++,,	string	phase, inorder, id,	string sDec =
backup, abort, \$, (,	functVoidBody = "(, ;,	++,";	"unit, digit, company,
)";	numlit, declit,	string	joe, response, }, +, -
string	stringlit, charlit";	AccessAssignDtype =	, *, /, =, (, ., )'';
datatype = "unit,	stringirt, charit , string	"id";	string index =
	arrUnitBody = "":		"[";
digit, company, joe,	, ,	string	
response";	string arrType = "":	assignValueChoice =	string body =
string	,	"=, ., {, +, -, *, /,	"post, capture,
Literals = "Numlit,	string N1 =	%, ^, ++,";	inquire, inorder, go,
Declit, Stringlit,	"";	string	phase, campaign, id,
Charlit, id,	string "	assigning = $''$ =, {, +,	++,, comment, post,
AFFIRMATIVE,	ArrayChoice = "=, ,,	-, <b>*</b> , /, %, ^, ., ++,	inquire, go, campaign,
		":	
NEGATIVE";	;, unit, digit,	•	capture, phase,
string	company, joe,	string ArrayID	capture, phase, inorder, id, ++,,
•	., , , ,	•	
<pre>string constant = "hold, comment,</pre>	company, joe, response, )"; string N2 =	string ArrayID = "{"; string	inorder, id, ++,,
string constant = "hold,	company, joe, response, )";	string ArrayID = "{";	<pre>inorder, id, ++,, }, backup, abort, \$,</pre>
<pre>string constant = "hold, comment,</pre>	company, joe, response, )"; string N2 =	string ArrayID = "{"; string ArrayIDTail = "="; string	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post,</pre>
string constant = "hold, comment, PrimaryMission, unit,	company, joe, response, )"; string N2 = "";	string ArrayID = "{"; string ArrayIDTail = "=";	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign,</pre>
string constant = "hold, comment, PrimaryMission, unit, digit, company, joe,	company, joe, response, )"; string N2 = ""; string index1	string ArrayID = "{"; string ArrayIDTail = "="; string	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign, capture, phase,</pre>
string constant = "hold, comment, PrimaryMission, unit, digit, company, joe, response, struct";	<pre>company, joe, response, )";           string N2 = "";           string index1 = "Numlit, id";</pre>	string ArrayID = "{";  string ArrayIDTail = "=";  string AssignSym = "+, -, *,	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign, capture, phase, inorder, id, ++,";</pre>
string constant = "hold, comment, PrimaryMission, unit, digit, company, joe, response, struct"; string	<pre>company, joe, response, )";</pre>	string ArrayID  = "{";  string  ArrayIDTail = "=";  string  AssignSym = "+, -, *, /, %, ^";	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign, capture, phase, inorder, id, ++,"; string print =</pre>
string constant = "hold, comment, PrimaryMission, unit, digit, company, joe, response, struct"; string localChoice = "unit, digit, company, joe,	<pre>company, joe, response, )"; string N2 = ""; string index1 = "Numlit, id"; string add = "+"; string index2</pre>	string ArrayID  = "{";  string  ArrayIDTail = "=";  string  AssignSym = "+, -, *, /, %, ^";  string  assignValue = "Numlit,	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign, capture, phase, inorder, id, ++,";</pre>
string constant = "hold, comment, PrimaryMission, unit, digit, company, joe, response, struct"; string localChoice = "unit, digit, company, joe, response, post,	<pre>company, joe, response, )";     string N2 = "";     string index1 = "Numlit, id";     string add = "+";     string index2 = "Numlit, id";</pre>	string ArrayID  = "{";  string  ArrayIDTail = "=";  string  AssignSym = "+, -, *,  /, %, ^";  string  assignValue = "Numlit,  Declit, Stringlit,	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign, capture, phase, inorder, id, ++,";</pre>
string constant = "hold, comment, PrimaryMission, unit, digit, company, joe, response, struct"; string localChoice = "unit, digit, company, joe, response, post, inquire, go, campaign,	<pre>company, joe, response, )"; string N2 = ""; string index1 = "Numlit, id"; string add = "+"; string index2 = "Numlit, id"; string indexEX</pre>	string ArrayID  = "{";  string  ArrayIDTail = "=";  string  AssignSym = "+, -, *,  /, %, ^";  string  assignValue = "Numlit,  Declit, Stringlit,  Charlit, id,	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign, capture, phase, inorder, id, ++,";</pre>
string constant = "hold, comment, PrimaryMission, unit, digit, company, joe, response, struct"; string localChoice = "unit, digit, company, joe, response, post,	<pre>company, joe, response, )";     string N2 = "";     string index1 = "Numlit, id";     string add = "+";     string index2 = "Numlit, id";</pre>	string ArrayID  = "{";  string  ArrayIDTail = "=";  string  AssignSym = "+, -, *,  /, %, ^";  string  assignValue = "Numlit,  Declit, Stringlit,	<pre>inorder, id, ++,, }, backup, abort, \$, comment, post, inquire, go, campaign, capture, phase, inorder, id, ++,";</pre>

string	post, inquire, go,	digit, company, joe,	string LogOp =
ConcatLit = ", , )";	campaign, capture,	response, post,	"(";
string scan =	phase, inorder, ++,	inquire, go, campaign,	string
"capture";	";	comment, capture,	ExtLogOp = "  , &";
string ExtI =	string	phase, inorder, id,	string LogOper
", , )";	elseifstatement =	++, , abort, }";	= "  , &";
string	"unit, digit, company,	string MathOp	string end =
for_state = "inquire";	joe, response, post,	= "(, Numlit, Declit,	"}";
string	inquire, go, campaign,	Stringlit, Charlit,	string
forstatement = "unit,	comment, capture,	id, AFFIRMATIVE,	StartProgram =
digit, company, joe,	phase, inorder, id,	NEGATIVE";	"comment, hold,
response, post,	++,";	string	PrimaryMission, unit,
inquire, go, campaign,	string	operCond = "(, Numlit,	digit, company, joe,
comment, capture,	else_state = "order,	Declit, Stringlit,	response, struct";
phase, inorder, id,	post, inquire, go,	Charlit, id,	
++,, }";	campaign, comment,	AFFIRMATIVE,	public string
string val1 =	capture, phase,	NEGATIVE";	GetPredictSet(int
"Numlit, 0";	inorder, id, ++,,	string	code)
string mntCond	}, backup, abort, \$";	operCondChoice = "+, -	{
= "++,, +, -, *, /,	string	, *, /, %, ^, =";	switch
>, <, >=, <=, ==,	elsestatement = "unit,	string operSym	(code)
numlit, declit,	digit, company, joe,	= "+, -, *, /, %, ^";	{
stringlit, charlit,	response, post,	string operEq	case
AFFIRMATIVE,	inquire, go, campaign,	= "+=, , -=, *=, /=,	2001: return
NEGATIVE";	comment, capture,	%= <b>,</b> =";	StartProgram;
string	phase, inorder, id,	string	case
mntCondT = "++,, +,	++,, }";	operExt_s = "Numlit,	2002: return program;
-, <b>*</b> , /, >, <, >=, <=,	string dowhile	Declit, Stringlit,	case
==, numlit, declit,	= "go";	Charlit, id,	2003: return comments;
stringlit, charlit,	string	AFFIRMATIVE, NEGATIVE,	case
AFFIRMATIVE,	dostatement = "unit,	(";	2004: return datatype;
NEGATIVE";	digit, company, joe,	string	case
string mnt =	response, post,	operExt_rep = "+, -,	2005: return Literals;
"++,, +, -, *, /,	inquire, go, campaign,	*, /, %, ^, ), ;";	case
>, <, >=, <=, ==,	comment, capture,	string operand	2006: return constant;
numlit, declit,	phase, inorder, id,	= "Numlit, Declit,	case
stringlit, charlit,	++,, }";	Stringlit, Charlit,	2007: return
AFFIRMATIVE,	string	id, AFFIRMATIVE,	localChoice;
NEGATIVE";	while_state = "phase,	NEGATIVE";	case
string ifelse	}, backup, abort, \$,	string	2008: return localdec;
= "inorder";	comment, post,	simMathOp = "Numlit,	case
string	inquire, go, campaign,	Declit, Stringlit,	2009: return
ifcondition = "Numlit,	capture, phase,	Charlit, id,	UnitaddID;
Declit, Stringlit,	inorder, id, ++,";	AFFIRMATIVE,	case
Charlit, id,	string	NEGATIVE";	2010: return
AFFIRMATIVE, NEGATIVE,	whilestatement =	string	UnitEXinit;
(";	"unit, digit, company,	$S_MathExt = "+, -, *,$	case
string	joe, response, post,	/, %, ^, ), ;";	2011: return main;
ifstatement = "unit,	inquire, go, campaign,	string operCondExt = "+, -,	case
digit, company, joe,	comment, capture,	* '	2012: return
response, post, inquire, go, campaign,	phase, inorder, id, ++,, }";	*, /, %, ^, ;"; string RelOp =	globalDec;
		0 1	case
comment, capture,	string	"Numlit, Declit,	2013: return
phase, inorder, id, ++,, }";	<pre>switch_state = "campaign, abort";</pre>	Stringlit, Charlit, id, AFFIRMATIVE,	localdecChoice;
string elseif	campaign, abort ; string	NEGATIVE";	case 2014: return
= "otherorder, Numlit,	case state =	string	decChoice;
Declit, Stringlit,	<pre>case_state - "operation";</pre>	RelopExt = "==, !=,	·
Charlit, id,	string def =	\\ \rightarrow = \langle - \cdot - \cd	case 2015: return
AFFIRMATIVE, NEGATIVE,	"DEFAULT, }";	/-, \-,      string op1 =	globalChoice;
(, order, }, backup,	string	"==, !=, >=, <=, >,	grobaronorde,
abort, \$, comment,	casestatement = "unit,	, :-, /-, \-, /, <";	
ασστι, ψ, comment,	cases ta coment unit,	` ,	

case	case	case	case
2016: return	2040: return	2065: return	2089: return operSym;
BodyChoice;	assigning;	for_state;	case
case	case	case	2090: return operEq;
2017: return	2041: return ArrayID;	2066: return	case
varUnitBody;	case	forstatement;	2091: return
•			
case	2042: return	case	operExt_s;
2018: return	ArrayIDTail;	2067: return val1;	case
functReturnBody;	case	case	2092: return
case	2043: return	2068: return mntCond;	operExt_rep;
2019: return	AssignSym;	case	case
functVoidBody;	case	2069: return mntCondT;	2093: return operand;
case	2044: return	case	case
2020: return	assignValue;	2070: return mnt;	2094: return
arrUnitBody;	case	case	simMathOp;
case	2045: return	2071: return ifelse;	case
2021: return arrType;	functParam;	case	2095: return
case	case	2072: return	S_MathExt;
2022: return N1;	2046: return	ifcondition;	case
case	<pre>functIDParam;</pre>	case	2096: return
2023: return	case	2073: return	operCondExt;
ArrayChoice;	2047: return	ifstatement;	case
case	addfunctIDParam;	case	2097: return RelOp;
2024: return N2;	case	2074: return elseif;	case
case	2048: return funct;	case	2098: return RelopExt;
2025: return index1;	case	2075: return	case
case	2049: return	elseifstatement;	2099: return op1;
2026: return add;	functReturn;	·	* *
		case	case
case	case	2076: return	2100: return LogOp;
2027: return index2;	2050: return	else_state;	case
case	functVoid;	case	2101: return ExtLogOp;
2028: return indexEX;	case	2077: return	case
case	2051: return dtypeA;	elsestatement;	2102: return LogOper;
2029: return unitAID;	case	case	case
case	2052: return EXdtypeA;	2078: return dowhile;	2103: return end;
2030: return	case	case	}
unitAIDTWO;	2053: return dtypef;	2079: return	return "";
case	case	dostatement;	}
2031: return unitElem;	2054: return ExID;	case	}
case	case	2080: return	
2032: return EXTelem;	2055: return arrIndex;	while_state;	}
case	case	case	
2033: return	2056: return struct U;	2081: return	/*
EXTelemChoice;	case	whilestatement;	* Production.cs
case	2057: return sDec;	case	*/
2034: return	case	2082: return	
unitElemTwo;	2058: return index;	switch_state;	using
case	case	case	System. Collections;
2035: return	2059: return body;	2083: return	Systems corrections,
ElemTwoLit;	case	case_state;	namespace Core.Library
·	2060: return print;		f amespace core. Eibrary
case 2036: return	• •	case	l .
	case	2084: return def;	/ starte
ElemTwoTail;	2061: return postval;	case	/**
case	case	2085: return	* A production
2037: return	2062: return	casestatement;	node. This class
assignChoice;	ConcatLit;	case	represents a grammar
case	case	2086: return MathOp;	production
2038: return	2063: return scan;	case	* (i.e. a list of
AccessAssignDtype;	case	2087: return operCond;	child nodes) in a
case	2064: return ExtI;	case	parse tree. The
2039: return		2088: return	productions
assignValueChoice;		operCondChoice;	

* are created by	*	if	* @return the
a parser, that adds	*/	(index < 0    index >=	production pattern
children a according	public	children.Count) {	*
to a	override int Id {		* @see
* set of	get {	return null;	#Pattern
production patterns	return	} else	*
(i.e. grammar rules).	pattern. Id;	{	* @deprecated
*	}		Use the Pattern
	}	return (Node)	property instead.
*		children[index];	*/
*/	/**	}	public
public class	* The node	}	ProductionPattern
Production : Node {	name property (read-	}	GetPattern() {
	only).		return
/**	*	/**	Pattern;
* The	*	* Adds a	}
production pattern	*/	child node. The node	,
used for this	public	will be added last in	/**
production.	override string Name {	the list of	* Checks if
*/	get {	* children.	this node is hidden.
private	return	*	i.e. if it should not
ProductionPattern	pattern. Name;		be visible
	pattern. Name,	* @param child the	* outside the
pattern;	) 1		
/	}	child node to add	parser.
/**	/	*/	*
* The child	/**	public void	* @return
nodes.	* The child	AddChild(Node child) {	true if the node
*/	node count property	if (child	should be hidden, or
private	(read-only).	!= null) {	*
ArrayList children;	*		false otherwise
	*	child.SetParent(this);	*/
/**	*/		internal
* Creates a	public	children.Add(child);	override bool
new production node.	override int Count {	}	IsHidden() {
*	get {	}	return
* @param	return		pattern.Synthetic;
pattern the	children.Count;	/**	}
production pattern	}	* The	
*/	}	production pattern	/**
public		property (read-only).	* Returns a
Production(ProductionP	/**	This property	string representation
attern pattern) {	* The child	* contains	of this production.
	node index (read-	the production pattern	*
this.pattern =	only).	linked to this	* @return a
pattern;	*	production.	string representation
_	* @param	*	of this production
this.children = new	index the	*	*/
ArrayList();	child index, 0 <=	*/	public
}	index < Count	public	override string
,	*	ProductionPattern	ToString() {
/**	* @return the	Pattern {	return
* The node	child node found, or	get {	pattern. Name + '(' +
type id property	*	return	pattern. Id + ')';
(read-only). This	null if index out of		pattern.iu / / ,
* * * * * * * * * * * * * * * * * * * *	hull II index out of bounds	pattern;	}
value is set as		) l	}
* a unique	*	Ĵ	l
identifier for each	*		/
type of node, in order	*/	/**	/*
to	public	* Returns the	*
* simplify	override Node this[int	production pattern for	ProductionPattern.cs
later identification.	index] {	this production.	*/
*	get {	*	

using	private bool		}
System. Collections;	synthetic;	this.alternatives =	,
using System. Text;	Synthetic,	new ArrayList();	/**
doing by brome force,	/**	now initagelist (),	* Returns the
namespace Core.Library	* The list of	this. defaultAlt = $-1$ ;	production pattern
{	production pattern		name.
	alternatives.	this.lookAhead = null;	*
/**	*/	}	* @return the
* A production	private	·	production pattern
pattern. This class	ArrayList	/**	name
represents a set of	alternatives;	* The	*
production		production pattern	* @see #Name
* alternatives	/**	identity property	*
that together forms a	* The default	(read-only). This	* @deprecated
single production. A	production pattern	* property	Use the Name property
* production	alternative. This	contains the unique	instead.
pattern is identified	alternative	identity value.	*/
by an integer id and a	* is used	*	public string
name,	when no other	*	GetName() {
* both provided	alternatives match. It	*/	return
upon creation. The	may be set to	public int Id	Name;
pattern id is used for	∗ -1, meaning	{	}
* referencing the	that there is no	get {	
production pattern	default (or fallback)	return	/**
from production	alternative.	id;	* The
pattern	*/	}	synthetic production
* elements.	private int	}	pattern property. If
*	defaultAlt;	,	this property
		/**	* is set, the
*	/**	* Returns the	production identified
*/	* The look-	unique production	by this pattern has
public class	ahead set associated	pattern identity	been
ProductionPattern {	with this pattern.	value.	*
/**	*/	*	artificially inserted
/** * The	private LookAheadSet	* @return the	into the grammar. No
production pattern	lookAhead;	production pattern id *	parse tree nodes * will be
identity.	100kAlleau,	* @see #Id	created for such
*/	/**	* esee #1u	nodes, instead the
private int	* Creates a	* @deprecated	child nodes
id;	new production	Use the Id property	* will be
14,	pattern.	instead.	added directly to the
/**	*	*/	parent node. By
* The	* @param id	public int	default this
production pattern	the production pattern	GetId() {	* property is
name.	id	return Id;	set to false.
*/	* @param name	}	*
private string	the production pattern		*
name;	name	/**	*/
	*/	* The	public bool
/**	public	production pattern	Synthetic {
* The	ProductionPattern(int	name property (read-	get {
synthectic production	id, string name) {	only).	return
flag. If this flag is	this.id =	*	synthetic;
set, the	id;	*	}
* production	this.name	*/	set {
identified by this	= name;	public string	
pattern has been		Name {	<pre>synthetic = value;</pre>
artificially	this.synthetic =	get {	}
* inserted	false;	return	}
into the grammar.		name;	
*/		}	/**

* Checks if	*		* @deprecated
the synthetic	* @deprecated	object obj =	Use the Count property
•	*	alternatives default Al	
production flag is set. If this	Use the Synthetic	t];	instead. */
* flag is	property instead. */	ι,,	public int
	,		
set, the production	public void	return	GetAlternativeCount()
identified by this	SetSyntetic (bool	(ProductionPatternAlte	1
pattern has	synthetic) {	rnative) obj;	return
* been	Synthetic	} else	Count;
artificially inserted	= synthetic;	ĺ	}
into the grammar. No	}	. 11	/
parse tree	/	return null;	/**
* nodes will	/**	}	* The
be created for such	* The look-	}	production pattern
nodes, instead the	ahead set property.	set {	alternative index
child	This property contains	1 0 1 11 0	(read-only).
* nodes will	the	defaultAlt = 0;	*
be added directly to	* look-ahead	for	* @param
the parent node.	set associated with	(int $i = 0$ ; $i <$	index the
*	this alternative.	alternatives.Count;	alternative index, 0
* @return	*/	i++) {	<= pos < Count
true if this	internal	if	*
production pattern is	LookAheadSet LookAhead	(alternatives[i] ==	* @return the
synthetic, or	{	value) {	alternative found
*	get {		*
false otherwise	return	<pre>defaultAlt = i;</pre>	*
*	lookAhead;	}	*/
* @see	}	}	public
#Synthetic	set {	}	ProductionPatternAlter
*		}	<pre>native this[int index]</pre>
* @deprecated	lookAhead = value;		{
			_
Use the Synthetic	}	/**	get {
Use the Synthetic property instead.	}	/** * The	get { return
·	}	,	_
property instead.	} /*c*	* The	return
property instead. */	} /** * The default	* The production pattern	return (ProductionPatternAlte
property instead.  */  public bool	<b>'</b>	* The production pattern alternative count	return (ProductionPatternAlte rnative)
property instead.  */  public bool IsSyntetic() {	* The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative)
property instead.  */  public bool IsSyntetic() {  return	* The default pattern alternative	* The production pattern alternative count property	return (ProductionPatternAlte rnative)
property instead.  */  public bool IsSyntetic() {  return	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative)
property instead.  */  public bool IsSyntetic() {  return	* The default pattern alternative property. The default * alternative	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }
<pre>property instead.     */     public bool IsSyntetic() {     return Synthetic; }</pre>	* The default pattern alternative property. The default * alternative is used when no other	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; } /** * Returns an
<pre>property instead.     */     public bool IsSyntetic() {     return Synthetic;     }     /**     * Sets the</pre>	* The default pattern alternative property. The default * alternative is used when no other alternative matches.	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**  * Sets the synthetic production	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; } /** * Returns an
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  *
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  *  * @param pos
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the	* The default pattern alternative property. The default     * alternative is used when no other alternative matches. The     * default alternative must previously have been added to the	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  * @param pos the alternative
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  * @param pos the alternative position, 0 <= pos <
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  * @param pos the alternative position, 0 <= pos < count
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,          * the production identified by this pattern has been artificially	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  * @param pos the alternative position, 0 <= pos < count  *
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,          * the production identified by this pattern has been artificially     * inserted	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  * @param pos the alternative position, 0 <= pos < count  *  * @return the
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index]; }  /**  * Returns an alternative in this pattern.  * @param pos the alternative position, 0 <= pos < count  *  * @return the alternative found
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is	* The default pattern alternative property. The default     * alternative is used when no other alternative matches. The     * default alternative must previously have been added to the     * list of alternatives. This property is set to null if no     * default pattern alternative	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to     * false.	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to     * false.     *	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to      * false.     *     * @param	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to     * false.     *     * @param syntetic the new	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to     * false.     *     * @param syntetic the new value of the synthetic	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to     * false.     *     * @param syntetic the new value of the synthetic flag	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**  * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to     * false.     *  * @param syntetic the new value of the synthetic flag  *	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];
property instead.  */ public bool IsSyntetic() {     return Synthetic; }  /**     * Sets the synthetic production pattern flag. If this flag is set,     * the production identified by this pattern has been artificially     * inserted into the grammar. By default this flag is set to     * false.     *     * @param syntetic the new value of the synthetic flag	* The default pattern alternative property. The default	* The production pattern alternative count property	return (ProductionPatternAlte rnative) alternatives[index];

return	alternative is right	alt =	*
this[pos];	recursive, or	(ProductionPatternAlte	* @return a
; }	*	rnative)	token string
J	false otherwise	alternatives[i];	representation
/**	*/	if	*/
* Checks if	public bool	(alt.IsMatchingEmpty()	public
this pattern is	IsRightRecursive() {	) {	override string
recursive on the left-	13K1ghtkeed131ve()	, (	ToString() {
hand side.	ProductionPatternAlter	return true;	Tooting()
* This method	native alt;	}	StringBuilder buffer
checks if any of the	navivo aiv,	}	= new StringBuilder();
production pattern	for (int i	return	new StringBarraci (),
*	= 0; i <	false;	StringBuilder indent
alternatives is left-	alternatives.Count;	}	= new StringBuilder();
recursive.	i++) {	,	int
*	alt =	/**	i;
* @return	(ProductionPatternAlte	* Adds a	-,
true if at least one	rnative)	production pattern	
alternative is left	alternatives[i];	alternative.	buffer. Append (name);
recursive, or	if	*	
*	(alt.IsRightRecursive(	* @param alt	<pre>buffer. Append("(");</pre>
false otherwise	)) {	the production pattern	
*/	,, (	alternative to add	buffer. Append(id);
public bool	return true;	*	, , , , , , , , , , , , , , , , , , ,
IsLeftRecursive() {	}	* @throws	<pre>buffer.Append(") ");</pre>
ibber (Meedi bive ()	}	ParserCreationExceptio	for (i =
ProductionPatternAlter	return	n if an identical	0; i < buffer.Length;
native alt;	false;	alternative has	i++) {
,	}	*	- / (
for (int i	,	already been added	<pre>indent.Append(" ");</pre>
= 0; i <	/**	*/	}
alternatives.Count;	* Checks if	public void	for (i =
i++) {	this pattern would	AddAlternative(Product	0; i <
alt =	match an empty stream	ionPatternAlternative	alternatives.Count;
(ProductionPatternAlte	of	alt) {	i++) {
rnative)	* tokens.	if	if (i
alternatives[i];	This method checks if	(alternatives.Contains	== 0) {
if	any one of the	(alt)) {	
(alt. IsLeftRecursive()	production	throw	<pre>buffer. Append("= ");</pre>
) {	* pattern	new	} else
	alternatives would	ParserCreationExceptio	{
return true;	match the empty token	n (	
}	stream.		<pre>buffer. Append("\n");</pre>
}	*	ParserCreationExceptio	
return	* @return	n.ErrorType.INVALID_PR	<pre>buffer. Append(indent);</pre>
false;	true if at least one	ODUCTION,	
}	alternative match no		<pre>buffer. Append("  ");</pre>
	tokens, or	name,	}
/**	*		
* Checks if	false otherwise	"two identical	buffer. Append (alternat
this pattern is	*/	alternatives exist");	ives[i]);
recursive on the	public bool	}	}
right-hand side.	<pre>IsMatchingEmpty() {</pre>		return
* This method		<pre>alt.SetPattern(this);</pre>	<pre>buffer.ToString();</pre>
checks if any of the	ProductionPatternAlter		}
production pattern	native alt;	alternatives.Add(alt);	}
*		}	}
alternatives is right-	for (int i		
recursive.	= 0; i <	/**	/*
*	alternatives.Count;	* Returns a	*
* @return	i++) {	string representation	ProductionPatternAlter
true if at least one		of this object.	native.cs

