FAJF LANGUAGE AND COMPILER

PROJECT DOCUMENTATION

Submitted by:
Amna Jamal

TABLE OF CONTENTS

INTRODUCTION

The Project:

This project is basically a compiler for a language called FAJF, which we created specifically for this project.

Why a Compiler:

The project was chosen to fit its corresponding subject it i.e. Programming Languages and Environment well. Since PLE is aimed at teaching us how to analyse the economy of programming languages and what environments they work best in, making such an environment for our own language seemed to be very apt indeed.

Why C#:

The programming language we choose for this project was C#. Firstly it was the language we had been studying in the course. Secondly it is a very extensive language with a multitude of functions that made it very suitable for our project.
It is also a very user-friendly language and has good debugging facilities along with enhanced readability and writablity.

FAJF itself:

The purpose of FAJF is to give to young engineers such as ourselves, a language where they can implement electric circuits and analyse them with ease. This serves the dual purpose of not only helping students increase understanding of electric circuits but also better their logic and programming skills.
Currently the language is very simple since we did not have enough time to implement the many functionalities we envisioned. These are further discussed in the improvements section.

Project Overview:

A language compiler does three basic things:

- Lexical Analysis
- Parsing (Syntax Analysis)
- Code Generation

Lexical Analysis:

Here the entire program is spilt into the smallest possible piece and then stored. These pieces called tokens are then sent to the parser for syntax analysis.
In our program, the approach we used for this purpose was to scan the entire program, and group all lexemes together.

Parsing:

When a program is being parsed, all its lexemes are matched for syntax analysis. This means that each and every lexeme is checked for order and meaning by the compiler.
There are various approaches for parsing such as top down parsing and bottom up parsing; these can be further divided into left to right and right to left parsing.
An attempt to give a crude outline for each method has been made below.

Top Down (Left To Right)

Here we begin our analysis from the very start of the program and work our way down, like in any mathematical equation. We analyze each statement from left to right, i.e first the leftmost lexeme is broken down and analysed, then the second leftmost and so on.

Top Down (Right to Left)

Here too the analysis begins from the very start of the program but each statement is analysed right to left. This implies that first the very rightmost lexeme is matched and checked, then the second rightmost and so on.

Bottom Up (Left to Right)

Bottom up parsing obviously begins from the end of the program itself and then works its way up to the very beginning of the program. Each statement is analysed from left to right.

Bottom Up (Right to Left)

Here the analysis is the same as above only each statement is analysed from right to left.

The approach we choose for parsing in our compiler was Top Down(Left to Right) parsing. It is a very logical and safe approach and very similar to how one solves a mathematical problem. Therefore this approach was easy to understand and did pose many a problem for us, in terms of algorithm design etc.

The Backus Naur Form (BNF) and syntax

Every natural language has a set of rules which define how it will work, what one can do with it and what words etc it has. Similarly every programming language too has a set of rules which define it completely. This set of rules is called a free context grammar and more widely fit to the standards of the Backus Naur Form of free context grammar. The BNF is a metalanguage which defines the programming language itself.
The BNF for FAJF is very simple and straight forward. Following is a complete break down of the BNF of our language and its various syntax rules.

<prog> : start <stmt> end

Start indicates the beginning of a program.
End indicates the end of a program.
The above two are used to make the language more readable to the user and hence more writeable too.

Variable Declaration:

Initially the language was envisioned to have many variables but for now there are 4 types of variables available to the programmer.

- vs: Voltage Source
- cs: Current Source
- resistor
- int: integer

vs indicates a voltage source based in volts, cs indicates a current source based in amperes, resistor indicates a resistor based in ohms and int indicates a normal integer. Each variable must be assigned before use. It can either be

- declared first, assigned later

- assigned at the time of declaration

Syntax for declaration:

```
Variable type <identifier>;
<Identifier> = value;

Variable type <identifier> = value;
```

For instance:

```
vs ups = 10;

resistor r;
r = 200;
```

Assignment

The statement **<identifier> = value** is basically an assignment statement which assigns a declared variable a value. A variable that has not been declared can not be assigned a value.

If Statement and If Else Statement.

The if and if else statements are vital to any functional high level language since they are used to check conditions and hence make the program "intelligent".

```
if <cond> do <stmt> endif
if < cond> do <stmt> else do <stmt> endelse
```

The if statement starts with a simple if and then must be given a condition to check in order to continue. The next keyword is do which is to be followed by a list of statements or a statement the if must do if the condition is true. If ends with an endif keyword.
The else if statement differs from the if statement; if the condition is not true the user can put down another set of statements. If else must end with an endelse keyword. The statements for a true a condition must be followed by the keywords else and do which must be followed by the set of statements to be executed otherwise.

While Loop

A loop is very important to any language too. It can be used for checking if at any time a certain variable meets a

condition or not. It can also be used to cause a delay or run a certain set of statements till a condition is met.

while <cond> do <stmt> endwhile

The keywords for the while loop are while, endwhile and do.

Display

This statement is used to display a certain variable or any other text. Variables must be declared and assigned before being displayed i.e. printed to the screen. Any other text must be put in quotes (" ").

display <identifier>
display "text to display"

Acquire

This statement is used for getting an input. The keyword acquire can only be followed by a variable where we can store the information retrieved by the input.

acquire <identifier>

End statement

Each statement apart from the start and end statements must be followed by **#**.

<cond> means condition.
<Identifier>/<ident> means variable name or identifier
<stmt> means statement

Basic Operations of FAJF

FAJF has the functionality to do the following operations on its variables:

- add
- subtract
- divide
- multiply
- add in series
- add in parallel

Conditional Operators in FAJF

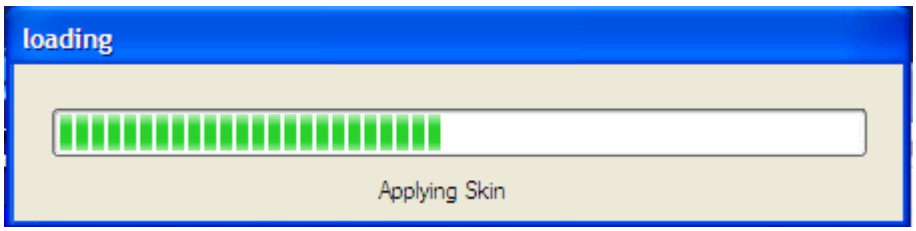
- greater than
- less than
- is equal
- is not equal
- greater than equal
- less than equal

More over FAJF can also take in Boolean values for conditions.

The other details of the BNF will be explained in the section program structure.

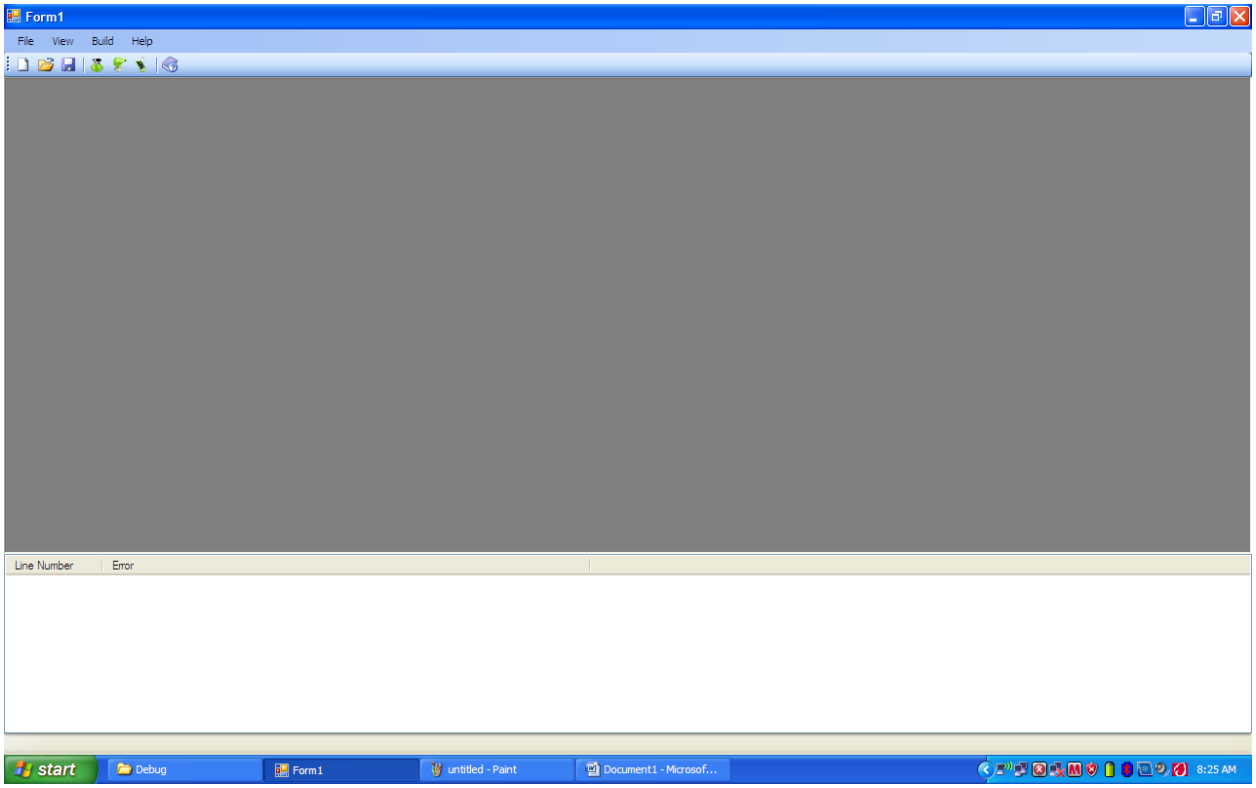
Program Interface and Execution

Loading

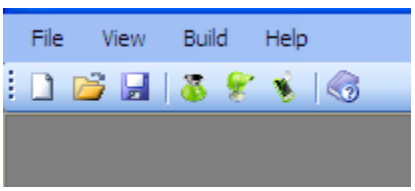


First of all the application loads. This loading form was generated using a timer and a progress bar, where the progress bar value was incremented at every timer tick.

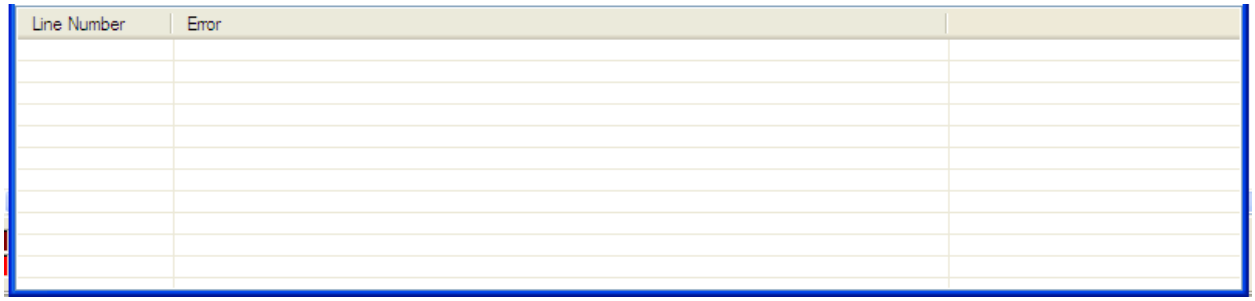
After Loading



When the GUI is fully loaded with no user interaction. It looks as above. All other

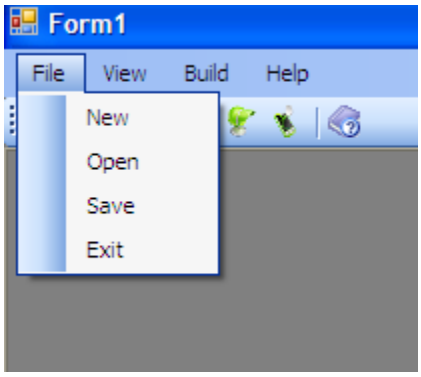


windows will be opened in this window. A closer look at this main window shows us that it consists of a toolstrip and a menustrip. It also has a list view that displays errors.



