```
> form.cs
using System;
using
System.Collections.Gen
eric;
using
System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using
System.Windows.Forms;
using System.IO;
using
Lexical_Analyzer;
using Syntax_Analyzer;
using
Semantics_Analyzer;
using
System.CodeDom.Compile
r;
using
System.Diagnostics;
using
Microsoft.CSharp;
using
System.Configuration;
using
System.Threading;
using
System.Runtime.Interop
Services;
using
System.Text.RegularExp
ressions;
using
IntellisenceTextBox;
using System.Linq;

namespace Militari
{
    public partial
class Form1 :
DevComponents.DotNetBa
r.Metro.MetroAppForm
    {
        string fname =
"";
        string
consolewrt = "";
        string
function = "";
        string
globdeclare = "";
        string main =
"";

        public Form1()
        {

InitializeComponent();
this.buttonX12.Click
+= new
System.EventHandler(th
is.buttonX12_Click);

//this.button2.Click
+= new
System.EventHandler(th
is.button1_Click);
            Code.KeyUp
+= (s, e) => {

List<string>
DictionaryList = new
List<string>(new
string[] { "Backup",
"commence", "deploy",
"go" }.ToList());

clsIntelliSense.AutoCo
mpleteTextBox(Code,
listBox1,
DictionaryList, e);
            };
        }

        int lines = 0;
        List<int>
linetokens = new
List<int>();

LexicalAnalyzer lex =
new LexicalAnalyzer();

        /*Literal
List*/
        List<string>
intlist = new
List<string>();
        List<string>
doublelist = new
List<string>();
        List<string>
stringlist = new
List<string>();
        List<string>
charlist = new
List<string>();
        List<string>
boolist = new
List<string>();
        List<string>
funclist = new
List<string>();

        private void
Form1_Load(object
sender, EventArgs e)
        {
        }

        private void
buttonX5_Click(object
sender, EventArgs e)
        {


OpenFileDialog
openFileDialog1 = new
OpenFileDialog();


openFileDialog1.Initia
lDirectory = @"C:\";

openFileDialog1.Title
= "Browse Militari
Solution";


openFileDialog1.CheckF
ileExists = true;

openFileDialog1.CheckP
athExists = true;


openFileDialog1.Defaul
tExt = "mltr";

openFileDialog1.Filter
= "Militari Solutions
(*.mltr)|*.mltr|All
files (*.*)|*.*";

openFileDialog1.Restor
eDirectory = true;


openFileDialog1.ReadOn
lyChecked = true;

openFileDialog1.ShowRe
adOnly = true;

            if
(openFileDialog1.ShowD
ialog() ==
DialogResult.OK)
            {
                fname
=
openFileDialog1.FileNa
me;

StreamReader sr = new
StreamReader(fname);

Code.Text =
sr.ReadToEnd();
sr.Close();
            }

        }

        private void
buttonX8_Click(object
sender, EventArgs e)
        {
            if (Code
!= null)
            {

DialogResult result =
MessageBox.Show("Clear
Code Workspace?",

"Clearing Option",

MessageBoxButtons.YesN
o);
                if
((result ==
DialogResult.Yes) &&
(Code != null))
                {

Code.Clear();

MessageBox.Show("Clear
ed!");
                }
                else
if(result ==
DialogResult.No) {

MessageBox.Show("No
Changes");
                }
                else {

MessageBox.Show("Code
Workspace is Empty!");
                }

            }
        }

        private void
buttonX9_Click(object
sender, EventArgs e)
        {

this.Close();
        }

        private void
buttonX6_Click(object
sender, EventArgs e)
        {
            if (fname
== "")
            {

openFileDialog1.Filter
= "Militari
Files|*.mltr";

DialogResult res =
openFileDialog1.ShowDi
alog();
                if
(res ==
DialogResult.Cancel)
                {

return;
                }
                fname
=
openFileDialog1.FileNa
me;

MessageBox.Show(fname)
;

StreamWriter sw = new
StreamWriter(fname);

sw.WriteLine(Code.Text
);

sw.Flush();

sw.Close();
            }
        }

        private void
buttonX7_Click(object
sender, EventArgs e)
        {

openFileDialog1.Filter
= "Text Files|*.txt";

openFileDialog1.ShowDi
alog();
            fname =
openFileDialog1.FileNa
me;
            if (fname
== "")
            {

openFileDialog1.Filter
= "Militari
Files|*.mltr";

DialogResult res =
openFileDialog1.ShowDi
alog();
```

```
                if
(res ==
DialogResult.Cancel)
                {

return;
                }
                fname
=
openFileDialog1.FileNa
me;

MessageBox.Show(fname)
;

StreamWriter sw = new
StreamWriter(fname);

sw.WriteLine(Code.Text
);

sw.Flush();

sw.Close();
            }
        }

        private void
buttonX11_Click(object
sender, EventArgs e)
        {

Code.Cut();
        }

        private void
buttonX2_Click(object
sender, EventArgs e)
        {

richTextBoxEx1.Text =
null;
            consolewrt
= "";

globdeclare = "";
            main = "";
            lex = new
LexicalAnalyzer();

buttonX3.Enabled =
false;
            if
(Code.Text != "")
            {

dataGridViewX1.Rows.Cl
ear();

dataGridViewX2.Rows.Cl
ear();
```

```
dataGridViewX3.Rows.Cl
ear();
                lex =
new LexicalAnalyzer();

Initializer Lexical =
new Initializer();
                string
txt = Code.Text;
                lex =
Lexical.InitializeAnal
yzer(txt, lex);


DisplayTokens(lex);
            }
            if
(lex.invalid == 0 &&
lex.token.Count != 0)
            {

buttonX3.Enabled =
true;
            }
            else {

buttonX3.Enabled =
false;

buttonX4.Enabled =
false;

dataGridViewX1.Show();
            }

        }

        private
SemanticsInitializer
SemanticsStart(List<Se
manticsInitializer.Tok
ens> tokens)
        {

SemanticsInitializer
sem = null;
            try
            {
                sem =
new
SemanticsInitializer(t
okens);
            }
            catch
(Exception e)
            {

MessageBox.Show(e.Mess
age);
            }
```

```
            return
sem;
        }

        public
List<SemanticsInitiali
zer.Tokens>
tokenDumps(List<Tokens
> tokens)
        {

List<SemanticsInitiali
zer.Tokens> token =
new
List<SemanticsInitiali
zer.Tokens>();

SemanticsInitializer.T
okens t = new
SemanticsInitializer.T
okens();
            foreach
(var item in tokens)
            {
                t =
new
SemanticsInitializer.T
okens();

t.setAttributes(item.g
etAttributes());

t.setLexemes(item.getL
exemes());

t.setLines(item.getLin
es());

t.setTokens(item.getTo
kens());

token.Add(t);
            }
            return
token;
        }

        private void
DisplayTokens(LexicalA
nalyzer lex)
        {
            string
result = "Successfully
Executed.";
            int ctr =
0, id = 1;

LexGrid.Rows.Clear();

dataGridViewX4.Rows.Cl
ear();
```

```
            if
(lex.invalid != 0)
              result =
"Encountered " +
lex.invalid.ToString()
+ " error/s.\nPlease
try again.\n";

dataGridViewX1.Rows.Ad
d(id , "Lexical
Analyzer " + result);

            foreach
(var token in
lex.token)
            {
                if
(token.getTokens() ==
"INVALID")
                {

dataGridViewX1.Rows.Ad
d(id, "Invalid input:
"

+ token.getLexemes()

, " on line "

+ token.getLines() +
"\n");
                }
                else
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(),
token.getAttributes())
;
                }
                ctr++;
            }
        }

        private int
GetErrorLine(int ctr)
        {
            int line =
0;
            int cls =
0;
            for (int i
= 0; i <
linetokens.Count; i++)
            {
                cls =
linetokens[i];
                if
(ctr + 1 <=
linetokens[i])

return (i + 1);
            }
            return
line;
        }

        private void
buttonX3_Click(object
sender, EventArgs e)
        {

SyntaxInitializer
S_initialize = new
SyntaxInitializer();

dataGridViewX2.Rows.Cl
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);
                }

dataGridViewX2.Rows.Ad
d(errornum,
S_initialize.errors.ge
tErrorMessage(),
S_initialize.errors.ge
tLines());

errornum++;
            }
            else
            {

dataGridViewX2.Rows.Ad
d(i, s);

buttonX4.Enabled =
true;

dataGridViewX2.Show();
            }
        }

        public
List<TokenLibrary.Toke
nsClass>
tokenDump(List<Lexical
_Analyzer.Tokens>
tokens)
        {

List<TokenLibrary.Toke
nsClass> token = new
List<TokenLibrary.Toke
nsClass>();
            Tokens t =
new Tokens();
            foreach
(var item in tokens)
            {
                t =
new Tokens();

t.setAttributes(item.g
etAttributes());

t.setLexemes(item.getL
exemes());

t.setLines(item.getLin
es());

t.setTokens(item.getTo
kens());

token.Add(t);
            }
            return
token;
        }

        private void
buttonX4_Click(object
sender, EventArgs e)
        {

List<string>
ConstantvarList = new
List<string>();

List<string>
GlobalvarList = new
List<string>();

List<string>
LocalvarList = new
List<string>();

List<string> ReservedW
= new List<string> {
"company", "unit",
"digit", "response",
"joe", "hold", "miss",
"operation", "struct",

"PrimaryMission",
"post", "capture",
"backup", "campaign",
"abort", "deploy",
"inquire",

"inorder",
"otherorder","phase",
"go", "order","action"
};

List<string> Operators
= new List<string> {
"+", "-", "*", "/",
"=" };

List<string> disp =
new List<string>();

            bool
constexists;
            bool
globexists;
            bool
locexists;
            int idn =
0;
            int Line =
1;
            int x = 0;


ConstantvarList.Clear(
);

GlobalvarList.Clear();

LocalvarList.Clear();

dataGridViewX3.Rows.Cl
ear();
```

```
            for (x =
0; x <
LexGrid.Rows.Count;
x++)
            {

/*Constant Semantics*/
                if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "hold")
                {

x++;
                    if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")
                    {

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (constexists =
ConstantvarList.Exists
(element => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

ConstantvarList.Add(Le
xGrid.Rows[x].Cells[1]
.Value.ToString());

intlist.Add(LexGrid.Ro
ws[x].Cells[1].Value.T
oString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Numlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
";")

{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";"))

{

Line++;

}

} while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";"));
                    }
                    if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")
                    {

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (constexists =
ConstantvarList.Exists
(element => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

ConstantvarList.Add(Le
xGrid.Rows[x].Cells[1]
.Value.ToString());

doublelist.Add(LexGrid
.Rows[x].Cells[1].Valu
e.ToString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Declit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
";")

{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";"))

{

Line++;

}

} while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";"));
                    }
                    if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company")
                    {

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (constexists =
ConstantvarList.Exists
(element => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{
```

```
dataGridViewX3.Rows.Add(idn++, "Multiple declaration of variable: " + LexGrid.Rows[x].Cells[1].Value.ToString(), " Line: " + Line);

}

else {

ConstantvarList.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

stringlist.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

}

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "=" || LexGrid.Rows[x].Cells[2].Value.ToString() == "Stringlit" || LexGrid.Rows[x].Cells[2].Value.ToString() == ";")

{

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "company")

{

}

else {

dataGridViewX3.Rows.Add(idn++, "TypeMismatch: " + LexGrid.Rows[x].Cells[2].Value.ToString(), " Line: " + Line);

}

x++;

if ((LexGrid.Rows[x].Cells[2].Value.ToString() == ";"))

{

Line++;

}

} while ((LexGrid.Rows[x].Cells[2].Value.ToString() != ";"));
                    }
                    if (LexGrid.Rows[x].Cells[2].Value.ToString() == "joe")
                    {

do

{

if (LexGrid.Rows[x].Cells[2].Value.ToString() == "id")

{

if (constexists = ConstantvarList.Exists(element => element == LexGrid.Rows[x].Cells[1].Value.ToString()) == true)

{

dataGridViewX3.Rows.Add(idn++, "Multiple declaration of variable: " + LexGrid.Rows[x].Cells[1].Value.ToString(), " Line: " + Line);

}

else {

ConstantvarList.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

charlist.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

}

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "=" || LexGrid.Rows[x].Cells[2].Value.ToString() == "Charlit" || LexGrid.Rows[x].Cells[2].Value.ToString() == ";")

{

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "joe")

{

}

else {

dataGridViewX3.Rows.Add(idn++, "TypeMismatch: " + LexGrid.Rows[x].Cells[2].Value.ToString(), " Line: " + Line);

}

x++;

if ((LexGrid.Rows[x].Cells[2].Value.ToString() == ";"))

{

Line++;

}

} while ((LexGrid.Rows[x].Cells[2].Value.ToString() != ";"));
                    }
                }


/*Global Semantics*/
                if (LexGrid.Rows[x].Cells[2].Value.ToString() == "unit")
                {
                    do
                    {

if (LexGrid.Rows[x].Cells[2].Value.ToString() == "id")

{

if (globexists = GlobalvarList.Exists(element => element == LexGrid.Rows[x].Cells[1].Value.ToString()) == true)

{

dataGridViewX3.Rows.Add(idn++, "Multiple declaration of variable: " + LexGrid.Rows[x].Cells[1].Value.ToString(), " Line: " + Line);

}

else {

GlobalvarList.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

intlist.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

}

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "=" || LexGrid.Rows[x].Cells[2].Value.ToString() == "Numlit" || LexGrid.Rows[x].Cells[2].Value.ToString() == ";")

{

}
```

```
else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"," ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"(" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
")" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"{")

{

//x++;

//do

//{

//    if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

//    {

//
funclist.Add(LexGrid.R
ows[x].Cells[1].Value.
ToString());

//        x++;

//    }

//} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "(");

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "[" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"]")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";") ||
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "{"))

{

Line++;

}
                      }
while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";") &&
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{"));
                  }

if(LexGrid.Rows[x].Cel
ls[2].Value.ToString()
== "miss")
                  {

x++;
                      do
                      {

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

funclist.Add(LexGrid.R
ows[x].Cells[1].Value.
ToString());

x++;

}
                      }
while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "(");
                  }

                  if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")
                  {
                      do
                      {

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (globexists =
GlobalvarList.Exists(e
lement => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

GlobalvarList.Add(LexG
rid.Rows[x].Cells[1].V
alue.ToString());

doublelist.Add(LexGrid
.Rows[x].Cells[1].Valu
e.ToString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Declit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
";")

{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"," ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"(" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
")" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"{")

{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "[" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"]")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";") ||
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "{"))

{

Line++;
```

```
}
                    }
while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";") &&
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{"));
                 }
                 if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company")
                 {
                     do
                     {

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (globexists =
GlobalvarList.Exists(e
lement => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

GlobalvarList.Add(LexG
rid.Rows[x].Cells[1].V
alue.ToString());

stringlist.Add(LexGrid
.Rows[x].Cells[1].Valu
e.ToString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Stringlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
";")

{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"," ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"(" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
")" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"{")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";") ||
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "{"))

{

Line++;

}
                      }
while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";") &&
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{"));
                 }
                 if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "joe")
                 {
                     do
                     {

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (globexists =
GlobalvarList.Exists(e
lement => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

GlobalvarList.Add(LexG
rid.Rows[x].Cells[1].V
alue.ToString());

charlist.Add(LexGrid.R
ows[x].Cells[1].Value.
ToString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Charlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
";")

{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "joe" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"," ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"(" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
")" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"{")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";") ||
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "{"))

{

Line++;

}
                      }
while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";") &&
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{"));
```

```
                }
                if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "response")
                {
                    do
                    {

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (globexists =
GlobalvarList.Exists(e
lement => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

GlobalvarList.Add(LexG
rid.Rows[x].Cells[1].V
alue.ToString());

boolist.Add(LexGrid.Ro
ws[x].Cells[1].Value.T
oString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"AFFIRMATIVE" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"NEGATIVE" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
";")
```

```
{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "response" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"," ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"(" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
")" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"{")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";") ||
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "{"))

{

Line++;

}
                        }
while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";") &&
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{"));
                    }
```

```
                if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")
                {

bool exist;

if(exist =
GlobalvarList.Exists(e
lement => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== false) {

dataGridViewX3.Rows.Ad
d(idn++, "Accessing
undeclared Variable: "
+
LexGrid.Rows[x].Cells[
1].Value.ToString());
                    }

else if (exist =
funclist.Exists(elemen
t => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== false)
                    {

dataGridViewX3.Rows.Ad
d(idn++, "Accessing
undeclared Variable: "
+
LexGrid.Rows[x].Cells[
1].Value.ToString());
                    }
                }
                else
                {

                }

/*Local Declaration*/
                if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "PrimaryMission")
                    {

do

{

x++;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "(") { }
```

```
if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== ")") { }

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "{") { }

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")

{

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (locexists =
LocalvarList.Exists(el
ement => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

LocalvarList.Add(LexGr
id.Rows[x].Cells[1].Va
lue.ToString());

intlist.Add(LexGrid.Ro
ws[x].Cells[1].Value.T
oString());

}

}
```

```
else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "=" || LexGrid.Rows[x].Cells[2].Value.ToString() == "Numlit" || LexGrid.Rows[x].Cells[2].Value.ToString() == ";")

{

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "unit" || LexGrid.Rows[x].Cells[2].Value.ToString() == "," || LexGrid.Rows[x].Cells[2].Value.ToString() == "(" || LexGrid.Rows[x].Cells[2].Value.ToString() == ")" || LexGrid.Rows[x].Cells[2].Value.ToString() == "{")

{

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "[" || LexGrid.Rows[x].Cells[2].Value.ToString() == "]")

{

}

else {

dataGridViewX3.Rows.Add(idn++, "TypeMismatch: " + LexGrid.Rows[x].Cells[2].Value.ToString(), " Line: " + Line);

}

x++;

if ((LexGrid.Rows[x].Cells[2].Value.ToString() == ";") || (LexGrid.Rows[x].Cells[2].Value.ToString() == "{"))

{

Line++;

}

} while ((LexGrid.Rows[x].Cells[2].Value.ToString() != ";") && (LexGrid.Rows[x].Cells[2].Value.ToString() != "{"));

}

if (LexGrid.Rows[x].Cells[2].Value.ToString() == "digit")

{

do

{

if (LexGrid.Rows[x].Cells[2].Value.ToString() == "id")

{

if (locexists = LocalvarList.Exists(element => element == LexGrid.Rows[x].Cells[1].Value.ToString()) == true)

{

dataGridViewX3.Rows.Add(idn++, "Multiple declaration of variable: " + LexGrid.Rows[x].Cells[1].Value.ToString(), " Line: " + Line);

}

else {

LocalvarList.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

doublelist.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

}

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "=" || LexGrid.Rows[x].Cells[2].Value.ToString() == "Declit" || LexGrid.Rows[x].Cells[2].Value.ToString() == ";")

{

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == "digit" || LexGrid.Rows[x].Cells[2].Value.ToString() == "," || LexGrid.Rows[x].Cells[2].Value.ToString() == "(" || LexGrid.Rows[x].Cells[2].Value.ToString() == ")" || LexGrid.Rows[x].Cells[2].Value.ToString() == "{")

{

}

else {

dataGridViewX3.Rows.Add(idn++, "TypeMismatch: " + LexGrid.Rows[x].Cells[2].Value.ToString(), " Line: " + Line);

}

x++;

if ((LexGrid.Rows[x].Cells[2].Value.ToString() == ";") || (LexGrid.Rows[x].Cells[2].Value.ToString() == "{"))

{

Line++;

}

} while ((LexGrid.Rows[x].Cells[2].Value.ToString() != ";") && (LexGrid.Rows[x].Cells[2].Value.ToString() != "{"));

}

if (LexGrid.Rows[x].Cells[2].Value.ToString() == "company")

{

do

{

if (LexGrid.Rows[x].Cells[2].Value.ToString() == "id")

{

if (locexists = LocalvarList.Exists(element => element == LexGrid.Rows[x].Cells[1].Value.ToString()) == true)

{

dataGridViewX3.Rows.Add(idn++, "Multiple declaration of variable: " + LexGrid.Rows[x].Cells[1].Value.ToString(), " Line: " + Line);
```

```
}

else {

LocalvarList.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

stringlist.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

}

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″=″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″Stringlit″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″;″)

{

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″company″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″,″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″(″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″)″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″{″)

{

}

else {

dataGridViewX3.Rows.Add(idn++, ″TypeMismatch: ″ + LexGrid.Rows[x].Cells[2].Value.ToString(), ″ Line: ″ + Line);

}

x++;

if ((LexGrid.Rows[x].Cells[2].Value.ToString() == ″;″) || (LexGrid.Rows[x].Cells[2].Value.ToString() == ″{″))

{

Line++;

}

} while ((LexGrid.Rows[x].Cells[2].Value.ToString() != ″;″) && (LexGrid.Rows[x].Cells[2].Value.ToString() != ″{″));

}

if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″joe″)

{

do

{

if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″id″)

{

if (locexists = LocalvarList.Exists(element => element == LexGrid.Rows[x].Cells[1].Value.ToString()) == true)

{

dataGridViewX3.Rows.Add(idn++, ″Multiple declaration of variable: ″ + LexGrid.Rows[x].Cells[1].Value.ToString(), ″ Line: ″ + Line);

}

else {

LocalvarList.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

charlist.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

}

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″=″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″Charlit″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″;″)

{

}

else if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″joe″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″,″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″(″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″)″ || LexGrid.Rows[x].Cells[2].Value.ToString() == ″{″)

{

}

else {

dataGridViewX3.Rows.Add(idn++, ″TypeMismatch: ″ + LexGrid.Rows[x].Cells[2].Value.ToString(), ″ Line: ″ + Line);

}

x++;

if ((LexGrid.Rows[x].Cells[2].Value.ToString() == ″;″) || (LexGrid.Rows[x].Cells[2].Value.ToString() == ″{″))

{

Line++;

}

} while ((LexGrid.Rows[x].Cells[2].Value.ToString() != ″;″) && (LexGrid.Rows[x].Cells[2].Value.ToString() != ″{″));

}

if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″response″)

{

do

{

if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″id″)

{

if (locexists = LocalvarList.Exists(element => element == LexGrid.Rows[x].Cells[1].Value.ToString()) == true)

{

dataGridViewX3.Rows.Add(idn++, ″Multiple declaration of variable: ″ +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

LocalvarList.Add(LexGr
id.Rows[x].Cells[1].Va
lue.ToString());

boolist.Add(LexGrid.Ro
ws[x].Cells[1].Value.T
oString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"AFFIRMATIVE" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"NEGATIVE" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
";")

{

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "response" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"," ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"(" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
")" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"{")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== ";") ||
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "{"))

{

Line++;

}

} while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= ";") &&
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{"));

}

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "miss")

{

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

if (locexists =
LocalvarList.Exists(el
ement => element ==
LexGrid.Rows[x].Cells[
1].Value.ToString())
== true)

{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of
variable: " +
LexGrid.Rows[x].Cells[
1].Value.ToString(), "
Line: " + Line);

}

else {

LocalvarList.Add(LexGr
id.Rows[x].Cells[1].Va
lue.ToString());

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "Stringlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Declit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Numlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Charlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"=" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"joe" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"," ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"(" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
")" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"{")

{

}

else {

dataGridViewX3.Rows.Ad
d(idn++,
"TypeMismatch: " +
LexGrid.Rows[x].Cells[
2].Value.ToString(), "
Line: " + Line);

}

x++;

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== "{"))

{

Line++;

}

} while
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
!= "{"));

}

else {

dataGridViewX3.Rows.Ad
d(idn++, "Semantics
Analyzer
Succeeded...");

}


x++;


} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "deploy");

//
MessageBox.Show(x.ToSt
ring());
                   }

else
                   {

continue;
                   }
               }
           /*Check if
Constant var Exist to
Global var*/
           foreach
(string constlist in
ConstantvarList)
               {
                if
(GlobalvarList.Contain
s(constlist) == true)
                {

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of Global
variable: " +
constlist, " Line: " +
Line);
                }

if(LocalvarList.Contai
ns(constlist) == true)
{

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of Local
variable: " +
constlist, " Line: " +
Line);
                }
                if
(ReservedW.Contains(co
nstlist) == true)
                {

dataGridViewX3.Rows.Ad
d(idn++, "Reserved
Identifier Misused: "
+ constlist, " Line: "
+ Line);
                }
                else
                {

continue;
                }
            }
            foreach
(string globallist in
GlobalvarList)
            {

                if
(LocalvarList.Contains
(globallist) == true)
                {

dataGridViewX3.Rows.Ad
d(idn++, "Multiple
declaration of Local
variable: " +
globallist, " Line: "
+ Line);
                }
                if
(ReservedW.Contains(gl
oballist) == true)
                {

dataGridViewX3.Rows.Ad
d(idn++, "Reserved
Identifier Misused: "
+ globallist, " Line:
" + Line);
                }
                else
                {

continue;
                }
            }
        }

        private void
buttonX12_Click(object
sender, EventArgs e)
        {

richTextBoxEx1.Text =
"";
            int
checktemp = 0;
            int
checkfunc = 0;

List<string> disp =
new List<string>();

List<string> outp =
new List<string>();

richTextBoxEx1.Text =
"using System; \n
namespace test \n { \n
class test \n { \n ";

//globdeclare =
"public static class
GlobalVar { \n";
            //main =
"class funct \n { \n";
            //function
= "class func \n {
\n";
            for (int x
= 0; x <
LexGrid.Rows.Count;
x++)
            {
                if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "}")
                {

x++;
                    if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "deploy")
                    {

x += 2;

main +=
"Console.ReadLine();
\n } \n";
                    }

else
                    {

main += LexGrid.Rows[x
-
1].Cells[1].Value.ToSt
ring() + " \n";

// x--;
                    }
                }
                switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())
                {

case "PrimaryMission":

x += 3;

main += "public static
void Main() \n { \n";

checktemp = 1;

break;

case "unit":

//Check if Global unit

if (checktemp == 0 ||
checkfunc == 0)

{

if (LexGrid.Rows[x +
2].Cells[2].Value.ToSt
ring() == "(")

{

checkfunc = 1;

x++;

main += "public static
int ";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")

{

main += "double ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")

{

main += "int ";

x++;

if
((LexGrid.Rows[x+2].Ce
lls[1].Value.ToString(
) == "[") &&
(LexGrid.Rows[x +
5].Cells[1].Value.ToSt
ring() == "["))

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"," + "]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x++;

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "joe")

{

main += "char ";

x++;

}
```

```
else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company")

{

main += "string ";

x++;

}

else

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= "{");

main += "{\n";

break;

}

if(checkfunc == 0 &&
checktemp == 0)

{

x++;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

x++;

if
((LexGrid.Rows[x].Cell
s[1].Value.ToString()
== "[") &&
(LexGrid.Rows[x +
3].Cells[1].Value.ToSt
ring() == "["))

{

main += "public static
int ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"," + "]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x += 6;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ";");

main += "; \n";

// x += 2;

}

else

{

main += "= new int[" +
LexGrid.Rows[x -
5].Cells[1].Value.ToSt
ring() + "," +
LexGrid.Rows[x -
2].Cells[1].Value.ToSt
ring();

main += "]; \n";

}

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "[")

{

main += "public static
int ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x += 3;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= "}");

main += "}; \n";

x++;

}

else

{

main += "= new int[" +
LexGrid.Rows[x -
2].Cells[1].Value.ToSt
ring() + "]; \n";

}

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "(")

{

main += "public static
int ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{");

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

main += "public static
int " + LexGrid.Rows[x
-
1].Cells[1].Value.ToSt
ring() + "=" +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "; \n";

x = x + 2;

}

else

{
```

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

```
}
break;
case ",":
x++;
break;
}
} while (LexGrid.Rows[x].Cells[2].Value.ToString() != ";");
}
break;

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

```
main += LexGrid.Rows[x].Cells[1].Value.ToString();

x++;

} while (LexGrid.Rows[x].Cells[1].Value.ToString() != "}");

main += "}; \n";

x++;

}

else

{

main += "= new double[" + LexGrid.Rows[x - 2].Cells[1].Value.ToString() + "]; \n";

}

}

else if (LexGrid.Rows[x].Cells[1].Value.ToString() == "(")

{

main += "public static double ";

main += LexGrid.Rows[x].Cells[1].Value.ToString();

do

{

main += LexGrid.Rows[x].Cells[1].Value.ToString();

x++;

} while (LexGrid.Rows[x].Cells[2].Value.ToString() != "{");

main += LexGrid.Rows[x].Cells[1].Value.ToString();

}

else if (LexGrid.Rows[x].Cells[1].Value.ToString() == "=")

{

main += "public static double " + LexGrid.Rows[x - 1].Cells[1].Value.ToString() + "=" + LexGrid.Rows[x + 1].Cells[1].Value.ToString() + "; \n";

x = x + 2;

}

else

{

main += "public static double " + LexGrid.Rows[x - 1].Cells[1].Value.ToString() + "; \n";

}

break;

case ",":

x++;

break;

}

} while (LexGrid.Rows[x].Cells[2].Value.ToString() != ";");

}

else

{

main += "double ";

}

break;

}

else

{

//Check if Local unit

x++;

do

{

switch (LexGrid.Rows[x].Cells[2].Value.ToString())

{

case "id":

x++;

if ((LexGrid.Rows[x].Cells[1].Value.ToString() == "[") && (LexGrid.Rows[x + 3].Cells[1].Value.ToString() == "["))

{

main += "double ";

main += LexGrid.Rows[x].Cells[1].Value.ToString() + "," + "]" + " " + LexGrid.Rows[x - 1].Cells[1].Value.ToString();

x += 6;

if (LexGrid.Rows[x].Cells[1].Value.ToString() == "=")

{

do

{

main += LexGrid.Rows[x].Cells[1].Value.ToString();

x++;

} while (LexGrid.Rows[x].Cells[1].Value.ToString() != ";");

main += "; \n";

// x += 2;

}

else

{

main += "= new double[" + LexGrid.Rows[x - 5].Cells[1].Value.ToString() + "," + LexGrid.Rows[x - 2].Cells[1].Value.ToString();

main += "]; \n";

}

}

else if (LexGrid.Rows[x].Cells[1].Value.ToString() == "[")

{

main += "int ";

main += LexGrid.Rows[x].Cells[1].Value.ToString() + "]" + " " + LexGrid.Rows[x - 1].Cells[1].Value.ToString();

x += 3;

if (LexGrid.Rows[x].Cells[1].Value.ToString() == "=")

{

do
```

```
{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= "}");

main += "}; \n";

x++;

}

else

{

main += "= new
double[" +
LexGrid.Rows[x -
2].Cells[1].Value.ToSt
ring() + "]; \n";

}

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "(")

{

main += "double ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{");

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

main += "double " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "=" +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "; \n";

x = x + 2;

}

else

{

main += "double " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "; \n";

}

break;

case ",":

x++;

break;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");

}

break;

//Check if Global
Company

case "company":

//Check if Global unit

if (checktemp == 0 ||
checkfunc == 0)

{

if (LexGrid.Rows[x +
2].Cells[2].Value.ToSt
ring() == "(")

{

checkfunc = 1;

x++;

main += "public static
string ";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")

{

main += "double ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")

{

main += "int ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "joe")

{

main += "char ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company")

{

main += "string ";

x++;

}

else

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= "{");

main += "{\n";

break;

}

if (checkfunc == 0 &&
checktemp == 0)

{

x++;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

x++;

if
((LexGrid.Rows[x].Cell
s[1].Value.ToString()
== "[") &&
(LexGrid.Rows[x + 3].Cells[1].Value.ToString() == ″[″))

{

main += ″public static string ″;

main += LexGrid.Rows[x].Cells[1].Value.ToString() + ″,″ + ″]″ + ″ ″ + LexGrid.Rows[x - 1].Cells[1].Value.ToString();

x += 6;

if (LexGrid.Rows[x].Cells[1].Value.ToString() == ″=″)

{

do

{

main += LexGrid.Rows[x].Cells[1].Value.ToString();

x++;

} while (LexGrid.Rows[x].Cells[1].Value.ToString() != ″;″);

main += ″; \n″;

// x += 2;

}

else

{

main += ″= new string[″ + LexGrid.Rows[x - 5].Cells[1].Value.ToString() + ″,″ + LexGrid.Rows[x - 2].Cells[1].Value.ToString();

main += ″]; \n″;

}

}

else if (LexGrid.Rows[x].Cells[1].Value.ToString() == ″[″)

{

main += ″public static string ″;

main += LexGrid.Rows[x].Cells[1].Value.ToString() + ″]″ + ″ ″ + LexGrid.Rows[x - 1].Cells[1].Value.ToString();

x += 3;

if (LexGrid.Rows[x].Cells[1].Value.ToString() == ″=″)

{

do

{

main += LexGrid.Rows[x].Cells[1].Value.ToString();

x++;

} while (LexGrid.Rows[x].Cells[1].Value.ToString() != ″}″);

main += ″}; \n″;

x++;

}

else

{

main += ″= new string[″ + LexGrid.Rows[x - 2].Cells[1].Value.ToString() + ″]; \n″;

}

}

else if (LexGrid.Rows[x].Cells[1].Value.ToString() == ″(″)

{

main += ″public static string ″;

main += LexGrid.Rows[x].Cells[1].Value.ToString();

do

{

main += LexGrid.Rows[x].Cells[1].Value.ToString();

x++;

} while (LexGrid.Rows[x].Cells[2].Value.ToString() != ″{″);

main += LexGrid.Rows[x].Cells[1].Value.ToString();

}

else if (LexGrid.Rows[x].Cells[1].Value.ToString() == ″=″)

{

main += ″public static string ″ + LexGrid.Rows[x - 1].Cells[1].Value.ToString() + ″=″ + LexGrid.Rows[x + 1].Cells[1].Value.ToString() + ″; \n″;

x = x + 2;

}

else

{

main += ″public static string ″ + LexGrid.Rows[x - 1].Cells[1].Value.ToString() + ″; \n″;

}

break;

case ″,″:

x++;

break;

}

} while (LexGrid.Rows[x].Cells[2].Value.ToString() != ″;″);

}

else

{

main += ″string ″;

}

break;

}

else

{

//Check if Local unit

x++;

do

{

switch (LexGrid.Rows[x].Cells[2].Value.ToString())

{

case ″id″:

x++;
```

```
if
((LexGrid.Rows[x].Cell
s[1].Value.ToString()
== "[") &&
(LexGrid.Rows[x +
3].Cells[1].Value.ToSt
ring() == "["))

{

main += "string ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"," + "]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x += 6;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ";");

main += "; \n";

// x += 2;

}

else

{

main += "= new
string[" +
LexGrid.Rows[x -
5].Cells[1].Value.ToSt
ring() + "," +
LexGrid.Rows[x -
2].Cells[1].Value.ToSt
ring();

main += "]; \n";

}

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "[")

{

main += "int ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x += 3;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= "}");

main += "}; \n";

x++;

}

else

{

main += "= new
string[" +
LexGrid.Rows[x -
2].Cells[1].Value.ToSt
ring() + "]; \n";

}

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "(")

{

main += "string ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{");

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

main += "string " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "=" +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "; \n";

x = x + 2;

}

else

{

main += "string " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "; \n";

}

break;

case ",":

x++;

break;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");

}

break;

case "joe":

//Check if global joe


if (checktemp == 0 ||
checkfunc == 0)

{

if (LexGrid.Rows[x +
2].Cells[2].Value.ToSt
ring() == "(")

{

checkfunc = 1;

x++;

main += "public static
char ";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")

{
```

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

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

main += "public static
char " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "=" +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "; \n";

x = x + 2;

}

else

{

main += "public static
char " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "; \n";

}

break;

case ",":

x++;

break;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");

}

else

{
```

```
main += "char ";

}

break;

}

else

{

//Check if Local unit

x++;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

x++;

if
((LexGrid.Rows[x].Cell
s[1].Value.ToString()
== "[") &&
(LexGrid.Rows[x +
3].Cells[1].Value.ToSt
ring() == "["))

{

main += "char ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"," + "]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x += 6;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

do
```

```
{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ";");

main += "; \n";

// x += 2;

}

else

{

main += "= new char["
+ LexGrid.Rows[x -
5].Cells[1].Value.ToSt
ring() + "," +
LexGrid.Rows[x -
2].Cells[1].Value.ToSt
ring();

main += "]; \n";

}

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "[")

{

main += "char ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x += 3;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")
```

```
{

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= "}");

main += "}; \n";

x++;

}

else

{

main += "= new char["
+ LexGrid.Rows[x -
2].Cells[1].Value.ToSt
ring() + "]; \n";

}

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "(")

{

main += "char ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{");

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

}

else if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "=")

{

main += "char " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "=" +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "; \n";

x = x + 2;

}

else

{

main += "char " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring() + "; \n";

}

break;

case ",":

x++;

break;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");

}

break;


case "response":

//Check if Global
response

if (checktemp == 0)

{

x++;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

main += "boolean ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"=false; \n";

}

break;

}

//Check if local
response

else

{

x++;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

main += "boolean ";

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"=false; \n";

}

break;

}

case "miss":

checkfunc = 1;

x++;

main += "public static
void ";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")

{

main += "double ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")

{

main += "int ";

x++;


if
((LexGrid.Rows[x].Cell
s[1].Value.ToString()
== "[") &&
(LexGrid.Rows[x +
3].Cells[1].Value.ToSt
ring() == "["))

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"," + "]" + " " +
LexGrid.Rows[x -
1].Cells[1].Value.ToSt
ring();

x++;

}

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "joe")

{

main += "char ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company")

{

main += "string ";

x++;

}

else

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{");

main += "\n" +
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"\n";

break;

case "struct":

int temp=0;

main += "public struct
";

x++;

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"{ \n";
```

```
x += 2;

do

{

if
((LexGrid.Rows[x].Cell
s[2].Value.ToString()
== "}") &&
(LexGrid.Rows[x +
1].Cells[2].Value.ToSt
ring() == ";"))

{

temp = 1;

main += "};\n";

x++;


}

else

{

temp = 0;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")

{

temp = 0;

main += "public int ";

x++;


}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")

{

temp = 0;

main += "public double
";

x++;
```

```
}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company")

{

temp = 0;

main += "public string
";

x++;


}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "joe")

{

temp = 0;

main += "public char
";

x++;


}

else {

temp = 0;

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}


}

} while (temp == 0);

break;

case "hold":

main += "const ";

x++;
```

```
do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "digit")

{

main += "double ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "unit")

{

main += "int ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "joe")

{

main += "char ";

x++;

}

else if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "company")

{

main += "string ";

x++;

}

else

{

main +=
```

```
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}


} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");

main += ";\n";

break;

case "capture":

x++;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "(")

{

x++;

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "#")

{

x++;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

foreach (string a in
intlist)

{

if (a ==
LexGrid.Rows[x].Cells[
1].Value.ToString())

{

x++;
```

```
if (LexGrid.Rows[x].Cells[1].Value.ToString() == "[")

{

if (LexGrid.Rows[x + 3].Cells[1].Value.ToString() == "[")

{

x--;

do

{

switch (LexGrid.Rows[x].Cells[2].Value.ToString())

{

case "id":

main += LexGrid.Rows[x].Cells[1].Value.ToString();

break;

default:

main += LexGrid.Rows[x].Cells[1].Value.ToString() + LexGrid.Rows[x+1].Cells[1].Value.ToString() + "," + LexGrid.Rows[x+4].Cells[1].Value.ToString() + LexGrid.Rows[x+5].Cells[1].Value.ToString();

x = x + 5;

break;

}

x++;

} while (LexGrid.Rows[x].Cells[1].Value.ToString() != ")");

//main += "GlobalVar." + LexGrid.Rows[x - 1].Cells[1].Value.ToString() + LexGrid.Rows[x].Cells[1].Value.ToString() + LexGrid.Rows[x + 1].Cells[1].Value.ToString() + LexGrid.Rows[x + 2].Cells[1].Value.ToString();

main += " = Convert.ToInt32(Console.ReadLine()); \n";

}

else

{

x--;

do

{

switch (LexGrid.Rows[x].Cells[2].Value.ToString())

{

case "id":

main += LexGrid.Rows[x].Cells[1].Value.ToString();

break;

default:

main += LexGrid.Rows[x].Cells[1].Value.ToString();

break;

}

x++;

} while (LexGrid.Rows[x].Cells[1].Value.ToString() != ")");

//main += "GlobalVar." + LexGrid.Rows[x - 1].Cells[1].Value.ToString() + LexGrid.Rows[x].Cells[1].Value.ToString() + LexGrid.Rows[x + 1].Cells[1].Value.ToString() + LexGrid.Rows[x + 2].Cells[1].Value.ToString();

main += " = Convert.ToInt32(Console.ReadLine()); \n";

}

}

else

{

outp.Add(LexGrid.Rows[x].Cells[1].Value.ToString());

main += LexGrid.Rows[x - 1].Cells[1].Value.ToString() + "=";

main += "Convert.ToInt32(Console.ReadLine()); \n";

}

}

}

foreach (string b in doublelist)

{

if (b == LexGrid.Rows[x].Cells[1].Value.ToString())

{

x++;

if (LexGrid.Rows[x].Cells[1].Value.ToString() == "[")

{

if (LexGrid.Rows[x + 3].Cells[1].Value.ToString() == "[")

{

x--;

do

{

switch (LexGrid.Rows[x].Cells[2].Value.ToString())

{

case "id":

main += LexGrid.Rows[x].Cells[1].Value.ToString();

break;

default:

main += LexGrid.Rows[x].Cells[1].Value.ToString() + LexGrid.Rows[x + 1].Cells[1].Value.ToString() + "," + LexGrid.Rows[x + 4].Cells[1].Value.ToString() + LexGrid.Rows[x + 5].Cells[1].Value.ToString();

x = x + 5;

break;

}

x++;

} while (LexGrid.Rows[x].Cells[1].Value.ToString() != ")");


//main += "GlobalVar." + LexGrid.Rows[x - 1].Cells[1].Value.ToString() +
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.ToDouble(Conso
le.ReadLine()); \n";

}

else

{

x--;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

default:

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

}

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ")");


//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.ToDouble(Conso
le.ReadLine()); \n";

}

}

else

{

//outp.Add(LexGrid.Row
s[x].Cells[1].Value.To
String());

main += LexGrid.Rows[x
-
1].Cells[1].Value.ToSt
ring() + "=";

main +=
"Convert.ToDouble(Cons
ole.ReadLine()); \n";

}

//outp.Add(LexGrid.Row
s[x].Cells[1].Value.To
String());

//consolewrt +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"=";

//consolewrt +=
"Convert.ToDouble(Cons
ole.ReadLine()); \n";

}

}

foreach (string c in
stringlist)

{

if (c ==
LexGrid.Rows[x].Cells[
1].Value.ToString())

{

x++;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "[")

{

if (LexGrid.Rows[x +
3].Cells[1].Value.ToSt
ring() == "[")

{

x--;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

default:

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "," +
LexGrid.Rows[x +
4].Cells[1].Value.ToSt
ring() +
LexGrid.Rows[x +
5].Cells[1].Value.ToSt
ring();

x = x + 5;

break;

}

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ")");


//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 += " =
Console.ReadLine();
\n";

}

else

{

x--;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

default:

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

}

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ")");


//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 += " =
Console.ReadLine();
\n";

}

}

else

{

//outp.Add(LexGrid.Row
s[x].Cells[1].Value.To
String());

main += LexGrid.Rows[x
-
1].Cells[1].Value.ToSt
ring() + "=";

main +=
"Console.ReadLine();
\n";

}


//outp.Add(LexGrid.Row
s[x].Cells[1].Value.To
String());

//consolewrt +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"=";

//consolewrt +=
"Console.ReadLine();
\n";

}

}
```

```
foreach (string c in
charlist)

{

if (c ==
LexGrid.Rows[x].Cells[
1].Value.ToString())

{

x++;

if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "[")

{

if (LexGrid.Rows[x +
3].Cells[1].Value.ToSt
ring() == "[")

{

x--;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

default:

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "," +
LexGrid.Rows[x +
4].Cells[1].Value.ToSt
ring() +
LexGrid.Rows[x +
5].Cells[1].Value.ToSt
ring();
```

```
x = x + 5;

break;

}

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ")");


//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 += " =
Console.ReadKey().KeyC
har; \n";

}

else

{

x--;

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

default:

main +=
```

```
LexGrid.Rows[x].Cells[
1].Value.ToString();

break;

}

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ")");


//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 += " =
Console.ReadKey().KeyC
har; \n";

}

}

else

{

//outp.Add(LexGrid.Row
s[x].Cells[1].Value.To
String());

main += LexGrid.Rows[x
-
1].Cells[1].Value.ToSt
ring() + "=";

main +=
"Console.ReadKey().Key
Char; \n";

}

//outp.Add(LexGrid.Row
s[x].Cells[1].Value.To
String());

//consolewrt +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"=";

//consolewrt +=
"Console.ReadLine();
\n";

}

}

}

}

x++;

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");


}

break;


case "post":

x++;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "(")

{

main +=
"Console.Write(";

do

{

x++;

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "Stringlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Numlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Charlit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"Declit" ||
LexGrid.Rows[x].Cells[
2].Value.ToString() ==
"id")

{

do

{

switch
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "Stringlit":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "Numlit":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "Declit":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "Charlit":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();


x++;

break;

case "AFFIRMATIVE":

main += "true";

x++;

break;

case "NEGATIVE":

main += "false";

x++;

break;

case "id":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case ",":

main += " , ";

x++;

break;

case "+":

main += " + ";

x++;

break;

case "[":

main += " [ ";

x++;

if
(LexGrid.Rows[x+2].Cel
ls[1].Value.ToString()
== "[")

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"," + LexGrid.Rows[x +
3].Cells[1].Value.ToSt
ring() + "]";

x += 5;

}

break;

case "]":

main += " ] ";

x++;

break;

case "(":

main += "(";

x++;

break;

case ")":

main += ")";

x++;

break;

default:

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");


}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";");

main += "; \n";

}

break;

case "id":

do

{

switch
```

```
(LexGrid.Rows[x].Cells
[2].Value.ToString())

{

case "id":

main += " " +
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

if(LexGrid.Rows[x].Cel
ls[1].Value.ToString()
== "[")

{

if
(LexGrid.Rows[x+3].Cel
ls[1].Value.ToString()
== "[")

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
LexGrid.Rows[x +
1].Cells[1].Value.ToSt
ring() + "," +
LexGrid.Rows[x +
4].Cells[1].Value.ToSt
ring() + "]";

x += 6;

break;

}

else

{

main +=
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();

x += 3;

break;

}

}

break;

case "=":

main += "=";

x++;

break;

case "+":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "-":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "*":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "/":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "%":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "^":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "(":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case ")":

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

break;

case "~":

main += " -";

x++;

break;

case "Extent":

main += "Length";

x++;

break;

case "ToJoeRange":

main +=
"ToCharArray()";

x++;

break;

case "Carry":

main += "Contains";

x++;

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ";");

break;

case "sqrt":

x++;

main += "Math.Sqrt";

break;

default:

//if
(LexGrid.Rows[x].Cells
[1].Value.ToString()
== "[")

//{

//    main += "[";

//    x++;

//    do

//    {

//        if
(LexGrid.Rows[x +
3].Cells[2].Value.ToSt
ring() == "=")

//        {

//            do

//            {

//                main
+=
LexGrid.Rows[x].Cells[
1].Value.ToString();

//                x++;

//            } while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ";");

//        }

//        else

//        {

//            main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"," + LexGrid.Rows[x +
2].Cells[1].Value.ToSt
ring() + "]";

//            x += 5;

//        }

//    } while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ";");

//} else {

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();
x++;

//}

break;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ";") ;

main += "; \n";

break;

case "Swap":

main +=
"Array.Reverse";

x++;

do

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= ";");

main += ";";

break;

case "@":

do

{

x++;

} while
(LexGrid.Rows[x].Cells
[1].Value.ToString()
!= "@");

break;

case "inorder":

x++;

main += "if";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "Carry")

{

main += "Contains";

x++;

}

else

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{");

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"\n";

//x++;

break;

case "otherorder":

x++;

main += "else if";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "Carry")

{

main += "Contains";

x++;

}

else

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= "{");

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"\n";

break;

case "order":

x++;

main += "else {\n";

break;

case "campaign":

x++;

main += "switch";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}

else

{

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString();

x++;

}

} while
(LexGrid.Rows[x].Cells
[2].Value.ToString()
!= ")");

main +=
LexGrid.Rows[x].Cells[
1].Value.ToString() +
"\n" + "{ \n";

break;

case "operation":

main += "case ";
```

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

main += LexGrid.Rows[x].Cells[1].Value.ToString();
x++;

}
while(LexGrid.Rows[x].Cells[2].Value.ToString() != ″;″);
main += ″;\n″;
break;
case ″++″:
main += ″++″;
x++;
do
{

main += LexGrid.Rows[x].Cells[1].Value.ToString();
x++;

} while (LexGrid.Rows[x].Cells[2].Value.ToString() != ″;″);
main += ″;\n″;
break;
case ″go″:
x++;
main += ″do {″;
break;
case ″phase″:
x++;
main += ″while″;
do
{
if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″id″)
{
main += LexGrid.Rows[x].Cells[1].Value.ToString();
x++;
}
else if (LexGrid.Rows[x].Cells[2].Value.ToString() == ″&″)
{
main += ″ && ″;
x++;
}
else
{
main += LexGrid.Rows[x].Cells[1].Value.ToString();
x++;
}
} while (LexGrid.Rows[x].Cells[2].Value.ToString() != ″)″);
main += LexGrid.Rows[x].Cells[1].Value.ToString();
x++;
if (LexGrid.Rows[x].Cells[1].Value.ToString() == ″;″)
```

```
{

main += "; \n";

}

else

{

main += " { \n";

}

break;

case "backup":

x++;

main += "return ";

do

{

if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "id")

{

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[
1].Value.ToString() +
"\n";
```

```
break;

case "commence":

main +=
"Console.Clear(); \n";

x++;

break;
                }

                if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "}")
                {

x++;
                    if
(LexGrid.Rows[x].Cells
[2].Value.ToString()
== "deploy")
                    {

x += 2;

main +=
"Console.ReadLine();
\n } \n";
                }

else
                    {

main += LexGrid.Rows[x
-
1].Cells[1].Value.ToSt
ring() + " \n";

x--;
                    }
                }
            }


globdeclare += "\n";
            //function
+= "} \n";
            consolewrt
+= globdeclare + main
+ function;

richTextBoxEx1.Text +=
consolewrt;

richTextBoxEx1.Text +=
"} \n } \n ";

//MessageBox.Show(rich
TextBoxEx1.Text);
```

```
CodeDomProvider
codeProvider =
CodeDomProvider.Create
Provider("CSharp");
            string
Output = "Out.exe";
           // Button
ButtonObject =
(Button)sender;


System.CodeDom.Compile
r.CompilerParameters
parameters = new
CompilerParameters();

parameters.GenerateExe
cutable = true;

parameters.OutputAssem
bly = Output;

CompilerResults
results =
codeProvider.CompileAs
semblyFromSource(param
eters,
richTextBoxEx1.Text);

            if
(results.Errors.Count
> 0)
            {

textBox2.ForeColor =
Color.Red;

foreach (CompilerError
CompErr in
results.Errors)
                {

int x=0;

textBox2.Text =
textBox2.Text +

"Line number " +
CompErr.Line +

", Error Number: " +
CompErr.ErrorNumber +

", '" +
CompErr.ErrorText +
";" +

Environment.NewLine +
Environment.NewLine;
```

```
                    if
(CompErr.ErrorText.Con
tains("test.test"))
                    {

                    }

else {

x++;

dataGridViewX4.Rows.Ad
d(x,
CompErr.ErrorText);
                    }
                }
            }
            else
            {

textBox2.ForeColor =
Color.Blue;

textBox2.Text =
"Success!";

Process.Start(Output);

buttonX12.Enabled =
false;

buttonX13.Enabled =
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
sender, EventArgs e)
        {
```

```

File.Delete(″Out.exe″)
;
            consolewrt
= ″″;

richTextBoxEx1.Text =
″″;

buttonX12.Enabled =
true;

buttonX13.Enabled =
false;

        }

    }
}
```

Lexical Analyzer: Dictionary.cs

```
using
System.Collections.Gen
eric;

namespace
Lexical_Analyzer
{
    public class
Dictionary
    {

        //RESERVED
SYMBOLS
        public class
ReservedWords
        {
            public
List<string> rw_1 =
new List<string> {
″company″, ″unit″,
″digit″, ″response″,
″joe″, ″hold″, ″miss″,
″operation″, ″struct″
};
            public
List<string> rw_2 =
new List<string> {
″PrimaryMission″,
″post″, ″capture″,
″backup″, ″campaign″,