*/	private	This property contains	* @param
	LookAheadSet lookAhead	the	index the
using System;	= null;	* look-ahead	element index, 0 <=
using		set associated with	pos < Count
System.Collections;	/**	this alternative.	*
using System.Text;	* Creates a	*/	* @return the
	new production pattern	internal	element found
namespace Core.Library	alternative.	LookAheadSet LookAhead	*
{	*/	{	*
	public	get {	*/
/**	ProductionPatternAlter	return	public
* A production	native() {	lookAhead;	ProductionPatternEleme
pattern alternative.	}	}	nt this[int index] {
This class represents	Atot	set {	get {
a list of	/**	1 1 1 1 - 1 - 1 - 1	return
* production	* The	lookAhead = value;	(ProductionPatternElem
pattern elements. In	production pattern	) 1	ent) elements[index];
order to provide productions that	property (read-only).	ı	}
* cannot be	This property * contains	/**	J
represented with the	the pattern having	* The	/**
element occurance	this alternative.	production pattern	* Returns an
counters, multiple	this afternative.	element count property	element in this
* alternatives	*	(read-only).	alternative.
must be created and	*/	(read only).	*
added to the same	public	*	* @param pos
production	ProductionPattern	*/	the element position,
* pattern. A	Pattern {	public int	0 <= pos < count
production pattern	get {	Count {	*
alternative is always	return	get {	* @return the
contained	pattern;	return	element found
* within a	}	elements.Count;	*
production pattern.	}	}	* @deprecated
*	•	}	Use the class indexer
	/**		instead.
*	* Returns the	/**	*/
*/	production pattern	* Returns the	public
public class	containing this	number of elements in	ProductionPatternEleme
ProductionPatternAlter	alternative.	this alternative.	nt GetElement(int pos)
native {	*	*	{
	* @return the	* @return the	return
/**	production pattern for	number of elements in	this[pos];
* The	this alternative	this alternative	}
production pattern.	*	*	
*/	* @see	* @see #Count	/**
private	#Pattern	*	* Checks if
ProductionPattern	*	* @deprecated	this alternative is
pattern;	* @deprecated	Use the Count property	recursive on the left-
/	Use the Pattern	instead.	hand
/**	property instead.	*/	* side. This
* The element	*/	public int	method checks all the
list.	public	GetElementCount() {	possible left side
*/	ProductionPattern GetPattern() {	return	* elements
private ArravList elements =	v (	Count;	and returns true if
new ArrayList();	return Pattern;	ſ	the pattern itself is
new AllayLISt(),	raccern, }	/**	among * them.
/**	J	/** * The	* them.
/** * The look-	/**	* The production pattern	* * @return
ahead set associated	* The look-	element index (read-	true if the
with this alternative.	ahead set property.	only).	alternative is left
	ancaa see property.	JIII ) / ·	arcomactve 15 1016
*/		*	side recursive, or

*	elem =	internal void	public int
false otherwise	(ProductionPatternElem	SetPattern (ProductionP	GetMaxElementCount() {
*/	ent) elements[i];	attern pattern) {	octmaxbrementeount ()
public bool	if	attern pattern, (	ProductionPatternEleme
IsLeftRecursive() {	(elem.Id ==	this.pattern =	nt elem;
isher theedisive ()	pattern. Id) {	pattern;	int
ProductionPatternEleme	pattern. ruy	pattern,	$\max = 0$ ;
nt elem;	return true;	J	max - 0,
it ciem,	} else	/**	for (int i
for (int i	if (elem. MinCount > 0)	* Returns the	= 0; i <
= 0; i <	{	minimum number of	elements.Count; i++) {
elements.Count; i++) {	(	elements needed to	elem =
elem =	break;	satisfy	(ProductionPatternElem
(ProductionPatternElem	break,	* this	ent) elements[i];
ent) elements[i];	}	alternative. The value	if
if	return	returned is the sum of	(elem. MaxCount >=
(elem. Id ==	false;	all the	Int32. MaxValue) {
pattern. Id) {	1415C,	* elements	intoz. maxvarue)
partern. ruy	J	minimum count.	return Int32.MaxValue;
return true;	/**	*	} else
} else	* Checks if	* @return the	) C13C
if (elem. MinCount > 0)	this alternative would	minimum number of	(
f (etem. wincount / 0)	match an empty stream	elements	max += elem.MaxCount;
(	of	*/	max - elem maxcount,
break:	* tokens.	public int	,
break,	This check is	GetMinElementCount() {	return
}	equivalent of	GetwinElementCount() (	
return	getMinElementCount()	ProductionPatternEleme	max;
false;	* returning	nt elem;	J
laise,	zero (0).	int elem,	/**
J	zero (0). *	$\min = 0$ ;	* Adds a
/**	* @return	min – O,	token to this
,		for (int i	
* Checks if	true if the rule can	for (int i	alternative. The token
* Checks if this alternative is	true if the rule can match an empty token	= 0; i <	alternative. The token is appended to
* Checks if this alternative is recursive on the	true if the rule can	= 0; i < elements.Count; i++) {	alternative. The token is appended to * the end of
* Checks if this alternative is recursive on the right-hand side.	true if the rule can match an empty token stream, or *	= 0; i < elements.Count; i++) { elem =	alternative. The token is appended to * the end of the element list. The
* Checks if this alternative is recursive on the right-hand side.	true if the rule can match an empty token stream, or * false otherwise	<pre>= 0; i &lt; elements.Count; i++) {      elem = (ProductionPatternElem</pre>	alternative. The token is appended to * the end of the element list. The multiplicity values
* Checks if this alternative is recursive on the right-hand side.	true if the rule can match an empty token stream, or  * false otherwise  */	<pre>= 0; i &lt; elements.Count; i++) {     elem =     (ProductionPatternElem ent) elements[i];</pre>	alternative. The token is appended to
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side	true if the rule can match an empty token stream, or  * false otherwise  */ public bool	<pre>= 0; i &lt; elements.Count; i++) {     elem =     (ProductionPatternElem ent) elements[i];     min +=</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {	<pre>= 0; i &lt; elements.Count; i++) {     elem =     (ProductionPatternElem ent) elements[i];</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required,
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() { return	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and
* Checks if this alternative is recursive on the right-hand side.           * This method checks all the possible right side elements and           * returns true if the pattern	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() { return GetMinElementCount()	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() { return	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.
* Checks if this alternative is recursive on the right-hand side.     * This method checks all the possible right side elements and     * returns true if the pattern itself is among them.     *	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() { return GetMinElementCount()	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return	<pre>true if the rule can match an empty token stream, or</pre>	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the	<pre>true if the rule can match an empty token stream, or</pre>	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *  * @param id the token (pattern) id
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right	<pre>true if the rule can match an empty token stream, or</pre>	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *  * @param id the token (pattern) id  * @param min
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or	<pre>true if the rule can match an empty token stream, or</pre>	<pre>= 0; i &lt; elements.Count; i++) {         elem =     (ProductionPatternElem     ent) elements[i];         min += elem.MinCount;     }     return min; }  /**     * Returns the maximum number of</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *  * @param id the token (pattern) id  * @param min the minimum number of
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  *	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0;  /**  * Changes the production pattern containing this	<pre>= 0; i &lt; elements.Count; i++) {         elem =     (ProductionPatternElem     ent) elements[i];         min += elem.MinCount;     }     return min; }  /**     * Returns the maximum number of elements needed to</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *  * @param id the token (pattern) id  * @param min the minimum number of occurancies
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0;  /**  * Changes the production pattern containing this alternative.	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0;  /**  * Changes the production pattern containing this alternative.  * This method	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */ public bool	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  *
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */  public bool IsRightRecursive() {	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() { return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production pattern	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  *  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  *  -1 for infinite
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */  public bool IsRightRecursive() {	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() { return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production pattern  * class.	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  * -1 for infinite  */
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */  public bool IsRightRecursive() {	true if the rule can match an empty token stream, or  * false otherwise  */  public bool IsMatchingEmpty() {  return GetMinElementCount() == 0;  }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production pattern  * class.  *	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  * -1 for infinite  */ public void
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */  public bool IsRightRecursive() {  ProductionPatternEleme nt elem;	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production pattern  * class.  *  * @param	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  * -1 for infinite  */ public void AddToken(int id, int
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */  public bool IsRightRecursive() {  ProductionPatternEleme nt elem;	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production pattern  * class.  *  * @param pattern the new	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  * -1 for infinite  */ public void
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */   public bool IsRightRecursive() {  ProductionPatternEleme nt elem;  for (int i = elements.Count - 1;	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() { return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production pattern  * class.  *  * @param pattern the new production pattern	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  * "-1 for infinite  */  public void AddToken(int id, int min, int max) {
* Checks if this alternative is recursive on the right-hand side.  * This method checks all the possible right side elements and  * returns true if the pattern itself is among them.  *  * @return true if the alternative is right side recursive, or  * false otherwise  */  public bool IsRightRecursive() {  ProductionPatternEleme nt elem;	true if the rule can match an empty token stream, or  * false otherwise  */ public bool IsMatchingEmpty() {  return GetMinElementCount() == 0; }  /**  * Changes the production pattern containing this alternative.  * This method should only be called by the production pattern  * class.  *  * @param pattern the new	<pre>= 0; i &lt; elements.Count; i++) {</pre>	alternative. The token is appended to  * the end of the element list. The multiplicity values  * specified define if the token is optional or required, and  * if it can be repeated.  * @param id the token (pattern) id  * @param min the minimum number of occurancies  * @param max the maximum number of occurancies, or  * -1 for infinite  */ public void AddToken(int id, int

	,		
nt(true, id, min,	/**	* @param obj	if
max));	* Adds a	the object to compare	(!elements[i].Equals(a
}	production pattern	with	lt.elements[i])) {
/	element to this	*	
/**	alternative. The	* @return	return false;
* Adds a	*	true if the object is	}
production to this	multiplicity values in	identical to this one,	}
alternative. The	the element will be	or	return
production is	overridden with	* false otherwise	true;
* appended to	* the	raise otherwise  */	J
the end of the element	specified values. The	'	/**
list. The multiplicity * values	element is appended to the end of	public override bool	/** * Returns a
specified define if	* the element	Equals (object obj) {	hash code for this
the production is	list.	if (obj is	object.
optional or	*	ProductionPatternAlter	*
* required,	* @param elem	native) {	* @return a
and if it can be	the production pattern	return	hash code for this
repeated.	element	Equals((ProductionPatt	object
*	* @param min	ernAlternative) obj);	*/
* @param id	the minimum number of	elikiteinative/ obj/, } else {	public
the production	occurancies	return	override int
(pattern) id	* @param max	false:	GetHashCode() {
* @param min	the maximum number of	}	return
the minimum number of	occurancies, or	}	elements.Count.GetHash
occurancies	*	J	Code ();
* @param max	-1 for infinite	/**	}
the maximum number of	*/	* Checks if	,
occurancies, or	public void	this alternative is	/**
*	AddElement (ProductionP	equal to another. This	* Returns a
-1 for infinite	atternElement elem,	method	string representation
*/	decerning among arom,	* returns	of this object.
public void	int min,	true if the other	*
AddProduction(int id,	,	production pattern	* @return a
int min, int max) {	int max) {	alternative	token string
, , ,	, ,	* has	representation
AddElement (new	if	identical elements in	*/
ProductionPatternEleme	(elem.IsToken()) {	the same order.	public
nt(false, id, min,		*	override string
max));	AddToken(elem.Id, min,	* @param alt	ToString() {
}	max);	the alternative to	
	} else {	compare with	StringBuilder buffer
/**		*	= new StringBuilder();
* Adds a	AddProduction(elem.Id,	* @return	
production pattern	min, max);	true if the object is	for (int i
element to this	}	identical to this one,	= 0; i <
alternative. The	}	or	elements.Count; i++) {
* element is		*	if (i
appended to the end of	/**	false otherwise	> 0) {
the element list.	* Checks if	*/	
*	this object is equal	public bool	<pre>buffer.Append(" ");</pre>
* @param elem	to another. This	Equals(ProductionPatte	}
the production pattern	method only	rnAlternative alt) {	
element	* returns	if	buffer. Append (elements
*/	true for another	(elements.Count !=	[i]);
public void	production pattern	alt.elements.Count) {	}
AddElement (ProductionP	alternative	return	return
atternElement elem) {	* with	false;	<pre>buffer.ToString();</pre>
	identical elements in	}	}
elements. Add(elem);	the same order.	for (int i	}
}	*	= 0; i <	}
		elements.Count; i++) {	

/*	*/	this.min =	}
*	private int	min;	}
ProductionPatternEleme	min;	if (max <=	J
nt. cs	штт,	0) {	/**
*/	/**	max =	* Returns the
•-/	* The maximum	Int32. MaxValue:	minimum occurence
using System;	occurance count.	} else if	count.
using System, using System. Text;	*/	(max < min) {	*
using System. Text,	private int		* @return the
		max = min;	minimum occurence
namagnasa Cana Libnany	max;	IIIII,	count
namespace Core.Library	/**	this.max =	
1	,		*
/**	* The look-	max;	* @see
,	ahead set associated	.1. 1 141 1 11	#MinCount
* A production	with this element.	this.lookAhead = null;	*
pattern element. This	*/	}	* @deprecated
class represents a	private	/	Use the MinCount
reference to	LookAheadSet	/**	property instead.
* either a token	lookAhead;	* The node	*/
or a production. Each	/	identity property	public int
element also contains	/**	(read-only).	<pre>GetMinCount() {</pre>
minimum	* Creates a	*	return
* and maximum	new element. If the	*	MinCount;
occurence counters,	maximum value if zero	*/	}
controlling the number	(0) or	public int Id	,
of	* negative,	{	/**
* repetitions	it will be set to	get {	* The maximum
allowed. A production	Int32. MaxValue.	return	occurence count
pattern element is	*	id;	property (read-only).
always	* @param	}	*
* contained	isToken the	}	*
within a production	token flag	,	*/
pattern rule.	* @param id	/**	public int
*	the node identity	* Returns the	MaxCount {
	* @param min	node identity.	get {
*	the minimum number of	*	return
*/	occurancies	* @return the	max;
public class	* @param max	node identity	}
ProductionPatternEleme	the maximum number of	*	}
nt {	occurancies, or	* @see #Id	,
,	*	*	/**
/**	negative for infinite	* @deprecated	* Returns the
* The token	*/	Use the Id property	maximum occurence
flag. This flag is	public	instead.	count.
true for token	ProductionPatternEleme	*/	*
elements, and	nt(bool isToken,	public int	* @return the
* false for		GetId() {	maximum occurence
production elements.	int id,	return Id;	count
*/		}	*
private bool	int min,	,	* @see
token;		/**	#MaxCount
	int max) {	* The minimum	*
/**		occurence count	* @deprecated
* The node	this. token	property (read-only).	Use the MaxCount
identity.	= isToken;	*	property instead.
*/	this.id =	*	*/
private int	id;	*/	public int
id;	if (min <	public int	<pre>GetMaxCount() {</pre>
	0) {	MinCount {	return
/**	min =	get {	MaxCount;
* The minimum	0;	return	}
occurance count.	}	min;	

/**	this element is a	return	
* The look-	token	this. token ==	<pre>buffer. Append("{");</pre>
ahead set property.	* element,	elem. token	ballelinppena ( ( ) ,
This is the look-ahead	and the token has the	&&	<pre>buffer. Append (min);</pre>
set	same id and this	this.id == elem.id	buller, appears (min),
* associated	element.	&&	<pre>buffer. Append(", ");</pre>
with this alternative.	*	this.min == elem.min	buller. Appena ( , ),
*/	* @param	&&	buffer. Append (max);
internal	token the	this.max == elem.max;	buller. Append (max),
LookAheadSet LookAhead	token to check	else {	<pre>buffer. Append(")");</pre>
Solution   Continue   Continue	*	return	buller. Append() /,
mat 1	* @return	false;	return
get {	true if the token	laise,	buffer. ToString();
return lookAhead;	matches this element.	) 1	builer. lostring(),
100kAllead,	,	,	,
}	or	/stude	}
set {	*	/**	}
1 1 1 1 1	false otherwise	* Returns a	/.
lookAhead = value;	*/	hash code for this	/*
}	public bool	object.	* ReaderBuffer.cs
}	IsMatch(Token token) {	*	*/
,	return	* @return a	
/**	IsToken() && token !=	hash code for this	using System;
* Returns	null && token.Id ==	object	using System.IO;
true if this element	id;	*/	
represents a token.	}	public	
*		override int	namespace Core.Library
* @return	/**	GetHashCode() {	{
true if the element is	* Checks if	return	
a token, or	this object is equal	this.id * 37;	/**
*	to another. This	}	* A character
false otherwise	method only		buffer that
*/	* returns	/**	automatically reads
public bool	true for another	* Returns a	from an input source
IsToken() {	identical production	string representation	* stream when
return	pattern	of this object.	needed. This class
token;	* element.	*	keeps track of the
}	*	* @return a	current position
	* @param obj	string representation	* in the buffer
/**	the object to compare	of this object	and its line and
* Returns	with	*/	column number in the
true if this element	*	public	original input
represents a	* @return	override string	* source. It
production.	true if the object is	ToString() {	allows unlimited look-
*	identical to this one,		ahead of characters in
* @return	or	StringBuilder buffer	the input,
true if the element is	*	<pre>= new StringBuilder();</pre>	* reading and
a production, or	false otherwise		buffering the required
*	*/		data internally. As
false otherwise	public	<pre>buffer. Append(id);</pre>	the
*/	override bool	if (token)	* position is
public bool	Equals(object obj) {	{	advanced, the buffer
IsProduction() {	1 ( 3 3)	•	content prior to the
return	ProductionPatternEleme	buffer. Append ("(Token)	current
!token;	nt elem;	");	* position is
}	,	} else {	subject to removal to
,	if (obj is	, 0100 (	make space for reading
/**	ProductionPatternEleme	buffer. Append ("(Produc	new
* Checks if a	nt) {	tion)");	* content. A few
specific token matches	elem =	}	characters before the
this element. This	(ProductionPatternElem	if (min !=	current position are
* method will	ent) obj;	1     max != 1) {	always
only noturn true if	ent/ obj,	1    max :- 1) (	aiways

only return true if

w bont to anable		if (input	
* kept to enable boundary condition	/**	!= null) {	/**
checks.	* The line	:- null) {	* The current
*	number of the next	ci y	character buffer
	character to read.	input.Close();	length property (read-
*	This value will	}	only).
*	* be	catch (Exception) {	* Note that
*/	incremented when	//	the length may
public class	reading past line	Do nothing	increase (and
ReaderBuffer {	breaks.	}	decrease) as more
	*/	input	* characters
/**	private int	= null;	are read from the
* The stream	line = 1;	}	input source or
reading block size.		}	removed to
All reads from the	/**	,	* free up
underlying	* The column	/**	space.
* character	number of the next	* The current	*/
stream will be made in	character to read.	buffer position	public int
multiples of this	This value	property (read-only).	Length {
block size.  * Also the	* will be updated for every	*/ public int	get {
	1	*	return
character buffer size	character read. */	Position {	length;
will always be a multiple of	private int	get { return	}
* this	column = 1;	pos;	J
factor.	Corumn – 1,	pos, }	/**
*/	/**	}	* Returns a
public const	* Creates a	,	substring already in
int BLOCK_SIZE = 1024;	new tokenizer	/**	the buffer. Note that
	character buffer.	* The current	this
/**	*	line number property	* method may
* The	* @param	(read-only). This	behave in unexpected
character buffer.	input the	number	ways when performing
*/	input source character	* is the line	* operations
private char[]	reader	number of the next	that modifies the
buffer = new	*/	character to read.	buffer content.
$char[BLOCK\_SIZE * 4];$	public	*/	*
	ReaderBuffer (TextReade	public int	* @param
/**	r input) {	LineNumber {	index the
* The current	this.input	get {	start index, inclusive
character buffer	= input;	return	* @param
position. */	}	line;	length the
,	/**	}	substring length
<pre>private int pos = 0;</pre>	* Discards all	,	* @return the
pos – 0,	resources used by this	/**	substring specified
/**	buffer. This will also	* The current	*
* The number	* close the	column number property	* @throws
of characters in the	source input stream.	(read-only). This	IndexOutOfBoundsExcept
buffer.	Disposing a previously	number	ion if one of the
*/	disposed	* is the	indices were
private int	* buffer has	column number of the	*
length = 0;	no effect.	next character to	negative or not less
	*/	read.	than (or equal) than
/**	public void	*/	length()
* The input	Dispose() {	public int	*/
source character	buffer =	ColumnNumber {	public string
reader.	null;	get {	Substring(int index,
*/	pos = 0;	return	int length) {
private	length =	column;	return new
TextReader input =	0;	}	string(buffer, index,
null;		}	length);

l	character offset, from	* returned	l
J	0 and up	string might be	return
/**	*	shorter than	result;
* Returns the	* @return the	requested. Any	}
current content of the	character found as an	* remaining	}
buffer as a string.	integer in the range 0	characters will always	
Note	to	be returned before	/**
* that	* 65535	returning	* Updates the
content before the	(0x00-0xffff), or $-1$	* null.	line and column
current position will	if the end of the	*	numbers counters. This
also be	stream was reached	* @param	method
* returned.	*	offset the	* requires
*	* @throws	character offset, from	all the characters to
* @return the	IOException if an I/O	0 and up	be processed (i.e.
current buffer content	error occurred	*	returned
*/	*/	* @return the	* as read) to
public	public int	string containing the	be present in the
override string	Peek(int offset) {	characters read, or	buffer, starting at
ToString() {	int index	*	the
return new	<pre>= pos + offset;</pre>	null no more	* current
string(buffer, 0,	// • • •	characters remain in	buffer position.
length);	// Avoid	the buffer	*
}	most calls to	*	* @param
/state	EnsureBuffered(), since we are in a	* @throws	offset the
/** * Returns a	since we are in a	IOException if an I/O error occurred	number of characters
character relative to	performance hotspot	*/	to process */
the current position.	here. This check is	public string	private void
This	not exhaustive,	Read(int offset) {	UpdateLineColumnNumber
* method may	// but	int	s(int offset) {
read from the input	only present here to	count;	for (int i
source and may also	speed things up.	string	= 0; i < offset; i++)
trim the	if (index	result;	{
* buffer	>= length) {	,	if
content prior to the	<b>3</b>		(buffer[pos + i] ==
current position. The	EnsureBuffered(offset	EnsureBuffered(offset	'\n') {
result of	+ 1);	+ 1);	
* calling	index	if (pos >=	line++;
this method may	<pre>= pos + offset;</pre>	length) {	
therefore be that the	}	return	column = 1;
buffer length	return	null;	} else
* and content	$(index \ge length) ? -1$	} else {	{
have been modified. $\langle p \rangle$	: buffer[index];	count	
*	}	= length - pos;	column++;
* The	,	if	}
character offset must	/**	(count > offset) {	}
be positive, but is	* Reads the		}
allowed to span	specified number of	<pre>count = offset;</pre>	/state
* the entire	characters from the	}	/** * Ensures
size of the input source stream. Note	current * position.	UndetalineColumnNumber	* Ensures that the specified
that the	This will also move	UpdateLineColumnNumber s(count);	offset is read into
* internal	the current position	result	the buffer.
buffer must hold all	forward.	= new string(buffer,	* This method
the intermediate	* This method	pos, count);	will read characters
characters,	will not attempt to	pos, count),	from the input stream
* which may	move beyond the end of	count;	and
be wasteful if the	the	if	* appends
offset is too large.	* input	(input == null && pos	them to the buffer if
*	source stream. When	>= length) {	needed. This method is
* @param	reaching the end of		safe to
offset the	file, the	Dispose();	