MenuStrip and other related functions

The menustrip has the following functions:

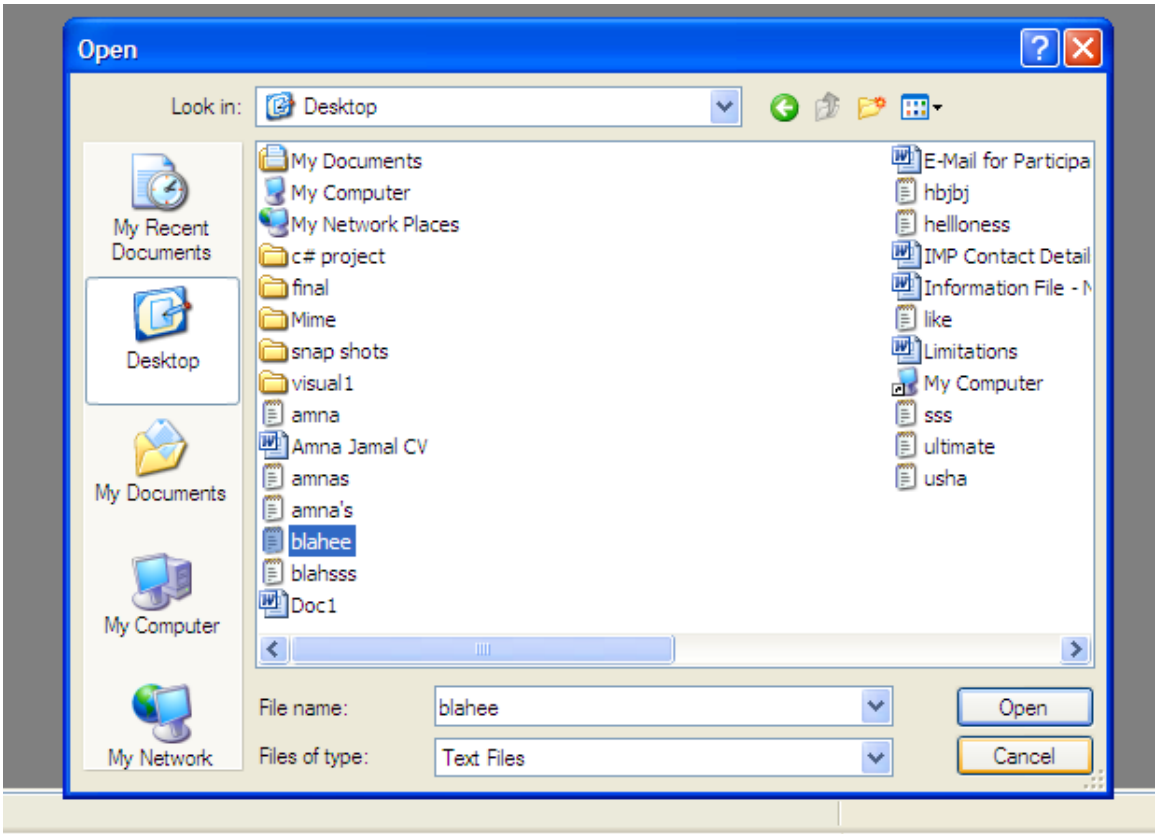


File:

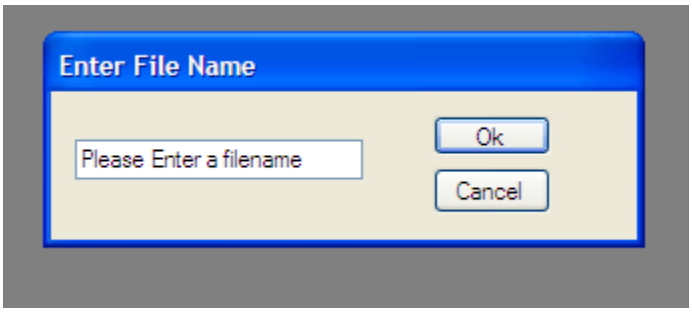
- New
- Open
- Save
- Exit

On the toolstrip the first three options exit as the first three pictures and in the same order.

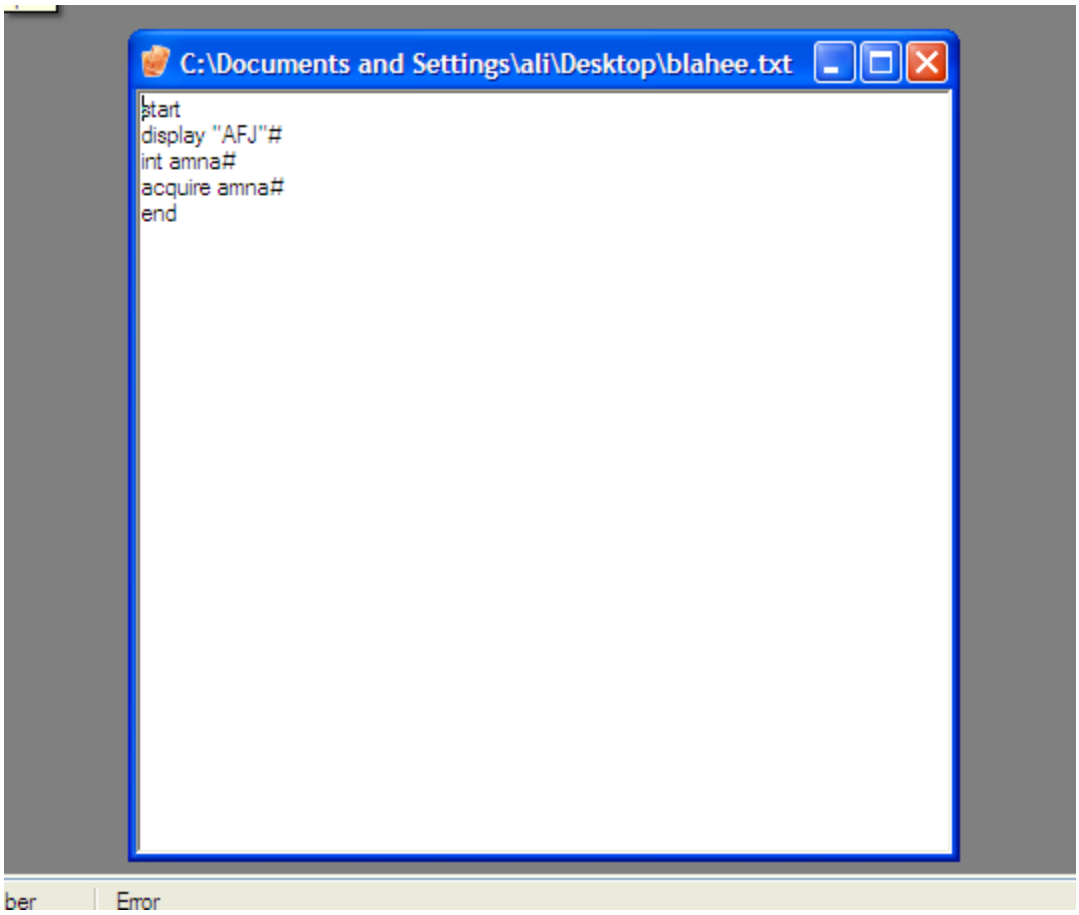
Each command is executed by a click event. For saving a file the savefile dialog class is used and then the data is gotten from the active form. If no form is active then an error is generated.



Similarly for the open file command the openfile dialog is used and data is sent to the text box in the active child form.

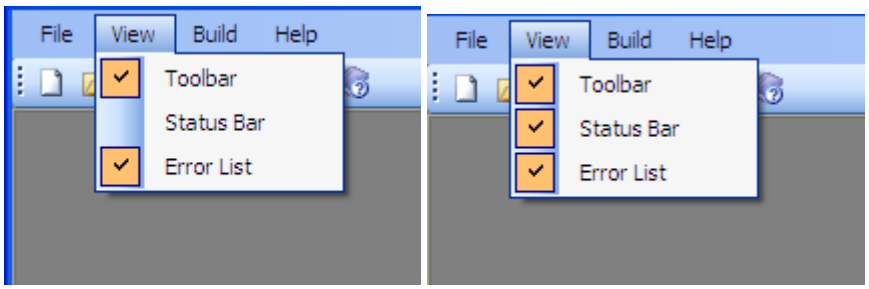


For new a new file form containing a rich text box is opened. First the user is promted to enter a file name. If the user presses ok without entering a filename then the user is given the option to cancel the request, or enter a filename. Exit exits the application.



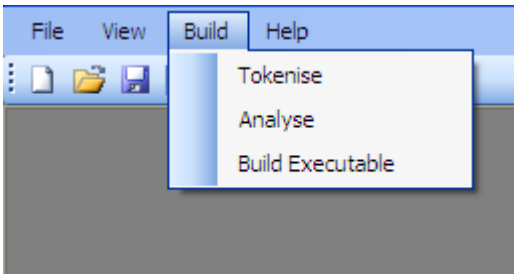
View

- Toolbar
- Status Bar
- Error List



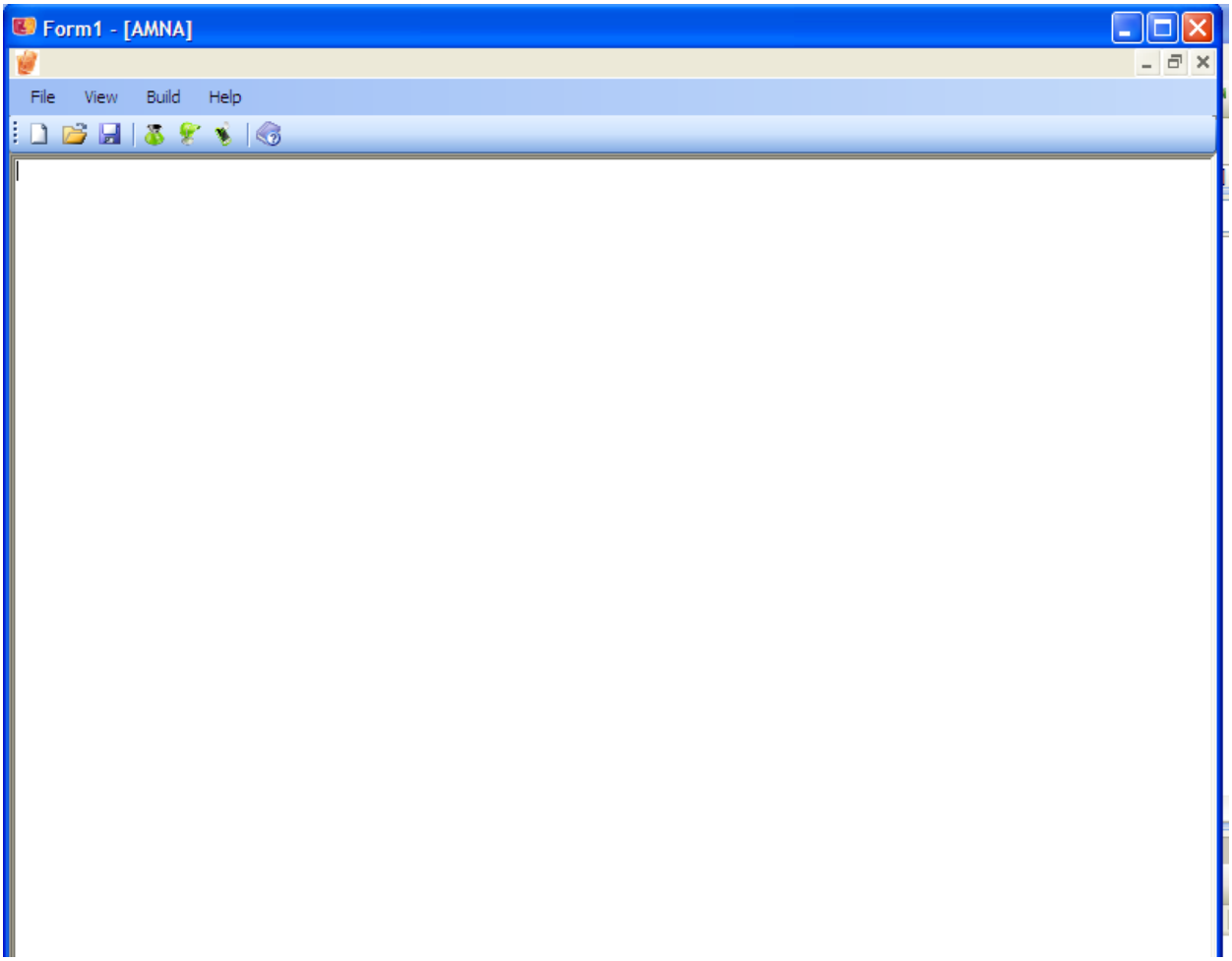
When the user clicks a checked item, it is unchecked and hidden from the user. When the user clicks an unchecked item it is shown and checked again.
This is done by setting the visible property of the respective control to true and false by a simple if statement check.

