Version: Final RC0

March 2023

A-LEVEL COMPUTER SCIENCE PROGRAMMING PROJECT

Ben Mullan

Note: *It is useful to open the Navigation Side-Bar in MS Word, to be able to see the structure of the document at all times.*

A DocX version of this document is available here.

Contents

Introduction

Hello, World!

This 187-page file documents the development of the ***DocScript* Programming Language**.

It was written over the course of 12 months, and covers the *planning*, *implementation*, and *testing* of the entire solution. Over 80,000 lines of computer code are written for the project, and they – along with other development logs, resources, and final binaries – can be found here:

- My final Programming-Project folder for OCR: Click Here
- DocScript, on my website: Click Here
- DocScript, on //GitHub: Click Here

DocScript in 3 Minutes

Readers of this document may wish to brace themselves, by firstly watching the following video. This was produced towards the end of the project, but nevertheless aids in elucidating the *structure* of what is – admittedly – rather a large codebase…

(https://youtu.be/ybl5pVSJOOk)

Analysis

Problem Identification

At present, one is hard-pressed to find an easy-to-use Programming Language built *from the ground-up* to handle computational numerical logic, including working with numbers in a variety of different bases and formats, along with specialist Computer Science operators and mathematical functions, all in the same package. Certain existing systems which go *some of the way* towards permitting the programmer to use a *select few* number bases and mathematical operations, are inflexible, and do not function across an adequate variety of different *types* of computer systems, due – in many cases – to the increasingly diverse range of CPU architectures and Operating Systems used today.

Even – for instance – whilst my class and I were learning about some of the different number bases frequently used in Computer Science earlier this year, we hadn't a toolset to use for the purposes of exploring and testing concepts, or validating answers to questions, e.g. for homework tasks. An environment wherein such experimentation and evaluation could occur, would be invaluable not only to students of CS, but potentially other disciplines too.

I will therefore be *creating* a new Programming Language from scratch, to solve these problems.

This involves:

- Designing a programming language **SPECIFICATION** (for the syntax, keywords, etc…)
- Implementing that specification into a **RUNTIME** "*translator*", to enable source code written in accordance with the specification, to be understood and executed

There are many areas of Computer Science which require simple to complex mathematical processes, including – but by no means limited to – the manipulation of numbers in different bases and formats (such as fixed- and floating-point representations), hashing, encryption, and compression principles, bitwise operations, and IP Address-related calculations, just to name a few! During the forthcoming stages of §Analysis, I shall be delineating precisely *which* features are the most important to the stakeholders of the product, by asking these stakeholders themselves.

But in addition to dedicated A-Level CS maths features, the language will need to implement a basic range of commonplace procedural programming constructs, and be Turing-complete. Of particular importance to the solution, is the fact that it ought to run on a range of different types of computer system, as used by the stakeholders in question.

Stakeholders

Conveniently, the primary stakeholders herefor consist predominantly of **students studying GCSE and A-Level Computer Science or Maths**. These clients are pupils between the ages of 16 and 19, with varying levels of experience in informatics and mathematics. This is an important factor for the project, because it is vital that I collect and analyse the viewpoints of a heterogeneous selection of stakeholders, due to the different observations they will make on account of their different backgrounds.

Stakeholder	Relevance to Project	Intended Use-Case
Oliver	• Has been **programming for a number of years** in a hobbyist context (C++) • Studies A-Level Mathematics • In same school	Experimentation with number bases, and concept testing for maths, as well as having a general-purpose Scripting Language
Kiran	• **Inexperienced programmer** – therefore knows what's hard to learn about existing languages, and what can be improved in the product I am developing • In same school, and **representative of the wider target group** for whom this product is intended	Potentially for learning and reinforcing some mathematical and programming concepts, and also for use of a general-purpose Programming Language

The proposed solution is appropriate to the needs of *these* stakeholders, as it is to be a simple and easily-learned environment, which the clients can therefore use for experimentational purposes without first having to spend a considerable amount of time familiarising themselves with.

In addition – however – to these *primary* stakeholders, I must consider some of the wider stakeholders who should influence the development of this system. To this end, I have identified the following *external* clients.

Stakeholder	Relevance to Project	Intended Use-Case
Joe	• Takes GCSE Computer Science • Uses **basic scripting** to automate tasks on his personal computer • Attends a different school	Some light administrative scripting, but also for learning various CS Programming and Maths concepts
Klara	• **Experienced professional software developer** • Uses, and is familiar with, a variety of mainstream Compiled and Interpreted Programming Languages • Is incidentally German, but I will translate the questions and responses into English for the interviews	For testing and experimentational purposes related to other products she is developing, as well as educational purposes

The proposed solution is appropriate to the needs of *these* stakeholders, as the software will aim to reinforce programming principles seen in more sophisticated higher-level languages. As befits users of smaller and simpler programming systems, they *will one day wish to progress onto more advanced languages*. The software being developed herein aims to provide this sort of educational basis. A particular focus will also be placed on how the user interacts with the language. Though I must further investigate exactly how these clients wish to be able to interact with the system, I have a few ideas presently about – for instance – not having to enter an *entire* program just to get some output from the language. This sort of feature would be suited to these clients who may often not need to write a complete program, but rather, a short expression or statement, for their experimentational and pedagogic resolves.

This range of stakeholders means that I will be targeting the following areas during the development of the software:

- **Hobbyist and Experimentational Use**	(Oliver and Kiran)
- **Educational Use**	(Klara, Kiran, and Joe)

Why this problem is suited to a Computational Solution

Let us imagine the following example scenario, as a use-case for the software: A user wishes list the prime numbers below 100, in each base from 2 (binary) to 32 (Duotrigesimal). Indeed, attempting to perform this inherently systematic and sequential task on anything *but* a computer, might, in this day and age, be considered preposterous.

Creating a Programming Language, therefore, to facilitate the computation of such problems, *does* fundamentally lend itself to a computational solution as computers provide an untiring, consistent, accurate, and scalable platform on which to build this software. In other words, there is no viable way to create such a system, if it does *not* rely on a computer to at least some extent.

In this instance, it rather makes sense for the *entirety* of the solution to be computer-based. Data can be easily stored on a computer, entered into the system (or by other means retrieved by the programming language), and thereafter be stored back on the computer for further manipulation, potentially in other programs. Such is the characteristically inter-compatible nature of data in standardised formats when kept on a computer. Having to manage, move, and manipulate all this data by hand, would be laborious at best, and – at worst – impossible.

Simply put, the Programming Language will need to take in some source code, analyse and validate it, and then execute the instructions given. Each of these three basic stages is best-suited to being carried out by a computer, not least because the user will expect their programs to be run in a fast and consistent and idempotent manner.

Specific Computational Principles Employed

Problem Recognition and Decomposition

Ostensibly, the overall problem is attempting to execute the instructions provided by a user, on their computer. However, the computational principles of problem recognition and decomposition allow us to identify a series of underlying sub-problems; the instructions must be syntactically and logically validated, comprehended by the computer, checked for irregularities, and finally, performed. Whilst it initially seems that executing the instructions themselves is the most complex and significant of these problems, it quickly becomes clear that in reality, the comprehension of the source code (initially just a long string) is the more mammoth undertaking. On the outset, I anticipate that breaking down the input source code will consist some of the following steps:

- Performing preliminary validation to ensure that no unexpected characters have been used in the source as a whole
- Disregarding comments, which needn't be executed
- Parsing the source, to derive a set of tokens from the keywords, identifiers, and literals of the language

- Lexically analysing the tokens and forming a tree of semantically-significant symbols
- Attesting that the order of these symbols is valid for the subsequent stages

When this is overcome, the remainder of the problem actually seems comparably simple; once the computer understands *what* to do, *doing it* is simply a matter of *applying* the instructions to a scenario, such as performing them as actions. However, the parsed input could – for instance – just as easily be used as the source for a translation program for natural languages, e.g. from English to German. Decomposition has hereby permitted us to recognise the compartmentalised and modular nature of this larger problem.

Incidentally, in this particular project there is the additional step of creating a *formal* (meaning "*concise*" and "*unambiguous*") specification for the programming language itself, often referred to as a "LangSpec". This is a separate problem to designing whatever it is that will actually interpret and execute the language's source code. Both problems, however, are amenable to decomposition.

Pattern Recognition

Recognising patterns is a fundament of interpreting source code. It is impossible to account for every possibility of input by hard-coding in the output, wherefore identifying a predefined *pattern* in the input is significantly more effective and flexible. In particular, I envisage using Regular Expressions to validate and find certain syntactical forms in the source. A *Regular Expression* is like a template for some text; it can be *satisfied* only by an instance of text which precisely matches the pattern specified in the RegEx. The RegEx "^\d{3}\-2$" – for instance – is satisfied by the string "974-2". Since getting a computer to comprehend and execute the instructions of a programming language involves a large amount of pattern recognition, this sub-problem pertains to a computational approach.

Automation

Characteristically, a medium- to high-level programming language facilitates the automation of many tasks requested by the programmer. In fact, having a system to run the language for you, constitutes automation; no manual overseeing of the execution is necessary.

In addition, a number of automation principles will be used during the *development* of the software. For instance, the IDE will automatically {Check Syntax, Resolve Dependencies, Compile, and Link the Project} on the press of [F5]. Hence, automation can speed up both the development and the execution of the system being developed here.

Divide and Conquer

The "divide and conquer" principle manifests itself in this project through the fact that several smaller problems were derived from the initial proposal. When viewed in an individual context, each smaller problem is very much manageable and can be *conquered*, before moving onto the next one.

In addition, the principle of *Multithreading* is a manifestation of "*Divide and Conquer*"; a substantial process can be divided into a number of sub-processes which can each be executed on their own CPU. Because many modern computer systems have multiple logical and physical CPUs or Cores, this multithreading can be an effective means of *conquering* a conventionally time-consuming computational task in a vastly-reduced time.

Abstraction

Considering the target stakeholders for whom this software is ultimately being designed, a large amount of *abstraction* will have to be employed in order to make the system easy to use and accessible. Naturally, many programming languages implement a wide range of abstractive features, from predefined functions and libraries, to providing object-orientated features to the programmer. The user needn't list hundreds of assembly instructions with their operands, or enumerate each of the Win32 API calls involved in a relatively simple procedure such as outputting text. Instead, this sort of process is usually outsourced to a singular inbuilt function or keyword, such as `Write()` or `Print()`. I will be interviewing stakeholders shortly to determine how exactly layers of abstraction should be implemented into the product.

In addition to abstracting complex processes into simple keywords in the language itself, the principle will also have to be used in the inner workings of the execution engine. Whilst the source code is liable to contain a large amount of information useful to the programmer, not *all* of this is of relevance when it comes to running the program. Comments, for instance, along with auxiliary whitespace and any programmer-specific syntactical layouts, are amongst the first elements to be ignored during the parsing. This is a form of abstraction; simplifying, and only paying attention to the most important components.

Primarily though, the purpose of the software being written here is to take *some* input, and produce an output based on it. Therefore, during the Design phases of the development, I will be placing a particular focus on this abstracted view of [Input ➜ Proceß ➜ Output].

Interviews and Analysis Thereof

In order to effectively identify the features most sought by the stakeholders, *and* find out what should be avoided, I shall be undertaking a series of interviews with the clients.

Questions

The first stage in this process is to devise a set of questions to ask. These questions need to establish specific pieces of data which I can use to design the product, and they also need to make sure that I understand the needs of the Stakeholders.

All Stakeholders

The following questions are for applicable to all of the stakeholders

1) Which areas of maths do you find yourself referring to most frequently, whilst programming?
2) Which computer system(s) do you currently use (a) *whilst* programming, and (b) recreationally? What are the architectures and operating systems of these computers?

3) Which components of existing programming languages or systems do you find annoying or cumbersome?
4) Which components or features of existing programming languages do you like and find useful?
5) …And what, therefore, would you perhaps like to see implemented in this system, to aid in its ease of use and functionality?

Hobbyist Use

The following additional questions are for the stakeholders involved in the hobbyist use area (represented by Oliver and Kiran)

6) How frequently do you use programming languages for your own personal experimentational and hobbyist purposes?
7) Where do you think a reasonable line can be drawn between hindering simplicity and needless complexity, in the context of a programming language?
8) Is there a particular *style* of programming (paradigm, nomenclature, or simply a set of tendencies) that you lean towards, for experimentation?
9) Are you satisfied by the programming toolset currently at your disposal?

Educational Use

The following additional questions are for the stakeholders involved in the educational use area (represented by Kiran, Joe, and Klara)

6) In your experience, what do you think makes some existing programming languages hard to learn?
7) How do you interact with the Programming Languages you already use?
8) What, if anything, do you find cumbersome about this, and how might you more ideally wish to use a programming language system?
9) What do you think some of the most important principles of higher-level programming languages are, that need to be learnt in order for students to progress onto systems requiring a *deeper* level of understanding?

Questions' Explanation and Justification

I have used a mixture of open and closed questions as appropriate, in an effort to evoke both specific responses, as well as broader and more substantial answers including detail I may not have considered thitherto.

- Question ***1*** aims to establish which mathematical utilities the programming language ought to have. This is important, as the primary point of specialisation for the language *is* its use as a mathematical experimentation and utility environment.
- Question ***2*** should enable me to determine which operating systems and computer types I need to support. This data contributes directly to some of the Hardware and Software Requirements of the product.
- Questions ***3*** and ***4***, and ***5*** investigate the problems and useful features of existing languages. The responses to this question will be very significant to the further development of the software, in terms of which components I should aim to implement.

- From question ***6H*** (Hobbyist Use Questions), I should be able to delineate what design choices need to be made, in order to make the product suitable for use at the frequency specified by the clients.
- Question ***7H***, is being asked to get the clients' views on another rather important principle of the development of the system; a balance must be met between an excessive level of *simplicity* – whereby the language would be difficult to use – and a superfluous level of *complexity* – whereby the language would be equally difficult to use, and indeed, learn.
- Question ***8H*** aims to establish whether or not a particular *style* of programming (paradigm, nomenclature, or simply a set of tendencies) might be best-suited to the experimentational uses for which the product is intended.
- Question ***9H*** identifies whether or not the client is content with their current programming toolset. It is a closed question (yes/no) and serves only to corroborate the previous responses.
- Question ***6E*** (Educational Use Questions) attempts to discover what it is about some of the existing products that makes them difficult for beginners to learn. This information is useful as I should aim to avoid implementing these features myself.
- Questions ***7E*** and ***8E*** enquire about the manner by which the clients currently interact with their programming languages. This is an interesting part of the overall problem, and affects the accessibility and ease of use of the system too.
- Question ***9E*** is an important one to ask, on the basis that the "educational use" clients are either looking to *learn* important programming concepts for themselves, or they will be teaching these concepts to others through the use of the programming language. It is therefore vital to establish which principles in particular ought to be prominent and well-implemented into the system.

I shall now ask the stakeholders each of these questions, and record their responses. *During* the interviews however, I may end up asking additional follow-up questions or requesting that a stakeholder substantiate one of their responses.

Responses

I conducted face-to-face interviews with Oliver and Kiran, whereas Joe was interviewed by Email, and I spoke to Klara over the phone in German, translating her responses into English for the purposes of this document. These, are the abridged findings…

1) Which areas of maths do you find yourself referring to most frequently, whilst programming?

- Just wanting to evaluate basic mathematical Expressions [2]
- Straightforward Boolean Logic (Kiran)
- Finding information from statistics; Mean, Mode, Median, Range etc. (Kiran)
- Geometry (Kiran)
- Manipulating Numbers in different Bases, and converting between them (Oliver)
- Bitwise Operations (Klara)

2) Which computer system(s) do you currently use (a) whilst programming, and (b) recreationally? What are the architectures and operating systems of these computers?

- An x86-Based IBM PC-Compatible Windows Computer [3]
- …Some of the stakeholders mentioned that their desktop computers are very powerful, with multiple logical processors, and even graphics coprocessors for "C.A.D. or video games". The versions of Windows NT ranged from XP to 10.
- An iPad or Tablet, iOS and Android (Recreational). Kiran commented that he often uses his iPad because it is "portable, quick, and easy to get out and use".
- (Occasionally) A SmartPhone Handset (Recreational) (Kiran)
- Enterprise Servers, running Windows Server. "KVM (Keyboard, Video, & Mouse) Console Access to the Server isn't always guaranteed, and we might only have a Command-Line Interface with the machine. The software might need to be able to handle this." (Klara)

3) Which components of existing programming languages or systems do you find annoying or cumbersome?

- Complex ways to perform ostensibly simple tasks. [2] Oliver mentioned the complexity of using iostreams and put-to operators in C++, just to output text to the console. Kiran mentioned the difficulty of navigating modern versions of Visual Studio, due to the large number of different windows and controls.
- Programs are often difficult to Debug; there is no clear trace of actions taken. (Oliver)
- The language being "pedantic" (Kiran) about small details which shouldn't have to make a difference, such as capital letters mattering for variable names.

4) Which components or features of existing programming languages do you like and find useful?

- Simplicity [2] and the organisation or encapsulation of parts of the source code (Klara)
- Some advanced high-level features such as (1) Immutability and (2) Lambda Expressions. These can help me to write safe and concise code. (Oliver)
- Something that allows the creation of elegant and even beautiful programs (Joe)
- Having lots of built-in libraries and functions "to do things for me" (Kiran)

5) …And what, therefore, would you perhaps like to see implemented in this system, to aid in its ease of use and functionality?

- "Potentially the ability to sub-divide the source code up into manageable blocks of some sort. Functions or even Namespaces would be a good idea, if a little challenging to implement" (Klara)
- A reasonable number of inbuilt functions and utility methods [2]
- Standardised and well-known procedural programming statements, which appear in other languages too. In this way, users of the language will "become familiar with common practice, easing their transition to other systems when they feel ready". (Kiran)
- Potentially some integrated *help*, if there are any more complex components in need of explanation.

6H) How frequently do you use programming languages for your own personal experimentational and hobbyist purposes?

Quite frequently	[3]
Occasionally	[1]

7H) Where do you think a reasonable line can be drawn between hindering simplicity and needless complexity, in the context of a programming language?

- "I think it's all about the structure of the system; if users can clearly see what to do, where they should enter text, or which button they should press, then there is no problem. Things tend to become needlessly complicated only when there is a poor system of organisation" (Oliver)
- Having an overly-large feature set can be too complex [2]
- Having consistency is important for the user experience [2]

8H) Is there a particular style of programming [paradigm, nomenclature, or simply a set of tendencies] that you lean towards, for experimentation?

- "I tend to want something quite high-level, with lots of pre-built functions available to me, so that I don't have to reinvent the wheel when it comes to doing something simple" (Oliver)
- "I don't like having to worry about small details when programming for experimentation" (Kiran)
- "I often find myself using an almost *functional* programming style, focused around expressions instead of statements" (Klara)
- "Choosing suitable names for the variables and other identifiers is often difficult, because you might not have fully conceptualised that that variable is actually for yet" (Joe)

9H) Are you satisfied by the programming toolset currently at your disposal?

No	[2]
Yes	[2]

6E) In your experience, what do you think makes some existing programming languages hard to learn?

- The existence of ambiguities to those who don't know the relevant rules. [3] Operator Precedence (order) and Associativity (right or left side resolved first) was mentioned by Klara. E.g. In "5 + 3 * 8 / 2 / 5", how does a beginner know whether the specific language will execute the multiplication, addition, or division first – and even if they know that, which of the two instances of the division operator would be evaluated first?
- The lack of important concepts being enforced at the basic level [3] E.g. "Python" doesn't make the user specify the DataType of a variable at its declaration. In fact, it doesn't even have a keyword for declaring variables. This makes the script very hard to follow. (Oliver)

- The specific syntactical and grammar rules [2]

7E) How do you interact with the Programming Languages you already use?

- With a Keyboard and Mouse [4]
- With Mouse via drag-and-drop blocks (Joe)
- I have to open an IDE (VS) and type the source into a file [2]
- "Sometimes I use the mshta.exe or cscript.exe to interpret VBScript and Jscript straight away, without having to create a project or anything first." (Kiran)

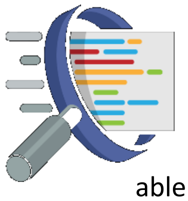
8E) What, if anything, do you find cumbersome about this, and how might you more ideally wish to use a programming language system?

- Example: "To experiment with some C++ I may have been thinking about, I have to open Visual Studio, create a new Project, type the source into the file, which I then have to save, before finally being able to run the script, only to be informed about a missing dll and having to start all over again. It would be nice to be able to simplify this process, but in the case of CPP, I think it's really just a fundamental limit of the language" (Oliver)
- Example: JavaScript's expression-based Console in Chrome facilitates the immediate evaluation of expressions and function calls. This is very useful for quick testing and experimentation. (Klara)
- I want a more *direct* way of getting the ideas in my head into code form. [2]

9E) What do you think some of the most important principles of higher-level programming languages are, that need to be learnt in order for students to progress onto systems requiring a deeper level of understanding?

- DataTypes [2]
- OS-Interaction; Exit Codes and Command-Line Arguments (Klara)
- Debugging principles; "getting a lower-level view at the high-level script you've written" (Oliver)
- Condition- and count-controlled loops [3], If statements [2], and Variable Declaration and Definition Clauses [2]

Analysing the Responses



Carrying out the interviews made it clear that, for the most part, the stakeholders' requirements for the application are actually not that complex or sophisticated. "Just wanting to evaluate basic mathematical expressions", for instance, was one of the key points from question 1. Stakeholders said that they just needed to be able to type in a quick mathematical statement (E.g. "400 – (80 * 6)") to work something out for the program they are currently writing. In other words, the stakeholders want to be able to use the programming language being developed here both as an assistive tool (for use whilst working with another programming system), and indeed as a stand-alone development environment for quick experimentation-style programs, and scripting tasks.

Several questions' responses seemed to indicate come common trends. For instance:

- The different mathematical use cases, along with the comments about "*simplicity coming through organisation*", suggest that it might be a good idea to have a system of modularity and classification in the programming language. Different components could be encapsulated into their own "boxes", making them easier to find and use.
- The clients mentioned wanting to have a number of predefined functions at their disposal, several times throughout the interview. This is important because it extends the functionality of the programming language, and facilitates the quick and experimentational style desired by the stakeholders, as mentioned in questions 8H and 5.
- The research into the varieties of computer currently used by the clients (and which they therefore would use the programming language on) suggests that a fairly wide variety of devices must be supported, including different operating systems and CPU Architectures (or indeed, *Instruction Sets*). These are MS Windows NT versions XP to 10 (KVM/GUI and 80-Column CLI), Apple iOS, Google Android, and both x86 and ARM Processors.
- The allegedly cumbersome ways of performing simple tasks in other programming languages will need to be abstracted with a layer of simplicity in this language. For instance, Input and Output functions could be built-in keywords or methods. The stakeholders also reported that they feel other languages can be needlessly "picky" or "pedantic" with features such as case-sensitivity (Kiran), operator associativity (Klara), and difficult debugging processes (Oliver)
- The reported frequency of use for the system (with 75% saying that they would use the language "quite frequently" (6H)) means that it will need to have a way to save any configuration and settings the user has applied, ready for the next use.
- Consistency and a structured organisation are a significant principle sought by the clients, as this aids in the ease of use, and extensibility of the system (the extent to which it can be extended, and interoperate with other programs).
- The stakeholders commented both on the fact that "*Having an overly-large feature set can be too complex*" and unwieldy, but at the same time, that "*Having lots of built-in libraries and functions*" is useful and improves the speed of development, wherefore I will have to strike a balance therebetween.
- The main factors affecting the ease of learning new programming languages (something a client of this system would inevitably have to do) seemed to be a lack of syntactical, grammatical, semantic, and conceptual consistency in some languages, and confusion about fundamental concepts, often because they are not enforced at a basic level. DataTypes in Python (or the lack thereof) was provided as an example. Operator Precedence and Associativity also seemed to cause confusion on occasion.
- Unsurprisingly, the clients reported interacting with the programming languages they used, by Keyboard and Mouse. They showed no real objection with this method, and acknowledged that it would be appropriate for the system being development here too.
- Use of IDEs was also mentioned, and it was said that having to open one up and setup a new project, just to write some code *can* be unnecessarily indirect, but that they do enable a feature-rich development experience, which would be hard to achieve otherwise. Having some sort of basic IDE – wherein the programmer can both write, execute, and debug their scripts – would be suitable for this programming language.

- In addition to this, however, it would be beneficial, according the stakeholder feedback, to have a more flexible, dynamic, and immediate method of executing or evaluating expressions or statements. This correlates with the comments made for the very first question concerning mathematical expression calculation being amongst the most sought features of the product.
- Finally, in order to better prepare educational users of the language for move advanced systems, it was established that the use of several fundamental principles had to be enforced in the language. These, the stakeholders reported, include DataTypes, Process Exit Codes, Command-Line Arguments, Debugging Principles, and standard procedural programming statements (Loops, If Statements, Variable Declarations and Definitions).

I acknowledge that there is a certain level of *irony* in the fact that I will be *using* a Programming Language inside of an IDE, to develop *another* Programming Language and IDE. This, however, makes it all-the-more entertaining.

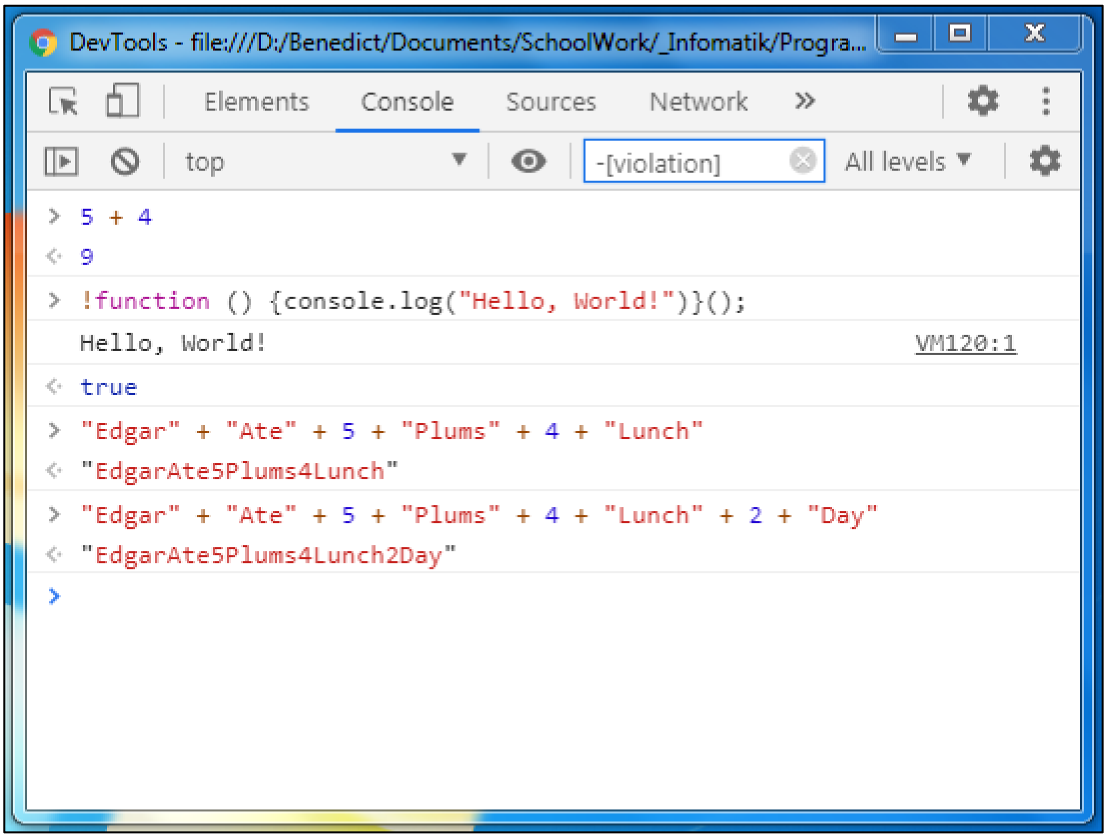
Existing Similar Products

1) Chrome's JavaScript Console

Since one of the interviewees mentioned this product as a system they currently use for experimentational programming, I have decided to take a closer look at it here.

Noteworthy Features

- The console is quickly accessible from within the browser; the user need only press "F12" to open the window whenever desired.
- Aside from being able to input meaningful, executable statements of JavaScript such as `var Age = 5;` or `if (IsOldEnough) { AllowEntry(); }`, the console also permits the programmer to simply enter *an expression*, and have it immediately resolved, based on whichever variables and functions are accessible in the current execution context. Several examples hereof can be seen in the screenshot of the product below.


DevTools - file:///D:/Benedict/Documents/SchoolWork/_Infomatik/Progra...
Elements
Console
Sources
Network
»
top
-[violation]
All levels ▼

```
> 5 + 4
< 9
> !function () {console.log("Hello, World!")}();
  Hello, World!                                              VM120:1
< true
> "Edgar" + "Ate" + 5 + "Plums" + 4 + "Lunch"
< "EdgarAte5Plums4Lunch"
> "Edgar" + "Ate" + 5 + "Plums" + 4 + "Lunch" + 2 + "Day"
< "EdgarAte5Plums4Lunch2Day"
>
```

[Above] The DevTools JavaScript Console

- Different parts of logic in JavaScript can be subdivided into different encapsulate units. For instance, Functions are supported both as conventional named methods, and as first-class objects for use with lambda expressions. This allows a series of instructions to be run sequentially, just by invoking the function. JavaScript additionally supports its own object mark-up format called JSON (JavaScript Object Notation), which further permits the programmer to neatly organise different data. E.g. `{ GetName : function () { return `Ben`; } , Age : 17}`.

- Google Chrome (which comes with this DevTools Console software) is available for almost all operating systems, though the company *is* becoming increasingly selective about supporting "*older*" operating systems such as Windows XP.
- The User Interface, though excessively *modern*, is largely comprehensible and clearly laid-out. Syntax Highlighting also aids in the readability and discernibility of the source code.

Limitations

- One *problem* with the system – just due to what it's actually intended for – is the inability to save and open script files of one's own (with any sort of ease).
- It also can't be used on all of the computer types the stakeholders mentioned; there is no Command-Line version, and Google Chrome does not include the DevTools Console on other devices such as iPads or Android phones.

Components Applicable to My Product

- The ability to evaluate standalone expressions (This appeals to the stakeholders' requests for a system that can be used for quick and easy experimentation)
- The almost instant accessibility of the console (This aids in making the product good for carrying out quick tests during one's own program writing)
- The relatively clear user interface, with its menus and syntax highlighting

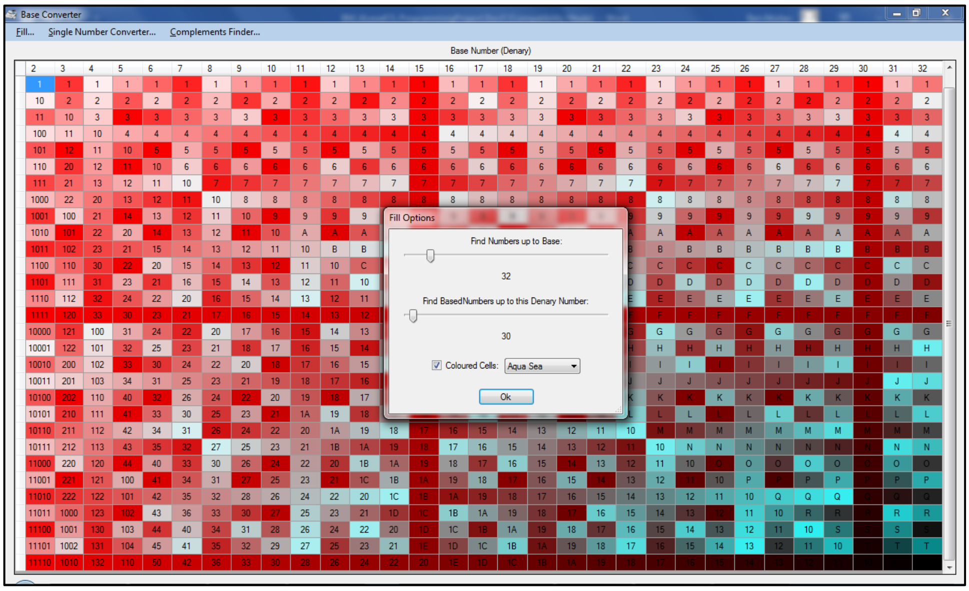
2) BaseConverter / BasedNumber Finder

This program's main feature is being able to compute tables of numbers in different bases. It doesn't have any sort of integrated programming language, but provides some of the mathematical capabilities the stakeholders are looking for.

Noteworthy Features

- Can easily convert numbers from most standard bases into other bases
- Has Complements Finder Built-in
- Includes several Colour-Formatting Algorithms to graphically visualise the results of a BasedNumber-Finding Query

[Below] The Main Window with the *Fill* dialog open



Base Number (Denary)

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
11	10	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
100	11	10	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
101	12	11	10	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
110	20	12	11	10	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
111	21	13	12	11	10	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
1000	22	20	13	12	11	10	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
1001	100	21	14	13	12	11	10	9	9	9	9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	9	9	9	9	9	9	9	9	9	9	9
1010	101	22	20	14	13	12	11	10	A	A	A	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	A	A	A	A	A	A	A	A	A	A	A
1011	102	23	21	15	14	13	12	11	10	B	B	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	B	B	B	B	B	B	B	B	B	B	B
1100	110	30	22	20	15	14	13	12	11	10	C	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	C	C	C	C	C	C	C	C	C	C	C
1101	111	31	23	21	16	15	14	13	12	11	10	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	D	D	D	D	D	D	D	D	D	D	D
1110	112	32	24	22	20	16	15	14	13	12	11	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	E	E	E	E	E	E	E	E	E	E	E
1111	120	33	30	23	21	17	16	15	14	13	12	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	F	F	F	F	F	F	F	F	F	F	F
10000	121	100	31	24	22	20	17	16	15	14	13	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	G	G	G	G	G	G	G	G	G	G	G
10001	122	101	32	25	23	21	18	17	16	15	14	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	H	H	H	H	H	H	H	H	H	H	H
10010	200	102	33	30	24	22	20	18	17	16	15	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	I	I	I	I	I	I	I	I	I	I	I
10011	201	103	34	31	25	23	21	19	18	17	16	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	J	J	J	J	J	J	J	J	J	J	J
10100	202	110	40	32	26	24	22	20	19	18	17	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	K	K	K	K	K	K	K	K	K	K	K
10101	210	111	41	33	30	25	23	21	1A	19	18	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	L	L	L	L	L	L	L	L	L	L	L
10110	211	112	42	34	31	26	24	22	20	1A	19	18	17	16	15	14	13	12	11	10	M	M	M	M	M	M	M	M	M	M
10111	212	113	43	35	32	27	25	23	21	1B	1A	19	18	17	16	15	14	13	12	11	10	N	N	N	N	N	N	N	N	N
11000	220	120	44	40	33	30	26	24	22	20	1B	1A	19	18	17	16	15	14	13	12	11	10	O	O	O	O	O	O	O	O
11001	221	121	100	41	34	31	27	25	23	21	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	P	P	P	P	P	P	P
11010	222	122	101	42	35	32	28	26	24	22	20	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	Q	Q	Q	Q	Q	Q
11011	1000	123	102	43	36	33	30	27	25	23	21	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	R	R	R	R	R
11100	1001	130	103	44	40	34	31	28	26	24	22	20	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	S	S	S	S
11101	1002	131	104	45	41	35	32	29	27	25	23	21	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	T	T	T
11110	1010	132	110	50	42	36	33	30	28	26	24	22	20	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	[illegible]	[illegible]

Fill Options

Find Numbers up to Base:

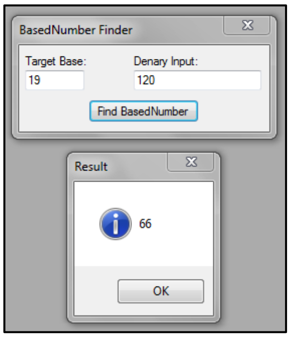
32

Find BasedNumbers up to this Denary Number:

30

☑ Coloured Cells: Aqua Sea

[Below] Additional Program Features: The BasedNumber Finder, and Complements Finder



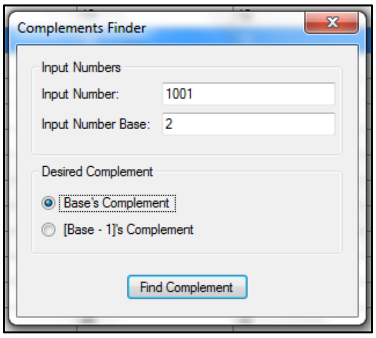
BasedNumber Finder

Target Base: 19

Denary Input: 120

Result

66



Complements Finder

Input Numbers

Input Number: 1001

Input Number Base: 2

Desired Complement

● Base's Complement

○ [Base - 1]'s Complement

Limitations

- There is no extensibility with the system; only the functionality hard-coded into it can be run.
- The current version of the Program only runs on Windows (over .NET).

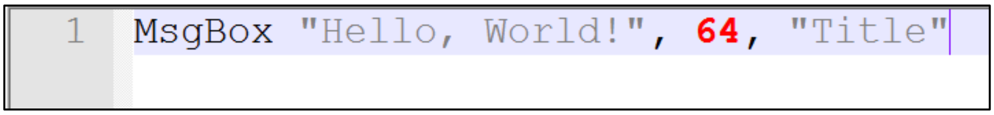
Components Applicable to My Product

- The mathematical features of converting between numbers of different positional notation (place value) bases.
- The clear User Interface with the MenuStrip at the top.
- The fact that the program is just a singular exe file; it needn't be installed with a setup utility.