* call even	//	if	* @param
after end of file has	Calculate number of	(buffer.Length >=	input the
been reached. This	characters to read	size) {	input stream to read
method also	size = pos		from
* handles	+ offset - length + 1;	return;	*
removal of characters	if (size %	; c ( ; , , , ,	* @throws
at the beginning of	BLOCK_SIZE != 0) {	if (size %	ParserCreationExceptio
the buffer	size =	BLOCK_SIZE != 0) {	n if the tokenizer
* once the	(1 + size /	size = (1 + size /	couldn't be *
current position is	BLOCK_SIZE) * BLOCK SIZE;	BLOCK SIZE) *	initialized correctly
high enough. It will	DLUCK_SIZE,	_	*
also enlarge * the buffer	ſ	BLOCK_SIZE;	*
as needed.	EnsureCapacity(length	}	*/
as needed.	+ size);	Array. Resize (ref	public
* @param	Size),	buffer, size);	RecursiveDescentParser
offset the	// Read	builer, Size/,	(TextReader input) :
read offset, from 0	characters	}	base(input) {
and up	try {	}	base (Input) (
*	while	J	J
* @throws	(input != null && size	/*	/**
IOException if an	> 0) {	*	* Creates a
error was encountered	/ 0) (	RecursiveDescentParser	new parser.
while reading	readSize =		new parser.
willie leading *	input. Read (buffer,	. cs */	* @param
the input stream	length, size);	*/	input the
*/	if	using System;	input stream to read
private void	(readSize > 0) {	using system,	from
EnsureBuffered(int	(10405120 / 0) (	System. Collections;	* @param
offset) {	<pre>length += readSize;</pre>	using System. IO;	analyzer the
int size;	rength · readSize,	dsing bystem, 10,	analyzer callback to
int	size -= readSize;	namespace Core.Library	use
readSize;	size readsize,	{	*
readsize,	else {	(	* @throws
// Check	CISC (	/**	ParserCreationExceptio
for end of stream or	input.Close();	* A recursive	n if the tokenizer
already read	impat. close (),	descent parser. This	couldn't be
characters	input = null;	parser handles LL(n)	*
if (input	}	grammars,	initialized correctly
== null    pos +	}	* selecting the	*
offset < length) {	} catch	appropriate pattern to	*
	(IOException e) {	parse based on the	*/
return;	input	next few	public
}	= null:	* tokens. The	RecursiveDescentParser
,	throw	parser is more	(TextReader input,
// Remove	e;	efficient the fewer	Analyzer analyzer)
(almost all) old	}	look-ahead tokens	:
characters from buffer	}	* that is has to	base(input, analyzer)
if (pos >	,	consider.	{
BLOCK SIZE) {	/**	*	}
length	* Ensures		
-= (pos - 16);	that the buffer has at	*	/**
,	least the specified	*/	* Creates a
Array.Copy(buffer, pos	capacity.	public class	new parser.
- 16, buffer, 0,	*	RecursiveDescentParser	*
length);	* @param size	: Parser {	* @param
pos =	the minimum buffer	•	tokenizer the
16;	size	/**	tokenizer to use
}	*/	* Creates a	*/
	private void	new parser.	public
	EnsureCapacity(int	*	RecursiveDescentParser
	- · · · · · · · · · · · · · · · · · · ·		
	size) {		(Tokenizer tokenizer)

:	// Check	internal data	protected
base(tokenizer) {	for empty matches	structures used during	override Node
}	if	* the	ParseStart() {
	(pattern.IsMatchingEmp	parsing.	Token
/**	ty()) {	*	token;
* Creates a	throw	* @throws	Node node;
new parser.	new	ParserCreationExceptio	ArrayList
*	ParserCreationExceptio	n if the parser	list;
* @param	n (	couldn't be	
tokenizer the		*	node =
tokenizer to use	ParserCreationExceptio	initialized correctly	ParsePattern(GetStartP
* @param	n.ErrorType.INVALID_PR	*/	attern());
analyzer the	ODUCTION,	public	token =
analyzer callback to		override void	PeekToken(0);
use	pattern.Name,	Prepare() {	if (token
*/	<b>"</b> 1	IP	!= null) {
public	"zero elements can be	IEnumerator e;	list =
RecursiveDescentParser	<pre>matched (minimum is one)");</pre>	//	new ArrayList(1);
(Tokenizer tokenizer,	one)),	Performs production	list.Add(" <eof>");</eof>
Analyzer analyzer)	,	pattern checks	throw
· · ·	// Check	pattern checks	new ParseException(
base(tokenizer,	for left-recusive	base. Prepare();	new rarseException(
analyzer) {	patterns	base. Trepare (),	ParseException.ErrorTy
}	if	SetInitialized(false);	pe. UNEXPECTED TOKEN,
J	(pattern. IsLeftRecursi	Setimitianizea (ranse),	pe. onem beleb_lonem,
/**	ve()) {	//	token.ToShortString(),
* Adds a new	throw	Calculate production	
production pattern to	new	look-ahead sets	list,
the parser. The	ParserCreationExceptio	e =	
pattern	n (	GetPatterns().GetEnume	token.StartLine,
* will be		rator();	
added last in the	ParserCreationExceptio	while	token.StartColumn);
list. The first	n.ErrorType.INVALID_PR	(e.MoveNext()) {	}
pattern added is	ODUCTION,		return
* assumed to		CalculateLookAhead((Pr	node;
be the starting point	pattern. Name,	oductionPattern)e.Curr	}
in the grammar. The		ent);	
* pattern	"left recursive	}	/**
will be validated	patterns are not	// 0	* Parses a
against the grammar	allowed");	// Set	production pattern. A
type to some	}	initialized flag	parse tree node may or
* extent.	// 411	C (T.:(1: 1/())	may
*	// Add	SetInitialized(true);	* not be created depending on
* @param pattern the	pattern	ſ	the analyzer
pattern to add	base. AddPattern(patter	/**	callbacks.
*	n);	* Parses the	*
* @throws	}	input stream and	* @param
ParserCreationExceptio	,	creates a parse tree.	pattern the
n if the pattern	/**	*	production pattern to
couldn't be	* Initializes	* @return the	parse
*	the parser. All the	parse tree	*
added correctly to the	added production	*	* @return the
parser	patterns	* @throws	parse tree node
*/	* will be	ParseException if the	created, or null
public	analyzed for	input couldn't be	*
override void	ambiguities and	parsed	* @throws
AddPattern(ProductionP	errors. This method	*	ParseException if the
attern pattern) {	* also	correctly	input couldn't be
	initializes the	*/	parsed

ale.	N. @4 h	:	
* correctly	<pre>* @throws ParseException if the</pre>	input couldn't be	production.AddProducti
•	input couldn't be	parsed *	onState ("Enter: <" +
*/	parsed		pr + ">\n");
private Node ParsePattern(Productio	par seu *	correctly */	pr + /\n ),
nPattern pattern) {		,	ob:14 -
nrattern pattern) {	correctly */	private void ParseElement(Productio	child = ParsePattern(GetPatter
ProductionPatternAlter	*/ private Node	n node,	n (elem. Id));
native alt;	ParseAlternative(Produ	n node,	n(eiem. id)),
native ait,	ctionPatternAlternativ	ProductionPatternEleme	AddNode(node, child);
ProductionPatternAlter	e alt) {	nt elem) {	Audivode (node, child),
native defaultAlt:	Production	пт етеш/ (	} else
mative defaulthit,	node;	Node	) 6136
defaultAlt	noue,	child;	pr
=	node =	CHIId,	= pr. Substring(5);
pattern. DefaultAlterna	NewProduction(alt.Patt	for (int i	pr. oubstring (o),
tive;	ern);	= 0; i <	production.AddRecursiv
for (int i	CIII),	elem.MaxCount; i++) {	eProduction("Enter:
= 0; i <	EnterNode(node);	string	NULL <" + pr + ">\n"):
pattern.Count; i++) {	for (int i	pr =	TOLL ( pr //m/,
alt =	= 0; i < alt.Count;	Enum. GetName(typeof(Sy	production.AddProducti
pattern[i];	i++) {	ntaxConstants),	onState ("NULL");
if	try {	elem.GetId());	one tate ( Nebb ),
(defaultAlt != alt &&	017	if (i	production.AddProducti
IsNext(alt)) {	ParseElement (node,	< elem. MinCount	onCode(elem.GetId());
151.6110 (011)	alt[i]);	IsNext(elem)) {	onesas (elem estra (//,
return	}	if	break;
ParseAlternative(alt);	catch (ParseException	(elem.IsToken()) {	}
}	e) {	, , , ,	}
}	, ,	child =	}
if	AddError(e, true);	<pre>NextToken(elem.Id);</pre>	
4			/**
(defaultAlt == null			/ <b>^</b>
(defaultAlt == null    !IsNext(defaultAlt)) {	NextToken();	EnterNode(child);	* Checks if
	NextToken();	<pre>EnterNode(child);</pre>	,
		EnterNode(child); AddNode(node,	* Checks if
!IsNext(defaultAlt)) {	i-		* Checks if the next tokens match
!IsNext(defaultAlt)) { ThrowParseException(Fi	i-	AddNode (node,	* Checks if the next tokens match a production pattern.
!IsNext(defaultAlt)) { ThrowParseException(Fi	i-	AddNode (node,	* Checks if the next tokens match a production pattern. The
<pre>!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern)); }</pre>	i- -; }	AddNode(node, ExitNode(child));	* Checks if the next tokens match a production pattern. The
<pre>!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern));      }      return</pre>	i- -; } return	AddNode(node, ExitNode(child)); if(ExitNode(child) !=	* Checks if the next tokens match a production pattern. The
!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern)); } return ParseAlternative(defau	i- -; } return	AddNode(node, ExitNode(child)); if(ExitNode(child) !=	* Checks if the next tokens match a production pattern. The
!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern)); } return ParseAlternative(defau	i- -; } return	AddNode(node, ExitNode(child)); if(ExitNode(child) != null)	* Checks if the next tokens match a production pattern. The
!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern)); } return ParseAlternative(defau	i; } return ExitNode(node); }	AddNode(node, ExitNode(child));  if(ExitNode(child) != null)  production.AddRecursiv	* Checks if the next tokens match a production pattern. The
<pre>!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern));</pre>	i; } return ExitNode(node); } /**	AddNode(node, ExitNode(child));  if(ExitNode(child) != null)  production. AddRecursiv eProduction("Enter: "	* Checks if the next tokens match a production pattern. The
<pre>!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern));</pre>	i; } return ExitNode(node); } /** * Parses a	AddNode(node, ExitNode(child));  if(ExitNode(child) != null)  production. AddRecursiv eProduction("Enter: "	* Checks if the next tokens match a production pattern. The
<pre>!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern));</pre>	i; } return ExitNode(node); }  /** * Parses a production pattern	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n");	* Checks if the next tokens match a production pattern. The     * pattern look-ahead set will be used if existing, otherwise     * this method returns false.     *     * @param
<pre>!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern));</pre>	i; } return ExitNode(node); }  /** * Parses a production pattern element. All nodes	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId());	* Checks if the next tokens match a production pattern. The
<pre>!IsNext(defaultAlt)) { ThrowParseException(Fi ndUnion(pattern));</pre>	i; } return ExitNode(node); }  /** * Parses a production pattern element. All nodes parsed may	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti	* Checks if the next tokens match a production pattern. The
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern)); } return  ParseAlternative(defau ltAlt); }  /**  * Parses a  production pattern alternative. A parse tree node  * may or may not be created	i; } return ExitNode(node); }  /** * Parses a production pattern element. All nodes parsed may * or may not	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr	* Checks if the next tokens match a production pattern. The
!IsNext(defaultAlt)) {  ThrowParseException(FindUnion(pattern)); } return  ParseAlternative(defaultAlt); }  /** * Parses a  production pattern alternative. A parse tree node * may or may	i; } return ExitNode(node); }  /** * Parses a production pattern element. All nodes parsed may * or may not be added to the parse	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti	* Checks if the next tokens match a production pattern. The
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern)); } return  ParseAlternative(defau ltAlt); }  /**  * Parses a  production pattern alternative. A parse tree node  * may or may not be created depending on the analyzer	i; } return ExitNode(node); }  /** * Parses a production pattern element. All nodes parsed may * or may not be added to the parse tree node specified, * depending on the analyzer	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  *
!IsNext(defaultAlt)) {  ThrowParseException(FindUnion(pattern)); } return  ParseAlternative(defaultAlt); }  /**  * Parses a  production pattern alternative. A parse tree node  * may or may not be created depending on the analyzer  * callbacks.	i; } return ExitNode(node); }  /** * Parses a production pattern element. All nodes parsed may * or may not be added to the parse tree node specified, * depending	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr + "\n");	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern));	i; } return ExitNode(node); }  /** * Parses a production pattern element. All nodes parsed may * or may not be added to the parse tree node specified, * depending on the analyzer callbacks. *	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */
!IsNext(defaultAlt)) {  ThrowParseException(FindUnion(pattern));  return  ParseAlternative(defaultAlt);  /**  * Parses a  production pattern  alternative. A parse  tree node  * may or may  not be created  depending on the  analyzer  * callbacks.  *  * @param alt	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr + "\n"); } else {	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern));	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node the production parse	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr + "\n");	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool IsNext(ProductionPatte
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern));	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node the production parse tree node	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr + "\n"); } else { pr = pr. Substring(5);	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool
!IsNext(defaultAlt)) {  ThrowParseException(FindUnion(pattern));	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node the production parse tree node  * @param elem	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr + "\n"); } else { pr = pr. Substring(5); production. AddRecursiv	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool IsNext(ProductionPatte rn pattern) {
!IsNext(defaultAlt)) {  ThrowParseException(FindUnion(pattern));	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node the production parse tree node  * @param elem the production pattern	AddNode(node, ExitNode(child));  if(ExitNode(child) != null)  production. AddRecursiv eProduction("Enter: " + pr + "\n");  production. AddProducti onCode(elem. GetId());  production. AddProducti onState("Enter: " + pr + "\n");  }  else {  pr = pr. Substring(5);  production. AddRecursiv eProduction("Enter: <"	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool IsNext(ProductionPatte rn pattern) { LookAheadSet set =
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern));	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node the production parse tree node  * @param elem the production pattern element to parse	AddNode(node, ExitNode(child)); if(ExitNode(child) != null) production. AddRecursiv eProduction("Enter: " + pr + "\n"); production. AddProducti onCode(elem. GetId()); production. AddProducti onState("Enter: " + pr + "\n"); } else { pr = pr. Substring(5); production. AddRecursiv	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool IsNext(ProductionPatte rn pattern) {
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern));	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node the production parse tree node  * @param elem the production pattern element to parse  *	AddNode(node, ExitNode(child));  if(ExitNode(child))!= null)  production. AddRecursiv eProduction("Enter: " + pr + "\n");  production. AddProducti onCode(elem. GetId());  production. AddProducti onState("Enter: " + pr + "\n");  }  else {  pr = pr. Substring(5);  production. AddRecursiv eProduction("Enter: <" + pr + ">\n");	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool IsNext(ProductionPatte rn pattern) {  LookAheadSet set = pattern.LookAhead;
!IsNext(defaultAlt)) {  ThrowParseException(Fi ndUnion(pattern));	i; } return ExitNode(node); }  /**  * Parses a production pattern element. All nodes parsed may  * or may not be added to the parse tree node specified,  * depending on the analyzer callbacks.  *  * @param node the production parse tree node  * @param elem the production pattern element to parse	AddNode(node, ExitNode(child));  if(ExitNode(child) != null)  production. AddRecursiv eProduction("Enter: " + pr + "\n");  production. AddProducti onCode(elem. GetId());  production. AddProducti onState("Enter: " + pr + "\n");  }  else {  pr = pr. Substring(5);  production. AddRecursiv eProduction("Enter: <"	* Checks if the next tokens match a production pattern. The  * pattern look-ahead set will be used if existing, otherwise  * this method returns false.  *  * @param pattern the pattern to check  *  * @return true if the next tokens match, or  * false otherwise  */ private bool IsNext(ProductionPatte rn pattern) { LookAheadSet set =

retu	rn *	ductionPattern	conflicts
false;	* @param elem	pattern) {	=
} else {	the pattern element to		FindConflicts(pattern,
retu	rn check	ProductionPatternAlter	1);
set.IsNext(this);	*	native alt;	
}	* @return		// Resolve
}	true if the next	LookAheadSet	conflicts
,	tokens match, or	result;	while
/**	*	1 141 10 . [7]	(conflicts.Size() > 0)
* Checks if		LookAheadSet[]	1
the next tokens matc	,	alternatives;	1
a production pattern		1 141 10 4	length++;
*	IsNext(ProductionPatte rnElement elem) {	LookAheadSet conflicts;	stack Clean().
alternative. The pattern alternative	Therement etem) (	Confilets,	stack.Clear();
look-ahead set will	be LookAheadSet set =	LookAheadSet	stack. Push (pattern. Nam
* used if	elem. LookAhead;	previous = new	e, length);
existing, otherwise	erem. Bookineda,	LookAheadSet (0);	c, rength,
this method returns	if (set !=	int	conflicts.AddAll(previ
false.	nu11) {	length = 1;	ous);
*	return	int	for (i
* @param al	t set. IsNext(this);	i;	= 0; i <
the pattern	} else if	CallStack	pattern.Count; i++) {
alternative to check	(elem.IsToken()) {	stack = new	
*	return	CallStack();	alt = pattern[i];
* @return	elem.IsMatch(PeekToken		if
true if the next	(0));	//	(alternatives[i].Inter
tokens match, or	} else {	Calculate simple look-	sects(conflicts)) {
*	return	ahead	
false otherwise	IsNext(GetPattern(elem		alternatives[i] =
*/	. Id));	stack.Push(pattern.Nam	FindLookAhead(alt,
private bool	}	e, 1);	
IsNext (ProductionPat	·	result =	length,
rnAlternative alt) {		new LookAheadSet(1);	0
I - 1 A1 - 1C - 4 4 -	/**	.14	0,
LookAheadSet set =	* Calculates	alternatives = new	-41-
alt.LookAhead;	the look-ahead needed for the specified	LookAheadSet[pattern.C ount];	stack,
if (set		for (i =	conflicts);
null) {	* pattern.	0; i < pattern.Count;	confilets),
retu		i++) {	alt.LookAhead =
false;	to resolve any	alt =	alternatives[i];
} else {		<pre>pattern[i];</pre>	}
retu	_		if
set.IsNext(this);	results in the pattern	alternatives[i] =	(alternatives[i].Inter
}	look-ahead object.	FindLookAhead(alt, 1,	sects(conflicts)) {
}	*	O, stack, null);	
	* @param		if
/**	pattern the	alt.LookAhead =	(pattern.DefaultAltern
* Checks if	•	alternatives[i];	ative == null) {
the next tokens matc			
a production pattern		result. AddAll(alternat	pattern.DefaultAlterna
* element.	-	ives[i]);	tive = alt;
the element has a	n if the look-ahead	} : c	l also if
look-ahead set it wi be	ll set couldn't *	if (pattern.LookAhead ==	} else if (pattern.DefaultAltern
* used,	be determined due to	null) {	ative != alt) {
otherwise the look-	inherent ambiguities	nuii/ (	active := arc/ (
ahead set of the	*/	pattern.LookAhead =	result =
referenced	private void	result;	alternatives[i].Create
* productio		}	Intersection (conflicts
or token will be use		•	);

	private void	new CallStack(),	
ThrowAmbiguityExceptio	CalculateLookAhead(Pro	nu11);	ThrowAmbiguityExceptio
n (pattern. Name,	${\tt ductionPatternAlternat}$	follow =	n (pattern. Name,
	ive alt,	FindLookAhead(alt, 1,	location, first);
null,		pos + 1, new	}
1, ) .	int pos) {	CallStack(), null);	
result);		// Resolve	<pre>previous = conflicts;</pre>
}	ProductionPattern	conflicts	conflicts =
}	pattern;	location =	FindConflicts(pattern.
}		"at position" + (pos	Name,
	ProductionPatternEleme	+ 1);	
<pre>previous = conflicts;</pre>	nt elem;	conflicts	location,
		=	
conflicts =	LookAheadSet	FindConflicts(pattern.	first,
<pre>FindConflicts(pattern, length);</pre>	first;	Name,	follow);
length,	LookAheadSet	location,	10110W),
J	follow;	rocation,	J
// Resolve	,	first,	// Check
conflicts inside rules	LookAheadSet		remaining elements
for $(i =$	conflicts;	follow);	
0; i < pattern.Count;		while	CalculateLookAhead(alt
i++) {	LookAheadSet	(conflicts.Size() > 0)	, pos + 1);
	previous = new	{	}
<pre>CalculateLookAhead(pat tern[i], 0);</pre>	LookAheadSet(0);	longth	/**
tern[1], 0);	String location;	length++;	/** * Finds the
}	int	conflicts.AddAll(previ	look-ahead set for a
,	length = 1;	ous);	production pattern.
/**	,	first	The maximum
* Calculates	// Check	= FindLookAhead(elem,	* look-ahead
the look-aheads needed	trivial cases		length must be
for the specified	if (pos >=	length,	specified. It is also
pattern	alt.Count) {	0.110. 1.0	possible to
*	notum.	new CallStack(),	* specify a
alternative. This method attempts to	return;	conflicts);	look-ahead set filter, which will make sure
resolve any conflicts	J	follow	that
in	// Check	= FindLookAhead(alt,	* unnecessary
* optional	for non-optional	, ,	token sequences will
elements by	element	length,	be avoided.
recalculating look-	pattern =		*
aheads for referenced	alt.Pattern;	pos + 1,	* @param
*	elem =	0.110, 1.()	pattern the
productions. *	alt[pos]; if	new CallStack(),	production pattern * @param
* @param alt	(elem.MinCount ==	conflicts);	length the
the production pattern	elem. MaxCount) {	first	maximum look-ahead
alternative	, ,	=	length
* @param pos	CalculateLookAhead(alt	first.CreateCombinatio	* @param
the pattern element	, pos + 1);	n(follow);	stack the
position			call stack used for
*	return;	elem. LookAhead =	loop detection
<pre>* @throws ParserCreationExceptio</pre>	}	first;	* @param filter the
n if the look-ahead	//	(first. Intersects (conf	look-ahead set filter
set couldn't	Calculate simple look-	licts)) {	*
*	aheads	_1000//	* @return the
be determined due to	first =	first =	look-ahead set for the
inherent ambiguities	FindLookAhead(elem, 1,	first.CreateIntersecti	production pattern
*/		on(conflicts);	*

* @throws		private	first.CreateCombinatio
ParserCreationExceptio	filter);	LookAheadSet	n(follow);
n if an infinite loop		FindLookAhead(Producti	}
was found	result.AddAll(temp);	onPatternAlternative	} else if
*	}	alt,	(filter.IsOverlap(firs
in the grammar			t)) {
*/	stack.Pop();	int length,	
private			overlaps =
LookAheadSet	return	int pos,	first.CreateOverlaps(f
FindLookAhead(Producti	result;		ilter);
onPattern pattern,	}	CallStack stack,	length
			-=
int length,	/**	LookAheadSet filter) {	overlaps.GetMinLength(
	* Finds the		);
CallStack stack,	look-ahead set for a		filter
	production pattern	LookAheadSet first;	=
LookAheadSet filter) {	alternative.		filter.CreateFilter(ov
	* The pattern	LookAheadSet follow;	erlaps);
	position and maximum		follow
LookAheadSet result;	look-ahead length must	LookAheadSet	= FindLookAhead(alt,
	be	overlaps;	length, pos + 1,
LookAheadSet temp;	* specified.		stack, filter);
•	It is also possible to	// Check	
// Check	specify a look-ahead	trivial cases	first.RemoveAll(overla
for infinite loop	set	if (length	ps);
if	* filter,	<= 0    pos >=	F - / /
(stack.Contains(patter	which will make sure	alt.Count) {	first.AddAll(overlaps.
n. Name, length)) {	that unnecessary token	return	CreateCombination(foll
throw	sequences	new LookAheadSet(0);	ow));
new	* will be	}	}
ParserCreationExceptio	avoided.	,	,
	*	// Find	return
n(	* * @param_alt	// Find look-ahead for this	return first:
n (	*     * @param alt     the production pattern	look-ahead for this	return first; }
n( ParserCreationExceptio	the production pattern	look-ahead for this element	
n ( ParserCreationExceptio n. ErrorType. INFINITE_L	the production pattern alternative	<pre>look-ahead for this element     first =</pre>	<pre>first; }</pre>
n( ParserCreationExceptio	the production pattern alternative * @param	look-ahead for this element first = FindLookAhead(alt[pos]	first; } /**
n ( ParserCreationExceptio n. ErrorType. INFINITE_L 00P,	the production pattern alternative * @param length the	<pre>look-ahead for this element</pre>	first; } /**
n ( ParserCreationExceptio n. ErrorType. INFINITE_L	the production pattern alternative * @param length the maximum look—ahead	<pre>look-ahead for this element</pre>	first; } /**  * Finds the look-ahead set for a
n ( ParserCreationExceptio n. ErrorType. INFINITE_L 00P, pattern. Name,	the production pattern alternative * @param length the maximum look-ahead length	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern
n ( ParserCreationExceptio n. ErrorType. INFINITE_L 00P,	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The
n ( ParserCreationExceptio n. ErrorType. INFINITE_L 00P, pattern. Name,	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The  * maximum
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null); }	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must
n(  ParserCreationExceptio n.ErrorType.INFINITE_L 00P,  pattern.Name,  (String) null);  }  // Find	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null); }	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null); } // Find pattern look-ahead	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null); } // Find pattern look-ahead stack.Push(pattern.Nam	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null); }  // Find pattern look-ahead stack.Push(pattern.Nam e, length);	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  * Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null); }  // Find pattern look-ahead  stack.Push(pattern.Nam e, length); result =	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the
n(  ParserCreationExceptio n.ErrorType.INFINITE_L 00P,  pattern.Name,  (String) null);  // Find pattern look-ahead  stack.Push(pattern.Nam e, length);  result = new	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead
n(  ParserCreationExceptio n.ErrorType.INFINITE_L  00P,  pattern.Name,  (String) null);  // Find  pattern look-ahead  stack.Push(pattern.Nam e, length);  result =  new  LookAheadSet(length);	the production pattern alternative	<pre>look-ahead for this element</pre>	first;  /**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT
<pre>n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null);</pre>	the production pattern alternative	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P,  pattern.Name,  (String) null);  // Find pattern look-ahead  stack.Push(pattern.Nam e, length);  result = new LookAheadSet(length);  for (int i = 0; i <	the production pattern alternative	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if
<pre>n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null);</pre>	the production pattern alternative	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum
<pre>n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null);</pre>	the production pattern alternative	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P,  pattern.Name,  (String) null);  // Find pattern look-ahead  stack.Push(pattern.Nam e, length);  result = new LookAheadSet(length);  for (int i = 0; i < pattern.Count; i++) {  temp = FindLookAhead(pattern[	the production pattern alternative	<pre>look-ahead for this element</pre>	/**  /**  * Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It is also possible to
<pre>n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null);</pre>	the production pattern alternative	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  * maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It is also possible to specify a
<pre>n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null);</pre>	the production pattern alternative  * @param length the maximum look-ahead length  * @param pos the pattern element position  * @param stack the call stack used for loop detection  * @param filter the look-ahead set filter  *  * @return the look-ahead set for the pattern alternative  *  * @throws ParserCreationExceptio n if an infinite loop	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  *maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It is also possible to specify a  * look-ahead
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P,  pattern.Name,  (String) null);  // Find pattern look-ahead  stack.Push(pattern.Nam e, length);  result = new LookAheadSet(length);  for (int i = 0; i < pattern.Count; i++) {  temp = FindLookAhead(pattern[	the production pattern alternative  * @param length the maximum look-ahead length  * @param pos the pattern element position  * @param stack the call stack used for loop detection  * @param filter the look-ahead set filter  * * @return the look-ahead set for the pattern alternative  * * @throws ParserCreationExceptio n if an infinite loop was found	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  *maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It is also possible to specify a  * look-ahead set filter, which will
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P,  pattern.Name,  (String) null);  // Find pattern look-ahead  stack.Push(pattern.Nam e, length);	the production pattern alternative  * @param length the maximum look-ahead length  * @param pos the pattern element position  * @param stack the call stack used for loop detection  * @param filter the look-ahead set filter  * * @return the look-ahead set for the pattern alternative  * * @throws ParserCreationExceptio n if an infinite loop was found  *	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  *maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It is also possible to specify a  * look-ahead set filter, which will make sure that
<pre>n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P, pattern.Name, (String) null);</pre>	the production pattern alternative  * @param length the maximum look-ahead length  * @param pos the pattern element position  * @param stack the call stack used for loop detection  * @param filter the look-ahead set filter  *  * @return the look-ahead set for the pattern alternative  *  * @throws ParserCreationExceptio n if an infinite loop was found  * in the grammar	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  *maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It is also possible to specify a  * look-ahead set filter, which will
n( ParserCreationExceptio n.ErrorType.INFINITE_L 00P,  pattern.Name,  (String) null);  // Find pattern look-ahead  stack.Push(pattern.Nam e, length);	the production pattern alternative  * @param length the maximum look-ahead length  * @param pos the pattern element position  * @param stack the call stack used for loop detection  * @param filter the look-ahead set filter  * * @return the look-ahead set for the pattern alternative  * * @throws ParserCreationExceptio n if an infinite loop was found  *	<pre>look-ahead for this element</pre>	/**  *Finds the look-ahead set for a production pattern element. The  *maximum look-ahead length must be specified. This method takes  * the element repeats into consideration when creating the  * look-ahead set, but does NOT include an empty sequence even if  * the minimum count is zero (0). It is also possible to specify a  * look-ahead set filter, which will make sure that