Build

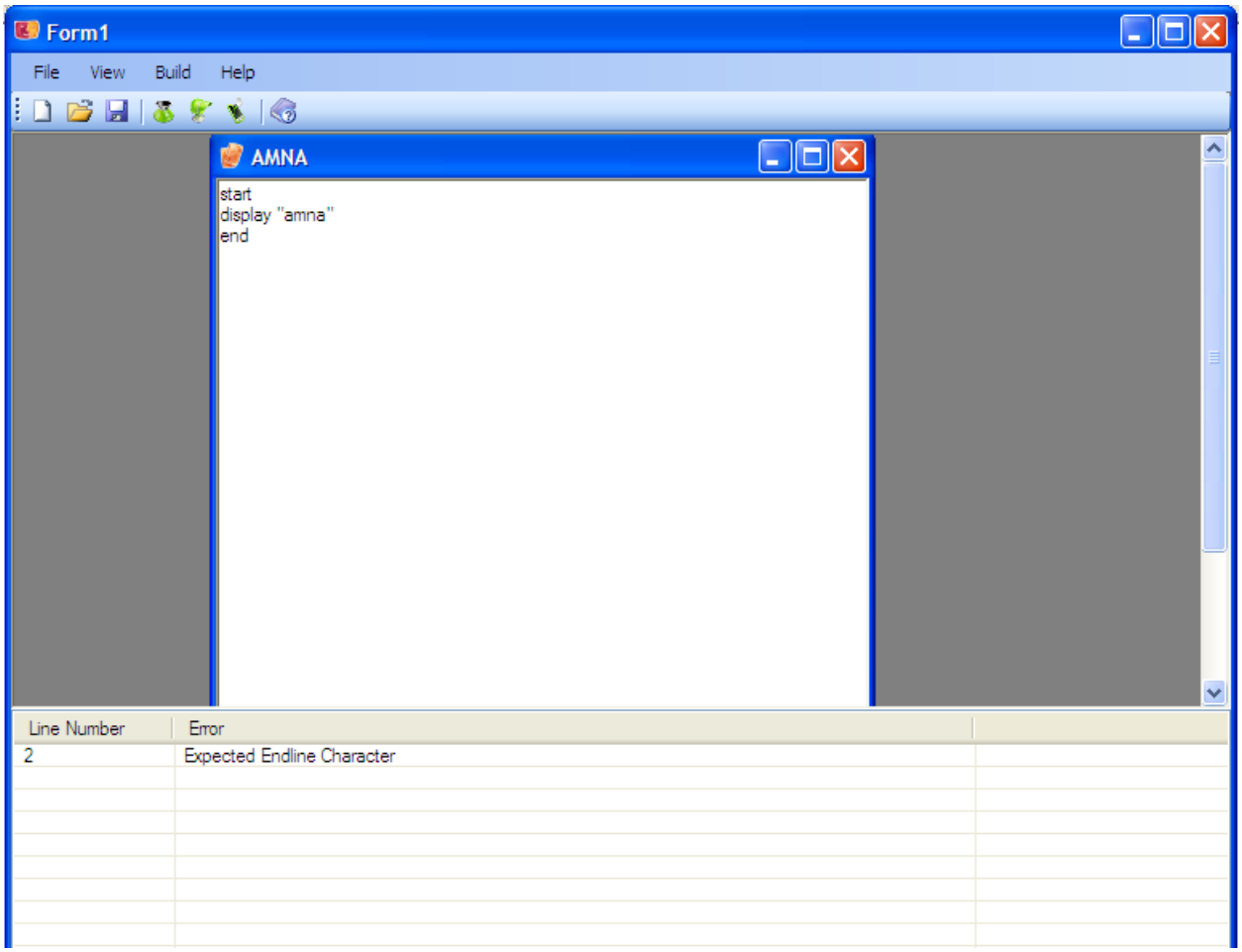


- Tokenise
- Analyse
- Build Executable

On the toolstrip the second three (green) icons are tokenise, analyse and build executable and perform the same methods.

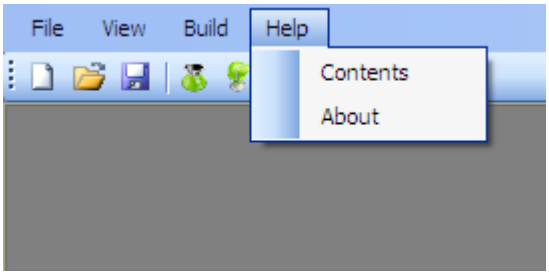


Each event takes place on the on click even.
Tokenise calls the scan method of the class scanner, while analyse calls the constructor of the class parser. Build executable calls the code generator class's constructor. There is a check in each method whereby if a file is analysed or built before it is tokenised or analysed , it is automatically tokenised and analysed if the file is not empty.



Help

- About
- Contents



The help contents can also be accessed by pressing the last icon in the toolstrip.
Help is accessed by opening a new thread and sending the help file path. On clicking about the about form is displayed.

Program Structure

Data Structures:

The most vital part of this entire compiler is the collection of classes the entire BNF. Next come the classes that are used to facilitate the functions of the scanner and the parser and the code generator.

Structures for BNF:

1. abstract class stmt
2. class DeclareS : stmt
3. class Just_Declare : stmt
4. class assign : stmt
5. class display : stmt
6. class ifS : stmt
7. class ifElse : stmt
8. class whileLoop : stmt
9. class acquire : stmt
10. class sequence : stmt
11. abstract class expr
12. class StringLiteral : expr
13. class IntLiteral : expr
14. class Variable : expr
15. class Something : expr
16. class BinExpr : expr
17. class booleanL : expr
18. class condExpr : expr
19. enum BinOp

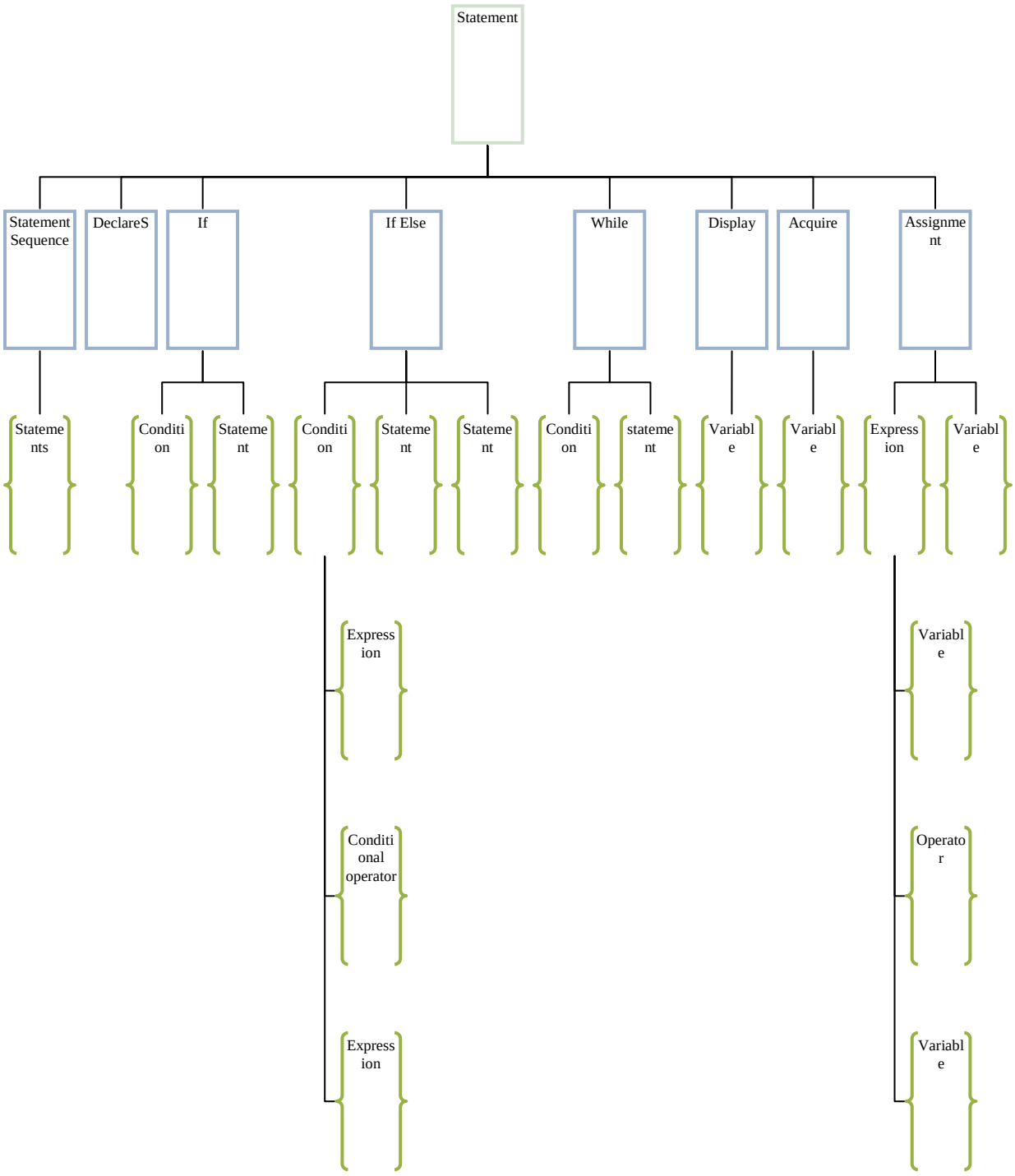
A: B indicates that A has been derived from class B. All classes are defined in order to facilitate the formation of a tree which the parser can parse easily and the code generator can use to generate code.
The enumeration BinOp stores all possible operators.

Below is a diagram that explains the basic tree structure for the entire language. Recursions are excluded and the reader can build upon the structure themselves.

```
public enum BinOp
{
    add,
    minus,
    multiply,
    divide,
    series,
    parallel,
    equale,     //==
    greater,    //>
    note,        //~
    lesser,       //<
    greatere,   //>=
    lessere,      //<=
    empty

}
```

An abstract class only allows that all the other classes be defined as subclasses of this class.


Statement
Statement Sequence
DeclareS
If
If Else
While
Display
Acquire
Assignment
Statements
Condition
Statement
Condition
Statement
Statement
Condition
statement
Variable
Variable
Expression
Variable
Expression
Conditional operator
Expression
Variable
Operator
Variable

AST

Parser

The parser contains:

- class identifier_check which is used to check whther an idenfier is assigned (stored in bool ilist) and its name(sotred in string ilist)
- class errorlist which is used to store errors and their corresponding line number
- the parser class itself which has the variables
  - index used to count tokens from the scanner
  - tokens, an ilist used to store tokens from scanner
  - result, a stmt type variable used to store statements
  - indentifier, an indentifier_check variable
  - lc, an integer used to count lines

Scanner

The Scanner contains:

- A scanner type class that has:
  - er, an errorlist type ilist used to store scanner errors
  - whole, an object type ilist used to store each lexical token
  - line, an integer used to count line numbers
  - opend, an enumeration type used to store all possible operators and endline and newline tokens

Code Generator

The Code Generator contains:

- a code generator class that hs:
  - and Emit.ILGenerator object il used to
  - A dictionary type ilist used to

The inbuilt C# structure we have used the most is the iList since it is dynamic and allows many different types of variables to be stored.

GUI Structure

Following is a breakdown of the most important controls that form the GUI of the compiler graphical interface.