3) CScript.exe / WScript.exe

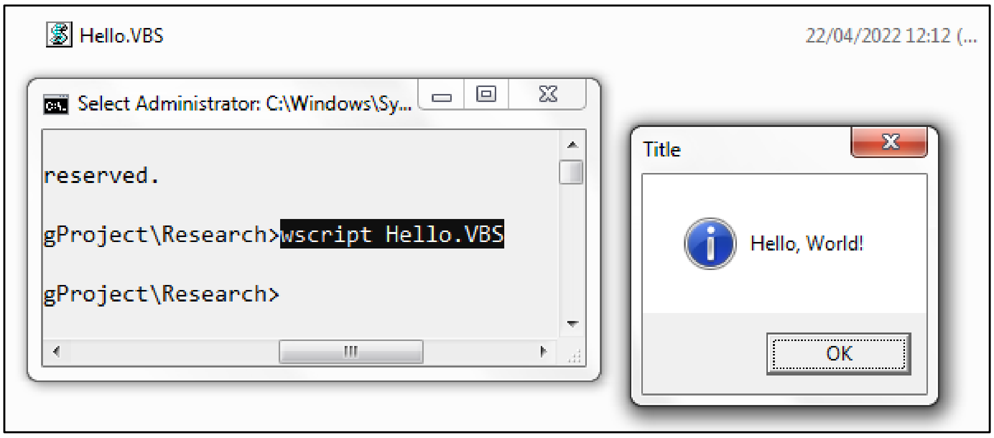
Microsoft Windows includes a few built-in script interpreters for different languages, including Batch, PowerShell, VBScript, and JScript. One of the clients mentioned that they use mshta.exe and cscript.exe to run these sorts of scripts quickly and easily, so that they don't have to create entire projects for a quick experimentational test. Therefore, I will take a closer look at these interpreters here. Incidentally, "*wscript.exe*" is the Windows Script Host (WSH) component for running scripts using graphical input and output, whereas "*cscript.exe*" is for running the same scripts in a command-line environment.

[Below] "*Hello.VBS*" open in a Text Editor



```
MsgBox "Hello, World!", 64, "Title"
```

[Below] Using "*wscript.exe*" to execute the Script File



```
reserved.

gProject\Research>wscript Hello.VBS

gProject\Research>
```

Hello, World!

Noteworthy Features

- Can easily be used run saved script files in either a windowed or command-line environment.
- Is widely-available because it comes with the Operating System.
- Is extensible, as it even works with multiple programming languages (VBScript & JScript).

Limitations

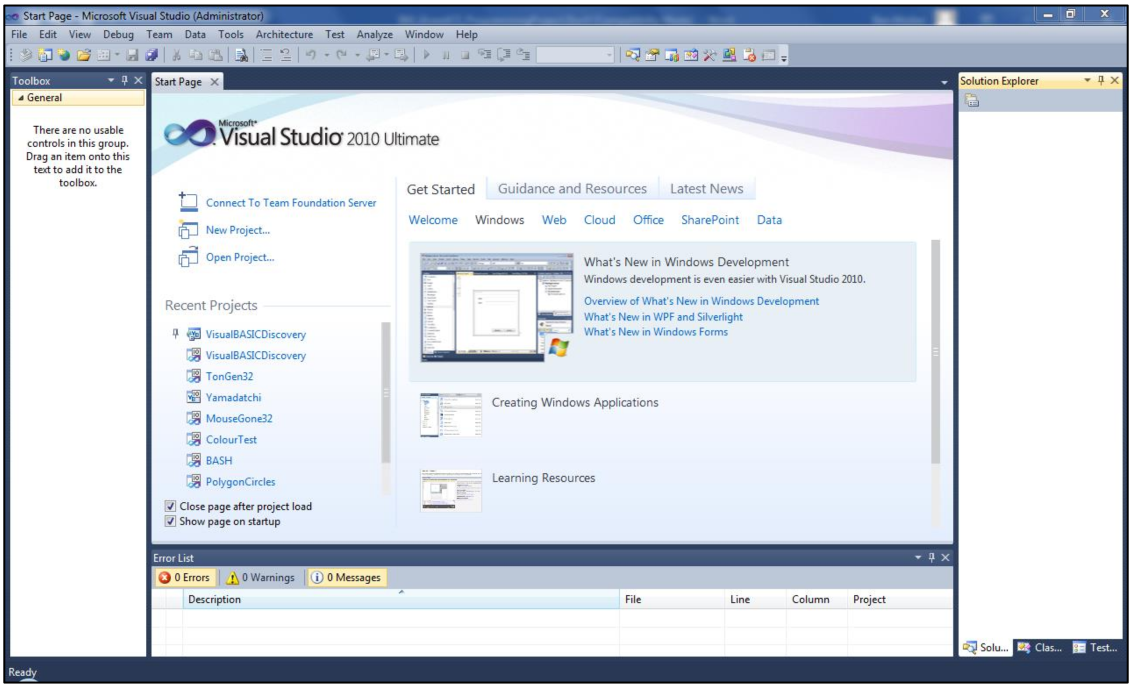
- The script files must first be created and saved to disk before they can be run. This can be annoying if the user wants very quick access for testing a line of code.
- Despite working on all versions of windows, it does not exist for other operating systems such as iOS and Android.
- There is no convenient and integrated way to *edit* the scripts. One has to use *another* program to accomplish this task.

Components Applicable to My Product

- Different components to execute scripts in different environments including both GDI-enabled and Command-Line situations.
- Interoperability with standardised operating system mechanisms.

4) Visual Studio

Microsoft Visual Studio is an industry-standard IDE used worldwide by millions of software developers. It too was mentioned several times throughout the interviews, and as such, I shall evaluate some of its features and functionality.



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

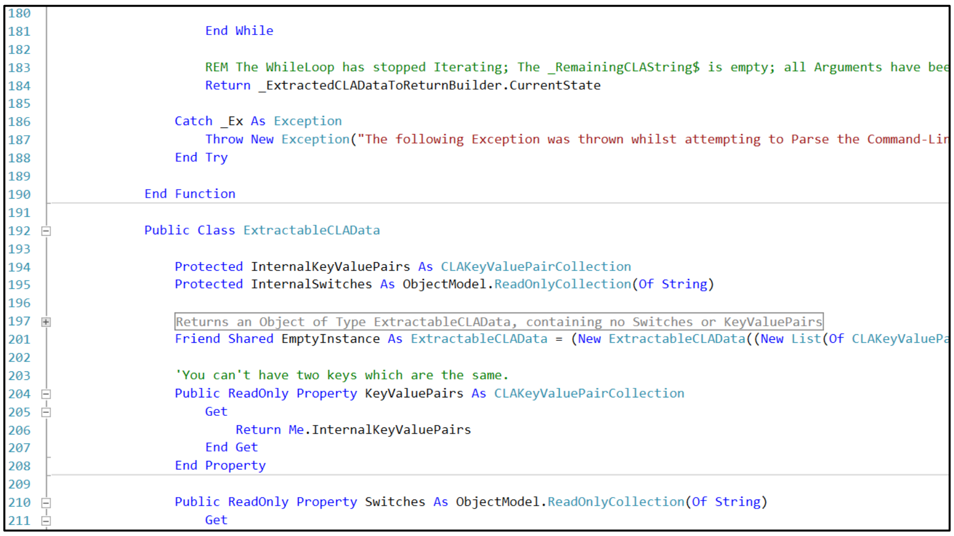
Windows development is even easier with Visual Studio 2010.

[Above] The Main Window with the *Start Page* open

Noteworthy Features

- Advanced debugging features including: Realtime Memory contents and Variables view, Output Window, Errors List, Remote Debugging Server, .NET Exception Handling, and JIT-Debugging
- Highly-customisable and versatile
- Helpful Syntax Highlighting and Keyboard shortcuts
- Well-designed, clear user interface, including identifiable coloured icons, and buttons with self-evident functions.

[Below] Visual Studio's Syntax Highlighting



```
        End While

        REM The WhileLoop has stopped Iterating; The _RemainingCLAString$ is empty; all Arguments have bee
        Return _ExtractedCLADataToReturnBuilder.CurrentState

    Catch _Ex As Exception
        Throw New Exception("The following Exception was thrown whilst attempting to Parse the Command-Lir
    End Try

End Function

Public Class ExtractableCLAData

    Protected InternalKeyValuePairs As CLAKeyValuePairCollection
    Protected InternalSwitches As ObjectModel.ReadOnlyCollection(Of String)

    Returns an Object of Type ExtractableCLAData, containing no Switches or KeyValuePairs
    Friend Shared EmptyInstance As ExtractableCLAData = (New ExtractableCLAData((New List(Of CLAKeyValuePa

    'You can't have two keys which are the same.
    Public ReadOnly Property KeyValuePairs As CLAKeyValuePairCollection
        Get
            Return Me.InternalKeyValuePairs
        End Get
    End Property

    Public ReadOnly Property Switches As ObjectModel.ReadOnlyCollection(Of String)
        Get
```

Limitations

- The software only runs on x86-based Windows (or – ostensibly – Macintosh) computers as a full desktop application with the GUI. At least two of the stakeholders commented that they may need to be able to use [the solution being developed herein] in a remote command-line, or mobile environment.

- Arguably, although Visual Studio provides an indispensable toolset for any programmer once understood, it can be challenging for a beginner to get to grips with.

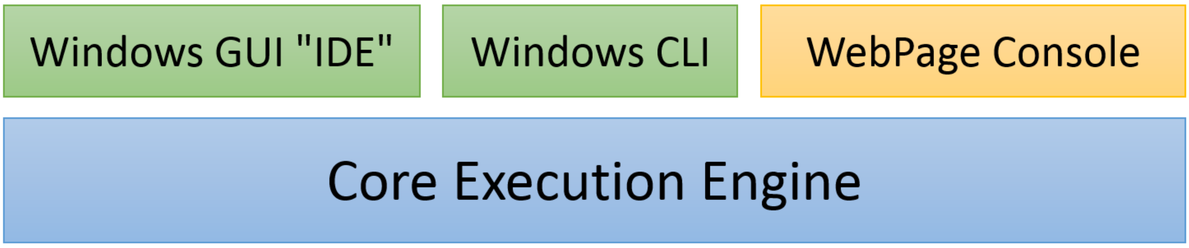
Components Applicable to My Product

- Clearly, whatever product *I* end up producing will not be comparable in its feature-set to the likes of Visual Studio; the scales of the two products are not even on the same order of magnitude, and I am not *a professional team of 500 full-time developers*.
- However, an appropriate approach for this project is to realise that many developers (and therefore stakeholders in this product) will be accustomed to programs such as Visual Studio, and that the product being developed, should not – therefore – differ so wildly from this as to be unfamiliar and hard-to-use.

Features of the Proposed Solution

The core component to be developed is an *engine of execution* for a Programming Language. A carefully-considered set of rules concerning syntax, grammar, and structure will be created as a formal specification for the programming language. The core "*engine*" component will need to be able to take in some source code – compliant with this specification – and execute it.

This *engine* will then need to be implemented into a number of different execution environments – namely, an IDE-resembling windows program with a GUI, a command-line script runner, and a web-based console for entering and running code on, for – importantly – *any device with a web browser*.



Only this *segmented* and *hierarchical* architecture can facilitate the use of the language on all the platforms sought by the clients. The system will also include a range of specialist inbuilt **mathematical features**, which, as a result of having conducted the interviews, can be delineated (mostly) as the following:

- Evaluating standard mathematical expressions (e.g. "*5 + 2 * 9*")
- Working with numbers in different bases
- Boolean Logic operations

The research has also made it clear that the following **programming features** ought to be implemented into the language:

- Data Types for Variables
- Functions, to facilitate the encapsulation of logical executable sections
- Commonplace procedural constructs including *While* and *If* statements

- OS Interoperability (Command-line Arguments, and Process Exit-Codes)
- Predefined functions for common tasks

These features have been chosen because the stakeholders placed an emphasis on desiring *simplicity* in the system. One way in which this is achieved is with structure and organisation, wherefore *Functions* and block-style statements (including [*While*] and [*If*]) will be features of the language.

Features to facilitate "*Experimentational Programming*" are important in this product because this is one of the primary use cases of it, along with its use in an *educational* capacity. For this reason, the [built-in mathematical functions] and [predefined functions for common tasks] are on the list above too.

Limitations of the Proposed Solution

Owing to the amount of time at my disposal to create this programming language, the main limitation is likely to be the **breath of the solution**. (E.g.) How many built-in functions will be available? Will there be additional encapsulative features such as namespaces? How many complex operators can be used for the mathematical expressions? Nevertheless, the requirements listed in the subsequent sections take into account the feature-set of an MVP (Minimal Viable Product), which still meets the stakeholders' needs and is founded on the research conducted.

Another limitation is likely to be an intrinsic level of *some* **complexity** in the system, despite the requirement that it be simple. This is because any worthwhile programming language must be sufficiently complex as to enable the programmer to develop at a reasonable pace, once familiar with the system. At the same time, *this* language has the paramount requirement that it be simple to use and learn. The compromises involved in meeting a balance herebetween are likely, therefore, to constitute a limitation of the product.

Having to interact with the programming language in a conventional, predominantly keyboard-based fashion (which was agreed upon by the stakeholders to be suitable) does of course mean that **people with certain disabilities**, which make it difficult for them to type, may be unable to make full use of the software. This problem does not, however, directly affect any of the stakeholders to whom I have hitherto spoken.

Since **I have not implemented any complex expression parsing** in my programming before, this is something I shall have to learn much more about. I have some initial ideas about how this can be done with infix to postfix conversion, and stack-based execution, but the complexity of expressions supported in the language will be to some extent dependant on my ability to write an expression parser and evaluator of sufficient complexity, dealing with such difficulties such operator overloading, precedence, and associativity, as well as brackets, unary operators, and embedded function calls.

There *may* also be some **security concerns** with the language's runtime engine; since it is a system of such complexity, and because there are a very large number of edge cases and unaccounted-for pieces of input, certain scripts may potentially cause insecure behaviour such as buffer overflows or injection. However, since the system will run on top of .NET (clarified just below), for there to be any serious vulnerability permitting system-wide damage, a vulnerability would need to exist in .NET itself: something which is seldom discovered, and rapidly patched.

Hardware and Software Requirements

I have settled on using the .NET framework to build this project. There are many programming languages which can be used with this framework, but I will predominantly – if not exclusively – be using *Visual B.A.S.I.C. .NET*.

This decision affects the software requirements for components of the proposed solution, inasmuch as only x86-Based Windows NT computers will be able to *host* the "*engine*", but it still means that the stakeholders who sometimes use mobile devices will be able to load the interactive Webpage as a means of accessing the Programming Language.

Hardware

- For the two Windows Implementations: an x**86-Based** [IBM PC]-Compatible ACPI Computer, with standard HID peripherals including **a Keyboard**, optionally a Mouse, and **a Monitor**.
- The .NET Framework (v4) also requires that the computer have at least a **1GHz Processor**, **512MB of Memory**, and **5GB of free Disk Space**.
- The Web Client only requires that the computer is capable of running a standard (modern) **Web Browser**, such as Google Chrome v80+.

Software

- The aforementioned computer running, or being able to run, a version of Microsoft **Windows** NT, versions **XP** to **7**, with the **.NET Framework**, version 4.0 or higher, installed.
- The Web Client will simply require a (Modern) Web Browser to be installed onto the computer, supporting client-side scripts, and AJAX. Specifically: HTML5, CSS3, and ECMA6.

Stakeholder Requirements

Further Meeting with the Stakeholders

I sent this Email to the stakeholders, to get them up-to-speed with some of the developments which have occurred since the interviews. I was also able to speak individually with two of the four primary users.

> *Dear Stakeholders,*
>
> *I have been analysing the feedback form the interviews we conducted, and have worked out what some of the features of the end product will be:*

> - *An implementation of the Programming Language Runtime for both Windows GUI, Windows CLI, and Web-Based clients*
> - *A basic IDE with text-editing and script-executing abilities*
> - *Built-in mathematical functions for based numbers, expressions, and Boolean logic*
> - *Standard procedural features including While and If Statements*
> - *DataTypes for variables*
> - *Functions, for encapsulating logic into sections*
>
> *The Runtime ("engine") for the language will be able to run only on x86 Windows over .NET, but the inclusion of the Web Client will enable a version of the language to run on any computer with a Web Browser, including of course the mobile devices you mentioned in the interviews.*
>
> *Ben*

The stakeholders were pleased to hear about this progress, and had no objections to the features listed. Their requirements for the product are therefore as follows…

Requirements for the Programming Language

Requirement	Explanation & Justification
An easy-to-use and learn **programming language**, suitable for pedagogical use	To meet the needs of the clients, and be of any real value at all, the system must be relatively easy to use and pick up. This also aids in making the product useful as an educational tool.
Implements the requested specialist **mathematical features**; BasedNumber manipulation, Boolean Logic, Mathematical Expression Evaluation	The primary point of specialisation for the language is its integrated mathematical abilities
The concept of **DataTypes** is enforced	This concept is a significant idea for anybody learning computer programming, and since two of the stakeholders represent the product's use in an educational capacity, this feature is important
Allows the programmer to use standardised, commonplace **procedural programming constructs** including *While* and *If* Statements	This is important both for the educational use cases, and the experimentational ones. Experimentation can occur more easily when the user is already familiar with some of the features of the language, in this case because these same

	features are implemented by many other programming languages.
Enables the programmer to **evaluate singular expressions** without having to run an entire program	Such functionality would permit the user to program in the desired "*experimentational*" manner
Permits the programmer to divide programs up into more manageable and organisable sections, E.g. via **Functions**	The clients said that one way in which simplicity (another requirement) can be achieved, is through organisation and structure
The language is able to interoperate and communicate with the operating system, E.g. via **Command-Line Arguments, and Process Exit Codes**	This concept is another one which assists newcomers to programming, teaching a useful and important OS mechanism
There are some useful **pre-defined functions** for common tasks	This facilitates the experimentational programming style, as well as making it easier for any learners to more quickly write a program to accomplish the task in question

Requirements for the Implementations

Requirement	Explanation & Justification
Runs under different OS environments and computer types	The stakeholders reported using several different computer types in the interview. The system will need to support these.
[Windows GUI] Has a familiar, simple, unconvoluted, User Interface with helpful icons, and buttons with self-evident functions.	This contributes to making the software easy to use, and organised
Some documentation and instructions concerning use of the language. This is applicable to all implementations, though there may be some minor differences between, for instance, the debugging features available on the Web Client, and those of the Windows GUI Client.	In order to make the system easy to use and learn, users must have appropriate, comprehensive, and *understandable* documentation at their disposal
[Windows GUI] The IDE should be able to edit the text of a source file, and run it, from the same Window or Program.	This integration provides convenience to the user
[Windows CLI] The executable should accept Command-Line Arguments (CLAs) as a means of specifying the script to be run (E.g. `Interpreter.exe /Run:Script.Ext`)	This is a standardised OS mechanism, thereby bringing familiarity to the user, and enabling programs written in the language to be executed automatically, E.g. by task scheduling in

	Microsoft® Windows™

Success Criteria

Criterium	How to Evidence This
A Programming Language Formal Specification (*LangSpec*)	Specification Document
Serviceable **Language Features** including:	Screenshot & Source & Binaries
Data Types	Screenshot & Source & Example
Predefined Functions (for common tasks)	Screenshot & Source & Example
Encapsulative Units (E.g. Functions)	Screenshot & Source & Example
Procedural Programming Constructs (While; If; etc…)	Screenshot & Source & Example
OS Interoperability (CLAs & Exit Codes)	Screenshot & Source & Example
Specialist **Mathematical Features**:	Screenshot & Source & Examples
BasedNumber Manipulation	Screenshot & Source & Example
Boolean Logic features	Screenshot & Source & Example
Maths Expression Evaluation	Screenshot & Source & Example
A Programming Language Runtime (*engine*) which can execute Scripts	Screenshot & Source
[Windows GUI] An IDE with text-editing and script-running abilities	Screenshot & Source & Binary
[Windows GUI] A simple, familiar graphical design	Screenshot
[Windows CLI] Takes a Command-Line argument for the script to run	Screenshot & Source & Binary
[Windows CLI] Returns the Exit Code of the Script just run	Screenshot & Source
[Web Client] Runs on a variety of different browsers on different devices	Screenshots
[Web Client] Allows the user to enter source code into a text field and thereafter execute it	Screenshot & Source

Singular Expressions can be evaluated independently (without having to run a whole script)	Screenshot
Instructions and Documentation are organised and easy to find	Screenshot
The Language is easy to learn and use	Get the stakeholders to learn and use the language, measuring how quickly they become accustomed to *it*, compared to other similar languages

These tables will be revisited at the end of §Development, when I have a prototype product against which to use the criteria.

Design

Overview

This programming language project fundamentally involves **two stages**:

1) **Delineate a Formal Language Specification** for the Programming Language (in accordance with the needs of the Stakeholders, and Requirements of the Hardware, Software, and Use cases)
2) **Implement a system to execute any programs** written *in* the Programming Language, and which comply to its specification. This system needs to run on all the computer types declared by the stakeholders during the investigations (x86 Windows Computers for GUI and CLI, and android mobile devices)

It would be impossible to perform the latter without having completed beforehand the former; a runtime engine cannot be written without knowing what it is supposed to be running, and what specific form and syntax its source code is supposed to be compliant with. Firstly therefore, I shall determine the nature of the language, and thereafter, discuss its architecture (*structure*), and the *computational methods* I will employ to execute it.

Formal Language Specification

Here I officially outline the specification for the Programming Language to be implemented. The language is a ***Formal*** one which means that it is ***concise*** (= succinct) and ***unambiguous*** (= never equivocal in meaning).

Choosing a Name

Before I go any further, it is important that I have a proper name for this project, because this will help in labelling and identifying files and resources associated with the project, which improves organisational efficacy.

I have settled on the name "***DocScript***" for the Programming language. "*Doc-*" comes from the Latin "*doctus*" meaning "*to teach*", as in *doctorate* or *indoctrinate*. Since one of the primary purposes of the language is to be an effective and approachable teaching tool, I feel that the name is at least somewhat fitting, and not just chosen out of the blue. *This claim – I fear – cannot justifiably be made of most modern products and services.*

The file extension for DocScript source is therefore "***.DS***".

General

Universal characteristics of the Programming Language

Spec. Point	Explanation & Justification
The language is **not case-sensitive** (apart from inside String Literals)	The stakeholders for the educational and hobbyist use cases wanted an easy-to-use programming language. Programming languages wherein "True" is a valid Boolean literal but "true" is not, are simply frustrating and do not lend themselves to a worry-free style of

	development. In fact, some such case-sensitive languages encourage programmers to use ambiguous and confusingly-similar variable names such as “name” for a constructor argument and “Name” for a corresponding Class Property member. I resent this nomenclature as it is needlessly unclear for beginners, which is **precisely what DocScript is trying to avoid**.
There are three **DataTypes**; *String*, *Number*, and *Boolean*	The stakeholders wanted to use the language for pedagogic purposes, wherefore it is important that the language implements some features found in more advanced languages – for instance DataTypes. Without DataTypes, a variable or expression’s value can be open to interpretation, and developing *knowing* the variety of data wanted by a Function or Operator is significantly easier and provides a framework. On the other hand, many programming languages implement a much more strict and diverse type system than DocScript will. Whilst it could be argued that more data types would be better, this would also hinder and **add unwarranted complexity** to the solution. There is an elegance to being able to perform a wide range of tasks with only a few essential tools (or in this case, DataTypes). These three specific DataTypes are rather essential and can essentially be used together to represent any type of data desired. Each of the Types has a corresponding literal (elucidated later). All Variables have a default value, according to their DataType: • `<Number>    0` • `<String>    ""` • `<Boolean>   False` • `<*@>          (An Empty Array)`
One-dimensional **Arrays** are supported for each of the DataTypes	Because it is common to have to process a list or collection of items in a program, having arrays makes a programming language more extensible. The stakeholders are seeking a versatile system, so this is a suitable feature. To retain the simplicity of the language however, I will keep the bounds of these Arrays to one dimension. This prevents arguably unnecessary complexity in the implementation runtime too.
There is a high degree of **verbosity and logging** during the execution of Programs written in DocScript.	This greatly aids in debugging both for users of the language in debugging their own programs, and for me whilst writing the runtime. It also means that if a runtime error were to occur, the user would essentially have a detailed and highly-verbose stack trace telling them exactly what had happened thitherto to cause the error.
Executable statements and instructions are contained in **Functions**	The stakeholders seek a modular and compartmentalisable system, so having all a program’s content in a singular file would encroach upon this requirement. Therefore, being able to have multiple Function Statements per file allows for better organisation.

	Functions can take in a series of Arguments (each which its own DataType) and return an instance of one of the DataTypes, or, return "*void*" to indicate that the Function was a SubRoutine. This principle of returning *void* is commonplace and can be found in other languages such as C, C++, Visual C# .NET, and Java. Thus, it is another example of how DocScript prepares its programmers for eventual graduation to more advanced programming languages.
A **Program** is a collection of Functions and Global Variable Declarations.	Global Variables make it easier to have shared data between Functions, and the Stakeholders seek an easy-to-use and quick development platform.
There must be one **EntryPoint Function** per DocScript Program	Since a DocScript Program contains a List of Functions (along with zero or more Global Variable Declarations), one of those functions must be executed first. The Function has the identifier "*Main*" and must have one of the following Function Signatures: • `Function <Void> Main ()` • `Function <Number> Main (<String@> _CLAs)` The principle of EntryPoints can be found in languages including C, Visual B.A.S.I.C. .NET, and even Python to an extent. It is another example of how DocScript prepares its programmers for more advanced programming languages, which was one of the stakeholders' desires.
Programs can take in **Command-Line Arguments** (CLAs) and return an **Exit Code**	One of the Stakeholder requirements was: "The language is able to interoperate and communicate with the operating system, E.g. via **Command-Line Arguments, and Process Exit Codes**". Because this is a standardised Operating System Mechanism, it is important that a teaching tool such as DocScript enables this mechanism to be taught, by implementing it!

Basic Interaction occurs through the built-in **Input()** and **Output()** Functions	Example:
	`Output("Hello, World!")`
	`# Returns the Input from the User:`
	`Input("What is your Name?")`
	These provide a simple and easy means of obtaining Input and delivering Output to the user. One of the Stakeholder requirements was: "There are some useful **pre-defined functions** for common tasks".
	Therefore, by having these axiomatically-essential functions, basic operations in the language become possible. The obvious names also aid in meeting the Stakeholder spec of the language being easy-to-use.

Syntax

Physical form and grammar used in DocScript Source

Spec. Point	Explanation & Justification

The **Keywords** are:
 	
`Function`
`If`
`Else`
`While`
`Loop`
`Return`
	The stakeholders for the educational and hobbyist use cases wanted an easy-to-use programming language, so having an immemorable litany of keywords (as seen in the likes of modern-day C++ with its 95) would be herefor unsuitable and unnecessary.
	
	These six keywords are enough to facilitate fully **Turing-complete** functionality, but there are also some simply nice-to-have features here such as the Loop keyword which works like this…
	
	`Loop (10)`
	`    Output("Hello, World!")`
	`EndLoop`
	
	…to output <String> "Hello, World! " 10 times. This would otherwise have to be done by manually setting up an iterator and interatee variable with a While Statement. Therefore, this is a language feature which makes DocScript suitable for the "experimentational" style of programming desired by the stakeholders.
	
	One of the Stakeholder requirements was:
	"Allows the programmer to use standardised, commonplace **procedural programming constructs** including *While* and *If* Statements"
	
	This is hereby facilitated through the If, While, and Loop Keywords.

Statements are closed with an **End{Statement}** block:
 	
`EndFunction`
`EndIf`
`EndWhile`
`EndLoop`
	Example:
	
	`Function <String> GetName (<Boolean> _WithSurname)`
	`    If (_WithSurname)`
	`        Return "Ben Mullan"`
	`    Else`
	`        Return "Ben"`
	`    EndIf`
	`EndFunction`
	
	This is a more suitable means of Statement closure than (E.g.) a closing curly bracket (Brace) **}** because it enables the programmer to see which Statement type if actually being closed. At the end of many C-style language programs, one is often greeted with the following atrocity:
	
	`                    }`
	`                }`
	`            }`
	`        …Another 10 Lines of this barbarism…`

	`    }`
	`}`
	
	In a language such as Visual B.A.S.I.C. .NET however, the same section looks like this…
	
	`                              End Get`
	`                        End Property`
	`                  End Class`
	`            #End Region`
	`End Namespace`
	
	…which is significantly more useful and readable. Therefore, DocScript takes a leaf out of this book, and uses the Statement Ends following the pattern End{Statement} E.g. **EndLoop**.

Comments are specified with a **#** at the start of a Line	Example
	
	`# The User’s Domain Username`
	`<String> UserName : "MullNet.NET\Ben"`
	
	Having comments at the source level is a much-needed feature of any language because it permits the programmer to annotate and clarify components of the script. This is suitable because the Stakeholders require a tool for teaching, so they’ll be writing scripts and revisiting them in due course, and needing to remember what the script does. This can be achieved by adding a comment at the top of the script to describe the file’s purpose. Incidentally, comments can only befall at the start of a line, not mid-way through one.

DataTypes are specified in Pointy Brackets **< >**
 	
@ is used to indicate that a type is an **array**
 	
`<String>`
`<Number>`
`<Boolean>`
`<Void>`
`<String@>`
`<Number@>`
`<Boolean@>`
	The char “@” was chosen on account of it resembling an “a” as in “array”. It is not a valid identifier char, so is always unambiguous and never part of the identifier (or in this case, DataType).
	
	Of course, there couldn’t possibly be a “`<Void@>`”, and `<Void>` is only valid as a return type form a Function, not for a Variable’s Type, because it represents the *absence* of data.
	
	Other languages use < > with relation to DataTypes too, so this is fitting and readies the programmers mind for exposure to alternative programming languages too.

Square Brackets **[ ]** are used to **increase the precedence** of an Expression Section	Example: `Output(5 + [2 – 6 * [7 ^ 7]])` This is a very necessary feature, because the Stakeholders need standard mathematical features, of which this is one.
The same character is	Example:

<table>
<tr>
<td><b>never used for more than one purpose</b></td>
<td>( )   Function Call & Declaration
[ ]   Expression encapsulation (Often ( ))

&   Concatenation Operator
'   Logical And Operator (Often &)

=   Comparison Operator
:   Assignment Operator (Often =)

By using different characters, the benefits are two-fold:
• The programmer/learner realises that the two elements are distinct and serve subtly or wildly different purposes. Whenever they see an instance of the character, they immediately know precisely what it’s being used for.
• I, the writer of the parser and lexer for the language, get to have an easier time determining if everything is in the right place in the source. Where ever there is a “<b>(</b>“, there should always be an identifier directly before it. In other languages such as C, these same brackets ( ) are also used for expressions so there isn’t the same level of certainty and regularity.

Of particular importance is the distinction between the equality and assignment operators. These serve dissimilar purposes and yet have sometimes the same char (E.g. “<b>=</b>” in Visual B.A.S.I.C. .NET). The language clearly differentiates these at the syntactical level, which meets the stakeholder need of creating an effective teaching tool.

This is therefore a <b>usability feature</b>.

Parenthetically, the Character $ is reserved for internal use by the Parser, and is not valid as an Identifier, Operator, or Grammar Char.</td>
</tr>
<tr>
<td>There are multiple <b>manifestations of numeric literal</b>:

9637426877
6543482740.9902
3AB4FF_16</td>
<td>One of the stakeholder requirements was:
“Implements the requested specialist <b>mathematical features</b>; BasedNumber manipulation, Boolean Logic, Mathematical Expression Evaluation”

To achieve this, I am therefore allowing numbers of different bases to be directly represented through literals. This greatly improves the ease with which numbers of different bases can be dealt with in DocScript programs; no specialise function call is needed.

For instance, to represent the number 12_{10} in base 10, the programmer types 12 or 12.0 or 12_10. To represent 12_{10} in base 2 however, the programmer can type 1100_2.

All numeric literals must therefore match this <b>Regular Expression</b>:
<b><i>^((\d{1,10}(\.\d{1,4})?)|[A-Za-z0-9]{1,10}_\d{1,3})$</i></b></td>
</tr>
<tr>
<td><b>Identifiers</b> can only</td>
<td>All Variables and Functions have a name associated with them, called</td>
</tr>
</table>

contain alphabetic characters and underscores	an Identifier. In DocScript, identifiers must match this Regular Expression: ***`^(_?[A-Z]+[A-Z_]*)$`*** This reduces ambiguity because identifiers cannot contain digit (`/\d/`) chars, meaning they are impossible to confuse with numeric literals. If identifiers *could* contain digits, then “A3B4FF_16” would be valid as both an identifier, *and* a numeric literal. This would mean the language were no longer unambiguous, which is not permitted in this *formal* programming language.

LineBreaks are significant, but **Tabs** and superfluous **Spaces** aren’t	Because instructions end on LineBreaks instead of (E.g.) semicolons in DocScript, LineBreaks are semantically-significant. However; this line…
	`<String> Name : GetName()`
	…and this one…
	`< String >  Name:GetName()`
	…are semantically identical despite their syntactical differences.
	This is done to make the language more permissive and tolerant of minor syntactical oddities or mistakes made by the programmer. This aids in meeting the stakeholder requirement of being an effective teaching tool.
	This is important, because one of the stakeholder requirements was: “An easy-to-use and learn **programming language**, suitable for pedagogical *(=teaching and instructional)* use”.

Operators

= Built-in useful logic used in expressions along with their Operands…

Research

To begin defining the fashion in which DocScript Operators should work, I have conducted some in-depth research on Operators...

Characteristics

Overloading

= when the same operator notation has multiple has multiple logical definitions and can therefore do different things, depending on how it appears syntactically.

Example

```
# Subtraction:
5 - 1
# Polarity Inversion:
-9
```

Associativity

= when operators of the same Precedence appear in the same bracketed level, the Associativity governs whether to evaluate from left to right or vice versa. Left-associative means from left-to-right.

Example

```
# Left-Associative
<Number> A = 96 / 8 / 4
# Same as:
<Number> B = (96 / 8) / 4
```

Precedence

= when one operator is executed before another different operator, despite the two being on the same bracketed level. Each operator has it's own precedence. Operators with a *higher* precedence are executed first.

Example

```
5 + 3 * 2
#Answer: 11
( 5 + 3 ) * 2
#Answer: 16
```

Commutativity

= when the operands of an operator can be swapped, without changing the result.

Example

```
8 * 4
#Answer: 32
4 * 8
#Answer: Still 32
```

Terminology

- Monadic, Dyadic, Triadic: The fact that there are {1, 2, or 3} separate states available
- Unary, Binary, Ternary: The fact that something is in *one* of {1, 2, or 3} separate states

Expression Operators Table

With this research in mind, I have defined the DocScript Operators as follows:

Operator	Description	Operands Type	Return Type	Precedence
=	Equality Comparison	2 (Anything)	Boolean	1
&	Concatenation	2 String	String	2
¬	Logical Not	1 Boolean	Boolean	8
'	Logical And	2 Boolean	Boolean	7
|	Logical Or	2 Boolean	Boolean	5
¦	Logical Xor	2 Boolean	Boolean	6
+	Addition	2 Number	Number	4
-	Subtraction	2 Number	Number	4
*	Multiplication	2 Number	Number	3
/	Division	2 Number	Number	3
^	Exponentiation	2 Number	Number	3
%	Modulo	2 Number	Number	3
~	InvertPolarity	1 Number	Number	8

Rules

- The Operators are Assign (:) and the Expression (Expr.) Operators
- Operators of the highest precedence are executed first
- The Logical Operators are Short-circuiting
- All DocScript Operators are left-associative.
- All Unary Operators have their Operand to their Right
- All Unary Operators must have the equally-highest precedence of all operators
- All DocScript Operators only ever read form their operands; they do not write to them

Explanations & Justifications

- All operator chars have been chosen as standard characters found on any ISO keyboard. Therefore, they are easy to type and do not require the memorisation of Alt Codes.
- Where possible, the operator chars have been chosen in accordance with what is standard practice for many programming languages. This makes meets the stakeholder need of the programming language being easy-to-use, and is therefore a **usability feature**.
- The unary operators have to be executed first (have the highest precedence) because the parser will generally read from left to right, in accordance with the language's operator associativity.