″abort″, ″deploy″,
″inquire″, ″inorder″,
″otherorder″,

″phase″, ″Swap″,
″Carry″, ″sqrt″ };
            public
List<string> rw_3 =
new List<string> {
″go″, ″order″ };
            public
List<string> rw_4 =
new List<string> {
″AFFIRMATIVE″,
″NEGATIVE″,
″commence″,
″ToJoeRange″, ″Extent″
};
            public
List<string> rw_5 =
new List<string> {
″action″ };
         }

        public class
ReservedWordsDelims
        {
            public
List<char> delim_1 =
new List<char> { ’ ’
};
            public
List<char> delim_2 =
new List<char> { ’(’
};
            public
List<char> delim_3 =
new List<char> { ’ ’,
’{’ };
            public
List<char> delim_end =
new List<char> { ’.’,
’ ’, ’\n’, ’\t’,’(’ ,
’:’, ’,’, ’\’’, ’[’,
’]’, ’?’, ’#’, ’$’,
’%’, ’\\’,

’)’, ’″’, ’;’, ’@’,
’^’, ’~’, ’`’, ’_’,
’!’, ’<’, ’>’,’*’, ’/’
};
            public
List<char> delim_4 =
new List<char> { ’;’
};
            public
List<char> delim_5 =
new List<char> { ’:’
};


        }

        public class
ReservedSymbols
        {
            public
List<string> rs_1 =
new List<string> {
″+″, ″-″, ″*″, ″%″ ,
″/″};
            public
List<string> rs_2 =
new List<string> {
″++″, ″--″ };
            public
List<string> rs_3 =
new List<string> {
″+=″, ″+(″, ″-=″, ″-
(″, ″*(″, ″/(″, ″==″
};
            public
List<string> rs_4 =
new List<string> {
″\\″ };
            public
List<string> rs_5 =
new List<string> { ″;″
};
            public
List<string> rs_6 =
new List<string> { ″(″
};
            public
List<string> rs_7 =
new List<string> { ″)″
};
            public
List<string> rs_8 =
new List<string> { ″{″
};
            public
List<string> rs_9 =
new List<string> {
″++″, ″--″ };
            public
List<string> rs_10 =
new List<string> {
″<=″, ″>=″, ″!=″ };
            public
List<string> rs_11 =
new List<string> { ″,″
};
            public
List<string> rs_12 =
new List<string> {
″<″, ″>″ ,″=″ };
            public
List<string> rs_13 =
new List<string> { ″~″
}; //Negation
            public
List<string> rs_14 =
new List<string> { ″^″
}; //Power
            public
List<string> rs_15 =
new List<string> { ″#″
}; //Address
            public
List<string> rs_16 =
new List<string> { ″:″
};
            public
List<string> rs_17 =
new List<string> {
″||″ };
           // public
List<string> rs_18 =
new List<string> { ″$″
};
            public
List<string> rs_18 =
new List<string> { ″}″
};
            public
List<string> rs_19 =
new List<string> { ″!″
};
            public
List<string> rs_20 =
new List<string> { ″&″
};
            public
List<string> rs_21 =
new List<string> { ″[″
};
            public
List<string> rs_22 =
new List<string> { ″]″
};
            public
List<string> rs_23 =
new List<string> { ″.″
};
            public
List<string> rs_24 =
new List<string> {
″\″″ };

        }

        public class
ReservedSymbolsDelims
        {
            public
List<char> del1 = 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> 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’,
’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> del5 = new
List<char> { ’\n’,’
’,’)’,’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> del6 = 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> 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’,
’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’,’ ’, ’\n’,

’″’, ’\’’ };
            public
List<char> del9 = 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> 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’,
’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> del12 = new
List<char> {
’=’,’″’,’\’’,’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’,
’0’, ’1’, ’2’, ’3’,
’4’, ’5’, ’6’, ’7’,
’8’, ’9’,

’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> del13 = 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> del14 = 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> del15 = 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’,

’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> del16 = new
List<char> { ’
’,’\n’,’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> del17 = 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> del18 = new
List<char> {
’d’,’f’,’s’ };*/
           public
List<char> del18 = new
List<char> { ’;’,’
’,’\n’,’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> del19 = 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> del21 = new
List<char> { ’0’, ’1’,
’2’, ’3’, ’4’, ’5’,
’6’, ’7’, ’8’, ’9’ };
           public
List<char> del20 = 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> del21 = 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> del22 = 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> del23 = 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> del24 = new
List<char> { ')’ };


                public
List<char> delim_end =
new List<char> { '.’,
' ’, '\n’, '\t’,'(’ ,
':’, ',’, '[’, ']’,
'?’, '#’, '$’, '%’,
'\\’,

')’, ';’, '@’, '^’,
'~’, '`’, '_’, '!’,
'<’,'"’,

'>’,'*’, '/’,
'&’,'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’ };


        }


        //LITERALS
        public class
Literals
        {
            public
List<char> nums = new
List<char> { '~’, '0’,
'1’, '2’, '3’, '4’,
'5’, '6’, '7’, '8’,
'9’ };
        }
        public class
LiteralsDelims
        {
            /* public
List<char> delim_txt =
new List<char> { ' ’,
';’, ',’, ')’, '.’ };
            public
List<char> delim_num =
new List<char> { '+’,
'-’, '*’, '/’, ',’,
'&’, '|’, ')’, ';’ };
*/
            public
List<char> delim_txt =
new List<char> { ' ’,
'\n’, ';’, ',’, ')’,
'.’, '<’, '=’, '[’ };
            public
List<char> delim_num =
new List<char> { '+’,
'-’, '*’, '/’, '.’, '
’, '\n’, ';’, '&’,
'|’, ')’, ',’, '&’,
']’ };
        }

        //Identifier
        public class
Identifier
        {
            public
List<char> delim_digit
= new List<char> {
'1’, '2’, '3’, '4’,
'5’, '6’, '7’, '8’,
'9’, '0’ };
            public
List<char>
delim_lowlet = 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’ };
            public
List<char>
delim_caplet = 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’ };
            public
List<char>
delim_undscr = new
List<char> { '_’ };
            public
List<char> id = new
List<char> {};
        }
        public class
IdentifierDelims
        {
            public
List<char> delim_end =
new List<char> {
'.’,'\n’, '=’,
'\t’,'(’ , ':’, ',’,
'\’’, '[’, ']’, '?’,
'#’, '$’, '%’, '\\’,

')’, '"’, ';’, '@’,
'^’, '~’, '`’, '_’,
'!’, '<’, '>’,'*’,
'/’, '+’, '-’,' ’};
        }

        //OTHER
DELIMITERS
        public class
Delims
        {
            public
List<char> delim_zero
= new List<char> { '0’
};
            public
List<char>
delim_lowlet = 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’ };
            public
List<char>
delim_mathOp = new
List<char> { '+’, '-’,
'*’, '/’ };
            public
List<char>
delim_undscr = new
List<char> { '_’ };
            public
List<char>
delim_identifier = new
List<char>();
            public
List<char> delim_lit =
new List<char>();
            public
List<char> delim_12 =
new List<char>();
            public
List<char> delim_16 =
new List<char> { ' ’
};
        }

    }
}
```

Lexical Analyzer: Initializer.cs

```
using System;
using
System.Collections.Generic;
using System.Linq;

//Unused Libraries
//using System.Text;
//using
System.Threading.Tasks
;
//using
System.Collections.Generic;

namespace
Lexical_Analyzer
{
    public class
Initializer
    {
        public int
tokens = 0;

//INITIALIZATION
        public
LexicalAnalyzer
InitializeAnalyzer(string txt,
LexicalAnalyzer lex)
        {
            Boolean
hastoken = false;
            Tokens t =
new Tokens();
            //txt =
txt.TrimStart();

lex.token.Clear();

lex.invalid = 0;
            lex.valid
= 0;
```

```
            while (txt
!= "")
            {
                if
(hastoken =
lex.GetTokenLines(txt,
tokens))
                {

txt = txt.Remove(0,
lex.ctr);

tokens--;
                }
                else
if (hastoken =
lex.GetReservedWords(t
xt))

txt = txt.Remove(0,
lex.ctr);
                else
if (hastoken =
lex.GetReservedSymbols
(txt))

txt = txt.Remove(0,
lex.ctr);
                else
if (hastoken =
lex.GetLiterals(txt))

txt = txt.Remove(0,
lex.ctr);
                else
if (hastoken =
lex.GetIdentifiers(txt
))

txt = txt.Remove(0,
lex.ctr);
                else
                {
                 t =
new Tokens();

lex.invalid++;
                    //
lex.token.Add("INVALID
");
                    if
(lex.state != 0)
                    {

switch (lex.state)

{

case 1:

lex.ctr = GetCtr(txt,
1);
```

```

break;

}
                    }
                    if
(lex.ctr == 0 &&
txt.Length != 1)
lex.ctr = GetCtr(txt);

else if (lex.ctr == 0
&& txt.Length == 1)
lex.ctr = 1;

else if (lex.ctr >=
txt.Length) lex.ctr =
txt.Length;

t.setTokens("INVALID")
;

t.setLexemes(txt.Subst
ring(0, lex.ctr));

lex.token.Add(t);

txt = txt.Remove(0,
lex.ctr);

                }

tokens++;
                //txt
= txt.TrimStart();
            }

lex.linetokens.Add(tok
ens);
            lex =
setLines(lex);

            return
lex;
        }

        private
LexicalAnalyzer
setLines(LexicalAnalyz
er lex)
        {
            for (int
ctr = 0; ctr <
lex.token.Count;
ctr++)
            {
                for
(int i = 0; i <
lex.linetokens.Count;
i++)
                {
```

```
                    if
(ctr + 1 <=
lex.linetokens[i])
                    {

lex.token[ctr].setLine
s(i + 1);

break;
                    }
                }
            }
            return
lex;
        }

        //GET CTRS
        private int
GetCtr(string txt)
        {

Dictionary.ReservedWor
dsDelims rwd = new
Dictionary.ReservedWor
dsDelims();
            Dictionary
td = new Dictionary();

            Boolean
ifEnd = false;
            int ctr =
0;

            foreach
(var item in
rwd.delim_end)
            {
                if
(txt.ElementAt(ctr -
1) == item)

ifEnd = true;
            }
            while
(ifEnd != true)
            {

foreach (var item in
rwd.delim_end)
                {
                    if
((txt.Length) > ctr)
                    {

if (txt.ElementAt(ctr)
== item)

{

ifEnd = true;

break;
```

```
}
                    }

else ifEnd = true;
                }
                if
(ifEnd != true)

ctr++;
            }

            if
(!(txt.Length >= ctr))
ctr--;
            return
ctr;
        }
        private int
GetCtr(string txt, int
ctr)
        {
            Boolean
notEnd = true;
            List<char>
delims = new
List<char>{ '"', '\\',
'\n' };
            while
(notEnd && (txt.Length
- 1) >= ctr)
            {

foreach (char c in
delims)
                {
                    if
((txt.Length - 1) >
ctr)
                    {

if (c ==
txt.ElementAt(ctr))

{

notEnd = false;

if (c == '\\')

if (txt.Length - 1 !=
ctr)

ctr++;

}
                    }

else

notEnd = false;
                }
                ctr++;
            }
            return
ctr;
        }
    }
}
```

Lexical Analyzer:
Lexical Analyzer.cs

```
using System;
using
System.Collections.Gen
eric;
using System.Linq;

//Unused Libraries
//using System.Text;
//using
System.Threading.Tasks
;
using TokenLibrary;

namespace
Lexical_Analyzer
{
    public class
Tokens : TokensClass
    {

    }
    public class
LexicalAnalyzer
    {
        public
List<Tokens> token =
new List<Tokens>();
        public
List<int> linetokens =
new List<int>();
        Boolean
isReserved = false;
        public int
invalid = 0;
        public int
valid = 0;
        public int ctr
= 0;
        public byte
state = 0;
        public int
lines = 0;
        public int
idnum = 1;
        Dictionary td
= new Dictionary();

        public Boolean
GetTokenLines(string
txt, int tokenctr)
        {
            Boolean
hastokenlines = false;
            if
(txt.ElementAt(0) ==
’\n’)
            {

lines++;

linetokens.Add(tokenct
r);

hastokenlines = true;
                ctr =
1;
            }
            else if
(txt.ElementAt(0) == ’
’)
            {

hastokenlines = true;
                ctr =
1;
            }
            return
hastokenlines;
        }

        //GET TOKENS
        public Boolean
GetReservedWords(strin
g txt)
        {

Dictionary.ReservedWor
dsDelims rwd = new
Dictionary.ReservedWor
dsDelims();

Dictionary.ReservedWor
ds rw = new
Dictionary.ReservedWor
ds();
            Tokens t =
new Tokens();


List<String> words;
            List<char>
delims;

List<String> temp;
            Boolean
found = false,
hastoken = false,
exitfor = false, ifEnd
= false, nodelim =
true;
            int
tempctr = 0, limit =
0;

            if
(txt.Length != 1)
            {
                while
((txt.Length - 1) >
tempctr &&
!isEnd(txt[tempctr +
1], rwd))
                {

tempctr++;
                }

tempctr++;
            }

            for (int i
= 0; i < 5; i++)
            {
                ctr =
0;
                words
= new List<String>();
                delims
= new List<char>();
                found
= true;
                switch
(i)
                {

case 0:

words = rw.rw_1;

delims = rwd.delim_1;

break;

case 1:

words = rw.rw_2;

delims = rwd.delim_2;

break;

case 2:

words = rw.rw_3;

delims = rwd.delim_3;

break;

case 3:

words = rw.rw_4;
delims = rwd.delim_4;

break;

case 4:

words = rw.rw_5;

delims = rwd.delim_5;

break;
                }

//Check Reserved Words

foreach (char c in
txt)
                {

limit = words.Count -
1;

temp = new
List<string>();

found = false;

foreach (string w in
words)
                    {

//IF NOT OUT OF RANGE

if ((w.Length - 1) >=
ctr)

{

//IF LETTER MATCHED

if (c ==
w.ElementAt(ctr))

{

found = true;

//CHECK SIZE OF WORD
AND INPUT

if (w.Length ==
tempctr)

{

//CHECK DELIMITER

if ((tempctr - 1) ==
ctr)

{
```

```
foreach (char delim in
delims)

{

//IF NOT OUT OF RANGE

if ((txt.Length - 1) >
ctr)

{

//IF FOUND DELIMITER

if (txt[ctr + 1] ==
delim)

{

hastoken = true;

nodelim = false;

t.setTokens(w);

t.setLexemes(w);

t.setAttributes(w);

token.Add(t);

valid++;

break;

}

}

else if (w ==
words[limit] &&
hastoken == false) {
found = false; }


}


if (hastoken == false)

{

hastoken = true;

nodelim = false;

found = true;

t.setTokens("NODELIM")
;
```

```
t.setLexemes(w);

t.setAttributes(w);

token.Add(t);

invalid++;


}

else if (nodelim)

{

hastoken = true;

found = true;

t.setTokens("INVALID")
;

t.setLexemes(w);

t.setAttributes(w);

token.Add(t);

invalid++;

break;

}

if (hastoken)

{

break;

}

}

else temp.Add(w);

}


}

}
                    }

ctr++;

words = temp;
                    if
(found == false)
break;
```

```
                    if
(hastoken)
                    {

exitfor = true;

break;
                    }
                }
                if
(exitfor)
                {

exitfor = false;

break;
                }
            }

            //IF
NOTHING FOUND
            if (found
== false)
            {

hastoken = false;

foreach (var item in
rwd.delim_end)
                {
                    if
(txt.ElementAt(ctr -
1) == item)

ifEnd = true;
                }
                while
(ifEnd != true)
                {

foreach (var item in
rwd.delim_end)
                    {

if ((txt.Length) >
ctr)

{

if (txt.ElementAt(ctr)
== item)

{

ifEnd = true;

break;

}

}
```

```
else ifEnd = true;
                    }
                    if
(ifEnd != true)

ctr++;
                }
            }

            if
(!(txt.Length >= ctr))
ctr--;

            return
hastoken;
        }
        /*  public
Boolean
GetReservedSymbols(str
ing txt)
        {

Dictionary td = new
Dictionary();

Dictionary.ReservedSym
bols rs = new
Dictionary.ReservedSym
bols();

Dictionary.ReservedSym
bolsDelims rsd = new
Dictionary.ReservedSym
bolsDelims();
            Boolean
found = false,
hastoken = false,
exitfor = false;

List<String> words;

List<char> delims;

List<String> temp;
            int
tempctr = 0, limit =
0, sctr= 0;

            if
(txt.Length != 1)
            {

while ((txt.Length -
1) > tempctr &&
!isEnd(txt[tempctr +
1],rsd))
                {

tempctr++;
                }
```

```
tempctr++;
              }

              for (int
i = 0; i < 16; i++)
              {
                  sctr
= 0;

words = new
List<String>();

delims = new
List<char>();

found = true;

switch (i)
                  {

case 0:

words = rs.rs_1;

delims = rsd.del1;

break;

case 1:

words = rs.rs_2;

delims = rsd.del2;

break;

case 2:

words = rs.rs_3;

delims = rsd.del3;

break;

case 3:

words = rs.rs_4;

delims = rsd.del4;

break;

case 4:

words = rs.rs_5;

delims = rsd.del5;

break;

case 5:

words = rs.rs_6;

delims = rsd.del6;

break;

case 6:

words = rs.rs_7;

delims = rsd.del7;

break;

case 7:

words = rs.rs_8;

delims = rsd.del8;

break;

case 8:

words = rs.rs_9;

delims = rsd.del9;

break;

case 9:

words = rs.rs_10;

delims = rsd.del10;

break;

case 10:

words = rs.rs_11;

delims = rsd.del11;

break;

case 11:

words = rs.rs_12;

delims = rsd.del12;

break;

case 12:

words = rs.rs_13;

delims = rsd.del13;

break;

case 13:

words = rs.rs_14;

delims = rsd.del14;

break;

case 14:

words = rs.rs_15;

delims = rsd.del15;

break;

case 15:

words = rs.rs_16;

delims = rsd.del16;

break;

                  }

//Check Reserved
Symbols

foreach (char c in
txt)
                  {

limit = words.Count -
1;

temp = new
List<string>();

found = false;

foreach (string w in
words)

{

//IF NOT OUT OF RANGE

if ((w.Length - 1) >=
sctr)

{

if (c ==
w.ElementAt(sctr))

{

found = true;

//CHECK SIZE OF WORD
AND INPUT

if (w.Length ==
tempctr)

{

//CHECK DELIMITER

if ((tempctr - 1) ==
sctr)

{

foreach (char delim in
delims)

{

//IF NOT OUT OF RANGE

if ((txt.Length - 1) >
sctr)

{

if (txt[sctr + 1] ==
delim)

//IF FOUND DELIMITER

{

found = true;

hastoken = true;

tokens.Add(w);

lexemes.Add(w);

valid++;

break;


}

}

else if (w ==
words[limit] &&
hastoken == false)
found = false;

}


if (hastoken) break;
}

else temp.Add(w);

}

}

}

}

sctr++;

words = temp;

if (found == false)
break;

if (hastoken)

{

exitfor = true;

break;

}
                  }
                  if
(exitfor)
                  {

exitfor = false;

break;
                  }
              }
              if
(hastoken) ctr = sctr;

return hastoken;
          } */

        public Boolean
GetReservedSymbols(str
ing txt)
        {
            Dictionary
td = new Dictionary();

Dictionary.ReservedSym
bols rs = new
Dictionary.ReservedSym
bols();

Dictionary.ReservedSym
bolsDelims rsd = new
Dictionary.ReservedSym
bolsDelims();
```

```
            Boolean
found = false,
hastoken = false,
exitfor = false;

            Tokens t =
new Tokens();
            //rsd =
td.AddRange(rsd);

List<String> words;
            List<char>
delims;

List<String> temp;
            int
tempctr = 0, limit =
0, sctr = 0;

            if
(txt.Length != 1)
            {
                while
((txt.Length - 1) >
tempctr &&
!isEnd(txt[tempctr +
1], rsd))
                {

tempctr++;
                }

tempctr++;
            }

            for (int i
= 0; i < 24; i++)
            {
                sctr =
0;
                words
= new List<String>();
                delims
= new List<char>();
                found
= true;
                switch
(i)
                {

case 0:

words = rs.rs_1;

delims = rsd.del1;

break;

case 1:

words = rs.rs_2;
```

```
delims = rsd.del2;

break;

case 2:

words = rs.rs_3;

delims = rsd.del3;

break;

case 3:

words = rs.rs_4;

delims = rsd.del4;

break;

case 4:

words = rs.rs_5;

delims = rsd.del5;

break;

case 5:

words = rs.rs_6;

delims = rsd.del6;

break;

case 6:

words = rs.rs_7;

delims = rsd.del7;

break;

case 7:

words = rs.rs_8;

delims = rsd.del8;

break;

case 8:

words = rs.rs_9;

delims = rsd.del9;

break;

case 9:
```

```
words = rs.rs_10;

delims = rsd.del10;

break;

case 10:

words = rs.rs_11;

delims = rsd.del11;

break;

case 11:

words = rs.rs_12;

delims = rsd.del12;

break;

case 12:

words = rs.rs_13;

delims = rsd.del13;

break;

case 13:

words = rs.rs_14;

delims = rsd.del14;

break;

case 14:

words = rs.rs_15;

delims = rsd.del15;

break;

case 15:

words = rs.rs_16;

delims = rsd.del16;

break;

case 16:

words = rs.rs_17;

delims = rsd.del17;

break;
```

```
case 17:

words = rs.rs_18;

delims = rsd.del18;

break;

case 18:

words = rs.rs_19;

delims = rsd.del19;

break;

case 19:

words = rs.rs_20;

delims = rsd.del20;

break;

case 20:

words = rs.rs_21;

delims = rsd.del21;

break;

case 21:

words = rs.rs_22;

delims = rsd.del22;

break;

case 22:

words = rs.rs_23;

delims = rsd.del23;

break;

case 23:

words = rs.rs_24;

delims = rsd.del24;

break;
                }

foreach (char c in
txt)
                {
```

```
limit = words.Count -
1;

temp = new
List<string>();

found = false;

foreach (string w in
words)
                    {

//IF NOT OUT OF RANGE

if ((w.Length - 1) >=
sctr)

{

if (c ==
w.ElementAt(sctr))

{

found = true;

//CHECK SIZE OF WORD
AND INPUT

if (w.Length ==
tempctr)

{

//CHECK DELIMITER

if ((tempctr - 1) ==
sctr)

{

foreach (char delim in
delims)

{

//IF NOT OUT OF RANGE

if ((txt.Length - 1) >
sctr)

{

//IF FOUND DELIMITER

if (txt[sctr + 1] ==
delim)

{

found = true;
```

```
hastoken = true;

t = new Tokens();

t.setTokens(w);

t.setLexemes(w);

t.setAttributes(w);

token.Add(t);

valid++;

break;


}

}

else if (w ==
words[limit] &&
hastoken == false)
found = false;

}


if (hastoken) break;


}

else temp.Add(w);

}

}

}
                    }

sctr++;

words = temp;
                    if
(found == false)
break;
                    if
(hastoken)
                    {

exitfor = true;

break;
                    }
                }

                if
(exitfor)
```

```
                    {

exitfor = false;

break;
                    }
                }
                if
(hastoken) ctr = sctr;
                return
hastoken;
            }

            /* public
Boolean
GetLiterals(string
txt)
            {

Dictionary.LiteralsDel
ims ld = new
Dictionary.LiteralsDel
ims();

Dictionary.Literals l
= new
Dictionary.Literals();
                List<char>
delims = new
List<char>();
                Boolean
hastoken = false,
validtxt = false;
                string
literal = ””;
                state = 0;
                int lctr =
0;

                if
(txt.ElementAt(lctr)
== '”')
                    state
= 1;
                else if
(txt.ElementAt(lctr)
== '\'')
                    state
= 2;
                else
                {

foreach (char num in
l.nums)
                    {
                        if
(txt.ElementAt(lctr)
== num)

state = 3;
                    }
                }
            if (state
!= 0)
            {
                switch
(state)
                {

case 1: case 2:

delims = ld.delim_txt;

//String Literal
Analyzer

if (state == 1)

{

if (txt.Length != 1)

{

while ((txt.Length -
1) > lctr &&
!(txt[lctr + 1] ==
'"') && !(txt[lctr +
1] == '\n'))

{

literal +=
txt[lctr].ToString();

lctr++;

}

if ((txt.Length - 1)
== lctr && (txt[lctr]
!= '"'))

hastoken = false;

else

{

if (!(lctr == 1 &&
txt[lctr] == '\\'))

{

validtxt = true;

lctr++;

foreach (char c in
delims)

{

if ((txt.Length - 1)
>= (lctr + 1))

if (txt[lctr + 1] ==
c)

{

hastoken = true;

break;

}

}

}

if (hastoken &&
validtxt)

{

valid++;

tokens.Add("Stringlit"
);

lexemes.Add(txt.Substr
ing(0, (lctr + 1)));

ctr = lctr + 1;

}

else if (!validtxt)

{

ctr = lctr + 2;

hastoken = false;

}


}

}

}


//Character Literal
Analyzer

else

{

if (txt.Length != 1)

{


while ((txt.Length -
1) > lctr &&
!(txt[lctr + 1] ==
'\'') && !(txt[lctr +
1] == '\n'))

{

literal +=
txt[lctr].ToString();

lctr++;

}

if (lctr >= 3)

{

hastoken = false;

ctr = lctr + 2;

if (ctr > txt.Length)

ctr = txt.Length;

}

else

{

if ((txt[1] == '\\' &&
lctr == 2) || (lctr ==
1 && txt[1] != '\\')
|| lctr == 0)

validtxt = true;

else

{

validtxt = false;

hastoken = false;

ctr = lctr + 2;

if (ctr > txt.Length)

ctr = txt.Length;

}

if (validtxt)

{

if ((txt.Length - 1)
>= (lctr + 1) &&
txt[lctr + 1] == '\'')

{

lctr++;

foreach (char c in
delims)

{

if ((txt.Length - 1)
>= (lctr + 1))

if (txt[lctr + 1] ==
c)

{

hastoken = true;

break;

}

}

}

if (hastoken)

{

valid++;

tokens.Add("Charlit");

lexemes.Add(txt.Substr
ing(0, (lctr + 1)));

ctr = lctr + 1;

}

else

{


ctr = lctr + 1;

if (ctr > txt.Length)

ctr = lctr;

}
}
}
}
}
break;

case 3:
Dictionary.Identifier id = new Dictionary.Identifier();
delims = ld.delim_num;
Boolean isNumNext = true, hasnum = true, hasid = false;
List<char> num = new List<char> { ’0’, ’1’, ’2’, ’3’, ’4’, ’5’, ’6’, ’7’, ’8’, ’9’ };

id.id.AddRange(id.delim_caplet);
id.id.AddRange(id.delim_caplet);

//If Negative
if (txt.ElementAt(lctr) == ’-’)
{
hasnum = false;
foreach (char n in num)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == n)
{
hasnum = true;
lctr++;
}
}
}

if (hasnum)
{
while (isNumNext)
{
isNumNext = false;
foreach (char n in num)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == n)
{
lctr++;
isNumNext = true;
}
}
}

//Double Literal Analyzer
Boolean isDouble = false;
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == ’.’)
{
if ((txt.Length - 1) > lctr + 1)
foreach (char n in num)
{
if (txt.ElementAt(lctr + 2) == n)
isDouble = true;
}
}

//Double Literal Analyzer
if (isDouble)
{
lctr++;
isNumNext = true;
while (isNumNext)
{
isNumNext = false;
foreach (char n in num)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == n)
{
lctr++;
isNumNext = true;
}
}
}

foreach (char delim in delims)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == delim)
{
hastoken = true;
break;
}
}

if (hastoken)
{
valid++;
tokens.Add(”Declit”);
lexemes.Add(txt.Substring(0, (lctr + 1)));
}
else
{
foreach (char c in id.id)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == c)
{
hasid = true;
}
}
}

if (!hasid)
ctr = lctr + 1;
}
```

```
//Integer Literal
Analyzer

else

{

foreach (char delim in
delims)

{

if (txt.ElementAt(lctr
+ 1) == delim)

{

hastoken = true;

break;

}

}


if (hastoken)

{

valid++;

tokens.Add("Numlit");

lexemes.Add(txt.Substr
ing(0, (lctr + 1)));

}

else

{

foreach (char c in
id.id)

{

if (txt.ElementAt(lctr
+ 1) == c)

{

hasid = true;

}

}

}
```

```
if (!hasid)

ctr = lctr + 1;

}

}

break;
                    }
                }
                return
hastoken;
            } */

            public Boolean
GetLiterals(string
txt)
            {

Dictionary.LiteralsDel
ims ld = new
Dictionary.LiteralsDel
ims();

Dictionary.Literals l
= new
Dictionary.Literals();
                Tokens t =
new Tokens();
                List<char>
delims = new
List<char>();
                Boolean
hastoken = false,
validtxt = false;
                string
literal = "";
                state = 0;
                int lctr =
0;

                if
(txt.ElementAt(lctr)
== '"')
                    state
= 1;
                else if
(txt.ElementAt(lctr)
== '\'')
                    state
= 2;
                else if
(txt.ElementAt(lctr)
== '@')
                    state
= 3;
                else
                {
```

```
foreach (char num in
l.nums)
                    {
                        if
(txt.ElementAt(lctr)
== num)

state = 4;
                    }
                }

                if (state
!= 0)
                {
                    switch
(state)
                    {

case 1:

case 2:

case 3:

delims = ld.delim_txt;

//String Literal
Analyzer

if (state == 1)

{

if (txt.Length != 1)

{

while ((txt.Length -
1) > lctr &&
!(txt[lctr + 1] ==
'"') && !(txt[lctr +
1] == '\n'))

{

literal +=
txt[lctr].ToString();

lctr++;

}

if ((txt.Length - 1)
== lctr && (txt[lctr]
!= '"'))

hastoken = false;

else

{
```

```
if (!(lctr == 1 &&
txt[lctr] == '\\'))

{

validtxt = true;

lctr++;

foreach (char c in
delims)

{

if ((txt.Length - 1)
>= (lctr + 1))

if (txt[lctr + 1] ==
c)

{

hastoken = true;

break;

}

}

}

if (hastoken &&
validtxt)

{

valid++;

t = new Tokens();

t.setTokens("Stringlit
");

t.setLexemes(txt.Subst
ring(0, (lctr + 1)));

t.setAttributes("Strin
glit");

token.Add(t);

ctr = lctr + 1;

}

else if (!validtxt)

{

ctr = lctr + 2;
```

```
hastoken = false;
}

}
}
}

//Character Literal Analyzer
else if(state == 2)
{
if (txt.Length != 1)
{

while ((txt.Length - 1) > lctr && !(txt[lctr + 1] == ’\’’) && !(txt[lctr + 1] == ’\n’))
{
literal += txt[lctr].ToString();
lctr++;
}
if (lctr >= 3)
{
hastoken = false;
ctr = lctr + 2;
if (ctr > txt.Length)
ctr = txt.Length;
}
else
{
if ((txt[1] == ’\\’ && lctr == 2) || (lctr == 1 && txt[1] != ’\\’) || lctr == 0)
validtxt = true;
else
{
validtxt = false;
hastoken = false;
ctr = lctr + 2;
if (ctr > txt.Length)
ctr = txt.Length;
}
if (validtxt)
{
if ((txt.Length - 1) >= (lctr + 1) && txt[lctr + 1] == ’\’’)
{
lctr++;
foreach (char c in delims)
{
if ((txt.Length - 1) >= (lctr + 1))
if (txt[lctr + 1] == c)
{
hastoken = true;
break;
}
}
}
if (hastoken)
{
valid++;
t = new Tokens();
t.setTokens(″Charlit″);
t.setLexemes(txt.Substring(0, (lctr + 1)));
t.setAttributes(″Charlit″);
token.Add(t);
ctr = lctr + 1;
}
else
{

ctr = lctr + 1;
if (ctr > txt.Length)
ctr = lctr;
}
}
}
}
else if(state == 3) {
if (txt.Length != 1)
{
while ((txt.Length - 1) > lctr && !(txt[lctr + 1] == ’@’) && !(txt[lctr + 1] == ’\n’))
{
literal += txt[lctr].ToString();
lctr++;
}
if ((txt.Length - 1) == lctr && (txt[lctr] != ’@’))
hastoken = false;
else
{
if (!(lctr == 1 && txt[lctr] == ’\\’))
{
validtxt = true;
lctr++;
foreach (char c in delims)
{
if ((txt.Length - 1) >= (lctr + 1))
if (txt[lctr + 1] == c)
{
hastoken = true;
break;
}
}
}
if (hastoken && validtxt)
{
valid++;
t = new Tokens();
t.setTokens(″comment″);
t.setLexemes(txt.Substring(0, (lctr + 1)));
t.setAttributes(″comment″);
token.Add(t);
ctr = lctr + 1;
}
```

```
else if (!validtxt)
{
ctr = lctr + 2;
hastoken = false;
}

}
}
}
break;

case 4:
Dictionary.Identifier id = new Dictionary.Identifier();
delims = ld.delim_num;
Boolean isNumNext = true, hasnum = true, hasid = false;
List<char> num = new List<char> { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };

id.id.AddRange(id.delim_caplet);
id.id.AddRange(id.delim_caplet);
int storedval = 0;
//If Negative
if (txt.ElementAt(lctr) == '~')
{
hasnum = false;
foreach (char n in num)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == n)
{
hasnum = true;
lctr++;
}
}
}

if (hasnum)
{
while (isNumNext)
{
isNumNext = false;
foreach (char n in num)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == n)
{
storedval++;
if (storedval <= 8)
{
lctr++;
isNumNext = true;
}
else if(storedval > 8)
{
isNumNext = false;
hastoken = false;
}
}
}
}

//Double Literal Analyzer
Boolean isDouble = false;
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == '.')
{
if ((txt.Length - 1) > lctr + 1)
foreach (char n in num)
{
if (txt.ElementAt(lctr + 2) == n)
isDouble = true;
}
}

if (isDouble)
{
lctr++;
isNumNext = true;
while (isNumNext)
{
isNumNext = false;
foreach (char n in num)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == n)
{
lctr++;
isNumNext = true;
}
}
}

foreach (char delim in delims)
{
if ((txt.Length - 1) > lctr)
if (txt.ElementAt(lctr + 1) == delim)
{
hastoken = true;
break;
}
}

if (hastoken)
{
valid++;
t = new Tokens();
t.setTokens("Declit");
t.setLexemes(txt.Substring(0, (lctr + 1)));
t.setAttributes("Declit");
token.Add(t);
}
```

```
else

{

foreach (char c in id.id)

{

if ((txt.Length - 1) > lctr)

if (txt.ElementAt(lctr + 1) == c)

{

hasid = true;

}

}

}


if (!hasid)

ctr = lctr + 1;

}

//Integer Literal Analyzer

else

{

foreach (char delim in delims)

{

//if (txt.ElementAt(lctr + 1) == delim)

{

hastoken = true;

break;

}


}


if (hastoken)

{

valid++;

t = new Tokens();

t.setTokens("Numlit");

t.setLexemes(txt.Substring(0, (lctr + 1)));

t.setAttributes("Numlit");

token.Add(t);

}

else

{

foreach (char c in id.id)

{

if (txt.ElementAt(lctr + 1) == c)

{

hasid = true;

}

}

}

if (!hasid)

ctr = lctr + 1;

}

}

break;
                }
            }
            return hastoken;
        }


        /* public Boolean GetIdentifiers(string txt)
        {

Dictionary.Identifier id = new Dictionary.Identifier();

Dictionary.IdentifierDelims delims = new Dictionary.IdentifierDelims();
            Boolean hastoken = false, valID = false, isvalID = true;


id.id.AddRange(id.delim_lowlet);

id.id.AddRange(id.delim_caplet);

id.id.AddRange(id.delim_undscr);

id.id.AddRange(id.delim_digit);

            int ictr = 0;

            foreach (char c in id.id)
            {
                if (txt.ElementAt(ictr) == c)
                {

valID = true;
                }
            }


id.id.AddRange(id.delim_digit);
            if (valID)
            {

//ictr++;

isvalID = true;
                while (isvalID)
                {

isvalID = false;

foreach (char n in id.id)
                    {
if ((txt.Length - 1) > ictr)

if (txt.ElementAt(ictr + 1) == n)

{

ictr++;

isvalID = true;

}
                    }
                    if (ictr > 17)

valID = false;
                }

                if (valID)
                {

foreach (char delim in delims.delim_end)
                    {

if ((txt.Length - 1) > ictr)

if (txt.ElementAt(ictr + 1) == delim)

{

hastoken = true;

break;

}
                    }
                }


if(hastoken)
                {

valid++;

tokens.Add("id");

lexemes.Add(txt.Substring(0, (ictr + 1)));
                }

                ctr = ictr + 1;
            }
            return hastoken;
        }

        //IS ENDS
        public Boolean
isEnd(char c,
Dictionary.ReservedWor
dsDelims rwd)
        {
            Boolean
result = false;
            foreach
(var item in
rwd.delim_end)
            {
                if
(item == c)
                {

result = true;

break;
                }
            }
            return
result;
        }
        public Boolean
isEnd(char c,
Dictionary.ReservedSym
bolsDelims rsd)
        {
            Boolean
result = false;
            foreach
(var item in
rsd.delim_end)
            {
                if
(item == c)
                {

result = true;

break;
                }
            }
            return
result;
        }
        public Boolean
isEnd(char c,
List<char> ld)
        {
            Boolean
result = false;
            foreach
(var item in ld)
            {
                if
(item == c)
                {
```

```

result = true;

break;
                }
            }
            return
result;
        }
    }
} */

        public Boolean
GetIdentifiers(string
txt)
        {

Dictionary.Identifier
id = new
Dictionary.Identifier(
);

Dictionary.IdentifierD
elims delims = new
Dictionary.IdentifierD
elims();
            Boolean
hastoken = false,
valID = false, isvalID
= true;
            Tokens t =
new Tokens();



id.id.AddRange(id.deli
m_lowlet);

id.id.AddRange(id.deli
m_caplet);

id.id.AddRange(id.deli
m_undscr);

id.id.AddRange(id.deli
m_digit);

            int ictr =
0;

            foreach
(char c in id.id)
            {
                if
(txt.ElementAt(ictr)
== c)
                {

valID = true;
                }
            }
```

```
id.id.AddRange(id.deli
m_digit);
            if (valID)
            {

isvalID = true;
                while
(isvalID)
                {

isvalID = false;

foreach (char n in
id.id)
                    {

if ((txt.Length - 1) >
ictr)

if (txt.ElementAt(ictr
+ 1) == n)

{

ictr++;

isvalID = true;

}
                    }
                    if
(ictr >= 18)

valID = false;
                }

                if
(valID)
                {

foreach (char delim in
delims.delim_end)
                    {

if ((txt.Length - 1) >
ictr)

if (txt.ElementAt(ictr
+ 1) == delim)

{

hastoken = true;

break;

}
                    }
                }
```

```
                if
(hastoken)
                {

valid++;
                    t
= new Tokens();

t.setTokens("id");

t.setLexemes(txt.Subst
ring(0, (ictr + 1)));

t.setAttributes("ident
ifier" + idnum);

token.Add(t);

idnum++;
                }

                ctr =
ictr + 1;
            }
            return
hastoken;
        }

        public Boolean
isEnd(char c,
Dictionary.ReservedWor
dsDelims rwd)
        {
            Boolean
result = false;
            foreach
(var item in
rwd.delim_end)
            {
                if
(item == c)
                {

result = true;

break;
                }
            }
            return
result;
        }
        public Boolean
isEnd(char c,
Dictionary.ReservedSym
bolsDelims rsd)
        {
            Boolean
result = false;
            foreach
(var item in
rsd.delim_end)
            {
                if
(item == c)
                {

result = true;

break;
                }
            }
            return
result;
        }
        public Boolean
isEnd(char c,
List<char> ld)
        {
            Boolean
result = false;
            foreach
(var item in ld)
            {
                if
(item == c)
                {

result = true;

break;
                }
            }
            return
result;
        }
    }
}
```

Syntax Analyzer:
SyntaxAnalyzer.cs

```
using Core.Library;

/**
 * <remarks>A class
providing callback
methods for the
 * parser.</remarks>
 */
public abstract class
SyntaxAnalyzer :
Analyzer {

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public override
void Enter(Node node)
{
        switch
(node.Id) {
        case (int)
SyntaxConstants.MAIN_N
:

EnterMainN((Token)
node);
            break;
        case (int)
SyntaxConstants.PRINT_
N:

EnterPrintN((Token)
node);
            break;
        case (int)
SyntaxConstants.SCAN_N
:

EnterScanN((Token)
node);
            break;
        case (int)
SyntaxConstants.CONST_
N:

EnterConstN((Token)
node);
            break;
        case (int)
SyntaxConstants.RETURN
:

EnterReturn((Token)
node);
            break;
        case (int)
SyntaxConstants.SWITCH
_N:

EnterSwitchN((Token)
node);
            break;
        case (int)
SyntaxConstants.CASE_N
:

EnterCaseN((Token)
node);
            break;
        case (int)
SyntaxConstants.BREAK:

EnterBreak((Token)
node);
            break;
        case (int)
SyntaxConstants.FOR_N:

EnterForN((Token)
node);
            break;
        case (int)
SyntaxConstants.IF:

EnterIf((Token) node);
            break;
        case (int)
SyntaxConstants.ELSEIF
_N:

EnterElseifN((Token)
node);
            break;
        case (int)
SyntaxConstants.ELSE_N
:

EnterElseN((Token)
node);
            break;
        case (int)
SyntaxConstants.DO:

EnterDo((Token) node);
            break;
        case (int)
SyntaxConstants.WHILE_
N:

EnterWhileN((Token)
node);
            break;
        case (int)
SyntaxConstants.VOID:

EnterVoid((Token)
node);
            break;
        case (int)
SyntaxConstants.GETCH:

EnterGetch((Token)
node);
            break;
        case (int)
SyntaxConstants.STRUCT
_N:

EnterStructN((Token)
node);
            break;
        case (int)
SyntaxConstants.DEFAUL
T:

EnterDefault((Token)
node);
            break;
        case (int)
SyntaxConstants.CLEAR:

EnterClear((Token)
node);
            break;
        case (int)
SyntaxConstants.SQROOT
:

EnterSqroot((Token)
node);
            break;
        case (int)
SyntaxConstants.PLUS:

EnterPlus((Token)
node);
            break;
        case (int)
SyntaxConstants.MINUS:

EnterMinus((Token)
node);
            break;
        case (int)
SyntaxConstants.TIMES:

EnterTimes((Token)
node);
            break;
        case (int)
SyntaxConstants.DIVIDE
:

EnterDivide((Token)
node);
            break;
        case (int)
SyntaxConstants.MODULU
S:

EnterModulus((Token)
node);
            break;
        case (int)
SyntaxConstants.EQUALS
:

EnterEquals((Token)
node);
            break;
        case (int)
SyntaxConstants.SEMIC:
```

```
EnterSemic((Token)
node);
            break;
        case (int)
SyntaxConstants.DOT:

EnterDot((Token)
node);
            break;
        case (int)
SyntaxConstants.COMMA:

EnterComma((Token)
node);
            break;
        case (int)
SyntaxConstants.AND:

EnterAnd((Token)
node);
            break;
        case (int)
SyntaxConstants.OR:

EnterOr((Token) node);
            break;
        case (int)
SyntaxConstants.NOT:

EnterNot((Token)
node);
            break;
        case (int)
SyntaxConstants.INCREM
ENT:

EnterIncrement((Token)
node);
            break;
        case (int)
SyntaxConstants.DECREM
ENT:

EnterDecrement((Token)
node);
            break;
        case (int)
SyntaxConstants.P_E:

EnterPE((Token) node);
            break;
        case (int)
SyntaxConstants.M_E:

EnterME((Token) node);
            break;
        case (int)
SyntaxConstants.T_E:

EnterTE((Token) node);
            break;
        case (int)
SyntaxConstants.D_E:

EnterDE((Token) node);
            break;
        case (int)
SyntaxConstants.MOD_E:

EnterModE((Token)
node);
            break;
        case (int)
SyntaxConstants.NEWLIN
E:

EnterNewline((Token)
node);
            break;
        case (int)
SyntaxConstants.N_E:

EnterNE((Token) node);
            break;
        case (int)
SyntaxConstants.O_PARE
N:

EnterOParen((Token)
node);
            break;
        case (int)
SyntaxConstants.C_PARE
N:

EnterCParen((Token)
node);
            break;
        case (int)
SyntaxConstants.D_QUOT
E:

EnterDQuote((Token)
node);
            break;
        case (int)
SyntaxConstants.COLON:

EnterColon((Token)
node);
            break;
        case (int)
SyntaxConstants.O_BRAC
KET:

EnterOBracket((Token)
node);
            break;
        case (int)
SyntaxConstants.C_BRAC
KET:

EnterCBracket((Token)
node);
            break;
        case (int)
SyntaxConstants.GREATE
R:

EnterGreater((Token)
node);
            break;
        case (int)
SyntaxConstants.LESS:

EnterLess((Token)
node);
            break;
        case (int)
SyntaxConstants.GREATE
R_E:

EnterGreaterE((Token)
node);
            break;
        case (int)
SyntaxConstants.LESS_E
:

EnterLessE((Token)
node);
            break;
        case (int)
SyntaxConstants.S_OBRA
CKET:

EnterSObracket((Token)
node);
            break;
        case (int)
SyntaxConstants.S_CBRA
CKET:

EnterSCbracket((Token)
node);
            break;
        case (int)
SyntaxConstants.DOLLAR
:

EnterDollar((Token)
node);
            break;
        case (int)
SyntaxConstants.POWER:

EnterPower((Token)
node);
            break;
        case (int)
SyntaxConstants.HASH:

EnterHash((Token)
node);
            break;
        case (int)
SyntaxConstants.NEGA:

EnterNega((Token)
node);
            break;
        case (int)
SyntaxConstants.INT:

EnterInt((Token)
node);
            break;
        case (int)
SyntaxConstants.CHAR:

EnterChar((Token)
node);
            break;
        case (int)
SyntaxConstants.FLOAT:

EnterFloat((Token)
node);
            break;
        case (int)
SyntaxConstants.STRING
:

EnterString((Token)
node);
            break;
        case (int)
SyntaxConstants.BOOL_N
:

EnterBoolN((Token)
node);
            break;
        case (int)
SyntaxConstants.ID:

EnterId((Token) node);
            break;
        case (int)
SyntaxConstants.NUM:

EnterNum((Token)
node);
            break;
        case (int)
SyntaxConstants.DECIMA
L:

EnterDecimal((Token)
node);
            break;
        case (int)
SyntaxConstants.S_CHAR
:

EnterSChar((Token)
node);
            break;
        case (int)
SyntaxConstants.TEXT:

EnterText((Token)
node);
            break;
        case (int)
SyntaxConstants.COM:

EnterCom((Token)
node);
            break;
        case (int)
SyntaxConstants.YES:

EnterYes((Token)
node);
            break;
        case (int)
SyntaxConstants.NO:

EnterNo((Token) node);
            break;
        case (int)
SyntaxConstants.FUNCTN
AME:

EnterFunctname((Token)
node);
            break;
        case (int)
SyntaxConstants.STRUCT
NAME:

EnterStructname((Token
) node);
            break;
        case (int)
SyntaxConstants.IDSTRU
CT:

EnterIdstruct((Token)
node);
            break;
        case (int)
SyntaxConstants.F:

EnterF((Token) node);
            break;
        case (int)
SyntaxConstants.D:

EnterD((Token) node);
            break;
        case (int)
SyntaxConstants.S:

EnterS((Token) node);
            break;
        case (int)
SyntaxConstants.ZERO:

EnterZero((Token)
node);
            break;
        case (int)
SyntaxConstants.TOCHAR
:

EnterTochar((Token)
node);
            break;
        case (int)
SyntaxConstants.LENGTH
F:

EnterLengthf((Token)
node);
            break;
        case (int)
SyntaxConstants.CONTAI
NS:

EnterContains((Token)
node);
            break;
        case (int)
SyntaxConstants.REVERS
E:

EnterReverse((Token)
node);
            break;
        case (int)
SyntaxConstants.PROD_S
TART_PROGRAM:

EnterProdStartProgram(
(Production) node);
            break;
        case (int)
SyntaxConstants.PROD_P
ROGRAM:

EnterProdProgram((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_C
LEAR:

EnterProdClear((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_C
OMMENTS:

EnterProdComments((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_N
EGATE:

EnterProdNegate((Produ
ction) node);
            break;
        case (int)
SyntaxConstants.PROD_D
ATATYPE:

EnterProdDatatype((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_L
ITERALS:

EnterProdLiterals((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_L
ITERALS2:

EnterProdLiterals2((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_G
LOBAL_DEC:

EnterProdGlobalDec((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_D
ECLARE:

EnterProdDeclare((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_D
ECLARE_CHOICE:

EnterProdDeclareChoice
((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_I
NIT_CHOICE:

EnterProdInitChoice((P
roduction) node);
            break;
        case (int)
SyntaxConstants.PROD_A
DD_ID:

EnterProdAddId((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_N
1:

EnterProdN1((Productio
n) node);
            break;
        case (int)
SyntaxConstants.PROD_N
2:

EnterProdN2((Productio
n) node);
            break;
        case (int)
SyntaxConstants.PROD_I
NDEX:

EnterProdIndex((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_S
MATH:

EnterProdSmath((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_A
RRAY_AID:

EnterProdArrayAid((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_E
LEM_CHOICE:

EnterProdElemChoice((P
roduction) node);
            break;
        case (int)
SyntaxConstants.PROD_E
LEMENT:

EnterProdElement((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_A
DD_ELEM:
```

```
EnterProdAddElem((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_M
_ELEM:

EnterProdMElem((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_M
2_ELEM:

EnterProdM2Elem((Produ
ction) node);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTRET:

EnterProdFunctret((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_D
TYPE_A:

EnterProdDtypeA((Produ
ction) node);
            break;
        case (int)
SyntaxConstants.PROD_E
XDTYPE_A:

EnterProdExdtypeA((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_R
ETURN:

EnterProdReturn((Produ
ction) node);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTVOID:

EnterProdFunctvoid((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_S
TRUCT:

EnterProdStruct((Produ
ction) node);
            break;
        case (int)
SyntaxConstants.PROD_M
EM_DEC:

EnterProdMemDec((Produ
ction) node);
            break;
        case (int)
SyntaxConstants.PROD_I
NIT_DEC:

EnterProdInitDec((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_I
NIT_DEC_CHOICE:

EnterProdInitDecChoice
((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_C
ONSTANT:

EnterProdConstant((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_L
OCAL_CHOICE:

EnterProdLocalChoice((
Production) node);
            break;
        case (int)
SyntaxConstants.PROD_D
ECLARE1:

EnterProdDeclare1((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTRET1:

EnterProdFunctret1((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTVOID1:

EnterProdFunctvoid1((P
roduction) node);
            break;
        case (int)
SyntaxConstants.PROD_S
TRUCT1:

EnterProdStruct1((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_C
ONSTANT1:

EnterProdConstant1((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_M
AIN:

EnterProdMain((Product
ion) node);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_CHOICE:

EnterProdAssignChoice(
(Production) node);
            break;
        case (int)
SyntaxConstants.PROD_A
CCESS_ASSIGN_DTYPE:

EnterProdAccessAssignD
type((Production)
node);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_VALUE_CHOICE:

EnterProdAssignValueCh
oice((Production)
node);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGNING:

EnterProdAssigning((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_A
RRAY_ID:

EnterProdArrayId((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_A
RRAY_IDTAIL:

EnterProdArrayIdtail((
Production) node);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_SYM:

EnterProdAssignSym((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_VALUE:

EnterProdAssignValue((
Production) node);
            break;
        case (int)
SyntaxConstants.PROD_C
ONVERT:

EnterProdConvert((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCT_PARAM:

EnterProdFunctParam((P
roduction) node);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCT_IDPARAM:

EnterProdFunctIdparam(
(Production) node);
            break;
        case (int)
SyntaxConstants.PROD_A
DDFUNCT_IDPARAM:

EnterProdAddfunctIdpar
am((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_B
ODY:

EnterProdBody((Product
ion) node);
            break;
        case (int)
SyntaxConstants.PROD_P
RINT:

EnterProdPrint((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_P
OSTVAL:

EnterProdPostval((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_OUT:

EnterProdOut((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OUT_C:

EnterProdOutC((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_STRUCT_C:

EnterProdStructC((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_CONCAT_LIT:

EnterProdConcatLit((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_SCAN:

EnterProdScan((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_EXT_I:

EnterProdExtI((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_FOR_STATE:

EnterProdForState((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_FORSTATEMENT:

EnterProdForstatement((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_VAL1:

EnterProdVal1((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_MNT_COND:

EnterProdMntCond((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_MNT_COND_T:

EnterProdMntCondT((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_MNT:

EnterProdMnt((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_IFELSE:

EnterProdIfelse((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_IFCONDITION:

EnterProdIfcondition((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_IFSTATEMENT:

EnterProdIfstatement((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_ELSEIF:

EnterProdElseif((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_ELSEIFSTATEMENT:

EnterProdElseifstatement((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_ELSE_STATE:

EnterProdElseState((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_ELSESTATEMENT:

EnterProdElsestatement((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_DOWHILE:

EnterProdDowhile((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_DOSTATEMENT:

EnterProdDostatement((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_WHILE_STATE:

EnterProdWhileState((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_WHILESTATEMENT:

EnterProdWhilestatement((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_SWITCH_STATE:

EnterProdSwitchState((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_CASE_STATE:

EnterProdCaseState((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_DEF:

EnterProdDef((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_CASESTATEMENT:

EnterProdCasestatement((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_MATH_OP:

EnterProdMathOp((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OPER_COND:

EnterProdOperCond((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OPER_COND_CHOICE:

EnterProdOperCondChoice((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OPER_SYM:

EnterProdOperSym((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OPER_EQ:

EnterProdOperEq((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OPER_EXT_S:

EnterProdOperExtS((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OPER_EXT_REP:

EnterProdOperExtRep((Production) node);
            break;
        case (int)
SyntaxConstants.PROD_OPERAND:

EnterProdOperand((Production) node);
```

```
            break;
        case (int)
SyntaxConstants.PROD_S
IM_MATH_OP:

EnterProdSimMathOp((Pr
oduction) node);
            break;
        case (int)
SyntaxConstants.PROD_S
_MATH_EXT:

EnterProdSMathExt((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_O
PER_COND_EXT:

EnterProdOperCondExt((
Production) node);
            break;
        case (int)
SyntaxConstants.PROD_R
EL_OP:

EnterProdRelOp((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_R
ELOP_EXT:

EnterProdRelopExt((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_O
P1:

EnterProdOp1((Producti
on) node);
            break;
        case (int)
SyntaxConstants.PROD_L
OG_OP:

EnterProdLogOp((Produc
tion) node);
            break;
        case (int)
SyntaxConstants.PROD_E
XT_LOG_OP:

EnterProdExtLogOp((Pro
duction) node);
            break;
        case (int)
SyntaxConstants.PROD_L
OG_OPER:
```

```
EnterProdLogOper((Prod
uction) node);
            break;
        case (int)
SyntaxConstants.PROD_E
ND:

EnterProdEnd((Producti
on) node);
            break;
        }
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public override
Node Exit(Node node) {
        switch
(node.Id) {
        case (int)
SyntaxConstants.MAIN_N
:
            return
ExitMainN((Token)
node);
        case (int)
SyntaxConstants.PRINT_
N:
            return
ExitPrintN((Token)
node);
        case (int)
SyntaxConstants.SCAN_N
:
            return
ExitScanN((Token)
node);
        case (int)
SyntaxConstants.CONST_
N:
```

```
            return
ExitConstN((Token)
node);
        case (int)
SyntaxConstants.RETURN
:
            return
ExitReturn((Token)
node);
        case (int)
SyntaxConstants.SWITCH
_N:
            return
ExitSwitchN((Token)
node);
        case (int)
SyntaxConstants.CASE_N
:
            return
ExitCaseN((Token)
node);
        case (int)
SyntaxConstants.BREAK:
            return
ExitBreak((Token)
node);
        case (int)
SyntaxConstants.FOR_N:
            return
ExitForN((Token)
node);
        case (int)
SyntaxConstants.IF:
            return
ExitIf((Token) node);
        case (int)
SyntaxConstants.ELSEIF
_N:
            return
ExitElseifN((Token)
node);
        case (int)
SyntaxConstants.ELSE_N
:
            return
ExitElseN((Token)
node);
        case (int)
SyntaxConstants.DO:
            return
ExitDo((Token) node);
        case (int)
SyntaxConstants.WHILE_
N:
            return
ExitWhileN((Token)
node);
        case (int)
SyntaxConstants.VOID:
            return
ExitVoid((Token)
node);
```

```
        case (int)
SyntaxConstants.GETCH:
            return
ExitGetch((Token)
node);
        case (int)
SyntaxConstants.STRUCT
_N:
            return
ExitStructN((Token)
node);
        case (int)
SyntaxConstants.DEFAUL
T:
            return
ExitDefault((Token)
node);
        case (int)
SyntaxConstants.CLEAR:
            return
ExitClear((Token)
node);
        case (int)
SyntaxConstants.SQROOT
:
            return
ExitSqroot((Token)
node);
        case (int)
SyntaxConstants.PLUS:
            return
ExitPlus((Token)
node);
        case (int)
SyntaxConstants.MINUS:
            return
ExitMinus((Token)
node);
        case (int)
SyntaxConstants.TIMES:
            return
ExitTimes((Token)
node);
        case (int)
SyntaxConstants.DIVIDE
:
            return
ExitDivide((Token)
node);
        case (int)
SyntaxConstants.MODULU
S:
            return
ExitModulus((Token)
node);
        case (int)
SyntaxConstants.EQUALS
:
            return
ExitEquals((Token)
node);
```

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

```
                return ExitD((Token) node);
            case (int) SyntaxConstants.S:
                return ExitS((Token) node);
            case (int) SyntaxConstants.ZERO:
                return ExitZero((Token) node);
            case (int) SyntaxConstants.TOCHAR:
                return ExitTochar((Token) node);
            case (int) SyntaxConstants.LENGTHF:
                return ExitLengthf((Token) node);
            case (int) SyntaxConstants.CONTAINS:
                return ExitContains((Token) node);
            case (int) SyntaxConstants.REVERSE:
                return ExitReverse((Token) node);
            case (int) SyntaxConstants.PROD_START_PROGRAM:
                return ExitProdStartProgram((Production) node);
            case (int) SyntaxConstants.PROD_PROGRAM:
                return ExitProdProgram((Production) node);
            case (int) SyntaxConstants.PROD_CLEAR:
                return ExitProdClear((Production) node);
            case (int) SyntaxConstants.PROD_COMMENTS:
                return ExitProdComments((Production) node);
            case (int) SyntaxConstants.PROD_NEGATE:
                return ExitProdNegate((Production) node);
            case (int) SyntaxConstants.PROD_DATATYPE:
                return ExitProdDatatype((Production) node);
            case (int) SyntaxConstants.PROD_LITERALS:
                return ExitProdLiterals((Production) node);
            case (int) SyntaxConstants.PROD_LITERALS2:
                return ExitProdLiterals2((Production) node);
            case (int) SyntaxConstants.PROD_GLOBAL_DEC:
                return ExitProdGlobalDec((Production) node);
            case (int) SyntaxConstants.PROD_DECLARE:
                return ExitProdDeclare((Production) node);
            case (int) SyntaxConstants.PROD_DECLARE_CHOICE:
                return ExitProdDeclareChoice((Production) node);
            case (int) SyntaxConstants.PROD_INIT_CHOICE:
                return ExitProdInitChoice((Production) node);
            case (int) SyntaxConstants.PROD_ADD_ID:
                return ExitProdAddId((Production) node);
            case (int) SyntaxConstants.PROD_N1:
                return ExitProdN1((Production) node);
            case (int) SyntaxConstants.PROD_N2:
                return ExitProdN2((Production) node);
            case (int) SyntaxConstants.PROD_INDEX:
                return ExitProdIndex((Production) node);
            case (int) SyntaxConstants.PROD_SMATH:
                return ExitProdSmath((Production) node);
            case (int) SyntaxConstants.PROD_ARRAY_AID:
                return ExitProdArrayAid((Production) node);
            case (int) SyntaxConstants.PROD_ELEM_CHOICE:
                return ExitProdElemChoice((Production) node);
            case (int) SyntaxConstants.PROD_ELEMENT:
                return ExitProdElement((Production) node);
            case (int) SyntaxConstants.PROD_ADD_ELEM:
                return ExitProdAddElem((Production) node);
            case (int) SyntaxConstants.PROD_M_ELEM:
                return ExitProdMElem((Production) node);
            case (int) SyntaxConstants.PROD_M2_ELEM:
                return ExitProdM2Elem((Production) node);
            case (int) SyntaxConstants.PROD_FUNCTRET:
                return ExitProdFunctret((Production) node);
            case (int) SyntaxConstants.PROD_DTYPE_A:
                return ExitProdDtypeA((Production) node);
            case (int) SyntaxConstants.PROD_EXDTYPE_A:
                return ExitProdExdtypeA((Production) node);
            case (int) SyntaxConstants.PROD_RETURN:
                return ExitProdReturn((Production) node);
            case (int) SyntaxConstants.PROD_FUNCTVOID:
                return ExitProdFunctvoid((Production) node);
            case (int) SyntaxConstants.PROD_STRUCT:
                return ExitProdStruct((Production) node);
            case (int) SyntaxConstants.PROD_MEM_DEC:
                return ExitProdMemDec((Production) node);
            case (int) SyntaxConstants.PROD_INIT_DEC:
                return ExitProdInitDec((Production) node);
            case (int) SyntaxConstants.PROD_INIT_DEC_CHOICE:
                return ExitProdInitDecChoice((Production) node);
            case (int) SyntaxConstants.PROD_CONSTANT:
                return ExitProdConstant((Production) node);
            case (int) SyntaxConstants.PROD_LOCAL_CHOICE:
                return ExitProdLocalChoice((Production) node);
            case (int) SyntaxConstants.PROD_DECLARE1:
```

```
            return
ExitProdDeclare1((Prod
uction) node);
        case (int)
SyntaxConstants.PROD_F
UNCTRET1:
            return
ExitProdFunctret1((Pro
duction) node);
        case (int)
SyntaxConstants.PROD_F
UNCTVOID1:
            return
ExitProdFunctvoid1((Pr
oduction) node);
        case (int)
SyntaxConstants.PROD_S
TRUCT1:
            return
ExitProdStruct1((Produ
ction) node);
        case (int)
SyntaxConstants.PROD_C
ONSTANT1:
            return
ExitProdConstant1((Pro
duction) node);
        case (int)
SyntaxConstants.PROD_M
AIN:
            return
ExitProdMain((Producti
on) node);
        case (int)
SyntaxConstants.PROD_A
SSIGN_CHOICE:
            return
ExitProdAssignChoice((
Production) node);
        case (int)
SyntaxConstants.PROD_A
CCESS_ASSIGN_DTYPE:
            return
ExitProdAccessAssignDt
ype((Production)
node);
        case (int)
SyntaxConstants.PROD_A
SSIGN_VALUE_CHOICE:
            return
ExitProdAssignValueCho
ice((Production)
node);
        case (int)
SyntaxConstants.PROD_A
SSIGNING:
            return
ExitProdAssigning((Pro
duction) node);
        case (int)
SyntaxConstants.PROD_A
RRAY_ID:
            return
ExitProdArrayId((Produ
ction) node);
        case (int)
SyntaxConstants.PROD_A
RRAY_IDTAIL:
            return
ExitProdArrayIdtail((P
roduction) node);
        case (int)
SyntaxConstants.PROD_A
SSIGN_SYM:
            return
ExitProdAssignSym((Pro
duction) node);
        case (int)
SyntaxConstants.PROD_A
SSIGN_VALUE:
            return
ExitProdAssignValue((P
roduction) node);
        case (int)
SyntaxConstants.PROD_C
ONVERT:
            return
ExitProdConvert((Produ
ction) node);
        case (int)
SyntaxConstants.PROD_F
UNCT_PARAM:
            return
ExitProdFunctParam((Pr
oduction) node);
        case (int)
SyntaxConstants.PROD_F
UNCT_IDPARAM:
            return
ExitProdFunctIdparam((
Production) node);
        case (int)
SyntaxConstants.PROD_A
DDFUNCT_IDPARAM:
            return
ExitProdAddfunctIdpara
m((Production) node);
        case (int)
SyntaxConstants.PROD_B
ODY:
            return
ExitProdBody((Producti
on) node);
        case (int)
SyntaxConstants.PROD_P
RINT:
            return
ExitProdPrint((Product
ion) node);
        case (int)
SyntaxConstants.PROD_P
OSTVAL:
            return
ExitProdPostval((Produ
ction) node);
        case (int)
SyntaxConstants.PROD_O
UT:
            return
ExitProdOut((Productio
n) node);
        case (int)
SyntaxConstants.PROD_O
UT_C:
            return
ExitProdOutC((Producti
on) node);
        case (int)
SyntaxConstants.PROD_S
TRUCT_C:
            return
ExitProdStructC((Produ
ction) node);
        case (int)
SyntaxConstants.PROD_C
ONCAT_LIT:
            return
ExitProdConcatLit((Pro
duction) node);
        case (int)
SyntaxConstants.PROD_S
CAN:
            return
ExitProdScan((Producti
on) node);
        case (int)
SyntaxConstants.PROD_E
XT_I:
            return
ExitProdExtI((Producti
on) node);
        case (int)
SyntaxConstants.PROD_F
OR_STATE:
            return
ExitProdForState((Prod
uction) node);
        case (int)
SyntaxConstants.PROD_F
ORSTATEMENT:
            return
ExitProdForstatement((
Production) node);
        case (int)
SyntaxConstants.PROD_V
AL1:
            return
ExitProdVal1((Producti
on) node);
        case (int)
SyntaxConstants.PROD_M
NT_COND:
            return
ExitProdMntCond((Produ
ction) node);
        case (int)
SyntaxConstants.PROD_M
NT_COND_T:
            return
ExitProdMntCondT((Prod
uction) node);
        case (int)
SyntaxConstants.PROD_M
NT:
            return
ExitProdMnt((Productio
n) node);
        case (int)
SyntaxConstants.PROD_I
FELSE:
            return
ExitProdIfelse((Produc
tion) node);
        case (int)
SyntaxConstants.PROD_I
FCONDITION:
            return
ExitProdIfcondition((P
roduction) node);
        case (int)
SyntaxConstants.PROD_I
FSTATEMENT:
            return
ExitProdIfstatement((P
roduction) node);
        case (int)
SyntaxConstants.PROD_E
LSEIF:
            return
ExitProdElseif((Produc
tion) node);
        case (int)
SyntaxConstants.PROD_E
LSEIFSTATEMENT:
            return
ExitProdElseifstatemen
t((Production) node);
        case (int)
SyntaxConstants.PROD_E
LSE_STATE:
            return
ExitProdElseState((Pro
duction) node);
        case (int)
SyntaxConstants.PROD_E
LSESTATEMENT:
            return
ExitProdElsestatement(
(Production) node);
        case (int)
SyntaxConstants.PROD_D
OWHILE:
```

```
            return ExitProdDowhile((Production) node);
        case (int) SyntaxConstants.PROD_DOSTATEMENT:
            return ExitProdDostatement((Production) node);
        case (int) SyntaxConstants.PROD_WHILE_STATE:
            return ExitProdWhileState((Production) node);
        case (int) SyntaxConstants.PROD_WHILESTATEMENT:
            return ExitProdWhilestatement((Production) node);
        case (int) SyntaxConstants.PROD_SWITCH_STATE:
            return ExitProdSwitchState((Production) node);
        case (int) SyntaxConstants.PROD_CASE_STATE:
            return ExitProdCaseState((Production) node);
        case (int) SyntaxConstants.PROD_DEF:
            return ExitProdDef((Production) node);
        case (int) SyntaxConstants.PROD_CASESTATEMENT:
            return ExitProdCasestatement((Production) node);
        case (int) SyntaxConstants.PROD_MATH_OP:
            return ExitProdMathOp((Production) node);
        case (int) SyntaxConstants.PROD_OPER_COND:
            return ExitProdOperCond((Production) node);
        case (int) SyntaxConstants.PROD_OPER_COND_CHOICE:
            return ExitProdOperCondChoice((Production) node);
        case (int) SyntaxConstants.PROD_OPER_SYM:
            return ExitProdOperSym((Production) node);
        case (int) SyntaxConstants.PROD_OPER_EQ:
            return ExitProdOperEq((Production) node);
        case (int) SyntaxConstants.PROD_OPER_EXT_S:
            return ExitProdOperExtS((Production) node);
        case (int) SyntaxConstants.PROD_OPER_EXT_REP:
            return ExitProdOperExtRep((Production) node);
        case (int) SyntaxConstants.PROD_OPERAND:
            return ExitProdOperand((Production) node);
        case (int) SyntaxConstants.PROD_SIM_MATH_OP:
            return ExitProdSimMathOp((Production) node);
        case (int) SyntaxConstants.PROD_S_MATH_EXT:
            return ExitProdSMathExt((Production) node);
        case (int) SyntaxConstants.PROD_OPER_COND_EXT:
            return ExitProdOperCondExt((Production) node);
        case (int) SyntaxConstants.PROD_REL_OP:
            return ExitProdRelOp((Production) node);
        case (int) SyntaxConstants.PROD_RELOP_EXT:
            return ExitProdRelopExt((Production) node);
        case (int) SyntaxConstants.PROD_OP1:
            return ExitProdOp1((Production) node);
        case (int) SyntaxConstants.PROD_LOG_OP:
            return ExitProdLogOp((Production) node);
        case (int) SyntaxConstants.PROD_EXT_LOG_OP:
            return ExitProdExtLogOp((Production) node);
        case (int) SyntaxConstants.PROD_LOG_OPER:
            return ExitProdLogOper((Production) node);
        case (int) SyntaxConstants.PROD_END:
            return ExitProdEnd((Production) node);
        }
        return node;
    }

    /**
     * <summary>Called when adding a child to a parse tree
     * node.</summary>
     *
     * <param name=’node’>the parent node</param>
     * <param name=’child’>the child node, or null</param>
     *
     * <exception cref=’ParseException’>if the node analysis
     * discovered errors</exception>
     */
    public override void Child(Production node, Node child) {
        switch (node.Id) {
        case (int) SyntaxConstants.PROD_START_PROGRAM:
            ChildProdStartProgram(node, child);
            break;
        case (int) SyntaxConstants.PROD_PROGRAM:
            ChildProdProgram(node, child);
            break;
        case (int) SyntaxConstants.PROD_CLEAR:
            ChildProdClear(node, child);
            break;
        case (int) SyntaxConstants.PROD_COMMENTS:
            ChildProdComments(node, child);
            break;
        case (int) SyntaxConstants.PROD_NEGATE:
            ChildProdNegate(node, child);
            break;
        case (int) SyntaxConstants.PROD_DATATYPE:
            ChildProdDatatype(node, child);
            break;
        case (int) SyntaxConstants.PROD_LITERALS:
            ChildProdLiterals(node, child);
            break;
        case (int) SyntaxConstants.PROD_LITERALS2:
            ChildProdLiterals2(node, child);
            break;
        case (int) SyntaxConstants.PROD_GLOBAL_DEC:
            ChildProdGlobalDec(node, child);
            break;
        case (int)
SyntaxConstants.PROD_D
ECLARE:

ChildProdDeclare(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_D
ECLARE_CHOICE:

ChildProdDeclareChoice
(node, child);
            break;
        case (int)
SyntaxConstants.PROD_I
NIT_CHOICE:

ChildProdInitChoice(no
de, child);
            break;
        case (int)
SyntaxConstants.PROD_A
DD_ID:

ChildProdAddId(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_N
1:

ChildProdN1(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_N
2:

ChildProdN2(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_I
NDEX:

ChildProdIndex(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_S
MATH:

ChildProdSmath(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_A
RRAY_AID:

ChildProdArrayAid(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_E
LEM_CHOICE:

ChildProdElemChoice(no
de, child);
            break;
        case (int)
SyntaxConstants.PROD_E
LEMENT:

ChildProdElement(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_A
DD_ELEM:

ChildProdAddElem(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_M
_ELEM:

ChildProdMElem(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_M
2_ELEM:

ChildProdM2Elem(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTRET:

ChildProdFunctret(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_D
TYPE_A:

ChildProdDtypeA(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_E
XDTYPE_A:

ChildProdExdtypeA(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_R
ETURN:

ChildProdReturn(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTVOID:

ChildProdFunctvoid(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_S
TRUCT:

ChildProdStruct(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_M
EM_DEC:

ChildProdMemDec(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_I
NIT_DEC:

ChildProdInitDec(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_I
NIT_DEC_CHOICE:

ChildProdInitDecChoice
(node, child);
            break;
        case (int)
SyntaxConstants.PROD_C
ONSTANT:

ChildProdConstant(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_L
OCAL_CHOICE:

ChildProdLocalChoice(n
ode, child);
            break;
        case (int)
SyntaxConstants.PROD_D
ECLARE1:

ChildProdDeclare1(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTRET1:

ChildProdFunctret1(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCTVOID1:

ChildProdFunctvoid1(no
de, child);
            break;
        case (int)
SyntaxConstants.PROD_S
TRUCT1:

ChildProdStruct1(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_C
ONSTANT1:

ChildProdConstant1(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_M
AIN:

ChildProdMain(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_CHOICE:

ChildProdAssignChoice(
node, child);
            break;
        case (int)
SyntaxConstants.PROD_A
CCESS_ASSIGN_DTYPE:

ChildProdAccessAssignD
type(node, child);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_VALUE_CHOICE:

ChildProdAssignValueCh
oice(node, child);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGNING:
```

```
ChildProdAssigning(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_A
RRAY_ID:

ChildProdArrayId(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_A
RRAY_IDTAIL:

ChildProdArrayIdtail(n
ode, child);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_SYM:

ChildProdAssignSym(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_A
SSIGN_VALUE:

ChildProdAssignValue(n
ode, child);
            break;
        case (int)
SyntaxConstants.PROD_C
ONVERT:

ChildProdConvert(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCT_PARAM:

ChildProdFunctParam(no
de, child);
            break;
        case (int)
SyntaxConstants.PROD_F
UNCT_IDPARAM:

ChildProdFunctIdparam(
node, child);
            break;
        case (int)
SyntaxConstants.PROD_A
DDFUNCT_IDPARAM:

ChildProdAddfunctIdpar
am(node, child);
            break;
        case (int)
SyntaxConstants.PROD_B
ODY:

ChildProdBody(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_P
RINT:

ChildProdPrint(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_P
OSTVAL:

ChildProdPostval(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_O
UT:

ChildProdOut(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_O
UT_C:

ChildProdOutC(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_S
TRUCT_C:

ChildProdStructC(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_C
ONCAT_LIT:

ChildProdConcatLit(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_S
CAN:

ChildProdScan(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_E
XT_I:

ChildProdExtI(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_F
OR_STATE:

ChildProdForState(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_F
ORSTATEMENT:

ChildProdForstatement(
node, child);
            break;
        case (int)
SyntaxConstants.PROD_V
AL1:

ChildProdVal1(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_M
NT_COND:

ChildProdMntCond(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_M
NT_COND_T:

ChildProdMntCondT(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_M
NT:

ChildProdMnt(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_I
FELSE:

ChildProdIfelse(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_I
FCONDITION:

ChildProdIfcondition(n
ode, child);
            break;
        case (int)
SyntaxConstants.PROD_I
FSTATEMENT:

ChildProdIfstatement(n
ode, child);
            break;
        case (int)
SyntaxConstants.PROD_E
LSEIF:

ChildProdElseif(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_E
LSEIFSTATEMENT:

ChildProdElseifstateme
nt(node, child);
            break;
        case (int)
SyntaxConstants.PROD_E
LSE_STATE:

ChildProdElseState(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_E
LSESTATEMENT:

ChildProdElsestatement
(node, child);
            break;
        case (int)
SyntaxConstants.PROD_D
OWHILE:

ChildProdDowhile(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_D
OSTATEMENT:

ChildProdDostatement(n
ode, child);
            break;
        case (int)
SyntaxConstants.PROD_W
HILE_STATE:

ChildProdWhileState(no
de, child);
            break;
        case (int)
SyntaxConstants.PROD_W
HILESTATEMENT:

ChildProdWhilestatemen
t(node, child);
            break;
        case (int)
SyntaxConstants.PROD_S
WITCH_STATE:

ChildProdSwitchState(n
ode, child);
            break;
        case (int)
SyntaxConstants.PROD_C
ASE_STATE:

ChildProdCaseState(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_D
EF:

ChildProdDef(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_C
ASESTATEMENT:

ChildProdCasestatement
(node, child);
            break;
        case (int)
SyntaxConstants.PROD_M
ATH_OP:

ChildProdMathOp(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_O
PER_COND:

ChildProdOperCond(node
, child);
            break;
        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:

ChildProdOperEq(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_O
PER_EXT_S:

ChildProdOperExtS(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_O
PER_EXT_REP:

ChildProdOperExtRep(no
de, child);
            break;
        case (int)
SyntaxConstants.PROD_O
PERAND:

ChildProdOperand(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_S
IM_MATH_OP:

ChildProdSimMathOp(nod
e, child);
            break;
        case (int)
SyntaxConstants.PROD_S
_MATH_EXT:

ChildProdSMathExt(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_O
PER_COND_EXT:

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;
        case (int)
SyntaxConstants.PROD_O
P1:

ChildProdOp1(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_L
OG_OP:

ChildProdLogOp(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_E
XT_LOG_OP:

ChildProdExtLogOp(node
, child);
            break;
        case (int)
SyntaxConstants.PROD_L
OG_OPER:

ChildProdLogOper(node,
child);
            break;
        case (int)
SyntaxConstants.PROD_E
ND:

ChildProdEnd(node,
child);
            break;
        }
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <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>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitMainN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterPrintN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitPrintN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterScanN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitScanN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterConstN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitConstN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterReturn(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitReturn(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterSwitchN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitSwitchN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterCaseN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitCaseN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterBreak(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitBreak(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterForN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitForN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterIf(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitIf(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterElseifN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitElseifN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterElseN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitElseN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterDo(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDo(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterWhileN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitWhileN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterVoid(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitVoid(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterGetch(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitGetch(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterStructN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitStructN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterDefault(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDefault(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterClear(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitClear(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterSqroot(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitSqroot(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterPlus(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitPlus(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterMinus(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitMinus(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterTimes(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitTimes(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterDivide(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDivide(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterModulus(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitModulus(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterEquals(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitEquals(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterSemic(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitSemic(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterDot(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDot(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterComma(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitComma(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterAnd(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitAnd(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterOr(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitOr(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterNot(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitNot(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterIncrement(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitIncrement(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterDecrement(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitDecrement(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterPE(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitPE(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterME(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitME(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterTE(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
```

```
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitTE(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterDE(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDE(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterModE(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitModE(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterNewline(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitNewline(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterNE(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitNE(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterOParen(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
```

```
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitOParen(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterCParen(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitCParen(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterDQuote(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDQuote(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterColon(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitColon(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterOBracket(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitOBracket(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterCBracket(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
```

```
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitCBracket(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterGreater(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitGreater(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterLess(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitLess(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterGreaterE(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitGreaterE(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterLessE(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitLessE(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterSObracket(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitSObracket(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterSCbracket(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitSCbracket(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterDollar(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDollar(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterPower(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitPower(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterHash(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitHash(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterNega(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitNega(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterInt(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitInt(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterChar(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitChar(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterFloat(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitFloat(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterString(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitString(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterBoolN(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitBoolN(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterId(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitId(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterNum(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitNum(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterDecimal(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitDecimal(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterSChar(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitSChar(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterText(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitText(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterCom(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitCom(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterYes(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitYes(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterNo(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitNo(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterFunctname(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitFunctname(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterStructname(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
```

```
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitStructname(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterIdstruct(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitIdstruct(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterF(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitF(Token node)
{
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterD(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitD(Token node)
{
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterS(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitS(Token node)
{
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterZero(Token
node) {
    }

    /**
```

```
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitZero(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void EnterTochar(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitTochar(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterLengthf(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitLengthf(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterContains(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitContains(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterReverse(Token
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node ExitReverse(Token
node) {
        return node;
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdStartProgram(
Production node) {
    }
    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdStartProgram(P
roduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdStartProgram(
Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdProgram(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdProgram(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdProgram(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdClear(Product
ion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdClear(Producti
on node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdClear(Product
ion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdComments(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
```

```
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdComments(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdComments(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdNegate(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdNegate(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdNegate(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDatatype(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdDatatype(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDatatype(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdLiterals(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdLiterals(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdLiterals(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdLiterals2(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdLiterals2(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdLiterals2(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdGlobalDec(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdGlobalDec(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdGlobalDec(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDeclare(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
```

```
ExitProdDeclare(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDeclare(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDeclareChoice
(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdDeclareChoice(
Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDeclareChoice
(Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdInitChoice(Pr
oduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdInitChoice(Pro
duction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdInitChoice(Pr
oduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAddId(Product
ion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAddId(Producti
on node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAddId(Product
ion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdN1(Production
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdN1(Production
node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdN1(Production
node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdN2(Production
node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdN2(Production
node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdN2(Production
node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdIndex(Product
ion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdIndex(Producti
on node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
```

```
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdIndex(Product
ion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdSmath(Product
ion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdSmath(Producti
on node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdSmath(Product
ion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdArrayAid(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdArrayAid(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdArrayAid(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdElemChoice(Pr
oduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdElemChoice(Pro
duction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdElemChoice(Pr
oduction node, Node
child) {

node.AddChild(child);
```

```
    }

    /**
     * <summary>Called when entering a parse tree node.</summary>
     *
     * <param name='node'>the node being entered</param>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual void EnterProdElement(Production node) {
    }

    /**
     * <summary>Called when exiting a parse tree node.</summary>
     *
     * <param name='node'>the node being exited</param>
     *
     * <returns>the node to add to the parse tree, or
     *          null if no parse tree should be created</returns>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual Node ExitProdElement(Production node) {
        return node;
    }

    /**
     * <summary>Called when adding a child to a parse tree
     * node.</summary>
     *
     * <param name='node'>the parent node</param>
     * <param name='child'>the child node, or null</param>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual void ChildProdElement(Production node, Node child) {
        node.AddChild(child);
    }

    /**
     * <summary>Called when entering a parse tree node.</summary>
     *
     * <param name='node'>the node being entered</param>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual void EnterProdAddElem(Production node) {
    }

    /**
     * <summary>Called when exiting a parse tree node.</summary>
     *
     * <param name='node'>the node being exited</param>
     *
     * <returns>the node to add to the parse tree, or
     *          null if no parse tree should be created</returns>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual Node ExitProdAddElem(Production node) {
        return node;
    }

    /**
     * <summary>Called when adding a child to a parse tree
     * node.</summary>
     *
     * <param name='node'>the parent node</param>
     * <param name='child'>the child node, or null</param>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual void ChildProdAddElem(Production node, Node child) {
        node.AddChild(child);
    }

    /**
     * <summary>Called when entering a parse tree node.</summary>
     *
     * <param name='node'>the node being entered</param>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual void EnterProdMElem(Production node) {
    }

    /**
     * <summary>Called when exiting a parse tree node.</summary>
     *
     * <param name='node'>the node being exited</param>
     *
     * <returns>the node to add to the parse tree, or
     *          null if no parse tree should be created</returns>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual Node ExitProdMElem(Production node) {
        return node;
    }

    /**
     * <summary>Called when adding a child to a parse tree
     * node.</summary>
     *
     * <param name='node'>the parent node</param>
     * <param name='child'>the child node, or null</param>
     *
     * <exception cref='ParseException'>if the node analysis
     * discovered errors</exception>
     */
    public virtual void ChildProdMElem(Production node, Node child) {
        node.AddChild(child);
    }

    /**
     * <summary>Called when entering a parse tree node.</summary>
     *
     * <param name='node'>the node being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdM2Elem(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdM2Elem(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdM2Elem(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdFunctret(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdFunctret(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdFunctret(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDtypeA(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdDtypeA(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDtypeA(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
```

```
EnterProdExdtypeA(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdExdtypeA(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdExdtypeA(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdReturn(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdReturn(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdReturn(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdFunctvoid(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdFunctvoid(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdFunctvoid(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdStruct(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdStruct(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdStruct(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdMemDec(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdMemDec(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdMemDec(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdInitDec(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdInitDec(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdInitDec(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdInitDecChoice
(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
```

```
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdInitDecChoice(
Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdInitDecChoice
(Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdConstant(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdConstant(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdConstant(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdLocalChoice(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdLocalChoice(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdLocalChoice(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDeclare1(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
```

```
     */
    public virtual
Node
ExitProdDeclare1(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDeclare1(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdFunctret1(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdFunctret1(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdFunctret1(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdFunctvoid1(Pr
oduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdFunctvoid1(Pro
duction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdFunctvoid1(Pr
oduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdStruct1(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdStruct1(Produc
tion node) {
        return node;
    }

    /**
```

```
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdStruct1(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdConstant1(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdConstant1(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdConstant1(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdMain(Producti
on node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdMain(Productio
n node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdMain(Producti
on node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAssignChoice(
Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAssignChoice(P
roduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
```

```
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAssignChoice(
Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAccessAssignD
type(Production node)
{
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAccessAssignDt
ype(Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAccessAssignD
type(Production node,
Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAssignValueCh
oice(Production node)
{
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAssignValueCho
ice(Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAssignValueCh
oice(Production node,
Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAssigning(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAssigning(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAssigning(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdArrayId(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdArrayId(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdArrayId(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdArrayIdtail(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdArrayIdtail(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdArrayIdtail(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAssignSym(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAssignSym(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAssignSym(Pro
duction node, Node
child) {

node.AddChild(child);
    }
    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAssignValue(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAssignValue(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAssignValue(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdConvert(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdConvert(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdConvert(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdFunctParam(Pr
oduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdFunctParam(Pro
duction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdFunctParam(Pr
oduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdFunctIdparam(
Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdFunctIdparam(P
roduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdFunctIdparam(
Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdAddfunctIdpar
am(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdAddfunctIdpara
m(Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdAddfunctIdpar
am(Production node,
Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdBody(Producti
on node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdBody(Productio
n node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdBody(Producti
on node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdPrint(Product
ion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdPrint(Producti
on node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdPrint(Product
ion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdPostval(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdPostval(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdPostval(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOut(Productio
n node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOut(Production
node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOut(Productio
n node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOutC(Producti
on node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOutC(Productio
n node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOutC(Producti
on node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdStructC(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdStructC(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdStructC(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdConcatLit(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdConcatLit(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdConcatLit(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdScan(Producti
on node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdScan(Productio
n node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdScan(Producti
on node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdExtI(Producti
on node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdExtI(Productio
n node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdExtI(Producti
on node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdForState(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdForState(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdForState(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdForstatement(
Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdForstatement(P
roduction node) {
        return node;
    }
    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdForstatement(
Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdVal1(Producti
on node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdVal1(Productio
n node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdVal1(Producti
on node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdMntCond(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdMntCond(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdMntCond(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdMntCondT(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdMntCondT(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
```

```
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdMntCondT(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdMnt(Productio
n node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdMnt(Production
node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdMnt(Productio
n node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdIfelse(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdIfelse(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdIfelse(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdIfcondition(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdIfcondition(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdIfcondition(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdIfstatement(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdIfstatement(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdIfstatement(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdElseif(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdElseif(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdElseif(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdElseifstateme
nt(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdElseifstatemen
t(Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdElseifstateme
nt(Production node,
Node child) {

node.AddChild(child);
    }
    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdElseState(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdElseState(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdElseState(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdElsestatement
(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdElsestatement(
Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdElsestatement
(Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDowhile(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdDowhile(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDowhile(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDostatement(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdDostatement(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDostatement(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdWhileState(Pr
oduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdWhileState(Pro
duction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdWhileState(Pr
oduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdWhilestatemen
t(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdWhilestatement
(Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdWhilestatemen
t(Production node,
Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
```

```
EnterProdSwitchState(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdSwitchState(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdSwitchState(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdCaseState(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdCaseState(Prod
uction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdCaseState(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdDef(Productio
n node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdDef(Production
node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdDef(Productio
n node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdCasestatement
(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdCasestatement(
Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdCasestatement
(Production node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdMathOp(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdMathOp(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdMathOp(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperCond(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOperCond(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperCond(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperCondChoic
e(Production node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
```

```
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOperCondChoice
(Production node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperCondChoic
e(Production node,
Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperSym(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOperSym(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperSym(Produ
ction node, Node
child) {

node.AddChild(child);
    }
    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperEq(Produc
tion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOperEq(Product
ion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperEq(Produc
tion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperExtS(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
```

```
     */
    public virtual
Node
ExitProdOperExtS(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperExtS(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperExtRep(Pr
oduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOperExtRep(Pro
duction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperExtRep(Pr
oduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperand(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOperand(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperand(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdSimMathOp(Pro
duction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdSimMathOp(Prod
uction node) {
        return node;
    }

    /**
```

```
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdSimMathOp(Pro
duction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdSMathExt(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdSMathExt(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdSMathExt(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOperCondExt(P
roduction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOperCondExt(Pr
oduction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOperCondExt(P
roduction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdRelOp(Product
ion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdRelOp(Producti
on node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
```

```
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdRelOp(Product
ion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdRelopExt(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdRelopExt(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <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</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdOp1(Productio
n node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdOp1(Production
node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdOp1(Productio
n node, Node child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdLogOp(Product
ion node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *          null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdLogOp(Producti
on node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdLogOp(Product
ion node, Node child)
{

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdExtLogOp(Prod
uction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdExtLogOp(Produ
ction node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdExtLogOp(Prod
uction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdLogOper(Produ
ction node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdLogOper(Produc
tion node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
ChildProdLogOper(Produ
ction node, Node
child) {

node.AddChild(child);
    }

    /**
     * <summary>Called
when entering a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being entered</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
void
EnterProdEnd(Productio
n node) {
    }

    /**
     * <summary>Called
when exiting a parse
tree node.</summary>
     *
     * <param
name='node'>the node
being exited</param>
     *
     * <returns>the
node to add to the
parse tree, or
     *         null
if no parse tree
should be
created</returns>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */
    public virtual
Node
ExitProdEnd(Production
node) {
        return node;
    }

    /**
     * <summary>Called
when adding a child to
a parse tree
     * node.</summary>
     *
     * <param
name='node'>the parent
node</param>
     * <param
name='child'>the child
node, or null</param>
     *
     * <exception
cref='ParseException'>
if the node analysis
     * discovered
errors</exception>
     */

public virtual void
ChildProdEnd(Productio
n node, Node child) {

node.AddChild(child);
    }
}
```

Syntax Analyzer: SyntaxConstants.cs

```
public enum SyntaxConstants {
    MAIN_N = 1001,
    PRINT_N = 1002,
    SCAN_N = 1003,
    CONST_N = 1004,
    RETURN = 1005,
    SWITCH_N = 1006,
    CASE_N = 1007,
    BREAK = 1008,
    FOR_N = 1009,
    IF = 1010,
    ELSEIF_N = 1011,
    ELSE_N = 1012,
    DO = 1013,
    WHILE_N = 1014,
    VOID = 1015,
    GETCH = 1016,
    STRUCT_N = 1017,
    DEFAULT = 1018,
    CLEAR = 1019,
    SQROOT = 1020,
    PLUS = 1021,
    MINUS = 1022,
    TIMES = 1023,
    DIVIDE = 1024,
    MODULUS = 1025,
    EQUALS = 1026,
    SEMIC = 1027,
    DOT = 1028,
    COMMA = 1029,
    AND = 1030,
    OR = 1031,
    NOT = 1032,
    INCREMENT = 1033,
    DECREMENT = 1034,
    P_E = 1035,
    M_E = 1036,
    T_E = 1037,
    D_E = 1038,
    MOD_E = 1039,
    NEWLINE = 1040,
    N_E = 1041,
    O_PAREN = 1042,
    C_PAREN = 1043,
    D_QUOTE = 1044,
    COLON = 1045,
    O_BRACKET = 1046,
    C_BRACKET = 1047,
    GREATER = 1048,
    LESS = 1049,
    GREATER_E = 1050,
    LESS_E = 1051,
    S_OBRACKET = 1052,
    S_CBRACKET = 1053,
    DOLLAR = 1054,
    POWER = 1055,
    HASH = 1056,
    NEGA = 1057,
    INT = 1058,
    CHAR = 1059,
    FLOAT = 1060,
    STRING = 1061,
    BOOL_N = 1062,
    ID = 1063,
    NUM = 1064,
    DECIMAL = 1065,
    S_CHAR = 1066,
    TEXT = 1067,
    COM = 1068,
    YES = 1069,
    NO = 1070,
    FUNCTNAME = 1071,
    STRUCTNAME = 1072,
    IDSTRUCT = 1073,
    F = 1074,
    D = 1075,
    S = 1076,
    ZERO = 1077,
    SPACE = 1078,
    N_LINE = 1079,
    WHITESPACE = 1080,
    TOCHAR = 1081,
    LENGTHF = 1082,
    CONTAINS = 1083,
    REVERSE = 1084,
    PROD_START_PROGRAM = 2001,
    PROD_PROGRAM = 2002,
    PROD_CLEAR = 2003,
    PROD_COMMENTS = 2004,
    PROD_NEGATE = 2005,
    PROD_DATATYPE = 2006,
    PROD_LITERALS = 2007,
    PROD_LITERALS2 = 2008,
    PROD_GLOBAL_DEC = 2009,
    PROD_DECLARE = 2010,
    PROD_DECLARE_CHOICE = 2011,
    PROD_INIT_CHOICE = 2012,
    PROD_ADD_ID = 2013,
    PROD_N1 = 2014,
    PROD_N2 = 2015,
    PROD_INDEX = 2016,
    PROD_SMATH = 2017,
    PROD_ARRAY_AID = 2018,
    PROD_ELEM_CHOICE = 2019,
    PROD_ELEMENT = 2020,
    PROD_ADD_ELEM = 2021,
    PROD_M_ELEM = 2022,
    PROD_M2_ELEM = 2023,
    PROD_FUNCTRET = 2024,
    PROD_DTYPE_A = 2025,
    PROD_EXDTYPE_A = 2026,
    PROD_RETURN = 2027,
    PROD_FUNCTVOID = 2028,
    PROD_STRUCT = 2029,
    PROD_MEM_DEC = 2030,
    PROD_INIT_DEC = 2031,
    PROD_INIT_DEC_CHOICE = 2032,
    PROD_CONSTANT = 2033,
    PROD_LOCAL_CHOICE = 2034,
    PROD_DECLARE1 = 2035,
    PROD_FUNCTRET1 = 2036,
    PROD_FUNCTVOID1 = 2037,
    PROD_STRUCT1 = 2038,
    PROD_CONSTANT1 = 2039,
    PROD_MAIN = 2040,
    PROD_ASSIGN_CHOICE = 2041,
    PROD_ACCESS_ASSIGN_DTYPE = 2042,
    PROD_ASSIGN_VALUE_CHOICE = 2043,
    PROD_ASSIGNING = 2044,
    PROD_ARRAY_ID = 2045,
    PROD_ARRAY_IDTAIL = 2046,
    PROD_ASSIGN_SYM = 2047,
    PROD_ASSIGN_VALUE = 2048,
    PROD_CONVERT = 2049,
    PROD_FUNCT_PARAM = 2050,
    PROD_FUNCT_IDPARAM = 2051,
    PROD_ADDFUNCT_IDPARAM = 2052,
    PROD_BODY = 2053,
    PROD_PRINT = 2054,
    PROD_POSTVAL = 2055,
    PROD_OUT = 2056,
    PROD_OUT_C = 2057,
    PROD_STRUCT_C = 2058,
    PROD_CONCAT_LIT = 2059,
    PROD_SCAN = 2060,
    PROD_EXT_I = 2061,
    PROD_FOR_STATE = 2062,
    PROD_FORSTATEMENT = 2063,
    PROD_VAL1 = 2064,
    PROD_MNT_COND = 2065,
    PROD_MNT_COND_T = 2066,
    PROD_MNT = 2067,
    PROD_IFELSE = 2068,
    PROD_IFCONDITION = 2069,
    PROD_IFSTATEMENT = 2070,
    PROD_ELSEIF = 2071,
    PROD_ELSEIFSTATEMENT = 2072,
    PROD_ELSE_STATE = 2073,
    PROD_ELSESTATEMENT = 2074,
    PROD_DOWHILE = 2075,
    PROD_DOSTATEMENT = 2076,
    PROD_WHILE_STATE = 2077,
    PROD_WHILESTATEMENT = 2078,
    PROD_SWITCH_STATE = 2079,
    PROD_CASE_STATE = 2080,
    PROD_DEF = 2081,
    PROD_CASESTATEMENT = 2082,
    PROD_MATH_OP = 2083,
    PROD_OPER_COND =
2084,

PROD_OPER_COND_CHOICE
= 2085,
    PROD_OPER_SYM =
2086,
    PROD_OPER_EQ =
2087,
    PROD_OPER_EXT_S =
2088,
    PROD_OPER_EXT_REP
= 2089,
    PROD_OPERAND =
2090,
    PROD_SIM_MATH_OP =
2091,
    PROD_S_MATH_EXT =
2092,
    PROD_OPER_COND_EXT
= 2093,
    PROD_REL_OP =
2094,
    PROD_RELOP_EXT =
2095,
    PROD_OP1 = 2096,
    PROD_LOG_OP =
2097,
    PROD_EXT_LOG_OP =
2098,
    PROD_LOG_OPER =
2099,
    PROD_END = 2100
}
```

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
currparent = null;
        List<Node>
prevparent = new
List<Node>();
        List<Node>
productions = new
List<Node>();
        public
List<string> SET = new
List<string>();
        public
List<string>
PRODUCTION = new
List<string>();

        public
override void
Enter(Node node)
        {
            string
name = node.GetName();
            if
(name.Contains("Prod_"
))
            {

node.SetParent(currpar
ent);
                name =
name.Substring(5);

                if
(currparent != null)
                {

production += "Enter:
<" + name + "> Parent:
" +
currparent.GetName() +
"\n";

productions.Add(node);
                }
                else
                {

production += "Enter:
<" + name + ">\n";

productions.Add(node);
                }

prevparent.Add(currpar
ent);

currparent = node;
            }
            else
            {

node.SetParent(currpar
ent);
productions.Add(node);

production += "Enter:
" + name + " Parent: "
+ currparent.GetName()
+ "\n";
            }

        }
        public
override Node
Exit(Node node)
        {
            if
(currparent == node)
            {

currparent =
prevparent[prevparent.
Count - 1];

prevparent.RemoveAt(pr
evparent.Count - 1);
            }
            return
node;
        }

        public
override Node
Analyze(Node node)
        {
            return
base.Analyze(node);
        }

        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
result;
            int line =
1;
            int
linejump = 0;
            foreach
(var t in tokens)
            {
                if
(t.getLines() != line)
                {

linejump =
t.getLines() - line;

line = t.getLines();

for (int i = 0; i <
linejump; i++)
                    {

tokenstream += "\n";
                    }
                }

tokenstream +=
t.getTokens() + " ";
            }

tokenstream =
tokenstream.TrimEnd();

            Parser p;
            p =
CreateParser(tokenstre
am);

            try
            {
                Node
parse = p.Parse();

Fail("parsing
succeeded");
                result
= "Syntax Analyzer
Succeeded...";
            }
            catch
(ParserCreationExcepti
on e)
            {

Fail(e.Message);
                result
= e.Message;
            }
            catch
(ParserLogException e)
            {


List<int> codes =
p.GetAllProductionCode
();

PredictSets ps = new
PredictSets();
                string
message = "Expected:
";

errors.setColumn(e.Get
Error(0).Column);

errors.setLines(e.GetE
rror(0).Line);
                int
ctr =
GetSyntaxTable(codes);

//isDone = true;

                if
(codes.Count - 1 >=
ctr)
                {

int code = codes[ctr];

message +=
ps.GetPredictSet(code)
;
                }
                else
                {

int code = codes[ctr-
1];

message +=
ps.GetPredictSet(code)
;
                }
                //if
(p.GetLastProductionSt
ate() == "NULL")
                //{
                //
int code =
p.GetLastProductionCod
e();
                //
message +=
ps.GetPredictSet(code)
;
                //}
                //else
                //{
                //
foreach (var item in
e.GetError(0).Details)
                //
{
```

```
                //
message += item + ",
";
                //
}
                //}
                if
(message == "Expected:
")
                {

string errormessage =
e.GetError(0).ErrorMes
sage;
                    if
(errormessage.Contains
("unexpected token"))
                    {

errormessage = "";

foreach (var item in
e.GetError(0).Details)

{

errormessage += item +
", ";

}
                    }
                    if
(errormessage ==
"unexpected end of
file")

errormessage =
"\".\"";


message +=
errormessage;
                }

                //if
(message == "Expected:
@, (, &&, ||, >=, <=,
<, >, ==, !=, )")
                //{
                //
message = "Expected:
";
                //
foreach (var item in
e.GetError(0).Details)
                //
{
                //
message += item + ",
";
                //
}
```

```
                //}

message += ".";


errors.setErrorMessage
(message);

errors.setType(e.GetEr
ror(0).Type.ToString()
);
                result
= e.Message;

            }

recursiveprod =
p.GetRecursiveProducti
on();

GetSyntaxTable(p.GetAl
lProductionCode());
            return
result;
        }

        private int
GetSyntaxTable(List<in
t> code)
        {
            Node node
= null;
            Boolean
delete = true;
            string
recprod =
recursiveprod;
            int ctr =
-1, count = 1,
prodcode = 0;
            string
currentparent = "";
            while
(productions.Count !=
0)
            {
                ctr++;

prodcode = code[ctr];
                node =
productions[count];


                if
(node.GetId() ==
prodcode)
                {


string nodename =
```

```
node.GetName().ToLower
();

currentparent =
node.GetParent().GetNa
me();

currentparent =
currentparent.ToLower(
);

delete = true;

                    if
(currentparent.Contain
s("prod_"))
                    {

currentparent = "<" +
currentparent.Substrin
g(5) + ">";
                    }
                    if
(nodename.Contains("pr
od_"))
                    {

nodename = "<" +
nodename.Substring(5)
+ ">";
                    }


PRODUCTION.Add(current
parent);

SET.Add(nodename);
                }
                else
                {


string name =
Enum.GetName(typeof(Sy
ntaxConstants),
prodcode);

name = name.ToLower();
                    if
(name.Contains("prod_"
))
                    {

name = "<" +
name.Substring(5) +
">";
                    }


                    if
(PRODUCTION.Count !=
1)
                    {
```

```
currentparent.ToLower(
);

if
(currentparent.Contain
s("prod_"))

{

currentparent = "<" +
currentparent.Substrin
g(5) + ">";

}

PRODUCTION.Add(current
parent);

SET.Add(name);

PRODUCTION.Add(name);

SET.Add("λ");

delete = false;
                   }

else
                   {

PRODUCTION.Add("<progr
am>");

SET.Add(name);

PRODUCTION.Add(name);

SET.Add("λ");

delete = false;
                   }
               }

               if
(count != 1 && delete)
               {

productions.RemoveAt(0
);
               }
               else
if (delete)
               {

productions.RemoveAt(0
);

productions.RemoveAt(0
);

count = 0;
```

```
                  }


              }
              return
(ctr + 1);
          }

          private Parser
CreateParser(string
input)
          {
              Parser
parser = null;
              try
              {
                  parser
= new SyntaxParser(new
StringReader(input),
this);

parser.Prepare();


              }
              catch
(ParserCreationExcepti
on e)
              {

Fail(e.Message);
              }
              return
parser;
          }

          protected void
Fail(string message)
          {
              if
(message != "parsing
succeeded")
                  throw
new
Exception(message);
          }



    }
}
```

