Technical Solution

# https://i.gyazo.com/bb71250d2f8ca83c3985f30b070d8849.pngFunctionalStudio.cs [Design]

# FunctionalStudio.cs

using System;

using System.Collections.Generic;

using System.Data;

using System.Drawing;

using System.IO;

using System.Linq;

using System.Runtime.InteropServices;

using System.Threading;

using System.Windows.Forms;

using Parse;

namespace IDE

{

    public partial class FunctionalStudio : Form

    {

        private List<FileEdit> edits;

        private bool running;

        public FunctionalStudio()

        {

            //Though in practice Visual Studio discourages editing the InitializeComponent function, I did so to assign custom event handlers, e.g. NewFile and OpenFile

            InitializeComponent();

            edits = new List<FileEdit>();

        }

        private void NewFile(object sender, EventArgs e)    //Handles newToolStripMenuItem.Click and toolStripSplitButtonNew.Click

        {

            var edit = new FileEdit(tabControl1);

            tabControl1.Resize += edit.OnResize;

            //User-friendliness, auto-focuses text editor in tab when opening a new file

            tabControl1.SelectTab(edit.Tab);

            edit.TextBox.Select();

            edits.Add(edit);

            UpdateUI();

        }

        private void OpenFile(object sender, EventArgs e)   //Handles openToolStripMenuItem.Click and toolStripButtonOpen.Click

        {

            //In-built file opening dialog

            using (var dialog = new OpenFileDialog())

            {

                dialog.Title = "Open file";

                dialog.Multiselect = false;

                dialog.Filter = "Paskell files (\*.ps)|\*.ps";

                dialog.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

                if (dialog.ShowDialog() == DialogResult.OK)

                {

                    try

                    {

                        var edit = new FileEdit(tabControl1, dialog.FileName);

                        tabControl1.Resize += edit.OnResize;

                        //User-friendliness, auto-focuses text editor in tab when opening a new file

                        tabControl1.SelectTab(edit.Tab);

                        edit.TextBox.Select();

                        edits.Add(edit);

                        UpdateUI();

                    }

                    catch (FileNotFoundException)

                    {

                        //In theory this exception catch should never occur, as the dialog handles all file path issues. It's here just in case

                        MessageBox.Show("File not found.");

                    }

                }

            }

        }

        private void SaveFile(object sender, EventArgs e)   //Handles saveToolStripMenuItem.Click and toolStripButtonSave.Click

        {

            foreach (FileEdit edit in edits)

            {

                if (tabControl1.SelectedTab == edit.Tab)

                {

                    if(edit.FilePath != "")

                    {

                        edit.SaveFile();

                    }

                    else

                    {

                        SaveTabFileAs(edit);

                    }

                    break;

                }

            }

            UpdateUI();

        }

        private void SaveFileAs(object sender, EventArgs e) //Handles saveAsToolStripMenuItem.Click

        {

            foreach (FileEdit edit in edits)

            {

                if (tabControl1.SelectedTab == edit.Tab)

                {

                    SaveTabFileAs(edit);

                    break;

                }

            }

            UpdateUI();

        }

        private void SaveAll(object sender, EventArgs e)    //Handles toolStripButtonSaveAll.Click

        {

            foreach (FileEdit edit in edits)

            {

                //Equivalent to the SaveFile sequence, except for a break in the loop when SaveAs fails

                if (edit.FilePath != "")

                {

                    edit.SaveFile();

                }

                else

                {

                    if (!SaveTabFileAs(edit))

                    {

                        break;

                    }

                }

            }

        }

        //Subroutine to handle save-as file dialog, placed in a separate function as both SaveFile and SaveFileAs call the same sequence

        private bool SaveTabFileAs(FileEdit edit)

        {

            bool success = false;

            using (var dialog = new SaveFileDialog())

            {

                dialog.Title = "Save file as";

                dialog.Filter = "Paskell files (\*.ps)|\*.ps";

                dialog.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

                if (dialog.ShowDialog() == DialogResult.OK)

                {

                    try

                    {

                        edit.SaveFileAs(dialog.FileName);

                        success = true;

                    }

                    catch (DirectoryNotFoundException)

                    {

                        //Again, this exception catch should never occur, as the dialog will handle file path issues

                        MessageBox.Show("Invalid file path");

                    }

                }

            }

            return success;

        }

        private void CloseFile(object sender, EventArgs e)      //Handles closeToolStripMenuItem.Click

        {

            foreach (FileEdit edit in edits)

            {

                if (tabControl1.SelectedTab == edit.Tab)

                {

                    CloseTabFile(edit);

                    break;

                }

            }

            UpdateUI();

        }

        private void CloseAll(object sender, FormClosingEventArgs e)    //Handles FormClosing

        {

            FileEdit[] copyEdits = new FileEdit[edits.Count];

            edits.CopyTo(copyEdits);

            foreach (FileEdit edit in copyEdits)

            {

                //Attempts to close every open edit and breaks from loop if one fails

                if (!CloseTabFile(edit))

                {

                    e.Cancel = true;

                    break;

                }

            }

        }

        //Subroutine to handle close dialog, placed in separate function as both CloseFile and CloseAll, also returns success or failure to cancel CloseAll

        private bool CloseTabFile(FileEdit edit)

        {

            var result = DialogResult.No;

            if (!edit.Saved)

            {

                string displayName = edit.FilePath;

                if (displayName == "")

                {

                    displayName = "Untitled";

                }

                result = MessageBox.Show($"Save file \"{displayName}\"?", "Save file?", MessageBoxButtons.YesNoCancel, MessageBoxIcon.Question);

            }

            if (result != DialogResult.Cancel)

            {

                if (result == DialogResult.Yes)

                {

                    tabControl1.SelectTab(edit.Tab);

                    SaveFile(this, EventArgs.Empty);        //Using this rather than edit.SaveFile in case it is a new file and hasn't been saved before

                }

                tabControl1.Resize -= edit.OnResize;

                edits.Remove(edit);

                edit.Dispose();

            }

            else

            {

                return false;

            }

            return true;

        }

        private void StartProgram(object sender, EventArgs e)   //Handles startToolStripMenuItem.Click and toolStripSplitButtonStart.Click

        {

            PContext context;

            CompilerReturnState returnState;

            string sourceCode = "";

            running = true;

            UpdateUI();

            //Gets source code from active tab

            foreach (FileEdit edit in edits)

            {

                if (tabControl1.SelectedTab == edit.Tab)

                {

                    edit.TextBox.Enabled = false;

                    sourceCode = edit.TextBox.Text;

                    edit.HighlightErrors();

                    break;

                }

            }

            output.Text = "";

            returnState = Translator.Compile(sourceCode, out context);

            if (!returnState.Success)

            {

                //Any compile errors prevent execution and are printed in output

                while (returnState.Exceptions.Count > 0)

                {

                    PaskellCompileException exception = returnState.Exceptions.Dequeue();

                    output.Text += Environment.NewLine;

                    output.Text += $"{exception.ErrorMessage} on line {exception.Line + 1}, token {exception.Index + 1}";

                }

            }

            else

            {

                try

                {

                    //Similar pattern to that within compiler when defining expressions, ensure that there is one and only one "main" definition

                    PExpression[] results = context.Expressions.Where(x => x.Identifier == "main").ToArray();

                    if (results.Length != 1)

                    {

                        throw new PaskellRuntimeException("No unique definition for main", null);

                    }

                    else

                    {

                        try

                        {

                            PExpression main = results[0];

                            PExpression result = main.Evaluate();       //Run the code

                            output.Text += result.Value;

                        }

                        catch

                        {

                            throw new PaskellRuntimeException("Failure evaluating main: was there a baseless recursion function?", null);

                        }

                    }

                }

                catch (PaskellRuntimeException f)

                {

                    output.Text += $"{f.ErrorMessage}";

                    if (f.PExpression != null)

                    {

                        output.Text += $" in expression {f.PExpression.Identifier}";

                    }

                }

            }

            running = false;

            UpdateUI();

            foreach (FileEdit edit in edits)

            {

                if (tabControl1.SelectedTab == edit.Tab)

                {

                    edit.TextBox.Enabled = true;

                    break;

                }

            }

        }

        private void CancelChangeTab(object sender, TabControlCancelEventArgs e)        //Handles tabControl1.Selecting

        {

            if (running)

            {

                e.Cancel = true;

            }

        }

        //This function should be called when at any point the state of the form changes such that the toolbar and menus need to have certain items enabled/disabled

        //Enabled/disable save and save as buttons depending on if files are open

        private void UpdateUI()

        {

            if (edits.Count == 0)

            {

                toolStripButtonNew.Enabled = true;

                toolStripButtonOpen.Enabled = true;

                newToolStripMenuItem.Enabled = true;

                openToolStripMenuItem.Enabled = true;

                toolStripButtonSave.Enabled = false;

                toolStripButtonSaveAll.Enabled = false;

                saveToolStripMenuItem.Enabled = false;

                saveAsToolStripMenuItem.Enabled = false;

                saveAllToolStripMenuItem.Enabled = false;

                toolStripSplitButtonStart.Enabled = false;

                closeToolStripMenuItem.Enabled = false;

                startToolStripMenuItem.Enabled = false;

            }

            else if (!running)

            {

                toolStripButtonNew.Enabled = true;

                toolStripButtonOpen.Enabled = true;

                newToolStripMenuItem.Enabled = true;

                openToolStripMenuItem.Enabled = true;

                toolStripButtonSave.Enabled = true;

                toolStripButtonSaveAll.Enabled = true;

                saveToolStripMenuItem.Enabled = true;

                saveAsToolStripMenuItem.Enabled = true;

                saveAllToolStripMenuItem.Enabled = true;

                toolStripSplitButtonStart.Enabled = true;

                closeToolStripMenuItem.Enabled = true;

                startToolStripMenuItem.Enabled = true;

            }

            else

            {

                toolStripButtonNew.Enabled = false;

                toolStripButtonOpen.Enabled = false;

                newToolStripMenuItem.Enabled = false;

                openToolStripMenuItem.Enabled = false;

                toolStripButtonSave.Enabled = false;

                toolStripButtonSaveAll.Enabled = false;

                saveToolStripMenuItem.Enabled = false;

                saveAsToolStripMenuItem.Enabled = false;

                saveAllToolStripMenuItem.Enabled = false;

                toolStripSplitButtonStart.Enabled = false;

                closeToolStripMenuItem.Enabled = false;

                startToolStripMenuItem.Enabled = false;