Instructions

With most of the language now clearly defined, I am able to see that these are the possible forms of valid line:

Possible Line Forms within a Function:

```
•  #Comment
•  <Number> Age
•  <Number> Age : 17       *E
•  Age : 17 + 1            *E
•  Return
•  Return "Value"          *E
•  SayHello()
•  SayHello("Ben")         *E
•  If (True)               *E
•  Else
•  EndIf
•  While (True)            *E
•  EndWhile
•  Loop (10)               *E
•  EndLoop
```

`*E = Includes an Expression (Expr.)`

Therefore, there are essentially **8 types of possible instruction** which could appear in a DocScript Source file:

Instruction Type	Example
Variable Declaration	`<String> Name`
Variable Assignment	`Name : "BenM"`
Return To Caller	`Return Name`
Function Call	`GetFullName(Name)`
Function	**`Function <String> GetFullName (<String> _Name)`**
If Statement	**`If (Name = "BenM")`**
While Statement	**`While (¬ [Name = "Ben"])`**
Loop Statement	**`Loop (GetStringLength(Name))`**

The last four Instructions (emboldened) are **Statements**, meaning that they can themselves *contain* other Instructions, **recursively**. Each Statement has its own variable scope, and therefore its own Symbol Table. That makes the Symbol (Identifier) Lookup order the following:

```
Variable Lookup Order: Localmost → FunctionLocal → Global
Function Lookup Order: Global (Program → BuiltIn's)
```

An Example DocScript Program

Now that there is a Formal (= concise and unambiguous) specification for the DocScript language, I have written and am providing this *example program* to show what it actually looks like:

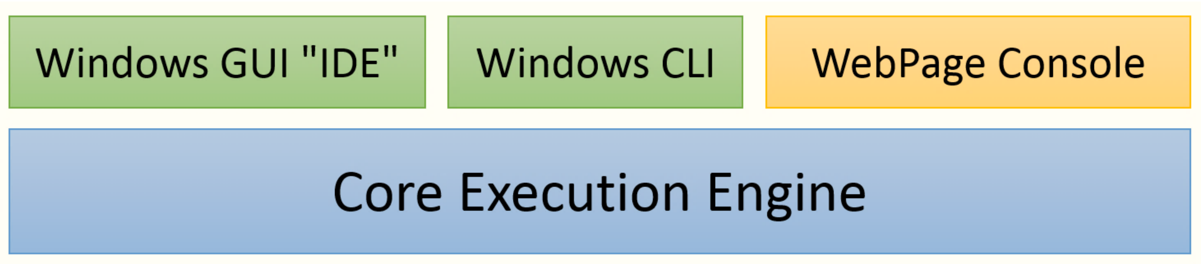
```
Function <Number> Main (<String@> _CLAs)

    #CLA Input looks like "/Name" "Ben" "/Age" "13"
    #Get the Value for an Input()'ed Key

    <String> _Key : Input("Sought Argument's Key:")
    <String> _Value : GetCLAValueFromKey(_CLAs, _Key)

    Output("Value: " & _Value)
    Return 0

EndFunction

Function <String> GetCLAValueFromKey(<String@> _CLAs, <String> _Key)

    <Number> _CurrentCLAIndex : 0

    While (_CurrentCLAIndex < [Array_MaxIndex(_CLAs) + 1])

        If (Array_At(_CLAs, _CurrentCLAIndex) = ["/" & _Key])
            Return StringArray_At(_CLAs, _CurrentCLAIndex + 1)
        EndIf

        CurrentCLAIndex : [CurrentCLAIndex + 1]

    EndWhile

    Return "No Value found for Key [" & _Key & "]"

EndFunction
```

DocScript System Architecture

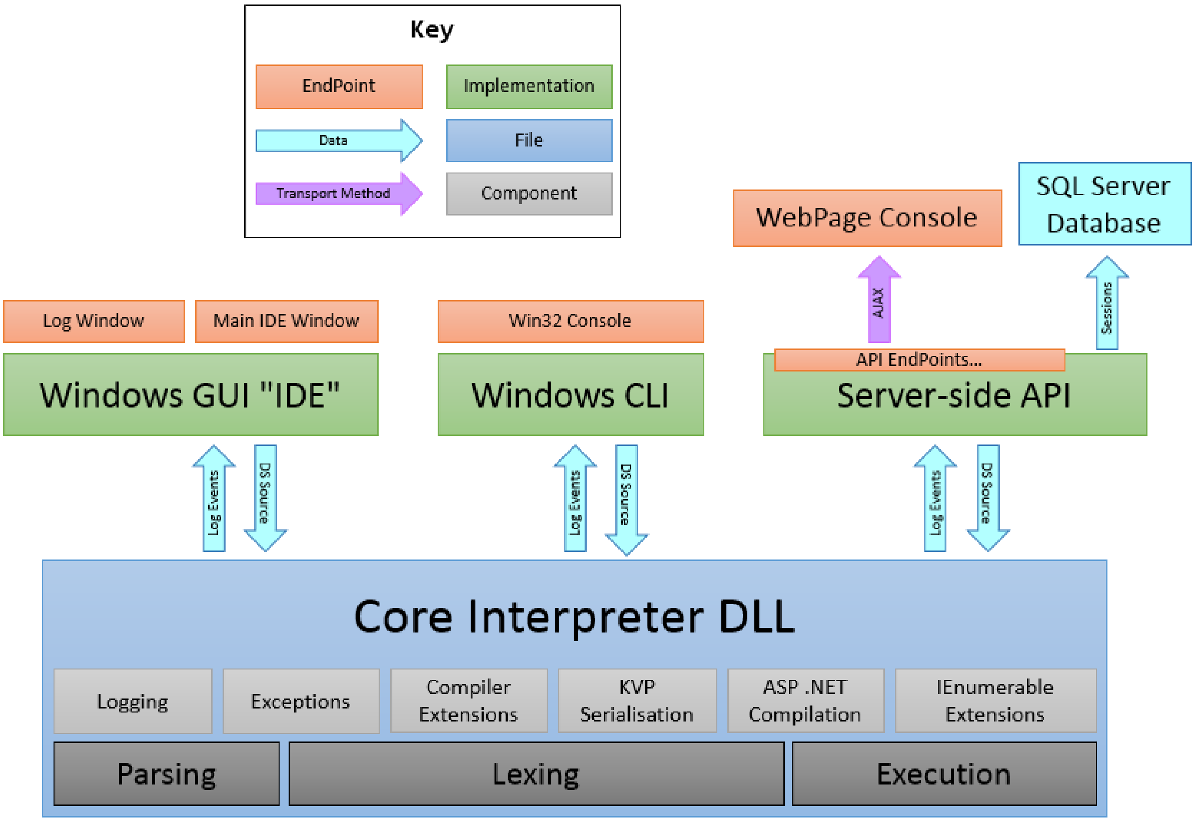
Because I now have a solid and objective specification for what the programming language is, I can design the system which will run programs what are written in compliance with that specification.

At a Glance

On account of the stakeholders' different intended use cases and correspondingly different computer types, it was decided in *§Analysis* that the system would be broken down into a base "*engine*" layer, on top of which would sit the various implementations. I provided this diagram:



However, because this is too vague and lacks the detail required to carry out the implementation of each of these four blocks, I have created this more in-depth architecture diagram:



Explanations & Justifications

- The core "*Engine*" is in reality to be implemented as a DLL (.NET Class Library) which I shall hereinafter refer to as the *DocScript Library DLL*. This is a suitable way of writing the base logic because a DLL provides reusable, maintainable (via successive in-situ recompilation) and scalable (via the addition of supplementary DLLs) Modules, Classes, and Methods. I would ***not* be**

afforded this same level of adaptability were I to simply copy and paste the interpretation logic from one implementation (E.g. the Windows IDE) to another (E.g. the Windows CLI).

- To meet the stakeholders' need of the verbose logging, any log messages generated by script in the Library DLL will be "*piped*" upwards to the implementation, for it to deal with. There is however, an interesting and nuanced problem herein; **how can these messages be displayed one a Win32 Window, Command-Line Interface, and Webpage, all with the same piping-up function?** My solution to this is to use a delegate *LogEventHandler* which must be assigned a function (Lambda Expression or AddressOf Function Pointer) by the implementer of the DLL. In other words, the DLL will simply say "*Here's a LogEvent for you; do what you want with it*". Then, the CLI, GUI, or WebPage (via the Server-Side API) can deal with the Log Message in whatever fashion is applicable. This is a highly-extensible architecture as any implementation can be made, including ones I have not hitherto conceptualised such as saving the LogEvent to a File or the Windows System Event Log. For this reason, I have detailed the "*Log Window*" as an endpoint component of the Windows IDE implementation (on the above diagram).
- The AJAX transport method for the API to WebPage console is suitable because this reduces server traffic and increases the speed with which distributed applications can be hosted. The principle of AJAX is that the client (WebPage) makes an AJAX request such as `/API/Get.ASPX?Item=CurrentTime` and this HTTP request **doesn't return a whole HTML webpage**, but instead, a serialised form of what was requested by the client, such as in XML; `<APIResponse CurrentTime="12:09" />`. This is therefore a very scalable and adaptable transport method, because almost any platform can make a simple HTTP request, and therefore new features and implementations can be added in the future, without having to alter the Server-Side API of which I speak.
- Having the DataBase as an SQL one is more *complex to configure*, but is significantly more powerful because SQL queries can themselves contain much of the logic which would otherwise have to be carried out by the server-side scripts. The main advantage however, is that if several requests come in at exactly the same time to the API, then they can be handled without fear of a DataBase write collision. Reading from a traditional File DataBase (such as a CSV or XML File) would soon lead to the occurrence of a collision; no request would be able to use the file because it would be IO-locked by another request.

To Compile, or not to Compile

I am principally faced with the choice of implementing the DocScript execution logic as either a Compiler (as seen with E.g. C++) or an Interpreter (as seen with E.g. VBScript). A brief comparison of these methods is as follows:

Interpretation	Compilation
Comprehends the Source, builds an IR*, and then executes it immediately	Comprehends the Source, builds an IR*, then constructs machine code instructions for the IR, which are saved to a binary executable file such as an *EXE*.
Execution phase is slower	Execution phase is faster
Permit more in-depth debugging and viewing of program contents during the	More difficult to deconstruct program during its execution

execution	
The entire interpretation process must occur each time the user wishes to execute the program. This requires the source each time.	The first two stages (Parsing and Lexing) need only occur once, during the construction of the binary. Thereafter, the program can be executed as many times as is desired, without having to look at the source again.
User must have a copy of the Interpreter on their computer	User needn't have any additional software on the computer

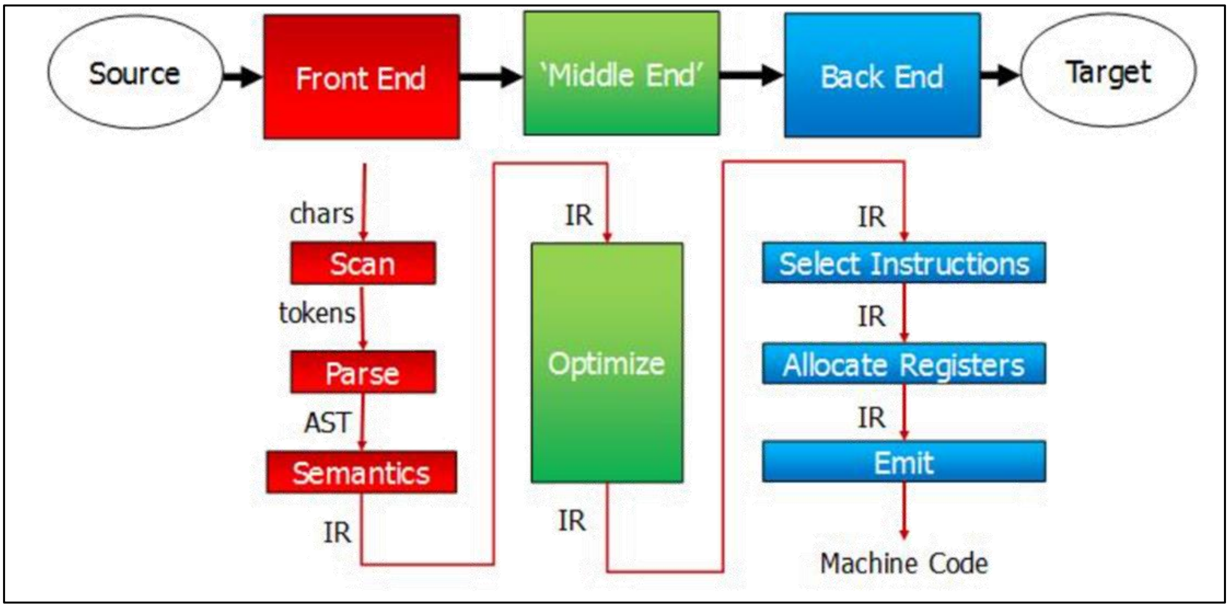
*IR = An Intermediate Representation (such as an Abstract Syntax Tree). In DocScript's case, the IR is an Instruction-Tree/Program-Tree (explained later)

DocScript will be an Interpreted Programming Language because the purpose of the language is to provide a highly-verbose debugging-friendly extensible development platform, which interpreters can achieve most effectively, owing to the fact that the source code is always available for comparison to the IR, and that **the IR remains in memory during the execution**. This also allows for other interesting possibilities including serialising the IR to XML and displaying a Program or Expression Tree. This wouldn't easily be possible with a compiled program.

Furthermore, the factors of *speed*, and *requiring the interpreter on the user's computer*, are less critical for this situation; of course DocScript won't be the world's *fastest* programming language, but that's not its purpose. In this instance, having extensive debugging abilities is more important than not needing an extra interpreter binary on the computer.

The Three Stages of DocScript Interpretation

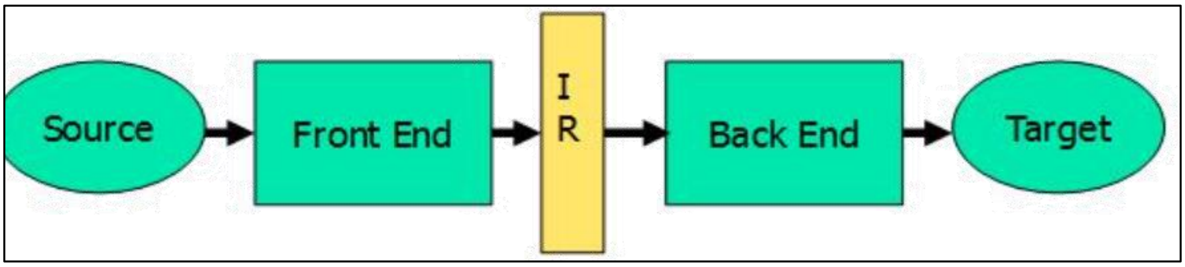
Most compilers [Below] have a *Front End* and *Back End*. The Front End Takes the source and produces an *Intermediate Representation* (IR) such as an AST. The Back End *takes in* the IR, and produces a machine code executable from it.



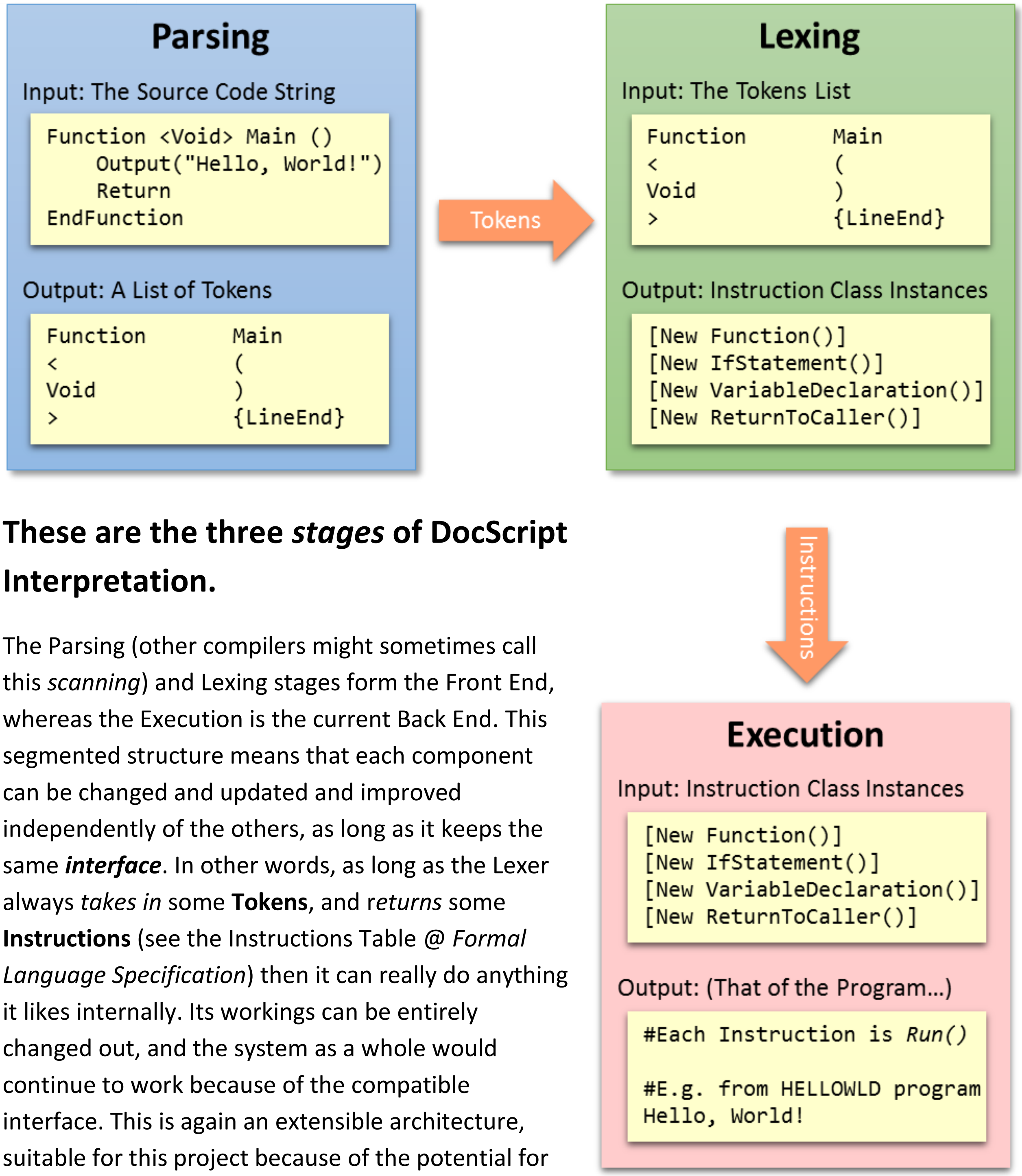
[Above] *A Compiler's Architecture*

Explanation: Although DocScript isn't a Compiled Language (at this point in time), it's interpreter will nevertheless have a Front End (to build an IR from Source) and a Back End (which takes in an IR, and executes each instruction in *Function Main* straight away).

Justification: This architecture means that if I actually wanted to be able to compile DocScript into EXE Files in the future, then all I would have to do, would be to take the already-existent IR, and generate machine code to correspond to it. Thus, this is an *extensible* architecture, which accounts for the potential needs of the **post-development** phase.



[Above] *A Source-To-IR view of a Compiler*



Parsing

Input: The Source Code String

```
Function <Void> Main ()
     Output("Hello, World!")
     Return
EndFunction
```

Output: A List of Tokens

```
Function          Main
<                 (
Void              )
>                 {LineEnd}
```

Lexing

Input: The Tokens List

```
Function          Main
<                 (
Void              )
>                 {LineEnd}
```

Output: Instruction Class Instances

```
[New Function()]
[New IfStatement()]
[New VariableDeclaration()]
[New ReturnToCaller()]
```

Execution

Input: Instruction Class Instances

```
[New Function()]
[New IfStatement()]
[New VariableDeclaration()]
[New ReturnToCaller()]
```

Output: (That of the Program...)

```
#Each Instruction is Run()

#E.g. from HELLOWLD program
Hello, World!
```

These are the three *stages* of DocScript Interpretation.

The Parsing (other compilers might sometimes call this *scanning*) and Lexing stages form the Front End, whereas the Execution is the current Back End. This segmented structure means that each component can be changed and updated and improved independently of the others, as long as it keeps the same ***interface***. In other words, as long as the Lexer always *takes in* some **Tokens**, and *returns* some **Instructions** (see the Instructions Table @ *Formal Language Specification*) then it can really do anything it likes internally. Its workings can be entirely changed out, and the system as a whole would continue to work because of the compatible interface. This is again an extensible architecture, suitable for this project because of the potential for future expansion of the project in the **post-development phase**.

Parsing & Tokens

Any comments which appear in the source (E.g. `#This is a Comment`) do not become tokens. They are discounted at this first stage. A Token is a small structure-defined Object with a *Value* (from the source), a *TokenType* (see below table), and a *LocationInSource* Property, which greatly aids in debugging for the user, because they can see the exact location of an erroneously-placed Token.

Token Types

These are the [TokenType]s in DocScript:

TokenType	Example
Unresolved	(Only used before a Token’s Type is determined)
StringLiteral	"Value"
NumericLiteral	0110_2
BooleanLiteral	True
Keyword	Function
DataType	Number
Identifier	Main
DSOperator	&
GrammarChar	(
LineEnd	*
StatementEnd	EndFunction

TokenType Regular Expressions

As a means of **determining a TokenType**, *from a Token’s Value*, I have written these Regular Expressions:

```
StringLiteral  ^("[^"]*")$
NumericLiteral ^((\d{1,10}(\.\d{1,4})?)|[A-Za-z0-9]{1,10}_\d{1,3})$
BooleanLiteral ^((TRUE)|(FALSE))$
Keyword        ^((IF)|(ELSE)|(WHILE)|(LOOP)|(RETURN)|(FUNCTION))$
DataType       ^(((STRING)|(NUMBER)|(BOOLEAN))@?|(VOID))$
Identifier     ^(_?[A-Z]+[A-Z_]*)$
DSOperator     ^(:|=|&|¬|'|\||¦|\+|\-|\*|/|\^|%|~)$
GrammarChar    ^(\(|\)|\[|\]|<|>|\,)$
LineEnd        ^(\r\n)$
StatementEnd   ^(END((IF)|(WHILE)|(LOOP)|(FUNCTION)))$
```

Explanation: These TokenTypes help with the subsequent *Lexing* stage, wherein accounting for all possible values of a Token would be impossible, and it is therefore necessary to have an indication of what the type of a Token’s value is, without having to know what the value itself is.

Justification: Having many different TokenTypes means that these is less additional analysis needed within the Lexing stage. For example, I could have just had a singular TokenType for both Keywords (E.g. Function) and StatementEnds (E.g. EndFunction). However, then within the Lexer logic, I would need to write an additional If Statement to identify where the StatementEnds are, each time one was expected. **This would be *needlessly*-indirect**. Therefore, it’s best to have this healthy, heterogeneous multiplicity of TokenTypes.

Parsing Algorithm Pseudocode

This is my pseudocode for what the **parsing Algorithm** (Source → Tokens) looks like *"from 1000 feet"* as it were:

```
REM ╔══════════════════════════════════════════════╗
REM ║          DocScript Parsing Process           ║
REM ╚══════════════════════════════════════════════╝

REM 1) Initialisation
'          - Ensure all LineBreaks are valid (CrLf)
'          - Load in Lines of Source

REM 2) Segmentation
'          - Blank out any #Comments or Whitespace Lines
'          - Ensure nothing already exists in the Source which matches the SLIT RegExp
'          - Replace StringLiterals with SLITs e.g. $SLIT_0$
'          - Generate Segmented Tokens:
'          - (For Each Line, and For Each Character thereof, evaluate if it's a WordChar
                    or SplitAtChar...)
'          - Remove any Null Tokens (Whitespace, etc...) (...Except from LineEnd Tokens)
'          - Ensure all remaining characters are valid (E.g. No SpeechMarks)
'          - Replace any SLITs with their original StringLiterals

REM 3) Classification
'          - For Each Token, attempt to match it to a RegExp for its TokenType
'          - Ensure all Bracket usage is balanced ( and ), [ and ], < and >
'          - Ensure all Statement Openings and Closings are balenced (Function to EndFunction, etc...)

REM [_RawSourceLines] → [_CleanSourceLines] → [_SegmentedTokens] → [_NonNullTokens]
→ [_TokensWithStringLiterals] → [_ClassifiedTokens]
```

Lexing & Instructions

As can be seen in the Output of the Lexer, and the Input of the Executer, each type of Instruction (see the Instructions Table @ *Formal Language Specification*) has a corresponding class in the interpretation logic. These classes have their own Properties which differ for each of the Instruction Types. The *IfStatement* class for instance, has a member called *Condition*, which is an expression to indicate whether or not the *IfStatement* should run its *Contents*.

In other words, the **Lexing occurs in the constructors for the Instruction Classes**. When invoked, these constructors are passed the part of the TokenList (from the Parser) required to create the Instruction.

Instruction Classes & Interfaces

Justification: I therefore need some way to represent each of these Instruction Classes in a logical object-orientated fashion. All the instructions have a method called `Execute()`, and all the Statement Instructions additionally have a Property called `Contents`, as well as a private Symbol Table called `ScopedVariables`.

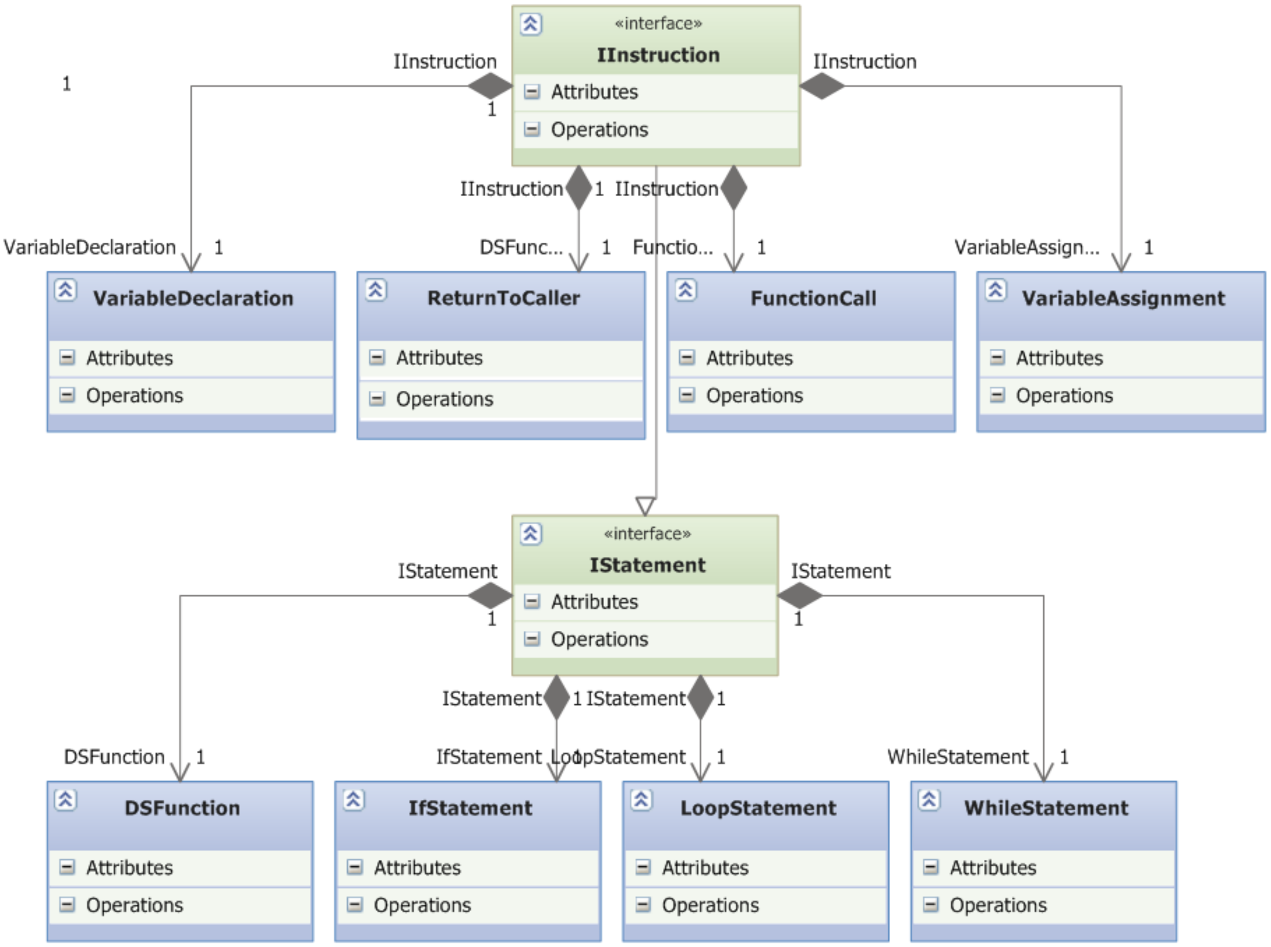
Explanation: To implement these Instruction Classes, and ensure that they have the required methods and properties within them, I shall therefore use the Object-Orientated feature of *Interfaces* like this:

- A Base Interface `IInstruction` declares the `Execute()` Method
- A Child Interface `IStatement` Inherits `IInstruction` and declares `Contents` and `ScopedVariables`

- Each Instruction Class then Implements the appropriate Interface and defines the methods declared by that Interface

Diagrammatically, that relationship looks like this…

Instruction Classes Diagram



(This, and several of the diagrams henceforth, were made with *Visual Studio Modelling Diagrams*, or the *Visual Studio .NET XAML and Windows Forms Designers*)

Instruction Class Members

```
VariableDeclaration
<String> Name
<String> Name = "Ben" & ToString(7)

Identifier As String
DataType As Type
DeclarationType As VarDecType {DeclareOnly, DeclareAndAssign}
InitialiserExpression As Expression
```

```
ReturnToCaller
Return
Return "Value"

ReturnType As ReturnInstructionType {ReturnOnly, ReturnWithValue}
ReturnValue As Expression
```

```
VariableAssignment
Name = "Ben" & ToString(7)

Identifier As String
InitialiserExpression As Expression
```

```
FunctionCall
Output()
Output("Value")

Identifier As String
Arguments As List(Of Expression)
```

These last four Classes **Implement** IStatement, which means implicitly that they implement IInstruction too (because IStatement Inherits IInstruction)

```
DSFunction
Function <Void> SayHello ()
Function <Number> Main (<String@> _CLAs)

Identifier As String
ReturnType As Type
Arguments As List(Of Parameter)
```

```
IfStatement
If (True)

Condition As Expression
```

```
WhileStatement
While (True)

Condition As Expression
```

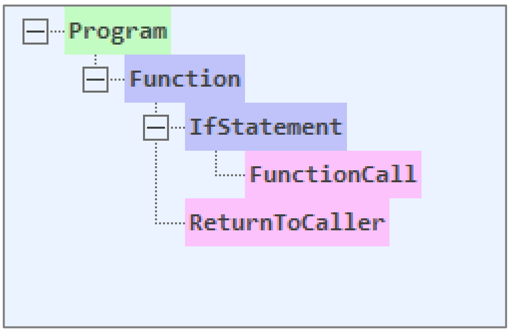
```
LoopStatement
Loop (10)

LoopCount As Expression
```

Instruction Trees

The output of the Lexer is an IR (Intermediate Representation), which in DocScript's case, is essentially an "***Instruction Tree***".

Here is an example DocScript Instruction Tree and associated Program:

```
Function <Void> Main ()
      If (System_GetTime() = "12:09")
            Output("The Time is Correct")
      EndIf
      Return
EndFunction
```



Lexing Algorithm Pseudocode

Because the Lexing occurs in the constructors for the Instruction Classes, those essentially contain the Lexing Logic, and are each very much specific to whichever IInstruction they represent. This is what the VariableDeclaration's constructor looks like:

```
REM Source should look like:
REM     <String> Name
REM     <Boolean@> _Pixels : GetImageRow(0)

REM Tokens should look like:
REM     [GrammarChar], [DataType], [GrammarChar], [Identifier], [LineEnd]
REM     [GrammarChar], [DataType], [GrammarChar], [Identifier],
        [ExprTokens...], [LineEnd]
```

```
REM Fields to Initialise:
REM     DataType
REM     Identifier
REM     AssignmentExpr

LogLexingMessage("Began constructing a VariableDeclaration...")

REM Ensure that there are enough tokens to construct the IInstruction
REM Ensure that the last Token is a {LineEnd}
REM Ensure that the main Token Pattern Validator herefor is satisfied

REM The DataType should be derivable from the 2nd Token
REM The Identifier should be derivable from the 4th Token
REM If there is an AssignmentExpr, derive it from all Tokens after the 5th one (6th onwards...)

'Token 4 should be the Assignment Operator
'Tokens after Token 4 (5 onwards) should form the AssignmentExpr, up to the {LineEnd}

LogLexingMessage("...Finished constructing a VariableDeclaration Object for & Me.Identifier)
```

Explanation: The Tokens must for the most part follow a pre-defined syntax and order. In this instance, the first Token *must* always be a **<**, but the next Token – the DataType – could be one of six possible values. The Assignment Expression (for initialising the variable with a value) could appear in so many forms that it is impossible to account for all of them. Therefore, my approach does not deal with the Token Stream as a whole, but instead, processes the Tokens *one-by-one*.

Justification: This incremental, *stepping*-forward-through-the-tokens approach means that if there is an erroneously-positioned or unexpected token, then it is *possible to pinpoint precisely where that token is*, and report it to the user. Although simply matching the whole token stream against a Regular Expression or TokenPatternValidator would be *easier*, it would not provide this level of verbosity.

Execution & I/O Delegates

This, the *third* in and final stage of DocScript Interpretation, is where a fully-formed Instruction Tree IR is executed, starting with any Global Variable Declarations, and followed by the Function `Main` EntryPoint. As can be seen from the valid EntryPoints in the Formal Language Spec., the program can **take in Command-Line Arguments** (forwarded by the Library DLL Implementer) and **Return an Exit Code** (0=Okay; ¬0=Error).

Because each Instruction implements the `Execute()` method, and each IStatement Instruction calls `Execute()` on all its child Instructions (in its `Contents` Property) recursively, only the top-most Instructions inside any Function need to be executed. By design, **the recursive nature of the Instruction Tree does not need to be dealt with by the executor.**

The ExecutionContext Class [DocScript.Runtime.ExecutionContext]

The *DocScript Library DLL* is implemented into three different application forms: The Command-Line Interpreter, the Windows IDE, and the Web Console. When running under each of these contexts, the built-in `Input()` and `Output()` Functions (see the Formal Language Spec.) need to do different things. For instance, in the Command-Line Interpreter, `Output()` should write text to the Console. In the Windows IDE however, it should show a graphical Win32 MsgBox-style Window to the user. This creates the problem of how to handle the different Input and Output modes at the Library DLL level.

Explanation: My solution to this problem is to have an ExecutionContext Class, which is passed in to the `Execute()` method of an `IInstruction`, and provides an InputDelegate, OutputDelegate, and RootFolder for the execution of the Instruction and its Children. Its declaration would therefore look something like this:

```
Public Class ExecutionContext
	Public ReadOnly RootFolder As IO.DirectoryInfo
	Public ReadOnly InputDelegate As Func(Of String, String)
	Public ReadOnly OutputDelegate As Action(Of String)
End Class
```

Justification: This design choice has the effect of allowing the DLL implementer to choose what do to when the `Input()` and `Output()` Functions are called from within the DocScript Program. It is therefore an extensible design satisfying the needs of all three implementations, as well as allowing for furtherance of the Input/output methods on the part of the DocScript programmer.

In addition to an ExecutionContext, the `Execute()` methods require a Stack of Symbol Tables (Global, FunctionLocal, and then one per Statement) and return an `ExecutionResult`, which contains data about whether or not to Return to the caller, and if there is an associated Return Value. That makes the `IInstruction.Execute()` Declaration look like this:

```
GetSurname("Ben", 17) & Ending & IsAllowed()
```

Other Key Classes

Aside from the Parser, Instruction Classes, and ExecutionContext, the following are some of the most important Classes in the Library DLL.

Expressions [DocScript.Language.Expressions.IExpression]

There are a number of shared constructs which all Instructions need to be able to construct. One of these is an **Expression**, which – in DocScript – is defined as:

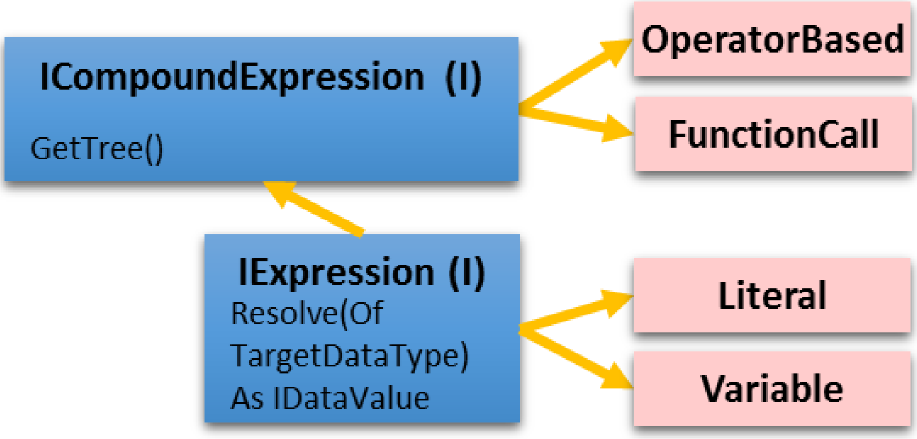
“A resolvable collection of Operators, Literals, Variables, and FunctionCalls, which produces a value”

Example valid DocScript Expressions include: (One per Line)

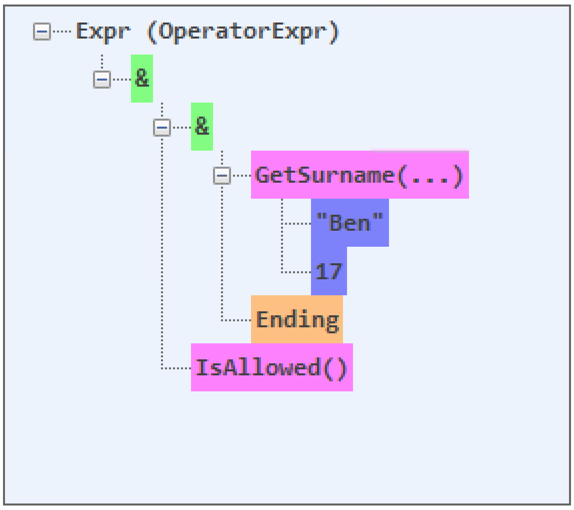
```
"Hello, World!"
23.6663
True
"Hello" & ", World!"
```

```
"Hello, " & GetName()
"Hello, " & GetFullName("Ben " & "Mullan")
ToString([5 + ~4]) & "9" & ToString(¬True¦False)
[5_12 + 6 - 7.4 ^ 3 ^ ~G()] > ~ 10110101_2
GetSurname("Ben", 17) & Ending & IsAllowed()
```

To represent these programmatically, I have designed a system of Interfaces and Classes to represent different Expression Components:



Then, to construct an IExpression Tree (ExprTree) from Tokens – something several of the Instruction Class constructors will *need* to do – the utility Function `ConstructExpressionFromTokens()` is called. **Here is an example of such an Expression Tree:**

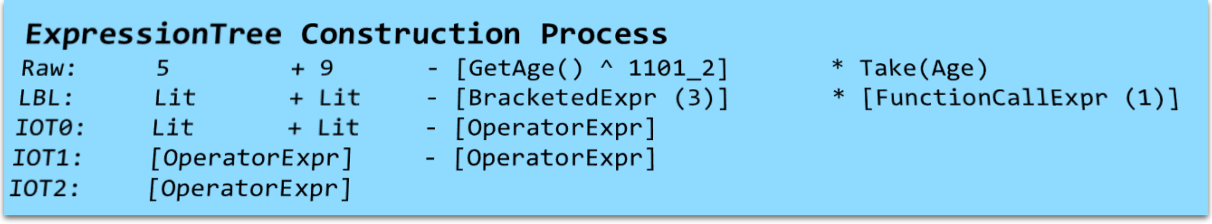


```
Expr (OperatorExpr)
  &
    &
      GetSurname(...)
        "Ben"
        17
      Ending
    IsAllowed()
```

And here is the corresponding RAW Expression:

```
GetSurname("Ben", 17) & Ending & IsAllowed()
```

The most obvious remaining question is therefore: **How can an ExprTree be constructed from a RAW Expression?** After some thought, I came up with the following solution:



```
ExpressionTree Construction Process
Raw:      5            + 9        - [GetAge() ^ 1101_2]        * Take(Age)
LBL:      Lit          + Lit      - [BracketedExpr (3)]        * [FunctionCallExpr (1)]
IOT0:     Lit          + Lit      - [OperatorExpr]
IOT1:     [OperatorExpr]          - [OperatorExpr]
IOT2:     [OperatorExpr]
```

Explanation: With a ***Linear Bracketed Level (LBL)*** constructed, we only need to worry about the operators and their precedence, because we *know*, that the contents of the BracketedExprs and FunctionCall Arguments must be resolved first. Collapsing to ***Intermediate Operator Trees (IOTs)*** is the subsequent stage whereby the operators with the highest precedence are the first to be collapsed into OperatorExprs. This eventually forms a Complete Tree, free from any LBL Placeholders (formerly for the Operators, BracketedExprs, and FunctionCalls).