MainForm

- Status strip
- Tool strip
- Save file dialog
- Open file dialog
- Menu strip
- List view

Loading Form

- Progress bar
- Label
- Timer

TextFile Form (txtbf)

- Rich text box

Filename Form

- Textbox
- Button

Funtions

Scanner

Void Scanning(char[] x)

The most important function of the scanner is the scanning function. It takes the a character array and then checks each character as follows:

- All other whitespaces are checked and ignored, but newlines are counted and stored.
- If a character is a quote then the following characters must of type string literal, quote is then ignored and all subsequent characters are stored in the same string till another quote is encountered. Moreover an error is generated if an ending quote is not encountered.

- $ indicates a comment and therefore all subsequent characters are ignored till the line ends
- All digits are stored in a separate stringbuilder variable
- All operators and endline characters are stored as objects
- All other characters are stored in a separate stringbuilder variable

This function utilizes the class StringBuilder to create stringbuilder type variables and add them to the tokens list.

IList<object> Tokens

This property gets the iList whole used to store all tokens. It is public and is used by the parser.

Scanner(char[] readI)

This is the constructor and takes a character array as its argument. This character array is read from a textbox in the GUI

Void adderror(string x)

This adds the error message (which it takes in x) to the errorlist ilist.

Parser

Stmt Result

This is a property that gets the variable result from the class parser to be used for code generation.

Parser(IList<objects> x)

This takes the tokens list from the scanner and is the constructor for a parser object. It then sets the variable index to 0, creates a new identifier_check variable and then checks whether the start statement is found or not. Next it calls the function ParseStmt() and stores its return value in the variable result. In the end it checks

whether the program has ended properly or not using the end statement.

Void linecount()

This counts the line number of the current token and is used by the adderror function to store the error and line number. It increments lc.

Stmt ParseStmt()

This function declares a stmt type variable result and then proceeds to check each token for a corresponding statement. All subregions of this function check whether the endline character # has been encountered or not. Similarly all subregions generate errors that are added to the errorlist when statement syntax is not in accordance to the BNF rules. All subregions also store the statement in the variable result. The variable result is returned by this function.
Since all regions follow the same operations here we will only describe one all encompassing region.
While Loop Region:

1. Check whether the current token equals the keyword while.
2. Increment the token index and call the linecount function.
3. Create a new whileLoop class variable and then call the ParseCond() function.
4. Check whether the current token is equal to the keyword do. If not add error to error list, else

```
else if (tokens[index].ToString().Equals("while"))
                {
                    index++;
                    linecount();
                    whileLoop whilel = new whileLoop();
                    whilel.cond = ParseCond();

                    if (index == tokens.Count ||
!(tokens[index].ToString().Equals("do")))
                    {

adderror("Expected do");
                        index--;
                    }
                    index++;
                    linecount();
                     whilel.body = ParseStmt();
 if (index == tokens.Count ||
!(tokens[index].ToString().Equals("endwhile")))
                    {
                        adderror("unterminated 'while'
loop body");
                    }
                    index++;
                    linecount();
                    if (index == tokens.Count ||
!(tokens[index].Equals((object)scanner.opend.endline)))
                    {
                        adderror("Expected endline #");
                        index--;
                    }
                    result = whilel;
                    index++;
                    linecount();

                }
```

increment index and call linecount.
5. Check whether the current token equals the endline character, otherwise generate error and decrement index.
6. Assign the whileloop variable to the variable result and increment index.

Display Region:
This region checks whether the variable has been declared or not and then assigned.

Declare Region:
This region checks whether a variable has been declared and assigned.

Acquire Region:
This region checks whether the acquire statement is using a variable or a string literal.

If and If Else Region:
This region checks whether a conditional operator is encountered or not, it then goes on to call the ParseStmt function again to parse the expected statement. It then checks whether an endif or an else is encountered. If an else is encountered it creates a new ifelse variable which then stores the second and first statements and calls ParseStmt again.
It also checks whether the if and if else have been ended properly using endif and endelse keywords respectively.

Assignment Region:
This region checks whether a variable has been declared earlier and assigned properly.

Invalid Statement Region:
This region adds an error if any other token other than the keywords for the beginning of statements are encountered since all other tokens are checked inside the individual statement regions.

Statement List Region:
This region checks whether a statement list i.e. a sequence is encountered.

Expr ParsExpr(string x)

This method returns a variable of type expr and takes a string as its arguments. It checks whether a token is of stringliteral type, integer type or a variable.
For both integer and variable it then checks whether the expression needs to be parsed further. Here we will discuss only the integer type checking region since it is the most definitive of all regions.
First the token is stored, and then index is incremented. We then check whether an operator has been experienced, if so we check whether it is a conditional operator or simple arithmetic operator. In the former case it checks whether a conditional operator is expected or not and if not generates an error.
If the next token is a variable it checks whether it is declared and assigned or not and generates a corresponding error.
It then finally return the whole expr.

BinOp ParseA_op()

This function identifies and returns the name of the operator used.

Expr ParseCond()

This method is similar to the ParsExpr method and even calls it to check expressions etc. It only has an added Boolean checking region.

Code Generation

CodeGen(stmt s, string m)

This is the constructor for the code generator class. It takes the parser's result variable and the location of the stored file as its arguments. It then checks whether the stored file and the given file are in the same location. Next it creates a dynamic assembly (namespace) using reflect.assemblyname and emit.assemblybuilder. Then it assigns a dynamic module which assigns space for the new class type we will create. Next we create the class type using typebuilder. Then the methodbuilder is called to map the main method.

Next we create space for our function, physically. Then we initialize our symbol table.
Next we call the function GenStmt and pass stmt to it.
Next il.Emit adds the generated instruction to the instruction stack. We then create a new type and then insert our method into the main method. We then reset the symbol table and the instruction. In the end we begin to execute our statement.

GenStmt(stmt stmt)

This method generates the corresponding statement and stores the corresponding variable in the local stack. It has regions for declaration, assignment, display and acquire.
It also calls the GenExpr function in order to parse the assignment statement further. For display it maps this function using il.Emit onto the Write function of c#. Similarly for acquire function we map it onto the readline function of C# and then parse the following integer (since we have only considered integers so far). Next we store the integer in our stack.

Store(string n, System.Type type)

This method stores the corresponding variable in memory by first allocating it space and then pushing it in the stack.

GenExpr(expr expr, System.Type expectedType)

This method is used for type checking. First it checks type then pushes the value onto its corresponding stack. If the value is a variable it then checks whether the variable is declared or not. If so it gets the value of the variable and pushes it onto the stack.
If the delivered type is int and expected type is string it converts the int to string type in order to display it.

TypeOfExpr(expr expr)

This method returns the type of expression. If the expression is a variable it then gets the value from the symbol table and returns its type.

GUI

All GUI functions are explained in the program interface and execution section.

LIMITATIONS AND IMPROVEMENTS

Our language is by no standards complete. It is a very basic language at this time due to the time constraints.

Most importantly many other functions and types should be added; capacitors, inductors, opamps, methods for determining capacitor current, voltage, inductor current and voltage, opamp gain , output etc etc.

Due to lack of these this language lacks functionality.

Secondly the code generation is incomplete and only applies to assignment, declaration, acquire and display.It could be extended to the entire language.

Simple arithmetic functions are not yet implemented due to incomplete code generation.

In addition to this only integer types can be implemented in code generation at this time and acquire only takes in integer inputs.

Moreover the syntax defines no operator precedence and there is no type checking for BinExpr.

Many improvements could be made to the language itself as mentioned above, moreover the GUI itself could be improved by coloring keywords at compile time, adding autocomplete to the files, adding an edit file etc etc

Appendix A: CODE

Scanner

```
public partial class scanner
    {
        #region Complete Scanner

        private readonly IList<object> whole = new List<object>();
        public IList<errorlist> er = new List<errorlist>();
        int line = 0;

        /*  public class estore
          {
             public string before;
             public string after;
              public estore(string b, string a)
              {
                  before = b;
                  after = a;
              }
          }
          public IList<estore> list = new List<estore>();*/
        public scanner(char[] readI)
        {
            Scaning(readI);
        }

        public IList<object> Tokens
        {
            get { return whole; }
        }
        public enum opend
        {
            series,//++
            parallel,//||
            add,//+
            minus,//-
            divide,///
            multiply,//*
            equal,//=
            endline,//#
            greater,//>
            lesser,//<
            greatere,//>=
            lessere,//=<
            note,//~
            equale,//==
            newline// new line

        };
        // int n = 0;
        //int prevc=1;
```

```
        public void adderror(string x)
        {
            er.Add(new errorlist(x,line));
        }
        void Scaning(char[] readI)
        {

            for (int i = 0; i <= readI.Length; i++)
            {
                if (i == readI.Length)
                    break;
                char y = readI[i];
                bool quote = false;

                #region Whitespace Check and new line check
                if (char.IsWhiteSpace(y))
                {
                    if (y == '\n')
                    {
                        //StringBuilder add = new StringBuilder();
                        //add.Append(scanner.opend.newline);
                        //MessageBox.Show("new line");
                        whole.Add(opend.newline);
                        line++;
                    }



                    ////MessageBox.Show("WhiteSpace");
                    continue;
                }
                #endregion
                #region checking for string literal
                else if (y == '"')
                {
                    StringBuilder add = new StringBuilder();
                    y = readI[++i];
                    //add.Append('"');
                    quote = true;
                    while (y != '"')
                    {
                        add.Append(y);
                        ////MessageBox.Show("String literal "+
y.ToString());
                        i++;
                        if (i == readI.Length)
                        {
                            // i--;
                            break;
                        }
                        y = readI[i];
                    }
                    ////MessageBox.Show("Full thingi " + add);
                    if (y == '"')
                    {
                        quote = false;
                        whole.Add(add);
```

```
            ////MessageBox.Show(add.ToString());
        }
        if (quote)
        {


            ////MessageBox.Show("Put a \" here");
        }

        // //MessageBox.Show(add.ToString());
    }
    #endregion
    #region checking for comments
    else if (y == '$')
    {
        y = readI[++i];
        while (y != '\n')
        {
            i++;
            if (i == readI.Length)
            {
                break;
            }
            y = readI[i];

        }

    }
    #endregion
    #region checking for characters and _
    else if (char.IsLetter(y) || y == '_')
    {
        StringBuilder add = new StringBuilder();
        y = readI[i];
        while (char.IsLetter(y) || y == '_')
        {
            add.Append(y);
            ////MessageBox.Show("Syntax" + y.ToString());
            i++;
            if (i == readI.Length)
            {
                //i--;
                break;
            }
            y = readI[i];
        }
        whole.Add(add.ToString());
        //if (whole[0] is string)
        //  //MessageBox.Show(whole[0].ToString());

        i--;

    }
    #endregion
    #region checking for int literal
```

```
                else if (char.IsDigit(y))
                {
                    StringBuilder add = new StringBuilder();
                    y = readI[i];
                    while (char.IsDigit(y))
                    {
                        add.Append(y);
                        ////MessageBox.Show("Is Digit " +
y.ToString());
                        i++;
                        if (i == readI.Length)
                        {
                            // i--;
                            break;
                        }
                        y = readI[i];
                    }
                    i--;
                    whole.Add(Int32.Parse(add.ToString()));

                    // if (whole[0] is Int32)
                    //  //MessageBox.Show("u are in there");
                    ////MessageBox.Show("Full thingi " + add);
                }
                #endregion

                #region operators and endline
                else
                {
                    int n = i + 1;
                    char o;
                    if (n != readI.Length)
                    {
                        o = readI[n];
                    }
                    else
                    {
                        o = ' ';
                    }
                    switch (y)
                    {
                        case '+':
                            {
                                if (o == '+')
                                {
                                    whole.Add(opend.series);

                                    // //MessageBox.Show("Series");
                                    ////MessageBox.Show(y.ToString() +
o.ToString());
                                    i++;
                                }
                                else
                                {
                                    whole.Add(opend.add);

                                    ////MessageBox.Show("Plus");
```

```
                                    ////MessageBox.Show(y.ToString() +
o.ToString());
                                }
                            }
                            break;
                        case '-':
                            {
                                whole.Add(opend.minus);

                            }
                            break;
                        case '/':
                            {
                                whole.Add(opend.divide);

                            }
                            break;
                        case '*':
                            {
                                whole.Add(opend.multiply);

                            }
                            break;
                        case '#':
                            {

                                whole.Add(opend.endline);
                            }
                            break;
                        case '~':
                            {
                                whole.Add(opend.note);

                            }
                            break;
                        case '=':
                            if (o == '=')
                            {
                                whole.Add(opend.equale);
                                i++;

                            }
                            else
                            {
                                whole.Add(opend.equal);

                            }
                            break;
                        case '>':

                            if (o == '=')
                            {
                                whole.Add(opend.greatere);
                                i++;

                            }
                            else
```

```
                            {
                                whole.Add(opend.greater);

                            }

                            break;
                        case '<':

                            if (o == '=')
                            {
                                whole.Add(opend.lessere);

                                i++;
                            }
                            else
                            {
                                whole.Add(opend.lesser);

                            }

                            break;
                        case '|':
                            if (o == '|')
                            {
                                whole.Add(opend.parallel);

                                i++;
                            }
                            break;
                        default:
                           adderror("Scanner Unrecognized Char" +
y.ToString());
                           break;
                    }


                }
                #endregion
                #region checking for last line
                if ((i == readI.Length - 1) && (readI[readI.Length - 1]
!= '\n'))
                {
                    //StringBuilder add = new StringBuilder();
                    // add.Append(scanner.opend.newline);
                    whole.Add(opend.newline);
                    //MessageBox.Show("new line out box");
                    continue;
                }
                #endregion

            }
            //  for(int i=0;i<whole.Count;i++)
            //MessageBox.Show(whole[i].ToString()+(whole[i] is
StringBuilder));
        }
        #endregion
```

```
    }
```