            }

        }

        //Classifies file-editing tab object, including handling nuance of unsaved files (titled 'Untitled'), saving and saving as, changing file path and tab title

        //when necessary i.e. once saved as a new file, title needs to change accordingly.

        private class FileEdit : IDisposable

        {

            public TabPage Tab { get; }

            public EditorTextBox TextBox { get; }

            public bool Saved { get; private set; }

            public string FilePath { get; private set; }

            private EventHandler tabResize;

            public FileEdit(TabControl tabControl)

            {

                Tab = new TabPage("Untitled");

                TextBox = new EditorTextBox { Font = new Font("Consolas", 9) };

                Saved = true;

                FilePath = "";

                Tab.Controls.Add(TextBox);

                TextBox.Dock = DockStyle.Fill;  //Necessary for textbox to scale with the window properly when resizing

                TextBox.TextChanged += TextChanged;

                tabControl.TabPages.Add(Tab);

            }

            //This overload is used when creating a new FileEdit instance but for a file which is being opened

            public FileEdit(TabControl tabControl, string filePath)

            {

                string text;

                try

                {

                    text = File.ReadAllText(filePath);

                }

                catch (FileNotFoundException e)

                {

                    Console.WriteLine("File path not found");

                    throw e;

                }

                Tab = new TabPage(Path.GetFileName(filePath));

                TextBox = new EditorTextBox {Text = text, Font = new Font("Consolas", 9)};

                Saved = true;

                FilePath = filePath;

                Tab.Controls.Add(TextBox);

                TextBox.Dock = DockStyle.Fill;

                TextBox.TextChanged += TextChanged;

                tabControl.Resize += OnResize;

                tabControl.TabPages.Add(Tab);

                TextBox.UpdateLineNumbers();//----->//Necessary for the reason that the size of the EditorTextBox control changes after it is instantiated and updated

            }                                       //therefore it scales the internal components incorrectly, so needs to be reupdated otherwise it remains incorrectly

                                                    //scaled until the text is changed

            public void SaveFile()

            {

                if (!Saved)

                {

                    try

                    {

                        File.WriteAllText(FilePath, TextBox.Text);

                        Saved = true;

                        Tab.Text = Path.GetFileName(FilePath);

                    }

                    catch (DirectoryNotFoundException e)

                    {

                        Console.WriteLine("Invalid file path");

                        throw e;

                    }

                }

            }

            public void SaveFileAs(string filePath)

            {

                FilePath = filePath;

                Saved = false;

                SaveFile();

            }

            //Allows for safe removal of these objects when tabs in the control are closed

            public void Dispose()

            {

                TextBox.Dispose();

                Tab.Dispose();

            }

            private void TextChanged(object sender, EventArgs e)

            {

                //UI feature: asterisk indicates changes aren't saved

                if (Saved)

                {

                    Saved = false;

                    Tab.Text += '\*';

                }

                //tokenSource.Cancel();

                //tokenSource = new CancellationTokenSource();

                //try

                //{

                //    await ExecuteAfterTime(Parse, 1000);

                //}

                //catch { }

            }

            public void OnResize(object sender, EventArgs e)

            {

                TextBox.UpdateLineNumbers();        //Called for the same reason as it is called in the constructor for opening a file

            }

            //Primary attempt of text highlighting using the EM\_SETCHARFORMAT message as part of the windows controls rich edit tools

            public void HighlightErrors()

            {

                IntPtr empty = IntPtr.Zero;

                CHARFORMAT format;

                SendMessage(TextBox.TextHandle, WM\_SETREDRAW, IntPtr.Zero, ref empty);      //Disables drawing of the textbox so the user doesn't see the text selection process

                int selectionStart = TextBox.SelectionStart;

                int selectionLength = TextBox.SelectionLength;

                TextBox.TextChanged -= TextChanged;         //This prevents all the app processing these changes as the file being edited

                TextBox.SelectAll();

                format = new CHARFORMAT();

                format.cbSize = Marshal.SizeOf(format);

                format.dwMask = CFM\_UNDERLINETYPE;

                format.bUnderlineType = 0;

                SendMessage(TextBox.TextHandle, EM\_SETCHARFORMAT, (IntPtr) SCF\_SELECTION, ref format);      //Removes all underlining of text

                Queue<TokeniserReturnError> Errors = Translator.GetTokeniserErrors(TextBox.Text);

                while (Errors.Count > 0)

                {

                    TokeniserReturnError error = Errors.Dequeue();

                    TextBox.SelectionStart = error.Index;

                    int i = 0;

                    //This while loops runs to the end of a word (or the end of the file) to select it to underline

                    while (TextBox.SelectionStart + i < TextBox.Text.Length && !string.IsNullOrWhiteSpace(TextBox.Text[TextBox.SelectionStart + i].ToString())) i++;

                    TextBox.SelectionLength = i;

                    format = new CHARFORMAT();

                    format.cbSize = Marshal.SizeOf(format);

                    format.dwMask = CFM\_UNDERLINETYPE;

                    format.bUnderlineType = WaveUnderlineStyle | RedUnderlineColour;

                    SendMessage(TextBox.TextHandle, EM\_SETCHARFORMAT, (IntPtr) SCF\_SELECTION, ref format);  //Underlines selected text with red wavy underline

                }

                //Putting everything back to normal again

                TextBox.SelectionStart = selectionStart;

                TextBox.SelectionLength = selectionLength;

                TextBox.TextChanged += TextChanged;

                SendMessage(TextBox.TextHandle, WM\_SETREDRAW, (IntPtr) 1, ref empty);

                TextBox.TextUpdate();

            }

            //private async Task ExecuteAfterTime(Action action, int timeoutInMilliseconds)

            //{

            //    await Task.Delay(timeoutInMilliseconds, tokenSource.Token);

            //    action();

            //}

        }

        //These are all the values used in the HighlightErrors sub when using SendMessage to manipulate the textbox

        private const uint CFM\_UNDERLINETYPE = 0x800000;

        private const int SCF\_SELECTION = 1;

        private const int EM\_SETCHARFORMAT = 0x0444;

        private const int WM\_SETREDRAW = 0x000b;

        private const byte WaveUnderlineStyle = 8;

        private const byte RedUnderlineColour = 0x50;

        //http://geekswithblogs.net/pvidler/archive/2003/10/15/188.aspx

        //The following struct is the structure used to send information about the rich text editing to the textbox

        //We only use dwMask and bUnderlineType i.e. set mask to only consider underline information, then provide that underline information

        //cbSize is a perculiar necessity of implementing C++ library functions in C# with variable sized structures,

        //where the size of an instantiated structure, in this case CHARFORMAT, needs to be given back to itself using Marshal.SizeOf()

        [StructLayout(LayoutKind.Sequential)]

        private struct CHARFORMAT

        {

            public int cbSize;

            public uint dwMask;

            public uint dwEffects;

            public int yHeight;

            public int yOffset;

            public int crTextColor;

            public byte bCharSet;

            public byte bPitchAndFamily;

            [MarshalAs(UnmanagedType.ByValArray, SizeConst = 32)]

            public char[] szFaceName;

            // CHARFORMAT2 from here onwards.

            public short wWeight;

            public short sSpacing;

            public int crBackColor;

            public int LCID;

            public uint dwReserved;

            public short sStyle;

            public short wKerning;

            public byte bUnderlineType;

            public byte bAnimation;

            public byte bRevAuthor;

        }

        [DllImport("User32.dll")]

        private static extern int SendMessage(IntPtr handle, int message, IntPtr wParam, ref CHARFORMAT lParam);

        [DllImport("User32.dll")]

        private static extern int SendMessage(IntPtr handle, int message, IntPtr wParam, ref IntPtr lParam);        //Importing it again for the disable drawing purpose

    }

}

# https://i.gyazo.com/068345e11cc1ca3c90c966e6611fb8e9.pngEditorTextBox.cs [Design]

# EditorTextBox.cs

using System;

using System.Drawing;

using System.Linq;

using System.Windows.Forms;

using System.Reflection;

namespace IDE

{

    public partial class EditorTextBox : UserControl

    {

        //Most of these overloads are to make interaction with the actual textBox component of this control as close as possible to a real textbox

        //They also interact with the lineNumbers component in places to make properties consistent between the two controls

        public override Font Font

        {

            get => textBox.Font;

            set

            {

                textBox.Font = value;

                lineNumbers.Font = value;

            }

        }

        public new string Text

        {

            get => textBox.Text;

            set => textBox.Text = value;

        }

        //This is used in particular for when code is running, where the editor needs to be disabled

        //I prefered instead changing the ReadOnly property of the textbox rather than Enabled, as the colour is nicer,

        //and text can still be selected without allowing changes.

        public new bool Enabled

        {

            get => !textBox.ReadOnly;

            set => textBox.ReadOnly = !value;

        }

        public int SelectionStart

        {

            get => textBox.SelectionStart;

            set => textBox.SelectionStart = value;

        }

        public int SelectionLength

        {

            get => textBox.SelectionLength;

            set => textBox.SelectionLength = value;

        }

        public Font SelectionFont

        {

            get => textBox.SelectionFont;

            set => textBox.SelectionFont = value;

        }

        public void SelectAll() => textBox.SelectAll();

        public void TextUpdate() => textBox.Update();

        public IntPtr TextHandle => textBox.Handle;

        public new event EventHandler TextChanged

        {

            add => textBox.TextChanged += value;

            remove => textBox.TextChanged -= value;

        }

        public new event MouseEventHandler MouseWheel

        {

            add

            {

                textBox.MouseWheel += value;

                lineNumbers.MouseWheel += value;

            }

            remove

            {

                textBox.MouseWheel -= value;

                lineNumbers.MouseWheel -= value;

            }

        }

        public new event KeyEventHandler KeyDown

        {

            add => textBox.KeyDown += value;

            remove => textBox.KeyDown -= value;

        }

        //End of overloads

        private int ScrollMax { get => vScrollBar.Maximum - vScrollBar.Minimum - vScrollBar.LargeChange + 1; }

        private int ScrollMin { get => vScrollBar.Minimum; }

        public EditorTextBox()

        {

            InitializeComponent();

            TextChanged += OnTextChanged;

            MouseWheel += OnMouseWheel;