Justification: By firstly forming the Linear Bracketed Level, a layer of abstraction is applied which permits the collapsing *of* the LBL *into* an operator tree. Without the LBL, all Tokens of the Expression would be exposed, which would render impossible coherent lexical analysis; the lexer would not know if a given Token is part of the current tree node, or a child one. This is the case because in the expression-constructing For Loop, the lexer only sees one Token at a time; it cannot contextually and peripherally comprehend the *entire* expression simultaneously.

Realisations:

- Resolve()ing an Expression requires all the same resources as Execute()ing an Instruction; Symbol Tables are needed for Variable and Function Lookups
- We don't actually need to know what the value of any of the expression components are, for the purposes of constructing the Expression Tree. All we care about at that stage is which token is an Operator and which a Literal or a Variable etc…
- The operators with the lowest precedence will be the highest-up in the Tree

  (I shall be further elucidating the ExprTree construction process during the Development Stage, by means of source code examples...)

Programs [DocScript.Runtime.Program]

This contains an array of Functions and Global Variable Declarations from the DocScript Source.

Explanation: When an Instruction Tree has been created (see the earlier example), it is loaded into a `Program` Object, ready to be executed. This class also handles the serialisation of a Program to XML, and the forwarding of Command-Line Arguments and the Exit Code.

Justification: Without the Program Class, the Functions and Global Variable Declarations would be floating around in Global Arrays. This would make it very difficult to pass around a DocScript Instruction Tree. In fact, this mechanism means that there can be multiple DocScript Programs loaded into a single implementation instance, which wouldn't otherwise be possible.

Key Algorithms

I have drafted these out in Visual B.A.S.I.C. .NET instead of some made-up pseudocode syntax, because it is more consistent (…and I can actually run and thereby test them during subsequent stages of development!).

The Stack-Balanced Algorithm

I will use this for ensuring that the Brackets `( ) [ ] < >` are well-balanced in the input Source. I also use the same logic for Statement Openings (E.g. `Function`) and Statement Closings (E.g. `EndFunction`) to ensure that they are properly and equally provided in the source. *(Also called the Bracket-Stack Algorithm)*

```
Public Function ContainsWellBalencedPairs(Of _TItems)(ByVal _AllItems As _TItems(), ByVal ParamArray _Pairs As Tuple(Of _TItems, _TItems)()) _
 As Boolean

    'Contains only Items which are also present in the _Pairs (E.g. only the Brackets [] {} <> out of all the Source Tokens)
    Dim _JustTheRelevantItems As _TItems() = ( _
     From _Item As _TItems In _AllItems Where ( _
      _Pairs.Any(Function(_Pair As Tuple(Of _TItems, _TItems)) (_Pair.Item1.Equals(_Item)) OrElse (_Pair.Item2.Equals(_Item))) _
     ) Select _Item _
    ).ToArray()

    Dim _ItemsStack As New Stack(Of _TItems)()
    Dim _NoOpeningComponents, _NoClosingComponents As UInt32

    For Each _Item As _TItems In _JustTheRelevantItems

        'Was advised to do this by VS 2010. Sure.
        Dim _LambdaCopyOf_Item As _TItems = _Item

        REM If we have an Opening Component, Push() it onto the Stack
        If _Pairs.Any(Function(_Pair As Tuple(Of _TItems, _TItems)) _Pair.Item1.Equals(_LambdaCopyOf_Item)) Then

            _ItemsStack.Push(_Item) : _NoOpeningComponents += 1UI

            REM If we have a Closing Component, find out if _ItemsStack.Pop() produces the corrosponding Opening Component
        ElseIf _Pairs.Any(Function(_Pair As Tuple(Of _TItems, _TItems)) _Pair.Item2.Equals(_LambdaCopyOf_Item)) Then

            'Get the Pair which contains the corrosponding Opening Component for out Closing Component
            Dim _Pair_WhereforWeHaveClosingComponent As Tuple(Of _TItems, _TItems) = _
             _Pairs.First(Function(_Pair As Tuple(Of _TItems, _TItems)) _Pair.Item2.Equals(_LambdaCopyOf_Item))

            'Now see if the Pop() produces the same Opening Component as we have in our Pair
            '(Additionally, if the Stack is empty, then the Items aren't well-balenced, because we just hit a Closing Component
            '   which(didn)'t follow a previous corrosponding Opening Component)
            If (_ItemsStack.Count = 0) OrElse (Not _ItemsStack.Pop().Equals(_Pair_WhereforWeHaveClosingComponent.Item1)) Then _
             Return False

            _NoClosingComponents += 1UI

        Else
            Throw New Exception("An Item was not recognised as either an Opening or Closing Component")
        End If

    Next

    REM Now determine if there is anything left on the Stack.'
    REM The only things on there could ever be Opening Components, because that's all we ever Push()
    If Not (_ItemsStack.Count = 0) Then Return False

    REM If we're here, then all Opening and Closing Components must have been balenced, because we haven't yet Return'd
    Return True

End Function
```

[Above] The ContainsWellBalencedPairs-of-Tea Algorithm (Included in DocScript (.sln) Solution)

Explanation: The Algorithm works like this:

```
Declare a _TItems Stack
```

```
For Each Item In _JustTheRelevantItems

       If the Item is an Opening Component (_Pair.Item1) then Push() it onto stack

       If the Item is a Closing Component (_Pair.Item2) then Pop() stack and if
       the popped Item is the matching Opening Component then fine, but otherwise
       the Items are not balanced

After complete traversal, if there is an Opening Component left in stack then the
source is not balanced
```

Justification: I had originally thought that I could just get the number of Opening Brackets, and then the number of Closing Brackets, compare them, and Throw an Exception if they weren't equal. However, I decided to implement *this* method **using Generics** so that I could use it for both the Brackets (which are Chars) and the Statements Pairs (which are Strings). The algorithm makes sure that the _Pairs components are opened in a balanced and in-order fashion. E.g. if done with brackets, then "([])" would be valid, whereas "([)]" would not be (even though there are the same number of brackets and squares in the latter).

The Unique Elements Algorithm

I will use this when adding items to the Symbol Tables. Within a given Symbol Table (*SymTbl*), all Identifiers must be unique; they act as the *Primary Key*.

```
Public Function AllElementsAreUnique(Of _TElement)(ByVal _Array As _TElement()) _
    As Boolean

        Dim _HashSet As HashSet(Of _TElement)
        _HashSet = New HashSet(Of _TElement)(_Array)

        Dim _LengthsMatch As [Boolean]
        _LengthsMatch = (_HashSet.Count = _Array.Length)

        Return _LengthsMatch

        REM Or, in one Line:
        Return (New HashSet(Of _TElement)(_Array)).Count = _Array.Length

End Function
```

The Function evaluates whether or not each Element in the _Array is unique. In other words, False is Returned if two or more elements are the same. This works because a HashSet (HashTable) cannot contain two identical elements, because their position-determining hashes would be the same. Therefore, if the size of the HashSet is the same as the size of the original Array, then no shrinkage has occurred during the HashSet construction, and all elements in the `_TElement` Array are unique.

Key (Global) Variables

To recap: I have herebefore explained and justified the need for several of the most important **Classes** in the solution; Parser, Token & TokenType, IInstruction & IStatement, IExpression and Program. Here, I explain the purpose of some of the key Variables in the Solution:

Variable	Description & Justification	Validation

`Public Shared DocScript.Logging.`
`CurrentLogEventHandler`
`As System.Action(Of DocScript.Logging.LogEvent)`
	(As previously mentioned) The value of this variable is assigned by the implementer of the Library DLL, to a Function Pointer, where the target function processes the LogEvent in whatever method is applicable for the implementation. It is needed in order to avoid having to hard-code a Logging Delegate into the Library DLL, which would be bad programming practice, and mean that the implementations would all have to show LogEvents in exactly the same way.	(None needed – apart from ensuring that the Value isn't Nothing (*nullptr*) before reading it. This small check will be done in the SubmitLogEvent() Method with a simple If Statement)

`Public Shared DocScript.Logging.`
`ProcessDebugEvents`
`As System.Boolean`
	Indicates whether or not LogEvents with the Severity *Debug* are processed by the CurrentLogEventHandler. This is needed because the number of DebugEvents produced can be so large, that the performance of the application is actually impeded by logging them all. Because the DebugEvents are not required during normal usage of the Interpreter, it is very prudent to be able to disable them to improve performance and make the other log messages not entirely inundated.	(None needed; the value can only possible be True or False – both of which are completely fine)

`Private Shared DocScript.Logging. BuiltInLogEventHandlers.`
`LogWindow_`
`As DocScript.Logging.LogWindow`
	Holds the one instance of a graphical Log Window (for the Windows IDE's LogEventHandler) for the entire implementation instance. Without this, the LogEventHandler would have no way of knowing if a LogWindow had already been instantiated.	(None needed; either the value is Nothing (*nullptr*) because GUI Logging is not is use, or it's a LogWindow Object)

`Private Shared ReadOnly DocScript.Runtime.Parser.`
`TokenTypeToRegExp_Table_`
`As System.Collections. Generic.Dictionary(Of System.String, DocScript.Runtime.Token. TokenType)`
	Provides a runtime-initialised Dictionary of the 11 TokenTypes to the Regular Expression responsible for detecting each TokenType (see @ *TokenType Regular Expressions*). Without this, a messy entanglement of If Statements would have to be used.	(None needed; the Collection Type is runtime-initialised, and is ReadOnly so no assignment can occur to it)

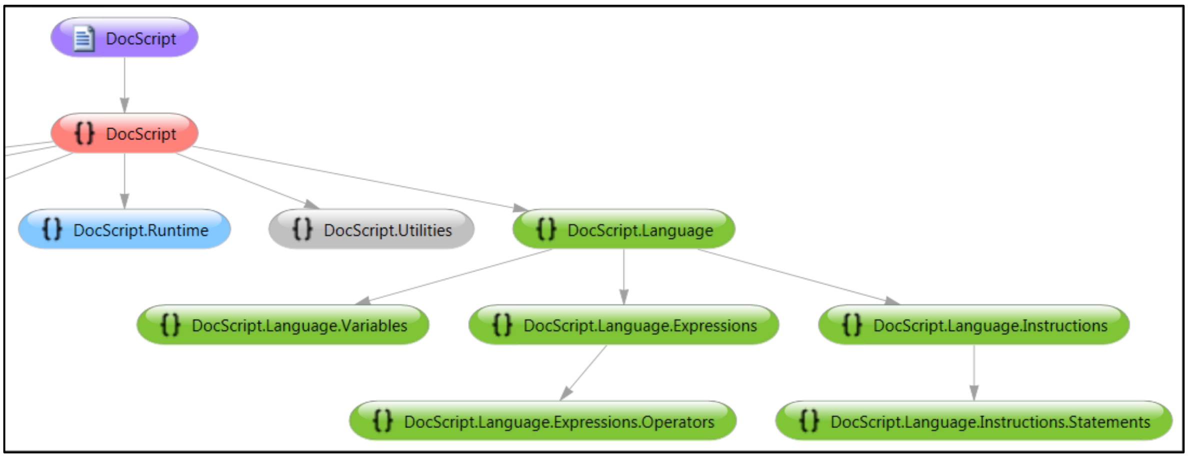
`Public Readonly DocScript.Runtime. TokenPatternValidation.`
`MinimumRequiredTokens`
`As System.Collections. Generic.Dictionary(Of`
	Provides a runtime-initialised Dictionary of the 8 Instruction Classes' Types to the lowest number of Tokens required to construct one of those Classes. This acts as an initial layer of validation on the	(None needed; the Collection Type is runtime-initialised, and is ReadOnly so no assignment can occur to

`System.Type, System.Byte)`	constructors for the IInstruction Classes, and without it, a NullReferanceException could be Thrown when attempting to read a Token which doesn't exit.	it)

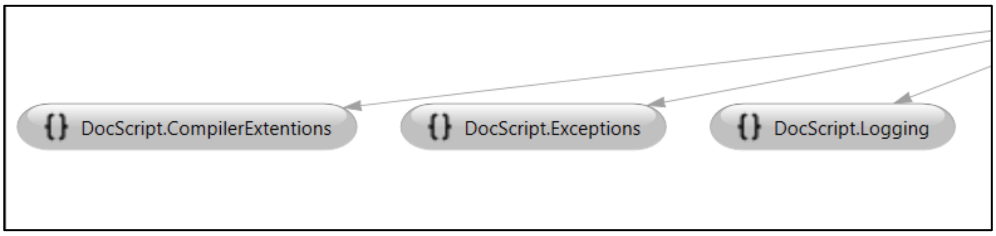
(There are very few Global Variables in the application ↑ because it is bad programming practice to use them in an Object-Orientated project such as DocScript. The preponderance of data are stored as local variables within Functions and Subroutines and Classes.)

Namespaces

The aforementioned key Variables and Classes are organised into a tree-like structure of Namespaces within the Library DLL. Here is a diagram of what that looks like:

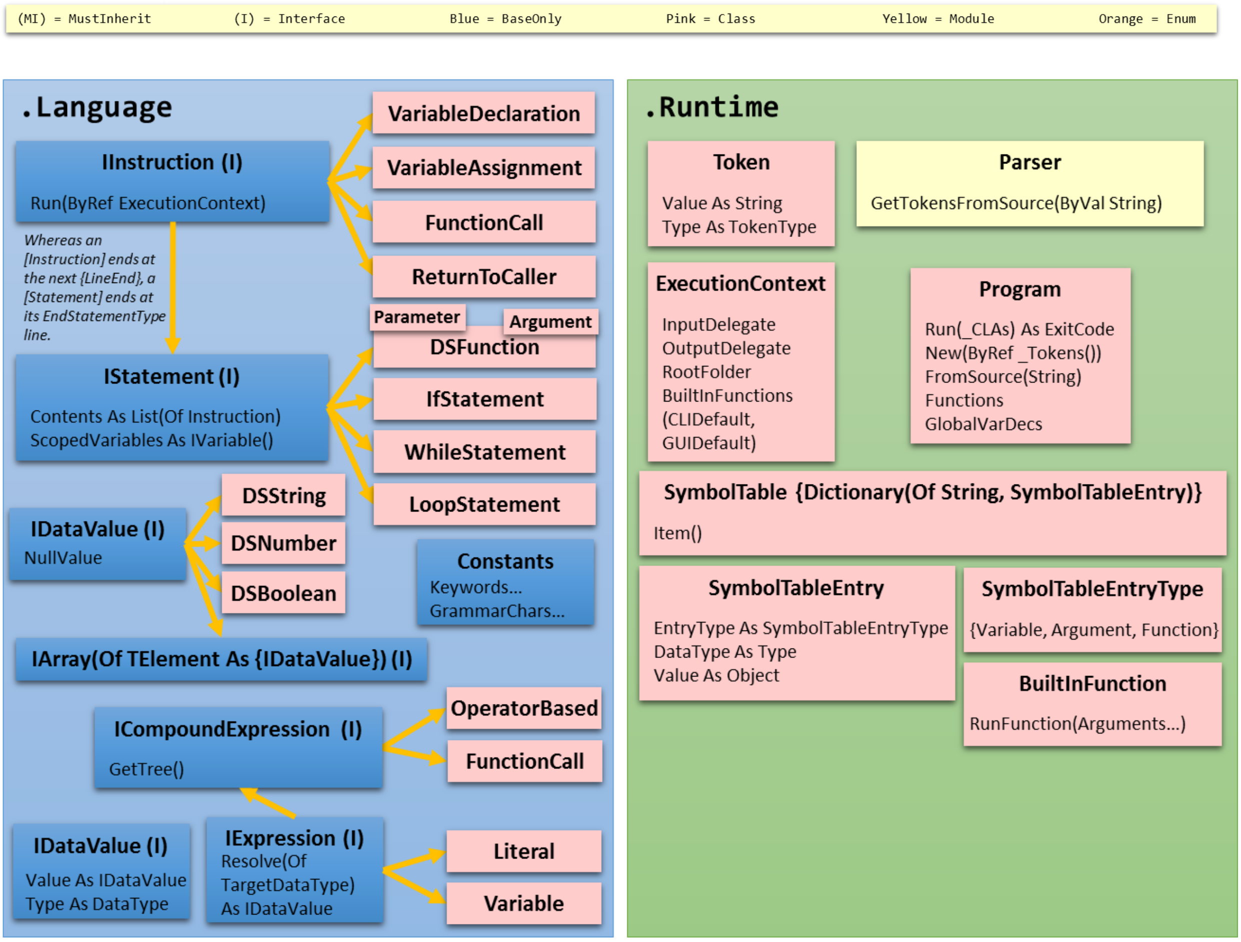

DocScript
{} DocScript
{} DocScript.Runtime
{} DocScript.Utilities
{} DocScript.Language
{} DocScript.Language.Variables
{} DocScript.Language.Expressions
{} DocScript.Language.Instructions
{} DocScript.Language.Expressions.Operators
{} DocScript.Language.Instructions.Statements

↓ The Left side of the Namespace Tree ↓



Justification: These "*Folders for Classes*" significantly improve the organisation of logical resources (Classes, Structures, Enums, Delegates, and Modules) within a large project such as DocScript. Without them, there would be simply a disjunct coagulation of .NET Types without any easy way of locating them.

Here is an (*abstracted*) view of the contents of the two main Namespaces, Language and Runtime:



Piecing It All Together!

With an explanation of what many of the individual key components do now written, I shall demonstrate how a DocScript Program can be easily constructed from the raw source. This is all that the **implementer of the Library DLL** needs to type:

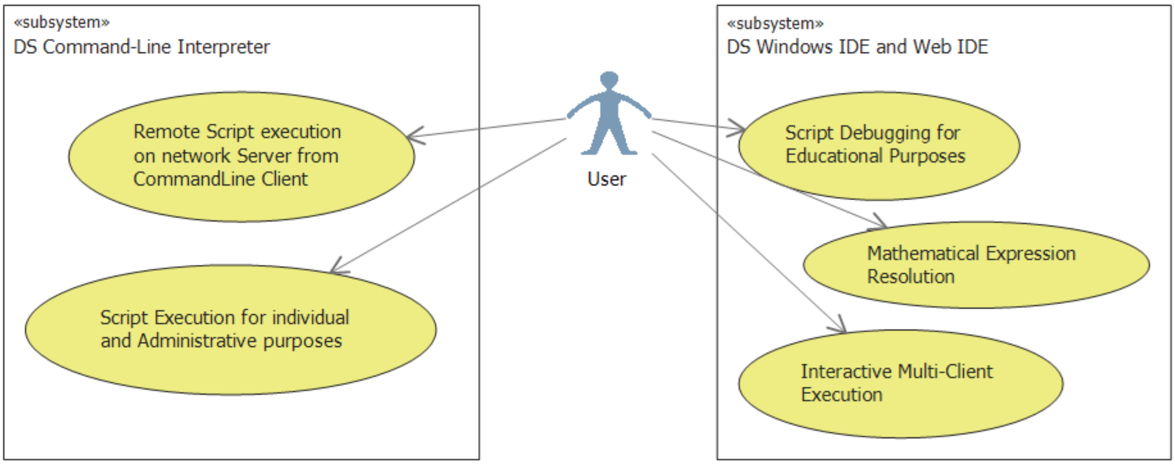
```
'Log via the Default LogWindow
DocScript.Logging.LogUtilities.CurrentLogEventHandler = _
        DocScript.Logging.BuiltInLogEventHandlers.GUIDefault

'Raw
Dim _Source As [String] = "..."

'Parse
Dim _Tokens As DocScript.Runtime.Token() = _
        DocScript.Runtime.Parser.GetTokensFromSource(_Source)

'Lex
Dim _Program As New DocScript.Runtime.Program(_Tokens)

'Execute
_Program.Run({})
```

Justification: The algorithms I have designed form a complete solution because:

- The **Tokens** provide a very useful layer of abstraction on top of the Raw Source, but still provide access to the location that each component occurs at within the source via the Line and Column. This is very useful for the next stage (Lexing) because it means that the lexer algorithms don't have to traverse the source character-by-character, which **would be error-prone and very slow!**
- The Program Object makes it easy to load in an array of Tokens (a `Runtime.Token()`) and execute the Instruction Tree created. Without this, the recursive nature of the Instruction Tree would mean that a complicated series of individual function calls would have to be made. With my current design however, each top-level Function's output feeds into the next; **Source → Tokens → Program**.
- Having the `CurrentLogEventHandler` assignment at the top means that the Logging *mode* **need only be specified once for the entire application**. This simplifies an otherwise complex process.

The DocScript Implementations

The *DocScript Library DLL* (whose architecture is fastidiously detailed hereabove) will be implemented into **three usable implementations**; the Command-Line Interpreter, Windows IDE, and Web Client (DS *Interactive*). This **Use Case Diagram** shows how each implementation might be used, based on what the Stakeholders have said so far:



«subsystem»
DS Command-Line Interpreter
Remote Script execution on network Server from CommandLine Client
Script Execution for individual and Administrative purposes
User
«subsystem»
DS Windows IDE and Web IDE
Script Debugging for Educational Purposes
Mathematical Expression Resolution
Interactive Multi-Client Execution

The Windows IDE (DSIDE.EXE)

At its rudiments, this implementation has a Window with a TextBox for DocScript source to be typed into, and an *Execute* Button to interpret that source using logic from the Library DLL.

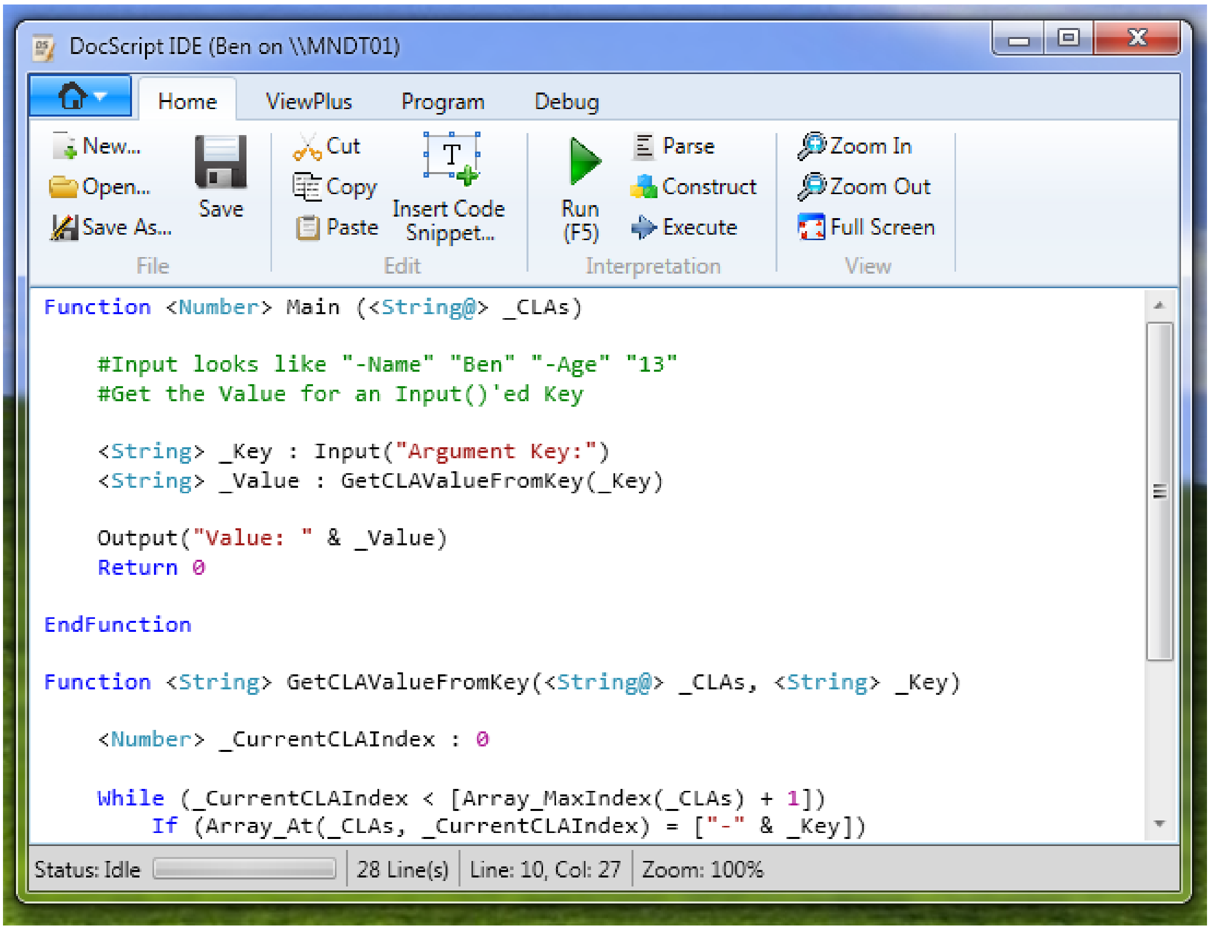
The Criteria and Requirements for the Windows GUI (DocScript^IDE^) were:

- [Windows GUI] An IDE with text-editing and script-running abilities
- [Windows GUI] A simple, familiar graphical design

With these in mind, I have designed this ↓ as the **Main Window**:

User Interface Modelling

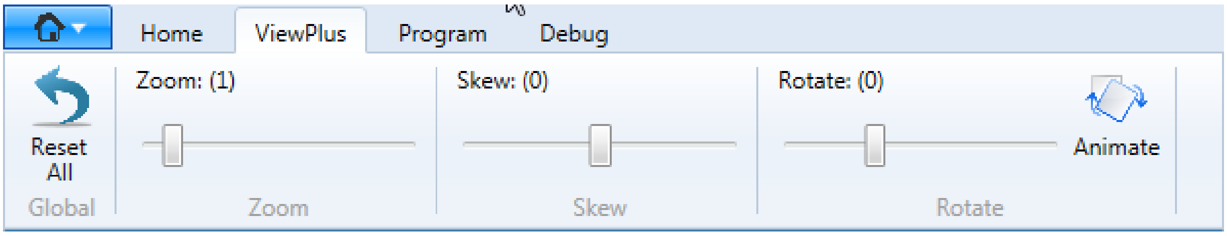
This mock-up was made with the *Visual Studio XAML Designer* and *.NET Ribbon SDK*:

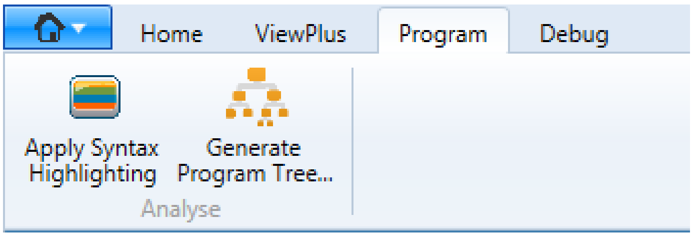


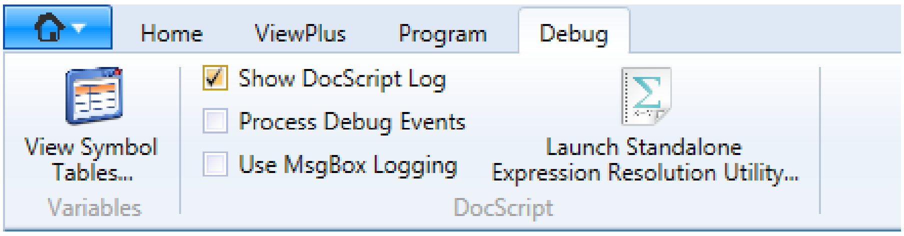
```
Function <Number> Main (<String@> _CLAs)

    #Input looks like "-Name" "Ben" "-Age" "13"
    #Get the Value for an Input()'ed Key

    <String> _Key : Input("Argument Key:")
    <String> _Value : GetCLAValueFromKey(_Key)

    Output("Value: " & _Value)
    Return 0

EndFunction

Function <String> GetCLAValueFromKey(<String@> _CLAs, <String> _Key)

    <Number> _CurrentCLAIndex : 0

    While (_CurrentCLAIndex < [Array_MaxIndex(_CLAs) + 1])
        If (Array_At(_CLAs, _CurrentCLAIndex) = ["-" & _Key])
```

Status: Idle | 28 Line(s) | Line: 10, Col: 27 | Zoom: 100%

These are the other Ribbon Tabs:

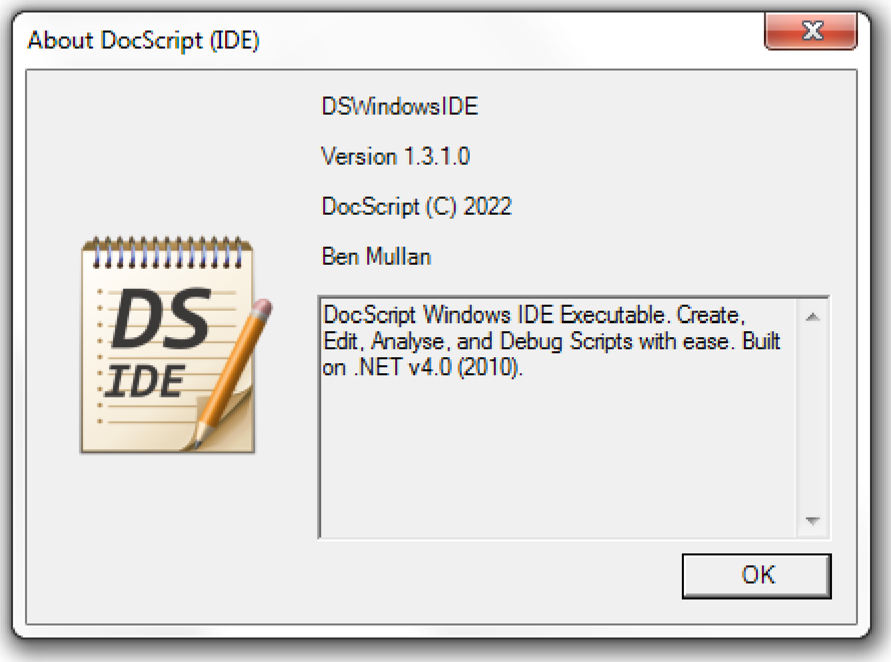






☑ Show DocScript Log
□ Process Debug Events
□ Use MsgBox Logging

An *About* Dialog:



About DocScript (IDE)

DSWindowsIDE

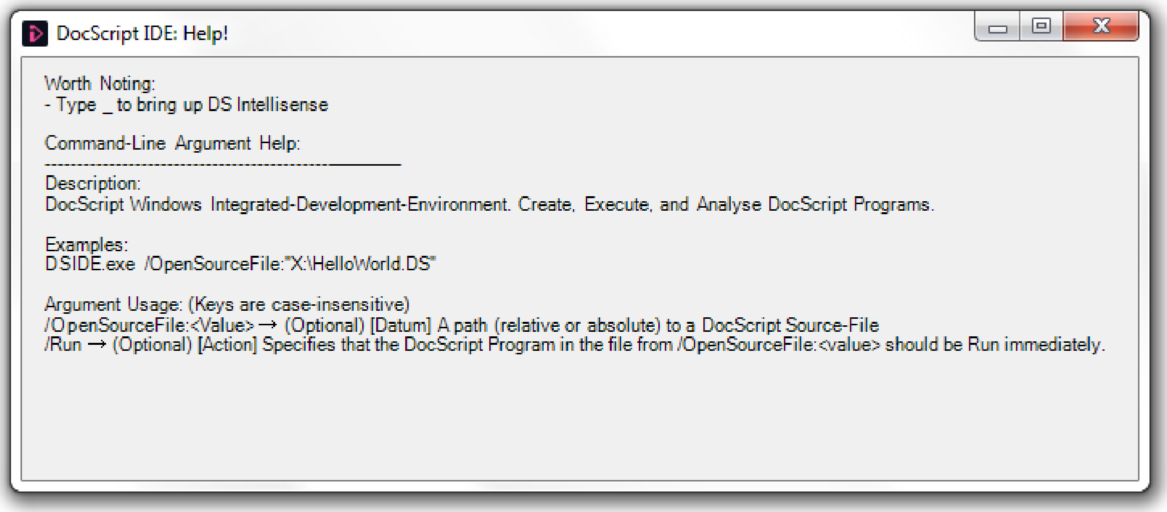
Version 1.3.1.0

DocScript (C) 2022

Ben Mullan

DocScript Windows IDE Executable. Create, Edit, Analyse, and Debug Scripts with ease. Built on .NET v4.0 (2010).

A *Help* Window:



DocScript IDE: Help!

Worth Noting:
- Type _ to bring up DS Intellisense

Command-Line Argument Help:

Description:
DocScript Windows Integrated-Development-Environment. Create, Execute, and Analyse DocScript Programs.

Examples:
DSIDE.exe /OpenSourceFile:"X:\HelloWorld.DS"

Argument Usage: (Keys are case-insensitive)
/OpenSourceFile:<Value> → (Optional) [Datum] A path (relative or absolute) to a DocScript Source-File
/Run → (Optional) [Action] Specifies that the DocScript Program in the file from /OpenSourceFile:<value> should be Run immediately.

(Consult the above images for what the following GUI Components look like…)

GUI Component Explanations & Justifications

GUI Component	Explanation & Justification
Run RibbonButton	The clear colour icon and standardised Keyboard shortcut herefor make this an easy-to-use feature, and it is in an obvious and well-labelled location. When interpretation begins, **all controls on the Window will be disabled**, to ensure that no other buttons are pressed whilst the program is busy interpreting the script. This is a form of validation.
Source RichTextBox	This control is clear and large so as to be easy to click on and enter text into.
InsertCodeSnippet RibbonButton	This makes is quick and easy for the user to start building DocScript programs. Amongst the possible insertables will be EntryPoint Functions, Statements, and full Sample Programs.
GenerateProgramTree RibbonButton	This button opens a new window which calls Program.GetProgTreeXML() to render the program onto a TreeView control, similar to the example provided @ *Instruction Trees*. It is a useful feature because it facilitates in-depth analysis of the program being written.
Status StatusRibbon	This control provides at-a-glance data concerning the current state of the program. It is needed because it indicates to the user that the program is busy, during interpretation, by means of the text "Status: Parsing…" and some percentage on the adjacent *ProgressBar*.

Where's the Validation?

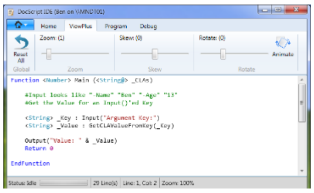
In choosing an appropriate variety of *Control* types, **the need to additional validation has been mitigated**. For instance, although the *Zoom* Slider could have been designed as a simple TextBox, into which a number could be entered, this would needlessly require supplementary validation: ensure only digits (0-9) have been entered; ensure there are an acceptable number of decimal places; ensure the value is in a valid range. Because it is a slider however, all these checks are – by design – implemented into the control and how it can be used.

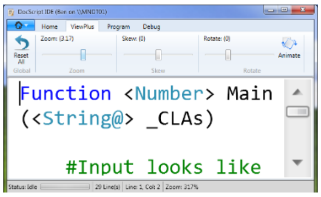
Axiomatically, the source text from the RichTextBox is – as described in the *Parsing* and *Lexing* sections - validated heavily when it is interpreted. Other areas of the DocScript system do also some require validation, which is covered later herein.

Usability Features

To make the application easier to use,

Usability Feature	Description & Justification
A profusion of Coloured Icons	These make the window easy to visually navigate, and aid in the user being able to quickly see which button they wish to click. Monochrome icons, on the other hand, would need to be stared at and deciphered, before any vague sense of their purpose could even be derived.
Zoom, Full Screen, and ViewPlus features	To make the source text easier to see, the zoom slider increases the scale of the text. This can be useful for people sitting far away from the monitor, or with poor

<table>
<tr>
<td></td>
<td>eyesight. Example:


Function &lt;Number&gt; Main (&lt;String@&gt; _CLAs)
#Input looks like "-Name" "Ben" "-Age" "13"
#Get the Value for an Input()'ed Key
<String&gt; _Key : Input("Argument Key:")
<String&gt; _Value : GetCLAValueFromKey(_Key)
Output("Value: " &amp; _Value)
Return 0
EndFunction
→


Function &lt;Number&gt; Main
(<String@&gt; _CLAs)
#Input looks like</td>
</tr>
<tr>
<td><b>Syntax Highlighting</b></td>
<td>To make the different source elements easier to read, Syntax Highlighting can be applied to the text. These are the colours which will be used by each TokenType:
<table>
<tr><th>TokenType</th><th>Colour</th></tr>
<tr><td>Unresolved</td><td>*</td></tr>
<tr><td>StringLiteral</td><td>"Value"</td></tr>
<tr><td>NumericLiteral</td><td>0110_2</td></tr>
<tr><td>BooleanLiteral</td><td>True</td></tr>
<tr><td>Keyword</td><td>Function</td></tr>
<tr><td>DataType</td><td>Number</td></tr>
<tr><td>Identifier</td><td>Main</td></tr>
<tr><td>DSOperator</td><td>&amp;</td></tr>
<tr><td>GrammarChar</td><td>(</td></tr>
<tr><td>LineEnd</td><td>*</td></tr>
<tr><td>StatementEnd</td><td>EndFunction</td></tr>
</table>
</td>
</tr>
<tr>
<td><b>The All-In-One Run (F5) Button</b></td>
<td>This button automates the process of Parsing, Lexing, and Executing the final Instruction Tree. The user need only click a singular item to invoke all three of these separate pieces of logic. It therefore makes the program easier to use.</td>
</tr>
</table>

The Windows CLI (DSCLI.EXE)

This is the simplest of all the implementations. The premise of this implementation is to enable DocScript programs to be easily run in an entirely automated fashion. This interpreter could be task scheduled from within Windows, for example, to run a DocScript program at a certain time each day.