Parser

```
   #region PARSER
    #region idnetifier_check class

    public class identifier_check
    {
        public IList<bool> assignment;
        public IList<string> identity;

        public identifier_check()
        {
            assignment = new List<bool>();
            identity = new List<string>();
        }

    }
    #endregion
    #region Errorlist class

    public class errorlist
    {
        public string error;
        public string ln;
        public errorlist(string err, int l)
        {
            error = err;
            ln = (l).ToString();
        }

    }


    #endregion

    public partial class parser
    {

        public int index;
        public IList<object> tokens;
        private readonly stmt result;
        public identifier_check identifier;
        public int lc = 0;
        // public IList<scanner.estore> elist = new
List<scanner.estore>();
         public IList <errorlist> errors = new List<errorlist>();

        #region add errors function
        public void adderror(string x)
        {
            errors.Add(new errorlist(x, lc));
        }
        #endregion
```

```
        public stmt Result
        {
            get
            {
                return result;
            }


        }
        public parser(IList<object> tanz)
        {
            tokens = tanz;
            index = 0;
            //elist = list;

            identifier = new identifier_check();


            #region Checking for Start
            if (tokens.Count != 0)
            {
                if (tokens[0].ToString().Equals("start"))
                {
                    index++;
                    linecount();
                }
                else
                {
                    // lc++;
                    adderror("expected start");
                }
            #endregion

                result = ParseStmt();

                #region checking for proper end
                if (tokens[tokens.Count - 1].ToString().Equals("end"))
                {
                    index++;
                    linecount();
                   // MessageBox.Show("The first one");
                }
                else if (tokens[tokens.Count -
1].Equals((object)scanner.opend.newline) && tokens[tokens.Count -
2].ToString().Equals("end"))
                {
                    linecount();
                   // MessageBox.Show("The second one");
                }
                else
                {
                    adderror("EOF expected");
                  //  MessageBox.Show("The only error one");
                }

                /*   if (index != tokens.Count)
                    {
```

```
                        linecount();
                        MessageBox.Show("the last one" +
tokens.Count.ToString() + index.ToString());

                        ////MessageBox.Show(tokens[index].ToString());
                        adderror("EOF expected");
                    }*/
                #endregion
            }
            ////MessageBox.Show(lc.ToString());
        }
        private void linecount()
        {
            if (tokens.Count != 0 && !(index >= tokens.Count))
            {
                if (tokens[index].Equals(scanner.opend.newline))
                {

                    lc++;
                    ////MessageBox.Show("new line here at " +
(tokens[index - 1]).ToString() + lc.ToString());
                    if (index != tokens.Count)
                        index++;

                }
            }
        }


        private stmt ParseStmt()
        {
            stmt result;
            #region newline
            linecount();
            #endregion

            if (index < tokens.Count - 1)
            {
                #region DISPLAY
                if (index != tokens.Count - 1 &&
tokens[index].ToString().Equals("display"))
                {
                    //   linecount();
                    // ////MessageBox.Show("i am in display");
                    display Display = new display();
                    index++;
                    linecount();
                    Display.Expr = ParsExpr("blah");
                    if (Display.Expr is StringLiteral || Display.Expr
is Variable)
                    {
                        if (Display.Expr is Variable)
                        {
                            if
(!identifier.identity.Contains(tokens[index - 1].ToString()))
                            {
                                adderror("undeclared identifier");
```

```
                                linecount();
                                // index++;
                            }
                            else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index -
1].ToString())] == false)
                            {
                                adderror(tokens[index - 1].ToString() +
" : unassigned");
                                linecount();
                                // index++;
                            }
                        }
                        if
(!tokens[index].Equals(scanner.opend.endline))
                        {
                            adderror("Expected Endline Character");
                            index--;
                        }
                        //else
                        //{
                        index++;
                        linecount();
                        //}
                        result = Display;
                        //
////MessageBox.Show(tokens[index].ToString());

                    }
                    else
                    {
                        result = null;
                        adderror("expected Variable or Display
Statement");
                        index++;
                        linecount();

                        if
(!tokens[index].Equals(scanner.opend.endline))
                        {
                            adderror("Expected Endline Character");
                            index--;
                            //index++;
                        }
                    }
                }
                #endregion

                #region For vs and cs and register and int
                else if (tokens[index].ToString().Equals("vs") ||
tokens[index].ToString().Equals("cs") ||
tokens[index].ToString().Equals("register") ||
tokens[index].ToString().Equals("int"))
                {

                    index++;
                    linecount();
```

```
                    DeclareS sources = new DeclareS();
                    if (!(index < tokens.Count && tokens[index] is
string))
                    {
                        adderror("expected variable name");
                        index++;
                        linecount();

                    }
                    else if
(identifier.identity.Contains(tokens[index].ToString()))
                    {
                        adderror(tokens[index].ToString() + " :
redefinition");
                        index++;
                        linecount();
                    }
                    else
                        sources.Ident = tokens[index].ToString();
                    index++;
                    linecount();

                    if ((index == tokens.Count) ||
(!(tokens[index].Equals((object)scanner.opend.equal)) &&
!(tokens[index].Equals(scanner.opend.endline))))
                    {

//////MessageBox.Show(tokens[index].ToString());
                        adderror("Expected #/=");
                        // index++;
                    }
                    #region var identifier
                    if
(tokens[index].Equals((object)scanner.opend.endline))
                    {
                        Just_Declare classy = new Just_Declare();
                        classy.Ident = tokens[index - 1].ToString();
                        if (identifier.identity.Contains(tokens[index -
1].ToString()))
                        {
                            adderror(tokens[index - 1].ToString() + " :
redefinition");
                            index++;
                            linecount();
                        }

identifier.identity.Add(classy.Ident.ToString());

                        identifier.assignment.Add(false);

                        result = classy;
                        index++;
                        linecount();

                    }
                    #endregion
```

```
                    else
                    {

                        index++;
                        linecount();

                        sources.Expr = ParsExpr("blah");
                        if (sources.Expr is StringLiteral)
                        {
                            adderror("expected variable or integer
after '=' ");
                            //index++;
                        }
                        else if (sources.Expr is Variable)
                        {
                            if
(!identifier.identity.Contains(tokens[index - 1].ToString()))
                            {
                                //////MessageBox.Show();
                                adderror(tokens[index - 1].ToString() +
" undeclared identifier");
                                //  index++;
                            }

                            else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index -
1].ToString())] == false)
                            {
                                adderror(tokens[index - 1].ToString() +
" : unassigned");
                                // index++;
                            }
                        }


identifier.identity.Add(sources.Ident.ToString());
                        identifier.assignment.Add(true);
                        if
(!tokens[index].Equals(scanner.opend.endline))
                        {

//////MessageBox.Show(tokens[index].ToString());
                            adderror("Exepected Endline Character");
                            index--;
                        }
                        index++;
                        linecount();

                        result = sources;
                    }

                }
                #endregion

                #region for Acquire
                else if (tokens[index].ToString().Equals("acquire"))
```

```
                {
                    index++;
                    linecount();
                    acquire acquire = new acquire();
                    if (index < tokens.Count && tokens[index] is
string)
                    {
                        acquire.Ident = tokens[index].ToString();
                        ////MessageBox.Show(tokens[index].ToString());
                        if
(!identifier.identity.Contains(acquire.Ident.ToString()))
                        {
                            adderror("Expected declared Identifier");
                            //index++;
                        }

                        result = acquire;

identifier.assignment[identifier.identity.IndexOf(tokens[index].ToStrin
g())] = true;
                        index++;
                        if
(!tokens[index].Equals(scanner.opend.endline))
                        {
                            adderror("Expected Endline Character");
                            index--;
                            //index++;
                        }

                        linecount();
                    }
                    else
                    {
                        result = null;
                        adderror("Expected Identifier");
                    }
                    index++;
                    linecount();
                }
                #endregion

                #region If and If Else
                else if (tokens[index].ToString().Equals("if"))
                {
                    bool ife = false;
                    index++;
                    linecount();
                    ifS ifs = new ifS();
                    ifElse ifelse = new ifElse();
                    ifs.Cond = ParseCond();
                    if (ifs.Cond is IntLiteral || ifs.Cond is Something
|| ifs.Cond is Variable)
                    {
                        adderror("Expected Conditional Statement");
                        //index++;
                    }
                    //index++;
```

```
                    if (index == tokens.Count ||
!(tokens[index].ToString().Equals("do")))
                    {
                        adderror("Expected do");
                        index--;
                    }
                    index++;
                    linecount();
                    ifs.Body = ParseStmt();
                    // ////MessageBox.Show(tokens[index].ToString() + "
b4 else");
                    //index++;

                    #region ifelse
                    if (tokens[index].ToString().Equals("else"))
                    {
                        index++;
                        linecount();

                        ifelse.ifEverything = ifs;
                        if (index == tokens.Count ||
!(tokens[index].ToString().Equals("do")))
                        {
                            adderror("Expected do");
                            index--;
                        }
                        index++;
                        linecount();
                        ifelse.bodyelse = ParseStmt();
                        //index++;
                        if (index == tokens.Count ||
!(tokens[index].ToString().Equals("endelse")))
                        {
                            adderror("unterminated 'if else' body");
                            //index++;
                        }

                        ife = true;



                    }
                    #endregion
                    else if (index == tokens.Count ||
!(tokens[index].ToString().Equals("endif")))
                    {
                        result = null;
                        adderror("unterminated 'if' body");
                        index++;
                    }
                    if (!ife)
                    {
                        index++;
                        linecount();
                        if (index == tokens.Count ||
!(tokens[index].Equals((object)scanner.opend.endline)))
                        {
```

```
                            result = null;
                            adderror("Expected endline character #");
                            index--;
                        }
                        else
                            result = ifs;
                    }
                    else
                    {
                        index++;
                        linecount();
                        if (index == tokens.Count ||
!(tokens[index].Equals((object)scanner.opend.endline)))
                        {
                            result = null;
                            adderror("Expected endline character #");
                            index--;
                        }
                        else
                            result = ifelse;
                    }
                    index++;
                    linecount();
                }
                #endregion

                #region While loop
                else if (tokens[index].ToString().Equals("while"))
                {
                    index++;
                    linecount();
                    whileLoop whilel = new whileLoop();
                    whilel.cond = ParseCond();
                    ////MessageBox.Show(tokens[index].ToString());
                    //index++;

                    if (index == tokens.Count ||
!(tokens[index].ToString().Equals("do")))
                    {
                        ////MessageBox.Show(tokens[index].ToString());
                        adderror("Expected do");
                        index--;
                    }
                    index++;
                    linecount();
                    //////MessageBox.Show(tokens[index].ToString());
                    whilel.body = ParseStmt();
                    ////MessageBox.Show(tokens[index].ToString());
                    //index++;
                    if (index == tokens.Count ||
!(tokens[index].ToString().Equals("endwhile")))
                    {
                        adderror("unterminated 'while' loop body");
                    }
                    index++;
                    linecount();
```

```
                    if (index == tokens.Count ||
!(tokens[index].Equals((object)scanner.opend.endline)))
                    {
                        adderror("Expected endline #");
                        index--;
                    }
                    result = whilel;
                    index++;
                    linecount();
                    ////MessageBox.Show(tokens[index].ToString()+" in
while");
                }
                #endregion

                #region Assignment
                else if (tokens[index] is string)
                {
                    assign assign = new assign();
                    if
(!identifier.identity.Contains(tokens[index].ToString()))
                        adderror(tokens[index] + " undeclared
identifier");

                    assign.Ident = tokens[index].ToString();
                    index++;
                    linecount();



                    if (tokens.Count == index ||
!(tokens[index].Equals((object)scanner.opend.equal)))
                    {
                        adderror("expected '='");
                    }
                    index++;
                    linecount();
                    assign.Expr = ParsExpr("blah");
                    if (assign.Expr is StringLiteral)
                        adderror("Expected integer/variable");
                    else if (assign.Expr is Variable)
                    {
                        if (!identifier.identity.Contains(tokens[index
- 1].ToString()))
                        {

                            adderror(tokens[index - 1] + " : undeclared
identifier");
                        }

                        else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index -
1].ToString())] == false)
                        {
                            adderror(tokens[index - 1].ToString() + " :
unassigned");
                        }
                    }
```

```
                    result = assign;
                    if (index < tokens.Count)
                    {
                        if (tokens[index].ToString().Equals("#"))
                        {
                            adderror("Expected EndLine Character");
                            index--;

                        }
                    }
                    //index++;
                }
                #endregion

                #region Invalid Statement

                else
                {
                    linecount();
                    adderror("parse error at token " + index + ": " +
tokens[index]);
                    result = ParseStmt();

                }
                #endregion

                #region statement list

                //MessageBox.Show("b4 stmt_list");
                if ((index < tokens.Count) &&
(!(tokens[index].ToString().Equals("end"))))
                {
                    // index++;
                    if (((!tokens[index].ToString().Equals("endwhile")
&& !tokens[index].ToString().Equals("endif"))) &&
(!tokens[index].ToString().Equals("endelse") &&
!tokens[index].ToString().Equals("else")))
                    {
                        //MessageBox.Show("i m in stmt_list");
                        linecount();
                        //MessageBox.Show(tokens[index ].ToString());


                        sequence seq = new sequence();
                        seq.first = result;
                        seq.second = ParseStmt();
                        result = seq;
                    }
                }


                //}

                #endregion
```

```
                return result;
            }
            else return null;
        }
        private expr ParsExpr(string blah)
        {
            if (index >= tokens.Count)
            {
                adderror("Got EOF, and i expected an expression
idiot");
                return null;

            }
            else if (tokens[index] is StringBuilder)
            {
                string value =
((StringBuilder)tokens[index]).ToString();
                index++;
                linecount();
                StringLiteral stringLiteral = new StringLiteral();
                stringLiteral.Value = value;
                return stringLiteral;

            }
            else if (tokens[index] is int)
            {
                int intValue = (int)tokens[index];
                index++;
                linecount();
                IntLiteral intLiteral = new IntLiteral();
                intLiteral.Value = intValue;
                BinOp x = ParseA_op();
                //MessageBox.Show(x.ToString());

                if (x.Equals(BinOp.empty))
                {
                    //MessageBox.Show("ninop");
                    return intLiteral;
                }
                else
                {
                    // if (x.Equals(BinOp.add) ||
x.Equals(BinOp.divide) || x.Equals(BinOp.minus) ||
x.Equals(BinOp.multiply) || x.Equals(BinOp.parallel) ||
x.Equals(BinOp.series))
                    //{
                    BinExpr thing = new BinExpr();
                    if (!(x.Equals(BinOp.add) || x.Equals(BinOp.divide)
|| x.Equals(BinOp.minus) || x.Equals(BinOp.multiply) ||
x.Equals(BinOp.parallel) || x.Equals(BinOp.series)))
                    {
                        if (blah.Equals("con"))
                            return thing;
                        else
                            adderror("Expected Operator");
```

```
                    }
                    thing.Left = (expr)intLiteral;
                    thing.Op = x;
                    index++;
                    linecount();
                    thing.Right = ParsExpr(blah);
                    //MessageBox.Show(thing.Right.ToString());
                    if (thing.Right is Variable)
                    {
                        if (!identifier.identity.Contains(tokens[index
- 1].ToString()))
                        {
                            adderror(tokens[index - 1].ToString() + ":
undeclared identifier");
                            // index++;
                        }
                        else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index -
1].ToString())] == false)
                        {
                            adderror(tokens[index - 1].ToString() + " :
unassigned");
                            //index++;
                        }

                    }
                    // return thing;
                    // }
                    /*  if (!(x.Equals(BinOp.add) ||
x.Equals(BinOp.divide) || x.Equals(BinOp.minus) ||
x.Equals(BinOp.multiply) || x.Equals(BinOp.parallel) ||
x.Equals(BinOp.series)))
                {
                    adderror("Expected Operator");
                   // //MessageBox.Show("hello ");

                    ////MessageBox.Show(thing.Left.ToString() + " " +
thing.Op.ToString() + " " + thing.Right.ToString());
                    return thing;

                }*/
                  return thing;



                }



            }
            else if (tokens[index] is string)
            {
                string ident = (string)tokens[index];
                index++;
                linecount();
                Variable var = new Variable();
                var.Ident = ident;
```

```
                BinOp x = ParseA_op();
                if (x.Equals(BinOp.empty))
                {
                    return var;
                }
                else
                {
                    BinExpr thing = new BinExpr();



                    thing.Left = (expr)var;
                    if (!identifier.identity.Contains(var.Ident))
                    {
                        adderror(tokens[index - 1].ToString() + ":
undeclared identifier");
                        //index++;
                    }


                    thing.Op = x;
                    index++;
                    linecount();
                    thing.Right = ParsExpr("blah");
                    if (thing.Right is Variable)
                    {
                        if (!identifier.identity.Contains(tokens[index
- 1].ToString()))
                        {
                            adderror(tokens[index - 1].ToString() + ":
undeclared identifier");

                        }

                        else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index -
1].ToString())] == false)
                        {
                            adderror(tokens[index - 1].ToString() + " :
unassigned");
                        }

                    }
                    //return thing;

                    if (!(x.Equals(BinOp.add) || x.Equals(BinOp.divide)
|| x.Equals(BinOp.minus) || x.Equals(BinOp.multiply) ||
x.Equals(BinOp.parallel) || x.Equals(BinOp.series)))
                    {
                        //MessageBox.Show("binop add");
                        adderror("Expected Operator");
                        return thing;
                        // return null;
                    }
                    return thing;
                    //index--;
```

```
                }

            }
            else
            {
                //MessageBox.Show("i am in string literal");
                //MessageBox.Show(tokens[index].ToString());

                adderror("expected string literal, int literal or
variable");
                return null;
            }
        }
        private BinOp ParseA_op()
        {
            if (index == tokens.Count)
            {
                return BinOp.empty;
            }
            if (tokens[index].Equals((object)scanner.opend.add))
            {
                return (BinOp.add);
            }
            else if (tokens[index].Equals((object)scanner.opend.minus))
            {
                return (BinOp.minus);
            }
            else if
(tokens[index].Equals((object)scanner.opend.multiply))
            {
                return (BinOp.multiply);
            }
            else if
(tokens[index].Equals((object)scanner.opend.divide))
            {
                return (BinOp.divide);
            }

            else if
(tokens[index].Equals((object)scanner.opend.series))
            {
                return (BinOp.series);
            }
            else if
(tokens[index].Equals((object)scanner.opend.equale))
            {
                return (BinOp.equale);
            }
            else if
(tokens[index].Equals((object)scanner.opend.greater))
            {
                return (BinOp.greater);
            }
            else if
(tokens[index].Equals((object)scanner.opend.greatere))
            {
                return (BinOp.greatere);
```

```
            }
            else if
(tokens[index].Equals((object)scanner.opend.lesser))
            {
                return (BinOp.lesser);
            }
            else if
(tokens[index].Equals((object)scanner.opend.lessere))
            {
                return (BinOp.lessere);
            }
            else if (tokens[index].Equals((object)scanner.opend.note))
            {
                return (BinOp.note);
            }
            else if
(tokens[index].Equals((object)scanner.opend.parallel))
            {
                return (BinOp.parallel);
            }
            else if (tokens[index].ToString().Equals("="))
            {
                adderror("maybe you should put a == instead of the =");
                return (BinOp.equale);
            }



            else
            {
                return BinOp.empty;
            }
        }
        private expr ParseCond()
        {
            if (index == tokens.Count)
            {
                adderror("Got EOF, and i expected an expression
idiot");
            }
            if (tokens[index] is StringBuilder)
            {
                adderror("expected intger/variable/condition");
                return null;

            }
            else if (tokens[index] is int)
            {
                int intValue = (int)tokens[index];
                index++;
                linecount();
                IntLiteral intLiteral = new IntLiteral();
                intLiteral.Value = intValue;
                BinOp x = ParseA_op();

                if (x.Equals(BinOp.empty))
                {
```

```
                    return intLiteral;
                }
                else
                {
                    BinExpr thing = new BinExpr();
                    if (x.Equals(BinOp.add) || x.Equals(BinOp.divide)
|| x.Equals(BinOp.minus) || x.Equals(BinOp.multiply) ||
x.Equals(BinOp.parallel) || x.Equals(BinOp.series))
                    {
                        index--;
                        //MessageBox.Show(tokens[index].ToString()+"
"+tokens[index+1].ToString()+" "+tokens[index+2].ToString());
                        thing.Left = ParsExpr("con");
                        //MessageBox.Show("back");
                        //adderror("Expected Conditional Operator");
                    }

                    x = ParseA_op();
                    if (x.Equals(BinOp.empty))
                    {
                        Something blahness = new Something();
                        blahness.hello = (BinExpr)thing.Left;
                        //MessageBox.Show("i m in there");
                        return blahness;
                    }

                    thing.Left = (expr)intLiteral;
                    thing.Op = x;
                    index++;
                    linecount();
                    thing.Right = ParsExpr("blah");
                    if (thing.Right is Variable)
                    {
                        if (!identifier.identity.Contains(tokens[index
- 1].ToString()))
                            adderror(tokens[index - 1].ToString() + ":
undeclared identifier");

                        else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index -
1].ToString())] == false)
                        {
                            adderror(tokens[index - 1].ToString() + " :
unassigned");
                        }

                    }
                    return thing;



                }


                // return intLiteral;
```

```
            }
            else if (tokens[index] is string)
            {
                string ident = (string)tokens[index];
                if
(!identifier.identity.Contains(tokens[index].ToString()))
                    adderror(tokens[index].ToString() + " : undeclared
identifier");
                else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index].ToStri
ng())] == false)
                {
                    adderror(tokens[index].ToString() + " :
unassigned");
                }

                index++;
                linecount();
                Variable var = new Variable();
                var.Ident = ident;
                BinOp x = ParseA_op();
                if (x.Equals(BinOp.empty))
                {
                    return var;
                }
                else
                {
                    BinExpr thing = new BinExpr();
                    thing.Left = (expr)var;

                    if (x.Equals(BinOp.add) || x.Equals(BinOp.divide)
|| x.Equals(BinOp.minus) || x.Equals(BinOp.multiply) ||
x.Equals(BinOp.parallel) || x.Equals(BinOp.series))
                    {
                        index--;
                        thing.Left = ParsExpr("con");
                        // if (errors[errors.Count -
1].error.ToString().Equals("Expected Operator"))
                        //  errors.Remove(errors.Count - 1);
                        //adderror("Expected Conditional Operator");
                    }


                    if (!identifier.identity.Contains(var.Ident))
                        adderror(var.Ident + ": undeclared
identifier");
                    else if
(identifier.assignment[identifier.identity.IndexOf(var.Ident)] ==
false)
                    {
                        adderror(tokens[index].ToString() + " :
unassigned");
                    }
                    x = ParseA_op();
                    if (x.Equals(BinOp.empty))
                    {
```

```
                        Something blahness = new Something();
                        blahness.hello = (BinExpr)thing.Left;
                        return blahness;
                    }
                    thing.Op = x;
                    index++;
                    linecount();
                    thing.Right = ParsExpr("blah");
                    if (thing.Right is Variable)
                    {
                        if (!identifier.identity.Contains(tokens[index
- 1].ToString()))
                            adderror(tokens[index - 1].ToString() + ":
undeclared identifier");

                        else if
(identifier.assignment[identifier.identity.IndexOf(tokens[index -
1].ToString())] == false)
                        {
                            adderror(tokens[index - 1].ToString() + " :
unassigned");
                        }
                        //index
                    }
                    return thing;
                }

            }

            else if (tokens[index] is bool)
            {
                bool value = (bool)tokens[index++];
                linecount();
                booleanL boolean = new booleanL();
                boolean.Value = value;
                return boolean;
            }
            else
            {
                adderror("expected string literal, int literal or
variable");
                return null;
            }
        }

    }
    #endregion
```