ve + -1	:	or Finds 4h.	
* token sequences will be	if (filter == null	* Finds the look-ahead set for a	LookAheadSet filter) {
avoided.	null    !filter.IsOverlap(resu	production pattern	Lookaneadset IIIter) (
avorueu.	1t)) {	element. The	
* @param elem	return	* maximum	LookAheadSet
the production pattern	result:	look-ahead length must	result;
element	}	be specified. This	resurt,
* @param	,	method does	ProductionPattern
length the	// Handle	* NOT take	pattern;
maximum look-ahead	element repetitions	the element repeat	partern,
length	if	into consideration	if
* @param	(elem. MaxCount ==	when creating	(elem.IsToken()) {
stack the	Int32. MaxValue) {	* the look-	result
call stack used for	first	ahead set. It is also	= new
loop detection	=	possible to specify a	LookAheadSet(length);
* @param	first.CreateRepetitive	look-ahead	, ,
filter the	();	* set filter,	result.Add(elem.Id);
look-ahead set filter	}	which will make sure	} else {
*	max =	that unnecessary token	, ,
* @return the	elem.MaxCount;	* sequences	pattern =
look-ahead set for the	if (length	will be avoided.	GetPattern(elem.Id);
pattern element	< max) {	*	result
*	max =	* @param elem	=
* @throws	length;	the production pattern	FindLookAhead(pattern,
ParserCreationExceptio	}	element	length, stack,
n if an infinite loop	for (int i	* @param	filter);
was found	$= 1; i < max; i++) {$	length the	if
*	first	maximum look-ahead	(stack.Contains(patter
in the grammar	=	length	n.Name)) {
*/	first.CreateOverlaps(f	* @param	
private	ilter);	dummy a	result =
LookAheadSet	if	parameter to	result.CreateRepetitiv
FindLookAhead(Producti	(first.Size() <= 0	distinguish the method	e();
onPatternElement elem,	first.GetMinLength()	* @param	}
	>= length) {	stack the	}
int length,		call stack used for	
	break;	loop detection	return
CallStack stack,	}	* @param	result;
	follow	filter the	}
LookAheadSet filter) {	= FindLookAhead(elem,	look-ahead set filter	
		*	/**
	length,	* @return the	* Returns a
LookAheadSet result;		look-ahead set for the	look-ahead set with
	0,	pattern element	all conflics between
LookAheadSet first;		*	*
	stack,	* @throws	alternatives in a
LookAheadSet follow;	011 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	ParserCreationExceptio	production pattern.
int	filter.CreateFilter(fi	n if an infinite loop	*
max;	rst));	was found	* @param
// P: 1	first	*	pattern the
// Find		in the grammar	production pattern
initial element look-	first.CreateCombinatio	*/	* @param
ahead	n(follow);	private	maxLength the
first =	1. 411411/6: ()	LookAheadSet	maximum token sequence
FindLookAhead(elem,	result.AddAll(first);	FindLookAhead (Producti	length
length, 0, stack,	}	onPatternElement elem,	*
filter);	**************************************	int longth	* @return a
result =	return	int length,	look-ahead set with
new LookAheadSet(length);	result;	int dummy,	the conflicts found
Looknieauset (Telig til),	J	THE dummy,	* @throws
result.AddAll(first);	/**	CallStack stack,	ParserCreationExceptio
roburt. Muunii (1115t/,	/	outionack stack,	1 albei el cationexceptio

n if an inherent	pattern name being	* @param	*
ambiguity was	analyzed	pattern the	* @throws
*	* @param	production pattern	ParseException always
found among the look-	location the	*	thrown by this method
ahead sets	pattern location	* @return a	*/
*/	* @param set1	unified look-ahead set	private void
private	the first look-ahead	*/	ThrowParseException(Lo
LookAheadSet	set	private	okAheadSet set) {
FindConflicts(Producti	* @param set2	LookAheadSet	Token
onPattern pattern,	the second look-ahead	FindUnion(ProductionPa	token;
F,	set	ttern pattern) {	ArrayList
int maxLength) {	*	, , , , , , , , , , , , , , , , , , ,	list = new
	* @return a	LookAheadSet result;	ArrayList();
	look-ahead set with	int	int[]
LookAheadSet result =	the conflicts found	length = 0;	initials;
new	*	int	,
LookAheadSet(maxLength	* @throws	i;	// Read
);	ParserCreationExceptio		tokens until mismatch
	n if an inherent	for (i =	while
LookAheadSet set1;	ambiguity was	0; i < pattern.Count;	(set.IsNext(this, 1))
	*	i++) {	{
LookAheadSet set2;	found among the look-	result	set =
	ahead sets	=	set.CreateNextSet(Next
for (int i	*/	<pre>pattern[i].LookAhead;</pre>	Token(). Id);
= 0; i <	private	if	}
pattern.Count; i++) {	LookAheadSet	<pre>(result.GetMaxLength()</pre>	
set1 =	FindConflicts(string	> length) {	// Find
pattern[i].LookAhead;	pattern,		next token
for		length =	descriptions
(int j = 0; j < i;	string location,	result.GetMaxLength();	initials =
j++) {		}	set.GetInitialTokens()
	LookAheadSet set1,	}	;
	,		
set2 =	,	result =	for (int i
set2 = pattern[j].LookAhead;	LookAheadSet set2) {	result = new	$\begin{array}{c} \text{for (int i} \\ = 0; \text{ i } < \end{array}$
			,
		new LookAheadSet(length); for (i =	= 0; i <
pattern[j].LookAhead;		<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre</pre>	LookAheadSet set2) { LookAheadSet result;	new LookAheadSet(length); for (i =	= 0; i < initials.Length; i++) { list.Add(GetTokenDescr
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre</pre>	LookAheadSet set2) {  LookAheadSet result;  result =	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2)); } }</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio	<pre>new LookAheadSet(length);     for (i = 0; i &lt; pattern.Count; i++) { result.AddAll(pattern[</pre>	= 0; i < initials.Length; i++) { list.Add(GetTokenDescr
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2);	<pre>new LookAheadSet(length);</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i])); }</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2)); } }</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2);  if	<pre>new LookAheadSet(length);     for (i = 0; i &lt; pattern.Count; i++) { result.AddAll(pattern[</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i]));</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersection(set2);  if (result.IsRepetitive()	<pre>new LookAheadSet(length);</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i]));</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2);  if	<pre>new LookAheadSet(length);</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i]));</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersection(set2);  if (result.IsRepetitive()) {	<pre>new LookAheadSet(length);</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i]));</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersection(set2);  if (result.IsRepetitive()) {  ThrowAmbiguityExceptio	<pre>new LookAheadSet(length);</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i]));</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersection(set2);  if (result.IsRepetitive()) {  ThrowAmbiguityException(pattern, location,	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i])); }  // Create exception token =  NextToken(); throw new ParseException(ParseEx
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersection(set2);  if (result.IsRepetitive()) {  ThrowAmbiguityExceptio	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token =  NextToken();  throw new ParseException(ParseException.ErrorType.UNEX
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2);  if (result.IsRepetitive())) {  ThrowAmbiguityExceptio n(pattern, location, result); }	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i])); }  // Create exception token =  NextToken(); throw new ParseException(ParseEx
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token = NextToken(); throw new ParseException(ParseException.ErrorType.UNEXPECTED_TOKEN,
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2);  if (result.IsRepetitive())) {  ThrowAmbiguityExceptio n(pattern, location, result); }	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token =  NextToken();  throw new ParseException(ParseException.ErrorType.UNEX
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return	<pre>new LookAheadSet(length);</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i]));</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; }	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token = NextToken(); throw new ParseException(ParseException.ErrorType.UNEXPECTED_TOKEN,
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; } /**	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token = NextToken(); throw new ParseException(ParseException.ErrorType.UNEXPECTED_TOKEN, token.ToShortString(), list,
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; }  /** * Returns the	<pre>new LookAheadSet(length);</pre>	<pre>= 0; i &lt; initials.Length; i++) {  list.Add(GetTokenDescr iption(initials[i]));</pre>
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2);  if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result);  return result;  }  /**  * Returns the union of all	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token = NextToken(); throw new ParseException(ParseException.ErrorType.UNEXPECTED_TOKEN, token.ToShortString(), list, token.StartLine,
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; }  /**  * Returns the union of all alternative look—ahead	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token = NextToken(); throw new ParseException(ParseException.ErrorType.UNEXPECTED_TOKEN, token.ToShortString(), list,
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; }  /**  * Returns the union of all alternative look—ahead sets in a	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i])); }  // Create exception token = NextToken(); throw new ParseException(ParseException.ErrorType.UNEXPECTED_TOKEN, token.ToShortString(), list, token.StartLine,
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; }  /**  * Returns the union of all alternative look—ahead sets in a  * production	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i]));  // Create exception  token = NextToken();  throw new ParseException(ParseException.ErrorType.UNEX PECTED_TOKEN,  token.ToShortString(),  list,  token.StartLine,  token.StartColumn);  }
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; }  /**  * Returns the union of all alternative look—ahead sets in a  * production pattern.	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i]));  // Create exception token = NextToken(); throw new ParseException(ParseException.ErrorType.UNEXPECTED_TOKEN,  token.ToShortString(),  list, token.StartLine, token.StartColumn); }  /**
<pre>pattern[j].LookAhead; result.AddAll(set1.Cre ateIntersection(set2));</pre>	LookAheadSet set2) {  LookAheadSet result;  result = set1.CreateIntersectio n(set2); if (result.IsRepetitive() ) {  ThrowAmbiguityExceptio n(pattern, location, result); } return result; }  /**  * Returns the union of all alternative look—ahead sets in a  * production	<pre>new LookAheadSet(length);</pre>	= 0; i < initials.Length; i++) {  list.Add(GetTokenDescription(initials[i]));  // Create exception  token = NextToken();  throw new ParseException(ParseException.ErrorType.UNEX PECTED_TOKEN,  token.ToShortString(),  list,  token.StartLine,  token.StartColumn);  }

<b>-</b>	THE PART OF THE PART A	/state	ate A J J
exception for an	n. ErrorType. INHERENT_A	/**	* Adds a
ambiguity. The	MBIGUITY,	* Checks	new element to the top
* specified		if the specified name	of the stack.
look-ahead set	pattern,	and value combination	*
contains the token		is on	* @param
conflicts to be	location,	* the	name the
* reported.	list);	stack.	stack name
*	}	*	* @param
* @param		* @param	value the
pattern the		name the	stack value
production pattern	/**	name to search for	*/
name	* A name	* @param	public
* @param	value stack. This	value the	void Push(string name,
location the	stack is used to	value to search for	int value) {
production pattern	detect loops and	*	
location, or null	* repetitions	* @return	nameStack.Add(name);
* @param set	of the same production	true if the	, , ,
the look-ahead set	during look-ahead	combination is on the	valueStack.Add(value);
with conflicts	analysis.	stack, or	}
*	*/	*	,
* @throws	private class	false otherwise	/**
	CallStack {	*/	* Removes
ParserCreationExceptio	CallStack (	,	
n always thrown by	/state	public	the top element of the
this method	/**	bool Contains(string	stack.
*/	* A stack	name, int value) {	*/
private void	with names.	for	public
ThrowAmbiguityExceptio	*/	(int $i = 0$ ; $i <$	void Pop() {
n(string pattern,	private	nameStack.Count; i++)	if
	ArrayList nameStack =	{	(nameStack.Count > 0)
string location,	new ArrayList();	if	{
		(nameStack[i].Equals(n	
LookAheadSet set) {	/**	ame)	nameStack.RemoveAt(nam
	* A stack		eStack.Count - 1);
ArrayList	with values.	&&	
list = new	*/	valueStack[i].Equals(v	valueStack.RemoveAt(va
ArrayList();	private	alue)) {	<pre>lueStack.Count - 1);</pre>
int[]	ArrayList valueStack =		}
initials;	new ArrayList();		}
,		return true;	}
// Find	/**	}	}
next token	* Checks	}	}
descriptions	if the specified name	return	,
initials =	is on the stack.	false;	namespace Core.Library
set.GetInitialTokens()	*	laise,	framespace core. Eibrary
set. detilitiatiokens ()		J	/**
; for (int i	* @param	/**	/** * ⟨remarks⟩An
	name the	,	
= 0; i <	name to search for	* Clears	enumeration with token
initials.Length; i++)	*	the stack. This method	and production node
1	* @return	removes all elements	*
11	true if the name is on	on	constants.
list.Add(GetTokenDescr	the stack, or	* the	*/
iption(initials[i]));	*	stack.	public enum
}	false otherwise	*/	SyntaxConstants
	*/	public	{
// Create	public	void Clear() {	$MAIN_N = 1001$ ,
exception	bool Contains(string		PRINT_N =
throw new	name) {	nameStack.Clear();	1002,
ParserCreationExceptio	return		$SCAN_N = 1003$ ,
n (	nameStack.Contains(nam	valueStack.Clear();	CONST_N =
	e);	}	1004,
ParserCreationExceptio	}		RETURN = 1005,
<del>-</del>		/**	•
		•	

	SWITCH_N =		S_OBRACKET =		
1006,	CASE_N = 1007,	1051,	S_CBRACKET =	PROD_GLOBAL_DEC = 2012,	PROD_ELEM_TWO_TAIL = 2036,
	BREAK = 1008, FOR N = 1009,	1052,	DOLLAR = 1053,	PROD LOCALDEC CHOICE =	PROD ASSIGN CHOICE =
	IF = 1010,		POWER = 1054,	2013,	2037,
1011	ELSEIF_N =		HASH = 1055,	DDOD DEC CHOICE -	DDOD ACCECC ACCION DAY
1011,	ELSE_N = 1012,		INT = 1056, CHAR = 1057,	PROD_DEC_CHOICE = 2014,	PROD_ACCESS_ASSIGN_DTY PE = 2038,
	DO = 1013,		FLOAT = 1058,		
1014,	WHILE_N =		$STRING = 1059, \\ BOOL_N = 1060,$	PROD_GLOBAL_CHOICE = 2015,	PROD_ASSIGN_VALUE_CHOI CE = 2039,
1014,	VOID = 1015,		ID = 1061,	2010,	PROD_ASSIGNING
	GETCH = 1016,		NUM = 1062,	PROD_BODY_CHOICE =	= 2040,
1017,	STRUCT_N =	1063,	DECIMAL =	2016,	PROD_ARRAY_ID = 2041,
	DEFAULT =		$S_{CHAR} = 1064,$	PROD_VAR_UNIT_BODY =	
1018,	PLUS = 1019,		TEXT = 1065, COM = 1066,	2017,	PROD_ARRAY_IDTAIL = 2042,
	MINUS = 1020,		YES = 1067,	PROD_FUNCT_RETURN_BODY	_ ,
	TIMES = 1021, DIVIDE = 1022,		NO = 1068, FUNCTNAME =	= 2018,	PROD_ASSIGN_SYM = 2043.
	MODULUS =	1069,	FUNCTIVAME -	PROD_FUNCT_VOID_BODY =	2043,
1023,	DOWN G. 1004	1050	STRUCTNAME =	2019,	PROD_ASSIGN_VALUE =
	EQUALS = 1024, SEMIC = 1025,	1070,	IDSTRUCT =	PROD_ARR_UNIT_BODY =	2044,
	DOT = 1026,	1071,	D 1050	2020,	PROD_FUNCT_PARAM =
	COMMA = 1027, AND = 1028,		F = 1072, $D = 1073,$	PROD_ARR_TYPE = 2021,	2045,
	OR = 1029,		S = 1074,	PROD_N1 =	PROD_FUNCT_IDPARAM =
	NOT = 1030, INCREMENT =		ZERO = 1075, SPACE = 1076,	2022,	2046,
1031,			$N_LINE = 1077$ ,	PROD_ARRAY_CHOICE =	PROD_ADDFUNCT_IDPARAM
1032,	DECREMENT =	1078,	WHITESPACE =	2023, PROD N2 =	= 2047, PROD FUNCT =
,	P_E = 1033,			2024,	2048,
	M_E = 1034, T E = 1035,	PROD_S 2001,	TART_PROGRAM =	PROD_INDEX1 = 2025,	PROD FUNCT RETURN =
	D_E = 1036,	2001,	PROD_PROGRAM =	PROD_ADD =	2049,
	MOD_E = 1037, NEWLINE =	2002,	PROD COMMENTS	2026, PROD INDEX2 =	PROD FUNCT VOID =
1038,	NEWEINE	= 2003	_	2027,	2050,
	N_E = 1039, O PAREN =	= 2004	PROD_DATATYPE	PROD_INDEX_EX = 2028,	PROD_DTYPE_A = 2051,
1040,	O_I AREN =	- 2004	PROD_LITERALS	PROD_UNIT_AID	PROD_EXDTYPE_A
1041,	C_PAREN =	= 2005	, PROD CONSTANT	= 2029,	= 2052, PROD DTYPEF =
1041,	D_QUOTE =	= 2006	_	PROD_UNIT_AIDTWO =	2053,
1042,	COLON - 1049	DDOD I	OCAL CHOICE -	2030,	PROD_EX_ID =
	COLON = 1043, O_BRACKET =	2007,	OCAL_CHOICE =	PROD_UNIT_ELEM = 2031,	2054, PROD_ARR_INDEX
1044,	C DDACKET	0000	PROD_LOCALDEC	PROD_EXTELEM =	= 2055,
1045,	C_BRACKET =	= 2008	,	2032,	PROD_STRUCT_U = 2056,
1040	GREATER =	_	NITADD_ID =	PROD_EXTELEM_CHOICE =	PROD_S_DEC =
1046,	LESS = 1047,	2009,		2033,	2057, PROD_INDEX =
1040	GREATER_E =	_	NIT_EXINIT =	PROD_UNIT_ELEM_TWO =	2058,
1048,	LESS E = 1049,	2010,	PROD MAIN =	2034,	PROD_BODY = 2059,
	$NOT_{E} = 1050,$	2011,		PROD_ELEM_TWO_LIT =	PROD_PRINT =
				2035,	2060,

DDOD DOGWIAL			
PROD_POSTVAL =	DDOD CACECTATEMENT -	private string	.11:
2061,	PROD_CASESTATEMENT = 2085,	<pre>RecursiveProductions = "";</pre>	public List <int></int>
PROD CONCAT LIT =	PROD_MATH_OP =	, private	GetAllProductionCode()
2062,	2086,	List <int></int>	{
PROD SCAN =	PROD OPER COND	ProductionCode = new	return
2063,	= 2087,	List <int>();</int>	this.ProductionCode;
PROD EXT I =	,	private	}
2064,	PROD_OPER_COND_CHOICE	List <string></string>	
PROD_FOR_STATE	= 2088,	ProductionState = new	public void
= 2065,	PROD_OPER_SYM	List <string>();</string>	AddProduction(string
	= 2089,		Productions)
PROD_FORSTATEMENT =	PROD_OPER_EQ =	public void	{
2066,	2090,	AddProductionCode(int	
PROD_VAL1 =		code)	this.Productions +=
2067,	PROD_OPER_EXT_S =	{	Productions;
PROD_MNT_COND	2091,		}
= 2068,		this.ProductionCode.Ad	public string
	PROD_OPER_EXT_REP =	d(code);	GetProductions()
PROD_MNT_COND_T =	2092,	}	{
2069,	PROD_OPERAND =		return
PROD_MNT =	2093,	public int	this. Productions;
2070,	DDOD GIN MATH OD	GetLastProductionCode(	}
PROD_IFELSE =	PROD_SIM_MATH_OP =	)	public void
2071,	2094,	1	AddRecursiveProduction
DDOD IECONDITION -	PROD S MATH EXT =	int last = ProductionCode.Count -	(string
PROD_IFCONDITION = 2072,	2095,		RecursiveProductions)
2012,	2090,	1;	l
PROD IFSTATEMENT =	PROD OPER COND EXT =	return this.ProductionCode[la	this.RecursiveProducti
2073,	2096,	st];	ons +=
PROD ELSEIF =	PROD_REL_OP =	StJ, }	RecursiveProductions;
2074,	2097,	J	Recursive roductions,
2011,	PROD_RELOP_EXT	public void	public string
PROD ELSEIFSTATEMENT =	= 2098,	AddProductionState(str	GetRecursiveProduction
2075,	PROD_OP1 =	ing state)	s()
	2099,	{	{
PROD ELSE STATE =	PROD LOG OP =	•	return
2076,	2100,	this.ProductionState.A	this.RecursiveProducti
		dd(state);	ons;
PROD_ELSESTATEMENT =	PROD_EXT_LOG_OP =	}	}
2077,	2101,		}
PROD_DOWHILE =	PROD_LOG_OPER	public string	}
2078,	= 2102,	${\tt GetLastProductionState}$	
	PROD_END =	()	/*
PROD_DOSTATEMENT =	2103	{	* Token.cs
2079,	}	int last =	*/
	}	ProductionState.Count	
PROD_WHILE_STATE =		- 1;	using System. Text;
2080,	using	return	
DD 0D WWW. DOWN WDW.	System. Collections. Gen	this.ProductionState[1	namespace Core.Library
PROD_WHILESTATEMENT =	eric;	ast];	{
2081,	0 1.1	}	/
DDOD CWITCH CTATE	namespace Core.Library	.11:	/**
PROD_SWITCH_STATE =	t	public	* A token node.
2082,	public class	List <string></string>	This class represents
DDOD CASE STATE -	SyntaxProductions	GetAllProductionState(	a token (i.e. a set of
PROD_CASE_STATE = 2083,	private string	, ,	adjacent * characters) in
PROD DEF =	Productions = "";	return	a parse tree. The
2084,	i i odde ti olis – ,	this.ProductionState;	tokens are created by
2001,		}	a tokenizer,
		,	a concurrent,