The Criteria and Requirements for the Command-Line Interpreter were:

- [Windows CLI] Takes a Command-Line argument for the script to run
- [Windows CLI] Returns the Exit Code of the Script just run

I have hence decided that these are to be the Command-Line options for the DSCLI.EXE Program:

```
Description:
----------------------------------------
DocScript Command-Line Interpreter. Interprets DocScript Source Files.

Examples:
----------------------------------------
DSCLI.EXE /RunSourceString:"Function <Void> Main ();Output(`Hello, World!`);EndFunction"
DSCLI.EXE /RunSourceFile:"X:\Programming\DocScript\HelloWorld.DS"

Argument Usage:
----------------------------------------
/RunSourceFile:<Value>        Interprets the specified DocScript Source File
/RunSourceString:<Value>      Interprets the specified DocScript Source String.
                              Use ; for NewLine and ` for StringLiteralStartEndChar.
/ShowLog                      Writes Events from the DocScript Log to the Console Output
                              Stream during Interpretation
/ProcessDebugEvents           Processes and shows Debugging Messages in the Log
                              (if the Log is shown)
```

Explanation & Justification: For running impromptu scripts, remote execution (one of the stakeholder use cases), and experimentation scenarios, it is convenient to be able to execute some source directly, without having to save it to a file. \Windows\System32\CScript.exe, for example, does not support this immediate style of execution – the Script must be saved to a file and run in the fashion `cscript.exe Script.VBS`. \Windows\System32\MSHTA.EXE however, *can* execute text-only scripts in the fashion `mshta.exe VBScript:MsgBox("Hello")`. Therefore, DocScript will implement this useful feature too.

The Web Console (DocScript Interactive)

This is the most multifaceted of all the implementations. With DS Interactive, I have the opportunity to create a *distributed, collaborative, real-time, multi-client execution environment*, and to do something quite original with it. The goal is to be able to host an *Execution Session* on the Server, and to have multiple clients *tune in* to the Session. Each client will be able to see Program Output and LogEvents, and can also respond to Input requests (generated by calls to `Input()` in the DocScript source).

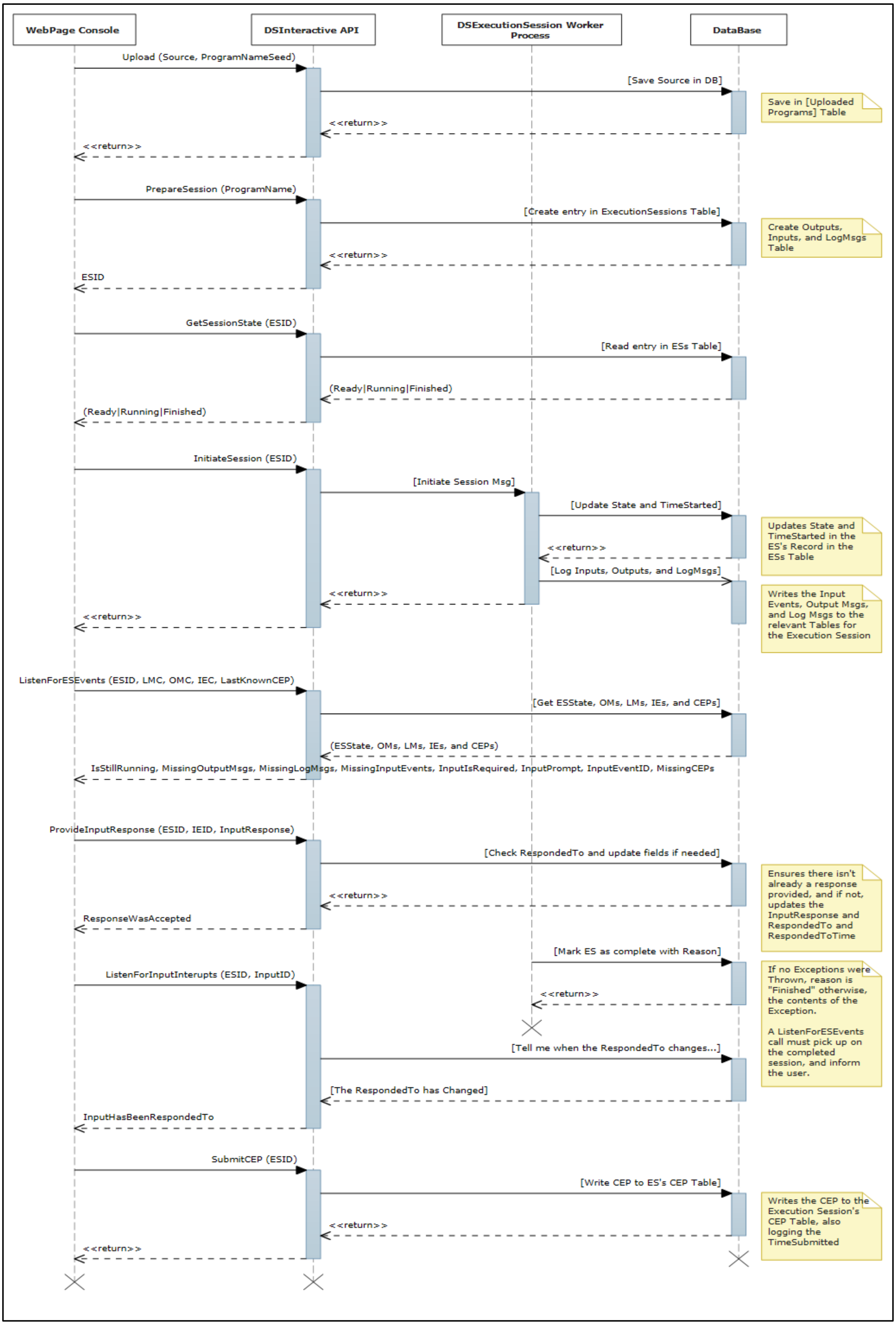
The Criteria and Requirements for the Web Client Implementation were:

- [Web Client] Runs on a variety of different browsers on different devices
- [Web Client] Allows the user to enter source code into a text field and thereafter execute it

I have therefore designed this API:

(See next page…)

DSInteractive API Sequence Diagram


WebPage Console
DSInteractive API
DSExecutionSession Worker Process
DataBase
Upload (Source, ProgramNameSeed)
[Save Source in DB]
Save in [Uploaded Programs] Table
<<return>>
<<return>>
PrepareSession (ProgramName)
[Create entry in ExecutionSessions Table]
Create Outputs, Inputs, and LogMsgs Table
<<return>>
ESID
GetSessionState (ESID)
[Read entry in ESs Table]
(Ready|Running|Finished)
(Ready|Running|Finished)
InitiateSession (ESID)
[Initiate Session Msg]
[Update State and TimeStarted]
Updates State and TimeStarted in the ES's Record in the ESs Table
<<return>>
[Log Inputs, Outputs, and LogMsgs]
Writes the Input Events, Output Msgs, and Log Msgs to the relevant Tables for the Execution Session
<<return>>
<<return>>
ListenForESEvents (ESID, LMC, OMC, IEC, LastKnownCEP)
[Get ESState, OMs, LMs, IEs, and CEPs]
(ESState, OMs, LMs, IEs, and CEPs)
IsStillRunning, MissingOutputMsgs, MissingLogMsgs, MissingInputEvents, InputIsRequired, InputPrompt, InputEventID, MissingCEPs
ProvideInputResponse (ESID, IEID, InputResponse)
[Check RespondedTo and update fields if needed]
Ensures there isn't already a response provided, and if not, updates the InputResponse and RespondedTo and RespondedToTime
<<return>>
ResponseWasAccepted
[Mark ES as complete with Reason]
If no Exceptions were Thrown, reason is "Finished" otherwise, the contents of the Exception.
A ListenForESEvents call must pick up on the completed session, and inform the user.
ListenForInputInterupts (ESID, InputID)
<<return>>
[Tell me when the RespondedTo changes...]
[The RespondedTo has Changed]
InputHasBeenRespondedTo
SubmitCEP (ESID)
[Write CEP to ES's CEP Table]
Writes the CEP to the Execution Session's CEP Table, also logging the TimeSubmitted
<<return>>
<<return>>

DSI DataBase Tables

The comments in the above sequence diagram make reference to some DataBase Tables, which looks like this:

↓ UploadedPrograms Table ↓

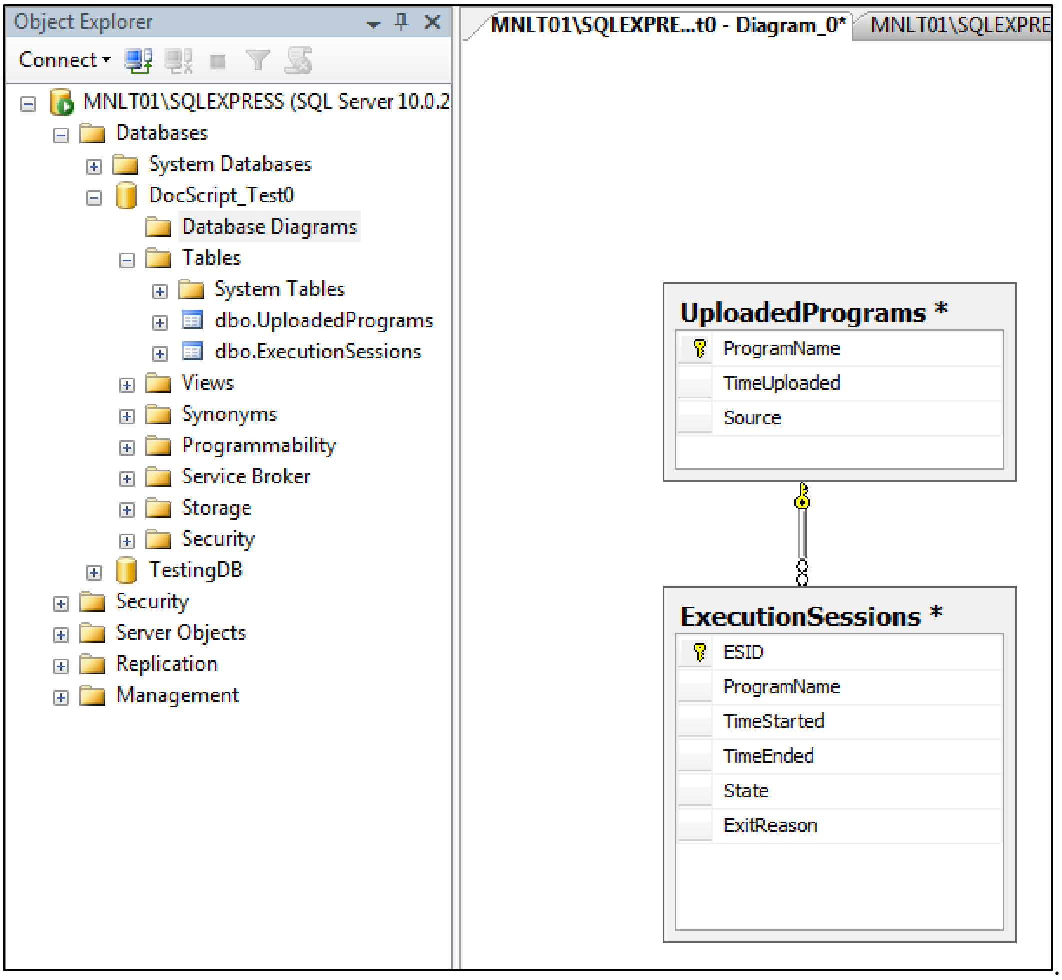
TimeSubmitted	Source	🔑 ProgramName
1:32:49 12-08-2022	`Function <Void> Main () …`	HelloWorld
1:32:49 12-08-2022	`Function <Void> Main () …`	InputName
1:32:49 12-08-2022	`Function <Void> Main () …`	AgeTest
1:32:49 12-08-2022	`Function <Void> Main () …`	Program1
1:32:49 12-08-2022	`Function <Void> Main () …`	PROGRAM3

↓ ExecutionSessions Table ↓

🔑 ESID	🔒 ProgramName	TimeStarted	TimeFinished	State	ExitReason
HELLO_AH42	HelloWorld.DS			Ready	
HELLO_AH43	HelloWorld.DS	1:32:49 12-08-2022		Running	
HELLO_AH44	HelloWorld.DS	1:32:49 12-08-2022	1:32:49 12-08-2022	Finished	Finished Successfully
HELLO_AH45	HelloWorld.DS	1:32:49 12-08-2022	1:32:49 12-08-2022	Finished	Input Timed Out for "…"
HELLO_AH46	HelloWorld.DS	1:32:49 12-08-2022	1:32:49 12-08-2022	Finished	Finished Successfully

Entity Relationship Diagram

This is the relationship between the `UploadedPrograms` and `ExecutionSessions` Tables; the **Primary Key** ProgramName becomes a **Foreign Key** in the ExecutionSessions Table. This is a one-to-many relationship; one ProgramName becomes many fields in UploadedPrograms:



- MNLT01\SQLEXPRESS (SQL Server 10.0.2
  - Databases
    - System Databases
    - DocScript_Test0
      - Database Diagrams
      - Tables
        - System Tables
        - dbo.UploadedPrograms
        - dbo.ExecutionSessions
      - Views
      - Synonyms
      - Programmability
      - Service Broker
      - Storage
      - Security
    - TestingDB
  - Security
  - Server Objects
  - Replication
  - Management

UploadedPrograms *

- 🔑 ProgramName
- TimeUploaded
- Source

ExecutionSessions *

- 🔑 ESID
- ProgramName
- TimeStarted
- TimeEnded
- State
- ExitReason

There is one instance of each of the following Tables per ExecutionSession…

↓ ExecutionSession Outputs Table ↓

🔑 ID	TimeSubmitted	OutputMessage
1	1:32:49 12-08-2022	"Enter Name"
2	1:32:49 12-08-2022	"Enter Age"
3	1:32:49 12-08-2022	"Enter A"
4	1:32:49 12-08-2022	"Enter B"
5	1:32:49 12-08-2022	"Enter C"

↓ ExecutionSession Inputs Table ↓

🔑 ID	TimeSubmitted	InputPrompt	InputResponse	RespondedTo	RespondedToTime
1	1:32:49 12-08-2022	"Enter Name"	"Ben"	True	1:32:49 12-08-2022

2	1:32:49 12-08-2022	`"Enter Age"`		False	1:32:49 12-08-2022
3	1:32:49 12-08-2022	`"Enter A"`		False	1:32:49 12-08-2022
4	1:32:49 12-08-2022	`"Enter B"`		False	1:32:49 12-08-2022
5	1:32:49 12-08-2022	`"Enter C"`		False	1:32:49 12-08-2022

↓ ExecutionSession LogMsgs Table ↓

🔑 ID	TimeSubmitted	LogMessage	Severity	Category
1	1:32:49 12-08-2022	`"Enter Name"`	Error	Execution
2	1:32:49 12-08-2022	`"Enter Age"`	Warning	Lexing
3	1:32:49 12-08-2022	`"Enter A"`	Information	Parsing
4	1:32:49 12-08-2022	`"Enter B"`	Verbose	Unspecified
5	1:32:49 12-08-2022	`"Enter C"`	Debug	System

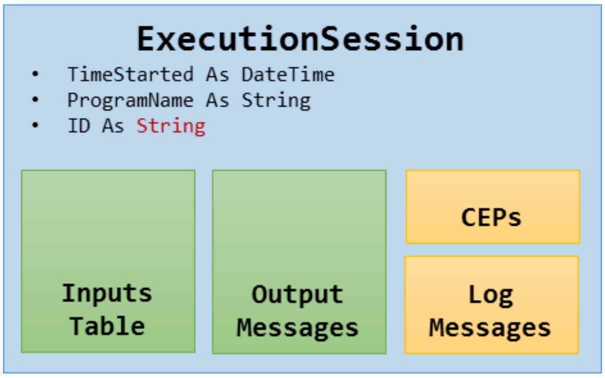
↓ ExecutionSession Client Execution Packages (CEPs) Table ↓

🔑 ID	TimeSubmitted	JavaScriptToRun
1	1:32:49 12-08-2022	function () { … }
2	1:32:49 12-08-2022	function () { … }
3	1:32:49 12-08-2022	function () { … }
4	1:32:49 12-08-2022	function () { … }
5	1:32:49 12-08-2022	function () { … }

NOTE: IDs start at 1 (not 0). This is so that 0 can be used to mean [none of the records].

The ExecutionSession Object

Those API EndPoints and DataBase Tables make an ExecutionSession look broadly like this:



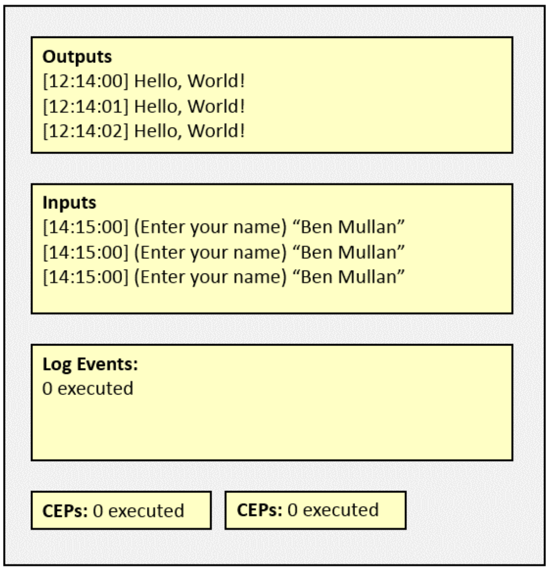
Explanation: By having a robust and comprehensive API like this, implementing the Client Pages (the ones with HTML that the browser actually loads) becomes almost trivial; the client need only *make the correct API request*. Many of the API EndPoints are for AJAX LongPolling, meaning that once a request is made (E.g. `/API/Interactive/?Action=AwaitExecutionSessionInitiation&ESID=HELLO_AH42`), the server only returns a response when it is worth doing so (E.g. as soon as the ExecutionSession with ID “HELLO_AH42” has its State change to “Running”).

Justification: Without these long-polling EndPoints, the client would have to make a high volume of requests, which would indubitably impact server performance. Pertaining to the aforementioned example, the client would have to make an `/API/Interactive?Action=GetSessionState&ESID=HELLO_AH42` request every second or so, instead of just the one long-polling request.

Front-end Pages

I have mocked-up the following for the user interfaces, in accordance with the stakeholder criteria of needing a simple, functional, and familiar style.

Interactive/ESParticipant



Outputs
[12:14:00] Hello, World!
[12:14:01] Hello, World!
[12:14:02] Hello, World!

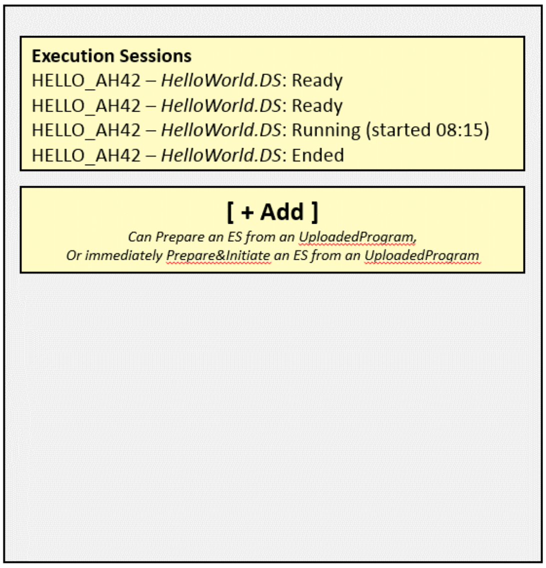
Inputs
[14:15:00] (Enter your name) “Ben Mullan”
[14:15:00] (Enter your name) “Ben Mullan”
[14:15:00] (Enter your name) “Ben Mullan”

Log Events:
0 executed

CEPs: 0 executed

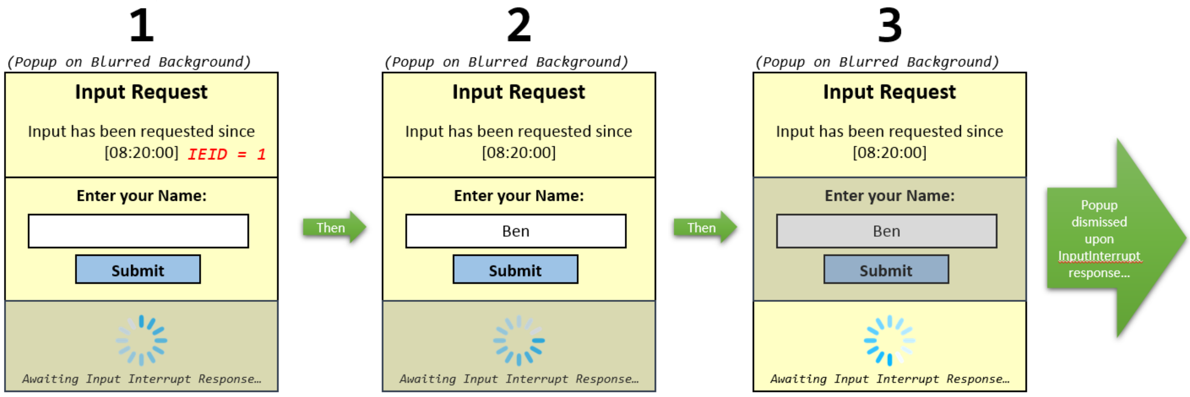
CEPs: 0 executed

Interactive/ESManager



Execution Sessions
HELLO_AH42 – *HelloWorld.DS*: Ready
HELLO_AH42 – *HelloWorld.DS*: Ready
HELLO_AH42 – *HelloWorld.DS*: Running (started 08:15)
HELLO_AH42 – *HelloWorld.DS*: Ended

[+ Add]
Can Prepare an ES from an UploadedProgram,
Or immediately Prepare&Initiate an ES from an UploadedProgram


1
(Popup on Blurred Background)
Input Request
Input has been requested since [08:20:00] IEID = 1
Enter your Name:
Submit
Awaiting Input Interrupt Response…
Then
2
(Popup on Blurred Background)
Input Request
Input has been requested since [08:20:00]
Enter your Name:
Ben
Submit
Awaiting Input Interrupt Response…
Then
3
(Popup on Blurred Background)
Input Request
Input has been requested since [08:20:00]
Enter your Name:
Ben
Submit
Awaiting Input Interrupt Response…
Popup dismissed upon InputInterrupt response…

[InputIsRequired] comes back as True; the Client sends a ListenForInputInterupts request…

Client enters InputResponse and clicks Submit

Client must await InputInterrupt Response…

[Above] The DSInteractive Input-requesting procedure

Server-side Validation

Each EndPoint on the API takes in a number of different QueryString Parameters. For example: The EndPoint…

```
/API/Interactive/ExecutionSession.ASPX?Action=GetSessionState&ESID=HELLO_AH42
```

…takes in the QueryStrings `Action` and `ESID`.

For each QueryString passed to an API EndPoint, I will need to validate that it is in an expected syntactical format. I will do this primarily via Regular Expressions. For instance, the ESID will be validated against the RegExp `^\w{10}$`.

One of the most important initial stages of the **validation** will be to ensure that each QueryString specified in the URL is properly-formed, with a corresponding value. In other words, each QueryString must be in the format `?{Key}={Value}`, with `&` used to join one QueryString KeyValuePair to the next. This means that the QueryStrings must match this Regular Expression:

```
^\?([A-Za-z0-9]+=\w+&)*$
```

In this way, it is guaranteed that none of the QueryStrings will be specified without a value, and that none of them are malformed. For instance, these QueryStrings would *not* pass the RegExp validation:

- Name=
- Name
- Age=&
- Age&

Stakeholder Input

To ascertain whether or not these design decisions I have made about the *Formal Language Specification*, *DocScript Architecture*, and *DocScript Implementations* are suitable for the stakeholders, *and* to ensure that this is broadly what they may have had in mind, I showed this

Design document and several drawings to the stakeholders, and asked for their comments – which I have **summarised as follows**:

- The Formal Lang. Spec. describes exactly the sort of thing the Stakeholders were looking for; simple, consistent, easy to get to grips with (only the 6 keywords etc…)
- The Architecture in its extensible nature is promising and provides possibility for potential future expansion of the system
- The `/RunSourceString:<Value>` system on the DSCLI will be very useful to some of the Stakeholders (*Clara and Joe*)
- The multi-client features of DocScript Interactive look very exciting and could be a very powerful educational tool

(The consensus is that I'm ratified to *continue* the development of the system…)

Testing

This section describes the fashion by which the entire DocScript Solution will be tested. It will be applicable both during the development, and thereafter, during the evaluation.

Inputs & Outputs

The main **areas where Input is received**, are:

- DocScript Source
- (…Including sometimes standalone Expressions)
- API URL Calls
- Command-Line Arguments

The main **areas where Output is displayed**, are:

- `Output()` Messages from DocScript Programs (MsgBoxes in DocScript^{IDE}, Console text in DSCLI)
- API Responses
- LogEvents

Testing Techniques

Unit Testing

I shall use **Unit Tests** in order to create automated tests for all areas of the project. This means that if I cause an unintended side effect by changing one function, and this impairs the operation of another function, then this will clearly show up in the Unit Test results. Visual Studio can *automatically* bulk-run Unit Tests for me, which saves time.

Destructive Testing

In addition to these routinely-run Unit Tests however, I shall also manually test the application **destructively** after I add each feature. This means deliberately attempting to break the system by entering malformed or potentially dangerous input, to see how this is handled. Suitable error messages need to be given, instead of the application crashing or becoming unresponsive.

Stakeholder Testing

Because I – *the programmer* – am aware of how the application works, what sort of input data are expected, *and* – which pieces of validation I *know* I have implemented, I might not always be the best person to actually test the software. The Stakeholder on the other hand, will end up using the final product, and they might expect it to work differently that the image I have in my head of it being used. They might, in other words, **do unexpected and unforeseen things**, which could have **unanticipated consequences**. Therefore, I shall – after every major incremental release of the compiled DocScript binaries – get the Stakeholders to test the application for themselves.

In order for me to know precisely what they did to cause a certain problem, I shall get the stakeholders to run the software on their own computers, and record their every click and keystroke with the built-in Windows Utility, **Problem Steps Recorder** – psr.exe. This tool automatically generates a full HTML report with screenshots, highlighted mouse clicks, and keystrokes, which I can easily replay to work out wherein the problem lies, with great acuity. It was, incidentally, one of the new Enterprise Toolkit Features to be shipped with Windows 7.

psr.exe

Test Data

These are the data I shall use for each of the four Input areas mentioned earlier…

DocScript Source

(One per Block)

```
# VALID: Simplest-possible DocScript Program
Function <Void> Main ()
    Return
EndFunction
```

```
# VALID: CLA Test
Function <Number> Main (<String@> _CLAs)
      Output("First CLA: " & StringArray_At(_CLAs, 0))
      Return 0
EndFunction
```

```
# INVALID: Malformed Function
Function <Void> Main ()
    Return
EndFunction
EndFunction
```

```
# INVALID: 2nd Line not syntactically-valid
Function <Void> Main ()
      {Output(2)}
EndFunction
```

Standalone Expressions

(One per Line)

```
# VALID
12
```

```
12_10
12.0
12.0009
12_4 * ~9
"Hello, World!"
"Hello, World!" & " More Text"
GetName(17) & GetAge("Ben")
Ident
F(A + B * [C - D]) & [E ¦ ¬F ' G]

# INVALID
12_2
"H
~4 + 0~
F(A + B * (C – D)) & [E ¦ ¬F ' G]
GetName[17]
```

API URL Calls

(One per Line)

```
# VALID
/API/Interactive/?Action=GetSessionState&ESID=HELLO_AH43
/API/Interactive/Upload.ASPX?Item=Source&ProgramNameSeed=Hello%20World

# INVALID
/API/Interactive/?ESID=HELLO_AH43
/API/Interactive/Upload.ASPX?Item=Source&ProgramNameSeed
```

Command-Line Arguments

(One per Line)

```
# VALID
/RunFile:"X:\Programming\DocScript\HelloWorld.DS"
/RunFile:HelloWorld.DS /ShowLog /ProcessDebugEvents
/RunSourceString:"Function <Void> Main ();Output(`Hello, World!`);EndFunction"

# INVALID
/RunFile:"X:\Programming\DocScript\HelloWorld.DS /ShowLog
/RunFile:
/RunSourceString:"Function <Void> Main () Output("Hello, World!") EndFunction"
```

Explanations & Justifications

The wide range of testing data specified here will enable me to effectively invoke all possible *corners* (*so to speak*) of the input-processing algorithms. This will thereby ensure that each component of logic within the Library DLL and Implementation EXE is functional and works as is intended.

I do not need to *exhaustively* test **every possible value** of input, but rather, **every possible format**. For example, the same parts of the Command-Line Argument processing algorithm are invoked for the input values `/RunFile:"HW.DS"` and `/RunFile:"WH.DS"`; the second test adds no value and is futile. However, the different syntaxes `/RunFile:"HW.DS"` and `/RunFile:HW.DS` ought to **both** be tested, because they are parsed slightly differently and therefore different side effects could befall in subsequent stages of the program. My chosen Testing Data are designed to test every part of the algorithms, without repeating any logically-identical values.

Testing Checklist (Interpreter DLL)

After the majority of the development and implementation is complete, *myself* and the *Stakeholders* shall perform each of the following tests, to ensure that the requirements have been met:

Assert-style Test	Passed?
An error is thrown if two variables are declared within the same (or relative downstream) scope, when both variables have the same identifier, or the identifiers differ only by case; the language is **not** case-sensitive	*
Variables can be declared in each of the 6 valid DataTypes (The 3 DataTypes, and their Array variants)	*
A high degree of verbosity is present in the error message resultant of an attempt to lexically analyse the expression (e.g.) `5 + + 4`	*
A Function can be declared with IInstruction-based contents statements inside, such as an IfStatement `If (Expr) {LineEnd} … EndIf`	*
A Global variable can be declared outside of any Functions (E.g. `<String> NameGlobal = "Ben Mullan; S7; Y13; U6;"`)	*
An error is thrown if a Program is written without an EntryPoint Function `Main`	*
A DocScript Program can be written to output "B" if Command-Line Argument [0] is "1", and "C" if it is "2" (Just an example, to prove that CLAs can affect programme output)	*
A DocScript program can be written to `Output()` "Hello, World!"	*
A DocScript program can be written to take `Input()` from the user	*
A Comment can be specified with `# Comment {LineEnd}`	*
The Expr `[5 + 3] * 9` resolves to 72 whereas `5 + [3 * 9]` resolves to 32 (proof that brackets work)	*
The numeric literals `10`, `10.0`, and `10_10` are all magnitudionally equivalent	*
All of the Test Data {Above} run successfully in the DocScript System & Implementations	*

Explanation: Each of these is an assert-style test; one attempts to **disprove** the statement, and on failing to do so, declares that the system/component has **passed** the test.

Justification: (The justifications for these are essentially the entirety of the preceding document…)

Post-Development

After the **MVP** (*Minimum Viable Product*) described herein has been developed to at least **RC0** (the first *Release Candidate*), the subsequent stages of development are likely to become increasingly indistinct; because the product works to *some* extent, but is constantly being r*efined* and *improved*, it is difficult to know when to call it *"done"*…

Post-Development Testing

…To aid, therefore, in this process of **long-term maintenance and improvement** for the product, I shall declare here some testing data to be used during the post-release development.

Explanation & Justification: These are some of the principle reasons for why it is important that I perform the post-development testing and maintenance:

- **Speed:** The DocScript system implementations could accumulate encumbering quantities of data over time, which could slow the software down.
- **Certainty:** After each new incremental release of the software, I shall continue to run the Unit Tests on all releases, to make sure that the individual features work as is intended.
- **Protecting Reputation:** If the quality of user experience were to deteriorate over time, this could be deleterious for the DocScript software (and brand)'s reputation. Ongoing maintenance is needed to establish and maintain this reputation, particularly for the Web Implementation (*DSInteractive*) because web technologies and standards change very quickly, and insultingly-modern users will begin to look *askance* on older (but perfectly functional) methods such as raw AJAX, or jQuery.

Data to be Used Herefor

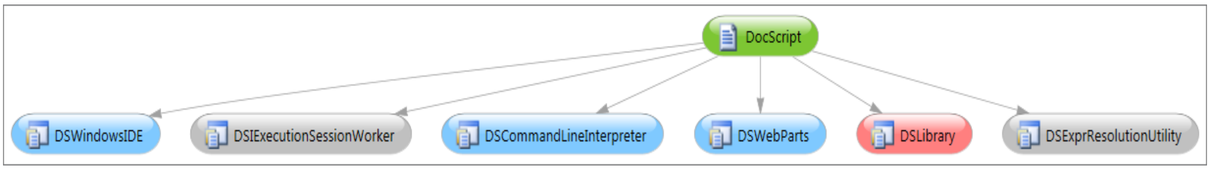
Although the system should always remain compatible with all the data stated hereinbefore, there are a number of **additional data, relevant exclusively to the post-development phase**:

- DocScript Programs written by the Stakeholders during their initial usage of the system
- Input commands being sent from different operating system clients to the Command-Line Interpreter running on a Remote Network Server (I only use Windows 7/Server 2008 R2, but the Stakeholders may use some other operating systems – e.g. which interact differently with PsExec or Telnet, for remote CLI usage)
- Input data which previously caused an error to occur
- Data which are the result of a change to the environment under which the DocScript software operates
- Obsolete and older Unit Tests from early project development – to be run every once in a while, but not on every release compilation because this would be too slow

Explanation: If someone were to develop the DocScript further, they would have scope to do this because of the extensible way in which DocScript and its components have been designed and built. The above data sources would be used to develop, test, and extend DocScript. Furtherance of the system is permitted.

Justification: If the DocScript system were to become legacy code (which in due time is an *inexorableness*), then post-development reimplementation and refactoring would occur to reimplement the “old” source code into new binaries, perhaps written in a different programming language. The mentioned testing data sources are suitable for this as they are applicable and pertinent not only to present data collection methods, but to potential future methods too.

This diagram ↓ shows the binaries of the DocScript Solution (.*sln*). The majority of testing must occur on the red nodes, because the other binaries are dependent on the base DLLs. All the *.exe binaries take Command-Line Arguments which will be tested with the specified data, and the DS Library DLL is tested with all the DocScript source tests and sample Expressions. The Web Parts (*Interactive*) are also to be tested with the aforementioned URLs.



*[**Blue** = Implementation Binary]   [**Grey** = Utility Binary]   [**Red** = Library DLL]*

Development and Testing

Overview

The *Design* stage has left me with a clearly-defined, vigorously-evaluated, and extensibly-structured specification for the DocScript Programming Language. To effectuate and corporealize this plan – however – I shall work through the following stages…

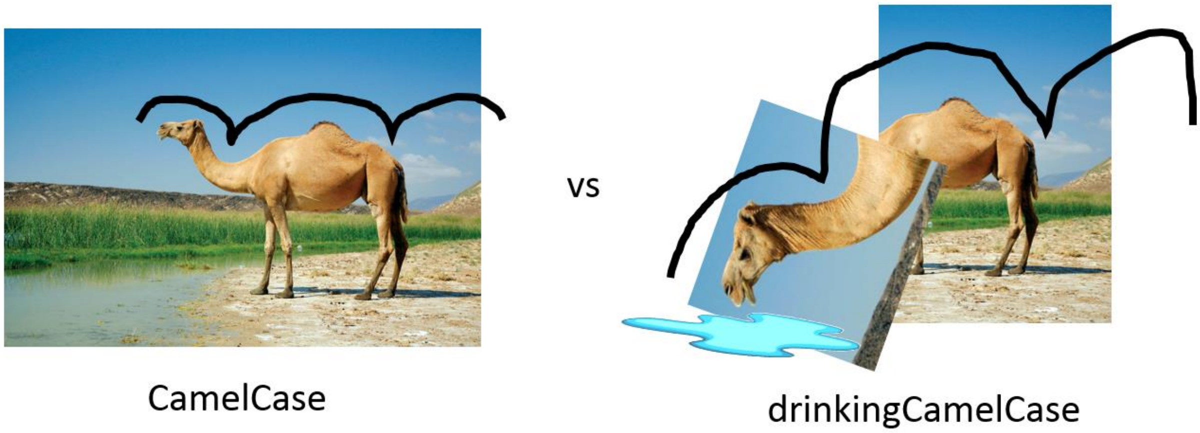
1) Develop the **Core Interpretation Engine DLL**
    a. Write the Parser (Source → Tokens)
    b. Write the Lexing logic (Tokens → Program)
    c. Write the Execution Logic (Program → {Output})
2) Develop the **Command-Line Interpreter** (DSCLI) Implementation
3) Develop the **Windows IDE** (DSIDE) Implementation
4) Develop the **Web-based** (DSInteractive) Implementation
5) Test the entire system with different **Programs** and **Scenarios**

Programming Conventions Used Herein

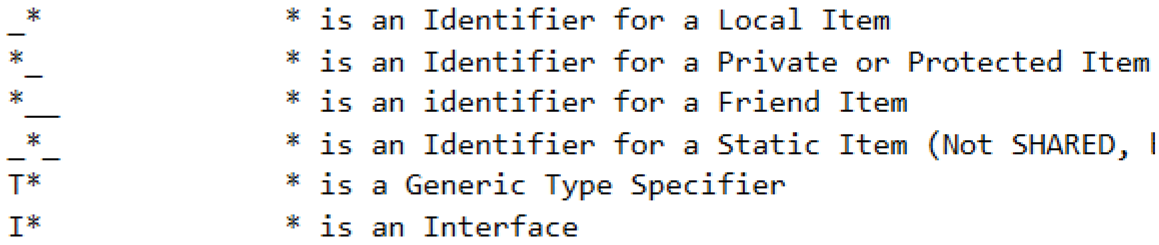
Allow me to briefly elucidate *how* the 75,894 lines of code I am about to write are structured…

Naming Conventions

I shall continue to use the beautiful and unswerving **CamelCase** style, for all identifiers. Note that this is **different from drinkingCamelCase** in the following way:

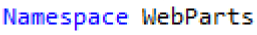


In addition, I employ the following **conventions** for identifiers within the solution:

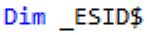


```
_*              * is an Identifier for a Local Item
*_              * is an Identifier for a Private or Protected Item
*__             * is an identifier for a Friend Item
_*_             * is an Identifier for a Static Item (Not SHARED, I
T*              * is a Generic Type Specifier
I*              * is an Interface
```

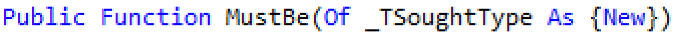
Here are some **examples**:



`Namespace WebParts`,



`Dim _ESID$`,



`Public Function MustBe(Of _TSoughtType As {New})`

Explanations and Justifications

I maintain vehemently that this is not pedantry. By employing these conventions, I can determine – just at a glance – precisely *where* I am able to use a given variable, and *what* a given identifier is for (Object, Type, Namespace, Class/Structure, Interface, or Type-Parameter). This provides a veritable improvement to the speed of development, and the clarity and lucidity of the code written.