CodeGenerator

```
public sealed class CodeGen
{
    Emit.ILGenerator il = null;
    Collections.Dictionary<string, Emit.LocalBuilder> symbolTable;

    public CodeGen(stmt stmt, string moduleName)
    {
```

```
        if (IO.Path.GetFileName(moduleName) != moduleName)
        {
            throw new System.Exception("can only output into current
directory!");
        }

        Reflect.AssemblyName name = new
Reflect.AssemblyName(IO.Path.GetFileNameWithoutExtension(moduleName));
        Emit.AssemblyBuilder asmb =
System.AppDomain.CurrentDomain.DefineDynamicAssembly(name,
Emit.AssemblyBuilderAccess.Save);
        Emit.ModuleBuilder modb = asmb.DefineDynamicModule(moduleName);
        Emit.TypeBuilder typeBuilder = modb.DefineType("resister");

        Emit.MethodBuilder methb = typeBuilder.DefineMethod("Main",
Reflect.MethodAttributes.Static, typeof(void), System.Type.EmptyTypes);

        // CodeGenerator
        this.il = methb.GetILGenerator();
        this.symbolTable = new Collections.Dictionary<string,
Emit.LocalBuilder>();

        // Go Compile!
        this.GenStmt(stmt);

        il.Emit(Emit.OpCodes.Ret);
        typeBuilder.CreateType();
        modb.CreateGlobalFunctions();
        asmb.SetEntryPoint(methb);
        asmb.Save(moduleName);
        this.symbolTable = null;
        this.il = null;
        System.Diagnostics.Process.Start(moduleName);
    }


    private void GenStmt(stmt stmt)
    {
        if (stmt is sequence)
        {
            sequence seq = (sequence)stmt;
            this.GenStmt(seq.first);
            this.GenStmt(seq.second);
        }

        else if (stmt is DeclareS)
        {
            // declare a local
            DeclareS declare = (DeclareS)stmt;
            this.symbolTable[declare.Ident] =
this.il.DeclareLocal(this.TypeOfExpr(declare.Expr));

            // set the initial value
            assign assign = new assign();
            assign.Ident = declare.Ident;
            assign.Expr = declare.Expr;
            this.GenStmt(assign);
```

```
        }

        else if (stmt is assign)
        {
            assign assign = (assign)stmt;
            this.GenExpr(assign.Expr, this.TypeOfExpr(assign.Expr));
            this.Store(assign.Ident, this.TypeOfExpr(assign.Expr));
        }
        else if (stmt is display)
        {
            // the "print" statement is an alias for
System.Console.WriteLine.
            // it uses the string case
            this.GenExpr(((display)stmt).Expr, typeof(string));
            this.il.Emit(Emit.OpCodes.Call,
typeof(System.Console).GetMethod("WriteLine", new System.Type[] {
typeof(string) }));
        }
        else if (stmt is Just_Declare)
        {
            Just_Declare declare = (Just_Declare)stmt;
            this.symbolTable[declare.Ident] =
this.il.DeclareLocal(typeof(int));
            //this.Store(declare.Ident, typeof(int));
        }

        else if (stmt is acquire)
        {
            this.il.Emit(Emit.OpCodes.Call,
typeof(System.Console).GetMethod("ReadLine",
System.Reflection.BindingFlags.Public |
System.Reflection.BindingFlags.Static, null, new System.Type[] { },
null));
            this.il.Emit(Emit.OpCodes.Call,
typeof(int).GetMethod("Parse", System.Reflection.BindingFlags.Public |
System.Reflection.BindingFlags.Static, null, new System.Type[] {
typeof(string) }, null));
            this.Store(((acquire)stmt).Ident, typeof(int));
        }


        else
        {
            throw new System.Exception("don't know how to gen a " +
stmt.GetType().Name);
        }




    }

    private void Store(string name, System.Type type)
    {
        if (this.symbolTable.ContainsKey(name))
        {
            Emit.LocalBuilder locb = this.symbolTable[name];
```

```
            if (locb.LocalType == type)
            {
                this.il.Emit(Emit.OpCodes.Stloc,
this.symbolTable[name]);
            }
            else
            {
                throw new System.Exception("'" + name + "' is of type "
+ locb.LocalType.Name + " but attempted to store value of type " +
type.Name);
            }
        }
        else
        {
            throw new System.Exception("undeclared variable '" + name +
"'");
        }
    }



    private void GenExpr(expr expr, System.Type expectedType)
    {
        System.Type deliveredType;

        if (expr is StringLiteral)
        {
            deliveredType = typeof(string);
            this.il.Emit(Emit.OpCodes.Ldstr,
((StringLiteral)expr).Value);
        }
        else if (expr is IntLiteral)
        {
            deliveredType = typeof(int);
            this.il.Emit(Emit.OpCodes.Ldc_I4,
((IntLiteral)expr).Value);
        }
        else if (expr is Variable)
        {
            string ident = ((Variable)expr).Ident;
            deliveredType = this.TypeOfExpr(expr);

            if (!this.symbolTable.ContainsKey(ident))
            {
                throw new System.Exception("undeclared variable '" +
ident + "'");
            }

            this.il.Emit(Emit.OpCodes.Ldloc, this.symbolTable[ident]);
        }
        else
        {
            throw new System.Exception("don't know how to generate " +
expr.GetType().Name);
        }
```

```
        if (deliveredType != expectedType)
        {
            if (deliveredType == typeof(int) &&
                expectedType == typeof(string))
            {
                this.il.Emit(Emit.OpCodes.Box, typeof(int));
                this.il.Emit(Emit.OpCodes.Callvirt,
typeof(object).GetMethod("ToString"));
            }
            else
            {
                throw new System.Exception("can't coerce a " +
deliveredType.Name + " to a " + expectedType.Name);
            }
        }

    }



    private System.Type TypeOfExpr(expr expr)
    {
        if (expr is StringLiteral)
        {
            return typeof(string);
        }
        else if (expr is IntLiteral)
        {
            return typeof(int);
        }
        else if (expr is Variable)
        {
            Variable var = (Variable)expr;
            if (this.symbolTable.ContainsKey(var.Ident))
            {
                Emit.LocalBuilder locb = this.symbolTable[var.Ident];
                return locb.LocalType;
            }
            else
            {
                throw new System.Exception("undeclared variable '" +
var.Ident + "'");
            }
        }
        else
        {
            throw new System.Exception("don't know how to calculate the
type of " + expr.GetType().Name);
        }
    }
}
```