            KeyDown += OnKeyDown;

            vScrollBar.Scroll += (sender, e) => { ScrollTextBox(); };

            UpdateLineNumbers();

        }

        private void OnTextChanged(object sender, EventArgs e)

        {

            UpdateLineNumbers();

            ScrollToLine(textBox.GetLineFromCharIndex(SelectionStart));

        }

        //Effectively passes on mousewheel events from the textbox to the scrollbar

        private void OnMouseWheel(object sender, MouseEventArgs e)

        {

            if (vScrollBar.Enabled)

            {

                MethodInfo methodInfo = typeof(VScrollBar).GetMethod("OnMouseWheel", BindingFlags.NonPublic | BindingFlags.Instance);

                methodInfo.Invoke(vScrollBar, new object[] { e });

            }

        }

        //Makes pageup and pagedown scroll, and forces textbox to update when caret is moved using arrow keys

        private void OnKeyDown(object sender, KeyEventArgs e)

        {

            if (e.KeyCode == Keys.PageUp)

            {

                vScrollBar.Value = Math.Max(ScrollMin, vScrollBar.Value - vScrollBar.LargeChange);

                ScrollTextBox();

                e.Handled = true;

            }

            else if (e.KeyCode == Keys.PageDown)

            {

                vScrollBar.Value = Math.Min(ScrollMax, vScrollBar.Value + vScrollBar.LargeChange);

                ScrollTextBox();

                e.Handled = true;

            }

            else if (e.KeyCode == Keys.Up || e.KeyCode == Keys.Down || e.KeyCode == Keys.Left || e.KeyCode == Keys.Right)

            {

                int line;

                switch (e.KeyCode)

                {

                    case Keys.Up:

                        line = textBox.GetLineFromCharIndex(SelectionStart) - 1;

                        break;

                    case Keys.Down:

                        line = textBox.GetLineFromCharIndex(SelectionStart) + 1;

                        break;

                    case Keys.Left:

                        if (textBox.SelectionStart > 0)

                        {

                            line = textBox.GetLineFromCharIndex(SelectionStart - 1);

                        }

                        else

                        {

                            line = textBox.GetLineFromCharIndex(SelectionStart);

                        }

                        break;

                    case Keys.Right:

                        if (textBox.SelectionStart < textBox.TextLength - 1)

                        {

                            line = textBox.GetLineFromCharIndex(SelectionStart + 1);

                        }

                        else

                        {

                            line = textBox.GetLineFromCharIndex(SelectionStart);

                        }

                        break;

                    default:    //Never reached

                        line = textBox.GetLineFromCharIndex(SelectionStart);

                        break;

                }

                ScrollToLine(Math.Min(ScrollMax, Math.Max(ScrollMin, line)));

            }

        }

        private void ScrollTextBox()

        {

            container.Location = new Point(0, -vScrollBar.Value \* (container.Height - Height) / ScrollMax);

        }

        private void ScrollToLine(int line)

        {

            //These just refocus the editor window by scrolling up/down to where the caret is

            //The distinction between up or down is so it only scrolls as far as it has to, which depends on direction, or if not at all

            if (line - vScrollBar.Value >= Height / Font.Height)

            {

                vScrollBar.Value = line - Height / Font.Height;

                ScrollTextBox();

            }

            else if (line - vScrollBar.Value <= 0)

            {

                vScrollBar.Value = line;

                ScrollTextBox();

            }

        }

        public void UpdateLineNumbers()

        {

            string newNumbers = "";

            for (int i = 0; i < textBox.Lines.Length; i++)

            {

                newNumbers += $"{i + 1}\n";

            }

            if (Text.Length == 0 || Text.Last() != '\n')

            {

                newNumbers += $"{textBox.Lines.Length + 1}";

            }

            lineNumbers.Text = newNumbers;

            container.Height = textBox.GetPositionFromCharIndex(Text.Length).Y + Height;

            if (textBox.Lines.Length <= 1)

            {

                vScrollBar.Enabled = false;

            }

            else

            {

                //Value uses line count minus two, because we only want to be able to scroll past all but one of the lines (so the last line cannot be scrolled past)

                //Therefore that makes the theoretical maximum the line count minus 1. However, the actual maximum scrollable value is Maximum - LargeChange + 1,

                //so Maximum must be set to actual maximum plus LargeChange minus one, therefore the line count minus 2 plus LargeChange, set afterwards

                vScrollBar.Maximum = textBox.Lines.Length - 2 + Height / Font.Height;

                vScrollBar.SmallChange = Math.Min(3, textBox.Lines.Length);

                vScrollBar.LargeChange = Height / Font.Height;

                vScrollBar.Enabled = true;

            }

        }

    }

}

# Tools.cs

using System;

using System.Collections.Generic;

namespace Utility

{

    public static class Tools

    {

        //I wanted to try building a merge sort algorithm that didn't use recursion, which was an interesting concept as it required handling memory

        //and using stacks appropriately without using a call stack.

        public static T[] MergeSort<T>(T[] sortingArray, bool descendingOrder = false) where T : IComparable

        {

            //Stack of tuple arrays, tuple describes the index and the length of a split segment from the array

            //As the stack gets pushed onto, more and more divisions are made, so more and more segments are made

            //The stack is used to 'remember' the algorithm's splitting and segments to be able to correctly merge back on the way up

            Stack<(int, int)[]> divisions = new Stack<(int, int)[]>();

            divisions.Push(new [] { (0, sortingArray.Length) });                                    //Not actually necessary, just demonstrates the complete array as one segment

            bool dividing = true;

            while (dividing)

            {

                dividing = false;                                                                   //Remains false if no divisions are made and loop is exited

                (int, int)[] currentDivides = divisions.Peek();

                List<(int, int)> newDivides = new List<(int, int)>();                               //Using a list instead of an array as it is dynamic, and we

                for (int i = 0; i < currentDivides.Length; i++)                                     //don't know how many divisions and segments will be made

                {

                    int startIndex = currentDivides[i].Item1;

                    int length = currentDivides[i].Item2;

                    if (length > 1)

                    {

                        dividing = true;

                        newDivides.Add((startIndex, length / 2));                                   //Halves length, rounded down (C-style integer arithmetic)

                        newDivides.Add((startIndex + length / 2, length / 2 + length % 2));         //Deliberately rounds up with mod so as to make up for round down from above

                    }

                }

                if (dividing)

                {

                    divisions.Push(newDivides.ToArray());                                       //Once divided, pushes the new state to the stack to be used when merging later

                }

            }

            while (divisions.Count > 1)

            {

                (int, int)[] divides = divisions.Pop();

                for (int i = 0; i < divides.Length; i += 2)

                {

                    int index1 = divides[i].Item1;                  //Splitting up the values from the tuples indication segment index and length to make code easier on the eyes

                    int index2 = divides[i + 1].Item1;

                    int length1 = divides[i].Item2;

                    int length2 = divides[i + 1].Item2;

                    T[] tempArray = new T[length1 + length2];

                    int j, k;                                       //Effectively j + k is the indexer for tempArray, but only one of either j or k will get incremented in each interation

                    for (j = 0, k = 0; j < length1 && k < length2; /\*Increments handled in loop\*/)

                    {

                        if (sortingArray[index1 + j].CompareTo(sortingArray[index2 + k]) < 0 ^ descendingOrder)     //XOR (^) with descending order just flips the comparison

                        {

                            tempArray[j + k] = sortingArray[index1 + j];

                            j++;

                        }

                        else

                        {

                            tempArray[j + k] = sortingArray[index2 + k];

                            k++;

                        }

                    }

                    //There will often be items left in one array if the other array finished copying first, this just transfers the remainder

                    //j + k (tempArray indexer) is less than the capacity of tempArray if the copying isn't finished

                    if (j + k < length1 + length2)

                    {

                        if (j < k)

                        {

                            Array.Copy(sortingArray, index1 + j, tempArray, j + k, length1 - j);

                        }

                        else

                        {

                            Array.Copy(sortingArray, index2 + k, tempArray, j + k, length2 - k);

                        }

                    }

                    Array.Copy(tempArray, 0, sortingArray, index1, length1 + length2);

                }

            }

            return sortingArray;

        }

        //Merge sort as an extension method

        public static void Sort<T>(this T[] sortingArray, bool descendingOrder = false) where T : IComparable => sortingArray = MergeSort(sortingArray, descendingOrder);

        //This is a small tool to help with array manipulation, for if you want to populate a chunk of an array with just one value

        public static void Populate<T>(this T[] array, T value, int startIndex, int length)

        {

            for (int i = 0; i < length; i++)

            {

                array[startIndex + i] = value;

            }

        }

    }

}

# Translator.cs

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Linq;

using System.Text.RegularExpressions;

using System.Reflection;

using Utility;

 namespace Parse

{

    public static class Translator

    {

        //Just retrieves tokeniser errors

        public static Queue<TokeniserReturnError> GetTokeniserErrors(string sourceCode)

        {

            return Tokenise(sourceCode, out \_).Errors;

        }

        //Public wrapper of tokeniser and compiler together, with singular compiler error indicating tokeniser error

        public static CompilerReturnState Compile(string sourceCode, out PContext context)

        {

            CompilerReturnState returnState;

            Token[] tokenCode;

            context = null;

            if (!Tokenise(sourceCode, out tokenCode).Success)

            {

                returnState = new CompilerReturnState(false);

                returnState.Exceptions.Enqueue(new PaskellCompileException("Couldn't parse tokens", 0));

            }

            else

            {

                returnState = Compile(tokenCode, out context);

            }

            return returnState;

        }

        //Converts source code into tokens of consistent type that make compiling easier

        private static TokeniserReturnState Tokenise(string sourceCode, out Token[] TokenCode)

        {

            List<Token> tokenCollection = new List<Token>();

            bool[] codeMatched = new bool[sourceCode.Length];   //Used for finding errors i.e. anything that wasn't parsed, so didn't match with any token Regex

            TokeniserReturnState returnState;

            MatchCollection matches;

            codeMatched.Populate(false, 0, codeMatched.Length);

            //Loops through every token for their regex patterns, as given by their RegexPattern custom attribute

            foreach (TokenType tokenType in Enum.GetValues(typeof(TokenType)))

            {

                matches = new Regex(tokenType.GetPattern()).Matches(sourceCode);

                foreach (Match match in matches)