* that groups	character in the token	image.IndexOf('\n',	*
characters together	image.	pos) >= 0;) {	*/
into tokens according	*/	pos =	public
to a set of	private int	image.IndexOf('\n',	override int StartLine
* token patterns.	endColumn;	pos) + 1;	{
*			get {
	/**	this.endLine++;	return
*	* The		startLine;
*/	previous token in the	endColumn =	}
public class Token	list of tokens.	image.Length - pos;	}
: Node {	*/	}	
	private Token	}	/**
/**	previous = null;		* The column
* The token		/**	number property of the
pattern used for this	/**	* The node	first character in
token.	* The next	type id property	this
*/	token in the list of	(read-only). This	* node (read-
private	tokens.	value is set as	only). If the node has
TokenPattern pattern;	*/	* a unique	child elements, this
	private Token	identifier for each	* value will
/**	next = null;	type of node, in order	be fetched from the
* The		to	first child.
characters that	/**	* simplify	*
constitute this token.	* Creates a	later identification.	*
This is normally	new token.	*	*/
* referred to	*	*	public
as the token image.	* @param	*/	override int
*/	pattern the	public	StartColumn {
private string	token pattern	override int Id {	get {
image;	* @param	get {	return
	image the	return	startColumn;
/**	token image (i.e.	pattern. Id;	}
* The line	characters)	}	}
number of the first	* @param line	}	
character in the token	the line number of the		/**
image.	first character	/**	* The line
*/	* @param col	* The node	number property of the
private int	the column number of	name property (read-	last character in this
startLine;	the first character	only).	node
	*/	*	* (read-
/**	public	*	only). If the node has
* The column	Token (TokenPattern	*/	child elements, this
number of the first	pattern, string image,	public	value
character in the token	int line, int col) {	override string Name {	* will be
image.		get {	fetched from the last
*/	this.pattern =	return	child.
private int	pattern;	pattern.Name;	*
startColumn;	this.image	}	*
	= image;	}	*/
/**			public
* The line	this.startLine = line;	/**	override int EndLine {
number of the last		* The line	get {
character in the token	this.startColumn =	number property of the	return
image.	col;	first character in	endLine;
*/		this	}
private int	this.endLine = line;	* node (read-	}
endLine;		only). If the node has	
,	this.endColumn = col +	child elements, this	/**
/**	image.Length - 1;	* value will	* The column
* The column	for (int	be fetched from the	number property of the
number of the last	pos = 0;	first child.	last character in this

* node (read-	* The token		The next token may be
only). If the node has	pattern property	<pre>previous.next = this;</pre>	a token that
child elements, this	(read-only).	}	* was ignored
* value will	*/	}	during the parsing,
be fetched from the	internal	}	due to it's ignore
last child.	TokenPattern Pattern {	,	flag
*	get {	/**	* being set.
*	return	* Returns the	If there is no next
*/	pattern;	previous token. The	token or if the token
public	}	previous token may be	list
override int EndColumn	}	a token	* feature
{		* that has	wasn't used in the
get {	/**	been ignored in the	tokenizer (the
return	* The	parsing. Note that if	default), the
endColumn;	previous token	the token	* next token
}	property. If the token	* list	will always be null.
}	list feature is	feature hasn't been	*
	* used in the	used in the tokenizer,	* @see
/**	tokenizer, all tokens	this method	#Previous
* The token	found will be chained	* will always	* @see
image property (read-	* together in	return null. By	Tokenizer#UseTokenList
only). The token image	a double-linked list.	default the token list	*
* consists of	The previous token may	feature is	*
the input characters	be	* not used.	*/
matched to form this	* a token	*	public Token
* token.	that was ignored	* @return the	Next {
*	during the parsing,	previous token, or	get {
*	due to it's	*	return
*/	* ignore flag	null if no such token	next;
public string	being set. If there is	is available	}
Image {	no previous token or	*	set {
get {	if	* @see	if
return	* the token	#Previous	(next != null) {
image;	list feature wasn't	* @see	
}	used in the tokenizer	#GetNextToken	<pre>next.previous = null;</pre>
}	(the	* @see	}
	* default),	Tokenizer#UseTokenList	next =
/**	the previous token	*	value;
* Returns the	will always be null.	*	if
token image. The token	*	*	(next != null) {
image consists of the	* @see #Next	* @deprecated	
* input	* @see	Use the Previous	<pre>next.previous = this;</pre>
characters matched to	Tokenizer#UseTokenList	property instead.	}
form this token.	*	*/	}
*	*	public Token	}
* @return the	*/	<pre>GetPreviousToken() {</pre>	
token image	public Token	return	/**
*	Previous {	Previous;	* Returns the
* @see #Image	get {	}	next token. The next
*	return		token may be a token
* @deprecated	previous;	/**	that has
Use the Image property	}	* The next	* been
instead.	set {	token property. If the	ignored in the
*/	if	token list feature is	parsing. Note that if
public string	(previous != null) {	used	the token list
GetImage() {		* in the	* feature
return	<pre>previous.next = null;</pre>	tokenizer, all tokens	hasn't been used in
Image;	}	found will be chained	the tokenizer, this
}		together	method will
	<pre>previous = value;</pre>	* in a	* always
/**	if	double-linked list.	return null. By
	(previous != null) {		

1.614.41.41.41.41.4	1. [. 11. 1]	: c	(" . 1 ") TI
default the token list	<pre>image[newline - 1] == '\r') {</pre>	if	("words"). The
feature is not	\r ) {	(newline > 0 &&	grouping is
* used. *	1:	<pre>image[newline - 1] == '\r') {</pre>	* controlled by
	newline;	\r ) (	token patterns that
* @return the	}	1.	contain either a fixed
next token, or	1 CC A 1/: C	newline;	string to
*	buffer. Append (image. Su	}	* search for, or
null if no such token	<pre>bstring(0, newline));</pre>	1 00 1 1/: 0	a regular expression.
is available	1 00 1 1/1/( )/1)	buffer. Append (image. Su	If the stream of
*	buffer.Append("()")	<pre>bstring(0, newline));</pre>	characters
* @see #Next	;		* don't match any
* @see	} else {	buffer.Append("()")	of the token patterns,
#GetPreviousToken		;	a parse exception is
* @see	buffer.Append(image);	} else {	thrown.
Tokenizer#UseTokenList	}		*
*		buffer.Append(image);	
*	buffer.Append("\",	}	*
*	line: ");		*/
* @deprecated		<pre>buffer.Append('"');</pre>	public class
Use the Next property	buffer.Append(startLin	if	Tokenizer {
instead.	e);	(pattern.Type ==	
*/		TokenPattern.PatternTy	/**
public Token	buffer.Append(", col:	pe.REGEXP) {	* The token
<pre>GetNextToken() {</pre>	") ;		list feature flag.
return		<pre>buffer.Append(" &lt;");</pre>	*/
Next;	buffer.Append(startCol		private bool
}	umn);	buffer. Append (pattern.	useTokenList = false;
		Name);	
/**	return		/**
* Returns a	<pre>buffer.ToString();</pre>	<pre>buffer.Append("&gt;");</pre>	* The string
string representation	}	}	DFA token matcher.
of this token.			This token matcher
*	/**	return	uses a
* @return a	* Returns a	<pre>buffer.ToString();</pre>	*
string representation	short string	}	deterministic finite
of this token	representation of this	}	automaton (DFA)
*/	token. The	}	implementation and is
public	* string will		* used for
override string	only contain the token	/*	all string token
ToString() {	image and possibly the	* Tokenizer.cs	patterns. It has a
	* token	*/	slight speed
StringBuilder buffer	pattern name.		* advantage
= new StringBuilder();	*	using System;	to the NFA
int	* @return a	using	implementation, but
newline =	short string	System. Collections;	should be equivalent
<pre>image. IndexOf('\n');</pre>	representation of this	using System. IO;	* on memory
	token	using System. Text;	usage.
	*/	using	*/
buffer. Append (pattern.	public string	System. Text. RegularExp	private
Name);	ToShortString() {	ressions;	StringDFAMatcher
• •		using Core.Library.RE;	stringDfaMatcher;
<pre>buffer. Append("(");</pre>	StringBuilder buffer	,	
	= new StringBuilder();	namespace Core.Library	/**
buffer. Append (pattern.	int	{	* The regular
Id);	newline =		expression NFA token
	image. IndexOf('\n');	/**	matcher. This token
buffer. Append("):	image. Indexel ( (ii ) ,	* A character	matcher
\"");		stream tokenizer. This	* uses a non-
if	<pre>buffer.Append('"');</pre>	class groups the	deterministic finite
$(\text{newline} >= 0) $ {	if	characters read	automaton (DFA)
if	$(\text{newline} >= 0)$ {	* from the stream	implementation
(newline > 0 &&	(Hewithe /- O) (	together into tokens	impiementa(10H
(HOWITHO / O WW		CORCUMENT THEO CONCHS	

	mi.		
* and is used	* The		*
for most regular	previous token in the	this.regExpMatcher =	false otherwise
expression token	token list.	new	*
patterns. It is	*/	RegExpMatcher(ignoreCa	* @see
* somewhat	private Token	se);	#UseTokenList
faster than the other	<pre>previousToken = null;</pre>		* @see
recursive regular	,	this.buffer = new	#SetUseTokenList
expression	/**	ReaderBuffer(input);	* @see
*	* Creates a	}	Token#GetPreviousToken
implementations	new case-sensitive	/	* @see
available, but doesn't	tokenizer for the	/**	Token#GetNextToken
support the full	specified	* The token	*
* syntax. It	* input	list flag property. If	*
conserves memory by	stream.	the token list flag is	*
using a fast queue	*	* set, all	* @deprecated
instead of	* @param	tokens (including	Use the UseTokenList
* the stack	input the	ignored tokens) link	property instead.
during processing (no	input stream to read	to each * other in a	*/
stack overflow).	*/	0 0 110 1 111 0	public bool
*/	public	double-linked list. By	GetUseTokenList() {
private	Tokenizer (TextReader	default the token list	return
NFAMatcher nfaMatcher;	input)	* flag is set	useTokenList;
/	:	to false.	}
/**	this(input, false) {	*	/
* The regular	}	* @see	/**
expression token	/	Token#Previous	* Sets the
matcher. This token	/**	* @see	token list feature
matcher is	* Creates a	Token#Next	flag. The token list
* used for	new tokenizer for the	*	feature makes
complex regular	specified input	*	* all tokens
expressions, but	stream. The	*/	(including ignored
should be avoided	* tokenizer	public bool	tokens) link to each
* due to	can be set to process	UseTokenList {	other in a
possibly degraded	tokens either in	get {	* linked list
speed and memory usage	* case-	return	when active. By
compared to	sensitive or case-	useTokenList;	default the token list feature is
* the	insensitive mode. *	got (	
automaton		set {	* not used. *
implementations.	* @param	wasTakanliat = walua.	
*/	input the	useTokenList = value;	* @param useTokenList the
private	input stream to read	) 1	token list feature
RegExpMatcher	* @param	l	
regExpMatcher;	ignoreCase the	/**	flag *
/**	character case ignore flag	* Checks if	* @see
* The	11ag *	the token list feature	
character stream	*	is used. The token	#UseTokenList * @see
reader buffer.	*/	list	#GetUseTokenList
*/	public	* feature	* @see
private	Tokenizer (TextReader	makes all tokens	Token#GetPreviousToken
ReaderBuffer buffer =	input, bool	(including ignored	* @see
null;	ignoreCase) {	tokens) link to	Token#GetNextToken
null,	Ignorecase) (	* each other	roken#Getnextroken *
/**	this.stringDfaMatcher	in a linked list. By	*
* The last	= new	default the token list	*
token match found.	- new StringDFAMatcher(ignor	feature	* @deprecated
*/	eCase);	* is not	Use the UseTokenList
≁/ private	ecase,,	used.	property instead.
TokenMatch lastMatch =	this.nfaMatcher = new	usea. *	property Instead. */
new TokenMatch();	NFAMatcher (ignoreCase)	* * @return	*/ public void
new rondinaten(),	nrama teller (rgnorecase)	* Wreturn true if the token list	SetUseTokenList (bool
/**	,	feature is used, or	useTokenList( 6001
Viene		Touture 15 useu, or	uscionciilist/ (

	*/	}	
this.useTokenList =	public int	catch (Exception e) {	pattern.Name,
useTokenList;	GetCurrentLine() {	(======================================	F,
}	return	throw new	"pattern type" +
,	buffer.LineNumber;	ParserCreationExceptio	pattern. Type +
/**	}	n (	"
* Returns a	,	(	is undefined");
description of the	/**	ParserCreationExceptio	}
token pattern with the	* Returns the	n. ErrorType. INVALID TO	}
* specified	current column number.	KEN,	
id.	This number will be		/**
*	the	pattern.Name,	* Resets this
* @param id	* column		tokenizer for usage
the token pattern id	number of the next	"error adding string	with another input
*	token returned.	token: " +	stream.
* @return the	*		* This method
token pattern	* @return the	e.Message);	will clear all the
description, or	current column number	}	internal state in the
*	*/	break;	* tokenizer
null if not present	public int	case	as well as close the
*/	<pre>GetCurrentColumn() {</pre>	TokenPattern.PatternTy	previous input stream.
public string	return	pe. REGEXP:	It
GetPatternDescription(	buffer.ColumnNumber;	try {	* is normally
int id) {	}		called in order to
		nfaMatcher.AddPattern(	reuse a parser and
TokenPattern pattern;	/**	pattern);	* tokenizer
	* Adds a new	}	pair with multiple
pattern =	token pattern to the	catch (Exception) {	input streams, thereby
stringDfaMatcher.GetPa	tokenizer. The pattern		* avoiding
ttern(id);	will be	try {	the cost of re-
if	* added last		analyzing the grammar
(pattern == null) {	in the list, choosing	regExpMatcher.AddPatte	structures.
	a previous token	rn(pattern);	*
pattern =	pattern in	}	* @param
nfaMatcher.GetPattern(	* case two	catch (Exception e) {	input the new
id);	matches the same		input stream to read
}	string.	throw new	*
if	*	ParserCreationExceptio	* @see
(pattern == null) {	* @param	n (	Parser#reset (Reader)
	pattern the	D C	*
pattern =	pattern to add *	ParserCreationExceptio	*
regExpMatcher. GetPatte		n. ErrorType. INVALID_TO	*/
rn(id);	<pre>* @throws ParserCreationExceptio</pre>	KEN,	public void Reset(TextReader
return	n if the pattern	pattern. Name,	input) {
(pattern == null) ?	couldn't be	pattern. Name,	Input) (
null:	couran t be	"regular expression	this.buffer.Dispose();
pattern. ToShortString(	added to the tokenizer	contains error(s): " +	this. buffer. Dispose (),
);	*/	contains cirol (s).	this.buffer = new
},	public void	e.Message);	ReaderBuffer(input);
,	AddPattern (TokenPatter	}	Roddelballer (Inpat),
/**	n pattern) {	}	this.previousToken =
* Returns the	switch	break;	null;
current line number.	(pattern. Type) {	default:	marr,
This number will be	case	throw	this.lastMatch.Clear()
the line	TokenPattern.PatternTy	new	;
* number of	pe. STRING:	ParserCreationExceptio	. }
the next token	try {	n (	•
returned.	7	•	/**
*	stringDfaMatcher.AddPa	ParserCreationExceptio	* Finds the
* @return the	ttern(pattern);	n.ErrorType.INVALID_TO	next token on the
current line number		KEN,	

stream. This method		try {	
will return	token = null;		e.Message,
* null when	} else	<pre>lastMatch.Clear();</pre>	
end of file has been	if		-1,
reached. It will	(token. Pattern. Error)	stringDfaMatcher.Match	
return a	{	(buffer, lastMatch);	-1);
* parse			}
exception if no token	throw new	nfaMatcher.Match(buffe	}
matched the input	ParseException(	r, lastMatch);	
stream, or if		_, , ,	/**
* a token	ParseException.ErrorTy	regExpMatcher.Match(bu	* Factory
pattern with the error	pe. INVALID TOKEN,	ffer, lastMatch);	method for creating a
flag set matched. Any	pe. IIWILID_TOILIN,	if	new token. This method
tokens	token. Pattern. ErrorMes	(lastMatch.Length > 0)	can be
		(lastmatch.Length / 0)	
* matching a	sage,	1	* overridden
token pattern with the	. 1	1.	to provide other token
ignore flag set will	token.StartLine,	line =	implementations than
be		buffer.LineNumber;	the
* silently	token.StartColumn);		* default
ignored and the next	}	column =	one.
token will be	} while	buffer.ColumnNumber;	*
returned.	(token == null);		* @param
*	return	str =	pattern the
* @return the	token;	buffer.Read(lastMatch.	token pattern
next token found, or	}	Length);	* @param
*		_	image the
null if end of file	/**	return	token image (i.e.
was encountered	* Finds the	NewToken(lastMatch.Pat	characters)
*	next token on the	tern, str, line,	* @param line
* @throws	stream. This method	column);	the line number of the
ParseException if the	will return	else	first character
input stream couldn't	* null when	if (buffer. Peek(0) <	* @param
•			
be read or	end of file has been	0) {	column the
·	reached. It will		column number of the
parsed correctly	return a	return null;	first character
*/	* parse	} else	*
public Token	exception if no token	{	* @return the
Next() {	matched the input		token created
Token	stream.	line =	*
token = null;	*	buffer.LineNumber;	*
	* @return the		*/
do {	next token found, or	column =	protected
token	*	buffer.ColumnNumber;	virtual Token
= NextToken();	null if end of file		NewToken(TokenPattern
if	was encountered	throw new	pattern,
(token == null) {	*	ParseException(	
, (	* @throws	,	string image,
<pre>previousToken = null;</pre>	ParseException if the	ParseException.ErrorTy	sering image,
previous roken narr,	input stream couldn't	pe. UNEXPECTED CHAR,	int line,
roturn null:	be read or	pe. UNEXI ECTED_CHAR,	int line,
return null;		1 C.C D 1(1)	:
}	*	buffer.Read(1),	int column) {
if	parsed correctly		
(useTokenList) {	*/	line,	return new
			Token(pattern, image,
	private Token		
token.Previous =	NextToken() {	column);	line, column);
<pre>token.Previous = previousToken;</pre>	*	}	line, column); }
	NextToken() {	<pre>column); } catch</pre>	line, column); }
	NextToken() {     string	}	line, column); } /**
<pre>previousToken;</pre>	<pre>NextToken() {           string      str;</pre>	} catch	}
<pre>previousToken;</pre>	NextToken() {	<pre>} } catch (IOException e) {</pre>	} /**
<pre>previousToken; previousToken = token; } if</pre>	NextToken() {	} catch (IOException e) { throw new	/**  * Returns a  string representation
<pre>previousToken; previousToken = token; }</pre>	NextToken() {	} } catch (IOException e) { throw	/** * Returns a

* string will	*/	* @return the	representation of this
contain the details of	protected bool	token pattern found,	matcher
all the token patterns	<pre>ignoreCase = false;</pre>	or *	*/
* contained in this tokenizer.	/**	null if not found	public override string
in this tokenizer.	* Creates a	#/	ToString() {
* @return a	new token matcher.	public	105t111ig() (
detailed string	*	TokenPattern	StringBuilder buffer
representation	* @param	GetPattern(int id) {	= new StringBuilder();
*/	ignoreCase the	for (int i	new Stringstriati () ,
public	character case ignore	= 0; i <	for (int i
override string	flag	patterns. Length; i++)	= 0; i <
ToString() {	*/	{	patterns.Length; i++)
	public	if	{
StringBuilder buffer	TokenMatcher(bool	(patterns[i].Id == id)	
<pre>= new StringBuilder();</pre>	ignoreCase) {	{	buffer. Append (patterns
			[i]);
	this.ignoreCase =	return patterns[i];	
buffer.Append(stringDf	ignoreCase;	}	buffer.Append("\n\n");
aMatcher);	}	}	}
	,	return	return
buffer. Append (nfaMatch	/**	null;	<pre>buffer. ToString();</pre>
er);	* Searches	}	}
1 CC 4 1/ F W	for matching token	/	}
buffer. Append (regExpMa	patterns at the start	/** * Adds a	
tcher);	of the		/**
return buffer.ToString();	* input stream. If a match is	string token pattern to this matcher.	* A token pattern
burrer. rostring(),	found, the token match	*	matcher using a DFA
}	object	* @param	for string tokens.
J	* is updated.	pattern the	This
	*	pattern to add	* class only
/**	* @param	*	supports string tokens
* A token pattern	buffer the	* @throws	and must be
matcher. This class is	input buffer to check	Exception if the	complemented
the base class for the	* @param	pattern couldn't be	* with another
* various types	match the	added to the matcher	matcher for regular
of token matchers that	token match to update	*/	expressions.
exist. The token	*	public virtual	Internally it
matcher	* @throws	void	* uses a DFA to
* checks for	IOException if an I/O	AddPattern(TokenPatter	provide high
matches with the	error occurred	n pattern) {	performance.
tokenizer buffer, and	*/	A	*/
maintains the	public	Array. Resize (ref	internal class
* state of the	abstract void Match(ReaderBuffer	patterns, patterns.Length + 1);	StringDFAMatcher:
last match. */	buffer, TokenMatch	patterns. Length + 1);	TokenMatcher {
internal abstract	match);	patterns[patterns.Leng	/**
class TokenMatcher {	match,	th - 1] = pattern;	* The
crass Tokenmatener (	/**	th Ij pattern,	deterministic finite
/**	* Returns the	,	state automaton used
* The array	token pattern with the	/**	for
of token patterns.	specified id. Only	* Returns a	* matching.
*/	* token	string representation	*/
protected	patterns handled by	of this matcher. This	private
TokenPattern[]	this matcher can be	will	TokenStringDFA
patterns = new	returned.	* contain all	automaton = new
TokenPattern[0];	*	the token patterns.	<pre>TokenStringDFA();</pre>
	* @param id	*	
/**	the token pattern id	* @return a	/**
* The ignore	*	detailed string	
character case flag.			

* Creates a	buffer, TokenMatch	*/	* @throws
new string token	match) {	public	IOException if an I/O
=	maten) (	*	•
matcher.	T. 1 . D. 11	NFAMatcher (bool	error occurred
*	TokenPattern res =	ignoreCase):	*/
* @param	automaton. Match (buffer	base(ignoreCase) {	public
ignoreCase the	, ignoreCase);	}	override void
character case ignore			Match(ReaderBuffer
flag	if (res !=	/**	buffer, TokenMatch
*/	null) {	* Adds a	match) {
public		token pattern to this	
StringDFAMatcher(bool	match.Update(res.Patte	matcher.	automaton.Match(buffer
ignoreCase) :	rn.Length, res);	*	, match);
<pre>base(ignoreCase) {</pre>	}	* @param	}
}	}	pattern the	}
	}	pattern to add	
/**		*	
* Adds a		* @throws	/**
string token pattern	/**	Exception if the	* A token pattern
to this matcher.	* A token pattern	pattern couldn't be	matcher for complex
*	matcher using a NFA	added to the matcher	regular expressions.
* @param	for both string and	*/	This
•	· ·	public	
1	* regular	_	* class only
pattern to add	expression tokens.	override void	supports regular
*/	This class has limited	AddPattern (TokenPatter	expression tokens and
public	support for	n pattern) {	must be
override void	* regular	if	* complemented
AddPattern (TokenPatter	expressions and must	(pattern.Type ==	with another matcher
n pattern) {	be complemented with	TokenPattern.PatternTy	for string tokens.
	another	pe.STRING) {	* Internally it
automaton.AddMatch(pat	* matcher		uses the Grammatica RE
tern.Pattern,	providing full regular	automaton.AddTextMatch	package for high
ignoreCase, pattern);	expression support.	(pattern. Pattern,	* performance or
	Internally	ignoreCase, pattern);	the native
base.AddPattern(patter	* it uses a NFA	} else {	java.util.regex
n);	to provide high		package for maximum
}	performance and low	automaton.AddRegExpMat	* compatibility.
·	memory	ch (pattern. Pattern,	*/
/**	* usage.	ignoreCase, pattern);	internal class
* Searches	*/	}	RegExpMatcher:
for matching token	internal class	,	TokenMatcher {
patterns at the start	NFAMatcher:	base. AddPattern(patter	Tokenma tener
of the	TokenMatcher {	n);	/**
	TOKETIMA CETTET	11),	,
* input stream. If a match is	/**	l	* The regular
	,	/state	expression handlers.
found, the token match	* The non-	/**	*/
object	deterministic finite	* Searches	private
* is updated.	state automaton used	for matching token	REHandler[] regExps =
*	for	patterns at the start	<pre>new REHandler[0];</pre>
* @param	* matching.	of the	,
buffer the	*/	* input	/**
input buffer to check	private	stream. If a match is	* Creates a
* @param	TokenNFA automaton =	found, the token match	new regular expression
match the	new TokenNFA();	object	token matcher.
token match to update		* is updated.	*
*	/**	*	* @param
* @throws	* Creates a	* @param	ignoreCase the
IOException if an I/O	new NFA token matcher.	buffer the	character case ignore
error occurred	*	input buffer to check	flag
*/	* @param	* @param	*/
public	ignoreCase the	match the	public
override void	character case ignore	token match to update	RegExpMatcher(bool
Match (ReaderBuffer	flag	*	<u> </u>
,	· ·		

ignoreCase) :	* input	*	public
base(ignoreCase) {	stream. If a match is	zero (0) if no match	GrammaticaRE(string
}	found, the token match	was found	regex, bool
	object	*	ignoreCase) {
/**	* is updated.	* @throws	regExp =
* Adds a	*	IOException if an I/O	new RegExp(regex,
regular expression	* @param	error occurred	ignoreCase);
token pattern to this	buffer the	*/	}
matcher.	input buffer to check	public	
*	* @param	abstract int	/**
* @param	match the	Match(ReaderBuffer	* Checks if
pattern the	token match to update	buffer);	the start of the input
pattern to add	*	}	stream matches this
*	* @throws	•	* regular
* @throws	IOException if an I/O		expression.
Exception if the	error occurred	/**	*
pattern couldn't be	*/	* The Grammatica	* @param
added to the matcher	public	built-in regular	buffer the
*/	override void	expression handler.	input buffer to check
public	Match (ReaderBuffer	*/	*
override void	buffer, TokenMatch	internal class	* @return the
			longest match found,
AddPattern (TokenPatter	match) {	GrammaticaRE : REHandler {	,
n pattern) {	for (int i	Kchandier (	or
REHandler	= 0; i <		*
re;	regExps.Length; i++) {	/**	zero (0) if no match
	int	* The	was found
try {	length =	compiled regular	*
re =	regExps[i].Match(buffe	expression.	* @throws
new	r);	*/	IOException if an I/O
GrammaticaRE(pattern.P	if	private RegExp	error occurred
attern, ignoreCase);	$(length > 0)$ {	regExp;	*/
			public
pattern.DebugInfo =	match.Update(length,	/**	override int
"Grammatica regexp\n"	patterns[i]);	* The regular	Match(ReaderBuffer
+ re;	}	expression matcher to	buffer) {
} catch	}	use.	if
(Exception) {	}	*/	(matcher == null) {
re =	}	private	
new		Matcher matcher =	matcher =
SystemRE(pattern.Patte		null;	regExp.Matcher(buffer)
rn, ignoreCase);	/**		;
· -	* The regular	/**	} else {
pattern.DebugInfo =	expression handler	* Creates a	,
"native .NET regexp";	base class.	new Grammatica regular	matcher.Reset(buffer);
}	*/	expression handler.	}
,	internal abstract	*	return
Array. Resize (ref	class REHandler {	* @param	matcher.MatchFromBegin
regExps,	orabb insianaror (	regex the	ning() ?
regExps. Length + 1);	/**	regular expression	matcher.Length(): 0;
regemps, length 17,	* Checks if	text	materier. Length () . 0,
regExps[regExps.Length	the start of the input	* @param	,
- 1] = re;		•	J
- IJ - Ie,	stream matches this	_	
1 A1ID ++ / -++	* regular	character case ignore	/starts
base. AddPattern(patter	expression.	flag	/**
n);	*	*	* The .NET system
}	* @param	* @throws	regular expression
	buffer the	Exception if the	handler.
/**	input buffer to check	regular expression	*/
* Searches	*	contained	internal class
for matching token	* @return the	*	SystemRE : REHandler {
patterns at the start	longest match found,	invalid syntax	,
of the	or	*/	/**