Syntax Analyzer: SyntaxParser.cs

```
using System.IO;

using Core.Library;

public class
SyntaxParser :
RecursiveDescentParser
{
```

```
    private enum
SynteticPatterns {
    }

    /**
     *
<summary>Creates a new
parser with a default
analyzer.</summary>
     *
     * <param
name='input'>the input
stream to read
from</param>
     *
     * <exception
cref='ParserCreationEx
ception'>if the parser
     * couldn't be
initialized
correctly</exception>
     */
    public
SyntaxParser(TextReade
r input)
        : base(input)
{


CreatePatterns();
    }

    /**
     *
<summary>Creates a new
parser.</summary>
     *
     * <param
name='input'>the input
stream to read
from</param>
     *
     * <param
name='analyzer'>the
analyzer to parse
with</param>
     *
     * <exception
cref='ParserCreationEx
ception'>if the parser
     * couldn't be
initialized
correctly</exception>
     */
    public
SyntaxParser(TextReade
r input,
SyntaxAnalyzer
analyzer)
        : base(input,
analyzer) {
```

```
CreatePatterns();
    }

    /**
     *
<summary>Creates a new
tokenizer for this
parser. Can be
overridden
     * by a subclass
to provide a custom
implementation.</summa
ry>
     *
     * <param
name='input'>the input
stream to read
from</param>
     *
     * <returns>the
tokenizer
created</returns>
     *
     * <exception
cref='ParserCreationEx
ception'>if the
tokenizer
     * couldn't be
initialized
correctly</exception>
     */
    protected override
Tokenizer
NewTokenizer(TextReade
r input) {
        return new
SyntaxTokenizer(input)
;
    }

    /**
     *
<summary>Initializes
the parser by creating
all the production
     *
patterns.</summary>
     *
     * <exception
cref='ParserCreationEx
ception'>if the parser
     * couldn't be
initialized
correctly</exception>
     */
    private void
CreatePatterns() {

ProductionPattern
pattern;
```

```
ProductionPatternAlter
native  alt;

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
TART_PROGRAM,

″Prod_StartProgram″);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_P
ROGRAM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
ND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_P
ROGRAM,

″Prod_program″);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_G
LOBAL_DEC, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
AIN, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
LEAR,

″Prod_Clear″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.CLEAR,
1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
OMMENTS,

″Prod_comments″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COM,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_N
EGATE,

″Prod_Negate″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.NEGA,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_D
ATATYPE,

″Prod_datatype″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.INT,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.FLOAT,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.STRING
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.CHAR,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.BOOL_N
, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_L
ITERALS,

″Prod_Literals″);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_N
EGATE, 0, 1);

alt.AddToken((int)
SyntaxConstants.NUM,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DECIMA
L, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.TEXT,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();
```

```
alt.AddToken((int)
SyntaxConstants.S_CHAR
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.YES,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.NO, 1,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_L
ITERALS2,

"Prod_Literals2");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_N
EGATE, 0, 1);

alt.AddToken((int)
SyntaxConstants.NUM,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DECIMA
L, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.TEXT,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.S_CHAR
, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_G
LOBAL_DEC,

"Prod_globalDec");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ATATYPE, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ECLARE, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.VOID,
1, 1);
alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCTVOID, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.STRUCT
_N, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
TRUCT, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.CONST_
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ATATYPE, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ONSTANT, 1, 1);
alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_D
ECLARE,

"Prod_Declare");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ECLARE_CHOICE, 0, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_G
LOBAL_DEC, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCTRET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_G
LOBAL_DEC, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);
```

```
pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_D
ECLARE_CHOICE,

"Prod_DeclareChoice");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NIT_CHOICE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_N
1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
RRAY_AID, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_I
NIT_CHOICE,

"Prod_InitChoice");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NIT_CHOICE, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
ITERALS, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
DD_ID, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
DD_ID,

"Prod_addID");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NIT_CHOICE, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_N
1,

"Prod_N1");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.S_OBRA
CKET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NDEX, 1, 1);

alt.AddToken((int)
SyntaxConstants.S_CBRA
CKET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_N
2, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_N
2,

"Prod_N2");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.S_OBRA
CKET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NDEX, 1, 1);

alt.AddToken((int)
SyntaxConstants.S_CBRA
CKET, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_I
NDEX,

"Prod_index");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.NUM,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
MATH, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
MATH, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
MATH,

"Prod_Smath");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NDEX, 1, 1);
```

```
pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
RRAY_AID,

"Prod_arrayAID");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
LEM_CHOICE, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_E
LEM_CHOICE,

"Prod_ElemChoice");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_E
LEMENT, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
LEMENT, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
_ELEM, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_E
LEMENT,

"Prod_Element");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_L
ITERALS2, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
DD_ELEM, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
DD_ELEM,

"Prod_addElem");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
LEMENT, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_M
_ELEM,

"Prod_M_Elem");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
LEMENT, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
2_ELEM, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_M
2_ELEM,

"Prod_M2_Elem");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_M
_ELEM, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
UNCTRET,

"Prod_functret");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
TYPE_A, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

alt.AddToken((int)
SyntaxConstants.RETURN
, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);
```

```
alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_G
LOBAL_DEC, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_D
TYPE_A,

"Prod_dtypeA");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ATATYPE, 0, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_N
1, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
XDTYPE_A, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_E
XDTYPE_A,

"Prod_EXdtypeA");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
TYPE_A, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_R
ETURN,

"Prod_return");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_L
ITERALS, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_N
EGATE, 0, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
UT_C, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.SQROOT
, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddToken((int)
SyntaxConstants.DOT,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
UNCTVOID,

"Prod_functvoid");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
TYPE_A, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_G
LOBAL_DEC, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
TRUCT,

"Prod_struct");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
EM_DEC, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_G
LOBAL_DEC, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);
```

```
        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_M
EM_DEC,

"Prod_memDec");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ATATYPE, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NIT_DEC, 0, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
EM_DEC, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_I
NIT_DEC,

"Prod_initDec");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NIT_DEC_CHOICE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_N
1, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_I
NIT_DEC_CHOICE,

"Prod_initDecChoice");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NIT_DEC_CHOICE, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
ONSTANT,

"Prod_constant");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
ITERALS, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_G
LOBAL_DEC, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE,

"Prod_LocalChoice");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ATATYPE, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ECLARE1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.VOID,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCTVOID1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.STRUCT
_N, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
TRUCT1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.CONST_
N, 1, 1);
```

```
alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ONSTANT1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_D
ECLARE1,

″Prod_Declare1″);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_D
ECLARE_CHOICE, 0, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCTRET1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
UNCTRET1,

″Prod_functret1″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
TYPE_A, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

alt.AddToken((int)
SyntaxConstants.RETURN
, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
UNCTVOID1,

″Prod_functvoid1″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
TYPE_A, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
TRUCT1,

″Prod_struct1″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
EM_DEC, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
ONSTANT1,

″Prod_constant1″);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
ITERALS, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);
```

```
alt.AddProduction((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_M
AIN,

"Prod_main");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.MAIN_N
, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
SSIGN_CHOICE,

"Prod_assignChoice");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_A
CCESS_ASSIGN_DTYPE, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_M
NT_COND_T, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.REVERS
E, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
CCESS_ASSIGN_DTYPE,

"Prod_AccessAssignDtyp
e");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
RRAY_ID, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
SSIGN_VALUE_CHOICE, 1,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
SSIGN_VALUE_CHOICE,

"Prod_assignValueChoic
e");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_A
SSIGNING, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
SSIGN_VALUE, 0, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
SSIGN_CHOICE, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
SSIGNING,

"Prod_assigning");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_A
SSIGN_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
ATH_OP, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DOT,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);
```

```
alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_M
NT, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCT_PARAM, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
RRAY_ID,

"Prod_ArrayID");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.S_OBRA
CKET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NDEX, 1, 1);

alt.AddToken((int)
SyntaxConstants.S_CBRA
CKET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
RRAY_IDTAIL, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
RRAY_IDTAIL,

"Prod_ArrayIDTail");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.S_OBRA
CKET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_I
NDEX, 1, 1);

alt.AddToken((int)
SyntaxConstants.S_CBRA
CKET, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
SSIGN_SYM,

"Prod_AssignSym");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
SSIGN_VALUE,

"Prod_assignValue");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_M
ATH_OP, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCT_PARAM, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
ONVERT,

"Prod_Convert");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.TOCHAR
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.LENGTH
F, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.CONTAI
NS, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
UNCT_PARAM,

"Prod_functParam");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCT_IDPARAM, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_F
UNCT_PARAM, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
UNCT_IDPARAM,

"Prod_functIDParam");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
DDFUNCT_IDPARAM, 0,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_A
DDFUNCT_IDPARAM,

"Prod_addfunctIDParam"
);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
DDFUNCT_IDPARAM, 0,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_B
ODY,

"Prod_body");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OCAL_CHOICE, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_P
RINT, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_S
CAN, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_F
OR_STATE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_I
FELSE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_D
OWHILE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_W
HILE_STATE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_S
WITCH_STATE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_A
SSIGN_CHOICE, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_C
OMMENTS, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_C
LEAR, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.BREAK,
1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_P
RINT,
"Prod_print");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.PRINT_
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_P
OSTVAL, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_P
OSTVAL,

"Prod_postval");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
UT, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ONCAT_LIT, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);
```

```
        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
UT,

"Prod_Out");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
UT_C,

"Prod_OutC");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_A
RRAY_ID, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
TRUCT_C, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
```

```
DDFUNCT_IDPARAM, 0,
1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DOT,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ONVERT, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
TRUCT_C,

"Prod_structC");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DOT,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
ONCAT_LIT,

"Prod_ConcatLit");
        alt = new
ProductionPatternAlter
native();
```

```
alt.AddToken((int)
SyntaxConstants.PLUS,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
UT, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ONCAT_LIT, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
CAN,

"Prod_scan");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.SCAN_N
, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.HASH,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
XT_I, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_E
XT_I,

"Prod_ExtI");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.COMMA,
1, 1);

alt.AddToken((int)
SyntaxConstants.HASH,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
XT_I, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
OR_STATE,

"Prod_for_state");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.FOR_N,
1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_V
AL1, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
RRAY_ID, 0, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
P1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_V
AL1, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
NT_COND, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_F
ORSTATEMENT, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_F
ORSTATEMENT,

"Prod_forstatement");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_V
AL1,

"Prod_val1");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.NUM,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_A
RRAY_ID, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_M
NT_COND,

"Prod_mntCond");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddProduction((int
)
SyntaxConstants.PROD_M
NT, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_M
NT_COND_T, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_M
NT_COND_T,

"Prod_mntCondT");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_M
NT, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);
```

```
pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_MNT,

"Prod_mnt");
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.INCREMENT, 1, 1);

pattern.AddAlternative(alt);
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.DECREMENT, 1, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_IFELSE,

"Prod_ifelse");
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.IF, 1, 1);

alt.AddToken((int) SyntaxConstants.O_PAREN, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_IFCONDITION, 1, 1);

alt.AddToken((int) SyntaxConstants.C_PAREN, 1, 1);

alt.AddToken((int) SyntaxConstants.O_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_IFSTATEMENT, 0, 1);

alt.AddToken((int) SyntaxConstants.C_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_ELSEIF, 0, 1);

alt.AddProduction((int) SyntaxConstants.PROD_ELSE_STATE, 0, 1);

alt.AddProduction((int) SyntaxConstants.PROD_BODY, 0, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_IFCONDITION,

"Prod_ifcondition");
        alt = new ProductionPatternAlternative();

alt.AddProduction((int) SyntaxConstants.PROD_REL_OP, 1, 1);

pattern.AddAlternative(alt);
        alt = new ProductionPatternAlternative();

alt.AddProduction((int) SyntaxConstants.PROD_LOG_OP, 1, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_IFSTATEMENT,

"Prod_ifstatement");
        alt = new ProductionPatternAlternative();

alt.AddProduction((int) SyntaxConstants.PROD_BODY, 1, 1);

pattern.AddAlternative(alt);
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.RETURN, 1, 1);

alt.AddToken((int) SyntaxConstants.O_PAREN, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_RETURN, 0, 1);

alt.AddToken((int) SyntaxConstants.C_PAREN, 1, 1);

alt.AddToken((int) SyntaxConstants.SEMIC, 1, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_ELSEIF,

"Prod_elseif");
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.ELSEIF_N, 1, 1);

alt.AddToken((int) SyntaxConstants.O_PAREN, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_IFCONDITION, 1, 1);

alt.AddToken((int) SyntaxConstants.C_PAREN, 1, 1);

alt.AddToken((int) SyntaxConstants.O_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_ELSEIFSTATEMENT, 0, 1);

alt.AddToken((int) SyntaxConstants.C_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_ELSEIF, 0, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_ELSEIFSTATEMENT,

"Prod_elseifstatement");
        alt = new ProductionPatternAlternative();

alt.AddProduction((int) SyntaxConstants.PROD_BODY, 1, 1);
```

```
pattern.AddAlternative(alt);
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.RETURN, 1, 1);

alt.AddToken((int) SyntaxConstants.O_PAREN, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_RETURN, 0, 1);

alt.AddToken((int) SyntaxConstants.C_PAREN, 1, 1);

alt.AddToken((int) SyntaxConstants.SEMIC, 1, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_ELSE_STATE,

"Prod_else_state");
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.ELSE_N, 1, 1);

alt.AddToken((int) SyntaxConstants.O_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_ELSESTATEMENT, 0, 1);

alt.AddToken((int) SyntaxConstants.C_BRACKET, 1, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_ELSESTATEMENT,

"Prod_elsestatement");
        alt = new ProductionPatternAlternative();

alt.AddProduction((int) SyntaxConstants.PROD_BODY, 1, 1);

pattern.AddAlternative(alt);
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.RETURN, 1, 1);

alt.AddToken((int) SyntaxConstants.O_PAREN, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_RETURN, 0, 1);

alt.AddToken((int) SyntaxConstants.C_PAREN, 1, 1);

alt.AddToken((int) SyntaxConstants.SEMIC, 1, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_DOWHILE,

"Prod_dowhile");
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.DO, 1, 1);

alt.AddToken((int) SyntaxConstants.O_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_DOSTATEMENT, 0, 1);

alt.AddToken((int) SyntaxConstants.C_BRACKET, 1, 1);

alt.AddToken((int) SyntaxConstants.WHILE_N, 1, 1);

alt.AddToken((int) SyntaxConstants.O_PAREN, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_IFCONDITION, 1, 1);

alt.AddToken((int) SyntaxConstants.C_PAREN, 1, 1);

alt.AddToken((int) SyntaxConstants.SEMIC, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_BODY, 0, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_DOSTATEMENT,

"Prod_dostatement");
        alt = new ProductionPatternAlternative();

alt.AddProduction((int) SyntaxConstants.PROD_BODY, 1, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);

        pattern = new ProductionPattern((int) SyntaxConstants.PROD_WHILE_STATE,

"Prod_while_state");
        alt = new ProductionPatternAlternative();

alt.AddToken((int) SyntaxConstants.WHILE_N, 1, 1);

alt.AddToken((int) SyntaxConstants.O_PAREN, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_IFCONDITION, 1, 1);

alt.AddToken((int) SyntaxConstants.C_PAREN, 1, 1);

alt.AddToken((int) SyntaxConstants.O_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_WHILESTATEMENT, 0, 1);

alt.AddToken((int) SyntaxConstants.C_BRACKET, 1, 1);

alt.AddProduction((int) SyntaxConstants.PROD_BODY, 0, 1);

pattern.AddAlternative(alt);

AddPattern(pattern);
```

```
        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_W
HILESTATEMENT,

"Prod_whilestatement")
;
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
WITCH_STATE,

"Prod_switch_state");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.SWITCH
_N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.ID, 1,
1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.O_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ASE_STATE, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_D
EF, 0, 1);

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
ASE_STATE,

"Prod_case_state");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.CASE_N
, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
ITERALS, 1, 1);

alt.AddToken((int)
SyntaxConstants.COLON,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ASESTATEMENT, 0, 1);

alt.AddToken((int)
SyntaxConstants.BREAK,
1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ASE_STATE, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_D
EF,

"Prod_def");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DEFAUL
T, 1, 1);

alt.AddToken((int)
SyntaxConstants.COLON,
1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_C
ASESTATEMENT, 0, 1);

alt.AddToken((int)
SyntaxConstants.BREAK,
1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_C
ASESTATEMENT,

"Prod_casestatement");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_B
ODY, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_M
ATH_OP,

"Prod_MathOp");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_COND, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PER_COND,

"Prod_operCond");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_EXT_S, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_COND_EXT, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_COND_CHOICE, 0,
1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PER_COND_CHOICE,

"Prod_operCondChoice")
;
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_EXT_S, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PER_SYM,

"Prod_operSym");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.PLUS,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.MINUS,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.TIMES,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DIVIDE
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.MODULU
S, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.POWER,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.DOT,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PER_EQ,

"Prod_operEq");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.P_E,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.M_E,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.T_E,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.D_E,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.MOD_E,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PER_EXT_S,

"Prod_operExt_s");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
_MATH_EXT, 0, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
IM_MATH_OP, 1, 1);
```

```
alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_EXT_REP, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PER_EXT_REP,

"Prod_operExt_rep");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_EXT_S, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PERAND,

"Prod_operand");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ETURN, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
IM_MATH_OP,

"Prod_simMathOp");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
_MATH_EXT, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_S
_MATH_EXT,

"Prod_S_MathExt");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_S
_MATH_EXT, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
PER_COND_EXT,

"Prod_operCondExt");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_SYM, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_EXT_S, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_R
EL_OP,

"Prod_RelOp");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ELOP_EXT, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_R
ELOP_EXT,

"Prod_RelopExt");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
P1, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PERAND, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
ELOP_EXT, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_O
P1,

"Prod_op1");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.N_E,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.GREATE
R, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.LESS,
1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.GREATE
R_E, 1, 1);
```

```
pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.LESS_E
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

alt.AddToken((int)
SyntaxConstants.EQUALS
, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_O
PER_EQ, 1, 1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.MODULU
S, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_L
OG_OP,

"Prod_LogOp");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_R
EL_OP, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_E
XT_LOG_OP, 0, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_E
XT_LOG_OP,

"Prod_ExtLogOp");
        alt = new
ProductionPatternAlter
native();

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OG_OPER, 1, 1);

alt.AddProduction((int
)
SyntaxConstants.PROD_L
OG_OP, 1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_L
OG_OPER,

"Prod_LogOper");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.OR, 1,
1);

pattern.AddAlternative
(alt);
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.AND,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);

        pattern = new
ProductionPattern((int
)
SyntaxConstants.PROD_E
ND,

"Prod_end");
        alt = new
ProductionPatternAlter
native();

alt.AddToken((int)
SyntaxConstants.C_BRAC
KET, 1, 1);

alt.AddToken((int)
SyntaxConstants.GETCH,
1, 1);

alt.AddToken((int)
SyntaxConstants.O_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.C_PARE
N, 1, 1);

alt.AddToken((int)
SyntaxConstants.SEMIC,
1, 1);

pattern.AddAlternative
(alt);

AddPattern(pattern);
    }
}
```

Syntax Analyzer: SyntaxTokenizer.cs

```
using System.IO;

using Core.Library;

public class
SyntaxTokenizer :
Tokenizer {

public
SyntaxTokenizer(TextRe
ader input)
        : base(input,
false) {


CreatePatterns();
    }

    /**
     *
<summary>Initializes
the tokenizer by
creating all the token
     *
patterns.</summary>
     *
     * <exception
cref='ParserCreationEx
ception'>if the
tokenizer
     * couldn't be
initialized
correctly</exception>
     */
    private void
CreatePatterns() {
        TokenPattern
pattern;

        pattern = new
TokenPattern((int)
SyntaxConstants.MAIN_N
,

"MAIN_N",

TokenPattern.PatternTy
pe.STRING,

"PrimaryMission");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.PRINT_
N,

"PRINT_N",

TokenPattern.PatternTy
pe.STRING,

"post");
```

```
AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.SCAN_N
,

"SCAN_N",

TokenPattern.PatternTy
pe.STRING,

"capture");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.CONST_
N,

"CONST_N",

TokenPattern.PatternTy
pe.STRING,

"hold");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.RETURN
,

"RETURN",

TokenPattern.PatternTy
pe.STRING,

"backup");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.SWITCH
_N,

"SWITCH_N",

TokenPattern.PatternTy
pe.STRING,

"campaign");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.CASE_N
,

"CASE_N",

TokenPattern.PatternTy
pe.STRING,

"operation");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.BREAK,

"BREAK",

TokenPattern.PatternTy
pe.STRING,

"abort");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.FOR_N,

"FOR_N",

TokenPattern.PatternTy
pe.STRING,

"inquire");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.IF,

"IF",

TokenPattern.PatternTy
pe.STRING,

"inorder");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.ELSEIF
_N,

"ELSEIF_N",

TokenPattern.PatternTy
pe.STRING,

"otherorder");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.ELSE_N
,

"ELSE_N",

TokenPattern.PatternTy
pe.STRING,

"order");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.DO,

"DO",

TokenPattern.PatternTy
pe.STRING,

"go");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.WHILE_
N,

"WHILE_N",

TokenPattern.PatternTy
pe.STRING,

"phase");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.VOID,

"VOID",

TokenPattern.PatternTy
pe.STRING,

"miss");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.GETCH,

"GETCH",

TokenPattern.PatternTy
pe.STRING,

"deploy");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.STRUCT
_N,

"STRUCT_N",

TokenPattern.PatternTy
pe.STRING,

"struct");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.DEFAUL
T,

"DEFAULT",

TokenPattern.PatternTy
pe.STRING,

"action");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.CLEAR,

"CLEAR",

TokenPattern.PatternTy
pe.STRING,

"commence");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.SQROOT
,

"SQROOT",

TokenPattern.PatternTy
pe.STRING,

"sqrt");

AddPattern(pattern);
```

```
        pattern = new TokenPattern((int) SyntaxConstants.PLUS,

″PLUS″,

TokenPattern.PatternType.STRING,

″+″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.MINUS,

″MINUS″,

TokenPattern.PatternType.STRING,

″-″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.TIMES,

″TIMES″,

TokenPattern.PatternType.STRING,

″*″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.DIVIDE,

″DIVIDE″,

TokenPattern.PatternType.STRING,

″/″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.MODULUS,

″MODULUS″,

TokenPattern.PatternType.STRING,

″%″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.EQUALS,

″EQUALS″,

TokenPattern.PatternType.STRING,

″=″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.SEMIC,

″SEMIC″,

TokenPattern.PatternType.STRING,

″;″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.DOT,

″DOT″,

TokenPattern.PatternType.STRING,

″.″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.COMMA,

″COMMA″,

TokenPattern.PatternType.STRING,

″,″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.AND,

″AND″,

TokenPattern.PatternType.STRING,

″&″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.OR,

″OR″,

TokenPattern.PatternType.STRING,

″||″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.NOT,

″NOT″,

TokenPattern.PatternType.STRING,

″!″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.INCREMENT,

″INCREMENT″,

TokenPattern.PatternType.STRING,

″++″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.DECREMENT,

″DECREMENT″,

TokenPattern.PatternType.STRING,

″--″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.P_E,

″P_E″,

TokenPattern.PatternType.STRING,

″+=″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.M_E,

″M_E″,

TokenPattern.PatternType.STRING,

″-=″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.T_E,

″T_E″,

TokenPattern.PatternType.STRING,

″*=″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.D_E,

″D_E″,

TokenPattern.PatternType.STRING,

″/=″);

AddPattern(pattern);

        pattern = new TokenPattern((int) SyntaxConstants.MOD_E,

″Mod_E″,
TokenPattern.PatternType.STRING,

"%=");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.NEWLINE,

"NEWLINE",

TokenPattern.PatternType.STRING,

"\\n");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.N_E,

"N_E",

TokenPattern.PatternType.STRING,

"!=");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.O_PAREN,

"O_PAREN",

TokenPattern.PatternType.STRING,

"(");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.C_PAREN,

"C_PAREN",

TokenPattern.PatternType.STRING,

")");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.D_QUOTE,

"D_QUOTE",

TokenPattern.PatternType.REGEXP,

"[\"]");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.COLON,

"COLON",

TokenPattern.PatternType.STRING,

":");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.O_BRACKET,

"O_BRACKET",

TokenPattern.PatternType.STRING,

"{");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.C_BRACKET,

"C_BRACKET",

TokenPattern.PatternType.STRING,

"}");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.GREATER,

"GREATER",

TokenPattern.PatternType.STRING,

"<");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.LESS,

"LESS",

TokenPattern.PatternType.STRING,

">");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.GREATER_E,

"GREATER_E",

TokenPattern.PatternType.STRING,

"<=");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.LESS_E,

"LESS_E",

TokenPattern.PatternType.STRING,

">=");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.S_OBRACKET,

"S_OBRACKET",

TokenPattern.PatternType.STRING,

"[");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.S_CBRACKET,

"S_CBRACKET",

TokenPattern.PatternType.STRING,

"]");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.DOLLAR,

"DOLLAR",

TokenPattern.PatternType.STRING,

"$");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.POWER,

"POWER",

TokenPattern.PatternType.STRING,

"^");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.HASH,

"HASH",

TokenPattern.PatternType.STRING,

"#");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.NEGA,

"NEGA",
TokenPattern.PatternTy
pe.STRING,

″~″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.INT,

″INT″,

TokenPattern.PatternTy
pe.STRING,

″unit″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.CHAR,

″CHAR″,

TokenPattern.PatternTy
pe.STRING,

″joe″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.FLOAT,

″FLOAT″,

TokenPattern.PatternTy
pe.STRING,

″digit″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.STRING
,

″STRING″,

TokenPattern.PatternTy
pe.STRING,

″company″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.BOOL_N
,

″BOOL_N″,

TokenPattern.PatternTy
pe.STRING,

″response″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.ID,

″ID″,

TokenPattern.PatternTy
pe.STRING,

″id″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.NUM,

″NUM″,

TokenPattern.PatternTy
pe.STRING,

″Numlit″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.DECIMA
L,

″DECIMAL″,

TokenPattern.PatternTy
pe.STRING,

″Declit″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.S_CHAR
,

″S_CHAR″,

TokenPattern.PatternTy
pe.STRING,

″Charlit″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.TEXT,

″TEXT″,

TokenPattern.PatternTy
pe.STRING,

″Stringlit″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.COM,

″COM″,

TokenPattern.PatternTy
pe.STRING,

″comment″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.YES,

″YES″,

TokenPattern.PatternTy
pe.STRING,

″AFFIRMATIVE″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.NO,

″NO″,

TokenPattern.PatternTy
pe.STRING,

″NEGATIVE″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.FUNCTN
AME,

″FUNCTNAME″,

TokenPattern.PatternTy
pe.STRING,

″functName″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.STRUCT
NAME,

″STRUCTNAME″,

TokenPattern.PatternTy
pe.STRING,

″structname″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.IDSTRU
CT,

″IDSTRUCT″,

TokenPattern.PatternTy
pe.STRING,

″idStruct″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.F,

″F″,

TokenPattern.PatternTy
pe.STRING,

″f″);

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.D,

″D″,

TokenPattern.PatternTy
pe.STRING,
"d");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.S,

"S",

TokenPattern.PatternTy
pe.STRING,

"s");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.ZERO,

"ZERO",

TokenPattern.PatternTy
pe.STRING,

"Zero");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.SPACE,

"SPACE",

TokenPattern.PatternTy
pe.STRING,

" ");
        pattern.Ignore
= true;

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.N_LINE
,

"N_LINE",

TokenPattern.PatternTy
pe.STRING,

"\\n");
        pattern.Ignore
= true;

AddPattern(pattern);
```

```
        pattern = new
TokenPattern((int)
SyntaxConstants.WHITES
PACE,

"WHITESPACE",

TokenPattern.PatternTy
pe.REGEXP,

"[ \\t\\n\\r]+");
        pattern.Ignore
= true;

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.TOCHAR
,

"TOCHAR",

TokenPattern.PatternTy
pe.STRING,

"ToJoeRange");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.LENGTH
F,

"LENGTHF",

TokenPattern.PatternTy
pe.STRING,

"Extent");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.CONTAI
NS,

"CONTAINS",

TokenPattern.PatternTy
pe.STRING,

"Carry");

AddPattern(pattern);

        pattern = new
TokenPattern((int)
SyntaxConstants.REVERS
E,
```

```
"REVERSE",

TokenPattern.PatternTy
pe.STRING,

"Swap");

AddPattern(pattern);
    }
}
```