                {

                    tokenCollection.Add(new Token(match.Value, tokenType, match.Index));

                    codeMatched.Populate(true, match.Index, match.Length);      //Token parsed, so the code it matched with must be valid

                }

            }

            //Finding whitespace just to fill codeMatched, so that it doesn't detect spaces or backslashes before newline as syntax errors

            matches = new Regex(@"\s|**\\**(?=\n)").Matches(sourceCode);

            foreach (Match match in matches)

            {

                codeMatched.Populate(true, match.Index, match.Length);

            }

            //Main useful output, the tokenised code

            TokenCode = tokenCollection.ToArray();

            TokenCode.Sort();

            //Error output, indicates success or partial failure and errors if there are failures

            if (codeMatched.Any(element => element == false))

            {

                returnState = new TokeniserReturnState(false);

                //Loops through to find each occurence of an error

                for (int i = 0; i < codeMatched.Length; /\*Increments handled in loop\*/)

                {

                    if (codeMatched[i] == false)

                    {

                        returnState.Errors.Enqueue(new TokeniserReturnError(i));

                        while (i < codeMatched.Length && codeMatched[i] == false) i++;

                    }

                    else i++;

                }

            }

            else

            {

                //True in return state indicates success i.e. no errors

                returnState = new TokeniserReturnState(true);

            }

            return returnState;

        }

        private static CompilerReturnState Compile(Token[] tokenCode, out PContext Context)

        {

            CompilerReturnState returnState;

            Queue<PaskellCompileException> exceptions = new Queue<PaskellCompileException>();

            int deletedLines = 0;

            List<(Token[], (int, int))> lines = new List<(Token[] line, (int, int))>();

            List<PExpression> Expressions = new List<PExpression>();

            //All the base expressions (included pseudocode to show equivalent C# code):

            //An underscore \_ represents a generic type i.e. could be any type valid for the function

            /\*Arithmetic\*/

            Expressions.Add(new PExpression(Add,

                new TypeSignature(new TypeSignature(), new TypeSignature(new TypeSignature(), new TypeSignature())), "Add"));       //\_ -> \_ -> \_ Add a b = a + b

            Expressions.Add(new PExpression(Subtract,

                new TypeSignature(new TypeSignature(), new TypeSignature(new TypeSignature(), new TypeSignature())), "Subtract"));  //\_ -> \_ -> \_ Subtract a b = a - b

            Expressions.Add(new PExpression(Multiply,

                new TypeSignature(new TypeSignature(), new TypeSignature(new TypeSignature(), new TypeSignature())), "Multiply"));  //\_ -> \_ -> \_ Multiply a b = a \* b

            Expressions.Add(new PExpression(Divide,

                new TypeSignature(new TypeSignature(), new TypeSignature(new TypeSignature(), new TypeSignature())), "Divide"));    //\_ -> \_ -> \_ Divide a b = a / b

            /\*Boolean logic\*/

            /\* bool -> bool Not a = !a \*/

            Expressions.Add(new PExpression(Not,

                new TypeSignature(new TypeSignature(typeof(bool)), new TypeSignature(typeof(bool))), "Not"));

            /\* bool -> bool -> bool And a b = a && b \*/

            Expressions.Add(new PExpression(And,

                new TypeSignature(new TypeSignature(typeof(bool)), new TypeSignature(new TypeSignature(typeof(bool)), new TypeSignature(typeof(bool)))), "And"));

            /\* bool -> bool -> bool Or a b = a || b \*/

            Expressions.Add(new PExpression(Or,

                new TypeSignature(new TypeSignature(typeof(bool)), new TypeSignature(new TypeSignature(typeof(bool)), new TypeSignature(typeof(bool)))), "Or"));

            /\* bool -> bool -> bool Xor a b = a ^ b \*/

            Expressions.Add(new PExpression(Xor,

                new TypeSignature(new TypeSignature(typeof(bool)), new TypeSignature(new TypeSignature(typeof(bool)), new TypeSignature(typeof(bool)))), "Xor"));

            /\*Comparison and inequalities\*/

            Expressions.Add(new PExpression(EqualTo,

                new TypeSignature(new TypeSignature(), new TypeSignature(new TypeSignature(), new TypeSignature(typeof(bool)))), "EqualTo"));    //\_ -> \_ -> bool EqualTo a b = a == b

            Expressions.Add(new PExpression(GreaterThan,

                new TypeSignature(new TypeSignature(), new TypeSignature(new TypeSignature(), new TypeSignature(typeof(bool)))), "GreaterThan"));//\_ -> \_ -> bool GreaterThan a b = a > b

            Expressions.Add(new PExpression(LessThan,

                new TypeSignature(new TypeSignature(), new TypeSignature(new TypeSignature(), new TypeSignature(typeof(bool)))), "LessThan"));   //\_ -> \_ -> bool LessThan a b = a < b

            int startIndex = 0;

            int endIndex;

            //Divided code into lines

            Token[] copyLine;

            for (int i = 0; i < tokenCode.Length; i++)

            {

                if (tokenCode[i].TokenType == TokenType.LineBreak)

                {

                    endIndex = i;

                    copyLine = new Token[endIndex - startIndex];

                    Array.Copy(tokenCode, startIndex, copyLine, 0, copyLine.Length);

                    lines.Add((copyLine, (0, 0)));

                    startIndex = endIndex + 1;

                }

            }

            endIndex = tokenCode.Length;

            copyLine = new Token[endIndex - startIndex];

            Array.Copy(tokenCode, startIndex, copyLine, 0, copyLine.Length);

            if (copyLine.Length > 1)

            {

                lines.Add((copyLine, (0, 0)));

            }

            //Instantiate expressions

            for (int i = 0; i < lines.Count; i++)

            {

                try

                {

                    (Token[] line, (int expressionSignatureIndex, int equateIndex) divisions) line = lines[i];

                    {

                        bool wasLastWord = false;

                        //Finding points to split at for type signature, expression signature, and expression definition

                        //For start of expression signature, there must be two consecutive words i.e. final type in type signature, then expression identifier

                        //For the start of the definition, an equals sign

                        for (int j = 0; j < line.line.Length; j++)

                        {

                            if (line.divisions.expressionSignatureIndex == 0)

                            {

                                if (line.line[j].TokenType == TokenType.Word)

                                {

                                    if (wasLastWord)

                                    {

                                        line.divisions.expressionSignatureIndex = j;

                                    }

                                    else

                                    {

                                        wasLastWord = true;

                                    }

                                }

                                else if (line.line[j].TokenType != TokenType.Bracket)

                                {

                                    wasLastWord = false;

                                }

                            }

                            else if (line.divisions.equateIndex == 0)

                            {

                                if (line.line[j].TokenType == TokenType.Equate)

                                {

                                    line.divisions.equateIndex = j;

                                    break;

                                }

                            }

                        }

                        if (line.divisions.expressionSignatureIndex == 0)

                        {

                            throw new PaskellCompileException("No type signature for expression given", 0, i);

                        }

                        if (line.divisions.equateIndex == 0)

                        {

                            throw new PaskellCompileException("Expression not defined", line.divisions.expressionSignatureIndex, i);

                        }

                        //Getting type signature and expression signature

                        //Takes subline of just type signature

                        Token[] subLine = new Token[line.divisions.expressionSignatureIndex];

                        Array.Copy(line.line, 0, subLine, 0, subLine.Length);

                        TypeSignature typeSignature;

                        try

                        {

                            typeSignature = ConstructTypeSignature(subLine);

                        }

                        catch (PaskellCompileException e)

                        {

                            throw new PaskellCompileException(e.ErrorMessage, e.Index, i);

                        }

                        //Takes just the first token in the expression signature and uses its identifier

                        subLine = new Token[line.divisions.equateIndex - line.divisions.expressionSignatureIndex];

                        Array.Copy(line.line, line.divisions.expressionSignatureIndex, subLine, 0, subLine.Length);

                        if (subLine.Length == 0 || subLine[0].TokenType != TokenType.Word)

                        {

                            throw new PaskellCompileException("Expected expression identifier", line.divisions.expressionSignatureIndex, i);

                        }

                        PExpression pExpression = new PExpression(typeSignature, subLine[0].Code);

                        Expressions.Add(pExpression);

                        lines[i] = line;

                    }

                }

                catch (PaskellCompileException e)

                {

                    //Adding to deletedLines compensates for how when errors are caught, lines get deleted

                    //It is important to still track which line the code is on

                    exceptions.Enqueue(new PaskellCompileException(e.ErrorMessage, e.Index, e.Line + deletedLines));

                    lines.Remove(lines[i]);

                    deletedLines++;

                    i--;

                }

            }

            //Create definitions

            for (int i = 0; i < lines.Count; i++)

            {

                try

                {

                    (Token[] line, (int expressionSignatureIndex, int equateIndex) divisions) line = lines[i];

                    List<PExpression> parameters = new List<PExpression>();

                    PExpression expression;

                    Token[] subLine = new Token[line.divisions.equateIndex - line.divisions.expressionSignatureIndex];

                    Array.Copy(line.line, line.divisions.expressionSignatureIndex, subLine, 0, subLine.Length);

                    //Matching line to expression previously instantiated

                    if (subLine[0].TokenType != TokenType.Word) //This should never be reached as this is handled in the for loop above, at expression instantiation

                    {

                        throw new PaskellCompileException("Expected expression identifier", line.divisions.expressionSignatureIndex, i);

                    }

                    else

                    {

                        PExpression[] results = Expressions.Where(x => x.Identifier == subLine[0].Code).ToArray();

                        if (results.Length != 1)

                        {

                            throw new PaskellCompileException($"No unique definition for expression {subLine[0].Code}", line.divisions.expressionSignatureIndex, i);

                        }

                        else

                        {

                            expression = results[0];

                        }

                    }

                    //Setting up parameters of expression

                    int parameterCount = 0;

                    for (int j = 1; j < subLine.Length; j++)

                    {

                        if (subLine[j].TokenType != TokenType.Word)

                        {

                            throw new PaskellCompileException("Expected function parameter", line.divisions.expressionSignatureIndex + j, i);

                        }

                        else

                        {

                            if (parameterCount < expression.TypeSignature.ArgumentCount)

                            {

                                parameters.Add(new PExpression(parameterCount, expression.TypeSignature[parameterCount].Parameter, subLine[j].Code));

                                parameterCount++;

                            }

                            else

                            {

                                throw new PaskellCompileException("Unexpected token", line.divisions.expressionSignatureIndex + j, i);