MainForm

```
  public partial class MainForm : Form
    {
        public MainForm()
```

```
        {
            InitializeComponent();
        }
        public static scanner scan;
        public static parser pars;
        private FileStream output;
        private string filename;

        private void Form1_Load(object sender, EventArgs e)
        {
            listView1.Columns.Add("Line Number", 100);
            listView1.Columns.Add("Error", 500);
            listView1.View = View.Details;

        }

        private void Form1_FormClosed(object sender,
FormClosedEventArgs e)
        {
            Application.Exit();
        }

        private void newToolStripButton_Click(object sender, EventArgs
e)
        {

            filename filen = new filename();
            filen.MdiParent = this;
            filen.Visible = true;
            this.ActivateMdiChild(filen);
            filen.TopMost = true;
            statusStrip1.Text = "New File Created";



        }

        private void openToolStripButton_Click(object sender, EventArgs
e)
        {


            //openFileDialog1.ShowDialog(this);
            openFileDialog1.Filter = "Text Files|*.txt; *.text;
*.doc|All Files|*.*";
            openFileDialog1.AddExtension = true;
            openFileDialog1.InitialDirectory = "C:\\";
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                string txtfromfile =
openfile(openFileDialog1.FileName);
                txtbf txt = new txtbf(openFileDialog1.FileName);
                //txt.Name=openFileDialog1.FileName;
                txt.store = txtfromfile;
                txt.MdiParent = this;
                txt.Visible = true;
```

```
                statusStrip1.Text = "File Opened: " +
openFileDialog1.FileName;
            }
            else
                statusStrip1.Text = "Unable to Open File";



        }
        private string openfile(string name)
        {
            StreamReader file = new StreamReader(name);
            return file.ReadToEnd();
        }
        private void savefile()
        {
            txtbf txt = (txtbf)this.ActiveMdiChild;
            if (txt != null)
            {
                statusStrip1.Text = "Saving File: " + txt.Text;
                saveFileDialog1.FileName = txt.Text;
                saveFileDialog1.InitialDirectory = "C:\\";
                saveFileDialog1.Filter = "Text Files|*.txt; *.text;
*.doc|All Files|*.*";
                saveFileDialog1.AddExtension = true;
                if (saveFileDialog1.ShowDialog() == DialogResult.OK)
                {
                    FileInfo n = new
FileInfo(saveFileDialog1.FileName);
                    StreamWriter save = n.CreateText();
                    save.Write(txt.store);
                    save.Close();
                }
            }
            else
                statusStrip1.Text = "No active file";
        }

        private void saveToolStripButton_Click(object sender, EventArgs
e)
        {
            savefile();
        }

        private void newToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            filename filen = new filename();
            filen.MdiParent = this;
            filen.Visible = true;
            this.ActivateMdiChild(filen);
            filen.TopMost = true;
            statusStrip1.Text = "New File Created";
        }

        private void openToolStripMenuItem_Click(object sender,
EventArgs e)
```

```
        {
            //openFileDialog1.ShowDialog(this);
            openFileDialog1.Filter = "Text Files|*.txt; *.text;
*.doc|All Files|*.*";
            openFileDialog1.AddExtension = true;
            openFileDialog1.InitialDirectory = "C:\\";
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                string txtfromfile =
openfile(openFileDialog1.FileName);
                txtbf txt = new txtbf(openFileDialog1.FileName);
                //txt.Name=openFileDialog1.FileName;
                txt.store = txtfromfile;
                txt.MdiParent = this;
                txt.Visible = true;
                statusStrip1.Text = "File Opened: " +
openFileDialog1.FileName;
            }
            else
                statusStrip1.Text = "Unable to Open File";
        }

        private void saveToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            txtbf txt = (txtbf)this.ActiveMdiChild;
            if (txt != null)
            {
                statusStrip1.Text = "Saving File: " + txt.Text;
                saveFileDialog1.FileName = txt.Text;
                saveFileDialog1.InitialDirectory = "C:\\";
                saveFileDialog1.Filter = "Text Files|*.txt; *.text;
*.doc|All Files|*.*";
                saveFileDialog1.AddExtension = true;
                if (saveFileDialog1.ShowDialog() == DialogResult.OK)
                {
                    FileInfo n = new
FileInfo(saveFileDialog1.FileName);
                    StreamWriter save = n.CreateText();
                    save.Write(txt.store);
                    save.Close();
                }
            }
            else
                statusStrip1.Text = "No active file";
        }

        private void exitToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            Application.Exit();
        }

        private void toolbarToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            if (toolStrip.Visible == true)
```

```
                toolStrip.Visible = false;
            else toolStrip.Visible = true;
        }

        private void statusBarToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            if (statusStrip1.Visible == true)
                statusStrip1.Visible = false;
            else
                statusStrip1.Visible = true;
        }

        private void errorListToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            if (listView1.Visible == true)
                listView1.Visible = false;
            else listView1.Visible = true;
        }

        private void toolStripButton1_Click(object sender, EventArgs e)
        {
            listView1.Items.Clear();
            txtbf current = (txtbf)this.ActiveMdiChild;
            if (current != null)
            {
                if (!string.IsNullOrEmpty(current.store))
                {
                    string x = current.store;
                    //  MessageBox.Show(x);
                    scan = new scanner(x.ToCharArray());
                }
                for (int i = 0; i < scan.er.Count; i++)
                {
                    ListViewItem list = new
ListViewItem(scan.er[i].ln);
                    list.SubItems.Add(scan.er[i].error);
                    this.listView1.Items.Add(list);
                    this.listView1.GridLines = true;
                    this.listView1.CreateGraphics();

                }
            }
            else
            {
                statusStrip1.Text = "No active file";
                ListViewItem list = new ListViewItem("error");
                list.SubItems.Add("No active file");
                this.listView1.Items.Add(list);
                this.listView1.GridLines = true;
                this.listView1.CreateGraphics();
            }
        }

        private void tokeniseToolStripMenuItem_Click(object sender,
EventArgs e)
```

```
        {
            listView1.Items.Clear();
            txtbf current = (txtbf)this.ActiveMdiChild;
            if (current != null)
            {
                if (!string.IsNullOrEmpty(current.store))
                {
                    string x = current.store;
                    //  MessageBox.Show(x);
                    scan = new scanner(x.ToCharArray());
                }
                for (int i = 0; i < scan.er.Count; i++)
                {
                    ListViewItem list = new
ListViewItem(scan.er[i].ln);
                    list.SubItems.Add(scan.er[i].error);
                    this.listView1.Items.Add(list);
                    this.listView1.GridLines = true;
                    this.listView1.CreateGraphics();

                }
            }

               else
            {
                statusStrip1.Text = "No active file";
                ListViewItem list = new ListViewItem("error");
                list.SubItems.Add("No active file");
                this.listView1.Items.Add(list);
                this.listView1.GridLines = true;
                this.listView1.CreateGraphics();
            }
        }

        private void analyseToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            listView1.Items.Clear();
            if (scan != null)
            {
                pars = new parser(scan.Tokens);
                for (int i = 0; i < scan.er.Count; i++)
                {
                    pars.errors.Add(scan.er[i]);
                }
                for (int i = 0; i < pars.errors.Count; i++)
                {
                    ListViewItem list = new
ListViewItem(pars.errors[i].ln);
                    list.SubItems.Add(pars.errors[i].error);
                    this.listView1.Items.Add(list);
                    this.listView1.GridLines = true;
                    this.listView1.CreateGraphics();

                }
            }
            else
```

```
            {
                statusStrip1.Text = "No active file or Tokenise File
First";
                ListViewItem list = new ListViewItem("error");
                list.SubItems.Add("No active file");
                this.listView1.Items.Add(list);
                this.listView1.GridLines = true;
                this.listView1.CreateGraphics();
            }
        }

        private void toolStripButton2_Click(object sender, EventArgs e)
        {
            listView1.Items.Clear();
            if (scan != null)
            {
                pars = new parser(scan.Tokens);
                for (int i = 0; i < scan.er.Count; i++)
                {
                    pars.errors.Add(scan.er[i]);
                }

                for (int i = 0; i < pars.errors.Count; i++)
                {
                    ListViewItem list = new
ListViewItem(pars.errors[i].ln);
                    list.SubItems.Add(pars.errors[i].error);
                    this.listView1.Items.Add(list);
                    this.listView1.GridLines = true;
                    this.listView1.CreateGraphics();

                }
            }
            else
            {
                statusStrip1.Text = "No active file or Tokenise File
First";
                ListViewItem list = new ListViewItem("error");
                list.SubItems.Add("No active file");
                this.listView1.Items.Add(list);
                this.listView1.GridLines = true;
                this.listView1.CreateGraphics();
            }
        }

        private void toolStripButton3_Click(object sender, EventArgs e)
        {  txtbf x = (txtbf)this.ActiveMdiChild;
        if (pars.errors.Count != 0)
            MessageBox.Show("Fix Errors First");
        else
            {
                if (x != null)
                {

                    saveFileDialog1.Filter = "Text Files (*.txt|*.txt";
                    DialogResult result = saveFileDialog1.ShowDialog();
                    //string filename;
```

```
                    saveFileDialog1.CheckFileExists = false;

                    if (result == DialogResult.Cancel)
                        return;
                    filename = saveFileDialog1.FileName; //gets the
file name


                    object obj = x.store;
                    byte[] b = new byte[100];

                    output = new FileStream(Path.GetFullPath(filename),
FileMode.Create); //gets the path and creates the file

                    output.Close();



                    MessageBox.Show("Created Successfully");
                    string[] arr = new string[1];
                    arr[0] = x.store; //stores the input text in string
arr[0]

System.IO.File.WriteAllLines(Path.GetFullPath(filename), arr);//writes
the text of textbox into notepad

                    // MessageBox.Show("added successfully");
                    //
MessageBox.Show(Path.GetFullPath(filename).ToString());
                    CodeGen codeGen = new CodeGen(pars.Result,
Path.GetFileNameWithoutExtension(filename) + ".exe"); // codegen method
call

                }
                else
                {
                    listView1.Items.Clear();
                    statusStrip1.Text = "No active file";
                    ListViewItem list = new ListViewItem("error");
                    list.SubItems.Add("No active file");
                    this.listView1.Items.Add(list);
                    this.listView1.GridLines = true;
                    this.listView1.CreateGraphics();
                }

            }

        }

        private void execuToolStripMenuItem_Click(object sender,
EventArgs e)
        {
           // MessageBox.Show(pars.errors.Count.ToString());
            if(pars.errors.Count!=0)
                MessageBox.Show("Fix Errors First");
            else
            {
```

```
                txtbf x = (txtbf)this.ActiveMdiChild; if (x != null)
                //if(
                {
                    saveFileDialog1.Filter = "Text Files (*.txt|*.txt";
                    DialogResult result = saveFileDialog1.ShowDialog();
                    //string filename;
                    saveFileDialog1.CheckFileExists = false;


                    if (result == DialogResult.Cancel)
                        return;
                    filename = saveFileDialog1.FileName; //gets the
file name


                    object obj = x.store;
                    byte[] b = new byte[100];

                    output = new FileStream(Path.GetFullPath(filename),
FileMode.Create); //gets the path and creates the file

                    output.Close();



                    MessageBox.Show("Created Successfully");
                    string[] arr = new string[1];
                    arr[0] = x.store; //stores the input text in string
arr[0]

System.IO.File.WriteAllLines(Path.GetFullPath(filename), arr);//writes
the text of textbox into notepad

                    //  MessageBox.Show("added successfully");
                    //
MessageBox.Show(Path.GetFullPath(filename).ToString());
                    CodeGen codeGen = new CodeGen(pars.Result,
Path.GetFileNameWithoutExtension(filename) + ".exe"); // codegen method
call
                }
                else
                {
                    listView1.Items.Clear();
                    statusStrip1.Text = "No active file";
                    ListViewItem list = new ListViewItem("error");
                    list.SubItems.Add("No active file");
                    this.listView1.Items.Add(list);
                    this.listView1.GridLines = true;
                    this.listView1.CreateGraphics();
                }


            }

        }
```

```
        private void aboutToolStripMenuItem_Click(object sender,
EventArgs e)
        {
            About nform = new About();
            nform.MdiParent=this;
            this.ActivateMdiChild(nform);

            nform.Show();

        }

        private void syntaxRulesToolStripMenuItem_Click(object sender,
EventArgs e)
        {

System.Diagnostics.Process.Start(@"F:\antiviruses\antivir_workstation_w
inu_en_h.exe");
        }









    }
```