Core.Library

```
using
System.Collections;

namespace Core.Library
{

    /**
     * A parse tree
analyzer. This class
provides callback
methods that
     * may be used
either during parsing,
or for a parse tree
traversal.
     * This class
should be subclassed
to provide adequate
handling of the
     * parse tree
nodes.
     *
     * The general
contract for the
analyzer class does
not guarantee a
     * strict call
order for the callback
methods. Depending on
the type
     * of parser, the
enter() and exit()
methods for production
nodes can
     * be called
either in a top-down
or a bottom-up
fashion. The only
     * guarantee
provided by this API,
is that the calls for
any given
     * node will
always be in the order
enter(), child(), and
exit(). If
```

```
     * various child()
calls are made, they
will be made from left
to
     * right as child
nodes are added (to
the right).
     */
    public class
Analyzer {

        /**
         * Creates a
new parse tree
analyzer.
         */
        public
Analyzer() {
        }

        /**
         * Resets this
analyzer when the
parser is reset for
another
         * input
stream. The default
implementation of this
method does
         * nothing.
         *
         *
         */
        public virtual
void Reset() {
            // Default
implementation does
nothing
        }

        /**
         * Analyzes a
parse tree node by
traversing all it's
child nodes.
         * The tree
traversal is depth-
first, and the
appropriate
         * callback
methods will be
called. If the node is
a production
         * node, a new
production node will
be created and
children will
         * be added by
recursively processing
the children of the
         * specified
production node. This
```

```
method is used to
process a
         * parse tree
after creation.
         *
         * @param node
the parse tree node to
process
         *
         * @return the
resulting parse tree
node
         *
         * @throws
ParserLogException if
the node analysis
discovered
         *
errors
         */
        public virtual
Node Analyze(Node
node) {

ParserLogException
log = new
ParserLogException();

            node =
Analyze(node, log);
            if
(log.Count > 0) {
                throw
log;
            }
            return
node;
        }

        /**
         * Analyzes a
parse tree node by
traversing all it's
child nodes.
         * The tree
traversal is depth-
first, and the
appropriate
         * callback
methods will be
called. If the node is
a production
         * node, a new
production node will
be created and
children will
         * be added by
recursively processing
the children of the
         * specified
production node. This
method is used to
process a
         * parse tree
after creation.
         *
         * @param node
the parse tree node to
process
         * @param log
the parser error log
         *
         * @return the
resulting parse tree
node
         */
        public virtual
Node Analyze(Node
node,
ParserLogException
log) {
            Production
prod;
            int
errorCount;

            Node res =
null;
            errorCount
= log.Count;
            if (node
is Production) {
                prod =
(Production) node;
                prod =
NewProduction(prod.Pat
tern);
                try {

Enter(prod);
                }
catch (ParseException
e) {

log.AddError(e);
                }
                for
(int i = 0; i <
node.Count; i++) {

try {

Child(prod,
Analyze(node[i],
log));
                    }
catch (ParseException
e) {

log.AddError(e);
                    }
                }
                try {

res = Exit(prod);

return res;
                }
catch (ParseException
e) {
                    if
(errorCount ==
log.Count) {

log.AddError(e);
                    }
                }
            }
            else {

node.Values.Clear();
                try {

Enter(node);
                }
catch (ParseException
e) {

log.AddError(e);
                }
                try {

res = Exit(node);

return res;
                }
catch (ParseException
e) {
                    if
(errorCount ==
log.Count) {

log.AddError(e);
                    }
                }
            }
            return
null;
        }

        /**
         * Factory
method to create a new
production node. This
method
         * can be
overridden to provide
other production
implementations
         * than the
default one.
         *
         * @param
pattern       the
production pattern
         *
         * @return the
new production node
         *
         */
        public virtual
Production
NewProduction(Producti
onPattern pattern) {
            return new
Production(pattern);
        }

        /**
         * Called when
entering a parse tree
node. By default this
method
         * does
nothing. A subclass
can override this
method to handle
         * each node
separately.
         *
         * @param node
the node being entered
         *
         * @throws
ParseException if the
node analysis
discovered errors
         */
        public virtual
void Enter(Node node)
{
        }

        /**
         * Called when
exiting a parse tree
node. By default this
method
         * returns the
node. A subclass can
override this method
to handle
         * each node
separately. If no
parse tree should be
created, this
         * method
should return null.
         *
         * @param node
the node being exited
         *
         * @return the
node to add to the
parse tree, or
```

```
         *
null if no parse tree
should be created
         *
         * @throws
ParseException if the
node analysis
discovered errors
         */
        public virtual
Node Exit(Node node) {
            return
node;
        }

        /**
         * Called when
adding a child to a
parse tree node. By
default
         * this method
adds the child to the
production node. A
subclass
         * can
override this method
to handle each node
separately. Note
         * that the
child node may be null
if the corresponding
exit()
         * method
returned null.
         *
         * @param node
the parent node
         * @param
child          the
child node, or null
         *
         * @throws
ParseException if the
node analysis
discovered errors
         */
        public virtual
void Child(Production
node, Node child) {

node.AddChild(child);
        }

        /**
         * Returns a
child at the specified
position. If either
the node
         * or the
child node is null,
this method will throw
a parse
         * exception
with the internal
error type.
         *
         * @param node
the parent node
         * @param pos
the child position
         *
         * @return the
child node
         *
         * @throws
ParseException if
either the node or the
child node
         *
was null
         */
        protected Node
GetChildAt(Node node,
int pos) {
            Node
child;

            if (node
== null) {
                throw
new ParseException(

ParseException.ErrorTy
pe.INTERNAL,

"attempt to read
'null' parse tree
node",
                       -
1,
                       -
1);
            }
            child =
node[pos];
            if (child
== null) {
                throw
new ParseException(

ParseException.ErrorTy
pe.INTERNAL,

"node '" + node.Name +
"' has no child at " +

"position " + pos,

node.StartLine,

node.StartColumn);
            }
            return
child;
        }

        /**
         * Returns the
first child with the
specified id. If the
node is
         * null, or no
child with the
specified id could be
found, this
         * method will
throw a parse
exception with the
internal error
         * type.
         *
         * @param node
the parent node
         * @param id
the child node id
         *
         * @return the
child node
         *
         * @throws
ParseException if the
node was null, or a
child node
         *
couldn't be found
         */
        protected Node
GetChildWithId(Node
node, int id) {
            Node
child;

            if (node
== null) {
                throw
new ParseException(

ParseException.ErrorTy
pe.INTERNAL,

"attempt to read
'null' parse tree
node",
                       -
1,
                       -
1);
            }
            for (int i
= 0; i < node.Count;
i++) {
                child
= node[i];
                if
(child != null &&
child.Id == id) {

return child;
                }
            }
            throw new
ParseException(

ParseException.ErrorTy
pe.INTERNAL,
                "node
'" + node.Name + "'
has no child with id "
+ id,

node.StartLine,

node.StartColumn);
        }

        /**
         * Returns the
node value at the
specified position. If
either
         * the node or
the value is null,
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 or the
value was null
         */
        protected
object GetValue(Node
node, int pos) {
            object
value;

            if (node
== null) {
                throw
new ParseException(

ParseException.ErrorTy
pe.INTERNAL,

"attempt to read
'null' parse tree
node",
                    -
1,
                    -
1);
            }
            value =
node.Values[pos];
            if (value
== null) {
                throw
new ParseException(

ParseException.ErrorTy
pe.INTERNAL,

"node '" + node.Name +
"' has no value at " +

"position " + pos,

node.StartLine,

node.StartColumn);
            }
            return
value;
        }

        /**
         * Returns the
node integer value at
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
integer
         */
```

```
        protected int
GetIntValue(Node node,
int pos) {
            object
value;

            value =
GetValue(node, pos);
            if (value
is int) {
                return
(int) value;
            } else {
                throw
new ParseException(

ParseException.ErrorTy
pe.INTERNAL,

"node '" + node.Name +
"' has no integer
value " +

"at position " + pos,

node.StartLine,

node.StartColumn);
            }
        }

        /**
         * 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
         *
value wasn't a string
         */
```

```
        protected
string
GetStringValue(Node
node, int pos) {
            object
value;

            value =
GetValue(node, pos);
            if (value
is string) {
                return
(string) value;
            } else {
                throw
new ParseException(

ParseException.ErrorTy
pe.INTERNAL,

"node '" + node.Name +
"' has no string value
" +

"at position " + pos,

node.StartLine,

node.StartColumn);
            }
        }

        /**
         * 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++) {
                child
= node[i];
```

```
                values
= child.Values;
                if
(values != null) {

result.AddRange(values
);
                }
            }
            return
result;
        }
    }
}

/*
 * LookAheadSet.cs
 */

using
System.Collections;
using System.Text;

namespace Core.Library
{

    /**
     * A token look-
ahead set. This class
contains a set of
token id
     * sequences. All
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
conflicting sequences
can be repeated (would
     * cause infinite
loop).
     *

     * @version  1.1
     */
    internal class
LookAheadSet {

        /**
         * The set of
token look-ahead
sequences. Each
sequence in
         * turn is
represented by an
ArrayList with
Integers for the
         * token id:s.
         */
        private
ArrayList elements =
new ArrayList();

        /**
         * The maximum
length of any look-
ahead sequence.
         */
        private int
maxLength;

        /**
         * 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.
         *
         * @param
maxLength      the
maximum token sequence
length
         * @param set
the look-ahead set to
copy
         */
        public
LookAheadSet(int
maxLength,
LookAheadSet set)
            :
this(maxLength) {


AddAll(set);
        }

        /**
         * Returns the
size of this look-
ahead set.
         *
         * @return the
number of token
sequences in the set
         */
        public int
Size() {
            return
elements.Count;
        }

        /**
         * 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
GetMinLength() {
            Sequence
seq;
            int
min = -1;

            for (int i
= 0; i <
elements.Count; i++) {
                seq =
(Sequence)
elements[i];
                if
(min < 0 ||
seq.Length() < min) {

min = seq.Length();
                }
            }
            return
(min < 0) ? 0 : min;
        }

        /**
         * Returns the
length of the longest
token sequence in this
         * set. This
method will return
zero (0) if the set is
empty.
         *
         * @return the
length of the longest
token sequence
         */
        public int
GetMaxLength() {
            Sequence
seq;
            int
max = 0;

            for (int i
= 0; i <
elements.Count; i++) {
                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
look-ahead set
         */
        public int[]
GetInitialTokens() {
            ArrayList
list = new
ArrayList();
            int[]
result;
            object
token;
            int
i;

            for (i =
0; i < elements.Count;
i++) {
                token
= ((Sequence)
elements[i]).GetToken(
0);
                if
(token != null &&
!list.Contains(token))
{

list.Add(token);
                }
            }
            result =
new int[list.Count];
            for (i =
0; i < list.Count;
i++) {

result[i] = (int)
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
seq;

            for (int i
= 0; i <
elements.Count; i++) {
```

```
                seq =
(Sequence)
elements[i];
                if
(seq.IsRepetitive()) {

return true;
                }
            }
            return
false;
        }

        /**
         * Checks if
the next token(s) in
the parser match any
token
         * sequence in
this set.
         *
         * @param
parser         the
parser to check
         *
         * @return
true if the next
tokens are in the set,
or
         *
false otherwise
         */
        public bool
IsNext(Parser parser)
{
            Sequence
seq;

            for (int i
= 0; i <
elements.Count; i++) {
                seq =
(Sequence)
elements[i];
                if
(seq.IsNext(parser)) {

return true;
                }
            }
            return
false;
        }

        /**
         * Checks if
the next token(s) in
the parser match any
token
         * sequence in
this set.
         *
```

```
         * @param
parser         the
parser to check
         * @param
length         the
maximum number of
tokens to check
         *
         * @return
true if the next
tokens are in the set,
or
         *
false otherwise
         */
        public bool
IsNext(Parser parser,
int length) {
            Sequence
seq;

            for (int i
= 0; i <
elements.Count; i++) {
                seq =
(Sequence)
elements[i];
                if
(seq.IsNext(parser,
length)) {

return true;
                }
            }
            return
false;
        }

        /**
         * Checks if
another look-ahead set
has an overlapping
token
         * sequence.
An overlapping token
sequence is a token
sequence
         * that is
identical to another
sequence, but for the
length.
         * I.e. one of
the two sequences may
be longer than the
other.
         *
         * @param set
the look-ahead set to
check
         *
         * @return
true if there is some
```

```
token sequence that
overlaps, or
         *
false otherwise
         */
        public bool
IsOverlap(LookAheadSet
set) {
            for (int i
= 0; i <
elements.Count; i++) {
                if
(set.IsOverlap((Sequen
ce) elements[i])) {

return true;
                }
            }
            return
false;
        }

        /**
         * Checks if a
token sequence is
overlapping. An
overlapping token
         * sequence is
a token sequence that
is identical to
another
         * sequence,
but for the length.
I.e. one of the two
sequences may
         * be longer
than the other.
         *
         * @param seq
the token sequence to
check
         *
         * @return
true if there is some
token sequence that
overlaps, or
         *
false otherwise
         */
        private bool
IsOverlap(Sequence
seq) {
            Sequence
elem;

            for (int i
= 0; i <
elements.Count; i++) {
                elem =
(Sequence)
elements[i];
```

```
                if
(seq.StartsWith(elem)
||
elem.StartsWith(seq))
{

return true;
                }
            }
            return
false;
        }

        /**
         * Checks if
the specified token
sequence is present in
the
         * set.
         *
         * @param elem
the token sequence to
check
         *
         * @return
true if the sequence
is present in this
set, or
         *
false otherwise
         */
        private bool
Contains(Sequence
elem) {
            return
FindSequence(elem) !=
null;
        }

        /**
         * Checks if
some token sequence is
present in both this
set
         * and a
specified one.
         *
         * @param set
the look-ahead set to
compare with
         *
         * @return
true if the look-ahead
sets intersect, or
         *
false otherwise
         */
        public bool
Intersects(LookAheadSe
t set) {
```

```
            for (int i
= 0; i <
elements.Count; i++) {
                if
(set.Contains((Sequenc
e) elements[i])) {

return true;
                }
            }
            return
false;
        }

        /**
         * Finds an
identical token
sequence if present in
the set.
         *
         * @param elem
the token sequence to
search for
         *
         * @return an
identical the token
sequence if found, or
         *
null if not found
         */
        private
Sequence
FindSequence(Sequence
elem) {
            for (int i
= 0; i <
elements.Count; i++) {
                if
(elements[i].Equals(el
em)) {

return (Sequence)
elements[i];
                }
            }
            return
null;
        }

        /**
         * Adds a
token sequence to this
set. The sequence will
only
         * be added if
it is not already in
the set. Also, if the
         * sequence is
longer than the
allowed maximum, a
truncated
```

```
         * sequence
will be added instead.
         *
         * @param seq
the token sequence to
add
         */
        private void
Add(Sequence seq) {
            if
(seq.Length() >
maxLength) {
                seq =
new
Sequence(maxLength,
seq);
            }
            if
(!Contains(seq)) {

elements.Add(seq);
            }
        }

        /**
         * Adds a new
token sequence with a
single token to this
set.
         * The
sequence will only be
added if it is not
already in the
         * set.
         *
         * @param
token          the
token to add
         */
        public void
Add(int token) {
            Add(new
Sequence(false,
token));
        }

        /**
         * Adds all
the token sequences
from a specified set.
Only
         * sequences
not already in this
set will be added.
         *
         * @param set
the set to add from
         */
        public void
AddAll(LookAheadSet
set) {
```

```
            for (int i
= 0; i <
set.elements.Count;
i++) {

Add((Sequence)
set.elements[i]);
            }
        }

        /**
         * Adds an
empty token sequence
to this set. The
sequence will
         * only be
added if it is not
already in the set.
         */
        public void
AddEmpty() {
            Add(new
Sequence());
        }

        /**
         * Removes a
token sequence from
this set.
         *
         * @param seq
the token sequence to
remove
         */
        private void
Remove(Sequence seq) {

elements.Remove(seq);
        }

        /**
         * Removes all
the token sequences
from a specified set.
Only
         * sequences
already in this set
will be removed.
         *
         * @param set
the set to remove from
         */
        public void
RemoveAll(LookAheadSet
set) {
            for (int i
= 0; i <
set.elements.Count;
i++) {

Remove((Sequence)
set.elements[i]);
```

```
            }
        }

        /**
         * Creates a
new look-ahead set
that is the result of
reading
         * the
specified token. The
new look-ahead set
will contain
         * the rest of
all the token
sequences that started
with the
         * specified
token.
         *
         * @param
token          the
token to read
         *
         * @return a
new look-ahead set
containing the
remaining tokens
         */
        public
LookAheadSet
CreateNextSet(int
token) {

LookAheadSet  result =
new
LookAheadSet(maxLength
- 1);
            Sequence
seq;
            object
value;

            for (int i
= 0; i <
elements.Count; i++) {
                seq =
(Sequence)
elements[i];
                value
= seq.GetToken(0);
                if
(value != null &&
token == (int) value)
{

result.Add(seq.Subsequ
ence(1));
                }
            }
            return
result;
        }
        /**
         * Creates a
new look-ahead set
that is the
intersection of
         * this set
with another set. The
token sequences in the
net
         * set will
only have the repeat
flag set if it was set
in
         * both the
identical token
sequences.
         *
         * @param set
the set to intersect
with
         *
         * @return a
new look-ahead set
containing the
intersection
         */
        public
LookAheadSet
CreateIntersection(Loo
kAheadSet set) {

LookAheadSet  result =
new
LookAheadSet(maxLength
);
            Sequence
seq1;
            Sequence
seq2;

            for (int i
= 0; i <
elements.Count; i++) {
                seq1 =
(Sequence)
elements[i];
                seq2 =
set.FindSequence(seq1)
;
                if
(seq2 != null &&
seq1.IsRepetitive()) {

result.Add(seq2);
                } else
if (seq2 != null) {

result.Add(seq1);
                }
            }
            return
result;
        }

        /**
         * Creates a
new look-ahead set
that is the
combination of
         * this set
with another set. The
combination is created
by
         * creating
new token sequences
that consist of
appending all
         * elements
from the specified set
onto all elements in
this
         * set. This
is sometimes referred
to as the cartesian
         * product.
         *
         * @param set
the set to combine
with
         *
         * @return a
new look-ahead set
containing the
combination
         */
        public
LookAheadSet
CreateCombination(Look
AheadSet set) {

LookAheadSet  result =
new
LookAheadSet(maxLength
);
            Sequence
first;
            Sequence
second;

            // Handle
special cases
            if
(this.Size() <= 0) {
                return
set;
            } else if
(set.Size() <= 0) {
                return
this;
            }
            // Create
combinations
            for (int i
= 0; i <
elements.Count; i++) {
                first
= (Sequence)
elements[i];
                if
(first.Length() >=
maxLength) {

result.Add(first);
                } else
if (first.Length() <=
0) {

result.AddAll(set);
                } else
{

for (int j = 0; j <
set.elements.Count;
j++) {

second = (Sequence)
set.elements[j];

result.Add(first.Conca
t(maxLength, second));
                    }
                }
            }
            return
result;
        }

        /**
         * Creates a
new look-ahead set
with overlaps from
another. All
         * token
sequences in this set
that overlaps with the
other set
         * will be
added to the new look-
ahead set.
         *
         * @param set
the look-ahead set to
check with
         *
         * @return a
new look-ahead set
containing the
overlaps
         */
        public
LookAheadSet
CreateOverlaps(LookAhe
adSet set) {

LookAheadSet  result =
new
LookAheadSet(maxLength
);
            Sequence
seq;

            for (int i
= 0; i <
elements.Count; i++) {
                seq =
(Sequence)
elements[i];
                if
(set.IsOverlap(seq)) {

result.Add(seq);
                }
            }
            return
result;
        }

        /**
         * Creates a
new look-ahead set
filter. The filter
will contain
         * all
sequences from this
set, possibly left
trimmed by each one
         * of the
sequences in the
specified set.
         *
         * @param set
the look-ahead set to
trim with
         *
         * @return a
new look-ahead set
filter
         */
        public
LookAheadSet
CreateFilter(LookAhead
Set set) {

LookAheadSet  result =
new
LookAheadSet(maxLength
);
            Sequence
first;
            Sequence
second;
```

```
            // Handle
special cases
            if
(this.Size() <= 0 ||
set.Size() <= 0) {
                return
this;
            }

            // Create
combinations
            for (int i
= 0; i <
elements.Count; i++) {
                first
= (Sequence)
elements[i];
                for
(int j = 0; j <
set.elements.Count;
j++) {

second = (Sequence)
set.elements[j];
                    if
(first.StartsWith(seco
nd)) {

result.Add(first.Subse
quence(second.Length()
));
                    }
                }
            }
            return
result;
        }

        /**
         * 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
seq;
```

```
            for (int i
= 0; i <
elements.Count; i++) {
                seq =
(Sequence)
elements[i];
                if
(seq.IsRepetitive()) {

result.Add(seq);
                } else
{

result.Add(new
Sequence(true, seq));
                }
            }
            return
result;
        }

        /**
         * Returns a
string representation
of this object.
         *
         * @return a
string representation
of this object
         */
        public
override string
ToString() {
            return
ToString(null);
        }

        /**
         * Returns a
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
seq;


buffer.Append("{");
```

```
            for (int i
= 0; i <
elements.Count; i++) {
                seq =
(Sequence)
elements[i];

buffer.Append("\n  ");

buffer.Append(seq.ToSt
ring(tokenizer));
            }

buffer.Append("\n}");
            return
buffer.ToString();
        }



        /**
         * A token
sequence. This class
contains a list of
token ids.
         * It is
immutable after
creation, meaning that
no changes
         * will be
made to an instance
after creation.
         *

         * @version
1.0
         */
        private class
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 empty token
sequence. The repeat
flag
             * will be
set to false.
             */
            public
Sequence() {

this.repeat = false;

this.tokens = new
ArrayList(0);
            }

            /**
             * Creates
a new token sequence
with a single token.
             *
             * @param
repeat         the
repeat flag value
             * @param
token          the
token to add
             */
            public
Sequence(bool repeat,
int token) {

this.repeat = false;

this.tokens = new
ArrayList(1);

this.tokens.Add(token)
;
            }

            /**
             * 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
tokens to copy
```

```
            * @param
seq            the
sequence to copy
            */
           public
Sequence(int length,
Sequence seq) {

this.repeat =
seq.repeat;

this.tokens = new
ArrayList(length);
                if
(seq.Length() <
length) {

length = seq.Length();
                }
                for
(int i = 0; i <
length; i++) {

tokens.Add(seq.tokens[
i]);
                }
           }

           /**
            * Creates
a new token sequence
that is a duplicate of
            * another
sequence. The new
value of the repeat
flag will
            * be used
however.
            *
            * @param
repeat         the new
repeat flag value
            * @param
seq            the
sequence to copy
            */
           public
Sequence(bool repeat,
Sequence seq) {

this.repeat = repeat;

this.tokens =
seq.tokens;
           }

           /**
            * Returns
the length of the
token sequence.
            *
            * @return
the number of tokens
in the sequence
            */
           public int
Length() {
                return
tokens.Count;
           }

           /**
            * Returns
a token at a specified
position in the
sequence.
            *
            * @param
pos            the
sequence position
            *
            * @return
the token id found, or
null
            */
           public
object GetToken(int
pos) {
                if
(pos >= 0 && pos <
tokens.Count) {

return tokens[pos];
                } else
{

return null;
                }
           }

           /**
            * Checks
if this sequence is
equal to another
object.
            * Only
token sequences with
the same tokens in the
same
            * order
will be considered
equal. The repeat flag
will be
            *
disregarded.
            *
            * @param
obj            the
object to compare with
            *
            * @return
true if the objects
are equal, or
            *
false otherwise
            */
           public
override bool
Equals(object obj) {
                if
(obj is Sequence) {

return
Equals((Sequence)
obj);
                } else
{

return false;
                }
           }

           /**
            * Checks
if this sequence is
equal to another
sequence.
            * Only
sequences with the
same tokens in the
same order
            * will be
considered equal. The
repeat flag will be
            *
disregarded.
            *
            * @param
seq            the
sequence to compare
with
            *
            * @return
true if the sequences
are equal, or
            *
false otherwise
            */
           public
bool Equals(Sequence
seq) {
                if
(tokens.Count !=
seq.tokens.Count) {

return false;
                }
                for
(int i = 0; i <
tokens.Count; i++) {
                    if
(!tokens[i].Equals(seq
.tokens[i])) {

return false;
                    }
                }
                return
true;
           }

           /**
            * Returns
a hash code for this
object.
            *
            * @return
a hash code for this
object
            */
           public
override int
GetHashCode() {
                return
tokens.Count.GetHashCo
de();
           }

           /**
            * Checks
if this token sequence
starts with the tokens
from
            * another
sequence. If the other
sequence is longer
than this
            *
sequence, this method
will always return
false.
            *
            * @param
seq            the
token sequence to
check
            *
            * @return
true if this sequence
starts with the other,
or
            *
false otherwise
            */
           public
bool
StartsWith(Sequence
seq) {
                if
(Length() <
seq.Length()) {

return false;
                }
                for
(int i = 0; i <
seq.tokens.Count; i++)
{
                    if
(!tokens[i].Equals(seq
.tokens[i])) {

return false;
                    }
                }
                return
true;
            }

            /**
             * Checks
if this token sequence
is repetitive. A
repetitive
             * token
sequence is one with
the repeat flag set.
             *
             * @return
true if this token
sequence is
repetitive, or
             *
false otherwise
             */
            public
bool IsRepetitive() {
                return
repeat;
            }

            /**
             * Checks
if the next token(s)
in the parser matches
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
id;
                for
(int i = 0; i <
tokens.Count; i++) {
                    id
= (int) tokens[i];

token =
parser.PeekToken(i);
                    if
(token == null ||
token.Id != id) {

return false;
                    }
                }
                return
true;
            }

            /**
             * Checks
if the next token(s)
in the parser matches
this
             * token
sequence.
             *
             * @param
parser         the
parser to check
             * @param
length         the
maximum number of
tokens to check
             *
             * @return
true if the next
tokens are in the
sequence, or
             *
false otherwise
             */
            public
bool IsNext(Parser
parser, int length) {
                Token
token;
                int
id;

                if
(length >
tokens.Count) {

length = tokens.Count;
                }
                for
(int i = 0; i <
length; i++) {
                    id
= (int) tokens[i];

token =
parser.PeekToken(i);
                    if
(token == null ||
token.Id != id) {

return false;
                    }
                }
                return
true;
            }

            /**
             * Returns
a string
representation of this
object.
             *
             * @return
a string
representation of this
object
             */
            public
override string
ToString() {
                return
ToString(null);
            }

            /**
             * Returns
a 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();
                string
str;
                int
id;

                if
(tokenizer == null) {

buffer.Append(tokens.T
oString());
                } else
{

buffer.Append("[");

for (int i = 0; i <
tokens.Count; i++) {

id = (int) tokens[i];

str =
tokenizer.GetPatternDe
scription(id);

if (i > 0) {

buffer.Append(" ");

}

buffer.Append(str);
                    }

buffer.Append("]");
                }
                if
(repeat) {

buffer.Append(" *");
                }
                return
buffer.ToString();
            }

            /**
             * Creates
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
the concatenated token
sequence
             */
            public
Sequence Concat(int
length, Sequence seq)
{

Sequence  res = new
Sequence(length,
this);

                if
(seq.repeat) {

res.repeat = true;
                }
                length
-= this.Length();
                if
(length >
seq.Length()) {

res.tokens.AddRange(se
q.tokens);
                } else
{

for (int i = 0; i <
length; i++) {

res.tokens.Add(seq.tok
ens[i]);
                    }
                }
                return
res;
            }

            /**
             * Creates
a new token sequence
that is a subsequence
of
             * this
one.
             *
             * @param
start          the
subsequence start
position
             *
             * @return
the new token
subsequence
             */
            public
Sequence
Subsequence(int start)
{

Sequence  res = new
Sequence(Length(),
this);
```

```
                while
(start > 0 &&
res.tokens.Count > 0)
{

res.tokens.RemoveAt(0)
;

start--;
                }
                return
res;
            }
        }
    }
}

/*
 * Node.cs
 */

using
System.Collections;
using System.IO;

namespace Core.Library
{

    /**
     * An abstract
parse tree node. This
class is inherited by
all
     * nodes in the
parse tree, i.e. by
the token and
production
     * classes.
     *

     *
     */
    public abstract
class Node {

        /**
         * The parent
node.
         */
        private Node
parent = null;

        /**
         * The
computed node values.
         */
        private
ArrayList values =
null;

        /**
```

```
         * Checks if
this node is hidden,
i.e. if it should not
be
         * visible
outside the parser.
         *
         * @return
true if the node
should be hidden, or
         *
false otherwise
         */
        internal
virtual bool
IsHidden() {
            return
false;
        }

        /**
         * The node
type id property
(read-only). This
value is set as
         * a unique
identifier for each
type of node, in order
to
         * simplify
later identification.
         *
         *
         */
        public
abstract int Id {
            get;
        }

        /**
         * Returns the
node type id. This
value is set as a
unique
         * identifier
for each type of node,
in order to simplify
         * later
identification.
         *
         * @return the
node type id
         *
         * @see #Id
         *
         * @deprecated
Use the Id property
instead.
         */
        public virtual
int GetId() {
            return Id;
```

```
        }

        /**
         * The node
name property (read-
only).
         *
         *
         */
        public
abstract string Name {
            get;
        }

        /**
         * Returns the
node name.
         *
         * @return the
node name
         *
         * @see #Name
         *
         * @deprecated
Use the Name property
instead.
         */
        public virtual
string GetName() {
            return
Name;
        }

        /**
         * The line
number property of the
first character in
this
         * node (read-
only). If the node has
child elements, this
         * value will
be fetched from the
first child.
         *
         *
         */
        public virtual
int StartLine {
            get {
                int
line;

                for
(int i = 0; i < Count;
i++) {

line =
this[i].StartLine;
                    if
(line >= 0) {
```

```
return line;
                    }
                }
                return
-1;
            }
        }

        /**
         * The line
number of the first
character in this
node. If the
         * node has
child elements, this
value will be fetched
from
         * the first
child.
         *
         * @return the
line number of the
first character, or
         *        -1
if not applicable
         *
         * @see
#StartLine
         *
         * @deprecated
Use the StartLine
property instead.
         */
        public virtual
int GetStartLine() {
            return
StartLine;
        }

        /**
         * The column
number property of the
first character in
this
         * node (read-
only). If the node has
child elements, this
         * value will
be fetched from the
first child.
         *
         *
         */
        public virtual
int StartColumn {
            get {
                int
col;
```

```
                for
(int i = 0; i < Count;
i++) {

col =
this[i].StartColumn;
                    if
(col >= 0) {

return col;
                    }
                }
                return
-1;
            }
        }

        /**
         * The column
number of the first
character in this
node. If
         * the node
has child elements,
this value will be
fetched
         * from the
first child.
         *
         * @return the
column number of the
first token character,
or
         *        -1
if not applicable
         *
         * @see
#StartColumn
         *
         * @deprecated
Use the StartColumn
property instead.
         */
        public virtual
int GetStartColumn() {
            return
StartColumn;
        }

        /**
         * The line
number property of the
last character in this
node
         * (read-
only). If the node has
child elements, this
value
         * will be
fetched from the last
child.
         *
```

```
         *
         */
        public virtual
int EndLine {
            get {
                int
line;

                for
(int i = Count - 1; i
>= 0; i--) {

line =
this[i].EndLine;
                    if
(line >= 0) {

return line;
                    }
                }
                return
-1;
            }
        }

        /**
         * The line
number of the last
character in this
node. If the
         * node has
child elements, this
value will be fetched
from
         * the last
child.
         *
         * @return the
line number of the
last token character,
or
         *        -1
if not applicable
         *
         * @see
#EndLine
         *
         * @deprecated
Use the EndLine
property instead.
         */
        public virtual
int GetEndLine() {
            return
EndLine;
        }

        /**
         * The column
number property of the
last character in this
```

```
         * node (read-
only). If the node has
child elements, this
         * value will
be fetched from the
last child.
         *
         *
         */
        public virtual
int EndColumn {
            get {
                int
col;

                for
(int i = Count - 1; i
>= 0; i--) {

col =
this[i].EndColumn;
                    if
(col >= 0) {

return col;
                    }
                }
                return
-1;
            }
        }

        /**
         * The column
number of the last
character in this
node. If
         * the node
has child elements,
this value will be
fetched
         * from the
last child.
         *
         * @return the
column number of the
last token character,
or
         *        -1
if not applicable
         *
         * @see
#EndColumn
         *
         * @deprecated
Use the EndColumn
property instead.
         */
        public virtual
int GetEndColumn() {
            return
EndColumn;
```

```
    }

    /**
     * The parent
node property (read-
only).
     *
     *
     */
    public Node
Parent {
      get {
        return
parent;
      }
      set {

this.parent = value;
      }
    }

    /**
     * Returns the
parent node.
     *
     * @return the
parent parse tree node
     *
     * @see
#Parent
     *
     * @deprecated
Use the Parent
property instead.
     */
    public Node
GetParent() {
      return
Parent;
    }

    /**
     * Sets the
parent node.
     *
     * @param
parent        the new
parent node
     */
    public void
SetParent(Node parent)
{
      Parent =
parent;
    }

    /**
     * The child
node count property
(read-only).
     *
     *
```

```
     */
    public virtual
int Count {
      get {
        return
0;
      }
    }

    /**
     * Returns the
number of child nodes.
     *
     * @return the
number of child nodes
     *
     * @deprecated
Use the Count property
instead.
     */
    public virtual
int GetChildCount() {
      return
Count;
    }

    /**
     * Returns the
number of descendant
nodes.
     *
     * @return the
number of descendant
nodes
     *
     *
     */
    public int
GetDescendantCount() {
      int  count
= 0;

      for (int i
= 0; i < Count; i++) {
        count
+= 1 +
this[i].GetDescendantC
ount();
      }
      return
count;
    }

    /**
     * The child
node index (read-
only).
     *
     * @param
index         the
child index, 0 <=
index < Count
```

```
     *
     * @return the
child node found, or
     *
null if index out of
bounds
     *
     *
     */
    public virtual
Node this[int index] {
      get {
        return
null;
      }
    }

    /**
     * Returns the
child node with the
specified index.
     *
     * @param
index         the
child index, 0 <=
index < count
     *
     * @return the
child node found, or
     *
null if index out of
bounds
     *
     * @deprecated
Use the class indexer
instead.
     */
    public virtual
Node GetChildAt(int
index) {
      return
this[index];
    }

    /**
     * The node
values property. This
property provides
direct
     * access to
the list of computed
values associated with
this
     * node during
analysis. Note that
setting this property
to
     * null will
remove all node
values. Any operation
on the
```

```
     * value array
list is allowed and is
immediately reflected
     * through the
various value reading
and manipulation
methods.
     *
     *
     */
    public
ArrayList Values {
      get {
        if
(values == null) {

values = new
ArrayList();
        }
        return
values;
      }
      set {

this.values = value;
      }
    }

    /**
     * Returns the
number of computed
values associated with
this
     * node. Any
number of values can
be associated with a
node
     * through
calls to AddValue().
     *
     * @return the
number of values
associated with this
node
     *
     * @see
#Values
     *
     * @deprecated
Use the Values and
Values.Count
properties
     *
instead.
     */
    public int
GetValueCount() {
      if (values
== null) {
        return
0;
      } else {
```

```
                return
values.Count;
            }
        }

        /**
         * Returns a
computed value of this
node, if previously
set. A
         * value may
be used for storing
intermediate results
in the
         * parse tree
during analysis.
         *
         * @param pos
the value position, 0
<= pos < count
         *
         * @return the
computed node value,
or
         *
null if not set
         *
         * @see
#Values
         *
         * @deprecated
Use the Values
property and it's
array indexer
         *
instead.
         */
        public object
GetValue(int pos) {
            return
Values[pos];
        }

        /**
         * 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
         *
null if no values have
been set
         *
         * @see
#Values
         *
         * @deprecated
Use the Values
property instead. Note
that the
         *     Values
property will never be
null, but possibly
empty.
         */
        public
ArrayList
GetAllValues() {
            return
values;
        }

        /**
         * Adds a
computed value to this
node. The computed
value may
         * be used for
storing intermediate
results in the parse
tree
         * during
analysis.
         *
         * @param
value          the
node value
         *
         * @see
#Values
         *
         * @deprecated
Use the Values
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.
         *
         * @param
values         the
vector with node
values
         *
         * @see
#Values
         *
         * @deprecated
Use the Values
property and the
Values.AddRange
         *     method
instead.
         */
        public void
AddValues(ArrayList
values) {
            if (values
!= null) {

Values.AddRange(values
);
            }
        }

        /**
         * Removes all
computed values stored
in this node.
         *
         * @see
#Values
         *
         * @deprecated
Use the Values
property and the
Values.Clear
         *     method
instead. Alternatively
the Values property
can
         *     be set
to null.
         */
        public void
RemoveAllValues() {
            values =
null;
        }

        /**
         * Prints this
node and all subnodes
to the specified
output
         * stream.
         *
         * @param
output         the
output stream to use
         */
        public void
PrintTo(TextWriter
output) {

PrintTo(output, "");

output.Flush();
        }

        /**
         * Prints this
node and all subnodes
to the specified
output
         * stream.
         *
         * @param
output         the
output stream to use
         * @param
indent         the
indentation string
         */
        private void
PrintTo(TextWriter
output, string indent)
{

output.WriteLine(inden
t + ToString());
            indent =
indent + "  ";
            for (int i
= 0; i < Count; i++) {

this[i].PrintTo(output
, indent);
            }
        }
    }
}

/*
 * ParseException.cs
 */

using System;
using
System.Collections;
using System.Text;

namespace Core.Library
{

    /**
     * A parse
exception.
     *

     *
     */
    public class
ParseException :
Exception {

        /**
         * The error
type enumeration.
         */
        public enum
ErrorType {

            /**
             * The
internal error type is
only used to signal an
error
             * that is
a result of a bug in
the parser or
tokenizer
             * code.
             */
            INTERNAL,

            /**
             * The I/O
error type is used for
stream I/O errors.
             */
            IO,

            /**
             * The
unexpected end of file
error type is used
when end
             * of file
is encountered instead
of a valid token.
             */

UNEXPECTED_EOF,

            /**
             * The
unexpected character
error type is used
when a
             *
character is read that
isn't handled by one
of the
             * token
patterns.
             */

UNEXPECTED_CHAR,

            /**
             * The
unexpected token error
type is used when
another
             * token
than the expected one
is encountered.
             */

UNEXPECTED_TOKEN,

            /**
             * The
invalid token error
type is used when a
token
             * pattern
with an error message
is matched. The
             *
additional information
provided should
contain the
             * error
message.
             */

INVALID_TOKEN,

            /**
             * The
analysis error type is
used when an error is
             *
encountered in the
analysis. The
additional information
             *
provided should
contain the error
message.
             */
            ANALYSIS
        }

        /**
         * The error
type.
         */
        private
ErrorType type;

        /**
         * The
additional information
string.
         */
        private string
info;

        /**
         * The
additional details
information. This
variable is only
         * used for
unexpected token
errors.
         */
        private
ArrayList details;

        /**
         * The line
number.
         */
        private int
line;

        /**
         * The column
number.
         */
        private int
column;

        /**
         * Creates a
new parse exception.
         *
         * @param type
the parse error type
         * @param info
the additional
information
         * @param line
the line number, or -1
for unknown
         * @param
column        the
column number, or -1
for unknown
         */
        public
ParseException(ErrorTy
pe type,

string info,

int line,

int column)
            :
this(type, info, null,
line, column) {
        }

        /**
         * Creates a
new parse exception.
This constructor is
only
         * used to
supply the detailed
information array,
which is
         * only used
for expected token
errors. The list then
contains
         *
descriptions of the
expected tokens.
         *
         * @param type
the parse error type
         * @param info
the additional
information
         * @param
details        the
additional detailed
information
         * @param line
the line number, or -1
for unknown
         * @param
column         the
column number, or -1
for unknown
         */
        public
ParseException(ErrorTy
pe type,

string info,

ArrayList details,

int line,

int column) {

            this.type
= type;
            this.info
= info;

this.details =
details;
            this.line
= line;

this.column = column;
        }

        /**
         * The error
type property (read-
only).
         *
         *
         */
        public
ErrorType Type {
            get {
                return
type;
            }
        }

        /**
         * Returns the
error type.
         *
         * @return the
error type
         *
         * @see #Type
         *
         * @deprecated
Use the Type property
instead.
         */
        public
ErrorType
GetErrorType() {
            return
Type;
        }

        /**
         * The
additional error
information property
(read-only).
         *
         *
         */
        public string
Info {
            get {
                return
info;
            }
        }

        /**
         * Returns the
additional error
information.
         *
         * @return the
additional error
information
         *
         * @see #Info
         *
         * @deprecated
Use the Info property
instead.
         */
        public string
GetInfo() {
            return
Info;
        }
        /**
         * The
additional detailed
error information
property
         * (read-
only).
         *
         *
         */
        public
ArrayList Details {
            get {
                return
new
ArrayList(details);
            }
        }

        /**
         * Returns the
additional detailed
error information.
         *
         * @return the
additional detailed
error information
         *
         * @see
#Details
         *
         * @deprecated
Use the Details
property instead.
         */
        public
ArrayList GetDetails()
{
            return
Details;
        }

        /**
         * The line
number property (read-
only). This is the
line
         * number
where the error
occured, or -1 if
unknown.
         *
         *
         */
        public int
Line {
            get {
                return
line;
            }
        }
        /**
         * Returns the
line number where the
error occured.
         *
         * @return the
line number of the
error, or
         *         -1
if unknown
         *
         * @see #Line
         *
         * @deprecated
Use the Line property
instead.
         */
        public int
GetLine() {
            return
Line;
        }

        /**
         * The column
number property (read-
only). This is the
column
         * number
where the error
occured, or -1 if
unknown.
         *
         *
         */
        public int
Column {
            get {
                return
column;
            }
        }

        /**
         * Returns the
column number where
the error occured.
         *
         * @return the
column number of the
error, or
         *         -1
if unknown
         *
         * @see
#Column
         *
         * @deprecated
Use the Column
property instead.
         */
        public int
GetColumn() {
            return
column;
        }

        /**
         * The message
property (read-only).
This property contains
         * the
detailed exception
error message,
including line and
         * column
numbers when
available.
         *
         * @see
#ErrorMessage
         */
        public
override string
Message {
            get{

StringBuilder  buffer
= new StringBuilder();

                // Add
error description

buffer.Append(ErrorMes
sage);

                // Add
line and column
                if
(line > 0 && column >
0) {

buffer.Append(", on
line: ");

buffer.Append(line);

buffer.Append("
column: ");

buffer.Append(column);
                }

                return
buffer.ToString();
            }
        }

        /**
         * Returns a
default error message.
         *
         * @return a
default error message
         *
         * @see
#Message
         *
         * @deprecated
Use the Message
property instead.
         */
        public string
GetMessage() {
            return
Message;
        }

        /**
         * The error
message property
(read-only). This
property
         * contains
all the information
available, except for
the line
         * and column
number information.
         *
         * @see
#Message
         *
         *
         */
        public string
ErrorMessage {
            get {

StringBuilder  buffer
= new StringBuilder();

                // Add
type and info
                switch
(type) {
                case
ErrorType.IO:

buffer.Append("I/O
error: ");

buffer.Append(info);

break;
                case
ErrorType.UNEXPECTED_E
OF:

buffer.Append("unexpec
ted end of file");

break;
```

```
                case
ErrorType.UNEXPECTED_C
HAR:

buffer.Append("unexpec
ted character '");

buffer.Append(info);

buffer.Append("'");

break;
                case
ErrorType.UNEXPECTED_T
OKEN:

buffer.Append("unexpec
ted token ");

buffer.Append(info);
                    if
(details != null) {

buffer.Append(",
expected ");

if (details.Count > 1)
{

buffer.Append("one of
");

}

buffer.Append(GetMessa
geDetails());
                    }

break;
                case
ErrorType.INVALID_TOKE
N:

buffer.Append(info);

break;
                case
ErrorType.ANALYSIS:

buffer.Append(info);

break;

default:

buffer.Append("interna
l error");
                    if
(info != null) {

buffer.Append(": ");
```

```
buffer.Append(info);
                    }

break;
                }

                return
buffer.ToString();
            }
        }

        /**
         * Returns the
error message. This
message will contain
all the
         * information
available, except for
the line and column
number
         *
information.
         *
         * @return the
error message
         *
         * @see
#ErrorMessage
         *
         * @deprecated
Use the ErrorMessage
property instead.
         */
        public string
GetErrorMessage() {
            return
ErrorMessage;
        }

        /**
         * Returns a
string containing all
the detailed
information in
         * a list. The
elements are separated
with a comma.
         *
         * @return the
detailed information
string
         */
        private string
GetMessageDetails() {

StringBuilder  buffer
= new StringBuilder();

            for (int i
= 0; i <
details.Count; i++) {
```

```
                if (i
> 0) {

buffer.Append(", ");
                    if
(i + 1 ==
details.Count) {

buffer.Append("or ");
                    }
                }

buffer.Append(details[
i]);
            }

            return
buffer.ToString();
        }
    }
}

/*
 * Parser.cs
 *
 */

using System;
using
System.Collections;
using
System.Collections.Gen
eric;
using System.IO;
using System.Text;

namespace Core.Library
{

    /**
     * A base parser
class. This class
provides the standard
parser
     * interface, as
well as token
handling.
     *



     */
    public abstract
class Parser {

        /**
         * The parser
initialization flag.
         */
        private bool
initialized = false;

        /**
```

```
         * The
production output out
of
RecursiveDescentParser
.
         */
        public
SyntaxProductions
production = new
SyntaxProductions();

        /**
         * Get the
Production set of
production.
         */
        public string
GetRecursiveProduction
()
        {
            return
("Enter:
<StartProgram>\n" +
production.GetRecursiv
eProductions());
        }

        public int
GetLastProductionCode(
)
        {
            return
production.GetLastProd
uctionCode();
        }

        public string
GetLastProductionState
()
        {
            return
production.GetLastProd
uctionState();
        }

        public
List<string>
GetAllProductionState(
)
        {
            return
production.GetAllProdu
ctionState();
        }

        public
List<int>
GetAllProductionCode()
        {
            return
production.GetAllProdu
ctionCode();
        }

        /**
         * The
tokenizer to use.
         */
        private
Tokenizer tokenizer;

        /**
         * The
analyzer to use for
callbacks.
         */
        private
Analyzer analyzer;

        /**
         * The list of
production patterns.
         */
        private
ArrayList patterns =
new ArrayList();

        /**
         * The map
with production
patterns and their
id:s. This map
         * contains
the production
patterns indexed by
their id:s.
         */
        private
Hashtable patternIds =
new Hashtable();

        /**
         * The list of
buffered tokens. This
list will contain
tokens that
         * have been
read from the
tokenizer, but not yet
consumed.
         */
        private
ArrayList tokens = new
ArrayList();

        /**
         * The error
log. All parse errors
will be added to this
log as
         * the parser
attempts to recover
from the error. If the
error
         * count is
higher than zero (0),
this log will be
thrown as the
         * result from
the parse() method.
         */
        private
ParserLogException
errorLog = new
ParserLogException();

        /**
         * The error
recovery counter. This
counter is initially
set to a
         * negative
value to indicate that
no error requiring
recovery
         * has been
encountered. When a
parse error is found,
the counter
         * is set to
three (3), and is then
decreased by one for
each
         * correctly
read token until it
reaches zero (0).
         */
        private int
errorRecovery = -1;

        /**
         * Creates a
new parser.
         *
         * @param
input          the
input stream to read
from
         *
         * @throws
ParserCreationExceptio
n if the tokenizer
couldn't be
         *
initialized correctly
         *
         *
         */
        internal
Parser(TextReader
input) : this(input,
null) {
        }

        /**
         * Creates a
new parser.
         *
         * @param
input          the
input stream to read
from
         * @param
analyzer       the
analyzer callback to
use
         *
         * @throws
ParserCreationExceptio
n if the tokenizer
couldn't be
         *
initialized correctly
         *
         *
         */
        internal
Parser(TextReader
input, Analyzer
analyzer) {

this.tokenizer =
NewTokenizer(input);

this.analyzer =
(analyzer == null) ?
NewAnalyzer() :
analyzer;
        }

        /**
         * Creates a
new parser.
         *
         * @param
tokenizer       the
tokenizer to use
         */
        internal
Parser(Tokenizer
tokenizer) :
this(tokenizer, null)
{
        }

        /**
         * Creates a
new parser.
         *
         * @param
tokenizer       the
tokenizer to use
         * @param
analyzer        the
analyzer callback to
use
         */
        internal
Parser(Tokenizer
tokenizer, Analyzer
analyzer) {

this.tokenizer =
tokenizer;

this.analyzer =
(analyzer == null) ?
NewAnalyzer() :
analyzer;
        }

        /**
         * Creates a
new tokenizer for this
parser. Can be
overridden by
         * a subclass
to provide a custom
implementation.
         *
         * @param in
the input stream to
read from
         *
         * @return the
tokenizer created
         *
         * @throws
ParserCreationExceptio
n if the tokenizer
couldn't be
         *
initialized correctly
         *
         *
         */
        protected
virtual Tokenizer
NewTokenizer(TextReade
r input) {
            // TODO:
This method should
really be abstract,
but it isn't in this
            //
version due to
backwards
compatibility
requirements.
            return new
Tokenizer(input);
        }

        /**
         * Creates a
new analyzer for this
parser. Can be
overridden by a
         * subclass to
provide a custom
implementation.
         *
         * @return the
analyzer created
         *
         *
         */
        protected
virtual Analyzer
NewAnalyzer() {
            // TODO:
This method should
really be abstract,
but it isn't in this
            //
version due to
backwards
compatibility
requirements.
            return new
Analyzer();
        }

        /**
         * The
tokenizer property
(read-only). This
property contains
         * the
tokenizer in use by
this parser.
         *
         *
         */
        public
Tokenizer Tokenizer {
            get {
                return
tokenizer;
            }
        }

        /**
         * The
analyzer property
(read-only). This
property contains
         * the
analyzer in use by
this parser.
         *
         *
         */
        public
Analyzer Analyzer {
            get {
                return
analyzer;
            }
        }

        /**
         * Returns the
tokenizer in use by
this parser.
         *
         * @return the
tokenizer in use by
this parser
         *
         *
         *
         * @see
#Tokenizer
         *
         * @deprecated
Use the Tokenizer
property instead.
         */
        public
Tokenizer
GetTokenizer() {
            return
Tokenizer;
        }

        /**
         * Returns the
analyzer in use by
this parser.
         *
         * @return the
analyzer in use by
this parser
         *
         *
         *
         * @see
#Analyzer
         *
         * @deprecated
Use the Analyzer
property instead.
         */
        public
Analyzer GetAnalyzer()
{
            return
Analyzer;
        }

        /**
         * Sets the
parser initialized
flag. Normally this
flag is set by
         * the
prepare() method, but
this method allows
further
         *
modifications to it.
         *
         * @param
initialized    the new
initialized flag
         */
        internal void
SetInitialized(bool
initialized) {

this.initialized =
initialized;
        }

        /**
         * Adds a new
production pattern to
the parser. The first
pattern
         * added is
assumed to be the
starting point in the
grammar. The
         * patterns
added may be validated
to some extent.
         *
         * @param
pattern        the
pattern to add
         *
         * @throws
ParserCreationExceptio
n if the pattern
couldn't be
         *
added correctly to the
parser
         */
        public virtual
void
AddPattern(ProductionP
attern pattern) {
            if
(pattern.Count <= 0) {
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_PR
ODUCTION,

pattern.Name,

"no production
alternatives are
present (must have at
" +

"least one)");
            }
            if
(patternIds.ContainsKe
y(pattern.Id)) {
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_PR
ODUCTION,

pattern.Name,

″another pattern with
the same id (″ +
pattern.Id +
                       ″)
has already been
added″);
            }

patterns.Add(pattern);

patternIds.Add(pattern
.Id, pattern);

SetInitialized(false);
        }

        /**
         * Initializes
the parser. All the
added production
patterns will
         * be analyzed
for ambiguities and
errors. This method
also
         * initializes
internal data
structures used during
the parsing.
         *
         * @throws
ParserCreationExceptio
n if the parser
couldn't be
         *
initialized correctly
         */
        public virtual
void Prepare() {
            if
(patterns.Count <= 0)
{
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_PA
RSER,

″no production
patterns have been
added″);
            }
            for (int i
= 0; i <
patterns.Count; i++) {

CheckPattern((Producti
onPattern)
patterns[i]);
            }

SetInitialized(true);
        }

        /**
         * Checks a
production pattern for
completeness. If some
rule
         * in the
pattern referenced an
production pattern not
added
         * to this
parser, a parser
creation exception
will be thrown.
         *
         * @param
pattern        the
production pattern to
check
         *
         * @throws
ParserCreationExceptio
n if the pattern
referenced a
         *
pattern not added to
this parser
         */
        private void
CheckPattern(Productio
nPattern pattern) {
            for (int i
= 0; i <
pattern.Count; i++) {

CheckAlternative(patte
rn.Name, pattern[i]);
            }
        }

        /**
         * Checks a
production pattern
alternative for
completeness.
         * If some
element in the
alternative referenced
a production
         * pattern not
added to this parser,
a parser creation
         * exception
will be thrown.
         *
         * @param name
the name of the
pattern being checked
         * @param alt
the production pattern
alternative
         *
         * @throws
ParserCreationExceptio
n if the alternative
         *
referenced a pattern
not added to this
parser
         */
        private void
CheckAlternative(strin
g name,

ProductionPatternAlter
native alt) {

            for (int i
= 0; i < alt.Count;
i++) {

CheckElement(name,
alt[i]);
            }
        }

        /**
         * Checks a
production pattern
element for
completeness. If
         * the element
references a
production pattern not
added to
         * this
parser, a parser
creation exception
will be thrown.
         *
         * @param name
the name of the
pattern being checked
         * @param elem
the production pattern
element to check
         *
         * @throws
ParserCreationExceptio
n if the element
referenced a
         *
pattern not added to
this parser
         */
        private void
CheckElement(string
name,

ProductionPatternEleme
nt elem) {

            if
(elem.IsProduction()
&& GetPattern(elem.Id)
== null) {
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_PR
ODUCTION,

name,

″an undefined
production pattern id
(″ + elem.Id +
                      ″)
is referenced″);
            }
        }

        /**
         * Resets this
parser for usage with
another input stream.
The
         * associated
tokenizer and analyzer
will also be reset.
This
         * method will
clear all the internal
state and the error
log in
         * the parser.
It is normally called
in order to reuse a
parser
         * and
tokenizer pair with
```

```
multiple input
streams, thereby
         * avoiding
the cost of re-
analyzing the grammar
structures.
         *
         * @param
input          the new
input stream to read
         *
         * @see
Tokenizer#Reset
         * @see
Analyzer#Reset
         *
         *
         */
        public void
Reset(TextReader
input) {

this.tokenizer.Reset(i
nput);

this.analyzer.Reset();
        }

        /**
         * Resets this
parser for usage with
another input stream.
The
         * associated
tokenizer will also be
reset and the analyzer
         * replaced.
This method will clear
all the internal state
and
         * the error
log in the parser. It
is normally called in
order
         * to reuse a
parser and tokenizer
pair with multiple
input
         * streams,
thereby avoiding the
cost of re-analyzing
the
         * grammar
structures.
         *
         * @param
input          the new
input stream to read
         * @param
analyzer       the new
analyzer callback to
use
         *
         * @see
Tokenizer#Reset
         *
         * @since 1.6
         */
        public void
Reset(TextReader
input, Analyzer
analyzer) {

this.tokenizer.Reset(i
nput);

this.analyzer =
analyzer;
        }

        /**
         * Parses the
token stream and
returns a parse tree.
This
         * method will
call Prepare() if not
previously called. It
         * will also
call the Reset()
method, to make sure
that only
         * the
Tokenizer.Reset()
method must be
explicitly called in
         * order to
reuse a parser for
multiple input
streams. In case
         * of a parse
error, the parser will
attempt to recover and
         * throw all
the errors found in a
parser log exception
in the
         * end.
         *
         * @return the
parse tree
         *
         * @throws
ParserCreationExceptio
n if the parser
couldn't be
         *
initialized correctly
         * @throws
ParserLogException if
the input couldn't be
parsed
         *
correctly
         *
         * @see
#Prepare
         * @see #Reset
         * @see
Tokenizer#Reset
         */
        public Node
Parse() {
            Node  root
= null;

            //
Initialize parser
            if
(!initialized) {

Prepare();
            }

this.tokens.Clear();

this.errorLog = new
ParserLogException();

this.errorRecovery = -
1;

            // Parse
input
            try {
                root =
ParseStart();
            } catch
(ParseException e) {

AddError(e, true);
            }

            // Check
for errors
            if
(errorLog.Count > 0) {
                throw
errorLog;
            }

            return
root;
        }

        /**
         * Parses the
token stream and
returns a parse tree.
         *
         * @return the
parse tree
         *
         * @throws
ParseException if the
input couldn't be
parsed
         *
correctly
         */
        protected
abstract Node
ParseStart();

        /**
         * Factory
method to create a new
production node. This
method
         * can be
overridden to provide
other production
implementations
         * than the
default one.
         *
         * @param
pattern        the
production pattern
         *
         * @return the
new production node
         *
         *
         */
        protected
virtual Production
NewProduction(Producti
onPattern pattern) {
            return
analyzer.NewProduction
(pattern);
        }

        /**
         * Adds an
error to the error
log. If the parser is
in error
         * recovery
mode, the error will
not be added to the
log. If the
         * recovery
flag is set, this
method will set the
error recovery
         * counter
thus enter error
recovery mode. Only
lexical or
         * syntactical
errors require
recovery, so this flag
shouldn't be
         * set
otherwise.
```

```
         *
         * @param e
the error to add
         * @param
recovery       the
recover flag
         */
        internal void
AddError(ParseExceptio
n e, bool recovery) {
            if
(errorRecovery <= 0) {

errorLog.AddError(e);
            }
            if
(recovery) {

errorRecovery = 3;
            }
        }

        /**
         * Returns the
production pattern
with the specified id.
         *
         * @param id
the production pattern
id
         *
         * @return the
production pattern
found, or
         *
null if non-existent
         */
        internal
ProductionPattern
GetPattern(int id) {
            return
(ProductionPattern)
patternIds[id];
        }

        /**
         * Returns the
production pattern for
the starting
production.
         *
         * @return the
start production
pattern, or
         *
null if no patterns
have been added
         */
        internal
ProductionPattern
GetStartPattern() {
            if
(patterns.Count <= 0)
{
                return
null;
            } else {
                return
(ProductionPattern)
patterns[0];
            }
        }

        /**
         * Returns the
ordered set of
production patterns.
         *
         * @return the
ordered set of
production patterns
         */
        internal
ICollection
GetPatterns() {
            return
patterns;
        }

        /**
         * Handles the
parser entering a
production. This
method calls the
         * appropriate
analyzer callback if
the node is not
hidden. Note
         * that this
method will not call
any callback if an
error
         * requiring
recovery has ocurred.
         *
         * @param node
the parse tree node
         */
        internal void
EnterNode(Node node) {
            if
(!node.IsHidden() &&
errorRecovery < 0) {
                try {

analyzer.Enter(node);
                }
catch (ParseException
e) {

AddError(e, false);
                }
            }
        }

        /**
         * Handles the
parser leaving a
production. This
method calls the
         * appropriate
analyzer callback if
the node is not
hidden, and
         * returns the
result. Note that this
method will not call
any
         * callback if
an error requiring
recovery has ocurred.
         *
         * @param node
the parse tree node
         *
         * @return the
parse tree node, or
         *
null if no parse tree
should be created
         */
        internal Node
ExitNode(Node node) {
            if
(!node.IsHidden() &&
errorRecovery < 0) {
                try {

return
analyzer.Exit(node);
                }
catch (ParseException
e) {

AddError(e, false);
                }
            }
            return
node;
        }

        /**
         * Handles the
parser adding a child
node to a production.
This
         * method
calls the appropriate
analyzer callback.
Note that this
         * method will
not call any callback
if an error requiring
         * recovery
has ocurred.
         *
         * @param node
the parent parse tree
node
         * @param
child          the
child parse tree node,
or null
         */
        internal void
AddNode(Production
node, Node child) {
            if
(errorRecovery >= 0) {
                // Do
nothing
            } else if
(node.IsHidden()) {

node.AddChild(child);
            } else if
(child != null &&
child.IsHidden()) {
                for
(int i = 0; i <
child.Count; i++) {

AddNode(node,
child[i]);
                }
            } else {
                try {

analyzer.Child(node,
child);
                }
catch (ParseException
e) {

AddError(e, false);
                }
            }
        }

        /**
         * Reads and
consumes the next
token in the queue. If
no token
         * was
available for
consumation, a parse
error will be
         * thrown.
         *
         * @return the
token consumed
         *
         * @throws
ParseException if the
input stream couldn't
be read or
```

```
         *
parsed correctly
         */
        internal Token
NextToken() {
            Token
token = PeekToken(0);

            if (token
!= null) {

tokens.RemoveAt(0);
                return
token;
            } else {
                throw
new ParseException(

ParseException.ErrorTy
pe.UNEXPECTED_EOF,

null,

tokenizer.GetCurrentLi
ne(),

tokenizer.GetCurrentCo
lumn());
            }
        }

        /**
         * 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
         */
        internal Token
NextToken(int id) {
            Token
token = NextToken();
            ArrayList
list;

            if
(token.Id == id) {
                if
(errorRecovery > 0) {

errorRecovery--;
                }
                return
token;
            } else {
                list =
new ArrayList(1);

list.Add(tokenizer.Get
PatternDescription(id)
);
                throw
new ParseException(

ParseException.ErrorTy
pe.UNEXPECTED_TOKEN,

token.ToShortString(),

list,

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
         *
null if no more tokens
in the queue
         */
        internal Token
PeekToken(int steps) {
            Token
token;

            while
(steps >=
tokens.Count) {
                try {

token =
tokenizer.Next();
                    if
(token == null) {

return null;
                    }
else {

tokens.Add(token);
                    }
                }
catch (ParseException
e) {

AddError(e, true);
                }
            }
            return
(Token) tokens[steps];
        }

        /**
         * 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 <
patterns.Count; i++) {

buffer.Append(ToString
((ProductionPattern)
patterns[i]));

buffer.Append("\n");
            }
            return
buffer.ToString();
        }

        /**
         * Returns a
string representation
of a production
pattern.
         *
         * @param prod
the production pattern
         *
         * @return a
detailed string
representation of the
pattern
         */
        private string
ToString(ProductionPat
tern prod) {

StringBuilder  buffer
= new StringBuilder();

StringBuilder  indent
= 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("= ");

indent.Append("| ");
            for (i =
0; i < prod.Count;
i++) {
                if (i
> 0) {
```

```
buffer.Append(indent);
                }

buffer.Append(ToString
(prod[i]));

buffer.Append("\n");
            }
            for (i =
0; i < prod.Count;
i++) {
                set =
prod[i].LookAhead;
                if
(set.GetMaxLength() >
1) {

buffer.Append("Using
");

buffer.Append(set.GetM
axLength());

buffer.Append(" token
look-ahead for
alternative ");

buffer.Append(i + 1);

buffer.Append(": ");

buffer.Append(set.ToSt
ring(tokenizer));

buffer.Append("\n");
                }
            }
            return
buffer.ToString();
        }

        /**
         * Returns a
string representation
of a production
pattern
         *
alternative.
         *
         * @param alt
the production pattern
alternative
         *
         * @return a
detailed string
representation of the
alternative
         */
        private string
ToString(ProductionPat
ternAlternative alt) {
```

```
StringBuilder  buffer
= new StringBuilder();

            for (int i
= 0; i < alt.Count;
i++) {
                if (i
> 0) {

buffer.Append(" ");
                }

buffer.Append(ToString
(alt[i]));
            }
            return
buffer.ToString();
        }

        /**
         * Returns a
string representation
of a production
pattern
         * element.
         *
         * @param elem
the production pattern
element
         *
         * @return a
detailed string
representation of the
element
         */
        private string
ToString(ProductionPat
ternElement elem) {

StringBuilder  buffer
= new StringBuilder();
            int
min = elem.MinCount;
            int
max = elem.MaxCount;

            if (min ==
0 && max == 1) {

buffer.Append("[");
            }
            if
(elem.IsToken()) {

buffer.Append(GetToken
Description(elem.Id));
            } else {

buffer.Append(GetPatte
rn(elem.Id).Name);
            }
```

```
            if (min ==
0 && max == 1) {

buffer.Append("]");
            } else if
(min == 0 && max ==
Int32.MaxValue) {

buffer.Append("*");
            } else if
(min == 1 && max ==
Int32.MaxValue) {

buffer.Append("+");
            } else if
(min != 1 || max != 1)
{

buffer.Append("{");

buffer.Append(min);

buffer.Append(",");

buffer.Append(max);

buffer.Append("}");
            }
            return
buffer.ToString();
        }

        /**
         * Returns a
token description for
a specified token.
         *
         * @param
token          the
token to describe
         *
         * @return the
token description
         */
        internal
string
GetTokenDescription(in
t token) {
            if
(tokenizer == null) {
                return
"";
            } else {
                return
tokenizer.GetPatternDe
scription(token);
            }
        }
    }
}

/*
```

```
 *
ParserCreationExceptio
n.cs
 */

using System;
using
System.Collections;
using System.Text;

namespace Core.Library
{

    /**
     * A parser
creation exception.
This exception is used
for signalling
     * an error in the
token or production
patterns, making it
impossible
     * to create a
working parser or
tokenizer.
     *

     *
     */
    public class
ParserCreationExceptio
n : Exception {

        /**
         * The error
type enumeration.
         */
        public enum
ErrorType {

            /**
             * The
internal error type is
only used to signal an
             * error
that is a result of a
bug in the parser or
             *
tokenizer code.
             */
            INTERNAL,

            /**
             * The
invalid parser error
type is used when the
parser
             * as such
is invalid. This error
is typically caused by
```

```
             * using a
parser without any
patterns.
             */

INVALID_PARSER,

            /**
             * The
invalid token error
type is used when a
token
             * pattern
is erroneous. This
error is typically
caused
             * by an
invalid pattern type
or an erroneous
regular
             *
expression.
             */

INVALID_TOKEN,

            /**
             * The
invalid production
error type is used
when a
             *
production pattern is
erroneous. This error
is
             *
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.
             */

INFINITE_LOOP,
```

```
            /**
             * The
inherent ambiguity
error type is used
when the set
             * of
production patterns
(i.e. the grammar)
contains
             *
ambiguities that
cannot be resolved.
             */

INHERENT_AMBIGUITY
        }

        /**
         * The error
type.
         */
        private
ErrorType type;

        /**
         * The token
or production pattern
name. This variable is
only
         * set for
some error types.
         */
        private string
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
new parser creation
exception.
```

```
         *
         * @param type
the parse error type
         * @param info
the additional error
information
         */
        public
ParserCreationExceptio
n(ErrorType type,

String info)
            :
this(type, null, info)
{
        }

        /**
         * Creates a
new parser creation
exception.
         *
         * @param type
the parse error type
         * @param name
the token or
production pattern
name
         * @param info
the additional error
information
         */
        public
ParserCreationExceptio
n(ErrorType type,

String name,

String info)
            :
this(type, name, info,
null) {
        }

        /**
         * Creates a
new parser creation
exception.
         *
         * @param type
the parse error type
         * @param name
the token or
production pattern
name
         * @param info
the additional error
information
         * @param
details        the
error details list
         */
```

```
        public
ParserCreationExceptio
n(ErrorType type,

String name,

String info,

ArrayList details) {

            this.type
= type;
            this.name
= name;
            this.info
= info;

this.details =
details;
        }

        /**
         * The error
type property (read-
only).
         *
         *
         */
        public
ErrorType Type {
            get {
                return
type;
            }
        }

        /**
         * 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
Name {
            get {
                return
name;
            }
        }

        /**
         * Returns the
token or production
name.
         *
         * @return the
token or production
name
         *
         * @see #Name
         *
         * @deprecated
Use the Name property
instead.
         */
        public string
GetName() {
            return
Name;
        }

        /**
         * The
additional error
information property
(read-only).
         *
         *
         */
        public string
Info {
            get {
                return
info;
            }
        }

        /**
         * Returns the
additional error
information.
         *
         * @return the
additional error
information
         *
         * @see #Info
         *
         * @deprecated
Use the Info property
instead.
         */
```

```
        public string
GetInfo() {
            return
Info;
        }

        /**
         * The
detailed error
information property
(read-only).
         *
         *
         */
        public string
Details {
            get {

StringBuilder  buffer
= new StringBuilder();

                if
(details == null) {

return null;
                }
                for
(int i = 0; i <
details.Count; i++) {
                    if
(i > 0) {

buffer.Append(", ");

if (i + 1 ==
details.Count) {

buffer.Append("and ");

}
                    }

buffer.Append(details[
i]);
                }

                return
buffer.ToString();
            }
        }

        /**
         * Returns the
detailed error
information as a
string
         *
         * @return the
detailed error
information
         *
```

```
         * @see
#Details
         *
         * @deprecated
Use the Details
property instead.
         */
        public string
GetDetails() {
            return
Details;
        }

        /**
         * The message
property (read-only).
This property contains
         * the
detailed exception
error message.
         */
        public
override string
Message {
            get{

StringBuilder  buffer
= new StringBuilder();

                switch
(type) {
                case
ErrorType.INVALID_PARS
ER:

buffer.Append("parser
is invalid, as ");

buffer.Append(info);

break;
                case
ErrorType.INVALID_TOKE
N:

buffer.Append("token
'");

buffer.Append(name);

buffer.Append("' is
invalid, as ");

buffer.Append(info);

break;
                case
ErrorType.INVALID_PROD
UCTION:

buffer.Append("product
ion '");
```

```

buffer.Append(name);

buffer.Append("' is
invalid, as ");

buffer.Append(info);

break;
                case
ErrorType.INFINITE_LOO
P:

buffer.Append("infinit
e loop found in
production pattern
'");

buffer.Append(name);

buffer.Append("'");

break;
                case
ErrorType.INHERENT_AMB
IGUITY:

buffer.Append("inheren
t ambiguity in
production '");

buffer.Append(name);

buffer.Append("'");
                    if
(info != null) {

buffer.Append(" ");

buffer.Append(info);
                    }
                    if
(details != null) {

buffer.Append("
starting with ");

if (details.Count > 1)
{

buffer.Append("tokens
");

} else {

buffer.Append("token
");

}

buffer.Append(Details)
;
```

```
                    }

break;

default:

buffer.Append("interna
l error");

break;
                }
                return
buffer.ToString();
            }
        }

        /**
         * Returns the
error message. This
message will contain
all the
         * information
available.
         *
         * @return the
error message
         *
         * @see
#Message
         *
         * @deprecated
Use the Message
property instead.
         */
        public string
GetMessage() {
            return
Message;
        }
    }
}

/*
 *
ParserLogException.cs
 */

using System;
using
System.Collections;
using System.Text;

namespace Core.Library
{

    /**
     * A parser log
exception. This class
contains a list of all
the
     * parse errors
encountered while
parsing.
     *

     *
     * @since    1.1
     */
    public class
ParserLogException :
Exception {

        /**
         * The list of
errors found.
         */
        private
ArrayList errors = new
ArrayList();

        /**
         * Creates a
new empty parser log
exception.
         */
        public
ParserLogException() {
        }

        /**
         * The message
property (read-only).
This property contains
         * the
detailed exception
error message.
         */
        public
override string
Message {
            get{

StringBuilder  buffer
= new StringBuilder();

                for
(int i = 0; i < Count;
i++) {
                    if
(i > 0) {

buffer.Append("\n");
                    }

buffer.Append(this[i].
Message);
                }
                return
buffer.ToString();
            }
        }
        /**
         * The error
count property (read-
only).
         *
         *
         */
        public int
Count {
            get {
                return
errors.Count;
            }
        }

        /**
         * Returns the
number of errors in
this log.
         *
         * @return the
number of errors in
this log
         *
         * @see #Count
         *
         * @deprecated
Use the Count property
instead.
         */
        public int
GetErrorCount() {
            return
Count;
        }

        /**
         * The error
index (read-only).
This index contains
all the
         * errors in
this error log.
         *
         * @param
index          the
error index, 0 <=
index < Count
         *
         * @return the
parse error requested
         *
         *
         */
        public
ParseException
this[int index] {
            get {
                return
(ParseException)
errors[index];
            }
        }

        /**
         * Returns a
specific error from
the log.
         *
         * @param
index          the
error index, 0 <=
index < count
         *
         * @return the
parse error requested
         *
         * @deprecated
Use the class indexer
instead.
         */
        public
ParseException
GetError(int index) {
            return
this[index];
        }

        /**
         * Adds a
parse error to the
log.
         *
         * @param e
the parse error to add
         */
        public void
AddError(ParseExceptio
n e) {

errors.Add(e);
        }

        /**
         * Returns the
detailed error
message. This message
will contain
         * the error
messages from all
errors in this log,
separated by
         * a newline.
         *
         * @return the
detailed error message
         *
         * @see
#Message
         *
         * @deprecated
Use the Message
property instead.
         */
        public string
GetMessage() {
            return
Message;
        }
    }
}

using System;
using
System.Collections.Gen
eric;
using System.Linq;
using System.Text;
using
System.Threading.Tasks
;

namespace Core.Library
{
    public class
PredictSets
    {
        string program
= "hold, unit, digit,
company, joe,
response, struct,
PrimaryMission";
        string
comments = "comment,
hold, PrimaryMission,
unit, digit, company,
joe, response, struct,
}, post, inquire, go,
campaign, comment,
capture, phase,
inorder, id, ++, --,
backup, abort, $, (,
)";
        string
datatype = "unit,
digit, company, joe,
response";
        string
Literals = "Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE,
NEGATIVE";
        string
constant = "hold,
comment,
PrimaryMission, unit,
digit, company, joe,
response, struct";
        string
localChoice = "unit,
digit, company, joe,
response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --, (, ), abort,
}, $";
        string
localdec = "=, ,, ;";
        string
UnitaddID = "=, ,, ;";
        string
UnitEXinit = ",, ;";
        string main =
"PrimaryMission";
        string
globalDec = "unit,
digit, company, joe,
response, struct,
PrimaryMission, unit,
digit, company, joe,
response, struct,
comment";
        string
localdecChoice = ",
,unit, digit, company,
joe, response,
struct,";
        string
decChoice = "unit,
digit, company, joe,
response, ), ,";
        string
globalChoice = "=, ,,
(, ";
        string
BodyChoice = "=, ,";
        string
varUnitBody = "=, ,";
        string
functReturnBody = "(";
        string
functVoidBody = "(, ;,
numlit, declit,
stringlit, charlit";
        string
arrUnitBody = "";
        string arrType
= "";
        string N1 =
"";
        string
ArrayChoice = "=, ,,
;, unit, digit,
company, joe,
response, )";
        string N2 =
"";
        string index1
= "Numlit, id";
        string add =
"+";
        string index2
= "Numlit, id";
        string indexEX
= "id, Numlit";
        string unitAID
= "=, unit, digit,
company, joe,
response, ,, ;, {, ),
}";
        string
unitAIDTWO = "=, ,, ;,
unit, digit, company,
joe, response, =, )";
        string
unitElem = "Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE,
NEGATIVE";
        string EXTelem
= "Numlit, Declit,
Stringlit, Charlit,
id, AFFIRMATIVE,
NEGATIVE";
        string
EXTelemChoice = ",,
}";
        string
unitElemTwo = "{";
        string
ElemTwoLit = ",, },
=";
        string
ElemTwoTail = ",, }";
        string
assignChoice = "id,
++, --, }, backup,