                            }

                        }

                    }

                    if (parameterCount < expression.TypeSignature.ArgumentCount)

                    {

                        throw new PaskellCompileException($"Too few parameters for function {expression.Identifier}", line.divisions.equateIndex - 1, i);

                    }

                    subLine = new Token[line.line.Length - line.divisions.equateIndex - 1];

                    Array.Copy(line.line, line.divisions.equateIndex + 1, subLine, 0, subLine.Length);

                    //Defining expression

                    try

                    {

                        //Here the expression available to be defined are the base expression first added, the other expressions in the file, and finally this expressions parameters

                        //The parameters are local to the expression and are not accessible from other expressions

                        //This also allows for parameters of different expressions to have the same name

                        PushSubExpressions(subLine, Expressions.Concat(parameters).ToList(), expression.TypeSignature.FinalType, expression, null, true);

                    }

                    catch (PaskellCompileException e)

                    {

                        throw new PaskellCompileException(e.ErrorMessage, e.Index + line.divisions.equateIndex + 1, i);

                    }

                }

                catch (PaskellCompileException e)

                {

                    exceptions.Enqueue(new PaskellCompileException(e.ErrorMessage, e.Index, e.Line + deletedLines));

                }

            }

            Context =  new PContext(Expressions.ToArray());

            if (exceptions.Count > 0)

            {

                returnState = new CompilerReturnState(false);

                while (exceptions.Count > 0)

                {

                    returnState.Exceptions.Enqueue(exceptions.Dequeue());

                }

            }

            else

            {

                returnState = new CompilerReturnState(true);

            }

            return returnState;

        }

        //Used to parse a type signature

        private static TypeSignature ConstructTypeSignature(Token[] tokenCode)

        {

            int bracketNesting = 0;

            int bracketStartIndex = 0;

            int bracketEndIndex = 0;

            int functionMapIndex = 0;

            TypeSignature typeSignature = null;

            TypeSignature left = null;

            bool isFunction = false;

            if (tokenCode.Length == 0)

            {

                throw new PaskellCompileException("Expected type signature expression", 0);

            }

            for (int i = 0; i < tokenCode.Length; i++)

            {

                Token token = tokenCode[i];

                //When there are bracket clauses, this function gets recursed and passed the contents of the bracket clause

                if (token.TokenType == TokenType.Bracket)

                {

                    if (token.Code == "(")

                    {

                        if (bracketNesting == 0)

                        {

                            bracketStartIndex = i + 1;

                        }

                        bracketNesting++;

                    }

                    else

                    {

                        if (bracketNesting == 0)

                        {

                            throw new PaskellCompileException("Unexpected bracket", i);

                        }

                        else

                        {

                            bracketEndIndex = i;

                            bracketNesting--;

                        }

                    }

                }

                else if (bracketNesting == 0)

                {

                    //This is where when a function map is found, the type signature is considered a function and the left and right side must be constructed as type signatures

                    //This uses the same technique as with bracket nesting

                    if (token.TokenType == TokenType.FunctionMap)

                    {

                        isFunction = true;

                        functionMapIndex = i;

                        if (bracketEndIndex == 0)

                        {

                            bracketEndIndex = i;

                        }

                        Token[] newTokenCode = new Token[bracketEndIndex - bracketStartIndex];

                        Array.Copy(tokenCode, bracketStartIndex, newTokenCode, 0, newTokenCode.Length);

                        try

                        {

                            left = ConstructTypeSignature(newTokenCode);

                        }

                        catch (PaskellCompileException e)

                        {

                            throw new PaskellCompileException(e.ErrorMessage, e.Index + bracketStartIndex);

                        }

                        break;

                    }

                }

            }

            //Once totally looped through, the type signature is either a function signature, a type, or is surrounded by brackets so must be recursed through with their contents

            if (bracketNesting > 0)

            {

                throw new PaskellCompileException("Expected bracket", tokenCode.Length - 1);

            }

            if (isFunction)

            {

                //Find what's on the right side of the function map

                Token[] newTokenCode = new Token[tokenCode.Length - functionMapIndex - 1];

                Array.Copy(tokenCode, functionMapIndex + 1, newTokenCode, 0, newTokenCode.Length);

                try

                {

                    typeSignature = new TypeSignature(left, ConstructTypeSignature(newTokenCode));

                }

                catch (PaskellCompileException e)

                {

                    throw new PaskellCompileException(e.ErrorMessage, e.Index + functionMapIndex + 1);

                }

            }

            else

            {

                if (bracketEndIndex != 0)

                {

                    Token[] newTokenCode = new Token[tokenCode.Length - 2];

                    Array.Copy(tokenCode, 1, newTokenCode, 0, newTokenCode.Length);

                    try

                    {

                        typeSignature = ConstructTypeSignature(newTokenCode);

                    }

                    catch (PaskellCompileException e)

                    {

                        throw new PaskellCompileException(e.ErrorMessage, e.Index + bracketStartIndex);

                    }

                }

                else if (tokenCode.Length > 1)

                {

                    throw new PaskellCompileException("Too many tokens in type signature", 0);

                }

                else

                {

                    //When there's only one token, the type signature must be just a type given by that token

                    if (tokenCode[0].TokenType != TokenType.Word)

                    {

                        throw new PaskellCompileException("Expected type", 0);

                    }

                    foreach (OperandType operandType in Enum.GetValues(typeof(OperandType)))

                    {

                        if (tokenCode[0].Code == operandType.ToString())

                        {

                            typeSignature = new TypeSignature(operandType.GetPType());

                            break;

                        }

                    }

                    if (typeSignature == null)

                    {

                        throw new PaskellCompileException("Invalid type", 0);

                    }

                }

            }

            return typeSignature;

        }

        //This works to take an expression definition in token code and push subexpressions into the stack in RPN fashion

        //It also applies condition specifiers to the subexpression depending on if they're part of a condition block, and where

        private static TypeSignature PushSubExpressions(Token[] tokenCode, List<PExpression> expressions, TypeSignature targetTypeSignature,

                                                            PExpression outExpression, Stack<ConditionSpecifier> conditionSpecifiers = null, bool typeSignatureMustMatch = false)

        {

            int bracketNesting = 0;

            int bracketStartIndex = 0;

            int conditionNesting = 0;

            int clauseStartIndex = 0;

            int argumentCount = 0;

            TypeSignature baseTypeSignature = null;

            TypeSignature argumentTypeSignature = targetTypeSignature;      //Only assigning as such for literal values before a base subexpression has been declared

            TypeSignature ifTrueTypeSignature = null;      //Used to match both then and else clause type signatures, if part of a condition block

            //If this function is recursed, it may be compiling part of a condition clause and so must retain the condition specifiers already applied

            //Otherwise a new stack must be initialised

            if (conditionSpecifiers == null)

            {

                conditionSpecifiers = new Stack<ConditionSpecifier>();

            }

            for (int i = 0; i < tokenCode.Length; i++)

            {

                Token token = tokenCode[i];

                if (token.TokenType == TokenType.FunctionMap || token.TokenType == TokenType.Equate)

                {

                    throw new PaskellCompileException("Unexpected token", i);

                }

                if (baseTypeSignature != null)

                {

                    //If a base subexpression has already been declared, then the next subexpression evaluated must be an argument

                    //The type signature must match that of the argument type of the type signature of the base subexpression for the given argument expected

                    if (argumentCount < baseTypeSignature.ArgumentCount)

                    {

                        argumentTypeSignature = baseTypeSignature[argumentCount].Parameter;

                    }

                    else

                    {

                        throw new PaskellCompileException("Unexpected token", i);

                    }

                }

                if (token.TokenType == TokenType.Bracket && conditionNesting == 0)

                {

                    //With bracket clauses, this function gets recursed with their contents

                    if (token.Code == "(")

                    {

                        if (bracketNesting == 0)

                        {

                            bracketStartIndex = i + 1;

                        }

                        bracketNesting++;

                    }

                    else

                    {

                        if (bracketNesting == 0)

                        {

                            throw new PaskellCompileException("Unexpected bracket", i);

                        }

                        else

                        {

                            bracketNesting--;

                            if (bracketNesting == 0)

                            {

                                Token[] newTokenCode = new Token[i - bracketStartIndex];

                                Array.Copy(tokenCode, bracketStartIndex, newTokenCode, 0, newTokenCode.Length);

                                try

                                {

                                    //Importantly, if the current bracket clause is expected to be the base expression (which by definition of the language wouldn't be necessary to

                                    //even include, but since it's valid to write it must be considered) then it must be compiled, but not necessarily match the target type signature

                                    //yet, as the base subexpression will still be passed arguments

                                    if (baseTypeSignature == null)

                                    {

                                        baseTypeSignature = PushSubExpressions(newTokenCode, expressions, targetTypeSignature, outExpression, conditionSpecifiers);

                                    }

                                    //Otherwise, the bracket clause must represent an argument of the base subexpression, and so the returned type signature must match that of the

                                    //argument type signature, hence passing true to typeSignatureMustMatch here

                                    else

                                    {

                                        PushSubExpressions(newTokenCode, expressions, argumentTypeSignature, outExpression, conditionSpecifiers, true);

                                        argumentCount++;        //Important to move on to the next argument type signature

                                    }

                                }

                                catch (PaskellCompileException e)

                                {

                                    throw new PaskellCompileException(e.ErrorMessage, e.Index + bracketStartIndex);

                                }

                            }

                        }

                    }

                }

                else if (token.TokenType == TokenType.ConditionStatement && bracketNesting == 0)

                {

                    //Throughout the condition block compiling, the exact same applies as with the bracket clauses regarding the base expression not having to match the target

                    //type signature, but the arguments having to match the argument type signature

                    //When recursing, the conditionSpecifiers stack is passed with an additional appropriate specifier at the top of the stack

                    if (token.Code == "if")

                    {

                        if (conditionNesting == 0)

                        {

                            conditionNesting = 1;

                            clauseStartIndex = i + 1;

                            conditionSpecifiers.Push(ConditionSpecifier.Condition);

                        }

                        else

                        {

                            //Nested condition statements are ignored and handled by the recursive call of this function, with another layer in the condition specifiers stack

                            conditionNesting++;

                        }

                    }

                    else if (conditionNesting == 1 && conditionSpecifiers.Peek() == ConditionSpecifier.Condition && token.Code == "then")