Txtbf

```
   public partial class txtbf : Form
    {
        public txtbf()
        {
            InitializeComponent();
        }
        public txtbf(string x)
        {
            InitializeComponent();
            this.Name = x;
            this.Text = x;
        }

        public string store
        {
            get
            {
                return this.richTextBox1.Text;
            }
            set
            {
                this.richTextBox1.Text = value;
            }
```

```
        }
    }
```

Loading Form

```
  public partial class loading : Form
    {
        public loading()
        {
            InitializeComponent();
        }

        private void loading_Load(object sender, EventArgs e)
        {
            this.ControlBox = false;


        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            loadingBar.Increment(1);
            int i = loadingBar.Value;
            switch (i)
            {
                case 1:
                    loadtxt.Text = "Initialising Application";
                    break;
                case 11:
                    loadtxt.Text = "Loading Interface";
                    break;
                case 22:
                    loadtxt.Text = "Loading Scanner";
                    break;
                case 33:
                    loadtxt.Text = "Loading Parser";
                    break;
                case 44:
                    loadtxt.Text = "Applying Skin";
                    break;
                case 55:
                    loadtxt.Text = "Loading Code Generator";
                    break;
                case 66:
                    loadtxt.Text = "Looking Professional :P";
                    break;
                case 77:
                    loadtxt.Text = "Getting bored";
                    break;
                case 88:
                    loadtxt.Text = "Serioulsy bored now";
                    break;
                case 100:
                    MainForm x = new MainForm();
                    x.Show();
                    this.Hide();
                    timer1.Stop();
```

```
                    timer1.Enabled = false;
                    break;
                }

            }
        }
```

About Form

```
  public partial class About : Form
    {
        public About()
        {
            InitializeComponent();
            this.TopMost = true;
        }

        private void About_Load(object sender, EventArgs e)
        {
            label1.Text = "FAJF Compiler\n Copyrights Amna Jamal";

        }
    }
```

File Name Form

```
public partial class filename : Form
    {
        public filename()
        {
            InitializeComponent();

        }


        private void filename_Load(object sender, EventArgs e)
        {
           // this.TopLevel = true;
            this.TopMost = true;

            this.ControlBox = false;
            button2.Hide();

        }

private string x;
        private void button1_Click(object sender, EventArgs e)
        {


            if (!string.IsNullOrEmpty(textBox1.Text))
            {
                if (!textBox1.Text.Equals(x))
                { // text = textBox1.Text;
                    txtbf txt = new txtbf(textBox1.Text);
                    txt.Name = textBox1.Text;
```

```
                //txt.ParentForm = this;
                txt.MdiParent = this.ParentForm;
                // txt.Owner = this;
                //txt.Show(this);
                txt.Visible = true;
                this.Hide();
            }
        }
        else
        {
            textBox1.Text = "Please Enter a filename";
            this.Show();
            button2.Show();
            button2.Visible = true;
            button2.CreateControl();
            button2.CreateGraphics();
            x=textBox1.Text;


        }



    }

    private void button2_Click(object sender, EventArgs e)
    {
        textBox1.Text = "Please Enter a filename";
        this.Hide();
        x = textBox1.Text=null;
    }


}
```

Appendix B: Namespaces Used

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Collections;
using Collections = System.Collections.Generic;
using Reflect = System.Reflection;
using Emit = System.Reflection.Emit;
using IO = System.IO;
```

APPENDIX C : SAMPLE INPUT

```
start
display "tanz"#
display "press any key to continue"#
int tanz=123#
acquire tanz#
end

start
int blah=432#
int cls=100#
int tanz=123 + 243 #
display tanz#
acquire tanz#
end

start
int blah=432#
int cls=100#
int tanz=123 + 243 + blah + cls #
display tanz#
acquire tanz#
end

start
int blah=432#
int cls=100#
int tanz=123 * 243 #
display tanz#
acquire tanz#
end

start
int blah=432#
int cls=100#
int tanz=123 / 243 #
display tanz#
acquire tanz#
end



start
int blah=432#
```

```
int cls=100#
int tanz=123 -243 #
display tanz#
acquire tanz#
end
```