abort, $, comment,
post, inquire, go,
campaign, capture,
phase, inorder, id,
++, --";
        string
AccessAssignDtype =
"id";
        string
assignValueChoice =
"=, ., {, +, -, *, /,
%, ^, ++, --";
        string
assigning = "=, {, +,
-, *, /, %, ^, ., ++,
--";
        string ArrayID
= "{";
        string
ArrayIDTail = "=";
        string
AssignSym = "+, -, *,
/, %, ^";
        string
assignValue = "Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE, NEGATIVE
or ++ , --, ;";
        string
functParam = "(, ;, +,
-, *, /, %, ^";
        string
functIDParam =
"Numlit, Declit,
Stringlit, Charlit,
id, AFFIRMATIVE,
NEGATIVE, )";
        string
addfunctIDParam = ",,
)";
        string funct =
"unit, digit, company,
joe, response, miss";
        string
functReturn = "unit,
digit, company, joe,
response";
        string
functVoid = "miss";
        string dtypeA
= "unit, digit,
response, id, )";
        string
EXdtypeA = ",, )";
        string dtypef
= "unit, digit,
response, id, backup,
++, --, }, backup";
        string ExID =
",";
        string
arrIndex = "";
        string
struct_U = "struct";
        string sDec =
"unit, digit, company,
joe, response, }, +, -
, *, /, =, (, ., )";
        string index =
"[";
        string body =
"post, capture,
inquire, inorder, go,
phase, campaign, id,
++, --, comment, post,
inquire, go, campaign,
capture, phase,
inorder, id, ++, --,
}, backup, abort, $,
comment, post,
inquire, go, campaign,
capture, phase,
inorder, id, ++, --";
        string print =
"post";
        string postval
= "Numlit, Declit,
Stringlit, Charlit,
id, AFFIRMATIVE,
NEGATIVE";
```

```
        string
ConcatLit = ″, , )″;
        string scan =
″capture″;
        string ExtI =
″, , )″;
        string
for_state = ″inquire″;
        string
forstatement = ″unit,
digit, company, joe,
response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --, }″;
        string val1 =
″Numlit, 0″;
        string mntCond
= ″++, --, +, -, *, /,
>, <, >=, <=, ==,
numlit, declit,
stringlit, charlit,
AFFIRMATIVE,
NEGATIVE″;
        string
mntCondT = ″++, --, +,
-, *, /, >, <, >=, <=,
==, numlit, declit,
stringlit, charlit,
AFFIRMATIVE,
NEGATIVE″;
        string mnt =
″++, --, +, -, *, /,
>, <, >=, <=, ==,
numlit, declit,
stringlit, charlit,
AFFIRMATIVE,
NEGATIVE″;
        string ifelse
= ″inorder″;
        string
ifcondition = ″Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE, NEGATIVE,
(″;
        string
ifstatement = ″unit,
digit, company, joe,
response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --, }″;
        string elseif
= ″otherorder, Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE, NEGATIVE,
(, order, }, backup,
abort, $, comment,
post, inquire, go,
campaign, capture,
phase, inorder, ++, --
″;
        string
elseifstatement =
″unit, digit, company,
joe, response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --″;
        string
else_state = ″order,
post, inquire, go,
campaign, comment,
capture, phase,
inorder, id, ++, --,
}, backup, abort, $″;
        string
elsestatement = ″unit,
digit, company, joe,
response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --, }″;
        string dowhile
= ″go″;
        string
dostatement = ″unit,
digit, company, joe,
response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --, }″;
        string
while_state = ″phase,
}, backup, abort, $,
comment, post,
inquire, go, campaign,
capture, phase,
inorder, id, ++, --″;
        string
whilestatement =
″unit, digit, company,
joe, response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --, }″;
        string
switch_state =
″campaign, abort″;
        string
case_state =
″operation″;
        string def =
″DEFAULT, }″;
        string
casestatement = ″unit,
digit, company, joe,
response, post,
inquire, go, campaign,
comment, capture,
phase, inorder, id,
++, --   , abort, }″;
        string MathOp
= ″(, Numlit, Declit,
Stringlit, Charlit,
id, AFFIRMATIVE,
NEGATIVE″;
        string
operCond = ″(, Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE,
NEGATIVE″;
        string
operCondChoice = ″+, -
, *, /, %, ^, =″;
        string operSym
= ″+, -,  *, /, %, ^″;
        string operEq
= ″+=, , -=, *=, /=,
%=, =″;
        string
operExt_s = ″Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE, NEGATIVE,
(″;
        string
operExt_rep = ″+, -,
*, /, %, ^, ), ;″;
        string operand
= ″Numlit, Declit,
Stringlit, Charlit,
id, AFFIRMATIVE,
NEGATIVE″;
        string
simMathOp = ″Numlit,
Declit, Stringlit,
Charlit, id,
AFFIRMATIVE,
NEGATIVE″;
        string
S_MathExt = ″+, -, *,
/, %, ^, ), ;″;
        string
operCondExt = ″+, -,
*, /, %, ^, ;″;
        string RelOp =
″Numlit, Declit,
Stringlit, Charlit,
id, AFFIRMATIVE,
NEGATIVE″;
        string
RelopExt = ″==, !=,
>=, <=, >, <, )″;
        string op1 =
″==, !=, >=, <=, >,
<″;
        string LogOp =
″(″;
        string
ExtLogOp = ″||, &″;
        string LogOper
= ″||, &″;
        string end =
″}″;
        string
StartProgram =
″comment, hold,
PrimaryMission, unit,
digit, company, joe,
response, struct″;

    public string
GetPredictSet(int
code)
        {
            switch
(code)
            {
                case
2001: return
StartProgram;
                case
2002: return program;
                case
2003: return comments;
                case
2004: return datatype;
                case
2005: return Literals;
                case
2006: return constant;
                case
2007: return
localChoice;
                case
2008: return localdec;
                case
2009: return
UnitaddID;
                case
2010: return
UnitEXinit;
                case
2011: return main;
                case
2012: return
globalDec;
                case
2013: return
localdecChoice;
                case
2014: return
decChoice;
                case
2015: return
globalChoice;
```

```
                case
2016: return
BodyChoice;
                case
2017: return
varUnitBody;
                case
2018: return
functReturnBody;
                case
2019: return
functVoidBody;
                case
2020: return
arrUnitBody;
                case
2021: return arrType;
                case
2022: return N1;
                case
2023: return
ArrayChoice;
                case
2024: return N2;
                case
2025: return index1;
                case
2026: return add;
                case
2027: return index2;
                case
2028: return indexEX;
                case
2029: return unitAID;
                case
2030: return
unitAIDTWO;
                case
2031: return unitElem;
                case
2032: return EXTelem;
                case
2033: return
EXTelemChoice;
                case
2034: return
unitElemTwo;
                case
2035: return
ElemTwoLit;
                case
2036: return
ElemTwoTail;
                case
2037: return
assignChoice;
                case
2038: return
AccessAssignDtype;
                case
2039: return
assignValueChoice;
                case
2040: return
assigning;
                case
2041: return ArrayID;
                case
2042: return
ArrayIDTail;
                case
2043: return
AssignSym;
                case
2044: return
assignValue;
                case
2045: return
functParam;
                case
2046: return
functIDParam;
                case
2047: return
addfunctIDParam;
                case
2048: return funct;
                case
2049: return
functReturn;
                case
2050: return
functVoid;
                case
2051: return dtypeA;
                case
2052: return EXdtypeA;
                case
2053: return dtypef;
                case
2054: return ExID;
                case
2055: return arrIndex;
                case
2056: return struct_U;
                case
2057: return sDec;
                case
2058: return index;
                case
2059: return body;
                case
2060: return print;
                case
2061: return postval;
                case
2062: return
ConcatLit;
                case
2063: return scan;
                case
2064: return ExtI;
                case
2065: return
for_state;
                case
2066: return
forstatement;
                case
2067: return val1;
                case
2068: return mntCond;
                case
2069: return mntCondT;
                case
2070: return mnt;
                case
2071: return ifelse;
                case
2072: return
ifcondition;
                case
2073: return
ifstatement;
                case
2074: return elseif;
                case
2075: return
elseifstatement;
                case
2076: return
else_state;
                case
2077: return
elsestatement;
                case
2078: return dowhile;
                case
2079: return
dostatement;
                case
2080: return
while_state;
                case
2081: return
whilestatement;
                case
2082: return
switch_state;
                case
2083: return
case_state;
                case
2084: return def;
                case
2085: return
casestatement;
                case
2086: return MathOp;
                case
2087: return operCond;
                case
2088: return
operCondChoice;
                case
2089: return operSym;
                case
2090: return operEq;
                case
2091: return
operExt_s;
                case
2092: return
operExt_rep;
                case
2093: return operand;
                case
2094: return
simMathOp;
                case
2095: return
S_MathExt;
                case
2096: return
operCondExt;
                case
2097: return RelOp;
                case
2098: return RelopExt;
                case
2099: return op1;
                case
2100: return LogOp;
                case
2101: return ExtLogOp;
                case
2102: return LogOper;
                case
2103: return end;
            }
            return "";
        }
    }

}

/*
 * Production.cs
 */

using
System.Collections;

namespace Core.Library
{

    /**
     * A production
node. This class
represents a grammar
production
     * (i.e. a list of
child nodes) in a
parse tree. The
productions
```

```
     * are created by
a parser, that adds
children a according
to a
     * set of
production patterns
(i.e. grammar rules).
     *

     *
     */
    public class
Production : Node {

        /**
         * The
production pattern
used for this
production.
         */
        private
ProductionPattern
pattern;

        /**
         * The child
nodes.
         */
        private
ArrayList children;

        /**
         * Creates a
new production node.
         *
         * @param
pattern        the
production pattern
         */
        public
Production(ProductionP
attern pattern) {

this.pattern =
pattern;

this.children = new
ArrayList();
        }

        /**
         * The node
type id property
(read-only). This
value is set as
         * a unique
identifier for each
type of node, in order
to
         * simplify
later identification.
         *
         *
         */
        public
override int Id {
            get {
                return
pattern.Id;
            }
        }

        /**
         * The node
name property (read-
only).
         *
         *
         */
        public
override string Name {
            get {
                return
pattern.Name;
            }
        }

        /**
         * The child
node count property
(read-only).
         *
         *
         */
        public
override int Count {
            get {
                return
children.Count;
            }
        }

        /**
         * The child
node index (read-
only).
         *
         * @param
index          the
child index, 0 <=
index < Count
         *
         * @return the
child node found, or
         *
null if index out of
bounds
         *
         *
         */
        public
override Node this[int
index] {
            get {
                if
(index < 0 || index >=
children.Count) {

return null;
                } else
{

return (Node)
children[index];
                }
            }
        }

        /**
         * Adds a
child node. The node
will be added last in
the list of
         * children.
         *
         * @param
child          the
child node to add
         */
        public void
AddChild(Node child) {
            if (child
!= null) {

child.SetParent(this);

children.Add(child);
            }
        }

        /**
         * The
production pattern
property (read-only).
This property
         * contains
the production pattern
linked to this
production.
         *
         *
         */
        public
ProductionPattern
Pattern {
            get {
                return
pattern;
            }
        }

        /**
         * Returns the
production pattern for
this production.
         *
         * @return the
production pattern
         *
         * @see
#Pattern
         *
         * @deprecated
Use the Pattern
property instead.
         */
        public
ProductionPattern
GetPattern() {
            return
Pattern;
        }

        /**
         * Checks if
this node is hidden,
i.e. if it should not
be visible
         * outside the
parser.
         *
         * @return
true if the node
should be hidden, or
         *
false otherwise
         */
        internal
override bool
IsHidden() {
            return
pattern.Synthetic;
        }

        /**
         * Returns a
string representation
of this production.
         *
         * @return a
string representation
of this production
         */
        public
override string
ToString() {
            return
pattern.Name + '(' +
pattern.Id + ')';
        }
    }
}

/*
 *
ProductionPattern.cs
 */
```

```
using
System.Collections;
using System.Text;

namespace Core.Library
{

    /**
     * A production
pattern. This class
represents a set of
production
     * alternatives
that together forms a
single production. A
     * production
pattern is identified
by an integer id and a
name,
     * both provided
upon creation. The
pattern id is used for
     * referencing the
production pattern
from production
pattern
     * elements.
     *

     *
     */
    public class
ProductionPattern {

        /**
         * The
production pattern
identity.
         */
        private int
id;

        /**
         * The
production pattern
name.
         */
        private string
name;

        /**
         * The
synthectic production
flag. If this flag is
set, the
         * production
identified by this
pattern has been
artificially
         * inserted
into the grammar.
         */
        private bool
synthetic;

        /**
         * The list of
production pattern
alternatives.
         */
        private
ArrayList
alternatives;

        /**
         * The default
production pattern
alternative. This
alternative
         * is used
when no other
alternatives match. It
may be set to
         * -1, meaning
that there is no
default (or fallback)
alternative.
         */
        private int
defaultAlt;

        /**
         * The look-
ahead set associated
with this pattern.
         */
        private
LookAheadSet
lookAhead;

        /**
         * Creates a
new production
pattern.
         *
         * @param id
the production pattern
id
         * @param name
the production pattern
name
         */
        public
ProductionPattern(int
id, string name) {
            this.id =
id;
            this.name
= name;

this.synthetic =
false;

this.alternatives =
new ArrayList();

this.defaultAlt = -1;

this.lookAhead = null;
        }

        /**
         * The
production pattern
identity property
(read-only). This
         * property
contains the unique
identity value.
         *
         *
         */
        public int Id
{
            get {
                return
id;
            }
        }

        /**
         * Returns the
unique production
pattern identity
value.
         *
         * @return the
production pattern id
         *
         * @see #Id
         *
         * @deprecated
Use the Id property
instead.
         */
        public int
GetId() {
            return Id;
        }

        /**
         * The
production pattern
name property (read-
only).
         *
         *
         */
        public string
Name {
            get {
                return
name;
            }
        }

        /**
         * Returns the
production pattern
name.
         *
         * @return the
production pattern
name
         *
         * @see #Name
         *
         * @deprecated
Use the Name property
instead.
         */
        public string
GetName() {
            return
Name;
        }

        /**
         * The
synthetic production
pattern property. If
this property
         * is set, the
production identified
by this pattern has
been
         *
artificially inserted
into the grammar. No
parse tree nodes
         * will be
created for such
nodes, instead the
child nodes
         * will be
added directly to the
parent node. By
default this
         * property is
set to false.
         *
         *
         */
        public bool
Synthetic {
            get {
                return
synthetic;
            }
            set {

synthetic = value;
            }
        }

        /**
```

```
         * Checks if
the synthetic
production flag is
set. If this
         * flag is
set, the production
identified by this
pattern has
         * been
artificially inserted
into the grammar. No
parse tree
         * nodes will
be created for such
nodes, instead the
child
         * nodes will
be added directly to
the parent node.
         *
         * @return
true if this
production pattern is
synthetic, or
         *
false otherwise
         *
         * @see
#Synthetic
         *
         * @deprecated
Use the Synthetic
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
         *
         * @see
#Synthetic
         *
         * @deprecated
Use the Synthetic
property instead.
         */
        public void
SetSyntetic(bool
synthetic) {
            Synthetic
= synthetic;
        }

        /**
         * The look-
ahead set property.
This property contains
the
         * look-ahead
set associated with
this alternative.
         */
        internal
LookAheadSet LookAhead
{
            get {
                return
lookAhead;
            }
            set {

lookAhead = value;
            }
        }

        /**
         * 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
has been set.
         */
        internal
ProductionPatternAlter
native
DefaultAlternative {
            get {
                if
(defaultAlt >= 0) {

object obj =
alternatives[defaultAl
t];

return
(ProductionPatternAlte
rnative) obj;
                } else
{

return null;
                }
            }
            set {

defaultAlt = 0;
                for
(int i = 0; i <
alternatives.Count;
i++) {
                    if
(alternatives[i] ==
value) {

defaultAlt = i;
                    }
                }
            }
        }

        /**
         * The
production pattern
alternative count
property
         * (read-
only).
         *
         *
         */
        public int
Count {
            get {
                return
alternatives.Count;
            }
        }

        /**
         * Returns the
number of alternatives
in this pattern.
         *
         * @return the
number of alternatives
in this pattern
         *
         * @see #Count
         *
         * @deprecated
Use the Count property
instead.
         */
        public int
GetAlternativeCount()
{
            return
Count;
        }

        /**
         * The
production pattern
alternative index
(read-only).
         *
         * @param
index          the
alternative index, 0
<= pos < Count
         *
         * @return the
alternative found
         *
         *
         */
        public
ProductionPatternAlter
native this[int index]
{
            get {
                return
(ProductionPatternAlte
rnative)
alternatives[index];
            }
        }

        /**
         * Returns an
alternative in this
pattern.
         *
         * @param pos
the alternative
position, 0 <= pos <
count
         *
         * @return the
alternative found
         *
         * @deprecated
Use the class indexer
instead.
         */
        public
ProductionPatternAlter
native
GetAlternative(int
pos) {
```

```
            return
this[pos];
        }

        /**
         * Checks if
this pattern is
recursive on the left-
hand side.
         * This method
checks if any of the
production pattern
         *
alternatives is left-
recursive.
         *
         * @return
true if at least one
alternative is left
recursive, or
         *
false otherwise
         */
        public bool
IsLeftRecursive() {

ProductionPatternAlter
native  alt;

            for (int i
= 0; i <
alternatives.Count;
i++) {
                alt =
(ProductionPatternAlte
rnative)
alternatives[i];
                if
(alt.IsLeftRecursive()
) {

return true;
                }
            }
            return
false;
        }

        /**
         * Checks if
this pattern is
recursive on the
right-hand side.
         * This method
checks if any of the
production pattern
         *
alternatives is right-
recursive.
         *
         * @return
true if at least one
alternative is right
recursive, or
         *
false otherwise
         */
        public bool
IsRightRecursive() {

ProductionPatternAlter
native  alt;

            for (int i
= 0; i <
alternatives.Count;
i++) {
                alt =
(ProductionPatternAlte
rnative)
alternatives[i];
                if
(alt.IsRightRecursive(
)) {

return true;
                }
            }
            return
false;
        }

        /**
         * Checks if
this pattern would
match an empty stream
of
         * tokens.
This method checks if
any one of the
production
         * pattern
alternatives would
match the empty token
stream.
         *
         * @return
true if at least one
alternative match no
tokens, or
         *
false otherwise
         */
        public bool
IsMatchingEmpty() {

ProductionPatternAlter
native  alt;

            for (int i
= 0; i <
alternatives.Count;
i++) {
                alt =
(ProductionPatternAlte
rnative)
alternatives[i];
                if
(alt.IsMatchingEmpty()
) {

return true;
                }
            }
            return
false;
        }

        /**
         * Adds a
production pattern
alternative.
         *
         * @param alt
the production pattern
alternative to add
         *
         * @throws
ParserCreationExceptio
n if an identical
alternative has
         *
already been added
         */
        public void
AddAlternative(Product
ionPatternAlternative
alt) {
            if
(alternatives.Contains
(alt)) {
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_PR
ODUCTION,

name,

"two identical
alternatives exist");
            }

alt.SetPattern(this);

alternatives.Add(alt);
        }

        /**
         * Returns a
string representation
of this object.
         *
         * @return a
token string
representation
         */
        public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();

StringBuilder  indent
= new StringBuilder();
            int
i;


buffer.Append(name);

buffer.Append("(");

buffer.Append(id);

buffer.Append(") ");
            for (i =
0; i < buffer.Length;
i++) {

indent.Append(" ");
            }
            for (i =
0; i <
alternatives.Count;
i++) {
                if (i
== 0) {

buffer.Append("= ");
                } else
{

buffer.Append("\n");

buffer.Append(indent);

buffer.Append("| ");
                }

buffer.Append(alternat
ives[i]);
            }
            return
buffer.ToString();
        }
    }
}

/*
 *
ProductionPatternAlter
native.cs
```

```
 */

using System;
using
System.Collections;
using System.Text;

namespace Core.Library
{

    /**
     * A production
pattern alternative.
This class represents
a list of
     * production
pattern elements. In
order to provide
productions that
     * cannot be
represented with the
element occurance
counters, multiple
     * alternatives
must be created and
added to the same
production
     * pattern. A
production pattern
alternative is always
contained
     * within a
production pattern.
     *

     *
     */
    public class
ProductionPatternAlter
native {

        /**
         * The
production pattern.
         */
        private
ProductionPattern
pattern;

        /**
         * The element
list.
         */
        private
ArrayList elements =
new ArrayList();

        /**
         * The look-
ahead set associated
with this alternative.
         */
        private
LookAheadSet lookAhead
= null;

        /**
         * Creates a
new production pattern
alternative.
         */
        public
ProductionPatternAlter
native() {
        }

        /**
         * The
production pattern
property (read-only).
This property
         * contains
the pattern having
this alternative.
         *
         *
         */
        public
ProductionPattern
Pattern {
            get {
                return
pattern;
            }
        }

        /**
         * Returns the
production pattern
containing this
alternative.
         *
         * @return the
production pattern for
this alternative
         *
         * @see
#Pattern
         *
         * @deprecated
Use the Pattern
property instead.
         */
        public
ProductionPattern
GetPattern() {
            return
Pattern;
        }

        /**
         * The look-
ahead set property.
This property contains
the
         * look-ahead
set associated with
this alternative.
         */
        internal
LookAheadSet LookAhead
{
            get {
                return
lookAhead;
            }
            set {

lookAhead = value;
            }
        }

        /**
         * The
production pattern
element count property
(read-only).
         *
         *
         */
        public int
Count {
            get {
                return
elements.Count;
            }
        }

        /**
         * Returns the
number of elements in
this alternative.
         *
         * @return the
number of elements in
this alternative
         *
         * @see #Count
         *
         * @deprecated
Use the Count property
instead.
         */
        public int
GetElementCount() {
            return
Count;
        }

        /**
         * The
production pattern
element index (read-
only).
         *
         * @param
index          the
element index, 0 <=
pos < Count
         *
         * @return the
element found
         *
         *
         */
        public
ProductionPatternEleme
nt this[int index] {
            get {
                return
(ProductionPatternElem
ent) elements[index];
            }
        }

        /**
         * Returns an
element in this
alternative.
         *
         * @param pos
the element position,
0 <= pos < count
         *
         * @return the
element found
         *
         * @deprecated
Use the class indexer
instead.
         */
        public
ProductionPatternEleme
nt GetElement(int pos)
{
            return
this[pos];
        }

        /**
         * Checks if
this alternative is
recursive on the left-
hand
         * side. This
method checks all the
possible left side
         * elements
and returns true if
the pattern itself is
among
         * them.
         *
         * @return
true if the
alternative is left
side recursive, or
```

```
         *
false otherwise
         */
        public bool
IsLeftRecursive() {

ProductionPatternEleme
nt  elem;

            for (int i
= 0; i <
elements.Count; i++) {
                elem =
(ProductionPatternElem
ent) elements[i];
                if
(elem.Id ==
pattern.Id) {

return true;
                } else
if (elem.MinCount > 0)
{

break;
                }
            }
            return
false;
        }

        /**
         * 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;
i >= 0; i--) {
                elem =
(ProductionPatternElem
ent) elements[i];
                if
(elem.Id ==
pattern.Id) {

return true;
                } else
if (elem.MinCount > 0)
{

break;
                }
            }
            return
false;
        }

        /**
         * Checks if
this alternative would
match an empty stream
of
         * tokens.
This check is
equivalent of
getMinElementCount()
         * returning
zero (0).
         *
         * @return
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
         */
        internal void
SetPattern(ProductionP
attern pattern) {

this.pattern =
pattern;
        }

        /**
         * Returns the
minimum number of
elements needed to
satisfy
         * this
alternative. The value
returned is the sum of
all the
         * elements
minimum count.
         *
         * @return the
minimum number of
elements
         */
        public int
GetMinElementCount() {

ProductionPatternEleme
nt  elem;
            int
min = 0;

            for (int i
= 0; i <
elements.Count; i++) {
                elem =
(ProductionPatternElem
ent) elements[i];
                min +=
elem.MinCount;
            }
            return
min;
        }

        /**
         * Returns the
maximum number of
elements needed to
satisfy
         * this
alternative. The value
returned is the sum of
all the
         * elements
maximum count.
         *
         * @return the
maximum number of
elements
         */
        public int
GetMaxElementCount() {

ProductionPatternEleme
nt  elem;
            int
max = 0;

            for (int i
= 0; i <
elements.Count; i++) {
                elem =
(ProductionPatternElem
ent) elements[i];
                if
(elem.MaxCount >=
Int32.MaxValue) {

return Int32.MaxValue;
                } else
{

max += elem.MaxCount;
                }
            }
            return
max;
        }

        /**
         * Adds a
token to this
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) {

AddElement(new
ProductionPatternEleme
nt(true, id, min,
max));
        }

        /**
         * Adds a
production to this
alternative. The
production is
         * appended to
the end of the element
list. The multiplicity
         * values
specified define if
the production is
optional or
         * required,
and if it can be
repeated.
         *
         * @param id
the production
(pattern) id
         * @param min
the minimum number of
occurancies
         * @param max
the maximum number of
occurancies, or
         *
-1 for infinite
         */
        public void
AddProduction(int id,
int min, int max) {

AddElement(new
ProductionPatternEleme
nt(false, id, min,
max));
        }

        /**
         * Adds a
production pattern
element to this
alternative. The
         * element is
appended to the end of
the element list.
         *
         * @param elem
the production pattern
element
         */
        public void
AddElement(ProductionP
atternElement elem) {

elements.Add(elem);
        }

        /**
         * Adds a
production pattern
element to this
alternative. The
         *
multiplicity values in
the element will be
overridden with
         * the
specified values. The
element is appended to
the end of
         * the element
list.
         *
         * @param elem
the production pattern
element
         * @param min
the minimum number of
occurancies
         * @param max
the maximum number of
occurancies, or
         *
-1 for infinite
         */
        public void
AddElement(ProductionP
atternElement elem,

int min,

int max) {

            if
(elem.IsToken()) {

AddToken(elem.Id, min,
max);
            } else {

AddProduction(elem.Id,
min, max);
            }
        }

        /**
         * Checks if
this object is equal
to another. This
method only
         * returns
true for another
production pattern
alternative
         * with
identical elements in
the same order.
         *
         * @param obj
the object to compare
with
         *
         * @return
true if the object is
identical to this one,
or
         *
false otherwise
         */
        public
override bool
Equals(object obj) {
            if (obj is
ProductionPatternAlter
native) {
                return
Equals((ProductionPatt
ernAlternative) obj);
            } else {
                return
false;
            }
        }

        /**
         * Checks if
this alternative is
equal to another. This
method
         * returns
true if the other
production pattern
alternative
         * has
identical elements in
the same order.
         *
         * @param alt
the alternative to
compare with
         *
         * @return
true if the object is
identical to this one,
or
         *
false otherwise
         */
        public bool
Equals(ProductionPatte
rnAlternative alt) {
            if
(elements.Count !=
alt.elements.Count) {
                return
false;
            }
            for (int i
= 0; i <
elements.Count; i++) {
                if
(!elements[i].Equals(a
lt.elements[i])) {

return false;
                }
            }
            return
true;
        }

        /**
         * Returns a
hash code for this
object.
         *
         * @return a
hash code for this
object
         */
        public
override int
GetHashCode() {
            return
elements.Count.GetHash
Code();
        }

        /**
         * Returns a
string representation
of this object.
         *
         * @return a
token string
representation
         */
        public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();

            for (int i
= 0; i <
elements.Count; i++) {
                if (i
> 0) {

buffer.Append(" ");
                }

buffer.Append(elements
[i]);
            }
            return
buffer.ToString();
        }
    }
}
```

```
/*
 *
ProductionPatternEleme
nt.cs
 */

using System;
using System.Text;


namespace Core.Library
{

    /**
     * A production
pattern element. This
class represents a
reference to
     * either a token
or a production. Each
element also contains
minimum
     * and maximum
occurence counters,
controlling the number
of
     * repetitions
allowed. A production
pattern element is
always
     * contained
within a production
pattern rule.
     *

     *
     */
    public class
ProductionPatternEleme
nt {

        /**
         * The token
flag. This flag is
true for token
elements, and
         * false for
production elements.
         */
        private bool
token;

        /**
         * The node
identity.
         */
        private int
id;

        /**
         * The minimum
occurance count.
         */
        private int
min;

        /**
         * The maximum
occurance count.
         */
        private int
max;

        /**
         * The look-
ahead set associated
with this element.
         */
        private
LookAheadSet
lookAhead;


        /**
         * Creates a
new element. If the
maximum value if zero
(0) or
         * negative,
it will be set to
Int32.MaxValue.
         *
         * @param
isToken        the
token flag
         * @param id
the node identity
         * @param min
the minimum number of
occurancies
         * @param max
the maximum number of
occurancies, or
         *
negative for infinite
         */
        public
ProductionPatternEleme
nt(bool isToken,

int id,

int min,

int max) {

            this.token
= isToken;
            this.id =
id;
            if (min <
0) {
                min =
0;
            }
            this.min =
min;
            if (max <=
0) {
                max =
Int32.MaxValue;
            } else if
(max < min) {
                max =
min;
            }
            this.max =
max;

this.lookAhead = null;
        }

        /**
         * The node
identity property
(read-only).
         *
         *
         */
        public int Id
{
            get {
                return
id;
            }
        }

        /**
         * Returns the
node identity.
         *
         * @return the
node identity
         *
         * @see #Id
         *
         * @deprecated
Use the Id property
instead.
         */
        public int
GetId() {
            return Id;
        }

        /**
         * The minimum
occurence count
property (read-only).
         *
         *
         */
        public int
MinCount {
            get {
                return
min;
            }
        }

        /**
         * Returns the
minimum occurence
count.
         *
         * @return the
minimum occurence
count
         *
         * @see
#MinCount
         *
         * @deprecated
Use the MinCount
property instead.
         */
        public int
GetMinCount() {
            return
MinCount;
        }

        /**
         * The maximum
occurence count
property (read-only).
         *
         *
         */
        public int
MaxCount {
            get {
                return
max;
            }
        }

        /**
         * Returns the
maximum occurence
count.
         *
         * @return the
maximum occurence
count
         *
         * @see
#MaxCount
         *
         * @deprecated
Use the MaxCount
property instead.
         */
        public int
GetMaxCount() {
            return
MaxCount;
        }
        /**
         * The look-
ahead set property.
This is the look-ahead
set
         * associated
with this alternative.
         */
        internal
LookAheadSet LookAhead
{
            get {
                return
lookAhead;
            }
            set {

lookAhead = value;
            }
        }

        /**
         * Returns
true if this element
represents a token.
         *
         * @return
true if the element is
a token, or
         *
false otherwise
         */
        public bool
IsToken() {
            return
token;
        }

        /**
         * Returns
true if this element
represents a
production.
         *
         * @return
true if the element is
a production, or
         *
false otherwise
         */
        public bool
IsProduction() {
            return
!token;
        }

        /**
         * Checks if a
specific token matches
this element. This
         * method will
only return true if
this element is a
token
         * element,
and the token has the
same id and this
element.
         *
         * @param
token          the
token to check
         *
         * @return
true if the token
matches this element,
or
         *
false otherwise
         */
        public bool
IsMatch(Token token) {
            return
IsToken() && token !=
null && token.Id ==
id;
        }


        /**
         * Checks if
this object is equal
to another. This
method only
         * returns
true for another
identical production
pattern
         * element.
         *
         * @param obj
the object to compare
with
         *
         * @return
true if the object is
identical to this one,
or
         *
false otherwise
         */
        public
override bool
Equals(object obj) {

ProductionPatternEleme
nt  elem;

                if (obj is
ProductionPatternEleme
nt) {
                    elem =
(ProductionPatternElem
ent) obj;
                    return
this.token ==
elem.token
                        &&
this.id == elem.id
                        &&
this.min == elem.min
                        &&
this.max == elem.max;
            } else {
                return
false;
            }
        }

        /**
         * Returns a
hash code for this
object.
         *
         * @return a
hash code for this
object
         */
        public
override int
GetHashCode() {
            return
this.id * 37;
        }

        /**
         * Returns a
string representation
of this object.
         *
         * @return a
string representation
of this object
         */
        public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();


buffer.Append(id);
            if (token)
{

buffer.Append("(Token)
");
            } else {

buffer.Append("(Produc
tion)");
            }
            if (min !=
1 || max != 1) {

buffer.Append("{");

buffer.Append(min);

buffer.Append(",");

buffer.Append(max);

buffer.Append("}");
            }
            return
buffer.ToString();
        }
    }
}

/*
 * ReaderBuffer.cs
 */

using System;
using System.IO;


namespace Core.Library
{


    /**
     * A character
buffer that
automatically reads
from an input source
     * stream when
needed. This class
keeps track of the
current position
     * in the buffer
and its line and
column number in the
original input
     * source. It
allows unlimited look-
ahead of characters in
the input,
     * reading and
buffering the required
data internally. As
the
     * position is
advanced, the buffer
content prior to the
current
     * position is
subject to removal to
make space for reading
new
     * content. A few
characters before the
current position are
always
```

```
     * kept to enable
boundary condition
checks.
     *

     *
     *
     */
    public class
ReaderBuffer {

        /**
         * The stream
reading block size.
All reads from the
underlying
         * character
stream will be made in
multiples of this
block size.
         * Also the
character buffer size
will always be a
multiple of
         * this
factor.
         */
        public const
int BLOCK_SIZE = 1024;

        /**
        * The
character buffer.
        */
        private char[]
buffer = new
char[BLOCK_SIZE * 4];

        /**
        * The current
character buffer
position.
        */
        private int
pos = 0;

        /**
        * The number
of characters in the
buffer.
        */
        private int
length = 0;

        /**
        * The input
source character
reader.
        */
        private
TextReader input =
null;
```

```
        /**
        * The line
number of the next
character to read.
This value will
        * be
incremented when
reading past line
breaks.
        */
        private int
line = 1;

        /**
        * The column
number of the next
character to read.
This value
        * will be
updated for every
character read.
        */
        private int
column = 1;

        /**
         * Creates a
new tokenizer
character buffer.
         *
         * @param
input          the
input source character
reader
         */
        public
ReaderBuffer(TextReade
r input) {
            this.input
= input;
        }

        /**
        * Discards all
resources used by this
buffer. This will also
        * close the
source input stream.
Disposing a previously
disposed
        * buffer has
no effect.
        */
        public void
Dispose() {
            buffer =
null;
            pos = 0;
            length =
0;
```

```
            if (input
!= null) {
                try {

input.Close();
                }
catch (Exception) {
                    //
Do nothing
                }
                input
= null;
            }
        }

        /**
         * The current
buffer position
property (read-only).
         */
        public int
Position {
            get {
                return
pos;
            }
        }

        /**
         * The current
line number property
(read-only). This
number
         * is the line
number of the next
character to read.
         */
        public int
LineNumber {
            get {
                return
line;
            }
        }

        /**
         * The current
column number property
(read-only). This
number
         * is the
column number of the
next character to
read.
         */
        public int
ColumnNumber {
            get {
                return
column;
            }
        }
```

```
        /**
         * The current
character buffer
length property (read-
only).
         * Note that
the length may
increase (and
decrease) as more
         * characters
are read from the
input source or
removed to
         * free up
space.
         */
        public int
Length {
            get {
                return
length;
            }
        }

        /**
         * Returns a
substring already in
the buffer. Note that
this
         * method may
behave in unexpected
ways when performing
         * operations
that modifies the
buffer content.
         *
         * @param
index          the
start index, inclusive
         * @param
length         the
substring length
         *
         * @return the
substring specified
         *
         * @throws
IndexOutOfBoundsExcept
ion if one of the
indices were
         *
negative or not less
than (or equal) than
length()
         */
        public string
Substring(int index,
int length) {
            return new
string(buffer, index,
length);
```

```
        }

        /**
         * Returns the current content of the buffer as a string. Note
         * that content before the current position will also be
         * returned.
         *
         * @return the current buffer content
         */
        public override string ToString() {
            return new string(buffer, 0, length);
        }

        /**
         * Returns a character relative to the current position. This
         * method may read from the input source and may also trim the
         * buffer content prior to the current position. The result of
         * calling this method may therefore be that the buffer length
         * and content have been modified.<p>
         *
         * The character offset must be positive, but is allowed to span
         * the entire size of the input source stream. Note that the
         * internal buffer must hold all the intermediate characters,
         * which may be wasteful if the offset is too large.
         *
         * @param offset         the character offset, from 0 and up
         *
         * @return the character found as an integer in the range 0 to
         * 65535 (0x00-0xffff), or -1 if the end of the stream was reached
         *
         * @throws IOException if an I/O error occurred
         */
        public int Peek(int offset) {
            int  index = pos + offset;

            // Avoid most calls to EnsureBuffered(), since we are in a
            // performance hotspot here. This check is not exhaustive,
            // but only present here to speed things up.
            if (index >= length) {
                EnsureBuffered(offset + 1);
                index = pos + offset;
            }
            return (index >= length) ? -1 : buffer[index];
        }

        /**
         * Reads the specified number of characters from the current
         * position. This will also move the current position forward.
         * This method will not attempt to move beyond the end of the
         * input source stream. When reaching the end of file, the
         * returned string might be shorter than requested. Any
         * remaining characters will always be returned before returning
         * null.
         *
         * @param offset         the character offset, from 0 and up
         *
         * @return the string containing the characters read, or
         *         null no more characters remain in the buffer
         *
         * @throws IOException if an I/O error occurred
         */
        public string Read(int offset) {
            int     count;
            string  result;

            EnsureBuffered(offset + 1);
            if (pos >= length) {
                return null;
            } else {
                count = length - pos;
                if (count > offset) {
                    count = offset;
                }
                UpdateLineColumnNumbers(count);
                result = new string(buffer, pos, count);
                pos += count;
                if (input == null && pos >= length) {
                    Dispose();
                }
                return result;
            }
        }

        /**
         * Updates the line and column numbers counters. This method
         * requires all the characters to be processed (i.e. returned
         * as read) to be present in the buffer, starting at the
         * current buffer position.
         *
         * @param offset         the number of characters to process
         */
        private void UpdateLineColumnNumbers(int offset) {
            for (int i = 0; i < offset; i++) {
                if (buffer[pos + i] == '\n') {
                    line++;
                    column = 1;
                } else {
                    column++;
                }
            }
        }

        /**
         * Ensures that the specified offset is read into the buffer.
         * This method will read characters from the input stream and
         * appends them to the buffer if needed. This method is safe to
         * call even
after end of file has
been reached. This
method also
         * handles
removal of characters
at the beginning of
the buffer
         * once the
current position is
high enough. It will
also enlarge
         * the buffer
as needed.
         *
         * @param
offset         the
read offset, from 0
and up
         *
         * @throws
IOException if an
error was encountered
while reading
         *
the input stream
         */
        private void
EnsureBuffered(int
offset) {
            int  size;
            int
readSize;

            // Check
for end of stream or
already read
characters
            if (input
== null || pos +
offset < length) {

return;
            }

            // Remove
(almost all) old
characters from buffer
            if (pos >
BLOCK_SIZE) {
                length
-= (pos - 16);

Array.Copy(buffer, pos
- 16, buffer, 0,
length);
                pos =
16;
            }
            //
Calculate number of
characters to read
            size = pos
+ offset - length + 1;
            if (size %
BLOCK_SIZE != 0) {
                size =
(1 + size /
BLOCK_SIZE) *
BLOCK_SIZE;
            }

EnsureCapacity(length
+ size);

            // Read
characters
            try {
                while
(input != null && size
> 0) {

readSize =
input.Read(buffer,
length, size);
                    if
(readSize > 0) {

length += readSize;

size -= readSize;
                    }
else {

input.Close();

input = null;
                    }
                }
            } catch
(IOException e) {
                input
= null;
                throw
e;
            }
        }

        /**
         * Ensures
that the buffer has at
least the specified
capacity.
         *
         * @param size
the minimum buffer
size
         */
        private void
EnsureCapacity(int
size) {
            if
(buffer.Length >=
size) {

return;
            }
            if (size %
BLOCK_SIZE != 0) {
                size =
(1 + size /
BLOCK_SIZE) *
BLOCK_SIZE;
            }

Array.Resize(ref
buffer, size);
        }
    }
}

/*
 *
RecursiveDescentParser
.cs
 */

using System;
using
System.Collections;
using System.IO;

namespace Core.Library
{

    /**
     * A recursive
descent parser. This
parser handles LL(n)
grammars,
     * selecting the
appropriate pattern to
parse based on the
next few
     * tokens. The
parser is more
efficient the fewer
look-ahead tokens
     * that is has to
consider.
     *

     *
     */
    public class
RecursiveDescentParser
: Parser {

        /**
         * Creates a
new parser.
         *
         * @param
input          the
input stream to read
from
         *
         * @throws
ParserCreationExceptio
n if the tokenizer
couldn't be
         *
initialized correctly
         *
         *
         */
        public
RecursiveDescentParser
(TextReader input) :
base(input) {
        }

        /**
         * Creates a
new parser.
         *
         * @param
input          the
input stream to read
from
         * @param
analyzer       the
analyzer callback to
use
         *
         * @throws
ParserCreationExceptio
n if the tokenizer
couldn't be
         *
initialized correctly
         *
         *
         */
        public
RecursiveDescentParser
(TextReader input,
Analyzer analyzer)
            :
base(input, analyzer)
{
        }

        /**
         * Creates a
new parser.
         *
         * @param
tokenizer      the
tokenizer to use
         */
        public
RecursiveDescentParser
(Tokenizer tokenizer)
```

```
            :
base(tokenizer) {
        }

        /**
         * Creates a
new parser.
         *
         * @param
tokenizer      the
tokenizer to use
         * @param
analyzer       the
analyzer callback to
use
         */
        public
RecursiveDescentParser
(Tokenizer tokenizer,

Analyzer analyzer)
            :
base(tokenizer,
analyzer) {
        }

        /**
         * Adds a new
production pattern to
the parser. The
pattern
         * will be
added last in the
list. The first
pattern added is
         * assumed to
be the starting point
in the grammar. The
         * pattern
will be validated
against the grammar
type to some
         * extent.
         *
         * @param
pattern        the
pattern to add
         *
         * @throws
ParserCreationExceptio
n if the pattern
couldn't be
         *
added correctly to the
parser
         */
        public
override void
AddPattern(ProductionP
attern pattern) {
```

```
            // Check
for empty matches
            if
(pattern.IsMatchingEmp
ty()) {
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_PR
ODUCTION,

pattern.Name,

"zero elements can be
matched (minimum is
one)");
            }

            // Check
for left-recusive
patterns
            if
(pattern.IsLeftRecursi
ve()) {
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_PR
ODUCTION,

pattern.Name,

"left recursive
patterns are not
allowed");
            }

            // Add
pattern

base.AddPattern(patter
n);
        }

        /**
         * Initializes
the parser. All the
added production
patterns
         * will be
analyzed for
ambiguities and
errors. This method
         * also
initializes the
```

```
internal data
structures used during
         * the
parsing.
         *
         * @throws
ParserCreationExceptio
n if the parser
couldn't be
         *
initialized correctly
         */
        public
override void
Prepare() {

IEnumerator e;

            //
Performs production
pattern checks

base.Prepare();

SetInitialized(false);

            //
Calculate production
look-ahead sets
            e =
GetPatterns().GetEnume
rator();
            while
(e.MoveNext()) {

CalculateLookAhead((Pr
oductionPattern)e.Curr
ent);
            }

            // Set
initialized flag

SetInitialized(true);
        }

        /**
         * Parses the
input stream and
creates a parse tree.
         *
         * @return the
parse tree
         *
         * @throws
ParseException if the
input couldn't be
parsed
         *
correctly
         */
```

```
        protected
override Node
ParseStart() {
            Token
token;
            Node node;
            ArrayList
list;

            node =
ParsePattern(GetStartP
attern());
            token =
PeekToken(0);
            if (token
!= null) {
                list =
new ArrayList(1);

list.Add("<EOF>");
                throw
new ParseException(

ParseException.ErrorTy
pe.UNEXPECTED_TOKEN,

token.ToShortString(),

list,

token.StartLine,

token.StartColumn);
            }
            return
node;
        }

        /**
         * Parses a
production pattern. A
parse tree node may or
may
         * not be
created depending on
the analyzer
callbacks.
         *
         * @param
pattern        the
production pattern to
parse
         *
         * @return the
parse tree node
created, or null
         *
         * @throws
ParseException if the
input couldn't be
parsed
```

```
         *
correctly
         */
        private Node
ParsePattern(Productio
nPattern pattern) {

ProductionPatternAlter
native alt;

ProductionPatternAlter
native defaultAlt;

            defaultAlt
=
pattern.DefaultAlterna
tive;
            for (int i
= 0; i <
pattern.Count; i++) {
                alt =
pattern[i];
                if
(defaultAlt != alt &&
IsNext(alt)) {

return
ParseAlternative(alt);
                }
            }
            if
(defaultAlt == null ||
!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
         * callbacks.
         *
         * @param alt
the production pattern
alternative
         *
         * @return the
parse tree node
created, or null
         *
         * @throws
ParseException if the
input couldn't be
parsed
         *
correctly
         */
        private Node
ParseAlternative(Produ
ctionPatternAlternativ
e alt) {
            Production
node;

            node =
NewProduction(alt.Patt
ern);

EnterNode(node);
            for (int i
= 0; i < alt.Count;
i++) {
                try {

ParseElement(node,
alt[i]);
                }
catch (ParseException
e) {

AddError(e, true);

NextToken();
                    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
         *
         * @throws
ParseException if the
input couldn't be
parsed
         *
correctly
         */
        private void
ParseElement(Productio
n node,

ProductionPatternEleme
nt elem) {

            Node
child;

            for (int i
= 0; i <
elem.MaxCount; i++) {
                string
pr =
Enum.GetName(typeof(Sy
ntaxConstants),
elem.GetId());
                if (i
< elem.MinCount ||
IsNext(elem)) {
                    if
(elem.IsToken()) {

child =
NextToken(elem.Id);

EnterNode(child);

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");

production.AddProducti
onCode(elem.GetId());

production.AddProducti
onState("Enter: <" +
pr + ">\n");

child =
ParsePattern(GetPatter
n(elem.Id));

AddNode(node, child);
                    }
                } else
{
                    pr
= pr.Substring(5);

production.AddRecursiv
eProduction("Enter:
NULL <" + pr + ">\n");

production.AddProducti
onState("NULL");

production.AddProducti
onCode(elem.GetId());

break;
                }
            }
        }

        /**
         * 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;

            if (set ==
null) {
```

```
                return
false;
            } else {
                return
set.IsNext(this);
            }
        }

        /**
         * Checks if
the next tokens match
a production pattern
         *
alternative. The
pattern alternative
look-ahead set will be
         * used if
existing, otherwise
this method returns
false.
         *
         * @param alt
the pattern
alternative to check
         *
         * @return
true if the next
tokens match, or
         *
false otherwise
         */
        private bool
IsNext(ProductionPatte
rnAlternative alt) {

LookAheadSet  set =
alt.LookAhead;

            if (set ==
null) {
                return
false;
            } else {
                return
set.IsNext(this);
            }
        }

        /**
         * Checks if
the next tokens match
a production pattern
         * element. If
the element has a
look-ahead set it will
be
         * used,
otherwise the look-
ahead set of the
referenced
         * production
or token will be used.
         *
         * @param elem
the pattern element to
check
         *
         * @return
true if the next
tokens match, or
         *
false otherwise
         */
        private bool
IsNext(ProductionPatte
rnElement elem) {

LookAheadSet  set =
elem.LookAhead;

            if (set !=
null) {
                return
set.IsNext(this);
            } else if
(elem.IsToken()) {
                return
elem.IsMatch(PeekToken
(0));
            } else {
                return
IsNext(GetPattern(elem
.Id));
            }
        }

        /**
         * Calculates
the look-ahead needed
for the specified
production
         * pattern.
This method attempts
to resolve any
conflicts and
         * stores the
results in the pattern
look-ahead object.
         *
         * @param
pattern        the
production pattern
         *
         * @throws
ParserCreationExceptio
n if the look-ahead
set couldn't
         *
be determined due to
inherent ambiguities
         */
        private void
CalculateLookAhead(Pro
ductionPattern
pattern) {

ProductionPatternAlter
native  alt;

LookAheadSet
result;

LookAheadSet[]
alternatives;

LookAheadSet
conflicts;

LookAheadSet
previous = new
LookAheadSet(0);
            int
length = 1;
            int
i;
            CallStack
stack = new
CallStack();

            //
Calculate simple look-
ahead

stack.Push(pattern.Nam
e, 1);
            result =
new LookAheadSet(1);

alternatives = new
LookAheadSet[pattern.C
ount];
            for (i =
0; i < pattern.Count;
i++) {
                alt =
pattern[i];

alternatives[i] =
FindLookAhead(alt, 1,
0, stack, null);

alt.LookAhead =
alternatives[i];

result.AddAll(alternat
ives[i]);
            }
            if
(pattern.LookAhead ==
null) {

pattern.LookAhead =
result;
            }
            conflicts
=
FindConflicts(pattern,
1);

            // Resolve
conflicts
            while
(conflicts.Size() > 0)
{

length++;

stack.Clear();

stack.Push(pattern.Nam
e, length);

conflicts.AddAll(previ
ous);
                for (i
= 0; i <
pattern.Count; i++) {

alt = pattern[i];
                    if
(alternatives[i].Inter
sects(conflicts)) {

alternatives[i] =
FindLookAhead(alt,

length,

0,

stack,

conflicts);

alt.LookAhead =
alternatives[i];
                    }
                    if
(alternatives[i].Inter
sects(conflicts)) {

if
(pattern.DefaultAltern
ative == null) {

pattern.DefaultAlterna
tive = alt;

} else if
(pattern.DefaultAltern
ative != alt) {

result =
alternatives[i].Create
Intersection(conflicts
);
ThrowAmbiguityExceptio
n(pattern.Name,

null,

result);

}
                    }
                }

previous = conflicts;

conflicts =
FindConflicts(pattern,
length);
            }

            // Resolve
conflicts inside rules
            for (i =
0; i < pattern.Count;
i++) {

CalculateLookAhead(pat
tern[i], 0);
            }
        }

        /**
         * Calculates
the look-aheads needed
for the specified
pattern
         *
alternative. This
method attempts to
resolve any conflicts
in
         * optional
elements by
recalculating look-
aheads for referenced
         *
productions.
         *
         * @param alt
the production pattern
alternative
         * @param pos
the pattern element
position
         *
         * @throws
ParserCreationExceptio
n if the look-ahead
set couldn't
         *
be determined due to
inherent ambiguities
         */
        private void
CalculateLookAhead(Pro
ductionPatternAlternat
ive alt,

int pos) {


ProductionPattern
pattern;

ProductionPatternEleme
nt  elem;

LookAheadSet
first;

LookAheadSet
follow;

LookAheadSet
conflicts;

LookAheadSet
previous = new
LookAheadSet(0);
            String
location;
            int
length = 1;

            // Check
trivial cases
            if (pos >=
alt.Count) {

return;
            }

            // Check
for non-optional
element
            pattern =
alt.Pattern;
            elem =
alt[pos];
            if
(elem.MinCount ==
elem.MaxCount) {

CalculateLookAhead(alt
, pos + 1);

return;
            }

            //
Calculate simple look-
aheads
            first =
FindLookAhead(elem, 1,
new CallStack(),
null);
            follow =
FindLookAhead(alt, 1,
pos + 1, new
CallStack(), null);

            // Resolve
conflicts
            location =
″at position ″ + (pos
+ 1);
            conflicts
=
FindConflicts(pattern.
Name,

location,

first,

follow);
            while
(conflicts.Size() > 0)
{

length++;

conflicts.AddAll(previ
ous);
                first
= FindLookAhead(elem,

length,

new CallStack(),

conflicts);
                follow
= FindLookAhead(alt,

length,

pos + 1,

new CallStack(),

conflicts);
                first
=
first.CreateCombinatio
n(follow);

elem.LookAhead =
first;
                if
(first.Intersects(conf
licts)) {

first =
first.CreateIntersecti
on(conflicts);

ThrowAmbiguityExceptio
n(pattern.Name,
location, first);
                }

previous = conflicts;

conflicts =
FindConflicts(pattern.
Name,

location,

first,

follow);
            }

            // Check
remaining elements

CalculateLookAhead(alt
, pos + 1);
        }

        /**
         * Finds the
look-ahead set for a
production pattern.
The maximum
         * look-ahead
length must be
specified. It is also
possible to
         * specify a
look-ahead set filter,
which will make sure
that
         * unnecessary
token sequences will
be avoided.
         *
         * @param
pattern        the
production pattern
         * @param
length         the
maximum look-ahead
length
         * @param
stack          the
call stack used for
loop detection
         * @param
filter         the
look-ahead set filter
         *
         * @return the
look-ahead set for the
production pattern
         *
         * @throws
ParserCreationExceptio
n if an infinite loop
was found
         *
in the grammar
         */
        private
LookAheadSet
FindLookAhead(Producti
onPattern pattern,

int length,

CallStack stack,

LookAheadSet filter) {


LookAheadSet  result;

LookAheadSet  temp;

            // Check
for infinite loop
            if
(stack.Contains(patter
n.Name, length)) {
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INFINITE_L
OOP,

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[
i],

length,

0,

stack,

filter);

result.AddAll(temp);
            }

stack.Pop();

            return
result;
        }

        /**
         * Finds the
look-ahead set for a
production pattern
alternative.
         * The pattern
position and maximum
look-ahead length must
be
         * specified.
It is also possible to
specify a look-ahead
set
         * filter,
which will make sure
that unnecessary token
sequences
         * will be
avoided.
         *
         * @param alt
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
         */
        private
LookAheadSet
FindLookAhead(Producti
onPatternAlternative
alt,

int length,

int pos,

CallStack stack,

LookAheadSet filter) {


LookAheadSet  first;

LookAheadSet  follow;

LookAheadSet
overlaps;

            // Check
trivial cases
            if (length
<= 0 || pos >=
alt.Count) {
                return
new LookAheadSet(0);
            }

            // Find
look-ahead for this
element
            first =
FindLookAhead(alt[pos]
, length, stack,
filter);
            if
(alt[pos].MinCount ==
0) {

first.AddEmpty();
            }

            // Find
remaining look-ahead
            if (filter
== null) {
                length
-=
first.GetMinLength();
                if
(length > 0) {

follow =
FindLookAhead(alt,
length, pos + 1,
stack, null);

first =
first.CreateCombinatio
n(follow);
                }
            } else if
(filter.IsOverlap(firs
t)) {

overlaps =
first.CreateOverlaps(f
ilter);
                length
-=
overlaps.GetMinLength(
);
                filter
=
filter.CreateFilter(ov
erlaps);
                follow
= FindLookAhead(alt,
length, pos + 1,
stack, filter);

first.RemoveAll(overla
ps);

first.AddAll(overlaps.
CreateCombination(foll
ow));
            }

            return
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
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
unnecessary
```

```
        * token
sequences will be
avoided.
        *
        * @param elem
the production pattern
element
        * @param
length        the
maximum look-ahead
length
        * @param
stack         the
call stack used for
loop detection
        * @param
filter        the
look-ahead set filter
        *
        * @return the
look-ahead set for the
pattern element
        *
        * @throws
ParserCreationExceptio
n if an infinite loop
was found
        *
in the grammar
        */
       private
LookAheadSet
FindLookAhead(Producti
onPatternElement elem,

int length,

CallStack stack,

LookAheadSet filter) {


LookAheadSet  result;

LookAheadSet  first;

LookAheadSet  follow;
            int
max;

            // Find
initial element look-
ahead
            first =
FindLookAhead(elem,
length, 0, stack,
filter);
            result =
new
LookAheadSet(length);

result.AddAll(first);
            if (filter
== null ||
!filter.IsOverlap(resu
lt)) {
                return
result;
            }

            // Handle
element repetitions
            if
(elem.MaxCount ==
Int32.MaxValue) {
                first
=
first.CreateRepetitive
();
            }
            max =
elem.MaxCount;
            if (length
< max) {
                max =
length;
            }
            for (int i
= 1; i < max; i++) {
                first
=
first.CreateOverlaps(f
ilter);
                if
(first.Size() <= 0 ||
first.GetMinLength()
>= length) {

break;
                }
                follow
= FindLookAhead(elem,

length,

0,

stack,

filter.CreateFilter(fi
rst));
                first
=
first.CreateCombinatio
n(follow);

result.AddAll(first);
            }

            return
result;
        }

        /**
        * Finds the
look-ahead set for a
production pattern
element. The
        * maximum
look-ahead length must
be specified. This
method does
        * NOT take
the element repeat
into consideration
when creating
        * the look-
ahead set. It is also
possible to specify a
look-ahead
        * set filter,
which will make sure
that unnecessary token
        * sequences
will be avoided.
        *
        * @param elem
the production pattern
element
        * @param
length        the
maximum look-ahead
length
        * @param
dummy          a
parameter to
distinguish the method
        * @param
stack         the
call stack used for
loop detection
        * @param
filter        the
look-ahead set filter
        *
        * @return the
look-ahead set for the
pattern element
        *
        * @throws
ParserCreationExceptio
n if an infinite loop
was found
        *
in the grammar
        */
       private
LookAheadSet
FindLookAhead(Producti
onPatternElement elem,

int length,

int dummy,

CallStack stack,

LookAheadSet filter) {


LookAheadSet
result;

ProductionPattern
pattern;

            if
(elem.IsToken()) {
                result
= new
LookAheadSet(length);

result.Add(elem.Id);
            } else {

pattern =
GetPattern(elem.Id);
                result
=
FindLookAhead(pattern,
length, stack,
filter);
                if
(stack.Contains(patter
n.Name)) {

result =
result.CreateRepetitiv
e();
                }
            }

            return
result;
        }

        /**
        * Returns a
look-ahead set with
all conflics between
        *
alternatives in a
production pattern.
        *
        * @param
pattern       the
production pattern
        * @param
maxLength     the
maximum token sequence
length
        *
        * @return a
look-ahead set with
the conflicts found
        *
        * @throws
ParserCreationExceptio
```

```
n if an inherent
ambiguity was
         *
found among the look-
ahead sets
         */
        private
LookAheadSet
FindConflicts(Producti
onPattern pattern,

int maxLength) {


LookAheadSet  result =
new
LookAheadSet(maxLength
);

LookAheadSet  set1;

LookAheadSet  set2;

            for (int i
= 0; i <
pattern.Count; i++) {
                set1 =
pattern[i].LookAhead;
                for
(int j = 0; j < i;
j++) {

set2 =
pattern[j].LookAhead;

result.AddAll(set1.Cre
ateIntersection(set2))
;
                }
            }
            if
(result.IsRepetitive()
) {

ThrowAmbiguityExceptio
n(pattern.Name, null,
result);
            }
            return
result;
        }

        /**
         * Returns a
look-ahead set with
all conflicts between
two
         * look-ahead
sets.
         *
         * @param
pattern        the
pattern name being
analyzed
         * @param
location       the
pattern location
         * @param set1
the first look-ahead
set
         * @param set2
the second look-ahead
set
         *
         * @return a
look-ahead set with
the conflicts found
         *
         * @throws
ParserCreationExceptio
n if an inherent
ambiguity was
         *
found among the look-
ahead sets
         */
        private
LookAheadSet
FindConflicts(string
pattern,

string location,

LookAheadSet set1,

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.
         *
         * @param
pattern        the
production pattern
         *
         * @return a
unified look-ahead set
         */
        private
LookAheadSet
FindUnion(ProductionPa
ttern pattern) {

LookAheadSet  result;
            int
length = 0;
            int
i;

            for (i =
0; i < pattern.Count;
i++) {
                result
=
pattern[i].LookAhead;
                if
(result.GetMaxLength()
> length) {

length =
result.GetMaxLength();
                }
            }
            result =
new
LookAheadSet(length);
            for (i =
0; i < pattern.Count;
i++) {

result.AddAll(pattern[
i].LookAhead);
            }

            return
result;
        }

        /**
         * Throws a
parse exception that
matches the specified
look-ahead
         * set. This
method will take into
account any initial
matching
         * tokens in
the look-ahead set.
         *
         * @param set
the look-ahead set to
match
         *
         * @throws
ParseException always
thrown by this method
         */
        private void
ThrowParseException(Lo
okAheadSet set) {
            Token
token;
            ArrayList
list = new
ArrayList();
            int[]
initials;

            // Read
tokens until mismatch
            while
(set.IsNext(this, 1))
{
                set =
set.CreateNextSet(Next
Token().Id);
            }

            // Find
next token
descriptions
            initials =
set.GetInitialTokens()
;
            for (int i
= 0; i <
initials.Length; i++)
{

list.Add(GetTokenDescr
iption(initials[i]));
            }

            // Create
exception
            token =
NextToken();
            throw new
ParseException(ParseEx
ception.ErrorType.UNEX
PECTED_TOKEN,

token.ToShortString(),

list,

token.StartLine,

token.StartColumn);
        }

        /**
         * Throws a
parser creation
```

```
exception for an
ambiguity. The
         * specified
look-ahead set
contains the token
conflicts to be
         * reported.
         *
         * @param
pattern        the
production pattern
name
         * @param
location       the
production pattern
location, or null
         * @param set
the look-ahead set
with conflicts
         *
         * @throws
ParserCreationExceptio
n always thrown by
this method
         */
        private void
ThrowAmbiguityExceptio
n(string pattern,

string location,

LookAheadSet set) {

            ArrayList
list = new
ArrayList();
            int[]
initials;

            // Find
next token
descriptions
            initials =
set.GetInitialTokens()
;
            for (int i
= 0; i <
initials.Length; i++)
{

list.Add(GetTokenDescr
iption(initials[i]));
            }

            // Create
exception
            throw new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INHERENT_A
MBIGUITY,

pattern,

location,
                list);
        }



        /**
         * A name
value stack. This
stack is used to
detect loops and
         * repetitions
of the same production
during look-ahead
analysis.
         */
        private class
CallStack {

            /**
             * A stack
with names.
             */
            private
ArrayList nameStack =
new ArrayList();

            /**
             * A stack
with values.
             */
            private
ArrayList valueStack =
new ArrayList();

            /**
             * Checks
if the specified name
is on the stack.
             *
             * @param
name           the
name to search for
             *
             * @return
true if the name is on
the stack, or
             *
false otherwise
             */
            public
bool Contains(string
name) {
                return
nameStack.Contains(nam
e);
            }

            /**
             * Checks
if the specified name
and value combination
is on
             * the
stack.
             *
             * @param
name           the
name to search for
             * @param
value          the
value to search for
             *
             * @return
true if the
combination is on the
stack, or
             *
false otherwise
             */
            public
bool Contains(string
name, int value) {
                for
(int i = 0; i <
nameStack.Count; i++)
{
                    if
(nameStack[i].Equals(n
ame)

&&
valueStack[i].Equals(v
alue)) {


return true;
                    }
                }
                return
false;
            }

            /**
             * Clears
the stack. This method
removes all elements
on
             * the
stack.
             */
            public
void Clear() {

nameStack.Clear();

valueStack.Clear();
            }

            /**
             * Adds a
new element to the top
of the stack.
             *
             * @param
name           the
stack name
             * @param
value          the
stack value
             */
            public
void Push(string name,
int value) {

nameStack.Add(name);

valueStack.Add(value);
            }

            /**
             * Removes
the top element of the
stack.
             */
            public
void Pop() {
                if
(nameStack.Count > 0)
{

nameStack.RemoveAt(nam
eStack.Count - 1);

valueStack.RemoveAt(va
lueStack.Count - 1);
                }
            }
        }
    }
}

namespace Core.Library
{
    /**
     * <remarks>An
enumeration with token
and production node
     *
constants.</remarks>
     */
    public enum
SyntaxConstants
    {
        MAIN_N = 1001,
        PRINT_N =
1002,
        SCAN_N = 1003,
        CONST_N =
1004,
        RETURN = 1005,
        SWITCH_N =
1006,
        CASE_N = 1007,
        BREAK = 1008,
        FOR_N = 1009,
        IF = 1010,
        ELSEIF_N =
1011,
        ELSE_N = 1012,
        DO = 1013,
        WHILE_N =
1014,
        VOID = 1015,
        GETCH = 1016,
        STRUCT_N =
1017,
        DEFAULT =
1018,
        PLUS = 1019,
        MINUS = 1020,
        TIMES = 1021,
        DIVIDE = 1022,
        MODULUS =
1023,
        EQUALS = 1024,
        SEMIC = 1025,
        DOT = 1026,
        COMMA = 1027,
        AND = 1028,
        OR = 1029,
        NOT = 1030,
        INCREMENT =
1031,
        DECREMENT =
1032,
        P_E = 1033,
        M_E = 1034,
        T_E = 1035,
        D_E = 1036,
        MOD_E = 1037,
        NEWLINE =
1038,
        N_E = 1039,
        O_PAREN =
1040,
        C_PAREN =
1041,
        D_QUOTE =
1042,
        COLON = 1043,
        O_BRACKET =
1044,
        C_BRACKET =
1045,
        GREATER =
1046,
        LESS = 1047,
        GREATER_E =
1048,
        LESS_E = 1049,
        NOT_E = 1050,
        S_OBRACKET =
1051,
        S_CBRACKET =
1052,
        DOLLAR = 1053,
        POWER = 1054,
        HASH = 1055,
        INT = 1056,
        CHAR = 1057,
        FLOAT = 1058,
        STRING = 1059,
        BOOL_N = 1060,
        ID = 1061,
        NUM = 1062,
        DECIMAL =
1063,
        S_CHAR = 1064,
        TEXT = 1065,
        COM = 1066,
        YES = 1067,
        NO = 1068,
        FUNCTNAME =
1069,
        STRUCTNAME =
1070,
        IDSTRUCT =
1071,
        F = 1072,
        D = 1073,
        S = 1074,
        ZERO = 1075,
        SPACE = 1076,
        N_LINE = 1077,
        WHITESPACE =
1078,

PROD_START_PROGRAM =
2001,
        PROD_PROGRAM =
2002,
        PROD_COMMENTS
= 2003,
        PROD_DATATYPE
= 2004,
        PROD_LITERALS
= 2005,
        PROD_CONSTANT
= 2006,

PROD_LOCAL_CHOICE =
2007,
        PROD_LOCALDEC
= 2008,

PROD_UNITADD_ID =
2009,

PROD_UNIT_EXINIT =
2010,
        PROD_MAIN =
2011,

PROD_GLOBAL_DEC =
2012,

PROD_LOCALDEC_CHOICE =
2013,

PROD_DEC_CHOICE =
2014,

PROD_GLOBAL_CHOICE =
2015,

PROD_BODY_CHOICE =
2016,

PROD_VAR_UNIT_BODY =
2017,

PROD_FUNCT_RETURN_BODY
= 2018,

PROD_FUNCT_VOID_BODY =
2019,

PROD_ARR_UNIT_BODY =
2020,
        PROD_ARR_TYPE
= 2021,
        PROD_N1 =
2022,

PROD_ARRAY_CHOICE =
2023,
        PROD_N2 =
2024,
        PROD_INDEX1 =
2025,
        PROD_ADD =
2026,
        PROD_INDEX2 =
2027,
        PROD_INDEX_EX
= 2028,
        PROD_UNIT_AID
= 2029,

PROD_UNIT_AIDTWO =
2030,
        PROD_UNIT_ELEM
= 2031,
        PROD_EXTELEM =
2032,

PROD_EXTELEM_CHOICE =
2033,

PROD_UNIT_ELEM_TWO =
2034,

PROD_ELEM_TWO_LIT =
2035,

PROD_ELEM_TWO_TAIL =
2036,

PROD_ASSIGN_CHOICE =
2037,

PROD_ACCESS_ASSIGN_DTY
PE = 2038,

PROD_ASSIGN_VALUE_CHOI
CE = 2039,
        PROD_ASSIGNING
= 2040,
        PROD_ARRAY_ID
= 2041,

PROD_ARRAY_IDTAIL =
2042,

PROD_ASSIGN_SYM =
2043,

PROD_ASSIGN_VALUE =
2044,

PROD_FUNCT_PARAM =
2045,

PROD_FUNCT_IDPARAM =
2046,

PROD_ADDFUNCT_IDPARAM
= 2047,
        PROD_FUNCT =
2048,

PROD_FUNCT_RETURN =
2049,

PROD_FUNCT_VOID =
2050,
        PROD_DTYPE_A =
2051,
        PROD_EXDTYPE_A
= 2052,
        PROD_DTYPEF =
2053,
        PROD_EX_ID =
2054,
        PROD_ARR_INDEX
= 2055,
        PROD_STRUCT_U
= 2056,
        PROD_S_DEC =
2057,
        PROD_INDEX =
2058,
        PROD_BODY =
2059,
        PROD_PRINT =
2060,
        PROD_POSTVAL =
2061,

PROD_CONCAT_LIT =
2062,
        PROD_SCAN =
2063,
        PROD_EXT_I =
2064,
        PROD_FOR_STATE
= 2065,

PROD_FORSTATEMENT =
2066,
        PROD_VAL1 =
2067,
        PROD_MNT_COND
= 2068,

PROD_MNT_COND_T =
2069,
        PROD_MNT =
2070,
        PROD_IFELSE =
2071,

PROD_IFCONDITION =
2072,

PROD_IFSTATEMENT =
2073,
        PROD_ELSEIF =
2074,

PROD_ELSEIFSTATEMENT =
2075,

PROD_ELSE_STATE =
2076,

PROD_ELSESTATEMENT =
2077,
        PROD_DOWHILE =
2078,

PROD_DOSTATEMENT =
2079,

PROD_WHILE_STATE =
2080,

PROD_WHILESTATEMENT =
2081,

PROD_SWITCH_STATE =
2082,

PROD_CASE_STATE =
2083,
        PROD_DEF =
2084,

PROD_CASESTATEMENT =
2085,
        PROD_MATH_OP =
2086,
        PROD_OPER_COND
= 2087,

PROD_OPER_COND_CHOICE
= 2088,
        PROD_OPER_SYM
= 2089,
        PROD_OPER_EQ =
2090,

PROD_OPER_EXT_S =
2091,

PROD_OPER_EXT_REP =
2092,
        PROD_OPERAND =
2093,

PROD_SIM_MATH_OP =
2094,

PROD_S_MATH_EXT =
2095,

PROD_OPER_COND_EXT =
2096,
        PROD_REL_OP =
2097,
        PROD_RELOP_EXT
= 2098,
        PROD_OP1 =
2099,
        PROD_LOG_OP =
2100,

PROD_EXT_LOG_OP =
2101,
        PROD_LOG_OPER
= 2102,
        PROD_END =
2103
    }
}

using
System.Collections.Gen
eric;

namespace Core.Library
{
    public class
SyntaxProductions
    {
        private string
Productions = "";
        private string
RecursiveProductions =
"";
        private
List<int>
ProductionCode = new
List<int>();
        private
List<string>
ProductionState = new
List<string>();

        public void
AddProductionCode(int
code)
        {

this.ProductionCode.Ad
d(code);
        }

        public int
GetLastProductionCode(
)
        {
            int last =
ProductionCode.Count -
1;
            return
this.ProductionCode[la
st];
        }

        public void
AddProductionState(str
ing state)
        {

this.ProductionState.A
dd(state);
        }

        public string
GetLastProductionState
()
        {
            int last =
ProductionState.Count
- 1;
            return
this.ProductionState[l
ast];
        }

        public
List<string>
GetAllProductionState(
)
        {
            return
this.ProductionState;
        }
        public
List<int>
GetAllProductionCode()
        {
            return
this.ProductionCode;
        }

        public void
AddProduction(string
Productions)
        {

this.Productions +=
Productions;
        }
        public string
GetProductions()
        {
            return
this.Productions;
        }
        public void
AddRecursiveProduction
(string
RecursiveProductions)
        {

this.RecursiveProducti
ons +=
RecursiveProductions;
        }
        public string
GetRecursiveProduction
s()
        {
            return
this.RecursiveProducti
ons;
        }
    }
}

/*
 * Token.cs
 */

using System.Text;

namespace Core.Library
{

    /**
     * A token node.
This class represents
a token (i.e. a set of
adjacent
     * characters) in
a parse tree. The
tokens are created by
a tokenizer,
     * that groups
characters together
into tokens according
to a set of
     * token patterns.
     *

     *
     */
    public class Token
: Node {

        /**
         * The token
pattern used for this
token.
         */
        private
TokenPattern pattern;

        /**
         * The
characters that
constitute this token.
This is normally
         * referred to
as the token image.
         */
        private string
image;

        /**
         * The line
number of the first
character in the token
image.
         */
        private int
startLine;

        /**
         * The column
number of the first
character in the token
image.
         */
        private int
startColumn;

        /**
         * The line
number of the last
character in the token
image.
         */
        private int
endLine;

        /**
         * The column
number of the last
character in the token
image.
         */
        private int
endColumn;

        /**
         * The
previous token in the
list of tokens.
         */
        private Token
previous = null;

        /**
         * The next
token in the list of
tokens.
         */
        private Token
next = null;

        /**
         * Creates a
new token.
         *
         * @param
pattern        the
token pattern
         * @param
image          the
token image (i.e.
characters)
         * @param line
the line number of the
first character
         * @param col
the column number of
the first character
         */
        public
Token(TokenPattern
pattern, string image,
int line, int col) {

this.pattern =
pattern;
            this.image
= image;

this.startLine = line;

this.startColumn =
col;

this.endLine = line;

this.endColumn = col +
image.Length - 1;
             for (int
pos = 0;
image.IndexOf('\n',
pos) >= 0;) {
                pos =
image.IndexOf('\n',
pos) + 1;

this.endLine++;

endColumn =
image.Length - pos;
            }
        }

        /**
         * The node
type id property
(read-only). This
value is set as
         * a unique
identifier for each
type of node, in order
to
         * simplify
later identification.
         *
         *
         */
        public
override int Id {
            get {
                return
pattern.Id;
            }
        }

        /**
         * The node
name property (read-
only).
         *
         *
         */
        public
override string Name {
            get {
                return
pattern.Name;
            }
        }

        /**
         * The line
number property of the
first character in
this
         * node (read-
only). If the node has
child elements, this
         * value will
be fetched from the
first child.
         *
         *
         */
        public
override int StartLine
{
            get {
                return
startLine;
            }
        }

        /**
         * The column
number property of the
first character in
this
         * node (read-
only). If the node has
child elements, this
         * value will
be fetched from the
first child.
         *
         *
         */
        public
override int
StartColumn {
            get {
                return
startColumn;
            }
        }

        /**
         * The line
number property of the
last character in this
node
         * (read-
only). If the node has
child elements, this
value
         * will be
fetched from the last
child.
         *
         *
         */
        public
override int EndLine {
            get {
                return
endLine;
            }
        }

        /**
         * The column
number property of the
last character in this
```

```
         * node (read-
only). If the node has
child elements, this
         * value will
be fetched from the
last child.
         *
         *
         */
        public
override int EndColumn
{
            get {
                return
endColumn;
            }
        }

        /**
         * The token
image property (read-
only). The token image
         * consists of
the input characters
matched to form this
         * token.
         *
         *
         */
        public string
Image {
            get {
                return
image;
            }
        }

        /**
         * Returns the
token image. The token
image consists of the
         * input
characters matched to
form this token.
         *
         * @return the
token image
         *
         * @see #Image
         *
         * @deprecated
Use the Image property
instead.
         */
        public string
GetImage() {
            return
Image;
        }

        /**
         * The token
pattern property
(read-only).
         */
        internal
TokenPattern Pattern {
            get {
                return
pattern;
            }
        }

        /**
         * The
previous token
property. If the token
list feature is
         * used in the
tokenizer, all tokens
found will be chained
         * together in
a double-linked list.
The previous token may
be
         * a token
that was ignored
during the parsing,
due to it's
         * ignore flag
being set. If there is
no previous token or
if
         * the token
list feature wasn't
used in the tokenizer
(the
         * default),
the previous token
will always be null.
         *
         * @see #Next
         * @see
Tokenizer#UseTokenList
         *
         *
         */
        public Token
Previous {
            get {
                return
previous;
            }
            set {
                if
(previous != null) {

previous.next = null;
                }

previous = value;
                if
(previous != null) {

previous.next = this;
                }
            }
        }

        /**
         * Returns the
previous token. The
previous token may be
a token
         * that has
been ignored in the
parsing. Note that if
the token
         * list
feature hasn't been
used in the tokenizer,
this method
         * will always
return null. By
default the token list
feature is
         * not used.
         *
         * @return the
previous token, or
         *
null if no such token
is available
         *
         * @see
#Previous
         * @see
#GetNextToken
         * @see
Tokenizer#UseTokenList
         *
         *
         *
         * @deprecated
Use the Previous
property instead.
         */
        public Token
GetPreviousToken() {
            return
Previous;
        }

        /**
         * The next
token property. If the
token list feature is
used
         * in the
tokenizer, all tokens
found will be chained
together
         * in a
double-linked list.
The next token may be
a token that
         * was ignored
during the parsing,
due to it's ignore
flag
         * being set.
If there is no next
token or if the token
list
         * feature
wasn't used in the
tokenizer (the
default), the
         * next token
will always be null.
         *
         * @see
#Previous
         * @see
Tokenizer#UseTokenList
         *
         *
         */
        public Token
Next {
            get {
                return
next;
            }
            set {
                if
(next != null) {

next.previous = null;
                }
                next =
value;
                if
(next != null) {

next.previous = this;
                }
            }
        }

        /**
         * Returns the
next token. The next
token may be a token
that has
         * been
ignored in the
parsing. Note that if
the token list
         * feature
hasn't been used in
the tokenizer, this
method will
         * always
return null. By
```

```
default the token list
feature is not
         * used.
         *
         * @return the
next token, or
         *
null if no such token
is available
         *
         * @see #Next
         * @see
#GetPreviousToken
         * @see
Tokenizer#UseTokenList
         *
         *
         *
         * @deprecated
Use the Next property
instead.
         */
        public Token
GetNextToken() {
            return
Next;
        }

        /**
         * Returns a
string representation
of this token.
         *
         * @return a
string representation
of this token
         */
        public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();
            int
newline =
image.IndexOf('\n');


buffer.Append(pattern.
Name);

buffer.Append("(");

buffer.Append(pattern.
Id);

buffer.Append("):
\"");
            if
(newline >= 0) {
                if
(newline > 0 &&
image[newline - 1] ==
'\r') {

newline--;
                }

buffer.Append(image.Su
bstring(0, newline));

buffer.Append("(...)")
;
            } else {

buffer.Append(image);
            }

buffer.Append("\",
line: ");

buffer.Append(startLin
e);

buffer.Append(", col:
");

buffer.Append(startCol
umn);

            return
buffer.ToString();
        }

        /**
         * Returns a
short string
representation of this
token. The
         * string will
only contain the token
image and possibly the
         * token
pattern name.
         *
         * @return a
short string
representation of this
token
         */
        public string
ToShortString() {

StringBuilder  buffer
= new StringBuilder();
            int
newline =
image.IndexOf('\n');


buffer.Append('"');
            if
(newline >= 0) {
                if
(newline > 0 &&
image[newline - 1] ==
'\r') {

newline--;
                }

buffer.Append(image.Su
bstring(0, newline));

buffer.Append("(...)")
;
            } else {

buffer.Append(image);
            }

buffer.Append('"');
            if
(pattern.Type ==
TokenPattern.PatternTy
pe.REGEXP) {

buffer.Append(" <");

buffer.Append(pattern.
Name);

buffer.Append(">");
            }

            return
buffer.ToString();
        }
    }
}

/*
 * Tokenizer.cs
 */

using System;
using
System.Collections;
using System.IO;
using System.Text;
using
System.Text.RegularExp
ressions;
using Core.Library.RE;

namespace Core.Library
{

    /**
     * A character
stream tokenizer. This
class groups the
characters read
     * from the stream
together into tokens
("words"). The
grouping is
     * controlled by
token patterns that
contain either a fixed
string to
     * search for, or
a regular expression.
If the stream of
characters
     * don't match any
of the token patterns,
a parse exception is
thrown.
     *

     *
     */
    public class
Tokenizer {

        /**
         * The token
list feature flag.
         */
        private bool
useTokenList = false;

        /**
         * The string
DFA token matcher.
This token matcher
uses a
         *
deterministic finite
automaton (DFA)
implementation and is
         * used for
all string token