                    {

                        Token[] newTokenCode = new Token[i - clauseStartIndex];

                        Array.Copy(tokenCode, clauseStartIndex, newTokenCode, 0, newTokenCode.Length);

                        try

                        {

                            //The exception for the type signature matching is for the if clause where it must always be a bool value

                            //As such the type signature is given as being a bool, and the clause is required to match that

                            PushSubExpressions(newTokenCode, expressions, new TypeSignature(typeof(bool)), outExpression, conditionSpecifiers, true);

                        }

                        catch (PaskellCompileException e)

                        {

                            throw new PaskellCompileException(e.ErrorMessage, e.Index + clauseStartIndex);

                        }

                        clauseStartIndex = i + 1;

                        conditionSpecifiers.Pop();

                        conditionSpecifiers.Push(ConditionSpecifier.IfTrue);

                    }

                    else if (conditionNesting == 1 && conditionSpecifiers.Peek() == ConditionSpecifier.IfTrue && token.Code == "else")

                    {

                        Token[] newTokenCode = new Token[i - clauseStartIndex];

                        Array.Copy(tokenCode, clauseStartIndex, newTokenCode, 0, newTokenCode.Length);

                        try

                        {

                            if (baseTypeSignature == null)

                            {

                                ifTrueTypeSignature = PushSubExpressions(newTokenCode, expressions, targetTypeSignature, outExpression, conditionSpecifiers);

                            }

                            else

                            {

                                PushSubExpressions(newTokenCode, expressions, argumentTypeSignature, outExpression, conditionSpecifiers, true);

                                //Doesn't increment argumentCount here as the else clause must be compiled first

                            }

                        }

                        catch (PaskellCompileException e)

                        {

                            throw new PaskellCompileException(e.ErrorMessage, e.Index + clauseStartIndex);

                        }

                        clauseStartIndex = i + 1;

                        conditionSpecifiers.Pop();

                        conditionSpecifiers.Push(ConditionSpecifier.IfFalse);

                    }

                    else if (token.Code == "endif")

                    {

                        if (conditionNesting == 1 && conditionSpecifiers.Peek() == ConditionSpecifier.IfFalse)

                        {

                            Token[] newTokenCode = new Token[i - clauseStartIndex];

                            Array.Copy(tokenCode, clauseStartIndex, newTokenCode, 0, newTokenCode.Length);

                            try

                            {

                                if (baseTypeSignature == null)

                                {

                                    baseTypeSignature = PushSubExpressions(newTokenCode, expressions, targetTypeSignature, outExpression, conditionSpecifiers);

                                    //The type signatures of both the then and else clauses must match, otherwise there is ambiguity about the type signature of the condition block

                                    if (ifTrueTypeSignature != baseTypeSignature)

                                    {

                                        throw new PaskellCompileException("Both clauses in condition block don't match type signature", clauseStartIndex);

                                    }

                                }

                                else

                                {

                                    //A check for if the clauses match isn't necessary here as they are both checked again the argument type signature in recursed calls

                                    PushSubExpressions(newTokenCode, expressions, argumentTypeSignature, outExpression, conditionSpecifiers, true);

                                    argumentCount++;

                                }

                            }

                            catch (PaskellCompileException e)

                            {

                                throw new PaskellCompileException(e.ErrorMessage, e.Index + clauseStartIndex);

                            }

                            conditionSpecifiers.Pop();

                            conditionNesting--;

                        }

                        else if (conditionNesting > 1)

                        {

                            conditionNesting--;

                        }

                    }

                }

                else if (bracketNesting == 0 && conditionNesting == 0)

                {

                    if (token.TokenType == TokenType.Word)

                    {

                        PExpression expression;

                        PExpression[] results = expressions.Where(x => x.Identifier == token.Code).ToArray();

                        if (results.Length != 1)

                        {

                            throw new PaskellCompileException($"No unique definition for expression {tokenCode[i].Code}", i);

                        }

                        else

                        {

                            expression = results[0];

                        }

                        //If the expression is set to be the base subexpression,  its type signature doesn't have to match the target type signature as it will be passed arguments

                        if (baseTypeSignature == null)

                        {

                            baseTypeSignature = expression.TypeSignature;

                            //Pushing the base subexpression onto the stack, argument count given as the number of arguments to make its type signature match the target type signature

                            outExpression.PushSubExpression(expression, baseTypeSignature.ArgumentCount - targetTypeSignature.ArgumentCount,

                                new Stack<ConditionSpecifier>(conditionSpecifiers));

                        }

                        //Otherwise, it must match the type signature of the next argument of the base subexpression

                        else

                        {

                            if (expression.TypeSignature != argumentTypeSignature)

                            {

                                throw new PaskellCompileException($"Argument must be of type {argumentTypeSignature.Value}", i);

                            }

                            else

                            {

                                //Pushing the subexpression onto the stack, argument count 0 as it will not evaluate anything

                                outExpression.PushSubExpression(expression, 0, new Stack<ConditionSpecifier>(conditionSpecifiers));

                                argumentCount++;

                            }

                        }

                    }

                    else

                    {

                        //If the token is an operand, it can either be the value of the whole expression, or must be an argument

                        if (!argumentTypeSignature.IsFunction)

                        {

                            try

                            {

                                Type type = null;

                                dynamic variable = null;

                                if (argumentTypeSignature.Type != null)

                                {

                                    type = argumentTypeSignature.Type;

                                    TypeConverter converter = TypeDescriptor.GetConverter(type);

                                    variable = converter.ConvertFromString(token.Code);

                                }

                                else

                                {

                                    //Every operand type must be checked if the type required is generic

                                    bool success = false;

                                    foreach (OperandType operandType in Enum.GetValues(typeof(OperandType)))

                                    {

                                        try

                                        {

                                            type = operandType.GetPType();

                                            TypeConverter converter = TypeDescriptor.GetConverter(type);

                                            variable = converter.ConvertFromString(token.Code);

                                            success = true;

                                            break;

                                        }

                                        catch

                                        {

                                            success = false;

                                        }

                                    }

                                    if (!success)

                                    {

                                        //This won't happen

                                        throw new PaskellCompileException("You messed up your code P", i);

                                    }

                                }

                                //Pushes the variable onto the stack, argument count 0 as it is not a function

                                outExpression.PushSubExpression(new PExpression(variable), 0, new Stack<ConditionSpecifier>(conditionSpecifiers));

                                if (baseTypeSignature == null)

                                {

                                    baseTypeSignature = new TypeSignature(type);

                                    //After this the current call of this function should end, as a literal value cannot take arguments

                                }

                                else

                                {

                                    argumentCount++;

                                }

                            }

                            catch (Exception)

                            {

                                throw new PaskellCompileException($"Expected literal value of type {argumentTypeSignature.Value}", i);

                            }

                        }

                        else

                        {

                            throw new PaskellCompileException($"Expected expression of type {argumentTypeSignature.Value}", i);

                        }

                    }

                }

            }

            //Checking for bad nesting or unclosed clauses

            if (bracketNesting > 0)

            {

                throw new PaskellCompileException("Expected close bracket", tokenCode.Length - 1);

            }

            if (conditionNesting > 0)

            {

                string statement = "";

                switch (conditionSpecifiers.Peek())

                {

                    case ConditionSpecifier.Condition:

                        statement = "then";

                        break;

                    case ConditionSpecifier.IfTrue:

                        statement = "else";

                        break;

                    case ConditionSpecifier.IfFalse:

                        statement = "endif";

                        break;

                }

                throw new PaskellCompileException($"Expected {statement} statement", tokenCode.Length - 1);

            }

            if (baseTypeSignature == null)

            {

                throw new PaskellCompileException("Expected expression", 0);

            }

            //Only if the type signature must match the target i.e. when the subexpression is an argument or when it composes the final definition of an expression

            if (typeSignatureMustMatch && baseTypeSignature[argumentCount] != targetTypeSignature)

            {

                throw new PaskellCompileException($"Expression must be of type {targetTypeSignature.Value}", 0);

            }

            //Returns the type signature of the new base subexpression, if nested

            return baseTypeSignature[argumentCount];

        }

        //All the base expression functions that are added at the beginning of compilation

        private static PExpression Add(Queue<PExpression> a)

        {

            return new PExpression(a.Dequeue().Evaluate().Value + a.Dequeue().Evaluate().Value);

        }

        private static PExpression Subtract(Queue<PExpression> a)

        {

            return new PExpression(a.Dequeue().Evaluate().Value - a.Dequeue().Evaluate().Value);

        }

        private static PExpression Multiply(Queue<PExpression> a)

        {

            return new PExpression(a.Dequeue().Evaluate().Value \* a.Dequeue().Evaluate().Value);

        }

        private static PExpression Divide(Queue<PExpression> a)

        {

            return new PExpression(a.Dequeue().Evaluate().Value / a.Dequeue().Evaluate().Value);

        }

        private static PExpression Not(Queue<PExpression> a)

        {

            return new PExpression(!(bool)a.Dequeue().Evaluate().Value);

        }

        private static PExpression And(Queue<PExpression> a)

        {

            return new PExpression((bool)a.Dequeue().Evaluate().Value && (bool)a.Dequeue().Evaluate().Value);

        }

        private static PExpression Or(Queue<PExpression> a)

        {

            return new PExpression((bool)a.Dequeue().Evaluate().Value || (bool)a.Dequeue().Evaluate().Value);

        }

        private static PExpression Xor(Queue<PExpression> a)

        {

            return new PExpression((bool)a.Dequeue().Evaluate().Value ^ (bool)a.Dequeue().Evaluate().Value);

        }

        private static PExpression EqualTo(Queue<PExpression> a)

        {

            return new PExpression(a.Dequeue().Evaluate().Value == a.Dequeue().Evaluate().Value);

        }

        private static PExpression GreaterThan(Queue<PExpression> a)

        {

            return new PExpression(a.Dequeue().Evaluate().Value > a.Dequeue().Evaluate().Value);

        }

        private static PExpression LessThan(Queue<PExpression> a)

        {

            return new PExpression(a.Dequeue().Evaluate().Value < a.Dequeue().Evaluate().Value);

        }

    }

    //https://stackoverflow.com/questions/479410/enum-tostring-with-user-friendly-strings

    static class CustomEnumExtensions

    {

        //Used to retrieve the Regex pattern given by the RegexPattern attribute in the TokenType enum

        public static string GetPattern(this TokenType enumValue)