Incidentally, single-letter identifiers (e.g. `i`) are **NEVER** admissible in my ruling; this is constitutes wanton laziness.

Modularity and Encapsulation

The solution is modular in the following ways:

- The Library is written as a separate DLL to the other assemblies/binaries which implement it
- Each assembly is split into a number of different `Namespace`s
- Each `Namespace` may contain several `Class`es, `Structure`s, `Module`s, `Interface`s, `Delegate`s, and `Enum`s
- The `Class`es and `Module`s contain both `Private`/`Protected` and `Public` members

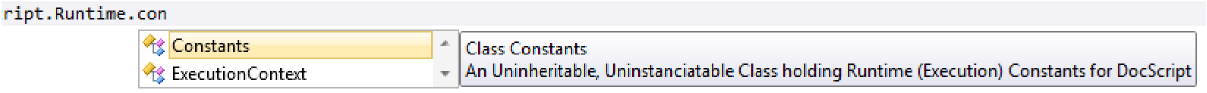
Explanations and Justifications

This tree-like, hierarchical structure for the solution, means that it is easy to find any given item, by logically following the path one would expect to lead to it. It also leads to a good level of discoverability of items; if one were looking to examine the structure of the DocScript Runtime system, one would look in `Namespace Global.DocScript.Runtime` – that much would be axiomatic I concede, but the modularity is certainly *conducive* to an ease-of-navigation.

Comments and Annotations

The Visual B.A.S.I.C. .NET Programming Language supports – I contend – **four types of comments**. Readers will be glad to hear that I possess a rigorous and pedantic system for the use of all 4 types, for different purposes…

1. **Ticks**: `'Comment`
   I use these for informal, quick notes.
2. **REMs**: `REM Comment`
   I use these for formal, properly-worded descriptions, including multi-line planning and breakdown for how a complex Sub of Function should work.
3. **If-False Blocks**: `#If False Then {LineBreak} Comment {LineBreak} #End If`
   This is admittedly a sort of *hack*, but is useful sometimes when I want to type lengthy comments with many interstitial line-breaks.
4. **XML Documentation**: `''' <summary>Comment</summary>`
   This is the most powerful form of comment, providing Intellisense text for the components I create. For example:



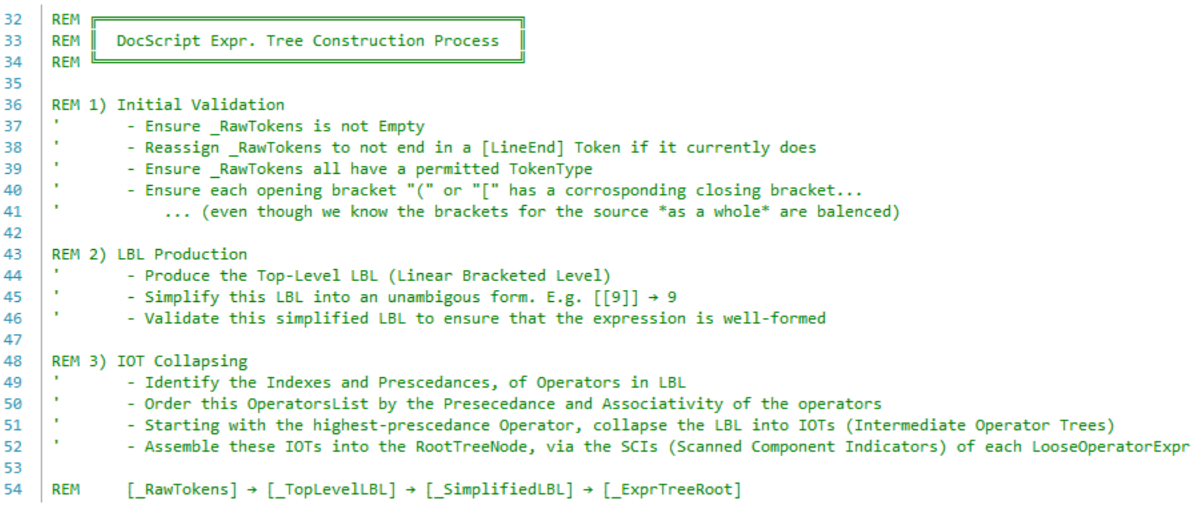
```
ript.Runtime.con
```

Class Constants
An Uninheritable, Uninstanciatable Class holding Runtime (Execution) Constants for DocScript

Explanations and Justifications

It is vital to stress the following points regarding the use of comments:

- Comments should not be written to explain **what** is being done. This much should be obvious from the code and identifiers; variable names should be chosen to make the source read like a comment. E.g. It would be mindless and inane to write `'Beeps if the input is nothing` above the line `If _Input Is Nothing Then Beep()`; that code is already a very nice English sentence, and this linguistic register of programming is afforded to me only in Visual Basic .NET. I know of no other language which provides the same clarity.
- Comments should be written, only to explain **why** or **how** a process occurs. For example, I find this sort of thing useful, at the top of a sizable Function:



```
REM ╔═══════════════════════════════════════╗
REM ║  DocScript Expr. Tree Construction Process  ║
REM ╚═══════════════════════════════════════╝

REM 1) Initial Validation
'       - Ensure _RawTokens is not Empty
'       - Reassign _RawTokens to not end in a [LineEnd] Token if it currently does
'       - Ensure _RawTokens all have a permitted TokenType
'       - Ensure each opening bracket "(" or "[" has a corrosponding closing bracket...
'           ... (even though we know the brackets for the source *as a whole* are balenced)

REM 2) LBL Production
'       - Produce the Top-Level LBL (Linear Bracketed Level)
'       - Simplify this LBL into an unambigous form. E.g. [[9]] → 9
'       - Validate this simplified LBL to ensure that the expression is well-formed

REM 3) IOT Collapsing
'       - Identify the Indexes and Prescedances, of Operators in LBL
'       - Order this OperatorsList by the Presecedance and Associativity of the operators
'       - Starting with the highest-prescedance Operator, collapse the LBL into IOTs (Intermediate Operator Trees)
'       - Assemble these IOTs into the RootTreeNode, via the SCIs (Scanned Component Indicators) of each LooseOperatorExpr

REM     [_RawTokens] → [_TopLevelLBL] → [_SimplifiedLBL] → [_ExprTreeRoot]
```

These commentary principles will be observable in the forthcoming screenshots of the DocScript Implementation.

Writeup Segments

The subsequent documentation of §Development makes interspersial use of these boxes, for particularly important considerations:

Progress Recap: Where am I in the development plan? *[Review]*

Prototype: This point marks a milestone in the product, which is now capable of…

Structure and Modulatory: This DSI Database is well-structured, because…

Validation: The following ensure that data of the correct variety is present in the …

Testing Table: Does this component function in accordance with the stipulated criteria?

{Yellow Progress Check Boxes, shoring the current line count for the whole solution}

[Stage 1]

Core Interpretation Engine (DLL) Development

(As a reminder, this DLL is the ***implementee***; it contains all the logic needed for parsing, lexing, and executing a DocScript Program, but not, for instance, implementation-specific user-interface components.)

Structure and Modulatory: This DLL (DocScript Library) is well-structured, because…

- It implements the principle of pipelining; the output of one operation is the *input to the next one*. In the case of DocScript, the pipeline looks like [Source → Tokens → Program → {Execution}]. As I mentioned back in §Design, that looks like this to an implementer of the DLL:

  ```
  'Raw
  Dim _Source As [String] = "..."

  'Parse
  Dim _Tokens As DocScript.Runtime.Token() = _
          DocScript.Runtime.Parser.GetTokensFromSource(_Source)

  'Lex
  Dim _Program As New DocScript.Runtime.Program(_Tokens)

  'Execute
  _Program.Run({})
  ```

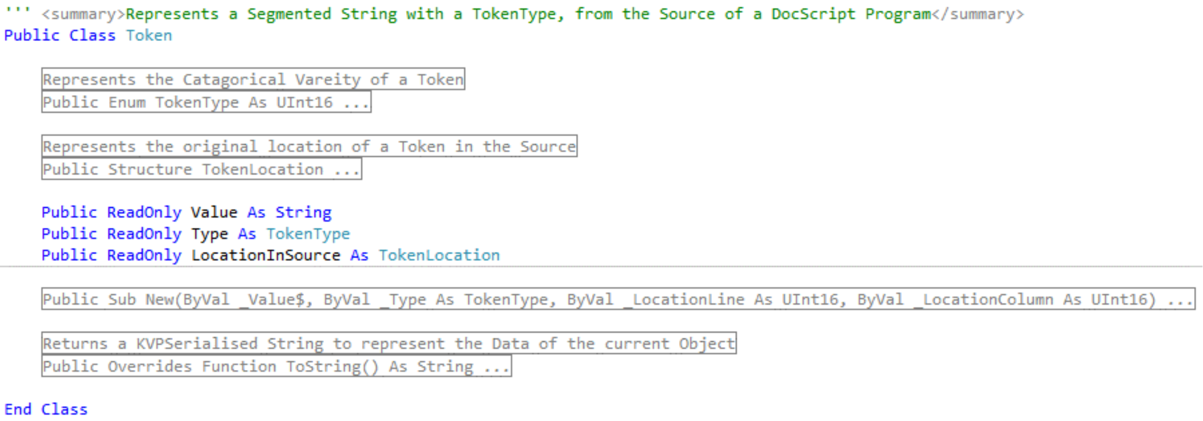
  Normally, pipelining is performed with the pipe operator `|`. For instance, the Windows® shell command `ipconfig | find "Address" | clip` gets the output from *systeminfo.exe*, pipes it into *find.exe*, and thereafter pipes the output from find, into *clip.exe*. It is a chain of small programs working together.
- There is a clear tree-like, hierarchical structure, whereby the Solution contains multiple projects, each of which have several namespaces, which themselves each contain numerous classes holding a litany of Functions and Statements.

Parser

This sub-section of the DLL lives in the `Runtime` Namespace, and has the main role of taking in a raw Source-Code String, to validate, analyse, and derive a series of `Token`s from.

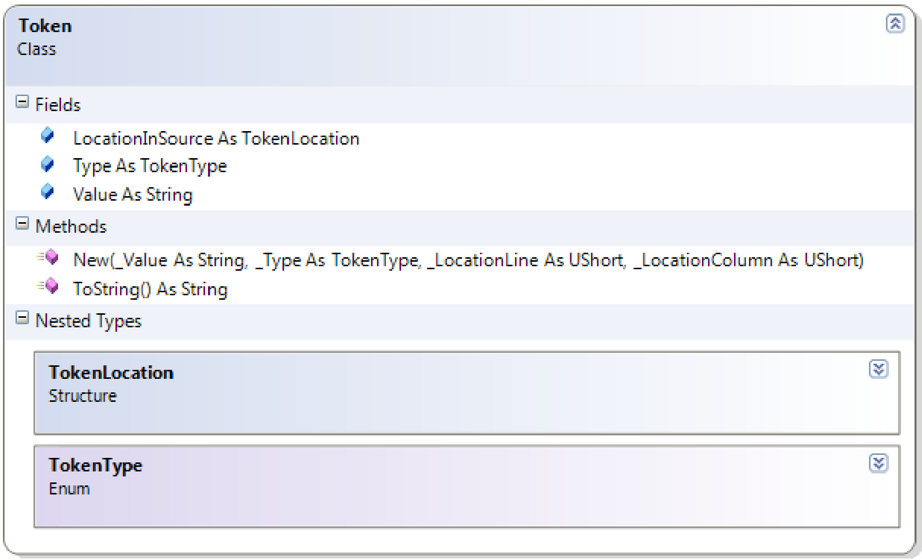
The Token Class

It befits me to begin by writing up the `Token` class, as it was defined in §Design:



```
''' <summary>Represents a Segmented String with a TokenType, from the Source of a DocScript Program</summary>
Public Class Token

    Represents the Catagorical Vareity of a Token
    Public Enum TokenType As UInt16 ...

    Represents the original location of a Token in the Source
    Public Structure TokenLocation ...

    Public ReadOnly Value As String
    Public ReadOnly Type As TokenType
    Public ReadOnly LocationInSource As TokenLocation

    Public Sub New(ByVal _Value$, ByVal _Type As TokenType, ByVal _LocationLine As UInt16, ByVal _LocationColumn As UInt16) ...

    Returns a KVPSerialised String to represent the Data of the current Object
    Public Overrides Function ToString() As String ...

End Class
```

This is the resultant automatically-generated Class Diagram:



Token
Class

Fields
- LocationInSource As TokenLocation
- Type As TokenType
- Value As String

Methods
- New(_Value As String, _Type As TokenType, _LocationLine As UShort, _LocationColumn As UShort)
- ToString() As String

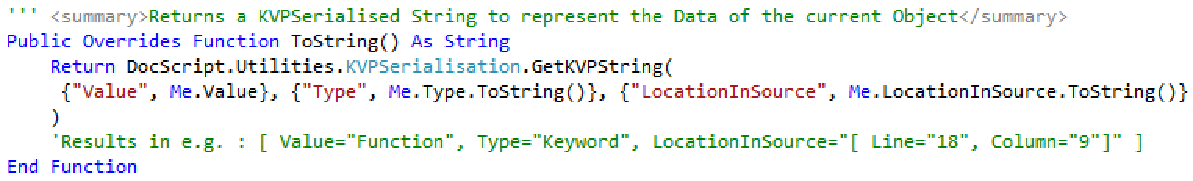
Nested Types

TokenLocation
Structure

TokenType
Enum

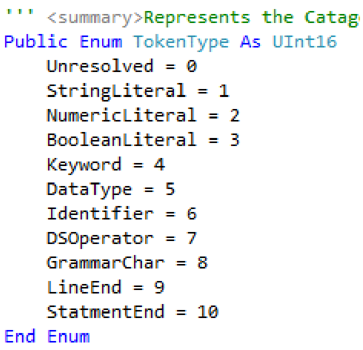
Despite being a simple, encapsulate, dictionary-style `Class`, several stylistically-salient components are worth herefrom pointing-out:

- I have written a custom **Key-Value-Pair Serialisation** method, to automate what I anticipate will become rather a commonplace method amongst Classes and Structures in the Solution. Here's how it's implemented in the `Token` Class:

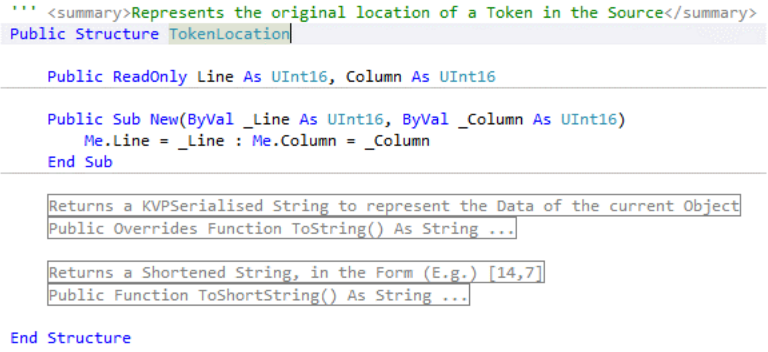


```
''' <summary>Returns a KVPSerialised String to represent the Data of the current Object</summary>
Public Overrides Function ToString() As String
    Return DocScript.Utilities.KVPSerialisation.GetKVPString(
     {"Value", Me.Value}, {"Type", Me.Type.ToString()}, {"LocationInSource", Me.LocationInSource.ToString()}
    )
    'Results in e.g. : [ Value="Function", Type="Keyword", LocationInSource="[ Line="18", Column="9"]" ]
End Function
```

- Type declarations directly relevant to the `Token` Class are defined as Nested Types within the Class itself. In this instance, the Nested Types are `Structure TokenLocation` and `Enum TokenType As UInt16`:

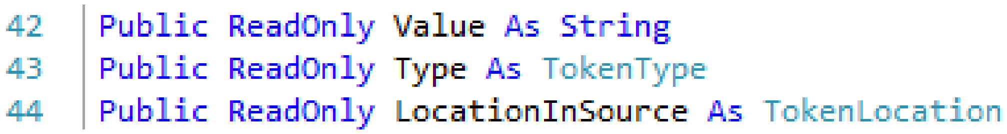


```
''' <summary>Represents the Catag
Public Enum TokenType As UInt16
    Unresolved = 0
    StringLiteral = 1
    NumericLiteral = 2
    BooleanLiteral = 3
    Keyword = 4
    DataType = 5
    Identifier = 6
    DSOperator = 7
    GrammarChar = 8
    LineEnd = 9
    StatmentEnd = 10
End Enum
```



```
''' <summary>Represents the original location of a Token in the Source</summary>
Public Structure TokenLocation

    Public ReadOnly Line As UInt16, Column As UInt16

    Public Sub New(ByVal _Line As UInt16, ByVal _Column As UInt16)
        Me.Line = _Line : Me.Column = _Column
    End Sub

    Returns a KVPSerialised String to represent the Data of the current Object
    Public Overrides Function ToString() As String ...

    Returns a Shortened String, in the Form (E.g.) [14,7]
    Public Function ToShortString() As String ...
End Structure
```

- The Class Members are immutable; once they have been assigned a value by the constructor, their value cannot be thereafter altered. This is implemented by means of language-level immutability through the `ReadOnly` keyword:

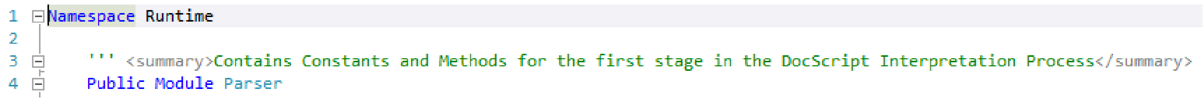


```
Public ReadOnly Value As String
Public ReadOnly Type As TokenType
Public ReadOnly LocationInSource As TokenLocation
```

This is good practice because it prevents unforeseen post-construction tampering: A `Token` is only supposed to represent three data about an occurrence of a character-sequence within DocScript source – since the character-sequence it represents will never change, it does not make sense for the members of this corresponding class to change either (wherefore they have been declared as `ReadOnly`).

The Parser Module

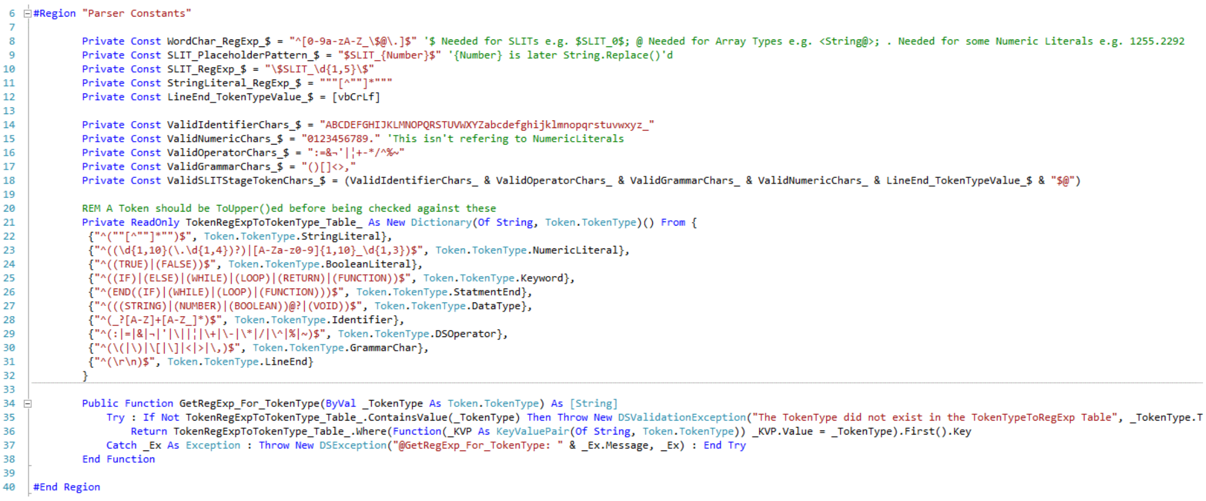
There is no need to make the Parser object-orientated (i.e having to do `(New Parser()).Parse(_Source)` instead of just `Parser.Parse(_Source)`), which is why I am implementing it simply as a Module:



```
Namespace Runtime

    ''' <summary>Contains Constants and Methods for the first stage in the DocScript Interpretation Process</summary>
    Public Module Parser
```

Constants

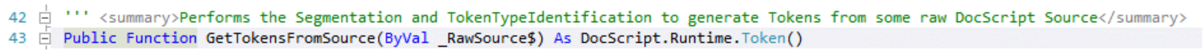
It contains a number of constants needed for the Parsing process…



```
#Region "Parser Constants"

        Private Const WordChar_RegExp_$ = "^[0-9a-zA-Z_\$@\.]$" '$ Needed for SLITs e.g. $SLIT_0$; @ Needed for Array Types e.g. <String@>; . Needed for some Numeric Literals e.g. 1255.2292
        Private Const SLIT_PlaceholderPattern_$ = "$SLIT_{Number}$" '{Number} is later String.Replace()'d
        Private Const SLIT_RegExp_$ = "\$SLIT_\d{1,5}\$"
        Private Const StringLiteral_RegExp_$ = """[^""]*"""
        Private Const LineEnd_TokenTypeValue_$ = [vbCrLf]

        Private Const ValidIdentifierChars_$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_"
        Private Const ValidNumericChars_$ = "0123456789." 'This isn't refering to NumericLiterals
        Private Const ValidOperatorChars_$ = ":=&¬'|¦+-*/^%~"
        Private Const ValidGrammarChars_$ = "()[]<>,"
        Private Const ValidSLITStageTokenChars_$ = (ValidIdentifierChars_ & ValidOperatorChars_ & ValidGrammarChars_ & ValidNumericChars_ & LineEnd_TokenTypeValue_$ & "$@")

        REM A Token should be ToUpper()ed before being checked against these
        Private ReadOnly TokenRegExpToTokenType_Table_ As New Dictionary(Of String, Token.TokenType)() From {
         {"^(""[^""]*"")$", Token.TokenType.StringLiteral},
         {"^((\d{1,10}(\.\d{1,4})?)|[A-Za-z0-9]{1,10}_\d{1,3})$", Token.TokenType.NumericLiteral},
         {"^((TRUE)|(FALSE))$", Token.TokenType.BooleanLiteral},
         {"^((IF)|(ELSE)|(WHILE)|(LOOP)|(RETURN)|(FUNCTION))$", Token.TokenType.Keyword},
         {"^(END((IF)|(WHILE)|(LOOP)|(FUNCTION)))$", Token.TokenType.StatmentEnd},
         {"^(((STRING)|(NUMBER)|(BOOLEAN))@?|(VOID))$", Token.TokenType.DataType},
         {"^(_?[A-Z]+[A-Z_]*)$", Token.TokenType.Identifier},
         {"^(:|=|&|¬|'|\||¦|\+|\-|\*|/|\^|%|~)$", Token.TokenType.DSOperator},
         {"^(\(|\)|\[|\]|<|>|\,)$", Token.TokenType.GrammarChar},
         {"^(\r\n)$", Token.TokenType.LineEnd}
        }

        Public Function GetRegExp_For_TokenType(ByVal _TokenType As Token.TokenType) As [String]
            Try : If Not TokenRegExpToTokenType_Table_.ContainsValue(_TokenType) Then Throw New DSValidationException("The TokenType did not exist in the TokenTypeToRegExp Table", _TokenType.T
                Return TokenRegExpToTokenType_Table_.Where(Function(_KVP As KeyValuePair(Of String, Token.TokenType)) _KVP.Value = _TokenType).First().Key
            Catch _Ex As Exception : Throw New DSException("@GetRegExp_For_TokenType: " & _Ex.Message, _Ex) : End Try
        End Function

#End Region
```

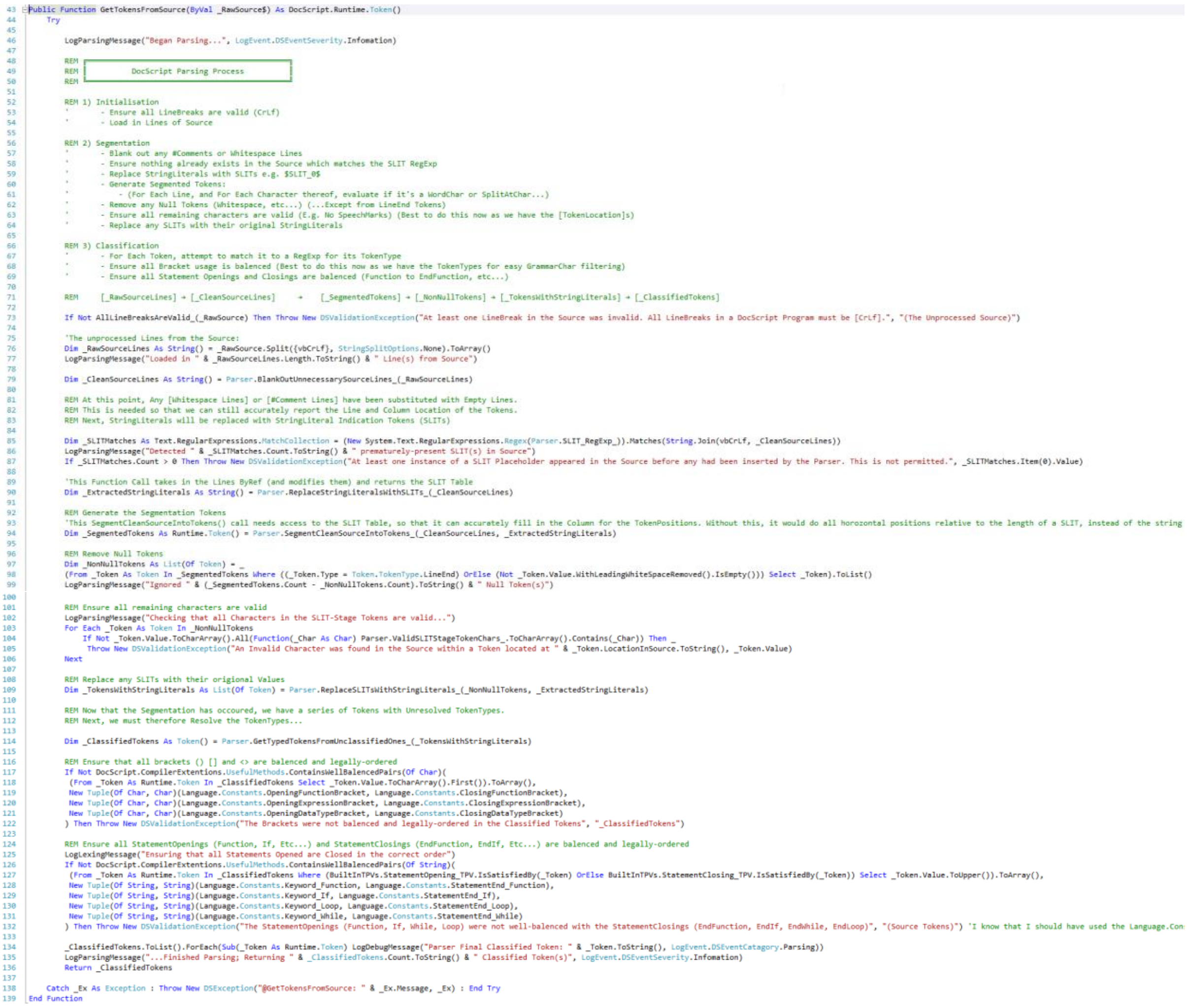
GetTokensFromSource()

…And the key method is:



```
''' <summary>Performs the Segmentation and TokenTypeIdentification to generate Tokens from some raw DocScript Source</summary>
Public Function GetTokensFromSource(ByVal _RawSource$) As DocScript.Runtime.Token()
```

Owing to my detailed planning and foresight during §Design, I was able to simply **follow my Pseudocode-English plan** for this method. It is implemented as follows:



```
Public Function GetTokensFromSource(ByVal _RawSource$) As DocScript.Runtime.Token()
    Try

        LogParsingMessage("Began Parsing...", LogEvent.DSEventSeverity.Infomation)

        REM
        REM             DocScript Parsing Process
        REM

        REM 1) Initialisation
        '       - Ensure all LineBreaks are valid (CrLf)
        '       - Load in Lines of Source

        REM 2) Segmentation
        '       - Blank out any #Comments or Whitespace Lines
        '       - Ensure nothing already exists in the Source which matches the SLIT RegExp
        '       - Replace StringLiterals with SLITs e.g. $SLIT_0$
        '       - Generate Segmented Tokens:
        '           - (For Each Line, and For Each Character thereof, evaluate if it's a WordChar or SplitAtChar...)
        '       - Remove any Null Tokens (Whitespace, etc...) (...Except from LineEnd Tokens)
        '       - Ensure all remaining characters are valid (E.g. No SpeechMarks) (Best to do this now as we have the [TokenLocation]s)
        '       - Replace any SLITs with their original StringLiterals

        REM 3) Classification
        '       - For Each Token, attempt to match it to a RegExp for its TokenType
        '       - Ensure all Bracket usage is balenced (Best to do this now as we have the TokenTypes for easy GrammarChar filtering)
        '       - Ensure all Statement Openings and Closings are balenced (Function to EndFunction, etc...)

        REM     [_RawSourceLines] → [_CleanSourceLines]     →     [_SegmentedTokens] → [_NonNullTokens] → [_TokensWithStringLiterals] → [_ClassifiedTokens]

        If Not AllLineBreaksAreValid_(_RawSource) Then Throw New DSValidationException("At least one LineBreak in the Source was invalid. All LineBreaks in a DocScript Program must be [CrLf].", "(The Unprocessed Source)")

        'The unprocessed Lines from the Source:
        Dim _RawSourceLines As String() = _RawSource.Split({vbCrLf}, StringSplitOptions.None).ToArray()
        LogParsingMessage("Loaded in " & _RawSourceLines.Length.ToString() & " Line(s) from Source")

        Dim _CleanSourceLines As String() = Parser.BlankOutUnnecessarySourceLines_(_RawSourceLines)

        REM At this point, Any [Whitespace Lines] or [#Comment Lines] have been substituted with Empty Lines.
        REM This is needed so that we can still accurately report the Line and Column Location of the Tokens.
        REM Next, StringLiterals will be replaced with StringLiteral Indication Tokens (SLITs)

        Dim _SLITMatches As Text.RegularExpressions.MatchCollection = (New System.Text.RegularExpressions.Regex(Parser.SLIT_RegExp_)).Matches(String.Join(vbCrLf, _CleanSourceLines))
        LogParsingMessage("Detected " & _SLITMatches.Count.ToString() & " prematurely-present SLIT(s) in Source")
        If _SLITMatches.Count > 0 Then Throw New DSValidationException("At least one instance of a SLIT Placeholder appeared in the Source before any had been inserted by the Parser. This is not permitted.", _SLITMatches.Item(0).Value)

        'This Function Call takes in the Lines ByRef (and modifies them) and returns the SLIT Table
        Dim _ExtractedStringLiterals As String() = Parser.ReplaceStringLiteralsWithSLITs_(_CleanSourceLines)

        REM Generate the Segmentation Tokens
        'This SegmentCleanSourceIntoTokens() call needs access to the SLIT Table, so that it can accurately fill in the Column for the TokenPositions. Without this, it would do all horozontal positions relative to the length of a SLIT, instead of the string
        Dim _SegmentedTokens As Runtime.Token() = Parser.SegmentCleanSourceIntoTokens_(_CleanSourceLines, _ExtractedStringLiterals)

        REM Remove Null Tokens
        Dim _NonNullTokens As List(Of Token) = _
        (From _Token As Token In _SegmentedTokens Where ((_Token.Type = Token.TokenType.LineEnd) OrElse (Not _Token.Value.WithLeadingWhiteSpaceRemoved().IsEmpty())) Select _Token).ToList()
        LogParsingMessage("Ignored " & (_SegmentedTokens.Count - _NonNullTokens.Count).ToString() & " Null Token(s)")

        REM Ensure all remaining characters are valid
        LogParsingMessage("Checking that all Characters in the SLIT-Stage Tokens are valid...")
        For Each _Token As Token In _NonNullTokens
            If Not _Token.Value.ToCharArray().All(Function(_Char As Char) Parser.ValidSLITStageTokenChars_.ToCharArray().Contains(_Char)) Then _
             Throw New DSValidationException("An Invalid Character was found in the Source within a Token located at " & _Token.LocationInSource.ToString(), _Token.Value)
        Next

        REM Replace any SLITs with their origional Values
        Dim _TokensWithStringLiterals As List(Of Token) = Parser.ReplaceSLITsWithStringLiterals_(_NonNullTokens, _ExtractedStringLiterals)

        REM Now that the Segmentation has occoured, we have a series of Tokens with Unresolved TokenTypes.
        REM Next, we must therefore Resolve the TokenTypes...

        Dim _ClassifiedTokens As Token() = Parser.GetTypedTokensFromUnclassifiedOnes_(_TokensWithStringLiterals)

        REM Ensure that all brackets () [] and <> are balenced and legally-ordered
        If Not DocScript.CompilerExtentions.UsefulMethods.ContainsWellBalencedPairs(Of Char)(
         (From _Token As Runtime.Token In _ClassifiedTokens Select _Token.Value.ToCharArray().First()).ToArray(),
         New Tuple(Of Char, Char)(Language.Constants.OpeningFunctionBracket, Language.Constants.ClosingFunctionBracket),
         New Tuple(Of Char, Char)(Language.Constants.OpeningExpressionBracket, Language.Constants.ClosingExpressionBracket),
         New Tuple(Of Char, Char)(Language.Constants.OpeningDataTypeBracket, Language.Constants.ClosingDataTypeBracket)
        ) Then Throw New DSValidationException("The Brackets were not balenced and legally-ordered in the Classified Tokens", "_ClassifiedTokens")

        REM Ensure all StatementOpenings (Function, If, Etc...) and StatementClosings (EndFunction, EndIf, Etc...) are balenced and legally-ordered
        LogLexingMessage("Ensuring that all Statements Opened are Closed in the correct order")
        If Not DocScript.CompilerExtentions.UsefulMethods.ContainsWellBalencedPairs(Of String)(
         (From _Token As Runtime.Token In _ClassifiedTokens Where (BuiltInTPVs.StatementOpening_TPV.IsSatisfiedBy(_Token) OrElse BuiltInTPVs.StatementClosing_TPV.IsSatisfiedBy(_Token)) Select _Token.Value.ToUpper()).ToArray(),
         New Tuple(Of String, String)(Language.Constants.Keyword_Function, Language.Constants.StatementEnd_Function),
         New Tuple(Of String, String)(Language.Constants.Keyword_If, Language.Constants.StatementEnd_If),
         New Tuple(Of String, String)(Language.Constants.Keyword_Loop, Language.Constants.StatementEnd_Loop),
         New Tuple(Of String, String)(Language.Constants.Keyword_While, Language.Constants.StatementEnd_While)
        ) Then Throw New DSValidationException("The StatementOpenings (Function, If, While, Loop) were not well-balenced with the StatementClosings (EndFunction, EndIf, EndWhile, EndLoop)", "(Source Tokens)") 'I know that I should have used the Language.Con

        _ClassifiedTokens.ToList().ForEach(Sub(_Token As Runtime.Token) LogDebugMessage("Parser Final Classified Token: " & _Token.ToString(), LogEvent.DSEventCatagory.Parsing))
        LogParsingMessage("...Finished Parsing; Returning " & _ClassifiedTokens.Count.ToString() & " Classified Token(s)", LogEvent.DSEventSeverity.Infomation)
        Return _ClassifiedTokens

    Catch _Ex As Exception : Throw New DSException("@GetTokensFromSource: " & _Ex.Message, _Ex) : End Try
End Function
```

Modularity and Structure Considerations

To segment this method and make it more manageable and easier to read, I have divided the *meat* of the algorithm into these `Private Function`s within the Parser Module:

```
AllLineBreaksAreValid_()
BlankOutUnnecessarySourceLines_()
ReplaceStringLiteralsWithSLITs_()
SegmentCleanSourceIntoTokens_()
ReplaceSLITsWithStringLiterals_()
GetTypedTokensFromUnclassifiedOnes_()
```

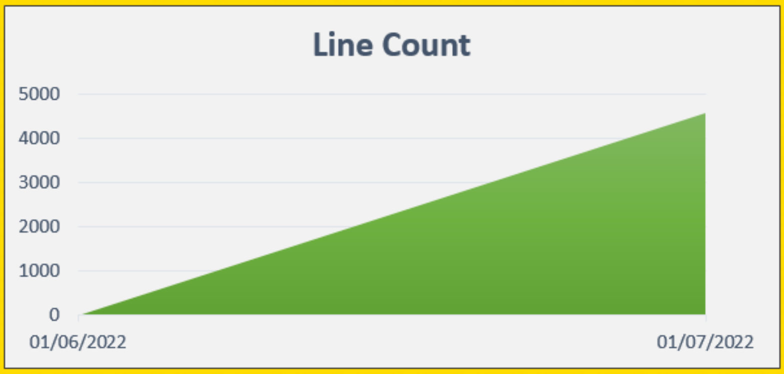
Validation

Naturally – in keeping with the nature of a *parser* – about half of the logic of the parser is just validation! Here are some of the checks performed:

- Are all Line-Breaks of the CrLf type?
- Do any SLIT-Placeholders appear in the source?
- Are there any Speech-marks remaining, after the SLIT-Stage Tokens are produced?
- Are there equal numbers of opening- and closing-brackets? `( ) [ ] < >`
- Are there equal numbers of Statement-Openings and -Closings? `If EndIf …`

Progress Check

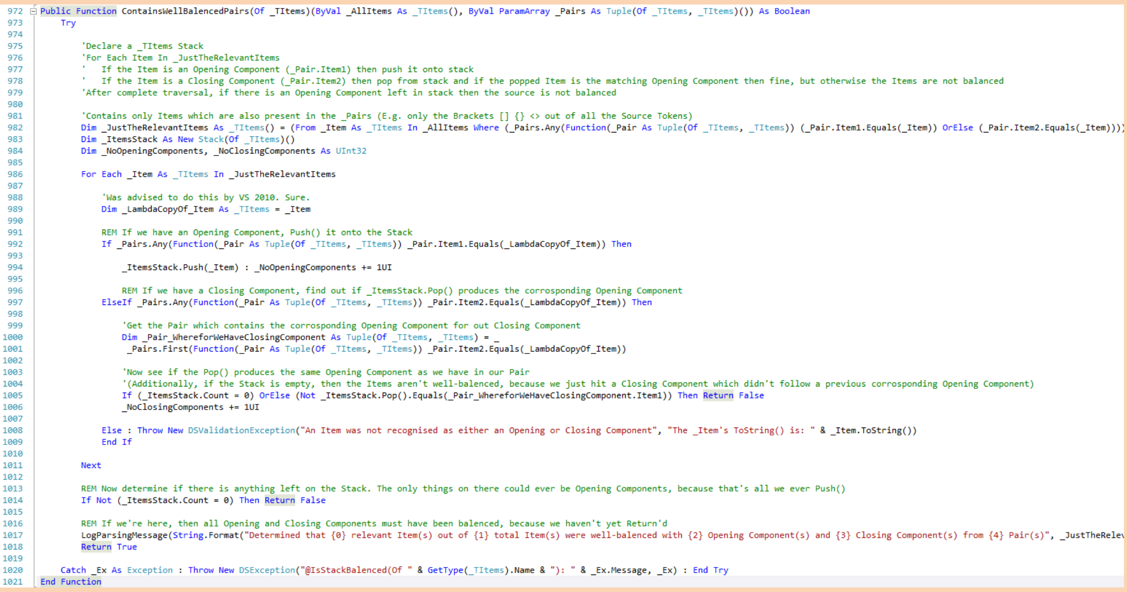
At this point in the development, I have written 4582 Lines of code...