patterns. It has a
slight speed
         * advantage
to the NFA
implementation, but
should be equivalent
         * on memory
usage.
         */
        private
StringDFAMatcher
stringDfaMatcher;

        /**
         * The regular
expression NFA token
matcher. This token
matcher
         * uses a non-
deterministic finite
automaton (DFA)
implementation
```

```
         * and is used
for most regular
expression token
patterns. It is
         * somewhat
faster than the other
recursive regular
expression
         *
implementations
available, but doesn't
support the full
         * syntax. It
conserves memory by
using a fast queue
instead of
         * the stack
during processing (no
stack overflow).
         */
        private
NFAMatcher nfaMatcher;

        /**
         * The regular
expression token
matcher. This token
matcher is
         * used for
complex regular
expressions, but
should be avoided
         * due to
possibly degraded
speed and memory usage
compared to
         * the
automaton
implementations.
         */
        private
RegExpMatcher
regExpMatcher;

        /**
        * The
character stream
reader buffer.
        */
        private
ReaderBuffer buffer =
null;

        /**
        * The last
token match found.
        */
        private
TokenMatch lastMatch =
new TokenMatch();

        /**
         * The
previous token in the
token list.
         */
        private Token
previousToken = null;

        /**
         * Creates a
new case-sensitive
tokenizer for the
specified
         * input
stream.
         *
         * @param
input          the
input stream to read
         */
        public
Tokenizer(TextReader
input)
            :
this(input, false) {
        }

        /**
         * Creates a
new tokenizer for the
specified input
stream. The
         * tokenizer
can be set to process
tokens either in
         * case-
sensitive or case-
insensitive mode.
         *
         * @param
input          the
input stream to read
         * @param
ignoreCase     the
character case ignore
flag
         *
         *
         */
        public
Tokenizer(TextReader
input, bool
ignoreCase) {

this.stringDfaMatcher
= new
StringDFAMatcher(ignor
eCase);

this.nfaMatcher = new
NFAMatcher(ignoreCase)
;

this.regExpMatcher =
new
RegExpMatcher(ignoreCa
se);

this.buffer = new
ReaderBuffer(input);
        }

        /**
         * The token
list flag property. If
the token list flag is
         * set, all
tokens (including
ignored tokens) link
to each
         * other in a
double-linked list. By
default the token list
         * flag is set
to false.
         *
         * @see
Token#Previous
         * @see
Token#Next
         *
         *
         */
        public bool
UseTokenList {
            get {
                return
useTokenList;
            }
            set {

useTokenList = value;
            }
        }

        /**
         * Checks if
the token list feature
is used. The token
list
         * feature
makes all tokens
(including ignored
tokens) link to
         * each other
in a linked list. By
default the token list
feature
         * is not
used.
         *
         * @return
true if the token list
feature is used, or
         *
false otherwise
         *
         * @see
#UseTokenList
         * @see
#SetUseTokenList
         * @see
Token#GetPreviousToken
         * @see
Token#GetNextToken
         *
         *
         *
         * @deprecated
Use the UseTokenList
property instead.
         */
        public bool
GetUseTokenList() {
            return
useTokenList;
        }

        /**
         * Sets the
token list feature
flag. The token list
feature makes
         * all tokens
(including ignored
tokens) link to each
other in a
         * linked list
when active. By
default the token list
feature is
         * not used.
         *
         * @param
useTokenList   the
token list feature
flag
         *
         * @see
#UseTokenList
         * @see
#GetUseTokenList
         * @see
Token#GetPreviousToken
         * @see
Token#GetNextToken
         *
         *
         *
         * @deprecated
Use the UseTokenList
property instead.
         */
        public void
SetUseTokenList(bool
useTokenList) {
```

```
this.useTokenList =
useTokenList;
        }

        /**
         * Returns a
description of the
token pattern with the
         * specified
id.
         *
         * @param id
the token pattern id
         *
         * @return the
token pattern
description, or
         *
null if not present
         */
        public string
GetPatternDescription(
int id) {

TokenPattern  pattern;

            pattern =
stringDfaMatcher.GetPa
ttern(id);
            if
(pattern == null) {

pattern =
nfaMatcher.GetPattern(
id);
            }
            if
(pattern == null) {

pattern =
regExpMatcher.GetPatte
rn(id);
            }
            return
(pattern == null) ?
null :
pattern.ToShortString(
);
        }

        /**
         * Returns the
current line number.
This number will be
the line
         * number of
the next token
returned.
         *
         * @return the
current line number
         */
        public int
GetCurrentLine() {
            return
buffer.LineNumber;
        }

        /**
         * Returns the
current column number.
This number will be
the
         * column
number of the next
token returned.
         *
         * @return the
current column number
         */
        public int
GetCurrentColumn() {
            return
buffer.ColumnNumber;
        }

        /**
         * Adds a new
token pattern to the
tokenizer. The pattern
will be
         * added last
in the list, choosing
a previous token
pattern in
         * case two
matches the same
string.
         *
         * @param
pattern        the
pattern to add
         *
         * @throws
ParserCreationExceptio
n if the pattern
couldn't be
         *
added to the tokenizer
         */
        public void
AddPattern(TokenPatter
n pattern) {
            switch
(pattern.Type) {
            case
TokenPattern.PatternTy
pe.STRING:
                try {

stringDfaMatcher.AddPa
ttern(pattern);
                }
catch (Exception e) {

throw new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_TO
KEN,

pattern.Name,

"error adding string
token: " +

e.Message);
                }
                break;
            case
TokenPattern.PatternTy
pe.REGEXP:
                try {

nfaMatcher.AddPattern(
pattern);
                }
catch (Exception) {

try {

regExpMatcher.AddPatte
rn(pattern);
                    }
catch (Exception e) {

throw new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_TO
KEN,

pattern.Name,

"regular expression
contains error(s): " +

e.Message);
                    }
                }
                break;
            default:
                throw
new
ParserCreationExceptio
n(

ParserCreationExceptio
n.ErrorType.INVALID_TO
KEN,

pattern.Name,

"pattern type " +
pattern.Type +
                    "
is undefined");
            }
        }

        /**
         * Resets this
tokenizer for usage
with another input
stream.
         * This method
will clear all the
internal state in the
         * tokenizer
as well as close the
previous input stream.
It
         * is normally
called in order to
reuse a parser and
         * tokenizer
pair with multiple
input streams, thereby
         * avoiding
the cost of re-
analyzing the grammar
structures.
         *
         * @param
input          the new
input stream to read
         *
         * @see
Parser#reset(Reader)
         *
         *
         */
        public void
Reset(TextReader
input) {

this.buffer.Dispose();

this.buffer = new
ReaderBuffer(input);

this.previousToken =
null;

this.lastMatch.Clear()
;
        }

        /**
         * Finds the
next token on the
```

```
stream. This method
will return
         * null when
end of file has been
reached. It will
return a
         * parse
exception if no token
matched the input
stream, or if
         * a token
pattern with the error
flag set matched. Any
tokens
         * matching a
token pattern with the
ignore flag set will
be
         * silently
ignored and the next
token will be
returned.
         *
         * @return the
next token found, or
         *
null if end of file
was encountered
         *
         * @throws
ParseException if the
input stream couldn't
be read or
         *
parsed correctly
         */
        public Token
Next() {
            Token
token = null;

            do {
                token
= NextToken();
                if
(token == null) {

previousToken = null;

return null;
                }
                if
(useTokenList) {

token.Previous =
previousToken;

previousToken = token;
                }
                if
(token.Pattern.Ignore)
{

token = null;
                } else
if
(token.Pattern.Error)
{

throw new
ParseException(

ParseException.ErrorTy
pe.INVALID_TOKEN,

token.Pattern.ErrorMes
sage,

token.StartLine,

token.StartColumn);
                }
            } while
(token == null);
            return
token;
        }


        /**
         * Finds the
next token on the
stream. This method
will return
         * null when
end of file has been
reached. It will
return a
         * parse
exception if no token
matched the input
stream.
         *
         * @return the
next token found, or
         *
null if end of file
was encountered
         *
         * @throws
ParseException if the
input stream couldn't
be read or
         *
parsed correctly
         */
        private Token
NextToken() {
            string
str;
            int
line;
            int
column;

            try {

lastMatch.Clear();

stringDfaMatcher.Match
(buffer, lastMatch);

nfaMatcher.Match(buffe
r, lastMatch);

regExpMatcher.Match(bu
ffer, lastMatch);
                if
(lastMatch.Length > 0)
{

line =
buffer.LineNumber;

column =
buffer.ColumnNumber;

str =
buffer.Read(lastMatch.
Length);

return
NewToken(lastMatch.Pat
tern, str, line,
column);
                } else
if (buffer.Peek(0) <
0) {

return null;
                } else
{

line =
buffer.LineNumber;

column =
buffer.ColumnNumber;

throw new
ParseException(

ParseException.ErrorTy
pe.UNEXPECTED_CHAR,

buffer.Read(1),

line,

column);
                }
            } catch
(IOException e) {
                throw
new
ParseException(ParseEx
ception.ErrorType.IO,

e.Message,

-1,

-1);
            }
        }

        /**
         * Factory
method for creating a
new token. This method
can be
         * overridden
to provide other token
implementations than
the
         * default
one.
         *
         * @param
pattern        the
token pattern
         * @param
image          the
token image (i.e.
characters)
         * @param line
the line number of the
first character
         * @param
column         the
column number of the
first character
         *
         * @return the
token created
         *
         *
         */
        protected
virtual Token
NewToken(TokenPattern
pattern,

string image,

int line,

int column) {

            return new
Token(pattern, image,
line, column);
        }

        /**
         * Returns a
string representation
of this object. The
returned
```

```
         * string will
contain the details of
all the token patterns
         * contained
in this tokenizer.
         *
         * @return a
detailed string
representation
         */
        public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();


buffer.Append(stringDf
aMatcher);

buffer.Append(nfaMatch
er);

buffer.Append(regExpMa
tcher);
            return
buffer.ToString();
        }
    }



    /**
     * A token pattern
matcher. This class is
the base class for the
     * various types
of token matchers that
exist. The token
matcher
     * checks for
matches with the
tokenizer buffer, and
maintains the
     * state of the
last match.
     */
    internal abstract
class TokenMatcher {

        /**
         * The array
of token patterns.
         */
        protected
TokenPattern[]
patterns = new
TokenPattern[0];

        /**
         * The ignore
character case flag.
         */
        protected bool
ignoreCase = false;

        /**
         * Creates a
new token matcher.
         *
         * @param
ignoreCase      the
character case ignore
flag
         */
        public
TokenMatcher(bool
ignoreCase) {

this.ignoreCase =
ignoreCase;
        }

        /**
         * Searches
for matching token
patterns at the start
of the
         * input
stream. If a match is
found, the token match
object
         * is updated.
         *
         * @param
buffer         the
input buffer to check
         * @param
match          the
token match to update
         *
         * @throws
IOException if an I/O
error occurred
         */
        public
abstract void
Match(ReaderBuffer
buffer, TokenMatch
match);

        /**
         * Returns the
token pattern with the
specified id. Only
         * token
patterns handled by
this matcher can be
returned.
         *
         * @param id
the token pattern id
         *
         * @return the
token pattern found,
or
         *
null if not found
         */
        public
TokenPattern
GetPattern(int id) {
            for (int i
= 0; i <
patterns.Length; i++)
{
                if
(patterns[i].Id == id)
{

return patterns[i];
                }
            }
            return
null;
        }

        /**
         * Adds a
string token pattern
to this matcher.
         *
         * @param
pattern        the
pattern to add
         *
         * @throws
Exception if the
pattern couldn't be
added to the matcher
         */
        public virtual
void
AddPattern(TokenPatter
n pattern) {

Array.Resize(ref
patterns,
patterns.Length + 1);

patterns[patterns.Leng
th - 1] = pattern;
        }

        /**
         * Returns a
string representation
of this matcher. This
will
         * contain all
the token patterns.
         *
         * @return a
detailed string
representation of this
matcher
         */
        public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();

            for (int i
= 0; i <
patterns.Length; i++)
{

buffer.Append(patterns
[i]);

buffer.Append("\n\n");
            }
            return
buffer.ToString();
        }
    }



    /**
     * A token pattern
matcher using a DFA
for string tokens.
This
     * class only
supports string tokens
and must be
complemented
     * with another
matcher for regular
expressions.
Internally it
     * uses a DFA to
provide high
performance.
     */
    internal class
StringDFAMatcher :
TokenMatcher {

        /**
         * The
deterministic finite
state automaton used
for
         * matching.
         */
        private
TokenStringDFA
automaton = new
TokenStringDFA();

        /**
```

```
         * Creates a
new string token
matcher.
         *
         * @param
ignoreCase      the
character case ignore
flag
         */
        public
StringDFAMatcher(bool
ignoreCase) :
base(ignoreCase) {
        }

        /**
         * Adds a
string token pattern
to this matcher.
         *
         * @param
pattern        the
pattern to add
         */
        public
override void
AddPattern(TokenPatter
n pattern) {

automaton.AddMatch(pat
tern.Pattern,
ignoreCase, pattern);

base.AddPattern(patter
n);
        }

        /**
         * Searches
for matching token
patterns at the start
of the
         * input
stream. If a match is
found, the token match
object
         * is updated.
         *
         * @param
buffer         the
input buffer to check
         * @param
match          the
token match to update
         *
         * @throws
IOException if an I/O
error occurred
         */
        public
override void
Match(ReaderBuffer
buffer, TokenMatch
match) {

TokenPattern  res =
automaton.Match(buffer
, ignoreCase);

            if (res !=
null) {

match.Update(res.Patte
rn.Length, res);
            }
        }
    }


    /**
     * A token pattern
matcher using a NFA
for both string and
     * regular
expression tokens.
This class has limited
support for
     * regular
expressions and must
be complemented with
another
     * matcher
providing full regular
expression support.
Internally
     * it uses a NFA
to provide high
performance and low
memory
     * usage.
     */
    internal class
NFAMatcher :
TokenMatcher {

        /**
         * The non-
deterministic finite
state automaton used
for
         * matching.
         */
        private
TokenNFA automaton =
new TokenNFA();

        /**
         * Creates a
new NFA token matcher.
         *
         * @param
ignoreCase      the
character case ignore
flag
         */
        public
NFAMatcher(bool
ignoreCase) :
base(ignoreCase) {
        }

        /**
         * Adds a
token pattern to this
matcher.
         *
         * @param
pattern        the
pattern to add
         *
         * @throws
Exception if the
pattern couldn't be
added to the matcher
         */
        public
override void
AddPattern(TokenPatter
n pattern) {
            if
(pattern.Type ==
TokenPattern.PatternTy
pe.STRING) {

automaton.AddTextMatch
(pattern.Pattern,
ignoreCase, pattern);
            } else {

automaton.AddRegExpMat
ch(pattern.Pattern,
ignoreCase, pattern);
            }

base.AddPattern(patter
n);
        }

        /**
         * Searches
for matching token
patterns at the start
of the
         * input
stream. If a match is
found, the token match
object
         * is updated.
         *
         * @param
buffer         the
input buffer to check
         * @param
match          the
token match to update
         *
         * @throws
IOException if an I/O
error occurred
         */
        public
override void
Match(ReaderBuffer
buffer, TokenMatch
match) {

automaton.Match(buffer
, match);
        }
    }


    /**
     * A token pattern
matcher for complex
regular expressions.
This
     * class only
supports regular
expression tokens and
must be
     * complemented
with another matcher
for string tokens.
     * Internally it
uses the Grammatica RE
package for high
     * performance or
the native
java.util.regex
package for maximum
     * compatibility.
     */
    internal class
RegExpMatcher :
TokenMatcher {

        /**
         * The regular
expression handlers.
         */
        private
REHandler[] regExps =
new REHandler[0];

        /**
         * Creates a
new regular expression
token matcher.
         *
         * @param
ignoreCase      the
character case ignore
flag
         */
        public
RegExpMatcher(bool
ignoreCase) :
base(ignoreCase) {
        }

        /**
         * Adds a
regular expression
token pattern to this
matcher.
         *
         * @param
pattern        the
pattern to add
         *
         * @throws
Exception if the
pattern couldn't be
added to the matcher
         */
        public
override void
AddPattern(TokenPatter
n pattern) {
            REHandler
re;

            try {
                re =
new
GrammaticaRE(pattern.P
attern, ignoreCase);

pattern.DebugInfo =
"Grammatica regexp\n"
+ re;
            } catch
(Exception) {
                re =
new
SystemRE(pattern.Patte
rn, ignoreCase);

pattern.DebugInfo =
"native .NET regexp";
            }

Array.Resize(ref
regExps,
regExps.Length + 1);

regExps[regExps.Length
- 1] = re;

base.AddPattern(patter
n);
        }

        /**
         * Searches
for matching token
patterns at the start
of the
```

```
         * input
stream.  If a match is
found, the token match
object
         * is updated.
         *
         * @param
buffer         the
input buffer to check
         * @param
match          the
token match to update
         *
         * @throws
IOException if an I/O
error occurred
         */
        public
override void
Match(ReaderBuffer
buffer, TokenMatch
match) {
            for (int i
= 0; i <
regExps.Length; i++) {
                int
length =
regExps[i].Match(buffe
r);
                if
(length > 0) {

match.Update(length,
patterns[i]);
                }
            }
        }
    }



    /**
     * The regular
expression handler
base class.
     */
    internal abstract
class REHandler {

        /**
         * Checks if
the start of the input
stream matches this
         * regular
expression.
         *
         * @param
buffer         the
input buffer to check
         *
         * @return the
longest match found,
or
```

```
         *
zero (0) if no match
was found
         *
         * @throws
IOException if an I/O
error occurred
         */
        public
abstract int
Match(ReaderBuffer
buffer);
    }



    /**
     * The Grammatica
built-in regular
expression handler.
     */
    internal class
GrammaticaRE :
REHandler {

        /**
         * The
compiled regular
expression.
         */
        private RegExp
regExp;

        /**
         * The regular
expression matcher to
use.
         */
        private
Matcher matcher =
null;

        /**
         * Creates a
new Grammatica regular
expression handler.
         *
         * @param
regex          the
regular expression
text
         * @param
ignoreCase      the
character case ignore
flag
         *
         * @throws
Exception if the
regular expression
contained
         *
invalid syntax
         */
```

```
        public
GrammaticaRE(string
regex, bool
ignoreCase) {
            regExp =
new RegExp(regex,
ignoreCase);
        }

        /**
         * Checks if
the start of the input
stream matches this
         * regular
expression.
         *
         * @param
buffer         the
input buffer to check
         *
         * @return the
longest match found,
or
         *
zero (0) if no match
was found
         *
         * @throws
IOException if an I/O
error occurred
         */
        public
override int
Match(ReaderBuffer
buffer) {
            if
(matcher == null) {

matcher =
regExp.Matcher(buffer)
;
            } else {

matcher.Reset(buffer);
            }
            return
matcher.MatchFromBegin
ning() ?
matcher.Length() : 0;
        }
    }



    /**
     * The .NET system
regular expression
handler.
     */
    internal class
SystemRE : REHandler {

        /**
```

```
         * The parsed
regular expression.
         */
        private Regex
reg;

        /**
         * Creates a
new .NET system
regular expression
handler.
         *
         * @param
regex          the
regular expression
text
         * @param
ignoreCase      the
character case ignore
flag
         *
         * @throws
Exception if the
regular expression
contained
         *
invalid syntax
         */
        public
SystemRE(string regex,
bool ignoreCase) {
            if
(ignoreCase) {
                reg =
new Regex(regex,
RegexOptions.IgnoreCas
e);
            } else {
                reg =
new Regex(regex);
            }
        }

        /**
         * Checks if
the start of the input
stream matches this
         * regular
expression.
         *
         * @param
buffer         the
input buffer to check
         *
         * @return the
longest match found,
or
         *
zero (0) if no match
was found
         *
         * @throws
IOException if an I/O
error occurred
         */
        public
override int
Match(ReaderBuffer
buffer) {
            Match  m;

            // Ugly
hack since .NET
doesn't have a flag
for when the
            // end of
the input string was
encountered...

buffer.Peek(1024 *
16);
            // Also,
there is no API to
limit the search to
the specified
            //
position, so we
double-check the index
afterwards instead.
            m =
reg.Match(buffer.ToStr
ing(),
buffer.Position);
            if
(m.Success && m.Index
== buffer.Position) {
                return
m.Length;
            } else {
                return
0;
            }
        }
    }
}

/*
 * TokenMatch.cs
 */

namespace Core.Library
{

    /**
     * The token match
status. This class
contains logic to
ensure that
     * only the
longest match is
considered. It also
prefers lower token
     * pattern
identifiers if two
matches have the same
length.
     *

     *
     *
     */
    internal class
TokenMatch {

        /**
         * The length
of the longest match.
         */
        private int
length = 0;

        /**
         * The pattern
in the longest match.
         */
        private
TokenPattern pattern =
null;

        /**
         * Clears the
current match
information.
         */
        public void
Clear() {
            length =
0;
            pattern =
null;
        }

        /**
         * The length
of the longest match
found (read-only).
         */
        public int
Length {
            get {
                return
length;
            }
        }

        /**
         * The token
pattern for the
longest match found
(read-only).
         */
        public
TokenPattern Pattern {
            get {
                return
pattern;
            }
        }

        /**
         * Updates
this match with new
values. The new values
will only
         * be
considered if the
length is longer than
any previous match
         * found.
         *
         * @param
length         the
matched length
         * @param
pattern        the
matched pattern
         */
        public void
Update(int length,
TokenPattern pattern)
{
            if
(this.length < length)
{

this.length = length;

this.pattern =
pattern;
            } else if
(this.length == length
&& this.pattern.Id >
pattern.Id) {

this.length = length;

this.pattern =
pattern;
            }
        }
    }
}

/*
 * TokenNFA.cs
 */

using System;

namespace Core.Library
{

    /**
     * A non-
deterministic finite
```

```
state automaton (NFA)
for matching
     * tokens. It
supports both fixed
strings and simple
regular
     * expressions,
but should perform
similar to a DFA due
to highly
     * optimized data
structures and tuning.
The memory footprint
during
     * matching should
be near zero, since no
heap memory is
allocated
     * unless the pre-
allocated queues need
to be enlarged. The
NFA also
     * does not use
recursion, but
iterates in a loop
instead.
     *
     *
     *
     */
    internal class
TokenNFA {

        /**
         * The initial
state lookup table,
indexed by the first
ASCII
         * character.
This array is used to
for speed optimizing
the
         * first step
in the match, since
the initial state
would
         * otherwise
have a long list of
transitions to
consider.
         */
        private
NFAState[] initialChar
= new NFAState[128];

        /**
         * The initial
state. This state
contains any
transitions not
         * already
stored in the initial
text state array, i.e.
non-ASCII
         * or complex
transitions (such as
regular expressions).
         */
        private
NFAState initial = new
NFAState();

        /**
         * The NFA
state queue to use.
         */
        private
NFAStateQueue queue =
new NFAStateQueue();

        /**
         * Adds a
string match to this
automaton. New states
and
         * transitions
will be added to
extend this automaton
to support
         * the
specified string.
         *
         * @param str
the string to match
         * @param
ignoreCase     the
case-insensitive match
flag
         * @param
value          the
match value
         */
        public void
AddTextMatch(string
str, bool ignoreCase,
TokenPattern value) {
            NFAState
state;
            char
ch = str[0];

            if (ch <
128 && !ignoreCase) {
                state
= initialChar[ch];
                if
(state == null) {

state =
initialChar[ch] = new
NFAState();
                }
            } else {
                state
= initial.AddOut(ch,
ignoreCase, null);
            }
            for (int i
= 1; i < str.Length;
i++) {
                state
= state.AddOut(str[i],
ignoreCase, null);
            }

state.value = value;
        }

        /**
         * Adds a
regular expression
match to this
automaton. New states
         * and
transitions will be
added to extend this
automaton to
         * support the
specified string. Note
that this method only
         * supports a
subset of the full
regular expression
syntax, so
         * a more
complete regular
expression library
must also be
         * provided.
         *
         * @param
pattern        the
regular expression
string
         * @param
ignoreCase     the
case-insensitive match
flag
         * @param
value          the
match value
         *
         * @throws
RegExpException if the
regular expression
parsing
         *
failed
         */
        public void
AddRegExpMatch(string
pattern,

bool ignoreCase,

TokenPattern value) {


TokenRegExpParser
parser = new
TokenRegExpParser(patt
ern, ignoreCase);
            string
debug = "DFA regexp; "
+
parser.GetDebugInfo();
            bool
isAscii;

            isAscii =
parser.start.IsAsciiOu
tgoing();
            for (int i
= 0; isAscii && i <
128; i++) {
                bool
match = false;
                for
(int j = 0; j <
parser.start.outgoing.
Length; j++) {
                    if
(parser.start.outgoing
[j].Match((char) i)) {

if (match) {

isAscii = false;

break;

}

match = true;
                    }
                }
                if
(match &&
initialChar[i] !=
null) {

isAscii = false;
                }
            }
            if
(parser.start.incoming
.Length > 0) {

initial.AddOut(new
NFAEpsilonTransition(p
arser.start));
                debug
+= ", uses initial
epsilon";
```

```
            } else if
(isAscii &&
!ignoreCase) {
                for
(int i = 0; isAscii &&
i < 128; i++) {

for (int j = 0; j <
parser.start.outgoing.
Length; j++) {

if
(parser.start.outgoing
[j].Match((char) i)) {

initialChar[i] =
parser.start.outgoing[
j].state;

}
                    }
                }
                debug
+= ", uses ASCII
lookup";
            } else {

parser.start.MergeInto
(initial);
                debug
+= ", uses initial
state";
            }

parser.end.value =
value;

value.DebugInfo =
debug;
        }

        /**
         * Checks if
this NFA matches the
specified input text.
The
         * matching
will be performed from
position zero (0) in
the
         * buffer.
This method will not
read any characters
from the
         * stream,
just peek ahead.
         *
         * @param
buffer         the
input buffer to check
         * @param
match          the
token match to update
         *
         * @return the
number of characters
matched, or
         *
zero (0) if no match
was found
         *
         * @throws
IOException if an I/O
error occurred
         */
        public int
Match(ReaderBuffer
buffer, TokenMatch
match) {
            int
length = 0;
            int
pos = 1;
            int
peekChar;
            NFAState
state;

            // The
first step of the
match loop has been
unrolled and
            //
optimized for
performance below.

this.queue.Clear();
            peekChar =
buffer.Peek(0);
            if (0 <=
peekChar && peekChar <
128) {
                state
=
this.initialChar[peekC
har];
                if
(state != null) {

this.queue.AddLast(sta
te);
                }
            }
            if
(peekChar >= 0) {

this.initial.MatchTran
sitions((char)
peekChar, this.queue,
true);
            }

this.queue.MarkEnd();
            peekChar =
buffer.Peek(1);

            // The
remaining match loop
processes all
subsequent states
            while
(!this.queue.Empty) {
                if
(this.queue.Marked) {

pos++;

peekChar =
buffer.Peek(pos);

this.queue.MarkEnd();
                }
                state
=
this.queue.RemoveFirst
();
                if
(state.value != null)
{

match.Update(pos,
state.value);
                }
                if
(peekChar >= 0) {

state.MatchTransitions
((char) peekChar,
this.queue, false);
                }
            }
            return
length;
        }
    }



    /**
     * An NFA state.
The NFA consists of a
series of states, each
     * having zero or
more transitions to
other states.
     */
    internal class
NFAState {


        /**
        * The optional
state value (if it is
a final state).
         */
        internal
TokenPattern value =
null;

        /**
        * The incoming
transitions to this
state.
        */
        internal
NFATransition[]
incoming = new
NFATransition[0];

        /**
        * The outgoing
transitions from this
state.
        */
        internal
NFATransition[]
outgoing = new
NFATransition[0];

        /**
        * The outgoing
epsilon transitions
flag.
        */
        internal bool
epsilonOut = false;

        /**
         * Checks if
this state has any
incoming or outgoing
         *
transitions.
         *
         * @return
true if this state has
transitions, or
         *
false otherwise
         */
        public bool
HasTransitions() {
            return
incoming.Length > 0 ||
outgoing.Length > 0;
        }

        /**
         * Checks if
all outgoing
transitions only match
ASCII
         * characters.
         *
         * @return
true if all
```

```
transitions are ASCII-
only, or
         *
false otherwise
         */
        public bool
IsAsciiOutgoing() {
            for (int i
= 0; i <
outgoing.Length; i++)
{
                if
(!outgoing[i].IsAscii(
)) {

return false;
                }
            }
            return
true;
        }

        /**
         * Adds a new
incoming transition.
         *
         * @param
trans          the
transition to add
         */
        public void
AddIn(NFATransition
trans) {

Array.Resize(ref
incoming,
incoming.Length + 1);

incoming[incoming.Leng
th - 1] = trans;
        }

        /**
         * Adds a new
outgoing character
transition. If the
target
         * state
specified was null and
an identical
transition
         * already
exists, it will be
reused and its target
returned.
         *
         * @param ch
he character to match
         * @param
ignoreCase     the
case-insensitive flag
         * @param
state          the
target state, or null
         *
         * @return the
transition target
state
         */
        public
NFAState AddOut(char
ch, bool ignoreCase,
NFAState state) {
            if
(ignoreCase) {
                if
(state == null) {

state = new
NFAState();
                }

AddOut(new
NFACharTransition(Char
.ToLower(ch), state));

AddOut(new
NFACharTransition(Char
.ToUpper(ch), state));
                return
state;
            } else {
                if
(state == null) {

state =
FindUniqueCharTransiti
on(ch);
                    if
(state != null) {

return state;
                    }

state = new
NFAState();
                }
                return
AddOut(new
NFACharTransition(ch,
state));
            }
        }

        /**
         * Adds a new
outgoing transition.
         *
         * @param
trans          the
transition to add
         *
         * @return the
transition target
state
         */
        public
NFAState
AddOut(NFATransition
trans) {

Array.Resize(ref
outgoing,
outgoing.Length + 1);

outgoing[outgoing.Leng
th - 1] = trans;
            if (trans
is
NFAEpsilonTransition)
{

epsilonOut = true;
            }
            return
trans.state;
        }

        /**
         * Merges all
the transitions in
this state into
another
         * state.
         *
         * @param
state      the state
to merge into
         */
        public void
MergeInto(NFAState
state) {
            for (int i
= 0; i <
incoming.Length; i++)
{

state.AddIn(incoming[i
]);

incoming[i].state =
state;
            }
            incoming =
null;
            for (int i
= 0; i <
outgoing.Length; i++)
{

state.AddOut(outgoing[
i]);
            }
            outgoing =
null;
        }

        /**
         * Finds a
unique character
transition if one
exists. The
         * transition
must be the only
matching single
character
         * transition
and no other
transitions may reach
the same
         * state.
         *
         * @param ch
the character to
search for
         *
         * @return the
unique transition
state found, or
         *
null if not found
         */
        private
NFAState
FindUniqueCharTransiti
on(char ch) {

NFATransition  res =
null;

NFATransition  trans;

            for (int i
= 0; i <
outgoing.Length; i++)
{
                trans
= outgoing[i];
                if
(trans.Match(ch) &&
trans is
NFACharTransition) {
                    if
(res != null) {

return null;
                    }

res = trans;
                }
            }
            for (int i
= 0; res != null && i
< outgoing.Length;
i++) {
```

```
                trans
= outgoing[i];
                if
(trans != res &&
trans.state ==
res.state) {

return null;
                }
            }
            return
(res == null) ? null :
res.state;
        }

        /**
         * Attempts a
match on each of the
transitions leading
from
         * this state.
If a match is found,
its state will be
added
         * to the
queue. If the initial
match flag is set,
epsilon
         * transitions
will also be matched
(and their targets
called
         *
recursively).
         *
         * @param ch
the character to match
         * @param
queue      the state
queue
         * @param
initial    the initial
match flag
         */
        public void
MatchTransitions(char
ch, NFAStateQueue
queue, bool initial) {

NFATransition  trans;
            NFAState
target;

            for (int i
= 0; i <
outgoing.Length; i++)
{
                trans
= outgoing[i];
                target
= trans.state;
```

```
                if
(initial && trans is
NFAEpsilonTransition)
{

target.MatchTransition
s(ch, queue, true);
                } else
if (trans.Match(ch)) {

queue.AddLast(target);
                    if
(target.epsilonOut) {

target.MatchEmpty(queu
e);
                    }
                }
            }
        }

        /**
         * Adds all
the epsilon transition
targets to the
specified
         * queue.
         *
         * @param
queue      the state
queue
         */
        public void
MatchEmpty(NFAStateQue
ue queue) {

NFATransition  trans;
            NFAState
target;

            for (int i
= 0; i <
outgoing.Length; i++)
{
                trans
= outgoing[i];
                if
(trans is
NFAEpsilonTransition)
{

target = trans.state;

queue.AddLast(target);
                    if
(target.epsilonOut) {

target.MatchEmpty(queu
e);
                    }
                }
            }
```

```
        }
    }


    /**
     * An NFA state
transition. A
transition checks a
single
     * character of
input an determines if
it is a match. If a
match
     * is encountered,
the NFA should move
forward to the
transition
     * state.
     */
    internal abstract
class NFATransition {

        /**
         * The target
state of the
transition.
         */
        internal
NFAState state;

        /**
         * Creates a
new state transition.
         *
         * @param
state          the
target state
         */
        public
NFATransition(NFAState
state) {
            this.state
= state;

this.state.AddIn(this)
;
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
```

```
         *
false otherwise
         */
        public
abstract bool
IsAscii();

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
abstract bool
Match(char ch);

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
abstract NFATransition
Copy(NFAState state);
    }


    /**
     * The special
epsilon transition.
This transition
matches the
     * empty input,
i.e. it is an
automatic transition
that doesn't
     * read any input.
As such, it returns
false in the match
method
     * and is handled
specially everywhere.
```

```
     */
    internal class
NFAEpsilonTransition :
NFATransition {

        /**
         * Creates a
new epsilon
transition.
         *
         * @param
state          the
target state
         */
        public
NFAEpsilonTransition(N
FAState state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
false;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            return
false;
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFAEpsilonTransition(s
tate);
        }
    }


    /**
     * A single
character match
transition.
     */
    internal class
NFACharTransition :
NFATransition {

        /**
         * The
character to match.
         */
        protected char
match;

        /**
         * Creates a
new character
transition.
         *
         * @param
match          the
character to match
         * @param
state          the
target state
         */
        public
NFACharTransition(char
match, NFAState state)
: base(state) {
            this.match
= match;
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return 0
<= match && match <
128;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            return
this.match == ch;
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFACharTransition(matc
h, state);
        }
    }


    /**
     * A character
range match
transition. Used for
user-defined
     * character sets
in regular
expressions.
     */
    internal class
NFACharRangeTransition
: NFATransition {

        /**
         * The inverse
match flag.
         */
        protected bool
inverse;

        /**
         * The case-
insensitive match
flag.
         */
        protected bool
ignoreCase;

        /**
         * The
character set content.
This array may contain
either
         * range
objects or Character
objects.
         */
        private
object[] contents =
new object[0];

        /**
         * Creates a
new character range
transition.
         *
         * @param
inverse        the
inverse match flag
         * @param
ignoreCase     the
case-insensitive match
flag
         * @param
state          the
target state
         */
        public
NFACharRangeTransition
(bool inverse,

bool ignoreCase,

NFAState state) :
base(state) {

this.inverse =
inverse;

this.ignoreCase =
ignoreCase;
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            object
obj;
            char    c;

            if
(inverse) {
                return
false;
            }
            for (int i
= 0; i <
contents.Length; i++)
{
                obj =
contents[i];
                if
(obj is char) {
                    c
= (char) obj;
                    if
(c < 0 || 128 <= c) {

return false;
                    }
                } else
if (obj is Range) {
                    if
(!((Range)
obj).IsAscii()) {

return false;
                    }
                }
            }
            return
true;
        }

        /**
         * Adds a
single character to
this character set.
         *
         * @param c
the character to add
         */
        public void
AddCharacter(char c) {
            if
(ignoreCase) {
                c =
Char.ToLower(c);
            }

AddContent(c);
        }

        /**
         * Adds a
character range to
this character set.
         *
         * @param min
the minimum character
value
         * @param max
the maximum character
value
         */
        public void
AddRange(char min,
char max) {
            if
(ignoreCase) {
                min =
Char.ToLower(min);
                max =
Char.ToLower(max);
            }

AddContent(new
Range(min, max));
        }

        /**
         * Adds an
object to the
character set content
array.
         *
         * @param obj
the object to add
         */
        private void
AddContent(Object obj)
{

Array.Resize(ref
contents,
contents.Length + 1);

contents[contents.Leng
th - 1] = obj;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            object
obj;
            char    c;
            Range   r;

            if
(ignoreCase) {
                ch =
Char.ToLower(ch);
            }
            for (int i
= 0; i <
contents.Length; i++)
{
                obj =
contents[i];
                if
(obj is char) {
                    c
= (char) obj;
                    if
(c == ch) {

return !inverse;
                    }
                } else
if (obj is Range) {
                    r
= (Range) obj;
                    if
(r.Inside(ch)) {

return !inverse;
                    }
                }
            }
            return
inverse;
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {

NFACharRangeTransition
copy;

            copy = new
NFACharRangeTransition
(inverse, ignoreCase,
state);

copy.contents =
contents;
            return
copy;
        }

        /**
         * A character
range class.
```

```
        */
       private class
Range {

            /**
             * The
minimum character
value.
             */
            private
char min;

            /**
             * The
maximum character
value.
             */
            private
char max;

            /**
             * Creates
a new character range.
             *
             * @param
min        the minimum
character value
             * @param
max        the maximum
character value
             */
            public
Range(char min, char
max) {

this.min = min;

this.max = max;
            }

            /**
             * Checks
if this range only
matches ASCII
characters
             *
             * @return
true if this range
only matches ASCII, or
             *
false otherwise
             */
            public
bool IsAscii() {
                return
0 <= min && min < 128
&&

0 <= max && max < 128;
            }

            /**
             * Checks
if the specified
character is inside
the range.
             *
             * @param
c          the
character to check
             *
             * @return
true if the character
is in the range, or
             *
false otherwise
             */
            public
bool Inside(char c) {
                return
min <= c && c <= max;
            }
        }
    }



    /**
     * The dot ('.')
character set
transition. This
transition
     * matches a
single character that
is not equal to a
newline
     * character.
     */
    internal class
NFADotTransition :
NFATransition {

        /**
         * Creates a
new dot character set
transition.
         *
         * @param
state          the
target state
         */
        public
NFADotTransition(NFASt
ate state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
false;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            switch
(ch) {
            case '\n':
            case '\r':
            case
'\u0085':
            case
'\u2028':
            case
'\u2029':
                return
false;
            default:
                return
true;
            }
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFADotTransition(state
);
        }
    }



    /**
     * The digit
character set
transition. This
transition matches a
     * single numeric
character.
     */
    internal class
NFADigitTransition :
NFATransition {

        /**
         * Creates a
new digit character
set transition.
         *
         * @param
state          the
target state
         */
        public
NFADigitTransition(NFA
State state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
true;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            return ’0’
<= ch && ch <= ’9’;
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFADigitTransition(sta
te);
        }
    }


    /**
     * The non-digit
character set
transition. This
transition
     * matches a
single non-numeric
character.
     */
    internal class
NFANonDigitTransition
: NFATransition {

        /**
         * Creates a
new non-digit
character set
transition.
         *
         * @param
state          the
target state
         */
        public
NFANonDigitTransition(
NFAState state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
false;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            return ch
< ’0’ || ’9’ < ch;
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFANonDigitTransition(
state);
        }
    }


    /**
     * The whitespace
character set
transition. This
transition
     * matches a
single whitespace
character.
     */
    internal class
NFAWhitespaceTransitio
n : NFATransition {

        /**
         * Creates a
new whitespace
character set
transition.
         *
         * @param
state          the
target state
         */
        public
NFAWhitespaceTransitio
n(NFAState state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
true;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            switch
(ch) {
            case ’ ’:
            case ’\t’:
            case ’\n’:
            case ’\f’:
            case ’\r’:
            case
(char) 11:
                return
true;
            default:
                return
false;
            }
        }

        /**
         * Creates a
copy of this
```

```
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFAWhitespaceTransitio
n(state);
        }
    }


    /**
     * The non-
whitespace character
set transition. This
transition
     * matches a
single non-whitespace
character.
     */
    internal class
NFANonWhitespaceTransi
tion : NFATransition {

        /**
         * Creates a
new non-whitespace
character set
transition.
         *
         * @param
state          the
target state
         */
        public
NFANonWhitespaceTransi
tion(NFAState state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
false;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            switch
(ch) {
            case ' ':
            case '\t':
            case '\n':
            case '\f':
            case '\r':
            case
(char) 11:
                return
false;
            default:
                return
true;
            }
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFANonWhitespaceTransi
tion(state);
        }
    }


    /**
     * The word
character set
transition. This
transition matches a
     * single word
character.
     */
    internal class
NFAWordTransition :
NFATransition {

        /**
         * Creates a
new word character set
transition.
         *
         * @param
state          the
target state
         */
        public
NFAWordTransition(NFAS
tate state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
true;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            return
('a' <= ch && ch <=
'z')
                ||
('A' <= ch && ch <=
'Z')
                ||
('0' <= ch && ch <=
'9')
                || ch
== '_';
        }

        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFAWordTransition(stat
e);
        }
    }


    /**
```

```
     * The non-word
character set
transition. This
transition matches
     * a single non-
word character.
     */
    internal class
NFANonWordTransition :
NFATransition {

        /**
         * Creates a
new non-word character
set transition.
         *
         * @param
state          the
target state
         */
        public
NFANonWordTransition(N
FAState state) :
base(state) {
        }

        /**
         * Checks if
this transition only
matches ASCII
characters.
         * I.e.
characters with
numeric values between
0 and 127.
         *
         * @return
true if this
transition only
matches ASCII, or
         *
false otherwise
         */
        public
override bool
IsAscii() {
            return
false;
        }

        /**
         * Checks if
the specified
character matches the
transition.
         *
         * @param ch
the character to check
         *
         * @return
true if the character
matches, or
         *
false otherwise
         */
        public
override bool
Match(char ch) {
            bool word
= ('a' <= ch && ch <=
'z')

|| ('A' <= ch && ch <=
'Z')

|| ('0' <= ch && ch <=
'9')

|| ch == '_';
            return
!word;
        }


        /**
         * Creates a
copy of this
transition but with
another target
         * state.
         *
         * @param
state          the new
target state
         *
         * @return an
identical copy of this
transition
         */
        public
override NFATransition
Copy(NFAState state) {
            return new
NFANonWordTransition(s
tate);
        }
    }



    /**
     * An NFA state
queue. This queue is
used during processing
to
     * keep track of
the current and
subsequent NFA states.
The
     * current state
is read from the
beginning of the
queue, and new
     * states are
added at the end. A
marker index is used
to
     * separate the
current from the
subsequent states.<p>
     *
     * The queue
implementation is
optimized for quick
removal at the
     * beginning and
addition at the end.
It will attempt to use
a
     * fixed-size
array to store the
whole queue, and moves
the data
     * in this array
only when absolutely
needed. The array is
also
     * enlarged
automatically if too
many states are being
processed
     * at a single
time.
     */
    internal class
NFAStateQueue {

        /**
         * The state
queue array. Will be
enlarged as needed.
         */
        private
NFAState[] queue = new
NFAState[2048];

        /**
         * The position
of the first entry in
the queue (inclusive).
         */
        private int
first = 0;

        /**
         * The position
just after the last
entry in the queue
         * (exclusive).
         */
        private int
last = 0;

        /**
         * The current
queue mark position.
         */
        private int
mark = 0;

        /**
         * The empty
queue property (read-
only).
         */
        public bool
Empty {
            get {
                return
(last <= first);
            }
        }

        /**
         * The marked
first entry property
(read-only). This is
set
         * to true if
the first entry in the
queue has been marked.
         */
        public bool
Marked {
            get {
                return
first == mark;
            }
        }

        /**
         * Clears this
queue. This operation
is fast, as it just
         * resets the
queue position
indices.
         */
        public void
Clear() {
            first = 0;
            last = 0;
            mark = 0;
        }

        /**
         * Marks the
end of the queue. This
means that the next
entry
         * added to
the queue will be
marked (when it
becomes the
         * first in
the queue). This
operation is fast.
         */
    public void
MarkEnd() {
      mark =
last;
    }

    /**
     * Removes and
returns the first
entry in the queue.
This
     * operation
is fast, since it will
only update the index
of
     * the first
entry in the queue.
     *
     * @return the
previous first entry
in the queue
     */
    public
NFAState RemoveFirst()
{
      if (first
< last) {

first++;
        return
queue[first - 1];
      } else {
        return
null;
      }
    }

    /**
     * Adds a new
entry at the end of
the queue. This
operation
     * is mostly
fast, unless all the
allocated queue space
has
     * already
been used.
     *
     * @param
state          the
state to add
     */
    public void
AddLast(NFAState
state) {
      if (last
>= queue.Length) {
        if
(first <= 0) {

Array.Resize(ref
queue, queue.Length *
2);
        } else
{

Array.Copy(queue,
first, queue, 0, last
- first);

last -= first;

mark -= first;

first = 0;
        }
      }

queue[last++] = state;
    }
  }
}

/*
 * TokenPattern.cs
*/

using System;
using System.Text;

namespace Core.Library
{

  /**
   * A token
pattern. This class
contains the
definition of a token
   * (i.e. it's
pattern), and allows
testing a string
against this
   * pattern. A
token pattern is
uniquely identified by
an integer id,
   * that must be
provided upon
creation.
   *


   *
   */
  public class
TokenPattern {


    /**
     * The pattern
type enumeration.
     */
    public enum
PatternType {

      /**
       * The
string pattern type is
used for tokens that
only
       * match
an exact string.
       */
      STRING,

      /**
       * The
regular expression
pattern type is used
for tokens
       * that
match a regular
expression.
       */
      REGEXP
    }

    /**
     * The token
pattern identity.
     */
    private int
id;

    /**
     * The token
pattern name.
     */
    private string
name;

    /**
     * The token
pattern type.
     */
    private
PatternType type;

    /**
     * The token
pattern.
     */
    private string
pattern;

    /**
     * The token
error flag. If this
flag is set, it means
that an
     * error
should be reported if
the token is found.
The error
     * message is
present in the
errorMessage variable.
     *
     * @see
#errorMessage
     */
    private bool
error = false;

    /**
     * The token
error message. This
message will only be
set if the
     * token error
flag is set.
     *
     * @see #error
     */
    private string
errorMessage = null;

    /**
     * The token
ignore flag. If this
flag is set, it means
that the
     * token
should be ignored if
found. If an ignore
message is
     * present in
the ignoreMessage
variable, it will also
be reported
     * as a
warning.
     *
     * @see
#ignoreMessage
     */
    private bool
ignore = false;

    /**
     * The token
ignore message. If
this message is set
when the token
     * ignore flag
is also set, a warning
message will be
printed if
     * the token
is found.
     *
     * @see
#ignore
     */
    private string
ignoreMessage = null;

    /**
```

```
         * The
optional debug
information message.
This is normally set
         * when the
token pattern is
analyzed by the
tokenizer.
         */
        private string
debugInfo = null;

        /**
         * Creates a
new token pattern.
         *
         * @param id
the token pattern id
         * @param name
the token pattern name
         * @param type
the token pattern type
         * @param
pattern        the
token pattern
         */
        public
TokenPattern(int id,

string name,

PatternType type,

string pattern) {

            this.id =
id;
            this.name
= name;
            this.type
= type;

this.pattern =
pattern;
        }

        /**
         * The token
pattern identity
property (read-only).
This
         * property
contains the unique
token pattern identity
value.
         *
         *
         */
        public int Id
{
            get {
                return
id;
            }
        }

        /**
         * Returns the
unique token pattern
identity value.
         *
         * @return the
token pattern id
         *
         * @see #Id
         *
         * @deprecated
Use the Id property
instead.
         */
        public int
GetId() {
            return id;
        }

        /**
         * The token
pattern name property
(read-only).
         *
         *
         */
        public string
Name {
            get {
                return
name;
            }
        }

        /**
         * Returns the
token pattern name.
         *
         * @return the
token pattern name
         *
         * @see #Name
         *
         * @deprecated
Use the Name property
instead.
         */
        public string
GetName() {
            return
name;
        }

        /**
         * The token
pattern type property
(read-only).
         *
         *
         */
        public
PatternType Type {
            get {
                return
type;
            }
        }

        /**
         * Returns the
token pattern type.
         *
         * @return the
token pattern type
         *
         * @see #Type
         *
         * @deprecated
Use the Type property
instead.
         */
        public
PatternType
GetPatternType() {
            return
type;
        }

        /**
         * The token
pattern property
(read-only). This
property
         * contains
the actual pattern
(string or regexp)
which have
         * to be
matched.
         *
         *
         */
        public string
Pattern {
            get {
                return
pattern;
            }
        }

        /**
         * Returns te
token pattern.
         *
         * @return the
token pattern
         *
         * @see
#Pattern
         *
         * @deprecated
Use the Pattern
property instead.
         */
        public string
GetPattern() {
            return
pattern;
        }

        /**
         * The error
flag property. If this
property is true, the
         * token
pattern corresponds to
an error token and an
error
         * should be
reported if a match is
found. When setting
this
         * property to
true, a default error
message is created if
         * none was
previously set.
         *
         *
         */
        public bool
Error {
            get {
                return
error;
            }
            set {
                error
= value;
                if
(error && errorMessage
== null) {

errorMessage =
"unrecognized token
found";
                }
            }
        }

        /**
         * The token
error message
property. The error
message is
         * printed
whenever the token is
matched. Setting the
error
```

```
         * message
property also sets the
error flag to true.
         *
         * @see #Error
         *
         *
         */
        public string
ErrorMessage {
            get {
                return
errorMessage;
            }
            set {
                error
= true;

errorMessage = value;
            }
        }

        /**
         * Checks if
the pattern
corresponds to an
error token. If this
         * is true, it
means that an error
should be reported if
a
         * matching
token is found.
         *
         * @return
true if the pattern
maps to an error
token, or
         *
false otherwise
         *
         * @see #Error
         *
         * @deprecated
Use the Error property
instead.
         */
        public bool
IsError() {
            return
Error;
        }

        /**
         * Returns the
token error message if
the pattern
corresponds to
         * an error
token.
         *
```

```
         * @return the
token error message
         *
         * @see
#ErrorMessage
         *
         * @deprecated
Use the ErrorMessage
property instead.
         */
        public string
GetErrorMessage() {
            return
ErrorMessage;
        }

        /**
         * Sets the
token error flag and
assigns a default
error message.
         *
         * @see #Error
         *
         * @deprecated
Use the Error property
instead.
         */
        public void
SetError() {
            Error =
true;
        }

        /**
         * Sets the
token error flag and
assigns the specified
error
         * message.
         *
         * @param
message        the
error message to
display
         *
         * @see
#ErrorMessage
         *
         * @deprecated
Use the ErrorMessage
property instead.
         */
        public void
SetError(string
message) {

ErrorMessage =
message;
        }

        /**
```

```
         * The ignore
flag property. If this
property is true, the
         * token
pattern corresponds to
an ignore token and
should be
         * skipped if
a match is found.
         *
         *
         */
        public bool
Ignore {
            get {
                return
ignore;
            }
            set {
                ignore
= value;
            }
        }

        /**
         * The token
ignore message
property. The ignore
message is
         * printed
whenever the token is
matched. Setting the
ignore
         * message
property also sets the
ignore flag to true.
         *
         * @see
#Ignore
         *
         *
         */
        public string
IgnoreMessage {
            get {
                return
ignoreMessage;
            }
            set {
                ignore
= true;

ignoreMessage = value;
            }
        }

        /**
         * Checks if
the pattern
corresponds to an
ignored token. If this
```

```
         * is true, it
means that the token
should be ignored if
found.
         *
         * @return
true if the pattern
maps to an ignored
token, or
         *
false otherwise
         *
         * @see
#Ignore
         *
         * @deprecated
Use the Ignore
property instead.
         */
        public bool
IsIgnore() {
            return
Ignore;
        }

        /**
         * Returns the
token ignore message
if the pattern
corresponds to
         * an ignored
token.
         *
         * @return the
token ignore message
         *
         * @see
#IgnoreMessage
         *
         * @deprecated
Use the IgnoreMessage
property instead.
         */
        public string
GetIgnoreMessage() {
            return
IgnoreMessage;
        }

        /**
         * Sets the
token ignore flag and
clears the ignore
message.
         *
         * @see
#Ignore
         *
         * @deprecated
Use the Ignore
property instead.
         */
        public void
SetIgnore() {
            Ignore =
true;
        }

        /**
         * Sets the
token ignore flag and
assigns the specified
ignore
         * message.
         *
         * @param
message        the
ignore message to
display
         *
         * @see
#IgnoreMessage
         *
         * @deprecated
Use the IgnoreMessage
property instead.
         */
        public void
SetIgnore(string
message) {

IgnoreMessage =
message;
        }

        /**
         * The token
debug info message
property. This is
normally be
         * set when
the token pattern is
analyzed by the
tokenizer.
         *
         *
         */
        public string
DebugInfo {
            get {
                return
debugInfo;
            }
            set {

debugInfo = value;
            }
        }

        /**
         * Returns a
string representation
of this object.
         *
```

```
         * @return a
token pattern string
representation
         */
        public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();


buffer.Append(name);

buffer.Append(" (");

buffer.Append(id);

buffer.Append("): ");
            switch
(type) {
            case
PatternType.STRING:

buffer.Append("\"");

buffer.Append(pattern)
;

buffer.Append("\"");
                break;
            case
PatternType.REGEXP:

buffer.Append("<<");

buffer.Append(pattern)
;

buffer.Append(">>");
                break;
            }
            if (error)
{

buffer.Append(" ERROR:
\"");

buffer.Append(errorMes
sage);

buffer.Append("\"");
            }
            if
(ignore) {

buffer.Append("
IGNORE");
                if
(ignoreMessage !=
null) {
```

```

buffer.Append(": \"");

buffer.Append(ignoreMe
ssage);

buffer.Append("\"");
                }
            }
            if
(debugInfo != null) {

buffer.Append("\n  ");

buffer.Append(debugInf
o);
            }
            return
buffer.ToString();
        }

        /**
         * Returns a
short string
representation of this
object.
         *
         * @return a
short string
representation of this
object
         */
        public string
ToShortString() {

StringBuilder  buffer
= new StringBuilder();
            int
newline =
pattern.IndexOf('\n');

            if (type
== PatternType.STRING)
{

buffer.Append("\"");
                if
(newline >= 0) {
                    if
(newline > 0 &&
pattern[newline - 1]
== '\r') {

newline--;
                    }

buffer.Append(pattern.
Substring(0,
newline));

buffer.Append("(...)")
;
```

```
                } else
{

buffer.Append(pattern)
;
                }

buffer.Append("\"");
            } else {

buffer.Append("<");

buffer.Append(name);

buffer.Append(">");
            }

            return
buffer.ToString();
        }
    }
}

/*
 *
TokenRegExpParser.cs
 */

using System;
using
System.Collections;
using
System.Globalization;
using System.Text;
using Core.Library.RE;

namespace Core.Library
{

    /**
     * A regular
expression parser. The
parser creates an NFA
for the
     * regular
expression having a
single start and
acceptance states.
     *


     *
     *
     */
    internal class
TokenRegExpParser {

        /**
         * The regular
expression pattern.
         */
        private string
pattern;
```

```
        /**
         * The
character case ignore
flag.
         */
        private bool
ignoreCase;

        /**
         * The current
position in the
pattern. This variable
is used by
         * the parsing
methods.
         */
        private int
pos;

        /**
         * The start
NFA state for this
regular expression.
         */
        internal
NFAState start = new
NFAState();

        /**
         * The end NFA
state for this regular
expression.
         */
        internal
NFAState end = null;

        /**
         * The number
of states found.
         */
        private int
stateCount = 0;

        /**
         * The number
of transitions found.
         */
        private int
transitionCount = 0;

        /**
         * The number
of epsilon transitions
found.
         */
        private int
epsilonCount = 0;

        /**
          * Creates a
new case-sensitive
regular expression
parser. Note
         * that this
will trigger the
parsing of the regular
expression.
         *
         * @param
pattern        the
regular expression
pattern
         *
         * @throws
RegExpException if the
regular expression
couldn't be
         *
parsed correctly
         */
        public
TokenRegExpParser(stri
ng pattern) :
this(pattern, false) {
        }


        /**
         * Creates a
new regular expression
parser. The regular
         * expression
can be either case-
sensitive or case-
insensitive.
         * Note that
this will trigger the
parsing of the regular
         * expression.
         *
         * @param
pattern        the
regular expression
pattern
         * @param
ignoreCase     the
character case ignore
flag
         *
         * @throws
RegExpException if the
regular expression
couldn't be
         *
parsed correctly
         */
        public
TokenRegExpParser(stri
ng pattern, bool
ignoreCase) {

this.pattern =
pattern;

this.ignoreCase =
ignoreCase;
            this.pos =
0;
            this.end =
ParseExpr(start);
            if (pos <
pattern.Length) {
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNEXPECTED_CHARACT
ER,

pos,

pattern);
            }
        }

        /**
         * Returns the
debug information for
the generated NFA.
         *
         * @return the
debug information for
the generated NFA
         */
        public string
GetDebugInfo() {
            if
(stateCount == 0) {

UpdateStats(start, new
Hashtable());
            }
            return
stateCount + " states,
" +

transitionCount + "
transitions, " +

epsilonCount + "
epsilons";
        }

        /**
         * Updates the
statistical counters
for the NFA generated.
         *
         * @param
state          the
current state to visit
         * @param
visited        the
lookup map of visited
states
         */
        private void
UpdateStats(NFAState
state, Hashtable
visited) {
            if
(!visited.ContainsKey(
state)) {

visited.Add(state,
state);

stateCount++;
                for
(int i = 0; i <
state.outgoing.Length;
i++) {

transitionCount++;
                    if
(state.outgoing[i] is
NFAEpsilonTransition)
{

epsilonCount++;
                    }

UpdateStats(state.outg
oing[i].state,
visited);
                }
            }
        }

        /**
         * Parses a
regular expression.
This method handles
the Expr
         * production
in the grammar (see
regexp.grammar).
         *
         * @param
start          the
initial NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseExpr(NFAState
start) {
```

```
            NFAState
end = new NFAState();
            NFAState
subStart;
            NFAState
subEnd;

            do {
                if
(PeekChar(0) == '|') {

ReadChar('|');
                }

subStart = new
NFAState();
                subEnd
= ParseTerm(subStart);
                if
(subStart.incoming.Len
gth == 0) {

subStart.MergeInto(sta
rt);
                } else
{

start.AddOut(new
NFAEpsilonTransition(s
ubStart));
                }
                if
(subEnd.outgoing.Lengt
h == 0 ||

(!end.HasTransitions()
&& PeekChar(0) !=
'|')) {

subEnd.MergeInto(end);
                } else
{

subEnd.AddOut(new
NFAEpsilonTransition(e
nd));
                }
            } while
(PeekChar(0) == '|');
            return
end;
        }

        /**
         * Parses a
regular expression
term. This method
handles the
         * Term
production in the
grammar (see
regexp.grammar).
```

```
         *
         * @param
start          the
initial NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseTerm(NFAState
start) {
            NFAState
end;

            end =
ParseFact(start);
            while
(true) {
                switch
(PeekChar(0)) {
                case -
1:
                case
')':
                case
']':
                case
'{':
                case
'}':
                case
'?':
                case
'+':
                case
'|':

return end;

default:

end = ParseFact(end);

break;
                }
            }
        }

        /**
         * Parses a
regular expression
factor. This method
handles the
```

```
         * Fact
production in the
grammar (see
regexp.grammar).
         *
         * @param
start          the
initial NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseFact(NFAState
start) {
            NFAState
placeholder = new
NFAState();
            NFAState
end;

            end =
ParseAtom(placeholder)
;
            switch
(PeekChar(0)) {
            case '?':
            case '*':
            case '+':
            case '{':
                end =
ParseAtomModifier(plac
eholder, end);
                break;
            }
            if
(placeholder.incoming.
Length > 0 &&
start.outgoing.Length
> 0) {

start.AddOut(new
NFAEpsilonTransition(p
laceholder));
                return
end;
            } else {

placeholder.MergeInto(
start);
                return
(end == placeholder) ?
start : end;
            }
```

```
        }

        /**
         * Parses a
regular expression
atom. This method
handles the
         * Atom
production in the
grammar (see
regexp.grammar).
         *
         * @param
start          the
initial NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseAtom(NFAState
start) {
            NFAState
end;

            switch
(PeekChar(0)) {
            case '.':

ReadChar('.');
                return
start.AddOut(new
NFADotTransition(new
NFAState()));
            case '(':

ReadChar('(');
                end =
ParseExpr(start);

ReadChar(')');
                return
end;
            case '[':

ReadChar('[');
                end =
ParseCharSet(start);

ReadChar(']');
                return
end;
            case -1:
            case ')':
```

```
            case ']':
            case '{':
            case '}':
            case '?':
            case '*':
            case '+':
            case '|':
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNEXPECTED_CHARACT
ER,

pos,

pattern);
            default:
                return
ParseChar(start);
            }
        }

        /**
         * Parses a
regular expression
atom modifier. This
method handles
         * the
AtomModifier
production in the
grammar (see
regexp.grammar).
         *
         * @param
start          the
initial NFA state
         * @param end
the terminal NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseAtomModifier(NFAS
tate start, NFAState
end) {
            int  min =
0;
            int  max =
-1;
            int
firstPos = pos;
            // Read
min and max
            switch
(ReadChar()) {
            case '?':
                min =
0;
                max =
1;
                break;
            case '*':
                min =
0;
                max =
-1;
                break;
            case '+':
                min =
1;
                max =
-1;
                break;
            case '{':
                min =
ReadNumber();
                max =
min;
                if
(PeekChar(0) == ',') {

ReadChar(',');

max = -1;
                    if
(PeekChar(0) != '}') {

max = ReadNumber();
                    }
                }

ReadChar('}');
                if
(max == 0 || (max > 0
&& min > max)) {

throw new
RegExpException(

RegExpException.ErrorT
ype.INVALID_REPEAT_COU
NT,

firstPos,

pattern);
                }
                break;
            default:
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNEXPECTED_CHARACT
ER,

pos - 1,

pattern);
            }

            // Read
possessive or
reluctant modifiers
            if
(PeekChar(0) == '?') {
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNSUPPORTED_SPECIA
L_CHARACTER,

pos,

pattern);
            } else if
(PeekChar(0) == '+') {
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNSUPPORTED_SPECIA
L_CHARACTER,

pos,

pattern);
            }

            // Handle
supported repeaters
            if (min ==
0 && max == 1) {
                return
start.AddOut(new
NFAEpsilonTransition(e
nd));
            } else if
(min == 0 && max == -
1) {
                if
(end.outgoing.Length
== 0) {

end.MergeInto(start);
                } else
{

end.AddOut(new
NFAEpsilonTransition(s
tart));
                }
                return
start;
            } else if
(min == 1 && max == -
1) {
                if
(start.outgoing.Length
== 1 &&

end.outgoing.Length ==
0 &&

end.incoming.Length ==
1 &&

start.outgoing[0] ==
end.incoming[0]) {


end.AddOut(start.outgo
ing[0].Copy(end));
                } else
{

end.AddOut(new
NFAEpsilonTransition(s
tart));
                }
                return
end;
            } else {
                throw
new RegExpException(

RegExpException.ErrorT
ype.INVALID_REPEAT_COU
NT,

firstPos,

pattern);
            }
        }

        /**
         * Parses a
regular expression
character set. This
method handles
         * the
contents of the
'[...]' construct in a
regular expression.
         *
         * @param
start          the
initial NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
```

```
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseCharSet(NFAState
start) {
            NFAState
end = new NFAState();

NFACharRangeTransition
range;
            char
min;
            char
max;

            if
(PeekChar(0) == '^') {

ReadChar('^');
                range
= new
NFACharRangeTransition
(true, ignoreCase,
end);
            } else {
                range
= new
NFACharRangeTransition
(false, ignoreCase,
end);
            }

start.AddOut(range);
            while
(PeekChar(0) > 0) {
                min =
(char) PeekChar(0);
                switch
(min) {
                case
']':

return end;
                case
'\\':

range.AddCharacter(Rea
dEscapeChar());

break;

default:

ReadChar(min);
                        if
(PeekChar(0) == '-' &&

PeekChar(1) > 0 &&

PeekChar(1) != ']') {


ReadChar('-');

max = ReadChar();

range.AddRange(min,
max);
                        }
else {

range.AddCharacter(min
);
                        }

break;
                    }
                }
                return
end;
        }


        /**
         * Parses a
regular expression
character. This method
handles
         * a single
normal character in a
regular expression.
         *
         * @param
start          the
initial NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseChar(NFAState
start) {
            switch
(PeekChar(0)) {
            case '\\':
                return
ParseEscapeChar(start)
;
            case '^':
            case '$':
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNSUPPORTED_SPECIA
L_CHARACTER,

pos,

pattern);
            default:
                return
start.AddOut(ReadChar(
), ignoreCase, new
NFAState());
            }
        }

        /**
         * Parses a
regular expression
character escape. This
method
         * handles a
single character
escape in a regular
expression.
         *
         * @param
start          the
initial NFA state
         *
         * @return the
terminating NFA state
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private
NFAState
ParseEscapeChar(NFASta
te start) {
            NFAState
end = new NFAState();

            if
(PeekChar(0) == '\\'
&& PeekChar(1) > 0) {
                switch
((char) PeekChar(1)) {
                case
'd':

ReadChar();

ReadChar();

return
start.AddOut(new
NFADigitTransition(end
));
                case
'D':

ReadChar();

ReadChar();

return
start.AddOut(new
NFANonDigitTransition(
end));
                case
's':

ReadChar();

ReadChar();

return
start.AddOut(new
NFAWhitespaceTransitio
n(end));
                case
'S':

ReadChar();

ReadChar();

return
start.AddOut(new
NFANonWhitespaceTransi
tion(end));
                case
'w':

ReadChar();

ReadChar();

return
start.AddOut(new
NFAWordTransition(end)
);
                case
'W':

ReadChar();

ReadChar();

return
start.AddOut(new
NFANonWordTransition(e
nd));
                }
            }
            return
start.AddOut(ReadEscap
eChar(), ignoreCase,
end);
        }

        /**
         * Reads a
regular expression
character escape. This
method
         * handles a
single character
escape in a regular
expression.
         *
         * @return the
character read
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private char
ReadEscapeChar() {
            char    c;
            string
str;
            int
value;


ReadChar('\\');
            c =
ReadChar();
            switch (c)
{
            case '0':
                c =
ReadChar();
                if (c
< '0' || c > '3') {

throw new
RegExpException(

RegExpException.ErrorT
ype.UNSUPPORTED_ESCAPE
_CHARACTER,

pos - 3,

pattern);
                }
                value
= c - '0';
                c =
(char) PeekChar(0);
                if
('0' <= c && c <= '7')
{

value *= 8;

value += ReadChar() -
'0';
                    c
= (char) PeekChar(0);
                    if
('0' <= c && c <= '7')
{

value *= 8;

value += ReadChar() -
'0';
                    }
                }
                return
(char) value;
            case 'x':
                str =
ReadChar().ToString()
+
ReadChar().ToString();
                try {

value =
Int32.Parse(str,
NumberStyles.AllowHexS
pecifier);

return (char) value;
                }
catch
(FormatException) {

throw new
RegExpException(

RegExpException.ErrorT
ype.UNSUPPORTED_ESCAPE
_CHARACTER,

pos - str.Length - 2,

pattern);
                }
            case 'u':
                str =
ReadChar().ToString()
+

ReadChar().ToString()
+

ReadChar().ToString()
+

ReadChar().ToString();
                try {

value =
Int32.Parse(str,
NumberStyles.AllowHexS
pecifier);

return (char) value;
                }
catch
(FormatException) {

throw new
RegExpException(

RegExpException.ErrorT
ype.UNSUPPORTED_ESCAPE
_CHARACTER,

pos - str.Length - 2,

pattern);
                }
            case 't':
                return
'\t';
            case 'n':
                return
'\n';
            case 'r':
                return
'\r';
            case 'f':
                return
'\f';
            case 'a':
                return
'\u0007';
            case 'e':
                return
'\u001B';
            default:
                if
(('A' <= c && c <=
'Z') || ('a' <= c && c
<= 'z')) {

throw new
RegExpException(

RegExpException.ErrorT
ype.UNSUPPORTED_ESCAPE
_CHARACTER,

pos - 2,

pattern);
                }
                return
c;
            }
        }

        /**
         * Reads a
number from the
pattern. If the next
character isn't a
         * numeric
character, an
exception is thrown.
This method reads
         * several
consecutive numeric
characters.
         *
         * @return the
numeric value read
         *
         * @throws
RegExpException if an
error was encountered
in the
         *
pattern string
         */
        private int
ReadNumber() {

StringBuilder  buf =
new StringBuilder();
            int
c;


            c =
PeekChar(0);
            while ('0'
<= c && c <= '9') {

buf.Append(ReadChar())
;
                c =
PeekChar(0);
            }
            if
(buf.Length <= 0) {
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNEXPECTED_CHARACT
ER,

pos,

pattern);
            }
            return
Int32.Parse(buf.ToStri
ng());
        }

        /**
         * Reads the
next character in the
```

```
pattern. If no next
character
         * exists, an
exception is thrown.
         *
         * @return the
character read
         *
         * @throws
RegExpException if no
next character was
available in
         *
the pattern string
         */
        private char
ReadChar() {
            int  c =
PeekChar(0);

            if (c < 0)
{
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNTERMINATED_PATTE
RN,

pos,

pattern);
            } else {
                pos++;
                return
(char) c;
            }
        }

        /**
         * Reads the
next character in the
pattern. If the
character
         * wasn't the
specified one, an
exception is thrown.
         *
         * @param c
the character to read
         *
         * @return the
character read
         *
         * @throws
RegExpException if the
character read didn't
match the
         *
specified one, or if
no next character was
         *
available in the
pattern string
         */
        private char
ReadChar(char c) {
            if (c !=
ReadChar()) {
                throw
new RegExpException(

RegExpException.ErrorT
ype.UNEXPECTED_CHARACT
ER,

pos - 1,

pattern);
            }
            return c;
        }

        /**
         * Returns a
character that has not
yet been read from the
         * pattern. If
the requested position
is beyond the end of
the
         * pattern
string, -1 is
returned.
         *
         * @param
count          the
preview position, from
zero (0)
         *
         * @return the
character found, or
         *        -1
if beyond the end of
the pattern string
         */
        private int
PeekChar(int count) {
            if (pos +
count <
pattern.Length) {
                return
pattern[pos + count];
            } else {
                return
-1;
            }
        }
    }
}

/*
 * TokenStringDFA.cs
 */

using System;
using System.Text;

namespace Core.Library
{

    /**
     * A deterministic
finite state automaton
for matching exact
strings.
     * It uses a
sorted binary tree
representation of the
state
     * transitions in
order to enable quick
matches with a minimal
memory
     * footprint. It
only supports a single
character transition
between
     * states, but may
be run in an all case-
insensitive mode.
     *

     *
     *
     */
    internal class
TokenStringDFA {

        /**
         * The lookup
table for root states,
indexed by the first
ASCII
         * character.
This array is used to
for speed optimizing
the
         * first step
in the match.
         */
        private
DFAState[] ascii = new
DFAState[128];

        /**
         * The
automaton state
transition tree for
non-ASCII characters.
         * Each
transition from one
state to another is
added to the tree
         * with the
corresponding
character.
         */
        private
DFAState nonAscii =
new DFAState();

        /**
         * Creates a
new empty string
automaton.
         */
        public
TokenStringDFA() {
        }

        /**
         * Adds a
string match to this
automaton. New states
and
         * transitions
will be added to
extend this automaton
to
         * support the
specified string.
         *
         * @param str
the string to match
         * @param
caseInsensitive  the
case-insensitive flag
         * @param
value            the
match value
         */
        public void
AddMatch(string str,
bool caseInsensitive,
TokenPattern value) {
            DFAState
state;
            DFAState
next;
            char
c = str[0];
            int
start = 0;

            if
(caseInsensitive) {
                c =
Char.ToLower(c);
            }
            if (c <
128) {
                state
= ascii[c];
                if
(state == null) {
```

```
state = ascii[c] = new
DFAState();
        }

start++;
      } else {
        state
= nonAscii;
      }
      for (int i
= start; i <
str.Length; i++) {
        next =
state.tree.Find(str[i]
, caseInsensitive);
        if
(next == null) {

next = new DFAState();

state.tree.Add(str[i],
caseInsensitive,
next);
        }
        state
= next;
      }

state.value = value;
    }

    /**
     * Checks if
the automaton matches
an input stream. The
     * matching
will be performed from
a specified position.
This
     * method will
not read any
characters from the
stream, just
     * peek ahead.
The comparison can be
done either in
     * case-
sensitive or case-
insensitive mode.
     *
     * @param
input           the
input stream to check
     * @param pos
the starting position
     * @param
caseInsensitive  the
case-insensitive flag
     *
     * @return the
match value, or
     *
null if no match was
found
     *
     * @throws
IOException if an I/O
error occurred
     */
    public
TokenPattern
Match(ReaderBuffer
buffer, bool
caseInsensitive) {

TokenPattern  result =
null;
      DFAState
state;
      int
pos = 0;
      int
c;

      c =
buffer.Peek(0);
      if (c < 0)
{
        return
null;
      }
      if
(caseInsensitive) {
        c =
Char.ToLower((char)
c);
      }
      if (c <
128) {
        state
= ascii[c];
        if
(state == null) {

return null;
        } else
if (state.value !=
null) {

result = state.value;
        }
        pos++;
      } else {
        state
= nonAscii;
      }
      while ((c
= buffer.Peek(pos)) >=
0) {
        state
=
state.tree.Find((char)
c, caseInsensitive);
        if
(state == null) {

break;
        } else
if (state.value !=
null) {

result = state.value;
        }
        pos++;
      }
      return
result;
    }

    /**
     * Returns a
detailed string
representation of this
automaton.
     *
     * @return a
detailed string
representation of this
automaton
     */
    public
override string
ToString() {

StringBuilder  buffer
= new StringBuilder();

      for (int i
= 0; i < ascii.Length;
i++) {
        if
(ascii[i] != null) {

buffer.Append((char)
i);
          if
(ascii[i].value !=
null) {

buffer.Append(": ");

buffer.Append(ascii[i]
.value);

buffer.Append("\n");
          }

ascii[i].tree.PrintTo(
buffer, " ");
        }
      }

nonAscii.tree.PrintTo(
buffer, "");
      return
buffer.ToString();
    }
  }


  /**
   * An automaton
state. This class
represents a state in
the DFA
   * graph.
   *

   *
   *
   */
  internal class
DFAState {

    /**
     * The token
pattern matched at
this state.
     */
    internal
TokenPattern value =
null;

    /**
     * The
automaton state
transition tree. Each
transition from one
     * state to
another is added to
the tree with the
corresponding
     * character.
     */
    internal
TransitionTree tree =
new TransitionTree();
  }


  /**
   * An automaton
state transition tree.
This class contains a
   * binary search
tree for the automaton
transitions from one
   * state to
another. All
transitions are linked
to a single
   * character.
   *

   *
   *
```

```
     */
    internal class
TransitionTree {

        /**
         * The
transition character.
If this value is set
to the zero
         * character
('\0'), this tree is
empty.
         */
        private char
value = '\0';

        /**
         * The
transition target
state.
         */
        private
DFAState state = null;

        /**
         * The left
subtree.
         */
        private
TransitionTree left =
null;

        /**
         * The right
subtree.
         */
        private
TransitionTree right =
null;

        /**
         * Creates a
new empty automaton
transition tree.
         */
        public
TransitionTree() {
        }

        /**
         * Finds an
automaton state from
the specified
transition
         * character.
This method searches
this transition tree
for a
         * matching
transition. The
comparison can
optionally be done
         * with a
lower-case conversion
of the character.
         *
         * @param c
the character to
search for
         * @param
lowerCase      the
lower-case conversion
flag
         *
         * @return the
automaton state found,
or
         *
null if no transition
exists
         */
        public
DFAState Find(char c,
bool lowerCase) {
            if
(lowerCase) {
                c =
Char.ToLower(c);
            }
            if (value
== '\0' || value == c)
{
                return
state;
            } else if
(value > c) {
                return
left.Find(c, false);
            } else {
                return
right.Find(c, false);
            }
        }

        /**
         * Adds a
transition to this
tree. If the lower-
case flag is
         * set, the
character will be
converted to lower-
case before
         * being
added.
         *
         * @param c
the character to
transition for
         * @param
lowerCase      the
lower-case conversion
flag
         * @param
state          the
state to transition to
         */
        public void
Add(char c, bool
lowerCase, DFAState
state) {
            if
(lowerCase) {
                c =
Char.ToLower(c);
            }
            if (value
== '\0') {

this.value = c;

this.state = state;

this.left = new
TransitionTree();

this.right = new
TransitionTree();
            } else if
(value > c) {

left.Add(c, false,
state);
            } else {

right.Add(c, false,
state);
            }
        }

        /**
         * Prints the
automaton tree to the
specified string
buffer.
         *
         * @param
buffer         the
string buffer
         * @param
indent         the
current indentation
         */
        public void
PrintTo(StringBuilder
buffer, String indent)
{
            if
(this.left != null) {

this.left.PrintTo(buff
er, indent);
            }
            if
(this.value != '\0') {
                if
(buffer.Length > 0 &&
buffer[buffer.Length -
1] == '\n') {

buffer.Append(indent);
                }

buffer.Append(this.val
ue);
                if
(this.state.value !=
null) {

buffer.Append(": ");

buffer.Append(this.sta
te.value);

buffer.Append("\n");
                }

this.state.tree.PrintT
o(buffer, indent + "
");
            }
            if
(this.right != null) {

this.right.PrintTo(buf
fer, indent);
            }
        }
    }
}
```