        {

            string pattern = "";

            MemberInfo memberInfo = enumValue.GetType().GetMember(enumValue.ToString())[0];

            foreach (var customAttribute in memberInfo.GetCustomAttributes(typeof(RegexPattern), false))

            {

                pattern = ((RegexPattern) customAttribute).Pattern;

            }

            return pattern;

        }

        //Used to retrieve the type given by the PaskellType attribute in the OperandType enum

        public static Type GetPType(this OperandType enumValue)

        {

            Type type = null;

            MemberInfo memberInfo = enumValue.GetType().GetMember(enumValue.ToString())[0];

            foreach (var customAttribute in memberInfo.GetCustomAttributes(typeof(PaskellType), false))

            {

                type = ((PaskellType)customAttribute).Type;

            }

            return type;

        }

    }

    class RegexPattern : Attribute

    {

        public string Pattern;

        public RegexPattern(string pattern)

        {

            Pattern = pattern;

        }

    }

    class PaskellType : Attribute

    {

        public Type Type;

        public PaskellType(Type type)

        {

            Type = type;

        }

    }

    struct Token : IComparable

    {

        public readonly string Code;            //What the actual source code contained (necessary to distinguish tokens of the same type such as operands)

        public readonly TokenType TokenType;

        public readonly int Index;

        public Token(string code, TokenType tokenType, int index)

        {

            Code = code;

            TokenType = tokenType;

            Index = index;

        }

        public int CompareTo(object obj)        //Necessary to be comparable by index so that the tokeniser can sort all tokens once made

        {

            Token token = (Token)obj;

            return Index - token.Index;

        }

    }

    //Just an easier object to return for the IDE

    public class PContext

    {

        public PExpression[] Expressions;

        public PContext(PExpression[] expressions)

        {

            Expressions = expressions;

        }

    }

    //Summarises tokeniser success and errors

    public struct TokeniserReturnState

    {

        public bool Success { get; }

        public Queue<TokeniserReturnError> Errors { get; }

        public TokeniserReturnState(bool success)

        {

            Success = success;

            Errors = new Queue<TokeniserReturnError>();

        }

    }

    public struct TokeniserReturnError

    {

        public int Index { get; }

        public TokeniserReturnError(int index)

        {

            Index = index;

        }

    }

    //Summarises compiler success and errors

    public struct CompilerReturnState

    {

        public bool Success { get; }

        public Queue<PaskellCompileException> Exceptions { get; }

        public CompilerReturnState(bool success)

        {

            Success = success;

            Exceptions = new Queue<PaskellCompileException>();

        }

    }

    enum TokenType

    {

        [RegexPattern(@"\(|\)")]

        Bracket,

        [RegexPattern("=")]

        Equate,

        [RegexPattern("->")]

        FunctionMap,

        [RegexPattern(@"((?<=\s|\(|\)|=|(->))|^)(?!((true|false|if|then|else|endif)\b))[A-Za-z][A-Za-z0-9]\*")]

        Word,

        [RegexPattern(@"((?<=\s|\(|\)|=|(->))|^)(-?([0-9]+(\.[0-9]+)?)|('.')|true|false)(?![A-Za-z0-9]|\.)")]

        Operand,

        [RegexPattern(@"((?<=\s|\(|\)|=|(->))|^)(if|then|else|endif)\b")]

        ConditionStatement,

        [RegexPattern(@"(?<!\\)\n\s\*")]

        LineBreak

    }

    //@ symbol so to allow type keywords as names

    public enum OperandType

    {

        [PaskellType(typeof(long))]

        @int,

        [PaskellType(typeof(double))]

        @float,

        [PaskellType(typeof(char))]

        @char,

        [PaskellType(typeof(bool))]

        @bool

    }

    //Thrown inside runtime, when things fail

    public class PaskellRuntimeException : Exception

    {

        public new PaskellRuntimeException InnerException;

        public PExpression PExpression;

        public string ErrorMessage;

        public PaskellRuntimeException(string errorMessage, PExpression pExpression, PaskellRuntimeException innerException = null)

        {

            ErrorMessage = errorMessage;

            PExpression = pExpression;

            InnerException = innerException;

        }

    }

    //Thrown by the compiler to indicate where errors occurred

    //These get caught and catalogued so that all the errors can be displayed an made useful

    public class PaskellCompileException : Exception

    {

        public int Index;

        public int Line;

        public string ErrorMessage;

        public PaskellCompileException(string errorMessage, int index, int line = 0)

        {

            ErrorMessage = errorMessage;

            Line = line;

            Index = index;

        }

    }}

# Runtime.cs

using System;

using System.Collections.Generic;

namespace Parse

{

    public class PExpression

    {

        public string Identifier { get; private set; }

        public dynamic Value { get; private set; }      //Used if the expression is an evaluated variable

        public TypeSignature TypeSignature { get; private set; }

        //Used if the expression is an unevaluated expression i.e. an unevaluated variable or a function definition

        private Stack<(PExpression, (int, Stack<ConditionSpecifier>))> SubExpressions { get; }

        //Used if the expression is a parameter

        private readonly bool isParamater = false;

        private int parameterIndex;

        private PExpressionState state = PExpressionState.Definition;   //Definitions should never be directly worked on; they must be cloned first

        //Variables used if the expresion is a base expression

        private readonly bool isBaseExpression = false;

        private readonly Queue<PExpression> arguments;

        private readonly Func<Queue<PExpression>, PExpression> function;

        //Instantiates expression representing data value

        public PExpression(dynamic value, string identifier = "")

        {

            Identifier = identifier;

            Value = value;

            state = PExpressionState.Evaluated;

            TypeSignature = new TypeSignature(value.GetType());

        }

        //Instatiates base function definition

        public PExpression(Func<Queue<PExpression>, PExpression> function, TypeSignature typeSignature, string identifier = "")

        {

            Identifier = identifier;

            TypeSignature = typeSignature;

            arguments = new Queue<PExpression>();

            isBaseExpression = true;

            this.function = function;

        }

        //Instantiates function or unevaluated variable definition (function should be constructed using SubExpressions stack instantiated here)

        public PExpression(TypeSignature typeSignature, string identifier = "")

        {

            Identifier = identifier;

            TypeSignature = typeSignature;

            SubExpressions = new Stack<(PExpression, (int, Stack<ConditionSpecifier>))>();

        }

        //Instantiates paramater of function (would be added to SubExpressions stack of function definition)

        public PExpression(int parameterIndex, TypeSignature typeSignature, string identifier = "")

        {

            isParamater = true;

            Identifier = identifier;

            TypeSignature = typeSignature;

            this.parameterIndex = parameterIndex;

        }

        //Since SubExpressions is a protected property, it needs a public accessor to push items

        public void PushSubExpression(PExpression subExpression, int argumentCount, Stack<ConditionSpecifier> conditionSpecifiers)

        {

            SubExpressions.Push((subExpression, (argumentCount, conditionSpecifiers)));

        }

        //This function is called when the value of an expression is trying to be retrieved

        //The expression must therefore be a variable or a parameterised function

        public PExpression Evaluate()

        {

            if (state == PExpressionState.Evaluated)

            {

                return this;

            }

            else

            {

                PExpression workedExpression = CloneWorkedExpression();

                PExpression result = EvaluateSubExpressions(workedExpression.SubExpressions);

                if (!result.TypeSignature.IsFunction)

                {

                    result.state = PExpressionState.Evaluated;

                }

                else

                {

                    throw new PaskellRuntimeException("Trying to find the value of unevaluated expression", result);

                }

                return result;

            }

        }

        private PExpression EvaluateSubExpressions(Stack<(PExpression, (int, Stack<ConditionSpecifier>))> subExpressions)

        {

            Stack<PExpression> workingStack = new Stack<PExpression>();

            EvaluateSubExpressions(new Queue<(PExpression, (int, Stack<ConditionSpecifier>))>(subExpressions), workingStack);

            //Using the working stack, the final result of the evaluated subexpressions should be the only element on the stack, which can then be popped

            return workingStack.Pop();

        }

        private void EvaluateSubExpressions(Queue<(PExpression, (int, Stack<ConditionSpecifier>))> subExpressions, Stack<PExpression> workingStack)

        {

            //These queues used as temporary holding of subexpressions, to ensure that only the necessary subexpressions are evaluated depending on a condition

            Queue<(PExpression, (int, Stack<ConditionSpecifier>))> condition = new Queue<(PExpression, (int, Stack<ConditionSpecifier>))>();

            Queue<(PExpression, (int, Stack<ConditionSpecifier>))> ifTrue = new Queue<(PExpression, (int, Stack<ConditionSpecifier>))>();

            Queue<(PExpression, (int, Stack<ConditionSpecifier>))> ifFalse = new Queue<(PExpression, (int, Stack<ConditionSpecifier>))>();

            while (subExpressions.Count > 0)

            {

                try

                {

                    (PExpression expression, (int argumentCount, Stack<ConditionSpecifier> conditionSpecifiers) evaluationSpecifiers) workingExpressionTuple = subExpressions.Dequeue();

                    PExpression expression = workingExpressionTuple.expression;

                    //This copy is necessary as the count may change within the loop, but the initial count must be remembered

                    int conditionSpecifierCount = workingExpressionTuple.evaluationSpecifiers.conditionSpecifiers.Count;

                    if (conditionSpecifierCount != 0)

                    {

                        ConditionSpecifier conditionSpecifier = workingExpressionTuple.evaluationSpecifiers.conditionSpecifiers.Pop();

                        switch (conditionSpecifier)

                        {

                            case ConditionSpecifier.Condition:

                                condition.Enqueue(workingExpressionTuple);

                                break;

                            case ConditionSpecifier.IfTrue:

                                ifTrue.Enqueue(workingExpressionTuple);

                                break;

                            case ConditionSpecifier.IfFalse:

                                //Condition clause would only ever be preceded by an else clause, or a non-condition subexpression

                                //This is for the case of a condition clause being preceded by an else clause

                                if (condition.Count != 0)

                                {

                                    EvaluateSubExpressions(condition, workingStack);

                                    bool conditionValue = workingStack.Pop().Evaluate().Value;

                                    if (conditionValue == true)

                                    {

                                        EvaluateSubExpressions(ifTrue, workingStack);

                                        ifFalse.Clear();