A considerable proportion of those lines are involved in validation, such as…

Validation: The following algorithm is what I came up with to ensure that there are an equal number of opening and closing brackets, and that they're in the correct order…

- This is my implementation of the bracket-stack algorithm, which ensures that for each opening bracket, there is a corresponding closing one.



```
Public Function ContainsWellBalencedPairs(Of _TItems)(ByVal _AllItems As _TItems(), ByVal ParamArray _Pairs As Tuple(Of _TItems, _TItems)()) As Boolean
    Try

        'Declare a _TItems Stack
        'For Each Item In _JustTheRelevantItems
        '   If the Item is an Opening Component (_Pair.Item1) then push it onto stack
        '   If the Item is a Closing Component (_Pair.Item2) then pop from stack and if the popped Item is the matching Opening Component then fine, but otherwise the Items are not balanced
        'After complete traversal, if there is an Opening Component left in stack then the source is not balanced

        'Contains only Items which are also present in the _Pairs (E.g. only the Brackets [] {} <> out of all the Source Tokens)
        Dim _JustTheRelevantItems As _TItems() = (From _Item As _TItems In _AllItems Where (_Pairs.Any(Function(_Pair As Tuple(Of _TItems, _TItems)) (_Pair.Item1.Equals(_Item)) OrElse (_Pair.Item2.Equals(_Item)))
        Dim _ItemsStack As New Stack(Of _TItems)()
        Dim _NoOpeningComponents, _NoClosingComponents As UInt32

        For Each _Item As _TItems In _JustTheRelevantItems

            'Was advised to do this by VS 2010. Sure.
            Dim _LambdaCopyOf_Item As _TItems = _Item

            REM If we have an Opening Component, Push() it onto the Stack
            If _Pairs.Any(Function(_Pair As Tuple(Of _TItems, _TItems)) _Pair.Item1.Equals(_LambdaCopyOf_Item)) Then

                _ItemsStack.Push(_Item) : _NoOpeningComponents += 1UI

                REM If we have a Closing Component, find out if _ItemsStack.Pop() produces the corrosponding Opening Component
            ElseIf _Pairs.Any(Function(_Pair As Tuple(Of _TItems, _TItems)) _Pair.Item2.Equals(_LambdaCopyOf_Item)) Then

                'Get the Pair which contains the corrosponding Opening Component for out Closing Component
                Dim _Pair_WhereforWeHaveClosingComponent As Tuple(Of _TItems, _TItems) = _
                 _Pairs.First(Function(_Pair As Tuple(Of _TItems, _TItems)) _Pair.Item2.Equals(_LambdaCopyOf_Item))

                'Now see if the Pop() produces the same Opening Component as we have in our Pair
                '(Additionally, if the Stack is empty, then the Items aren't well-balenced, because we just hit a Closing Component which didn't follow a previous corrosponding Opening Component)
                If (_ItemsStack.Count = 0) OrElse (Not _ItemsStack.Pop().Equals(_Pair_WhereforWeHaveClosingComponent.Item1)) Then Return False
                _NoClosingComponents += 1UI

            Else : Throw New DSValidationException("An Item was not recognised as either an Opening or Closing Component", "The _Item's ToString() is: " & _Item.ToString())
            End If

        Next

        REM Now determine if there is anything left on the Stack. The only things on there could ever be Opening Components, because that's all we ever Push()
        If Not (_ItemsStack.Count = 0) Then Return False

        REM If we're here, then all Opening and Closing Components must have been balenced, because we haven't yet Return'd
        LogParsingMessage(String.Format("Determined that {0} relevant Item(s) out of {1} total Item(s) were well-balenced with {2} Opening Component(s) and {3} Closing Component(s) from {4} Pair(s)", _JustTheRelev
        Return True

    Catch _Ex As Exception : Throw New DSException("@IsStackBalenced(Of " & GetType(_TItems).Name & "): " & _Ex.Message, _Ex) : End Try
End Function
```

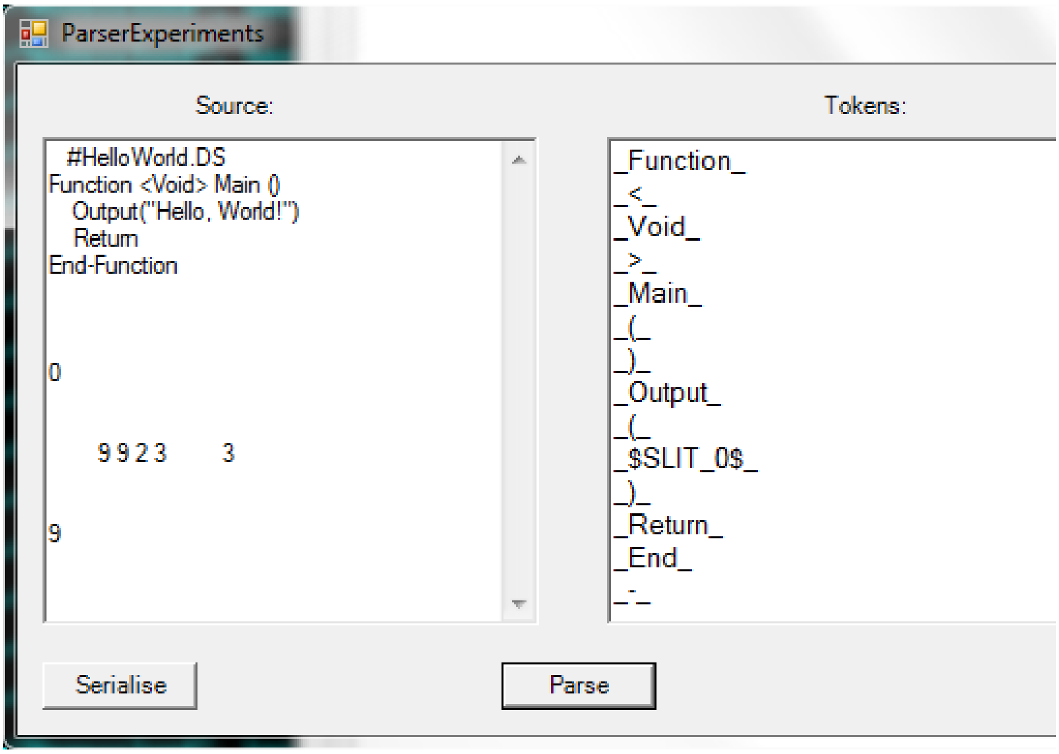
- For example: ([]) would be valid, whereas ([)] would not be (even though there are the same number of brackets and squares in the latter).
- This ensures than an expression which is malformed because of bracket misplacement or mismatch, ***does not even make it past the parser***. This means that by the time the lexer gets hold of the tokens, it can be much more confident that the expression is well-formed.

Iterative Testing

It's time for the first test of the parser!

Creating the Scratch-Testing Project

I started a new Project in the DocScript Solution for experimentation and debugging purposes. Then I created a simple Windows-Form to take in some source, and show the segmented Tokens resulting from that source:



```
#HelloWorld.DS
Function <Void> Main ()
   Output("Hello, World!")
   Return
End-Function



0


     9923    3


9
```

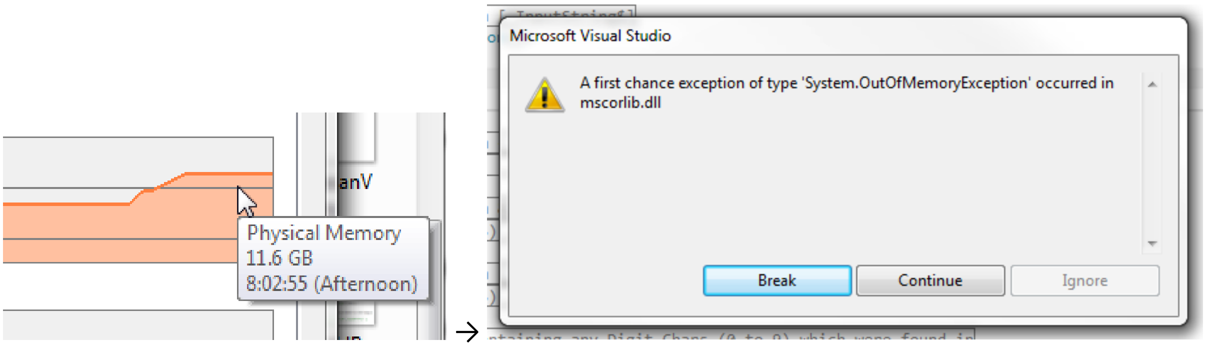
```
_Function_
_<_
_Void_
_>_
_Main_
_(_
_)_
_Output_
_(_
_$SLIT_0$_
_)_
_Return_
_End_
_-_
```

Debugging

On clicking the [Parse] Button – however – I was surprised to see the application **freeze up and nothing happen** for a few seconds, and then, suddenly, the debugging session would Throw a `System.OutOfMemoryException` and crash entirely. This (is must be said) is certainly one of the most exciting exceptions to come across. I knew that the likely causes were:

- *A stack overflow caused by an infinite loop*

To examine what was going on more closely, I reproduced the circumstances under which the `OutOfMemoryException` had been thrown. I saw this befall in **Process Explorer's System-Information Graph**:

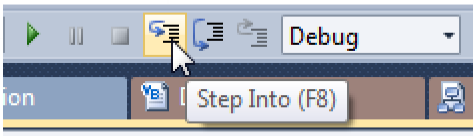


Physical Memory
11.6 GB
8:02:55 (Afternoon)

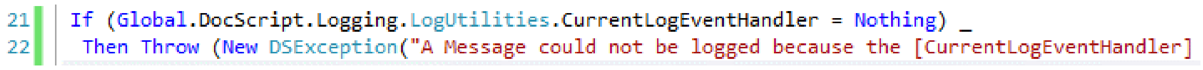
Microsoft Visual Studio

A first chance exception of type 'System.OutOfMemoryException' occurred in mscorlib.dll

I used the Step-Through feature of VS to trace what the loop might be:



I realised that ↓ *this* ↓ line was being hit…

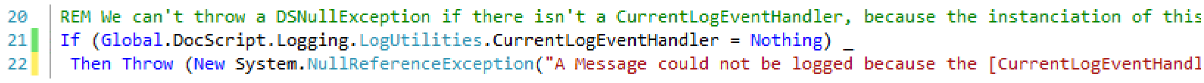


```
If (Global.DocScript.Logging.LogUtilities.CurrentLogEventHandler = Nothing) _
 Then Throw (New DSException("A Message could not be logged because the [CurrentLogEventHandler]
```

…Which caused the constructor for `DSException` *to be called, which attempted to Log the Exception, which couldn't occur because the* `CurrentLogEventHandler` *was* `Nothing`*, so it constructs a* `New` `DSException` *to explain this, which attempts to Log the Exception, which couldn't occur because the* `CurrentLogEventHandler` *was Nothing, which then constructs a* `New DSException`*, which attempts to Log the Exception, which it can't because there's no* `CurrentLogEventHandler`*…*

…And so on… …Recursively… …Forever….
(Or, indeed, until my 16GB of system memory had been expended.)

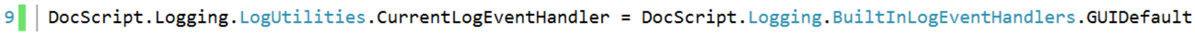
To fix this (admittedly rather-catastrophic) bug, I changed the `DSException` to a `System.NullReferenceException`, which **dosen't** forward the Message onto DocScript Logging, thereby breaking the chain!



```
REM We can't throw a DSNullException if there isn't a CurrentLogEventHandler, because the instanciation of this
If (Global.DocScript.Logging.LogUtilities.CurrentLogEventHandler = Nothing) _
 Then Throw (New System.NullReferenceException("A Message could not be logged because the [CurrentLogEventHandl
```

I also added a comment in, to indicate this change ↑.

However: There was one more amendment to make; the root cause of this problem had been the uninitialized `CurrentLogEventHandler`. I had forgotten to type this line at the start of the Experimentation Project's effective EntryPoint:

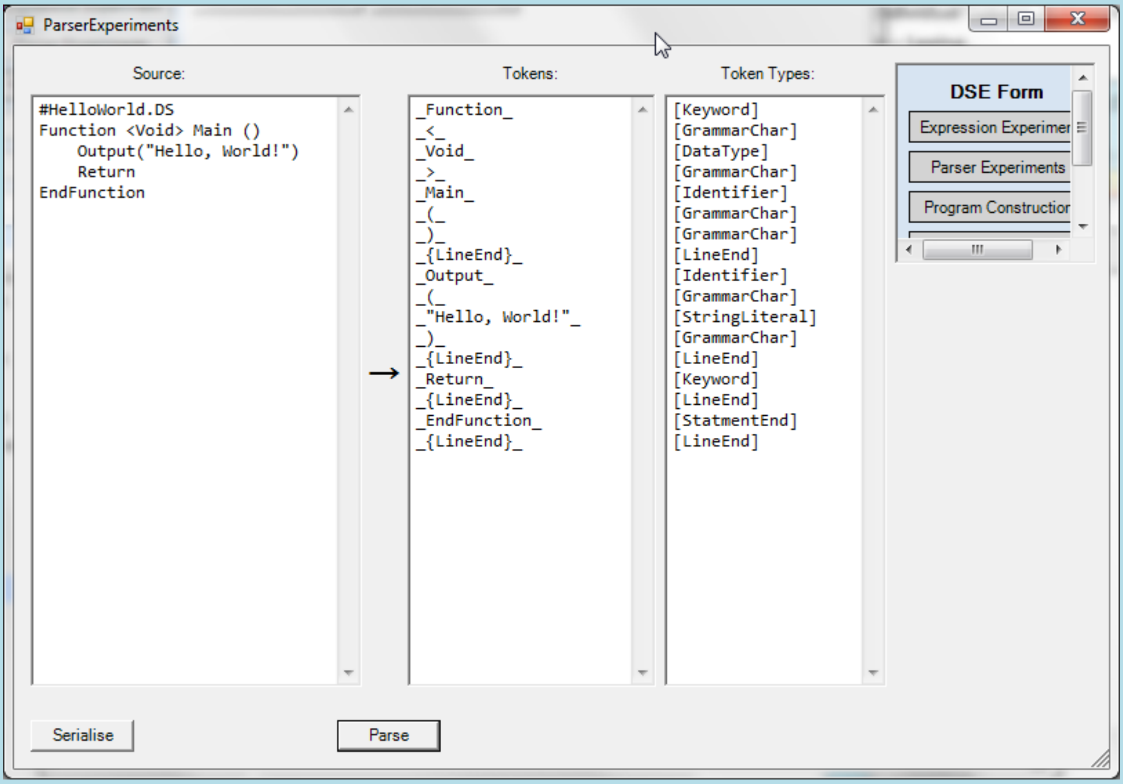


```
DocScript.Logging.LogUtilities.CurrentLogEventHandler = DocScript.Logging.BuiltInLogEventHandlers.GUIDefault
```

With this addition, the Parser Testing continued, and worked rather well. I tested every possible eventuality (using many of the data declared in §Design) and made sure to attempt to break the system. This is an example of **destructive testing**. I used both source which *would* be valid, as well as entirely-erroneous source, which certainly *wouldn't* be valid DocScript. Of course, the parser is somewhat *dumb*, in that it is oblivious to the validity of anything beyond the structure of the individual Tokens. It does not validate the *order* or *frequency* of the Tokens; that's for the next stage – Lexing.

Prototype: This point marks a milestone in the product, which is now capable of…

- Taking in some raw Source, and thence deriving DocScript Tokens. This is how it appears in the `DocScript.Experimentation` project:



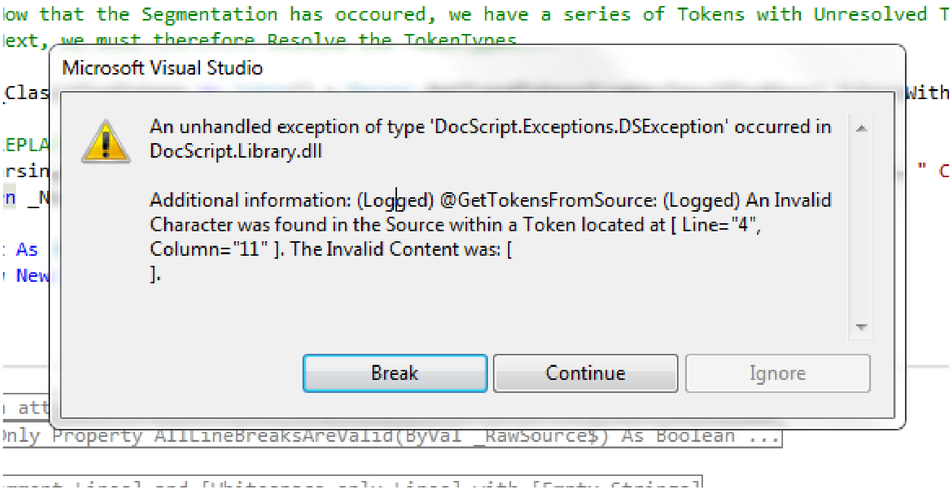
```
#HelloWorld.DS
Function <Void> Main ()
    Output("Hello, World!")
    Return
EndFunction
```

```
_Function_
_<_
_Void_
_>_
_Main_
_(_
_)_
_{LineEnd}_
_Output_
_(_
_"Hello, World!"_
_)_
_{LineEnd}_
_Return_
_{LineEnd}_
_EndFunction_
_{LineEnd}_
```

```
[Keyword]
[GrammarChar]
[DataType]
[GrammarChar]
[Identifier]
[GrammarChar]
[GrammarChar]
[LineEnd]
[Identifier]
[GrammarChar]
[StringLiteral]
[GrammarChar]
[LineEnd]
[Keyword]
[LineEnd]
[StatmentEnd]
[LineEnd]
```

- Performing preliminary syntactical validation on the source, including ensuring that the number of opening- and closing-brackets is the same.
- Ensuring that there is a {LineEnd} appending the source, readying it for the imminent *lexing* stage.

My custom Exceptions are also making debugging significantly more targeted and direct. I can see, for example, precisely which method this Exception is coming from, thanks to the integrated StackTrace:

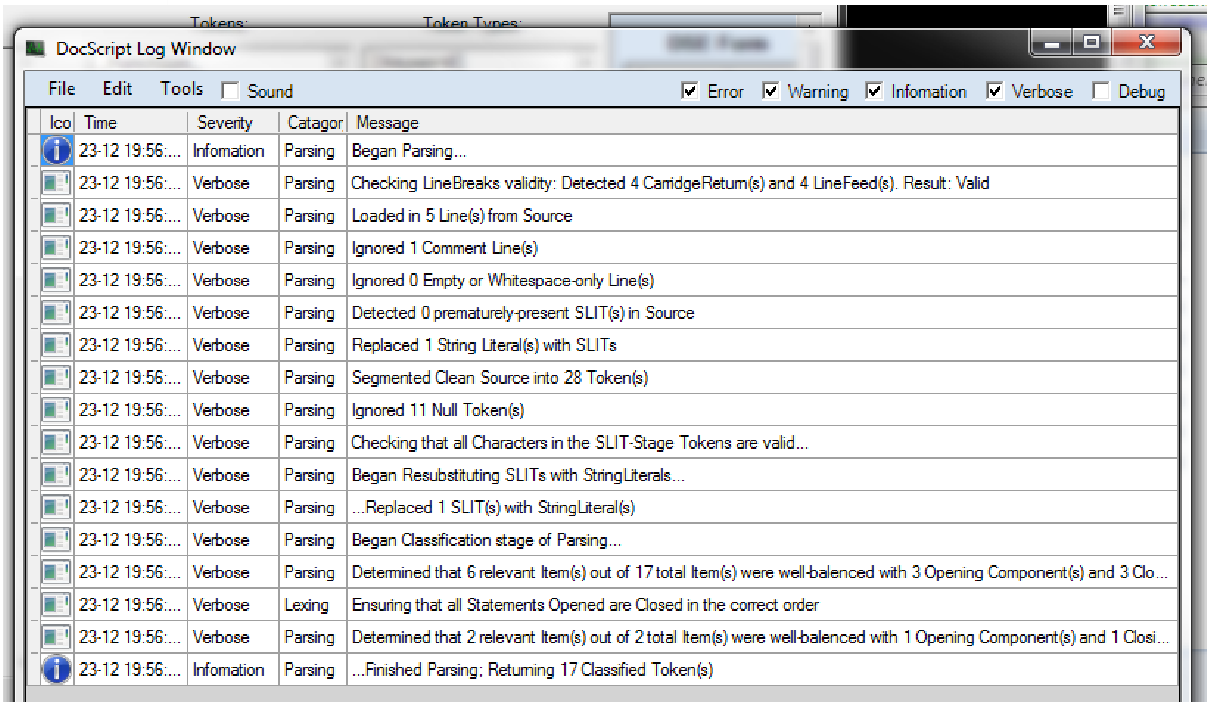

```
low that the Segmentation has occoured, we have a series of Tokens with Unresolved T
lext, we must therefore Resolve the TokenTypes
Clas                                                                      With
EPLA
rsin                                                                      " C
n _N
: As
 New
 att
nly Property AllLineBreaksAreValid(ByVal _RawSource$) As Boolean ...
```

Microsoft Visual Studio

An unhandled exception of type 'DocScript.Exceptions.DSException' occurred in DocScript.Library.dll

Additional information: (Logged) @GetTokensFromSource: (Logged) An Invalid Character was found in the Source within a Token located at [Line="4", Column="11"]. The Invalid Content was: [
].

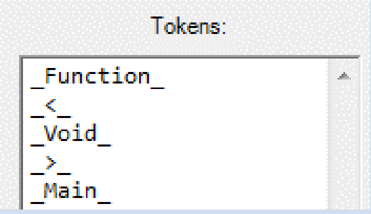
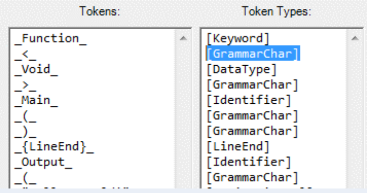
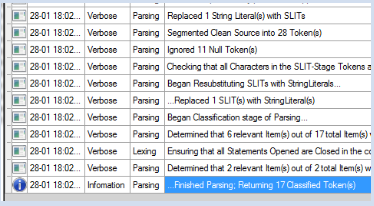
The Logging with the `GUIDefault As LogEventHandler` now works well too:

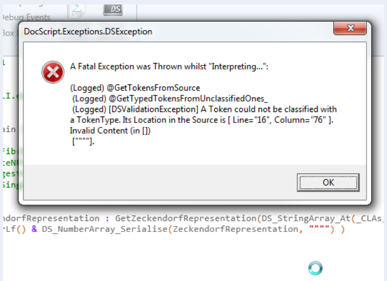


```
Ico  Time           Severity    Catagor  Message
     23-12 19:56:...  Infomation  Parsing  Began Parsing...
     23-12 19:56:...  Verbose     Parsing  Checking LineBreaks validity: Detected 4 CarridgeReturn(s) and 4 LineFeed(s). Result: Valid
     23-12 19:56:...  Verbose     Parsing  Loaded in 5 Line(s) from Source
     23-12 19:56:...  Verbose     Parsing  Ignored 1 Comment Line(s)
     23-12 19:56:...  Verbose     Parsing  Ignored 0 Empty or Whitespace-only Line(s)
     23-12 19:56:...  Verbose     Parsing  Detected 0 prematurely-present SLIT(s) in Source
     23-12 19:56:...  Verbose     Parsing  Replaced 1 String Literal(s) with SLITs
     23-12 19:56:...  Verbose     Parsing  Segmented Clean Source into 28 Token(s)
     23-12 19:56:...  Verbose     Parsing  Ignored 11 Null Token(s)
     23-12 19:56:...  Verbose     Parsing  Checking that all Characters in the SLIT-Stage Tokens are valid...
     23-12 19:56:...  Verbose     Parsing  Began Resubstituting SLITs with StringLiterals...
     23-12 19:56:...  Verbose     Parsing  ...Replaced 1 SLIT(s) with StringLiteral(s)
     23-12 19:56:...  Verbose     Parsing  Began Classification stage of Parsing...
     23-12 19:56:...  Verbose     Parsing  Determined that 6 relevant Item(s) out of 17 total Item(s) were well-balenced with 3 Opening Component(s) and 3 Clo...
     23-12 19:56:...  Verbose     Lexing   Ensuring that all Statements Opened are Closed in the correct order
     23-12 19:56:...  Verbose     Parsing  Determined that 2 relevant Item(s) out of 2 total Item(s) were well-balenced with 1 Opening Component(s) and 1 Closi...
     23-12 19:56:...  Infomation  Parsing  ...Finished Parsing; Returning 17 Classified Token(s)
```

Testing Table: Does this component function in accordance with the stipulated criteria?
I will now test the Prototype Parser against criteria from the §Design, and some new criteria.

Does the parser operate reliably, speedily, and consistently?

Test	☑ Passed?
Raw source is split at the `SplitAtChar`s, and remains concatenated at `WordChar`s.	**Yes**;  Tokens: _Function_ _<_ _Void_ _>_ _Main_
Tokens are assigned their correct TokenTypes. E.g. `<` must be identified as a `GrammarChar`.	**Yes**;  Tokens: | Token Types: _Function_ | [Keyword] _<_ | [GrammarChar] _Void_ | [DataType] _>_ | [GrammarChar] _Main_ | [Identifier] _(_ | [GrammarChar] _)_ | [GrammarChar] _{LineEnd}_ | [LineEnd] _Output_ | [Identifier] _(_ | [GrammarChar]
DocScript Logging reports the number of Tokens output by the Parser (`GetTokensFromSource()`).	**Yes**;  28-01 18:02... Verbose Parsing Replaced 1 String Literal(s) with SLITs 28-01 18:02... Verbose Parsing Segmented Clean Source into 28 Token(s) 28-01 18:02... Verbose Parsing Ignored 11 Null Token(s) 28-01 18:02... Verbose Parsing Checking that all Characters in the SLIT-Stage Tokens a 28-01 18:02... Verbose Parsing Began Resubstituting SLITs with StringLiterals... 28-01 18:02... Verbose Parsing ...Replaced 1 SLIT(s) with StringLiteral(s) 28-01 18:02... Verbose Parsing Began Classification stage of Parsing... 28-01 18:02... Verbose Parsing Determined that 6 relevant Item(s) out of 17 total Item(s) 28-01 18:02... Verbose Lexing Ensuring that all Statements Opened are Closed in the co 28-01 18:02... Verbose Parsing Determined that 2 relevant Item(s) out of 2 total Item(s) w 28-01 18:02... Infomation Parsing ...Finished Parsing; Returning 17 Classified Token(s)

The Parser detects if there is a mismatch between tokens which should appear in pairs; it must ensure that for each opening bracket, statement, or quote, there is a corresponding closing component. The validation herefor occurs in `ContainsWellBalencedPairs(Of _TItems)(ByVal _AllItems As _TItems(), ByVal ParamArray _Pairs As Tuple(Of _TItems, _TItems)()) As Boolean`. Exceptions herefrom are passed down the call stack, until the user sees some form of message, depending on the DocScript Implementation being used.	**Partially**: Unbalanced brackets `<> [] ()` and Statements `If While Loop Function` are caught by the mentioned function, whereas unbalanced string literals e.g. `"""` or `""""` are caught be the Token Classifier instead:  DocScript.Exceptions.DSException A Fatal Exception was Thrown whilst "Interpreting...": (Logged) @GetTokensFromSource (Logged) @GetTypedTokensFromUnclassifiedOnes_ (Logged) [DSValidationException] A Token could not be classified with a TokenType. Its Location in the Source is [Line="16", Column="76"]. Invalid Content (in []) ["""]. This still catches the Error, and prevents further stages of interpretation from getting confused. Action to take: Add a note about this pair-like component being caught at a later-than-might-be-assumed stage of the interpretation process, to the Solution's Documentation txt.

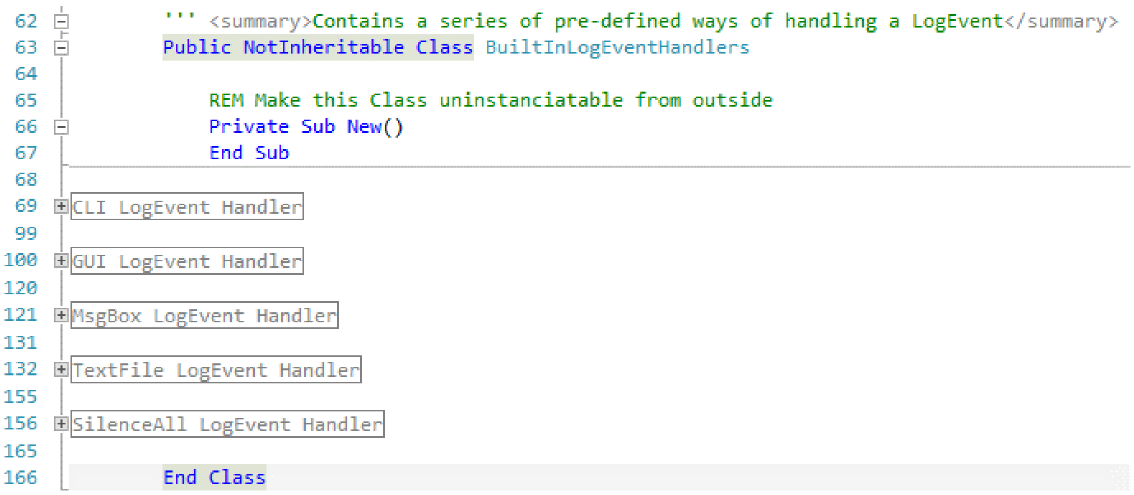
Justifications for Actions Taken: By documenting the oddity, it can be found by anyone who might look into why this behaviour occurs. It isn't a large enough oddity to warrant re-writing a component of the parser, because for the oddity to be noticed, erroneous source is required in the first-place. In other words, for this problem to occur, there already needs to be something quite erroneous about the input source code, so restructuring how this downstream problem is handled, would do nothing to solve the foundational issue.

Wait: Bootstrapping!

Writing the Parser has enabled me to notice a number of prerequisite utilities and components, which – as I continue development – an increasing number of modules and classes will rely on. Before I continue, therefore, I will take some time to properly implement each of the following…

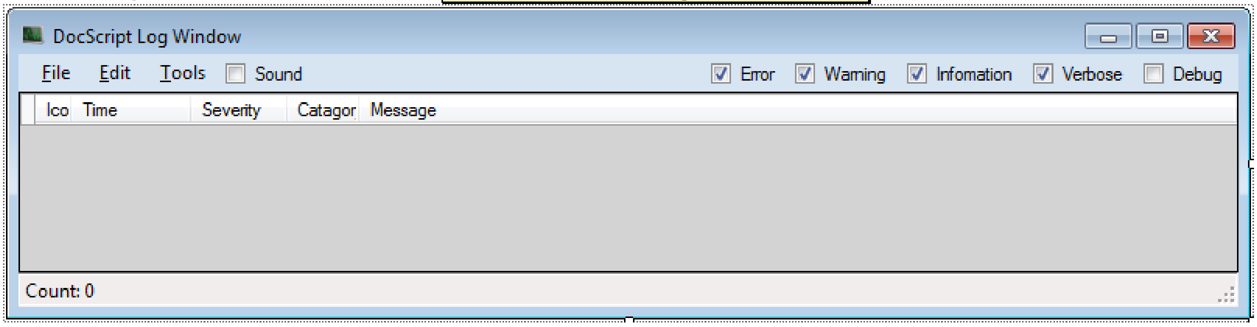
Logging

I established in §Design that I would have a `CurrentLogEventHandler` object; a delegate pointing to the Function to use for Logging. In addition to this declaration, however, I am also implementing four `BuiltInLogEventHandlers`, to make it easy to pump `LogEvent`s out to the surface for debugging:



```
        ''' <summary>Contains a series of pre-defined ways of handling a LogEvent</summary>
        Public NotInheritable Class BuiltInLogEventHandlers

            REM Make this Class uninstanciatable from outside
            Private Sub New()
            End Sub

CLI LogEvent Handler

GUI LogEvent Handler

MsgBox LogEvent Handler

TextFile LogEvent Handler

SilenceAll LogEvent Handler

        End Class
```

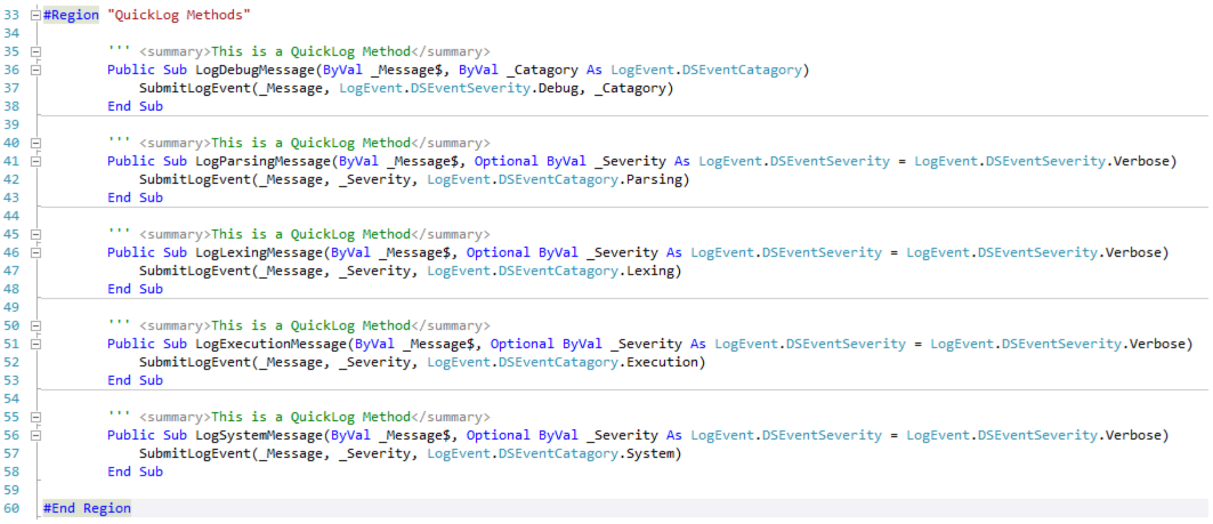
I then designed this window, for the GUIDefault As LogEventHandler:



The SubmitLogEvent() method requires all parameters for a LogEvent…

```
''' <summary>Invokes the CurrentLogEventHandler with the specified Data for the LogEvent. A
lso checks that the CurrentLogEventHandler has been initialised with a Delegate.</summary>
Public Function SubmitLogEvent(ByVal _Message$, ByVal _Severity As LogEvent.DSEventSeverity
, ByVal _Catagory As LogEvent.DSEventCatagory) As LogEventSubmissionResult
```

…and is somewhat inconvenient to call when one only wants to log a single string with default Severity and a predefined Category. Therefore, I am also defining some pre-defined *QuickLog* Methods:

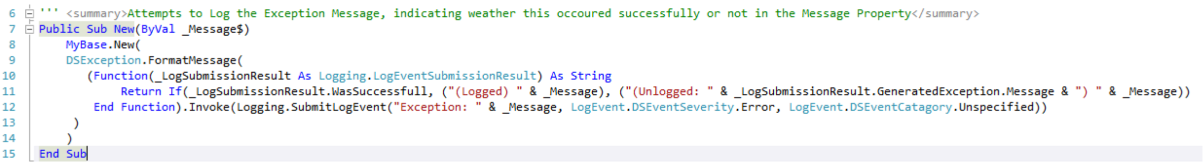


```
#Region "QuickLog Methods"

        ''' <summary>This is a QuickLog Method</summary>
        Public Sub LogDebugMessage(ByVal _Message$, ByVal _Catagory As LogEvent.DSEventCatagory)
            SubmitLogEvent(_Message, LogEvent.DSEventSeverity.Debug, _Catagory)
        End Sub

        ''' <summary>This is a QuickLog Method</summary>
        Public Sub LogParsingMessage(ByVal _Message$, Optional ByVal _Severity As LogEvent.DSEventSeverity = LogEvent.DSEventSeverity.Verbose)
            SubmitLogEvent(_Message, _Severity, LogEvent.DSEventCatagory.Parsing)
        End Sub

        ''' <summary>This is a QuickLog Method</summary>
        Public Sub LogLexingMessage(ByVal _Message$, Optional ByVal _Severity As LogEvent.DSEventSeverity = LogEvent.DSEventSeverity.Verbose)
            SubmitLogEvent(_Message, _Severity, LogEvent.DSEventCatagory.Lexing)
        End Sub

        ''' <summary>This is a QuickLog Method</summary>
        Public Sub LogExecutionMessage(ByVal _Message$, Optional ByVal _Severity As LogEvent.DSEventSeverity = LogEvent.DSEventSeverity.Verbose)
            SubmitLogEvent(_Message, _Severity, LogEvent.DSEventCatagory.Execution)
        End Sub

        ''' <summary>This is a QuickLog Method</summary>
        Public Sub LogSystemMessage(ByVal _Message$, Optional ByVal _Severity As LogEvent.DSEventSeverity = LogEvent.DSEventSeverity.Verbose)
            SubmitLogEvent(_Message, _Severity, LogEvent.DSEventCatagory.System)
        End Sub

#End Region
```

Custom Exceptions

In the Exceptions Namespace, I have defined the following Exception Types…

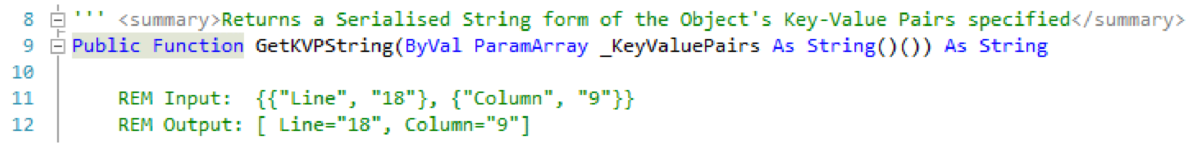
I will have to add to these as the Project Develops. The clever thing about them is that they will attempt to **Log** whatever the Exception Message is, through DocScript Logging, when Thrown. This means that I don't need to write something like `LogError("Some Error…") : Throw New DSException("Some Error…")` because the latter instantiation of the Exception will automatically perform the logging with a LogEvent of the Error Severity. Here's how the constructor calls DocScropt Logging:



```
''' <summary>Attempts to Log the Exception Message, indicating weather this occoured successfully or not in the Message Property</summary>
Public Sub New(ByVal _Message$)
    MyBase.New(
    DSException.FormatMessage(
       (Function(_LogSubmissionResult As Logging.LogEventSubmissionResult) As String
            Return If(_LogSubmissionResult.WasSuccessfull, ("(Logged) " & _Message), ("(Unlogged: " & _LogSubmissionResult.GeneratedException.Message & ") " & _Message))
        End Function).Invoke(Logging.SubmitLogEvent("Exception: " & _Message, LogEvent.DSEventSeverity.Error, LogEvent.DSEventCatagory.Unspecified))
      )
     )
End Sub
```

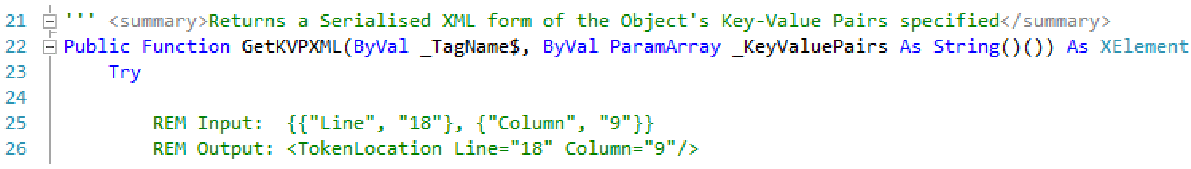
KVP-Serialisation

This Module in the `DocScript.Utilities` Namespace has been written to make it much easier to consistently write the `ToString()` methods on Classes and Structures, which serialise Key-Value Pairs within the object. For instance: `Name` could be a Key with the Value `"Ben"`. I have written one method to output a **String**…



```
''' <summary>Returns a Serialised String form of the Object's Key-Value Pairs specified</summary>
Public Function GetKVPString(ByVal ParamArray _KeyValuePairs As String()()) As String

    REM Input:  {{"Line", "18"}, {"Column", "9"}}
    REM Output: [ Line="18", Column="9"]
```

…And another to output **XML** via an XElement:



```
''' <summary>Returns a Serialised XML form of the Object's Key-Value Pairs specified</summary>
Public Function GetKVPXML(ByVal _TagName$, ByVal ParamArray _KeyValuePairs As String()()) As XElement
    Try

        REM Input:  {{"Line", "18"}, {"Column", "9"}}
        REM Output: <TokenLocation Line="18" Column="9"/>
```

Compiler-Extension Methods

There were a number of common tasks which I had to perform in the Parser's logic (such as [removing trailing whitespace from a string], [repeating an object N times], or [matching a string against a Regular-Expression]) which are made significantly more elegant and effortless when implemented as Compiler-Extension Methods, rather than simple procedural functions.