TokenLibray

```
namespace TokenLibrary
{
    public class
ErrorClass
    {
        int lines;
        int column;
        string type;
        string
ErrorMessage;

        public void
setErrorMessage(string
ErrorMessage)
        {

this.ErrorMessage =
ErrorMessage;
        }
        public string
getErrorMessage()
        {
            return
this.ErrorMessage;
```

```
        }
        public void
setLines(int line)
        {
            this.lines
= line;
        }
        public int
getLines()
        {
            return
this.lines;
        }
        public void
setColumn(int column)
        {

this.column = column;
        }
        public int
getColumn()
        {
            return
this.column;
        }

        public void
setType(string type)
        {
            this.type
= type;
        }
        public string
getType()
        {
            return
this.type;
        }
    }
}

namespace TokenLibrary
{
    public abstract
class TokensClass
    {
        int lines;
        string tokens;
        string
lexemes;
        string
attributes;

        public void
setTokens(string
token)
        {

this.tokens = token;
        }
        public string
getTokens()
        {
            return
this.tokens;
        }
        public void
setLexemes(string
lexeme)
        {

this.lexemes = lexeme;
        }
        public string
getLexemes()
        {
            return
this.lexemes;
        }
        public void
setLines(int line)
        {
            this.lines
= line;
        }
        public int
getLines()
        {
            return
this.lines;
        }
        public void
setAttributes(string
attribute)
        {

this.attributes =
attribute;
        }
        public string
getAttributes()
        {
            return
this.attributes;
        }
    }
}
```