                                    }

                                    else

                                    {

                                        EvaluateSubExpressions(ifFalse, workingStack);

                                        ifTrue.Clear();

                                    }

                                }

                                ifFalse.Enqueue(workingExpressionTuple);

                                break;

                        }

                    }

                    //This is for the case of a condition clause being preceded by a non-condition subexpression or the start of the subexpressions

                    if (condition.Count != 0 && (conditionSpecifierCount == 0 || subExpressions.Count == 0))

                    {

                        //The bool value of the condition must be evaluated, then the correct clause evaluated according to that

                        //When recursively calling EvaluateSubExpressions here, nested condition blocks are considered as the top condition specifier is popped above

                        EvaluateSubExpressions(condition, workingStack);

                        bool conditionValue = workingStack.Pop().Evaluate().Value;

                        if (conditionValue == true)

                        {

                            EvaluateSubExpressions(ifTrue, workingStack);

                            ifFalse.Clear();

                        }

                        else

                        {

                            EvaluateSubExpressions(ifFalse, workingStack);

                            ifTrue.Clear();

                        }

                    }

                    //If there is no condition specified, then the subexpression can just be normally evaluated

                    if (conditionSpecifierCount == 0)

                    {

                        for (int i = 0; i < workingExpressionTuple.evaluationSpecifiers.argumentCount; i++)

                        {

                            expression = expression.Evaluate(workingStack.Pop());

                        }

                        if (!expression.TypeSignature.IsFunction)

                        {

                            expression = expression.Evaluate();

                        }

                        workingStack.Push(expression);

                    }

                }

                catch

                {

                    throw new PaskellRuntimeException("Failed to evaluate expression", this);

                }

            }

        }

        //Called when passing an argument to a function

        public PExpression Evaluate(PExpression argument)

        {

            //Important here however if that the expression is an definition, it remains unchanged and a new instance

            //of the class is created as the worked expression, protecting the expression definition

            PExpression workedExpression = CloneWorkedExpression();     //Only clones if not already worked expression (handled within method)

            if (!isBaseExpression)

            {

                if (workedExpression.state == PExpressionState.WorkedExpression)

                {

                    //Using a temp stack effectively allows looping through a stack by transferring items to the temp stack, then back to the original once done

                    Stack<(PExpression, (int, Stack<ConditionSpecifier>))> tempStack = new Stack<(PExpression, (int, Stack<ConditionSpecifier>))>();

                    while (workedExpression.SubExpressions.Count > 0)

                    {

                        (PExpression expression, (int, Stack<ConditionSpecifier>)) expression = workedExpression.SubExpressions.Pop();

                        if (expression.expression.isParamater)

                        {

                            //It is important to clone the parameter, else each of a parameter in an expression where there are multiple will be changed

                            expression.expression = expression.expression.CloneWorkedExpression();

                            //Parameters are indexed, 0 upwards, where the highest numbered are for the final argument

                            if (expression.expression.parameterIndex == 0)

                            {

                                expression.expression = argument;

                            }

                            else

                            {

                                //So that for the next argument passed, the next index i.e. index 1 will be replaced

                                //This is done by decrementing all indicies, so 1 becomes 0 etc.

                                expression.expression.parameterIndex--;

                            }

                        }

                        tempStack.Push(expression);

                    }

                    while (tempStack.Count > 0)

                    {

                        workedExpression.SubExpressions.Push(tempStack.Pop());

                    }

                    workedExpression.TypeSignature = workedExpression.TypeSignature.Return;

                    if (!workedExpression.TypeSignature.IsFunction)

                    {

                        //Function fully parameterised

                        workedExpression = workedExpression.Evaluate();

                    }

                }

                else

                {

                    throw new PaskellRuntimeException("Trying to pass argument to non-function expression", workedExpression);

                }

                return workedExpression;

            }

            //For if the expression is a base expression

            else

            {

                workedExpression.arguments.Enqueue(argument);

                workedExpression.TypeSignature = workedExpression.TypeSignature.Return;

                if (!workedExpression.TypeSignature.IsFunction)

                {

                    try

                    {

                        //Calls the function of the base expression and passes it the stack of arguments passed

                        //Calling clone function just allows the identifier to be assigned

                        PExpression result = function(workedExpression.arguments);

                        result.Identifier = Identifier;

                        workedExpression = result;

                    }

                    catch

                    {

                        throw new PaskellRuntimeException("Failure trying to evaluate base expression", workedExpression);

                    }

                }

                return workedExpression;

            }

        }

        //When compiled, every expression definition is classed as a definition, which indicates that it represents the original definition of any expression.

        //It is important that when evaluating expressions, the original definition remains untouched as a function or expression may be referenced

        //in multiple places, for example in recursion. Therefore a clone must be used instead, and it is identified as a worked expression which is

        //safe to work on

        private PExpression CloneWorkedExpression()

        {

            PExpression workedExpression;

            if (state == PExpressionState.Definition)

            {

                if (isParamater)

                {

                    workedExpression = new PExpression(parameterIndex, TypeSignature, Identifier);

                }

                else if (isBaseExpression)

                {

                    workedExpression = new PExpression(function, TypeSignature, Identifier);

                }

                else

                {

                    workedExpression = new PExpression(TypeSignature, Identifier);

                    Stack<(PExpression, (int, Stack<ConditionSpecifier>))> tempStack = new Stack<(PExpression, (int, Stack<ConditionSpecifier>))>();

                    while (SubExpressions.Count > 0)

                    {

                        tempStack.Push(SubExpressions.Pop());

                    }

                    while (tempStack.Count > 0)

                    {

                        //If subexpressions contain references to other expression definitions, they shouldn't be cloned until they have to be evaluated

                        //Therefore they are left alone, and will naturally be cloned when evaluated i.e. this function gets called from their Evaluate call

                        (PExpression expression, (int argumentCount, Stack<ConditionSpecifier> conditionSpecifiers) evaluationSpecifiers) subExpression = tempStack.Pop();

                        PExpression workedSubExpression = subExpression.expression;

                        if (workedSubExpression.state != PExpressionState.Definition)

                        {

                            workedSubExpression = workedSubExpression.CloneWorkedExpression();

                        }

                        //Important to clone the evaluation specifier stacks too

                        workedExpression.SubExpressions.Push((workedSubExpression, (subExpression.evaluationSpecifiers.argumentCount,

                                                                new Stack<ConditionSpecifier>(new Stack<ConditionSpecifier>(subExpression.evaluationSpecifiers.conditionSpecifiers)))));

                        SubExpressions.Push(subExpression);

                    }

                }

                workedExpression.state = PExpressionState.WorkedExpression;

            }

            //If already cloned

            else

            {

                workedExpression = this;

            }

            return workedExpression;

        }

    }

    public class TypeSignature

    {

        public bool IsFunction { get; }

        public Type Type { get; }                   // --Used if IsFunction is false

        public TypeSignature Parameter { get; }     // --Used if IsFunction is true

        public TypeSignature Return { get; }        // \_/

        public string Value => ToString(); //Mostly for debug purposes, writes type signature in Paskell syntax

        public int ArgumentCount

        {

            get

            {

                if (IsFunction)

                {

                    return 1 + Return.ArgumentCount;

                }

                else

                {

                    return 0;

                }

            }

        }

        public TypeSignature FinalType

        {

            get

            {

                if (IsFunction)

                {

                    return Return.FinalType;

                }

                else

                {

                    return this;

                }

            }

        }

        //Every next index of a type signature is just that type signature's return

        //From this, every argument type of a type signature can be indexed as TypeSignature[i].Parameter

        public TypeSignature this[int i]

        {

            get

            {

                if (i > ArgumentCount || i < 0)

                {

                    throw new IndexOutOfRangeException();

                }

                else

                {

                    if (i == 0)

                    {

                        return this;

                    }

                    else

                    {

                        return Return[i - 1];

                    }

                }

            }

        }

        //Used to check if type signatures are the same (generic types count as equalling anything)

        //Also necessary to allow comparison with null if uninstantiated, though I don't think I've fully implemented a solution for this

        //However it works for my limited use of comparisons

        public override bool Equals(object obj)

        {

            //Check for null and compare run-time types.

            if ((obj == null) || !GetType().Equals(obj.GetType()))

            {

                return false;

            }

            else

            {

                TypeSignature typeSignature = (TypeSignature)obj;

                if (IsFunction && typeSignature.IsFunction)

                {

                    return Parameter == typeSignature.Parameter && Return == typeSignature.Return;

                }

                else if (!IsFunction && !typeSignature.IsFunction)

                {

                    if (Type == null || typeSignature.Type == null)

                    {

                        return true;

                    }

                    else

                    {

                        return Type == typeSignature.Type;

                    }

                }

                else

                {

                    return false;

                }

            }

        }

        public static bool operator == (TypeSignature t1, TypeSignature t2)

        {

            return ReferenceEquals(t1, null) && ReferenceEquals(t2, null) || t1.Equals(t2);

        }

        public static bool operator !=(TypeSignature t1, TypeSignature t2)

        {

            return !(t1 == t2);

        }

        //Instantiates generic variable type

        public TypeSignature()

        {

            IsFunction = false;

        }

        //Instantiates type signature of a variable of type given

        public TypeSignature(Type type)

        {

            IsFunction = false;

            Type = type;

            GetTypeString(type);

        }

        //Instantiates type signature of function with given parameter and return signatures

        public TypeSignature(TypeSignature parameter, TypeSignature returnt)

        {

            IsFunction = true;

            Parameter = parameter;

            Return = returnt;

        }

        //Produces Paskell syntax type signature

        public string ToString(bool brackets = true)

        {

            return IsFunction ? $"{(brackets ? "(" : "")}{Parameter.ToString()} -> {Return.ToString(false)}{(brackets ? ")" : "")}" : (Type != null ? GetTypeString(Type) : "\_");

        }

        //Ensures that the types given by the enum are returned as strings rather than the C# types

        private string GetTypeString(Type type)

        {

            foreach (OperandType operandType in Enum.GetValues(typeof(OperandType)))

            {

                if (operandType.GetPType() == type)

                {

                    return operandType.ToString();

                }

            }

            throw new InvalidOperationException("Not valid type");

        }

    }

    enum PExpressionState

    {

        Definition,

        WorkedExpression,

        Evaluated

    }

    public enum ConditionSpecifier

    {

        Condition,

        IfTrue,

        IfFalse

    }

}