* The parsed	* @throws	* pattern	return
regular expression.	IOException if an I/O	identifiers if two	pattern;
*/	error occurred	matches have the same	}
private Regex	*/	length.	}
reg;	public	*	
	override int		/**
/**	Match(ReaderBuffer	*	* Updates
* Creates a	buffer) {	*	this match with new
new .NET system	Match m;	*/	values. The new values
regular expression		internal class	will only
handler.	// Ugly	TokenMatch {	* be
*	hack since .NET	(	considered if the
* @param	doesn't have a flag	/**	length is longer than
regex the	for when the	* The length	any previous match
regular expression	// end of	of the longest match.	* found.
text	the input string was	*/	*
* @param	encountered	private int	* @param
ignoreCase the	cheountereu	length = 0;	length the
character case ignore	buffer.Peek(1024 *	rength 0,	matched length
flag	16);	/**	* @param
11ag *	// Also,	* The pattern	pattern the
* @throws	there is no API to	in the longest match.	matched pattern
Exception if the	limit the search to	in the longest match.	matched pattern */
•	the specified	private	public void
regular expression	•	_	•
contained *	//	TokenPattern pattern =	Update(int length, TokenPattern pattern)
•	position, so we double-check the index	null;	TokenPattern pattern)
invalid syntax		/steste	( :£
*/	afterwards instead.	/**	if
public	m =	* Clears the	(this.length < length)
SystemRE(string regex,	reg. Match (buffer. ToStr	current match	{
bool ignoreCase) {	ing(),	information.	
if	buffer.Position);	*/	this.length = length;
(ignoreCase) {	if	public void	
reg =	(m. Success && m. Index	Clear() {	this.pattern =
new Regex (regex,	== buffer.Position) {	length =	pattern;
RegexOptions.IgnoreCas	return	0;	} else if
e);	m. Length;	pattern =	(this.length == length
} else {	} else {	null;	&& this.pattern.Id >
reg =	return	}	pattern.Id) {
new Regex(regex);	0;	,	
}	}	/**	this.length = length;
}	}	* The length	
	}	of the longest match	this.pattern =
/**	}	found (read-only).	pattern;
* Checks if		*/	}
the start of the input	/*	public int	}
stream matches this	* TokenMatch.cs	Length {	}
* regular	*/	get {	}
expression.		return	
*	namespace Core.Library	length;	/*
* @param	{	}	* TokenNFA.cs
buffer the		}	*/
input buffer to check	/**		
*	* The token match	/**	using System;
* @return the	status. This class	* The token	
longest match found,	contains logic to	pattern for the	namespace Core.Library
or	ensure that	longest match found	{
*	* only the	(read-only).	
zero (0) if no match	longest match is	*/	/**
was found	considered. It also	public	* A non-
*	prefers lower token	TokenPattern Pattern {	deterministic finite
		get {	

state automaton (NFA)	toxt state error i e	state	
for matching	text state array, i.e. non-ASCII	= initial.AddOut(ch,	TokenPattern value) {
* tokens. It	* or complex	ignoreCase, null);	Tokem attern value) (
supports both fixed	transitions (such as	ignorecase, nuii),	
strings and simple	regular expressions).	for (int i	TokenRegExpParser
regular	*/	= 1; i < str. Length;	parser = new
* expressions,	private	i++) {	TokenRegExpParser(patt
but should perform	NFAState initial = new	state	ern, ignoreCase);
similar to a DFA due	NFAState();	= state.AddOut(str[i],	string
to highly	THE COUNTY ,	ignoreCase, null);	debug = "DFA regexp; "
* optimized data	/**	}	+
structures and tuning.	* The NFA	,	parser.GetDebugInfo();
The memory footprint	state queue to use.	state.value = value;	bool
during	*/	}	isAscii;
* matching should	private	,	,
be near zero, since no	NFAStateQueue queue =	/**	isAscii =
heap memory is	new NFAStateQueue();	* Adds a	parser. start. IsAscii0u
allocated	, , , , , , , , , , , , , , , , , , ,	regular expression	tgoing();
* unless the pre-	/**	match to this	for (int i
allocated queues need	* Adds a	automaton. New states	= 0: isAscii && i <
to be enlarged. The	string match to this	* and	128; i++) {
NFA also	automaton. New states	transitions will be	bool
* does not use	and	added to extend this	match = false:
recursion, but	* transitions	automaton to	for
iterates in a loop	will be added to	* support the	(int j = 0; j < 0)
instead.	extend this automaton	specified string. Note	parser. start. outgoing.
*	to support	that this method only	Length; j++) {
*	* the	* supports a	if
*	specified string.	subset of the full	(parser.start.outgoing
*/	*	regular expression	[j]. Match((char) i)) {
internal class	* @param str	syntax, so	
TokenNFA {	the string to match	* a more	if (match) {
	* @param	complete regular	
/**	ignoreCase the	expression library	isAscii = false;
* The initial	case-insensitive match	must also be	
state lookup table,	flag	* provided.	break;
indexed by the first	* @param	*	
ASCII	value the	* @param	}
* character.	match value	pattern the	
This array is used to	*/	regular expression	match = true;
for speed optimizing	public void	string	}
the	AddTextMatch(string	* @param	}
* first step	str, bool ignoreCase,	ignoreCase the	if
in the match, since	TokenPattern value) {	case-insensitive match	(match &&
the initial state	NFAState	flag	initialChar[i] !=
would	state;	* @param	nu11) {
* otherwise	char	value the	
have a long list of	ch = str[0];	match value	isAscii = false;
transitions to		*	}
consider.	if (ch <	* @throws	}
*/	128 && !ignoreCase) {	RegExpException if the	if
private	state	regular expression	(parser.start.incoming
NFAState[] initialChar	= initialChar[ch];	parsing	.Length > 0) {
= new NFAState[128];	if	*	
	(state == null) {	failed	initial.AddOut(new
/**		*/	NFAEpsilonTransition(p
* The initial	state =	public void	arser.start));
state. This state	initialChar[ch] = new	AddRegExpMatch(string	debug
contains any	NFAState();	pattern,	+= ", uses initial
transitions not	}		epsilon";
* already	} else {	bool ignoreCase,	
stored in the initial			

} else if	* @param		internal
(isAscii &&	match the	this.queue.MarkEnd();	TokenPattern value =
!ignoreCase) {	token match to update	peekChar =	null;
for	*	buffer. Peek(1);	marr,
(int i = 0; isAscii &&	* @return the	561161 <b>(</b> 17, <b>)</b>	/**
i < 128; i++) {	number of characters	// The	* The incoming
1 (120, 1 ) (	matched, or	remaining match loop	transitions to this
for (int $j = 0$ ; $j <$	*	processes all	state.
parser. start. outgoing.	zero (0) if no match	subsequent states	*/
Length; j++) {	was found	while	internal
, , , ,	*	(!this.queue.Empty) {	NFATransition[]
if	* @throws	if	incoming = new
(parser. start. outgoing	IOException if an I/O	(this.queue.Marked) {	NFATransition[0];
[j]. Match((char) i)) {	error occurred	(enris queue marnes) (	
2331 ( ( ) - / / )	*/	pos++;	/**
initialChar[i] =	public int	pos ,	* The outgoing
parser. start. outgoing[	Match (ReaderBuffer	peekChar =	transitions from this
j]. state;	buffer, TokenMatch	buffer. Peek(pos);	state.
51. 50000,	match) {	bullet. From (pob),	*/
}	int	this.queue.MarkEnd();	internal
}	length = 0;	}	NFATransition[]
}	int	state	outgoing = new
debug	pos = 1;	=	NFATransition[0];
+= ", uses ASCII	int	this.queue.RemoveFirst	Williams Cloned,
lookup",	peekChar;	();	/**
} else {	NFAState	if	* The outgoing
) cisc (	state;	(state.value != null)	epsilon transitions
parser.start.MergeInto	state,	(State: variae : marr)	flag.
(initial);	// The	(	*/
debug	first step of the	match. Update (pos,	internal bool
+= ", uses initial	match loop has been	state. value);	epsilonOut = false;
state";	unrolled and	state. value,	epsironout raise,
}	//	if	/**
J	optimized for	(peekChar >= 0) {	* Checks if
parser.end.value =	performance below.	(peckenar / 0) (	this state has any
value;	performance below.	state.MatchTransitions	incoming or outgoing
varae,	this.queue.Clear();	((char) peekChar,	*
value.DebugInfo =	peekChar =	this.queue, false);	transitions.
debug;	buffer. Peek(0);	}	*
}	if (0 <=	}	* @return
,	peekChar && peekChar <	return	true if this state has
/**	128) {	length;	transitions, or
* Checks if	state	}	*
this NFA matches the	=	}	false otherwise
specified input text.	this.initialChar[peekC	,	*/
The	har];		public bool
* matching	if	/**	HasTransitions() {
will be performed from	(state != null) {	* An NFA state.	return
position zero (0) in	(State Finally (	The NFA consists of a	incoming. Length > 0
the	this.queue.AddLast(sta	series of states, each	outgoing. Length > 0;
* buffer.	te);	* having zero or	}
This method will not	}	more transitions to	,
read any characters	}	other states.	/**
from the	if	*/	* Checks if
* stream,	(peekChar >= 0) {	internal class	all outgoing
just peek ahead.	(positional / v/ (	NFAState {	transitions only match
*	this.initial.MatchTran	-12.125.0000	ASCII
* @param	sitions ((char)	/**	* characters.
buffer the	peekChar, this.queue,	* The optional	* characters.
input buffer to check	true);	state value (if it is	* @return
input builter to encer	}	a final state).	true if all
	J	a IIIIaI State). */	truc ir ali
		• /	

transitions are ASCII-	* @param	* @return the	outgoing =
only, or	state the	transition target	null;
onry, or <b>*</b>	target state, or null	state	11411, }
false otherwise	*	*/	J
*/	* @return the	public	/**
public bool	transition target	NFAState	* Finds a
IsAsciiOutgoing() {	state	AddOut(NFATransition	unique character
for (int i	*/	trans) {	transition if one
= 0; i <	public	trans, (	exists. The
outgoing. Length; i++)	NFAState AddOut(char	Array. Resize (ref	* transition
{	ch, bool ignoreCase,	outgoing,	must be the only
if	NFAState state) {	outgoing. Length + 1);	matching single
(!outgoing[i].IsAscii(	if	odegoing. Bengen 1/,	character
)) {	(ignoreCase) {	outgoing[outgoing.Leng	* transition
,, (	if	th - 1] = trans;	and no other
return false;	(state == null) {	if (trans	transitions may reach
}	(State Hall) (	is	the same
}	state = new	NFAEpsilonTransition)	* state.
return	NFAState();	{	*
true;	}		* @param ch
}	,	epsilonOut = true;	the character to
,	AddOut (new	}	search for
/**	NFACharTransition(Char	return	*
* Adds a new	. ToLower (ch), state));	trans. state;	* @return the
incoming transition.	, ,	}	unique transition
*	AddOut (new	,	state found, or
* @param	NFACharTransition(Char	/**	*
trans the	. ToUpper(ch), state));	* Merges all	null if not found
transition to add	return	the transitions in	*/
*/	state;	this state into	private
public void	} else {	another	NFAState
AddIn(NFATransition	if	* state.	FindUniqueCharTransiti
trans) {	(state == null) {	*	on(char ch) {
, (	(=====,	* @param	(
Array. Resize (ref	state =	state the state	NFATransition res =
incoming,	FindUniqueCharTransiti	to merge into	null;
incoming.Length + 1);	on(ch);	*/	
	if	public void	NFATransition trans;
incoming[incoming.Leng	(state != null) {	MergeInto(NFAState	
th - 1] = trans;		state) {	for (int i
}	return state;	for (int i	= 0; i <
	}	= 0; i <	outgoing.Length; i++)
/**		incoming.Length; i++)	{
* Adds a new	state = new	{	trans
outgoing character	NFAState();		= outgoing[i];
transition. If the	}	state.AddIn(incoming[i	if
target	return	]);	(trans.Match(ch) &&
* state	AddOut (new		trans is
specified was null and	NFACharTransition(ch,	<pre>incoming[i].state =</pre>	NFACharTransition) {
an identical	state));	state;	if
transition	}	}	(res != null) {
* already	}	incoming =	
exists, it will be		null;	return null;
reused and its target	/**	for (int i	}
returned.	* Adds a new	= 0; i <	
*	outgoing transition.	outgoing.Length; i++)	res = trans;
* @param ch	*	{	}
he character to match	* @param		}
* @param	trans the	state.AddOut(outgoing[	for (int i
ignoreCase the	transition to add	i]);	= 0; res != null && i
case-insensitive flag	*	}	<pre>&lt; outgoing.Length;</pre>
			i++) {

trans	if	}	*
= outgoing[i];	(initial && trans is	}	false otherwise
if	NFAEpsilonTransition)	J	*/
(trans != res &&	{		public
trans. state ==	(	/**	abstract bool
res. state) {	target.MatchTransition	* An NFA state	IsAscii();
res. state, (	s (ch, queue, true);	transition. A	15/15011 (/ ,
return null;	} else	transition checks a	/**
}	if (trans.Match(ch)) {	single	* Checks if
}	II (dranst materi (dr.))	* character of	the specified
return	queue.AddLast(target);	input an determines if	character matches the
(res == null) ? null :	if	it is a match. If a	transition.
res. state;	(target.epsilonOut) {	match	*
}	(	* is encountered,	* @param ch
•	target.MatchEmpty(queu	the NFA should move	the character to check
/**	e);	forward to the	*
* Attempts a	}	transition	* @return
match on each of the	}	* state.	true if the character
transitions leading	}	*/	matches, or
from	}	internal abstract	*
* this state.	,	class NFATransition {	false otherwise
If a match is found,	/**		*/
its state will be	* Adds all	/**	public
added	the epsilon transition	* The target	abstract bool
* to the	targets to the	state of the	Match(char ch);
queue. If the initial	specified	transition.	material en,
match flag is set,	* queue.	*/	/**
epsilon	*	internal	* Creates a
* transitions	* @param	NFAState state;	copy of this
will also be matched	queue the state	,	transition but with
(and their targets	queue	/**	another target
called	*/	* Creates a	* state.
*	public void	new state transition.	*
recursively).	MatchEmpty(NFAStateQue	*	* @param
*	ue queue) {	* @param	state the new
* @param ch	•	state the	target state
the character to match	NFATransition trans;	target state	*
* @param	NFAState	*/	* @return an
queue the state	target;	public	identical copy of this
queue	-	NFATransition(NFAState	transition
* @param	for (int i	state) {	*/
initial the initial	= 0; i <	this.state	public
match flag	outgoing.Length; i++)	= state;	abstract NFATransition
*/	{		<pre>Copy(NFAState state);</pre>
public void	trans	this.state.AddIn(this)	}
MatchTransitions(char	= outgoing[i];	,	
ch, NFAStateQueue	if	}	
queue, bool initial) {	(trans is		/**
	NFAEpsilonTransition)	/**	* The special
NFATransition trans;	{	* Checks if	epsilon transition.
NFAState		this transition only	This transition
target;	<pre>target = trans.state;</pre>	matches ASCII	matches the
		characters.	* empty input,
for (int i	queue.AddLast(target);	* I. e.	i.e. it is an
= 0; i <	if	characters with	automatic transition
outgoing.Length; i++)	(target.epsilonOut) {	numeric values between	that doesn't
{		0 and 127.	* read any input.
trans	target.MatchEmpty(queu	*	As such, it returns
= outgoing[i];	e);	* @return	false in the match
target	}	true if this	method
= trans.state;	}	transition only	* and is handled
	}	matches ASCII, or	specially everywhere.

*/	return	this.match	* @return an
internal class	false;	= match;	identical copy of this
NFAEpsilonTransition:	}	}	transition
NFATransition {			*/
	/**	/**	public
/**	* Creates a	* Checks if	override NFATransition
* Creates a	copy of this	this transition only	Copy(NFAState state) {
new epsilon	transition but with	matches ASCII	return new
transition.	another target	characters.	NFACharTransition(matc
*	* state.	* I. e.	h, state);
* @param	*	characters with	}
state the	* @param	numeric values between	}
target state	state the new	0 and 127.	
*/	target state	*	
public	*	* @return	/**
NFAEpsilonTransition(N	* @return an	true if this	* A character
FAState state) :	identical copy of this	transition only	range match transition. Used for
base(state) {	transition */	matches ASCII, or	user-defined
J	public	false otherwise	* character sets
/**	override NFATransition	*/	in regular
* Checks if	Copy (NFAState state) {	public	expressions.
this transition only	return new	override bool	*/
matches ASCII	NFAEpsilonTransition(s	IsAscii() {	internal class
characters.	tate);	return 0	NFACharRangeTransition
* I. e.	}	<= match && match <	: NFATransition {
characters with	}	128;	
numeric values between		}	/**
0 and 127.			* The inverse
*	/**	/**	match flag.
* @return	* A single	* Checks if	*/
true if this	character match	the specified	protected bool
transition only	transition.	character matches the	inverse;
matches ASCII, or	*/	transition.	,
*	internal class	*	/**
false otherwise	NFACharTransition:	* @param ch	* The case-
*/	NFATransition {	the character to check	insensitive match
public override bool	/**	* * @return	flag. */
IsAscii() {	* The	true if the character	protected bool
return	character to match.	matches, or	ignoreCase;
false;	*/	*	ignorcoase,
}	protected char	false otherwise	/**
,	match;	*/	* The
/**	,	public	character set content.
* Checks if	/**	override bool	This array may contain
the specified	* Creates a	Match(char ch) {	either
character matches the	new character	return	* range
transition.	transition.	this.match == ch;	objects or Character
*	*	}	objects.
* @param ch	* @param		*/
the character to check	match the	/**	private
*	character to match	* Creates a	object[] contents =
* @return	* @param	copy of this	<pre>new object[0];</pre>
true if the character	state the	transition but with	/
matches, or	target state	another target	/**
false atherwise	*/	* state.	* Creates a
false otherwise  */	public	* @param	new character range
*/ public	NFACharTransition(char match, NFAState state)	* @param state the new	transition. *
override bool	: base(state) {	target state	<b>т</b>
Match(char ch) {	. 5450 (31410) (	target State *	

* @param	if	max =	obj =
inverse the	(obj is char) {	Char. ToLower (max);	contents[i];
inverse match flag	c c	}	if
* @param	= (char) obj;		(obj is char) {
ignoreCase the	if	AddContent(new	c
case-insensitive match	$(c < 0 \mid \mid 128 \le c) $ {	Range(min, max));	= (char) obj;
flag		}	if
* @param	return false;	/stute	$(c == ch) {$
state the target state	} else	/** * Adds an	return !inverse;
*/	if (obj is Range) {	object to the	return : inverse,
public	if	character set content	} else
NFACharRangeTransition	(!((Range)	array.	if (obj is Range) {
(bool inverse,	obj).IsAscii()) {	*	r
		* @param obj	= (Range) obj;
bool ignoreCase,	return false;	the object to add	if
	}	*/	(r.Inside(ch)) {
NFAState state) :	}	private void	
base(state) {	}	AddContent(Object obj)	return !inverse;
	return	{	}
this.inverse =	true;	A D	}
inverse;	}	Array. Resize (ref contents,	return
this.ignoreCase =	/**	contents, contents. Length + 1);	inverse;
ignoreCase;	* Adds a	Contents. Length 17,	inverse,
}	single character to	contents contents. Leng	,
,	this character set.	th - 1] = obj;	/**
/**	*	}	* Creates a
* Checks if	* @param c		copy of this
this transition only	the character to add	/**	transition but with
matches ASCII	*/	* Checks if	another target
characters.	public void	the specified	* state.
* I. e.	AddCharacter(char c) {	character matches the	*
characters with	if	transition.	* @param
numeric values between	(ignoreCase) {	*	state the new
0 and 127.	Chan Tallawan(a)	* @param ch the character to check	target state *
* @return	Char. ToLower(c);	the character to check	* @return an
true if this	J	* @return	identical copy of this
transition only	AddContent(c);	true if the character	transition
matches ASCII, or	}	matches, or	*/
*		*	public
false otherwise	/**	false otherwise	override NFATransition
*/	* Adds a	*/	Copy(NFAState state) {
public	character range to	public	
override bool	this character set.	override bool	NFACharRangeTransition
IsAscii() {	*	Match(char ch) {	copy;
object	* @param min	object	
obj; char c;	the minimum character value	obj; char c;	<pre>copy = new NFACharRangeTransition</pre>
char c;	* @param max	char c; Range r;	(inverse, ignoreCase,
if	the maximum character	Range 1,	state);
(inverse) {	value	if	State,
return	*/	(ignoreCase) {	copy.contents =
false;	public void	ch =	contents;
}	AddRange(char min,	Char. ToLower(ch);	return
for (int i	char max) {	}	copy;
= 0; i <	if	for (int i	}
contents.Length; i++)	(ignoreCase) {	= 0; i <	,
{	min =	contents. Length; i++)	/**
obj =	Char. ToLower(min);	1	* A character
contents[i];			range class.

ate /	ate Charalan		N. (A
*/ private class	* Checks if the specified	numeric values between 0 and 127.	* @param
	character is inside	0 and 121.	state the new
Range {	the range.	* @return	target state *
/**	the range.	true if this	* @return an
* The	* @param	transition only	identical copy of this
minimum character	c the	matches ASCII, or	transition
value.	character to check	*	*/
*/	*	false otherwise	public
private	* @return	*/	override NFATransition
char min;	true if the character	public	Copy(NFAState state) {
,	is in the range, or	override bool	return new
/**	*	IsAscii() {	NFADotTransition(state
* The	false otherwise	return	);
maximum character	*/	false;	}
value.	public	}	}
*/	bool Inside(char c) {		
private	return	/**	
char max;	min <= c && c <= max;	* Checks if	/**
	}	the specified	* The digit
/**	}	character matches the	character set
* Creates	}	transition.	transition. This
a new character range.		*	transition matches a
*		* @param ch	* single numeric
* @param	/**	the character to check	character.
min the minimum	* The dot ('.')	*	*/
character value	character set	* @return	internal class
* @param	transition. This	true if the character	NFADigitTransition :
max the maximum	transition	matches, or	NFATransition {
character value	* matches a	*	/
*/	single character that	false otherwise	/ <b>**</b>
public	is not equal to a	*/	* Creates a
Range(char min, char	newline	public	new digit character
max) {	* character. */	override bool Match(char ch) {	set transition. *
this.min = min;	internal class	switch	* @param
tills, min	NFADotTransition :	(ch) {	state the
this.max = max;	NFATransition {	case '\n':	target state
}	THE THE TOTAL CONTROL OF THE TOTAL CONTROL OT THE TOTAL CONTROL OF THE T	case '\r':	*/
,	/**	case	public
/**	* Creates a	'\u0085':	NFADigitTransition(NFA
* Checks	new dot character set	case	State state) :
if this range only	transition.	'\u2028':	base(state) {
matches ASCII	*	case	}
characters	* @param	'\u2029':	
*	state the	return	/**
* @return	target state	false;	* Checks if
true if this range	*/	default:	this transition only
only matches ASCII, or	public	return	matches ASCII
*	NFADotTransition(NFASt	true;	characters.
false otherwise	ate state) :	}	* I. e.
*/	base(state) {	}	characters with
public	}	/state	numeric values between
bool IsAscii() {	/ steale	/**	0 and 127.
return 0 <= min && min < 128	/** * Checks if	* Creates a copy of this	* * @return
0 \- min && min \ 120	this transition only	transition but with	true if this
au.	matches ASCII	another target	transition only
0 <= max && max < 128;	characters.	* state.	matches ASCII, or
}	* I.e.	*	*
,	characters with	•	false otherwise
/**	CHALGO COLO HIVII		*/
,			,

public	*/	public	* Checks if
override bool	internal class	override bool	this transition only
IsAscii() {	NFANonDigitTransition	Match(char ch) {	matches ASCII
return	: NFATransition {	return ch	characters.
true;		< '0'    '9' < ch;	* I. e.
}	/**	}	characters with
	* Creates a		numeric values between
/**	new non-digit	/**	0 and 127.
* Checks if	character set	* Creates a	*
the specified	transition.	copy of this	* @return
character matches the	*	transition but with	true if this
transition.	* @param	another target	transition only
*	state the	* state.	matches ASCII, or
* @param ch	target state	*	*
the character to check	*/	* @param	false otherwise
*	public	state the new	*/
* @return	NFANonDigitTransition(	target state	public
true if the character	NFAState state) :	*	override bool
matches, or	base(state) {	* @return an	IsAscii() {
*	}	identical copy of this	return
false otherwise		transition	true;
*/	/**	*/	}
public	* Checks if	public	
override bool	this transition only	override NFATransition	/**
Match(char ch) {	matches ASCII	Copy(NFAState state) {	* Checks if
return '0'	characters.	return new	the specified
<= ch && ch <= '9';	* I. e.	NFANonDigitTransition(	character matches the
}	characters with	state);	transition.
,	numeric values between	}	*
/**	0 and 127.	}	* @param ch
* Creates a	*		the character to check
copy of this	* @return		*
transition but with	true if this	/**	* @return
another target	transition only	* The whitespace	true if the character
* state.	matches ASCII, or *	character set	matches, or
*	•	transition. This	•
* @param state the new	false otherwise */	transition * matches a	false otherwise */
	public	single whitespace	public
target state *	override bool	character.	override bool
* @return an	IsAscii() {	*/	Match(char ch) {
identical copy of this	return	internal class	switch
transition	false;	NFAWhitespaceTransitio	(ch) {
*/	}	n: NFATransition {	case ' ':
public	j	ii . idiiiidiistotoii (	case '\t':
override NFATransition	/**	/**	case '\n':
Copy (NFAState state) {	* Checks if	* Creates a	case '\f':
return new	the specified	new whitespace	case '\r':
NFADigitTransition(sta	character matches the	character set	case
te);	transition.	transition.	(char) 11:
}	*	*	return
}	* @param ch	* @param	true;
	the character to check	state the	default:
	*	target state	return
/**	* @return	*/	false;
* The non-digit	true if the character	public	}
character set	matches, or	NFAWhitespaceTransitio	}
transition. This	*	n(NFAState state) :	
transition	false otherwise	base(state) {	/**
* matches a	*/	}	* Creates a
single non-numeric			copy of this
character.		/**	

transition but with	* @return	* @return an	return
another target	true if this	identical copy of this	true;
* state.	transition only	transition	}
*	matches ASCII, or	*/	/
* @param	*	public	/**
state the new	false otherwise	override NFATransition	* Checks if
target state	*/	Copy(NFAState state) {	the specified
*	public	return new	character matches the
* @return an	override bool	NFANonWhitespaceTransi	transition.
identical copy of this	IsAscii() {	tion(state);	*
transition	return	}	* @param ch
*/	false;	}	the character to check
public override NFATransition	,		* * @return
	/**	/**	
Copy (NFAState state) {	,	,	true if the character
return new NFAWhitespaceTransitio	* Checks if the specified	* The word character set	matches, or *
	character matches the	transition. This	false otherwise
n(state);	transition.	transition matches a	*/
}	*	* single word	public
J	* @param ch	character.	override bool
	the character to check	character. */	Match(char ch) {
/**	the character to check	internal class	return
* The non-	* @return	NFAWordTransition:	('a' <= ch && ch <=
whitespace character	true if the character	NFATransition {	'z')
set transition. This	matches, or	WAITAIIST CIOII (	Z )
transition	*	/**	('A' <= ch && ch <=
* matches a	false otherwise	* Creates a	'Z')
single non-whitespace	*/	new word character set	]
character.	public	transition.	('0' <= ch && ch <=
*/	override bool	*	'9')
internal class	Match (char ch) {	* @param	ch
NFANonWhitespaceTransi	switch	state the	== ' ':
tion : NFATransition {	(ch) {	target state	- <b>,</b>
tion / millianortion (	case ' ':	*/	,
/**	case '\t':	public	/**
* Creates a	case '\n':	NFAWordTransition(NFAS	* Creates a
new non-whitespace	case '\f':	tate state) :	copy of this
character set	case '\r':	base(state) {	transition but with
transition.	case	}	another target
*	(char) 11:		* state.
* @param	return	/**	*
state the	false;	* Checks if	* @param
target state	default:	this transition only	state the new
*/	return	matches ASCII	target state
public	true;	characters.	*
NFANonWhitespaceTransi	}	* I. e.	* @return an
tion(NFAState state) :	}	characters with	identical copy of this
base(state) {		numeric values between	transition
}	/**	0 and 127.	*/
	* Creates a	*	public
/**	copy of this	* @return	override NFATransition
* Checks if	transition but with	true if this	Copy(NFAState state) {
this transition only	another target	transition only	return new
matches ASCII	* state.	matches ASCII, or	NFAWordTransition(stat
characters.	*	*	e);
* I. e.	* @param	false otherwise	}
characters with	state the new	*/	}
numeric values between	target state	public	
0 and 127.	*	override bool	
*		IsAscii() {	/**

st. Th	ate.		
* The non-word	*	marker index is used	private int
character set	false otherwise	to	mark = 0;
transition. This	*/	* separate the	/steate
transition matches * a single non-	public override bool	current from the	/**
word character.	Match(char ch) {	subsequent states.*	* The empty
word character.	bool word	* The queue	queue property (read- only).
internal class	= ('a' <= ch && ch <=	implementation is	*/
NFANonWordTransition:	- ( a <- cn & cn <-	optimized for quick	public bool
NFATransition {	2 )	removal at the	Empty {
Williams Clon	('A' <= ch && ch <=	* beginning and	get {
/**	'Z')	addition at the end.	return
* Creates a	L	It will attempt to use	(last <= first):
new non-word character	('0' <= ch && ch <=	a	{(1450 ( 11150),
set transition.	'9')	* fixed-size	}
*	,	array to store the	,
* @param	ch == '_';	whole queue, and moves	/**
state the	return	the data	* The marked
target state	!word;	* in this array	first entry property
*/	}	only when absolutely	(read-only). This is
public		needed. The array is	set
NFANonWordTransition(N	/**	also	* to true if
FAState state) :	* Creates a	* enlarged	the first entry in the
base(state) {	copy of this	automatically if too	queue has been marked.
}	transition but with	many states are being	*/
	another target	processed	public bool
/**	* state.	* at a single	Marked {
* Checks if	*	time.	get {
this transition only	* @param	*/	return
matches ASCII	state the new	internal class	first == mark;
characters.	target state	NFAStateQueue {	}
* I. e.	*		}
characters with	* @return an	/**	
numeric values between	identical copy of this	* The state	/**
0 and 127.	transition	queue array. Will be	* Clears this
*	*/	enlarged as needed.	queue. This operation
* @return	public	*/	is fast, as it just
true if this	override NFATransition	private	* resets the
transition only	Copy(NFAState state) {	NFAState[] queue = new	queue position
matches ASCII, or	return new NFANonWordTransition(s	NFAState[2048];	indices. */
false otherwise	tate):	/**	public void
*/	(ate),	* The position	Clear() {
public	}	of the first entry in	first = 0;
override bool	J	the queue (inclusive).	last = 0;
IsAscii() {		*/	mark = 0:
return	/**	private int	}
false;	* An NFA state	first = 0;	,
}	queue. This queue is	,	/**
,	used during processing	/**	* Marks the
/**	to	* The position	end of the queue. This
* Checks if	* keep track of	just after the last	means that the next
the specified	the current and	entry in the queue	entry
character matches the	subsequent NFA states.	* (exclusive).	* added to
transition.	The	*/	the queue will be
*	* current state	private int	marked (when it
* @param ch	is read from the	last = 0;	becomes the
the character to check	beginning of the		* first in
*	queue, and new	/**	the queue). This
* @return	* states are	* The current	operation is fast.
true if the character	added at the end. A	queue mark position.	*/
matches, or		*/	

public void	queue, queue.Length *	/**	*
MarkEnd() {	2);	* The	* @see
mark =	} else	string pattern type is	#errorMessage
last;	{	used for tokens that	*/
}		only	private bool
	Array. Copy (queue,	* match	error = false;
/**	first, queue, O, last	an exact string.	
* Removes and	- first);	*/	/**
returns the first		STRING,	* The token
entry in the queue.	last -= first;		error message. This
This		/**	message will only be
* operation	mark -= first;	* The	set if the
is fast, since it will	first = 0:	regular expression	* token error
only update the index of	11rst - 0;	pattern type is used for tokens	flag is set.
* the first	}	* that	* @see #error
entry in the queue.	J	match a regular	*/
*	queue[last++] = state;	expression.	private string
* @return the	}	*/	errorMessage = null;
previous first entry	}	REGEXP	_
in the queue	}	}	/**
*/			* The token
public	/*	/**	ignore flag. If this
NFAState RemoveFirst()	* TokenPattern.cs	* The token	flag is set, it means
{	*/	pattern identity.	that the
if (first		*/	* token
< last) {	using System;	private int	should be ignored if
£:	using System.Text;	id;	found. If an ignore
first++;	namespace Core.Library	/**	message is * present in
queue[first - 1];	{	* The token	the ignoreMessage
} else {	(	pattern name.	variable, it will also
return	/**	*/	be reported
null;	* A token	private string	* as a
}	pattern. This class	name;	warning.
}	contains the		*
	definition of a token	/**	* @see
/**	* (i.e. it's	* The token	#ignoreMessage
* Adds a new	pattern), and allows	pattern type.	*/
entry at the end of	testing a string	*/	private bool
the queue. This	against this	private	ignore = false;
operation	* pattern. A	PatternType type;	/
* is mostly fast, unless all the	token pattern is uniquely identified by	/**	/** * The token
allocated queue space	an integer id,	* The token	ignore message. If
has	* that must be	pattern.	this message is set
* already	provided upon	*/	when the token
been used.	creation.	private string	* ignore flag
*	*	pattern;	is also set, a warning
* @param			message will be
state the	*	/**	printed if
state to add	*/	* The token	* the token
*/	public class	error flag. If this	is found.
public void	TokenPattern {	flag is set, it means	*
AddLast (NFAState	/	that an	* @see
state) {	/**	* error	#ignore
if (last	* The pattern	should be reported if	*/
>= queue.Length) {     if	type enumeration. */	the token is found.	private string
(first <= 0) {	*/ public enum	The error * message is	ignoreMessage = null;
(11130 / 0) [	PatternType {	present in the	/**
Array. Resize (ref	racterini, po (	errorMessage variable.	/ test-
		or or mossage furtante.	

token pattern         /**         public         true, a default           */         * The token         PatternType         message is crea           public         pattern name property         GetPatternType() {         * none           TokenPattern(int id,         (read-only).         return         previously set.           **         type;         *           string name,         *         type;         *           **         public string         /**         public string           Name         *         */*         public string         **           PatternType type,         public string         /**         public string         **           String pattern) {         get {         pattern property         get           string pattern) {         get {         pattern property         get           this.id =         name;         property         get reror;           id;         }         * contains         }           this.name         }         * contains         }           this.name         }         * Returns the         * to be           this.type         /**         Returns the         * to be           this.pattern = <t< th=""><th></th></t<>	
Information mossage,	recated
This is normally set	
when the token pattern is	
analyzed by the tokenizer. unique token pattern type; private string	
window token pattern   type;	string
# identity value.	
Description   Section	urn
debugInfo = muil;	
token pattern id	
/**         * Returns the         * The * The * The * The * The * The * * * The * * * The * * The * * * The * * * The	
# Creates a	
# * @deprecated	
* # @deprecated	
# @param id token pattern id instead.  * # eparam name	
## Operation in the token pattern id a segment to the token pattern name	
# @param name to token pattern name	
# Can be token pattern name	and an
# @param type to token pattern type	ld be
# (Sparam   Sparam   Sparam	atch is
# @param	ting
token pattern         /**         public         true, a default           */         * The token         PatternType         message is crea           public         pattern name property         GetPatternType() {         * none           TokenPattern(int id,         (read-only).         return         previously set.           **         type;         *           string name,         *         type;         *           **         public string         /**         public string           Name {         * The token         Error {           string pattern) {         get {         pattern property         get           string pattern) {         get {         pattern property         get           this.id =         name;         property         error;           id;         }         * contains         error;           id;         *         * contains         error;           id; <th< td=""><td></td></th<>	
#/ * The token public pattern name property GetPatternType () {	erty to
public         pattern name property         GetPatternType() {         * none           TokenPattern(int id,         * type;         * *           string name,         * type;         * *           string name,         * type;         * *           */*         * type;         * *           */*         * pattern property         public type;           Name {         * the token         Error {           string pattern) {         get {         pattern property         get           string pattern) {         get {         pattern property         get           this.id =         name;         property         error;           id;         }         * contains         }           this.name         }         * contains         }           this.name         }         (string or regexp)         **           this.type         /**         * Returns the         * to be           this.pattern =         * Returns the         * to be           this.pattern =         * @return the         *           pattern;         * @return the         *           * @return the         *         * (error & error & er	error
TokenPattern(int id,	ted if
# type; # # # # # # # # # # # # # # # # # # #	was
string name,         *         */         */         */         */         */         */         */         */         */         */         PatternType type,         public string         /**         public string         */         public string         */         */         public string         */         public string         */         public string         */         get         */         public string         */         get         */         */         public string         get         */	
*/ PatternType type,	
PatternType type,         public string         /**         public string           Name {         * The token         Error {           string pattern) {         get {         pattern property         get {           this.id =         name;         property         error;           id;         }         * contains         }           this.name         }         the actual pattern         set           = name;         (string or regexp)         *           this.type         /**         which have         = value;           = type;         * Returns the         * to be           this.pattern =         *         *         = null) {           pattern;         * @return the         *         = null) {           token pattern name         */         errorMessage =         public string         "unrecognized to found";           /**         * @see #Name         Pattern {         found";         pattern identity         * @deprecated         return         }           property (read-only).         Use the Name property         pattern;         }         }	
Name {	haal
string pattern) {         get {         pattern property (read-only). This         get {           this.id =         name;         property         error;           id;         }         * contains         }           this.name         }         the actual pattern         set           = name;         (string or regexp)         *           this.type         /**         which have         = value;           = type;         * Returns the         * to be           token pattern name.         matched.         (error && error &           this.pattern =         * @return the         *           pattern;         * @return the         *           *         public string         "unrecognized to found";           ***         * @see #Name         Pattern {         found";           * The token         * @deprecated         return         }           property (read-only).         Use the Name property         pattern;         }           This         instead.         }	0001
return (read-only). This this.id = name; property error; id; } contains } this.name } contains } this.name } contains } this.type   /** which have   = value; type; *Returns the   *to be   token pattern name.   matched.   (error && error)   this.pattern =   * greturn the   *   pattern; *@return the   *   } token pattern name   */ errorMessage =	. {
this.id = name; property error;  id;	return
id;	1004111
this.name	
this.type	. {
= type;	error
token pattern name. matched. (error && error this.pattern =	
this.pattern =	if
pattern;	
token pattern name	
* public string "unrecognized to found";  ** * Gsee #Name	
/**	1
* The token	oken
pattern identity	١
property (read-only). Use the Name property pattern; This instead. }	}
This instead.	
* property	
contains the unique public string * The	token
token pattern identity GetName() { /** error message	
value. return * Returns te property. The ex	rror
* name; token pattern. message is	
* * * prin	ted
*/ * @return the whenever the to	ken is
public int Id /** token pattern matched. Setting	g the
* The token * error	
get { pattern type property * @see	
(read-only). #Pattern	

at .	d. G	d. Ti	and the account of
* message	* @return the	* The ignore	* is true, it
property also sets the	token error message	flag property. If this	means that the token
error flag to true.	*	property is true, the	should be ignored if
*	* @see	* token	found.
* @see #Error	#ErrorMessage	pattern corresponds to	*
*	*	an ignore token and	* @return
*	* @deprecated	should be	true if the pattern
*/	Use the ErrorMessage	* skipped if	maps to an ignored
public string	property instead.	a match is found.	token, or
ErrorMessage {	*/	*	*
get {	public string	*	false otherwise
return	<pre>GetErrorMessage() {</pre>	*/	*
errorMessage;	return	public bool	* @see
}	ErrorMessage;	Ignore {	#Ignore
set {	}	get {	*
error		return	* @deprecated
= true;	/**	ignore;	Use the Ignore
,	* Sets the	}	property instead.
errorMessage = value;	token error flag and	set {	*/
}	assigns a default	ignore	public bool
}	error message.	= value;	IsIgnore() {
J	*	value,	return
/**	* @see #Error	}	Ignore;
* Checks if	* #See #Ellol	J	ignore,
		/**	J
the pattern	* @deprecated	<i>'</i>	/**
corresponds to an	Use the Error property	* The token	,
error token. If this	instead.	ignore message	* Returns the
* is true, it	*/	property. The ignore	token ignore message
means that an error	public void	message is	if the pattern
should be reported if	SetError() {	* printed	corresponds to
a	Error =	whenever the token is	* an ignored
* matching	true;	matched. Setting the	token.
token is found.	}	ignore	*
*		* message	* @return the
* @return	/**	property also sets the	token ignore message
true if the pattern	* Sets the	ignore flag to true.	*
maps to an error	token error flag and	*	* @see
token, or	assigns the specified	* @see	#IgnoreMessage
*	error	#Ignore	*
false otherwise	* message.	*	* @deprecated
*	*	*	Use the IgnoreMessage
* @see #Error	* @param	*/	property instead.
*	message the	public string	*/
* @deprecated	error message to	IgnoreMessage {	public string
Use the Error property	display	get {	<pre>GetIgnoreMessage() {</pre>
instead.	*	return	return
*/	* @see	ignoreMessage;	IgnoreMessage;
public bool	#ErrorMessage	}	}
IsError() {	*	set {	
return	* @deprecated	ignore	/**
Error;	Use the ErrorMessage	= true;	* Sets the
}	property instead.	or do,	token ignore flag and
,	*/	ignoreMessage = value;	clears the ignore
/**	public void	landiomobbage value,	message.
* Returns the	SetError(string	}	message.
token error message if	message) {	J	* @see
	message/ (	/**	
the pattern	EnnonWooder =	,	#Ignore
corresponds to	ErrorMessage =	* Checks if	*
* an error	message;	the pattern	* @deprecated
token.	)	corresponds to an	Use the Ignore
*	/stat	ignored token. If this	property instead.
	/**		*/

public void	* @return a		} else
SetIgnore() {	token pattern string	<pre>buffer.Append(": \"");</pre>	{
Ignore =	representation		
true;	*/	buffer.Append(ignoreMe	buffer. Append(pattern)
}	public	ssage);	;
/state	override string	1	}
/** * Sets the	ToString() {	<pre>buffer. Append("\"");</pre>	<pre>buffer. Append("\"");</pre>
token ignore flag and	StringBuilder buffer	}	} else {
assigns the specified	= new StringBuilder();	if	, 6136 (
ignore		(debugInfo != null) {	<pre>buffer. Append("&lt;");</pre>
* message.			
*	buffer.Append(name);	buffer. Append (" $\n$ ");	<pre>buffer. Append (name);</pre>
* @param	1CC . A1(" (")	1 o CC - A 1 (1 1 o To C	166 1
message the	<pre>buffer. Append(" (");</pre>	<pre>buffer.Append(debugInf o);</pre>	<pre>buffer. Append ("&gt;");</pre>
ignore message to display	<pre>buffer.Append(id);</pre>	}	J
*	salisivinppolia (14) ,	return	return
* @see	<pre>buffer.Append("): ");</pre>	<pre>buffer.ToString();</pre>	<pre>buffer.ToString();</pre>
#IgnoreMessage	switch	}	}
*	(type) {		}
* @deprecated	Case	/**	}
Use the IgnoreMessage property instead.	PatternType.STRING:	* Returns a short string	/*
*/	<pre>buffer. Append("\"");</pre>	representation of this	*
public void	barrer append ( ( ),	object.	TokenRegExpParser.cs
SetIgnore(string	buffer. Append (pattern)	*	*/
message) {	;	* @return a	
		short string	using System;
IgnoreMessage =	<pre>buffer.Append("\"");</pre>	representation of this	using
message;	break;	object */	System.Collections; using
J	case PatternType.REGEXP:	public string	System.Globalization;
/**	ravvernijpe. idebin .	ToShortString() {	using System. Text;
* The token	<pre>buffer.Append("&lt;&lt;");</pre>		using Core.Library.RE;
debug info message		StringBuilder buffer	
property. This is	buffer.Append(pattern)	<pre>= new StringBuilder();</pre>	namespace Core.Library
normally be	;	int	{
* set when	<pre>buffer.Append("&gt;&gt;");</pre>	newline =	/**
the token pattern is analyzed by the	buller. Append ( // ); break;	pattern.IndexOf('\n');	* A regular
tokenizer.	}	if (type	expression parser. The
*	if (error)	== PatternType.STRING)	parser creates an NFA
*	{	{	for the
*/			* regular
<pre>public string DebugInfo {</pre>	<pre>buffer.Append(" ERROR:</pre>	<pre>buffer.Append("\"");     if</pre>	expression having a
get {	( ),	$(\text{newline} >= 0)$ {	single start and acceptance states.
return	buffer. Append (errorMes	if	*
debugInfo;	sage);	(newline > 0 &&	
}		<pre>pattern[newline - 1]</pre>	*
set {	<pre>buffer. Append("\"");</pre>	== '\r') {	*
11 7 0 1	}	1.	*/
debugInfo = value;	if	newline;	internal class
}	(ignore) {	}	TokenRegExpParser {
J	buffer.Append("	buffer. Append (pattern.	/**
/**	IGNORE");	Substring(0,	* The regular
* Returns a	if	newline));	expression pattern.
string representation	(ignoreMessage !=		*/
of this object.	null) {	buffer.Append("()")	private string
*		;	pattern;

	regular expression		*/
/**	parser. Note	this.ignoreCase =	private void
* The	* that this	ignoreCase;	UpdateStats(NFAState
character case ignore	will trigger the	this.pos =	state, Hashtable
flag.	parsing of the regular	0;	visited) {
*/	expression.	this.end =	if
private bool	*	<pre>ParseExpr(start);</pre>	(!visited.ContainsKey(
ignoreCase;	* @param	if (pos <	state)) {
	pattern the	pattern.Length) {	
/**	regular expression	throw	visited.Add(state,
* The current	pattern	new RegExpException(	state);
position in the	*		
pattern. This variable	* @throws	RegExpException.ErrorT	stateCount++;
is used by	RegExpException if the	ype. UNEXPECTED_CHARACT	for
* the parsing	regular expression	ER,	(int $i = 0$ ; $i <$
methods.	couldn't be		state.outgoing.Length;
*/	*	pos,	i++) {
private int	parsed correctly		
pos;	*/	pattern);	transitionCount++;
	public	}	if
/**	TokenRegExpParser(stri	}	(state.outgoing[i] is
* The start	ng pattern) :		NFAEpsilonTransition)
NFA state for this	this(pattern, false) {	/**	{
regular expression.	}	* Returns the	
*/		debug information for	epsilonCount++;
internal	/**	the generated NFA.	}
NFAState start = new	* Creates a	*	
NFAState();	new regular expression	* @return the	UpdateStats(state.outg
	parser. The regular	debug information for	oing[i].state,
/**	* expression	the generated NFA	visited);
* The end NFA	can be either case-	*/	}
state for this regular	sensitive or case-	public string	}
expression.	insensitive.	GetDebugInfo() {	}
*/	* Note that	if	,
internal	this will trigger the	(stateCount == 0) {	/**
NFAState end = null;	parsing of the regular		* Parses a
	* expression.	UpdateStats(start, new	regular expression.
/**	*	Hashtable());	This method handles
* The number	* @param	}	the Expr
of states found.	pattern the	return	* production
*/	regular expression	stateCount + " states, " +	in the grammar (see
private int	pattern	+	regexp. grammar).
<pre>stateCount = 0;</pre>	* @param	transitionCount + "	*
/**	ignoreCase the character case ignore	transitions, " +	* @param start the
* The number	flag	transitions,	start the initial NFA state
of transitions found.	11ag *	epsilonCount + "	*
*/	* @throws	epsilons";	* @return the
private int	RegExpException if the	cp3110fi3 ,	terminating NFA state
transitionCount = 0:	regular expression	J	*
transferoncount o,	couldn't be	/**	* @throws
/**	*	* Updates the	RegExpException if an
* The number	parsed correctly	statistical counters	error was encountered
of epsilon transitions	*/	for the NFA generated.	in the
found.	public	*	*
*/	TokenRegExpParser(stri	* @param	pattern string
private int	ng pattern, bool	state the	*/
epsilonCount = 0;	ignoreCase) {	current state to visit	private
	, ,	* @param	NFAState
/**	this.pattern =	visited the	ParseExpr(NFAState
* Creates a	pattern;	lookup map of visited	start) {
new case-sensitive		states	

NFAState	*	* Fact	}
end = new NFAState();	* @param	production in the	,
NFAState	start the	grammar (see	/**
subStart;	initial NFA state	regexp. grammar).	* Parses a
NFAState	*	*	regular expression
subEnd;	* @return the	* @param	atom. This method
SubElia,		•	handles the
do {	terminating NFA state	start the initial NFA state	
•			* Atom
if	* @throws	*	production in the
(PeekChar(0) == ' ') {	RegExpException if an	* @return the	grammar (see
(2.12)	error was encountered	terminating NFA state	regexp.grammar).
ReadChar(' ');	in the	*	*
}	*	* @throws	* @param
	pattern string	RegExpException if an	start the
subStart = new	*/	error was encountered	initial NFA state
NFAState();	private	in the	*
subEnd	NFAState	*	* @return the
= ParseTerm(subStart);	ParseTerm(NFAState	pattern string	terminating NFA state
if	start) {	*/	*
(subStart.incoming.Len	NFAState	private	* @throws
gth == 0) {	end;	NFAState	RegExpException if an
, ,	,	ParseFact(NFAState	error was encountered
subStart.MergeInto(sta	end =	start) {	in the
rt);	ParseFact(start);	NFAState	*
} else	while	placeholder = new	pattern string
) eise	(true) {	NFAState();	*/
l .	switch	NFAState(), NFAState	,
-+ AddOu+ (			private NFAState
start. AddOut (new	(PeekChar(0)) {	end;	
NFAEpsilonTransition(s	case -	,	ParseAtom(NFAState
ubStart));	1:	end =	start) {
}	case	ParseAtom(placeholder)	NFAState
if	')':	;	end;
(subEnd.outgoing.Lengt	case	switch	
h == 0	']':	(PeekChar(0)) {	switch
	case	case '?':	(PeekChar(0)) {
(!end.HasTransitions()	'{':	case '*':	case '.':
&& PeekChar(0) !=	case	case '+':	
' ')) {	'}':	case '{':	<pre>ReadChar('.');</pre>
	case	end =	return
<pre>subEnd. MergeInto(end);</pre>	'?':	ParseAtomModifier(plac	start.AddOut(new
} else	case	eholder, end);	NFADotTransition(new
{	'+':	break;	NFAState()));
•	case	}	case '(':
subEnd. AddOut (new	,  , :	if	
NFAEpsilonTransition(e	1 .	(placeholder.incoming.	ReadChar('(');
nd));	return end;	Length > 0 &&	end =
1	return end,	start. outgoing. Length	ParseExpr(start);
} while	default:	> 0) {	rarseExpr(start),
-	deraurt.	/ 0) i	D = 101 - (2)2)
(PeekChar(0) == ' ');			ReadChar(')');
return	$1 - P \cdot \cdot \cdot P \cdot \cdot \cdot (\cdot \cdot \cdot 1)$		
1	<pre>end = ParseFact(end);</pre>	start. AddOut (new	return
end;		NFAEpsilonTransition(p	end;
end; }	<pre>end = ParseFact(end); break;</pre>	<pre>NFAEpsilonTransition(p laceholder));</pre>	
}		<pre>NFAEpsilonTransition(p laceholder));     return</pre>	end; case '[':
} /**		<pre>NFAEpsilonTransition(p laceholder));</pre>	end; case '[': ReadChar('[');
}		<pre>NFAEpsilonTransition(p laceholder));     return</pre>	end; case '[': ReadChar('['); end =
} /**		<pre>NFAEpsilonTransition(p laceholder));</pre>	end; case '[': ReadChar('[');
} /** * Parses a		<pre>NFAEpsilonTransition(p laceholder));</pre>	end; case '[': ReadChar('['); end =
/**  * Parses a regular expression	break;	<pre>NFAEpsilonTransition(p laceholder));</pre>	end; case '[': ReadChar('['); end =
/**  * Parses a regular expression term. This method	break; } } /**	<pre>NFAEpsilonTransition(p laceholder));</pre>	<pre>end;</pre>
/**  * Parses a regular expression term. This method handles the	break;	<pre>NFAEpsilonTransition(p laceholder));</pre>	<pre>end;</pre>
/**  * Parses a regular expression term. This method handles the  * Term	break;  }  /**  * Parses a regular expression	<pre>NFAEpsilonTransition(p laceholder));</pre>	<pre>end;</pre>
/**  * Parses a regular expression term. This method handles the  * Term production in the	break;  }  /**  * Parses a  regular expression  factor. This method	<pre>NFAEpsilonTransition(p laceholder));</pre>	<pre>end;</pre>

case ']':	// Read	UNE LINEVDECTED CHADACT	} else if
case ] :	min and max	ype.UNEXPECTED_CHARACT ER,	(min == 1 && max == -
case '}':	switch	ER,	1) {
case '?':	(ReadChar()) {	pos - 1,	if
case '*':	case '?':	pos 1,	(start.outgoing.Length
case '+':	min =	pattern);	== 1 &&
case ' ':	0;	pattern,	1 &&
throw	max =	,	end.outgoing.Length ==
new RegExpException(	1;	// Read	0 &&
ne. neganpaneep vien (	break;	possessive or	0 444
RegExpException.ErrorT	case '*':	reluctant modifiers	end.incoming.Length ==
ype. UNEXPECTED CHARACT	min =	if	1 &&
ER,	0;	(PeekChar(0) == '?') {	
•	max =	throw	start.outgoing[0] ==
pos,	-1;	new RegExpException(	end.incoming[0]) {
•	break;		0
pattern);	case '+':	RegExpException.ErrorT	
default:	min =	ype. UNSUPPORTED_SPECIA	end. AddOut(start.outgo
return	1;	L_CHARACTER,	ing[0].Copy(end));
ParseChar(start);	max =		} else
}	-1;	pos,	{
}	break;		
	case '{':	pattern);	end. AddOut (new
/**	min =	} else if	NFAEpsilonTransition(s
* Parses a	ReadNumber();	$(PeekChar(0) = '+') $ {	tart));
regular expression	max =	throw	}
atom modifier. This	min;	new RegExpException(	return
method handles	if		end;
* the	(PeekChar(0) == ',') {	RegExpException.ErrorT	} else {
AtomModifier		ype. UNSUPPORTED_SPECIA	throw
production in the	ReadChar(',');	L_CHARACTER,	new RegExpException(
grammar (see			
regexp.grammar).	$\max = -1;$	pos,	RegExpException. ErrorT
*	if		ype. INVALID_REPEAT_COU
* @param	(PeekChar(0) != '}') {	pattern);	NT,
start the	D IV 1 ()	}	0.1 D
initial NFA state	max = ReadNumber();	// 11 11	firstPos,
* @param end	}	// Handle	
the terminal NFA state	}	supported repeaters	pattern);
* * @return the	ReadChar('}');	if (min == 0 && max == 1) {	ì
terminating NFA state	if	return	J
terminating NA State	$(\max == 0 \mid   (\max > 0)$	start. AddOut (new	/**
* @throws	&& min > max)) {	NFAEpsilonTransition(e	* Parses a
RegExpException if an	ccc min / max//	nd));	regular expression
error was encountered	throw new	} else if	character set. This
in the	RegExpException(	(min == 0 && max == -	method handles
*	0 · · ·	1) {	* the
pattern string	RegExpException.ErrorT	if	contents of the
*/	ype. INVALID REPEAT COU	(end. outgoing. Length	'[]' construct in a
private	NT,	== 0) {	regular expression.
NFAState			*
ParseAtomModifier(NFAS	firstPos,	end.MergeInto(start);	* @param
tate start, NFAState		} else	start the
end) {	pattern);	{	initial NFA state
int min =	}		*
0;	break;	end. AddOut (new	* @return the
int max =	default:	NFAEpsilonTransition(s	terminating NFA state
-1;	throw	tart));	*
int	new RegExpException(	}	* @throws
firstPos = pos;		return	RegExpException if an
	RegExpException.ErrorT	start;	

error was encountered			NFADigitTransition(end
in the	PeekChar(1) != ']') {	RegExpException.ErrorT	));
*		ype. UNSUPPORTED_SPECIA	case
pattern string	(2.1)	L_CHARACTER,	'D':
*/	ReadChar('-');		D 101 ()
private NFAState	max = ReadChar();	pos,	ReadChar();
ParseCharSet(NFAState	max - Readchar (),	pattern);	ReadChar();
start) {	range. AddRange (min,	default:	Reddenal (),
NFAState	max);	return	return
<pre>end = new NFAState();</pre>	}	start.AddOut(ReadChar(	start.AddOut(new
	else {	), ignoreCase, new	NFANonDigitTransition(
NFACharRangeTransition		<pre>NFAState());</pre>	end));
range;	range.AddCharacter(min	}	case
char	);	}	's':
min;	}	/	D 101 ()
char	hmaala	/** * Parses a	ReadChar();
max;	break;	regular expression	ReadChar();
if	}	character escape. This	Reductial (),
(PeekChar(0) == '^') {	return	method	return
(r senenar (s)	end;	* handles a	start.AddOut(new
ReadChar('^');	}	single character	NFAWhitespaceTransitio
range		escape in a regular	n(end));
= new	/**	expression.	case
NFACharRangeTransition	* Parses a	*	'S':
(true, ignoreCase,	regular expression	* @param	
end);	character. This method	start the	ReadChar();
} else {	handles	initial NFA state	D 101 ()
range	* a single	*	ReadChar();
<pre>= new NFACharRangeTransition</pre>	normal character in a	* @return the	no.+1170
(false, ignoreCase,	regular expression.	terminating NFA state *	return start.AddOut(new
end);	* @param	* @throws	NFANonWhitespaceTransi
}	start the	RegExpException if an	tion (end));
,	initial NFA state	error was encountered	case
start.AddOut(range);	*	in the	'w':
while	* @return the	*	
$(PeekChar(0) > 0) $ {	terminating NFA state	pattern string	ReadChar();
min =	*	*/	
(char) PeekChar(0);	* @throws	private	ReadChar();
switch	RegExpException if an	NFAState	
(min) {	error was encountered	ParseEscapeChar(NFASta	return
case	in the	te start) { NFAState	start.AddOut(new NFAWordTransition(end)
J •	pattern string	end = new NFAState();	);
return end:	*/	ond new minotate (),	case
case	private	if	'W':
`\\`:	NFAState	(PeekChar(0) == '\\'	
	ParseChar(NFAState	<b>&amp;&amp;</b> PeekChar(1) > 0) {	ReadChar();
range.AddCharacter(Rea	start) {	switch	
dEscapeChar());	switch	((char) PeekChar(1)) {	ReadChar();
	(PeekChar(0)) {	case	
break;	case '\\':	'd':	return
1.614.	return	D 101 ()	start. AddOut (new
default:	ParseEscapeChar(start)	ReadChar();	NFANonWordTransition(e
ReadChar(min);	; case '^':	ReadChar();	nd));
reacciai (min),	case '\$':	reactial (),	}
(PeekChar(0) == '-' &&	throw	return	return
. ,	new RegExpException(	start. AddOut(new	start.AddOut(ReadEscap
PeekChar(1) > 0 &&			

()		/	
eChar(), ignoreCase,		Int32.Parse(str,	* Reads a
end);	value *= 8;	NumberStyles.AllowHexS	number from the
}		pecifier);	pattern. If the next
	value += ReadChar() -		character isn't a
/**	'0';	return (char) value;	* numeric
* Reads a	C	}	character, an
regular expression	= (char) PeekChar(0);	catch	exception is thrown.
character escape. This	if	(FormatException) {	This method reads
method	('0' <= c && c <= '7')	(TOT mathxccption) (	* several
	(0 <= 0 && 0 <= 1)	* h	
* handles a	1	throw new	consecutive numeric
single character		RegExpException(	characters.
escape in a regular	value *= 8;		*
expression.		RegExpException. ErrorT	* @return the
*	value += ReadChar() -	ype. UNSUPPORTED_ESCAPE	numeric value read
* @return the	'0';	_CHARACTER,	*
character read	}		* @throws
*	}	pos - str. Length - 2,	RegExpException if an
* @throws	return	r	error was encountered
RegExpException if an	(char) value;	pattern);	in the
error was encountered	case 'x':	pattern,	*
		J	
in the	str =	case 't':	pattern string
*	ReadChar().ToString()	return	*/
pattern string	+	'\t';	private int
*/	<pre>ReadChar().ToString();</pre>	case 'n':	ReadNumber() {
private char	try {	return	
ReadEscapeChar() {		'\n';	StringBuilder buf =
char c;	value =	case 'r':	new StringBuilder();
string	Int32. Parse(str,	return	int
str;	NumberStyles.AllowHexS	`\r`;	с;
int	pecifier);	case 'f':	ς,
value;	pecifici),	return	c =
value,		'\f';	
	return (char) value;		PeekChar(0);
(1) 1)	}	case 'a':	while ('0'
ReadChar('\\');	catch	return	<= c && c <= '9') {
c =	(FormatException) {	'\u0007';	
ReadChar();		case 'e':	buf.Append(ReadChar())
switch (c)	throw new	return	;
{	RegExpException(	'\u001B';	c =
case '0':		default:	PeekChar(0);
c =	RegExpException.ErrorT	if	}
ReadChar();	ype. UNSUPPORTED ESCAPE	(('A' <= c && c <=	if
if (c	CHARACTER,	'Z')    ('a' <= c && c	(buf. Length <= 0) {
<'0'    c > '3') {	_Clinical Lik,		
( 0    0 / 3 / 1	I 9	<= 'z')) {	throw
	pos - str.Length - 2,		new RegExpException(
throw new		throw new	
RegExpException(	pattern);	RegExpException(	RegExpException. ErrorT
	}		ype. UNEXPECTED_CHARACT
RegExpException.ErrorT	case 'u':	RegExpException. ErrorT	ER,
ype. UNSUPPORTED_ESCAPE	str =	ype. UNSUPPORTED_ESCAPE	
CHARACTER,	ReadChar().ToString()	_CHARACTER,	pos,
_ ,	+	_ ,	• ,
pos - 3,		pos - 2,	pattern);
pos 0,	<pre>ReadChar().ToString()</pre>	pos 2,	parterny,
pattern);	+	pattern);	no tump
pattern),	+	pattern),	return
}	D 101 () T 0: ()	} .	Int32. Parse (buf. ToStri
value	ReadChar().ToString()	return	ng());
= c - '0';	+	с;	}
С =		}	
(char) PeekChar(0);	<pre>ReadChar().ToString();</pre>	}	/**
if	try {		* Reads the
('0' <= c && c <= '7')	-	/**	next character in the
{	value =	•	
•			

pattern. If no next	*	*/	* with the
character	available in the		corresponding
* exists, an	pattern string	using System;	character.
exception is thrown.	*/	using System.Text;	*/
*	private char		private
* @return the	ReadChar(char c) {	namespace Core.Library	DFAState nonAscii =
character read	if (c !=	{	new DFAState();
*	ReadChar()) {		, ,
* @throws	throw	/**	/**
RegExpException if no	new RegExpException(	* A deterministic	* Creates a
next character was	new RegExpException(	finite state automaton	new empty string
available in	RegExpException.ErrorT	for matching exact	automaton.
*	ype. UNEXPECTED CHARACT		*/
·	_	strings.	,
the pattern string	ER,	* It uses a	public
*/		sorted binary tree	TokenStringDFA() {
private char	pos - 1,	representation of the	}
ReadChar() {		state	
int c =	pattern);	* transitions in	/**
PeekChar(0);	}	order to enable quick	* Adds a
	return c;	matches with a minimal	string match to this
if $(c < 0)$	}	memory	automaton. New states
{		* footprint. It	and
throw	/**	only supports a single	* transitions
new RegExpException(	* Returns a	character transition	will be added to
	character that has not	between	extend this automaton
RegExpException.ErrorT	vet been read from the	* states, but may	to
ype. UNTERMINATED PATTE	* pattern. If	be run in an all case-	* support the
RN,	the requested position	insensitive mode.	specified string.
101,	is beyond the end of	*	*
nos	the	4	* @param str
pos,		ale.	
	* pattern	*	the string to match
pattern);	string, -1 is		* @param
} else {	returned.	*/	caseInsensitive the
pos++;	*	internal class	case-insensitive flag
return	* @param	TokenStringDFA {	* @param
(char) c;	count the		value the
}	preview position, from	/**	match value
}	zero (0)	* The lookup	*/
	*	table for root states,	public void
/**	* @return the	indexed by the first	AddMatch(string str,
* Reads the	character found, or	ASCII	bool caseInsensitive,
next character in the	* -1	* character.	TokenPattern value) {
pattern. If the	if beyond the end of	This array is used to	DFAState
character	the pattern string	for speed optimizing	state;
* wasn't the	*/	the	DFAState
specified one, an	private int	* first step	next;
exception is thrown.	PeekChar(int count) {	in the match.	char
*	if (pos +	*/	c = str[0]:
* @param c	count <	private	int
the character to read	pattern. Length) {	DFAState[] ascii = new	start = 0;
*	return	DFAState[128];	Start o,
* @return the	pattern[pos + count];	Drastate[120],	if
	<pre>partern[pos   count], } else {</pre>	/**	(caseInsensitive) {
character read		,	
*	return	* The	c =
* @throws	-1;	automaton state	Char. ToLower(c);
RegExpException if the	}	transition tree for	}
character read didn't	}	non-ASCII characters.	if (c <
match the	}	* Each	128) {
*	}	transition from one	state
specified one, or if		state to another is	= ascii[c];
no next character was	/*	added to the tree	if
	* TokenStringDFA.cs		$(state == null) {$

	*	if	return
state = ascii[c] = new	null if no match was	(state == null) {	buffer. ToString();
DFAState();	found	, ,	}
}	*	break;	}
	* @throws	} else	
start++;	IOException if an I/O	if (state.value !=	
} else {	error occurred	null) {	/**
state	*/		* An automaton
= nonAscii;	public	result = state.value;	state. This class
}	TokenPattern	}	represents a state in
for (int i	Match(ReaderBuffer	pos++;	the DFA
= start; i <	buffer, bool	}	* graph.
str.Length; i++) {	caseInsensitive) {	return	*
next =		result;	
state.tree.Find(str[i]	TokenPattern result =	}	*
, caseInsensitive);	null;		*
if	DFAState	/**	*/
$(next == null) {$	state;	* Returns a	internal class
DD10	int	detailed string	DFAState {
<pre>next = new DFAState();</pre>	pos = 0;	representation of this	
	int	automaton.	/**
state.tree.Add(str[i],	с;	*	* The token
caseInsensitive,		* @return a	pattern matched at
next);	c =	detailed string	this state.
}	buffer. Peek(0);	representation of this	*/ internal
state	if (c < 0)	automaton */	Internal TokenPattern value =
= next;	roturn	*/ public	null;
J	return null;	override string	nuii,
state.value = value;	nuii,	ToString() {	/**
state. value - value,	if	105t11lig() (	* The
J	(caseInsensitive) {	StringBuilder buffer	automaton state
/**	c =	= new StringBuilder();	transition tree. Each
* Checks if	Char. ToLower ((char)	new Stringbarraer (),	transition from one
the automaton matches	c);	for (int i	* state to
an input stream. The	}	= 0; i < ascii.Length;	another is added to
* matching	if (c <	i++) {	the tree with the
will be performed from	128) {	if	corresponding
a specified position.	state	(ascii[i] != null) {	* character.
This	= ascii[c];		*/
* method will	if	buffer.Append((char)	internal
not read any	(state == null) {	i);	TransitionTree tree =
characters from the		if	<pre>new TransitionTree();</pre>
stream, just	return null;	(ascii[i].value !=	}
* peek ahead.	} else	null) {	
The comparison can be	if (state.value !=		
done either in	nu11) {	<pre>buffer.Append(": ");</pre>	/**
* case-			* An automaton
sensitive or case-	result = state.value;	buffer.Append(ascii[i]	state transition tree.
insensitive mode.	}	.value);	This class contains a
*	pos++;		* binary search
* @param	} else {	buffer. Append ("\n");	tree for the automaton
input the	state	}	transitions from one
input stream to check	= nonAscii;	::[:]	* state to
* @param pos	} h:1- //-	ascii[i].tree.PrintTo(	another. All
the starting position	while ((c	buffer, "");	transitions are linked
* @param	= buffer. Peek(pos)) >=	}	to a single
caseInsensitive the case-insensitive flag	0) {	J	* character. *
case-insensitive flag	state	nonAscii.tree.PrintTo(	*
* @return the	- state.tree.Find((char)	buffer, "");	*
match value, or	c, caseInsensitive);	Duffer, /,	*
macon varue, or	o, ousernsensitive/,		P.

ate /		4. 6	: £
*/ internal class	* with a lower-case conversion	* @param state the	if (buffer.Length > 0 &&
TransitionTree {	of the character.	state the state to transition to	buffer[buffer.Length -
Transferontiee (	*	*/	1] == '\n') {
/**	* @param c	public void	1] (11)
* The	the character to	Add(char c, bool	<pre>buffer. Append (indent);</pre>
transition character.	search for	lowerCase, DFAState	}
If this value is set	* @param	state) {	·
to the zero	lowerCase the	if	buffer. Append (this. val
* character	lower-case conversion	(lowerCase) {	ue);
$('\0')$ , this tree is	flag	c =	if
empty.	*	Char. ToLower(c);	(this. state. value !=
*/	* @return the	}	nu11) {
private char	automaton state found,	if (value	1 00 1 1/1 1/1
value = '\0';	or .	== '\0') {	<pre>buffer. Append(": ");</pre>
/**	*	4hi	1££ A1/+1:+-
/** * The	null if no transition exists	this.value = c;	<pre>buffer. Append(this. sta te. value);</pre>
transition target	exists */	this.state = state;	te. value),
state.	public	this. State - State,	<pre>buffer. Append("\n");</pre>
*/	DFAState Find(char c,	this.left = new	buffer: Append ( \land \text{if } \gamma,
private	bool lowerCase) {	TransitionTree();	,
DFAState state = null;	if	Transferonties (, ,	this. state. tree. PrintT
,	(lowerCase) {	this.right = new	o(buffer, indent + "
/**	c =	TransitionTree();	");
* The left	Char. ToLower(c);	} else if	}
subtree.	}	$(value > c) $ {	if
*/	if (value		(this.right != null) {
private	== '\0'    value == c)	left.Add(c, false,	
TransitionTree left =	{	state);	this.right.PrintTo(buf
null;	return	} else {	fer, indent);
	state;		}
/**	} else if	right. Add(c, false,	}
* The right	(value > c) {	state);	}
subtree. */	return left.Find(c, false);	) l	}
private	} else {	J	TokenLibray
TransitionTree right =	return	/**	TokenLibray
null;	right.Find(c, false);	* Prints the	namespace TokenLibrary
110111	}	automaton tree to the	{
/**	}	specified string	public class
* Creates a		buffer.	ErrorClass
new empty automaton	/**	*	{
transition tree.	* Adds a	* @param	int lines;
*/	transition to this	buffer the	int column;
public	tree. If the lower-	string buffer	string type;
TransitionTree() {	case flag is	* @param	string
}	* set, the	indent the	ErrorMessage;
/stude	character will be	current indentation	.11111
/**	converted to lower-	*/	public void
* Finds an automaton state from	case before * being	public void PrintTo(StringBuilder	setErrorMessage(string ErrorMessage)
the specified	added.	buffer, String indent)	Elioimessage)
transition	*	f string indent/	· ·
* character.	* @param c	if	this.ErrorMessage =
This method searches	the character to	(this.left != null) {	ErrorMessage;
this transition tree	transition for	(	}
for a	* @param	this.left.PrintTo(buff	public string
* matching	lowerCase the	er, indent);	<pre>getErrorMessage()</pre>
transition. The	lower-case conversion	}	{
comparison can	flag	if	return
optionally be done		(this. value != '\0') {	this.ErrorMessage;

```
return
  public void
setLines(int line)
                         this.tokens;
                          }
                            public void
       this.lines
= line;
                         setLexemes(string
  public int
getLines()
                         this.lexemes = lexeme;
                         }
     return
this.lines;
                             public string
                         getLexemes()
    public void
                         {
setColumn(int column)
                             return
  {
                         this.lexemes;
this.column = column;
                             public void
  }
                         setLines(int line)
    public int
                         {
getColumn()
                                  this.lines
                         = line;
      return
                             public int
this.column;
  }
                         getLines()
   public void
                               return
                         this.lines;
setType(string type)
  {
                          }
         this. type
                              public void
                         setAttributes(string
= type;
 }
                         attribute)
   public string
getType()
                         this.attributes =
                         attribute;
this.type;
                              public string
                         getAttributes()
                                  return
namespace TokenLibrary
                         this.attributes;
                         }
  public abstract
class TokensClass
     int lines;
     string tokens;
     string
  string
attributes;
     public void
setTokens(string
token)
this. tokens = token;
    }
    public string
getTokens()
```