For instance: Instead of declaring…

```
Public Function MatchesRegEx(ByVal _StringToValidate$, ByVal _RegExPattern$) As Boolean
```

…And calling in the fashion…

```
 If MatchesRegEx(_Token.Value, Parser.Constants.StringLiteralRegExp) …
```

…I instead declare the method as an Extension, thusly:

```
<Global.System.Runtime.CompilerServices.Extension()>
Public Function MatchesRegEx(ByVal _StringToValidate$, ByVal _RegExPattern$) As Boolean
```

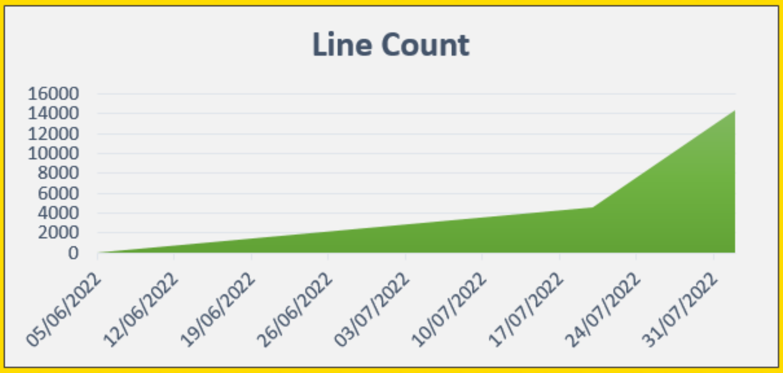
And simply call it as if it is a member of the String type:

```
If _Token.Value.MatchesRegEx(Parser.Constants.StringLiteralRegExp) …
```

This reduces intractable layers of brackets like `A(B(C(D)))`, and also gives a more object-orientated feel; `D.C().B().A()` is easier to read.

Progress Check

At this point in the development, I have written 14355 Lines of code…



Progress Recap: Where am I in the development plan? *[Review]*

- **Done**: I have written the Parser, and a number of bootstrapping prerequisites to aid in efficient and robust implementation of the forthcoming stages.
- **Next**: I will write the lexing system. This is the second of three stages of DocScript interpretation.

Lexing

As a reminder (*partly to myself*), DocScript Lexing works like this:

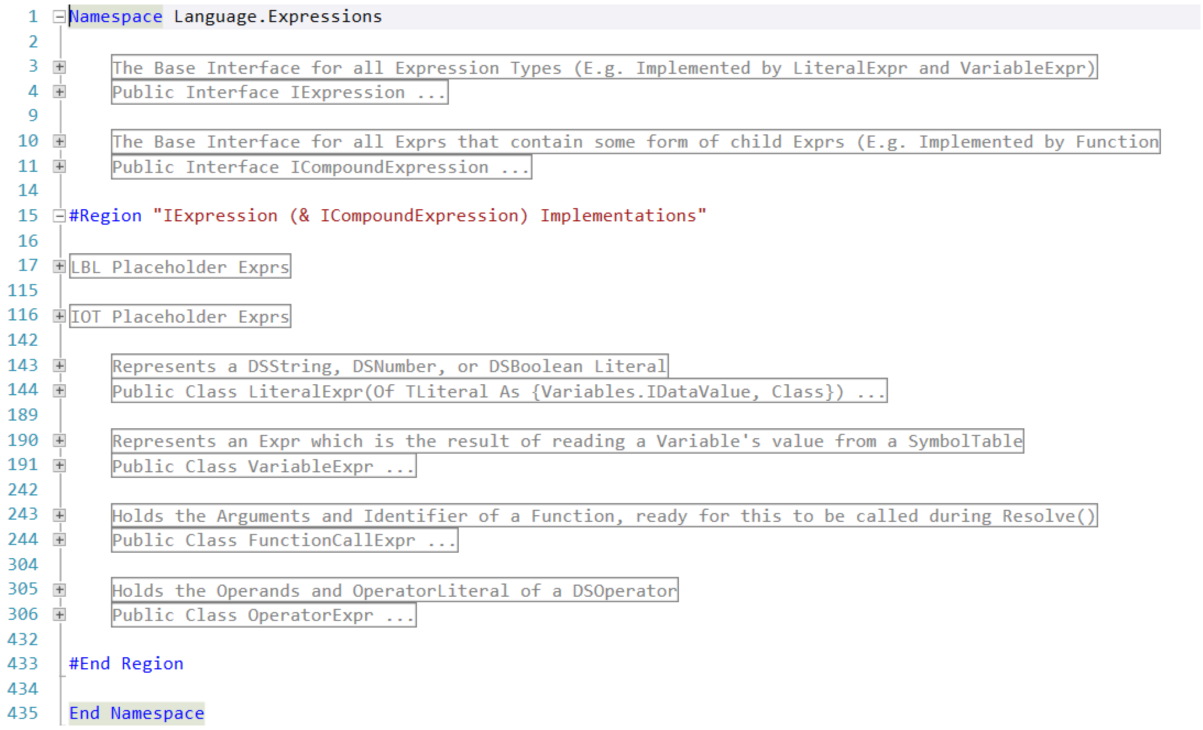
- The Tokens generated by the Parser are passed into the constructor to the `Program` class.
- Inside this constructor, the Global `VariableDeclaration`s and `Function`s are derived.
- A `New DSFunction` is constructed for each Function of the `Program`; the Tokens for the Function are passed to its constructor.
- Inside the `DSFunction` constructor, all the contents Instructions are derived from the remaining Tokens, in a loop.
- Each `IInstruction` has a constructor which takes in the `Token`s required to construct it.

Expressions

The four Expression Classes (`VariableExpr`, `LiteralExpr`, `OperatorExpr`, and `FunctionCallExpr`) are very similar to the eight Instruction Classes, except that they are not independently valid as lines within a Function. Expressions appear as components of the following Instruction Types: `ReturnToCaller`, `VariableDeclaration`, `VariableAssignment`, `FunctionCall`, `IfStatement`, `WhileStatement`, and `LoopStatement` (everything except from `DSFunction`).

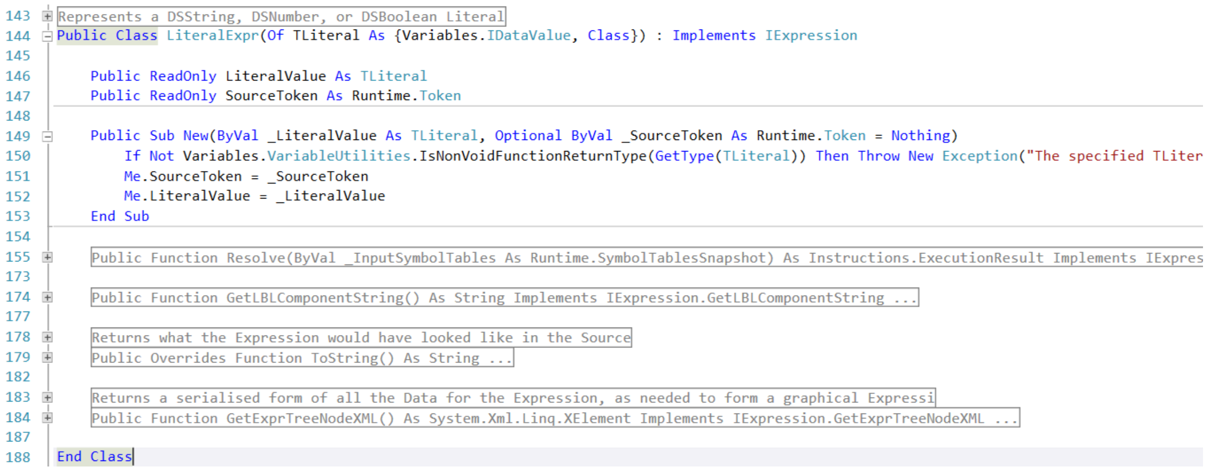
Expression Classes

I have laid-out the DS.L.Expressions.VB File like this:



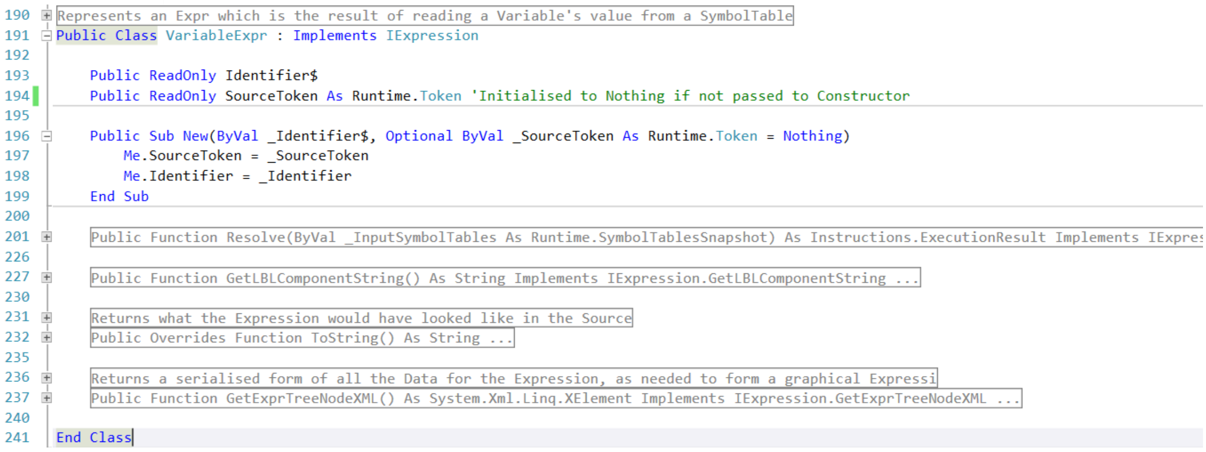
```
Namespace Language.Expressions

    The Base Interface for all Expression Types (E.g. Implemented by LiteralExpr and VariableExpr)
    Public Interface IExpression ...

    The Base Interface for all Exprs that contain some form of child Exprs (E.g. Implemented by Function
    Public Interface ICompoundExpression ...

#Region "IExpression (& ICompoundExpression) Implementations"

LBL Placeholder Exprs

IOT Placeholder Exprs

    Represents a DSString, DSNumber, or DSBoolean Literal
    Public Class LiteralExpr(Of TLiteral As {Variables.IDataValue, Class}) ...

    Represents an Expr which is the result of reading a Variable's value from a SymbolTable
    Public Class VariableExpr ...

    Holds the Arguments and Identifier of a Function, ready for this to be called during Resolve()
    Public Class FunctionCallExpr ...

    Holds the Operands and OperatorLiteral of a DSOperator
    Public Class OperatorExpr ...

#End Region

End Namespace
```

[Above] Namespace: DocScript.Language.Expressions



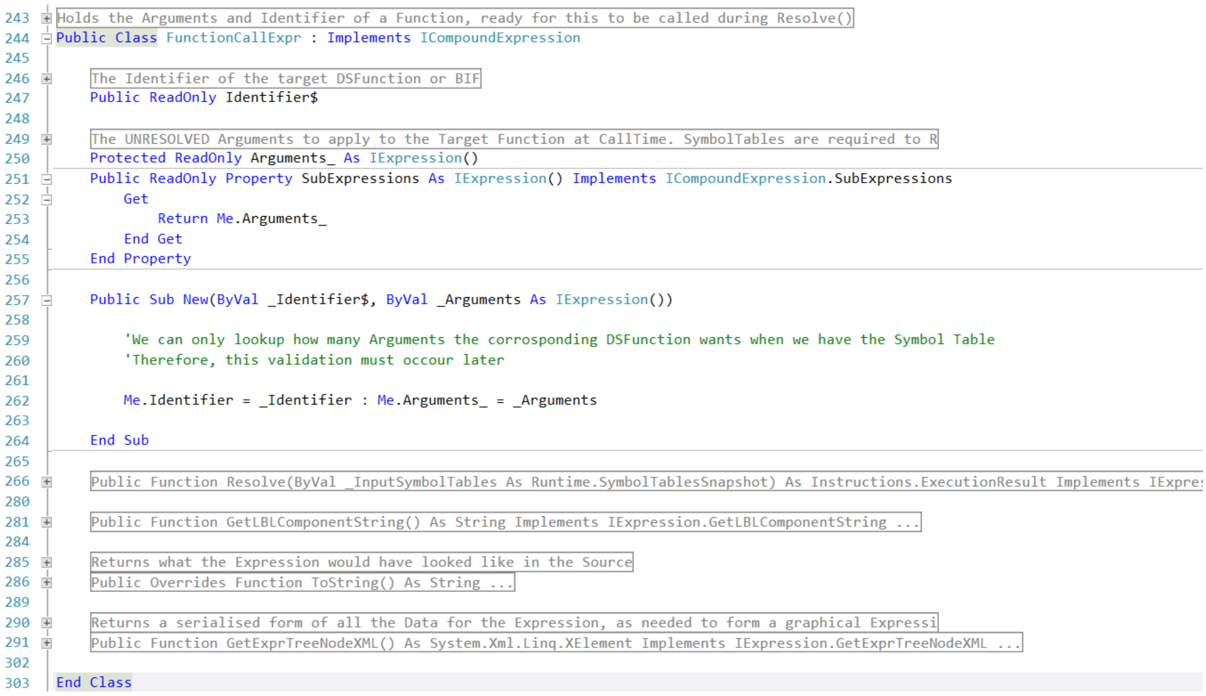
```
Represents a DSString, DSNumber, or DSBoolean Literal
Public Class LiteralExpr(Of TLiteral As {Variables.IDataValue, Class}) : Implements IExpression

    Public ReadOnly LiteralValue As TLiteral
    Public ReadOnly SourceToken As Runtime.Token

    Public Sub New(ByVal _LiteralValue As TLiteral, Optional ByVal _SourceToken As Runtime.Token = Nothing)
        If Not Variables.VariableUtilities.IsNonVoidFunctionReturnType(GetType(TLiteral)) Then Throw New Exception("The specified TLiter
        Me.SourceToken = _SourceToken
        Me.LiteralValue = _LiteralValue
    End Sub

    Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult Implements IExpres

    Public Function GetLBLComponentString() As String Implements IExpression.GetLBLComponentString ...

    Returns what the Expression would have looked like in the Source
    Public Overrides Function ToString() As String ...

    Returns a serialised form of all the Data for the Expression, as needed to form a graphical Expressi
    Public Function GetExprTreeNodeXML() As System.Xml.Linq.XElement Implements IExpression.GetExprTreeNodeXML ...

End Class
```

[Above] Class: LiteralExpr



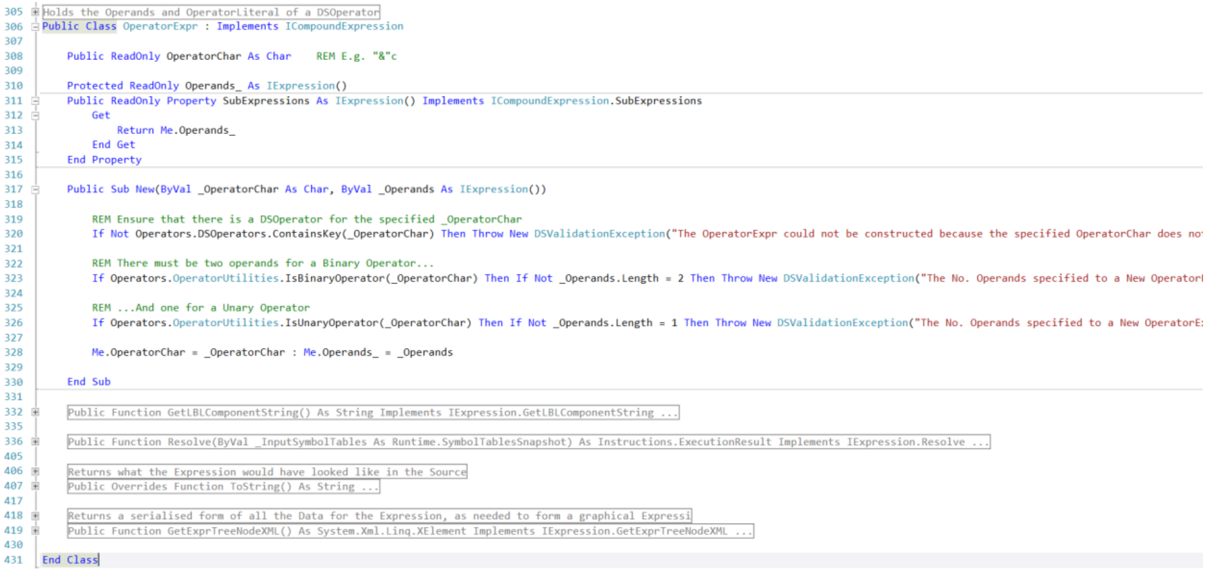
```
Represents an Expr which is the result of reading a Variable's value from a SymbolTable
Public Class VariableExpr : Implements IExpression

    Public ReadOnly Identifier$
    Public ReadOnly SourceToken As Runtime.Token 'Initialised to Nothing if not passed to Constructor

    Public Sub New(ByVal _Identifier$, Optional ByVal _SourceToken As Runtime.Token = Nothing)
        Me.SourceToken = _SourceToken
        Me.Identifier = _Identifier
    End Sub

    Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult Implements IExpres

    Public Function GetLBLComponentString() As String Implements IExpression.GetLBLComponentString ...

    Returns what the Expression would have looked like in the Source
    Public Overrides Function ToString() As String ...

    Returns a serialised form of all the Data for the Expression, as needed to form a graphical Expressi
    Public Function GetExprTreeNodeXML() As System.Xml.Linq.XElement Implements IExpression.GetExprTreeNodeXML ...

End Class
```

[Above] Class: VariableExpr



```
Holds the Arguments and Identifier of a Function, ready for this to be called during Resolve()
Public Class FunctionCallExpr : Implements ICompoundExpression

    The Identifier of the target DSFunction or BIF
    Public ReadOnly Identifier$

    The UNRESOLVED Arguments to apply to the Target Function at CallTime. SymbolTables are required to R
    Protected ReadOnly Arguments_ As IExpression()
    Public ReadOnly Property SubExpressions As IExpression() Implements ICompoundExpression.SubExpressions
        Get
            Return Me.Arguments_
        End Get
    End Property

    Public Sub New(ByVal _Identifier$, ByVal _Arguments As IExpression())

        'We can only lookup how many Arguments the corrosponding DSFunction wants when we have the Symbol Table
        'Therefore, this validation must occour later

        Me.Identifier = _Identifier : Me.Arguments_ = _Arguments

    End Sub

    Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult Implements IExpre:

    Public Function GetLBLComponentString() As String Implements IExpression.GetLBLComponentString ...

    Returns what the Expression would have looked like in the Source
    Public Overrides Function ToString() As String ...

    Returns a serialised form of all the Data for the Expression, as needed to form a graphical Expressi
    Public Function GetExprTreeNodeXML() As System.Xml.Linq.XElement Implements IExpression.GetExprTreeNodeXML ...

End Class
```

[Above] Class: FunctionCallExpr



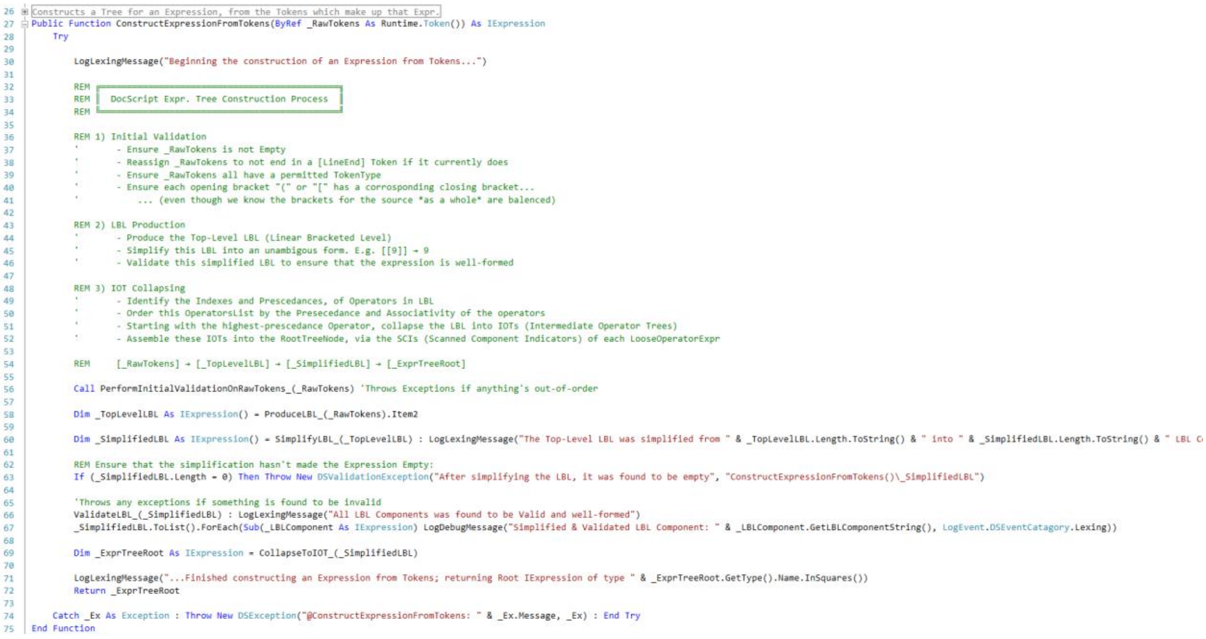
```
Holds the Operands and OperatorLiteral of a DSOperator
Public Class OperatorExpr : Implements ICompoundExpression

    Public ReadOnly OperatorChar As Char    REM E.g. "&"c

    Protected ReadOnly Operands_ As IExpression()
    Public ReadOnly Property SubExpressions As IExpression() Implements ICompoundExpression.SubExpressions
        Get
            Return Me.Operands_
        End Get
    End Property

    Public Sub New(ByVal _OperatorChar As Char, ByVal _Operands As IExpression())

        REM Ensure that there is a DSOperator for the specified _OperatorChar
        If Not Operators.DSOperators.ContainsKey(_OperatorChar) Then Throw New DSValidationException("The OperatorExpr could not be constructed because the specified OperatorChar does no

        REM There must be two operands for a Binary Operator...
        If Operators.OperatorUtilities.IsBinaryOperator(_OperatorChar) Then If Not _Operands.Length = 2 Then Throw New DSValidationException("The No. Operands specified to a New Operator

        REM ...And one for a Unary Operator
        If Operators.OperatorUtilities.IsUnaryOperator(_OperatorChar) Then If Not _Operands.Length = 1 Then Throw New DSValidationException("The No. Operands specified to a New OperatorE

        Me.OperatorChar = _OperatorChar : Me.Operands_ = _Operands

    End Sub

    Public Function GetLBLComponentString() As String Implements IExpression.GetLBLComponentString ...

    Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult Implements IExpression.Resolve ...

    Returns what the Expression would have looked like in the Source
    Public Overrides Function ToString() As String ...

    Returns a serialised form of all the Data for the Expression, as needed to form a graphical Expressi
    Public Function GetExprTreeNodeXML() As System.Xml.Linq.XElement Implements IExpression.GetExprTreeNodeXML ...

End Class
```

[Above] Class: OperatorExpr

Notably, the constructors to the IExpression types do **not** take in the Tokens required to construct each type of expression. This is because it is very difficult to determine what type of expression a set of tokens represents, just by looking at the tokens in order. At the token-level, it cannot even be guaranteed that the tokens make up a syntactically-valid expression. If done via the token-accepting constructors' method, each compound-expression type (`OperatorExpr` (because of Operands) and `FunctionCallExpr` (because of Function Arguments)) would also have to determine the type of its child expressions. This would be needlessly over-complicated.

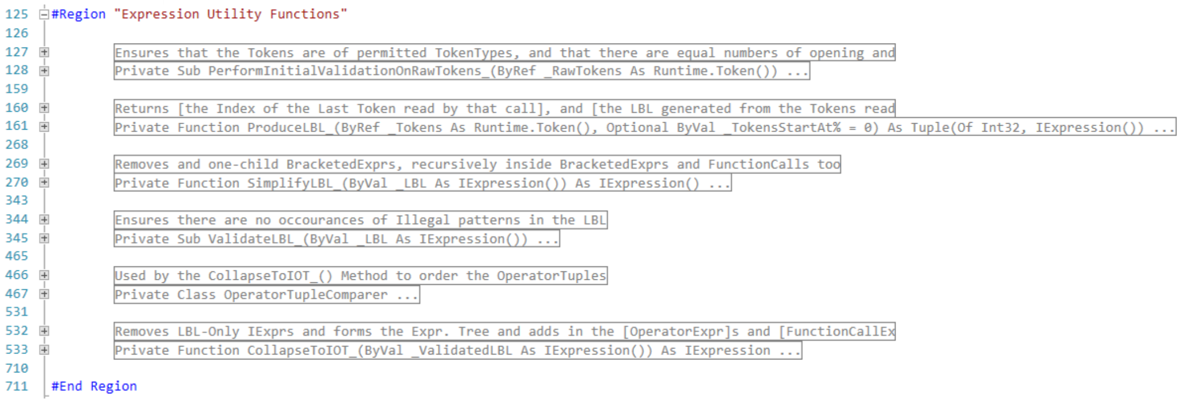
ConstructExpressionFromTokens()

For these reasons, I am instead going to use a single function called `ConstructExpressionFromTokens()`, which takes in an array of Tokens, and returns whatever the top-most Expression of the resultant Tree is. I have implemented it thusly:



```
Constructs a Tree for an Expression, from the Tokens which make up that Expr.
Public Function ConstructExpressionFromTokens(ByRef _RawTokens As Runtime.Token()) As IExpression
    Try

        LogLexingMessage("Beginning the construction of an Expression from Tokens...")

        REM ╔═══════════════════════════════════╗
        REM ║  DocScript Expr. Tree Construction Process  ║
        REM ╚═══════════════════════════════════╝

        REM 1) Initial Validation
        '       - Ensure _RawTokens is not Empty
        '       - Reassign _RawTokens to not end in a [LineEnd] Token if it currently does
        '       - Ensure _RawTokens all have a permitted TokenType
        '       - Ensure each opening bracket "(" or "[" has a corrosponding closing bracket...
        '           ... (even though we know the brackets for the source *as a whole* are balenced)

        REM 2) LBL Production
        '       - Produce the Top-Level LBL (Linear Bracketed Level)
        '       - Simplify this LBL into an unambigous form. E.g. [[9]] → 9
        '       - Validate this simplified LBL to ensure that the expression is well-formed

        REM 3) IOT Collapsing
        '       - Identify the Indexes and Prescedances, of Operators in LBL
        '       - Order this OperatorsList by the Presecedance and Associativity of the operators
        '       - Starting with the highest-prescedance Operator, collapse the LBL into IOTs (Intermediate Operator Trees)
        '       - Assemble these IOTs into the RootTreeNode, via the SCIs (Scanned Component Indicators) of each LooseOperatorExpr

        REM     [_RawTokens] → [_TopLevelLBL] → [_SimplifiedLBL] → [_ExprTreeRoot]

        Call PerformInitialValidationOnRawTokens_(_RawTokens) 'Throws Exceptions if anything's out-of-order

        Dim _TopLevelLBL As IExpression() = ProduceLBL_(_RawTokens).Item2

        Dim _SimplifiedLBL As IExpression() = SimplifyLBL_(_TopLevelLBL) : LogLexingMessage("The Top-Level LBL was simplified from " & _TopLevelLBL.Length.ToString() & " into " & _SimplifiedLBL.Length.ToString() & " LBL C

        REM Ensure that the simplification hasn't made the Expression Empty:
        If (_SimplifiedLBL.Length = 0) Then Throw New DSValidationException("After simplifying the LBL, it was found to be empty", "ConstructExpressionFromTokens()\_SimplifiedLBL")

        'Throws any exceptions if something is found to be invalid
        ValidateLBL_(_SimplifiedLBL) : LogLexingMessage("All LBL Components was found to be Valid and well-formed")
        _SimplifiedLBL.ToList().ForEach(Sub(_LBLComponent As IExpression) LogDebugMessage("Simplified & Validated LBL Component: " & _LBLComponent.GetLBLComponentString(), LogEvent.DSEventCatagory.Lexing))

        Dim _ExprTreeRoot As IExpression = CollapseToIOT_(_SimplifiedLBL)

        LogLexingMessage("...Finished constructing an Expression from Tokens; returning Root IExpression of type " & _ExprTreeRoot.GetType().Name.InSquares())
        Return _ExprTreeRoot

    Catch _Ex As Exception : Throw New DSException("@ConstructExpressionFromTokens: " & _Ex.Message, _Ex) : End Try
End Function
```

It relies of these 600 lines of backend functions:



```
#Region "Expression Utility Functions"

        Ensures that the Tokens are of permitted TokenTypes, and that there are equal numbers of opening and
        Private Sub PerformInitialValidationOnRawTokens_(ByRef _RawTokens As Runtime.Token()) ...

        Returns [the Index of the Last Token read by that call], and [the LBL generated from the Tokens read
        Private Function ProduceLBL_(ByRef _Tokens As Runtime.Token(), Optional ByVal _TokensStartAt% = 0) As Tuple(Of Int32, IExpression()) ...

        Removes and one-child BracketedExprs, recursively inside BracketedExprs and FunctionCalls too
        Private Function SimplifyLBL_(ByVal _LBL As IExpression()) As IExpression() ...

        Ensures there are no occourances of Illegal patterns in the LBL
        Private Sub ValidateLBL_(ByVal _LBL As IExpression()) ...

        Used by the CollapseToIOT_() Method to order the OperatorTuples
        Private Class OperatorTupleComparer ...

        Removes LBL-Only IExprs and forms the Expr. Tree and adds in the [OperatorExpr]s and [FunctionCallEx
        Private Function CollapseToIOT_(ByVal _ValidatedLBL As IExpression()) As IExpression ...

#End Region
```

How does that work then?

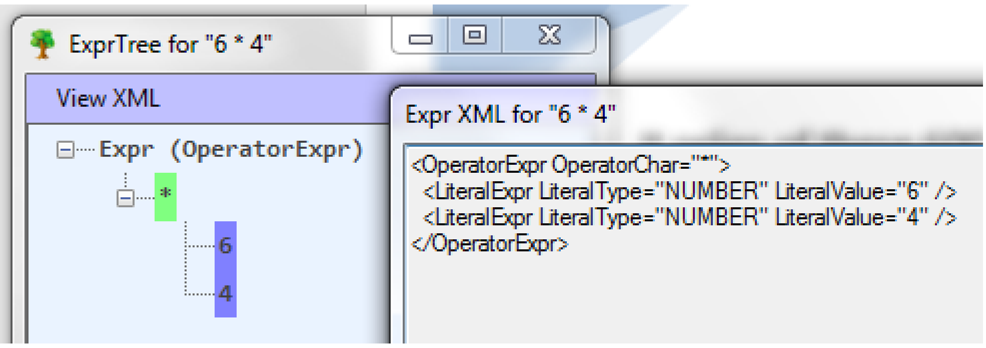
The basic principle for Expression Construction in DocScript is that a **Linear Bracketed Level (LBL)** is recursively produced for the expression. This is a top-level view of the expression, only caring about the sub-expressions which are connected *by operators*. Any sub-expressions implicated only by being arguments to a FunctionCall Expr, or by being components inside a BracketedExpr, are not visible from the LBL; they are abstracted away. The **recursive** part comes in here: Each [BracketedExpr] or FunctionCall() is itself represented as a Linear Bracketed Level. Each LBL can *contain* child LBLs of its own.

Once the LBL is produced, it's just a matter of **simplifying** it to the base-most form (for instance: `[[[3]]]` would be simplified to just `3`), and then **validating** it. During the LBL validation, the LBL is compared against a number of different invalid patterns. For instance {A Binary Operator} followed by {Another Binary Operator} is an invalid pattern for a DocScript Expression. Detailed and verbose error messages result from a validation-pattern-violating expression, which is beneficial to the programmer.

Testing: Example Constructed Expression Trees

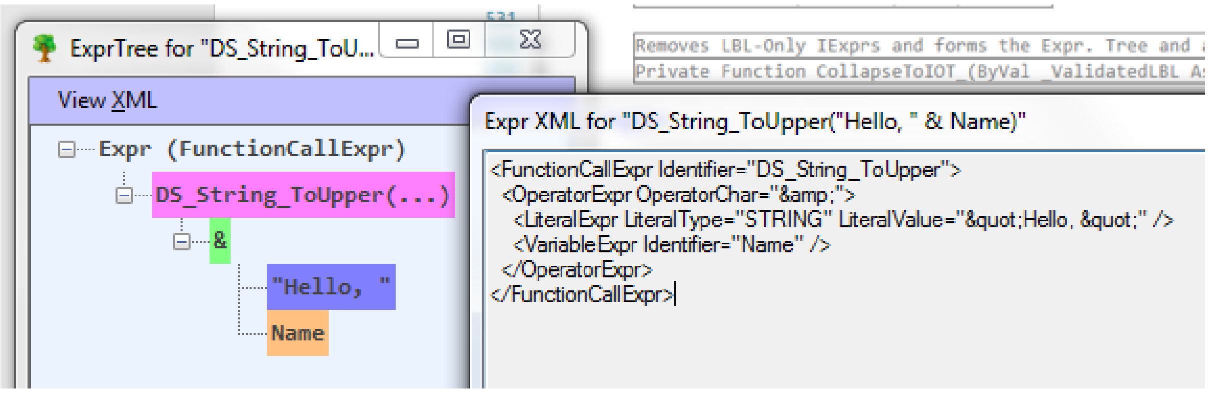
Implementing that into a simple [Source → Tokens → Expression] window on the DocScript.Experimentation project, shows clearly what the expression trees look like for input expressions:

`6 * 4`



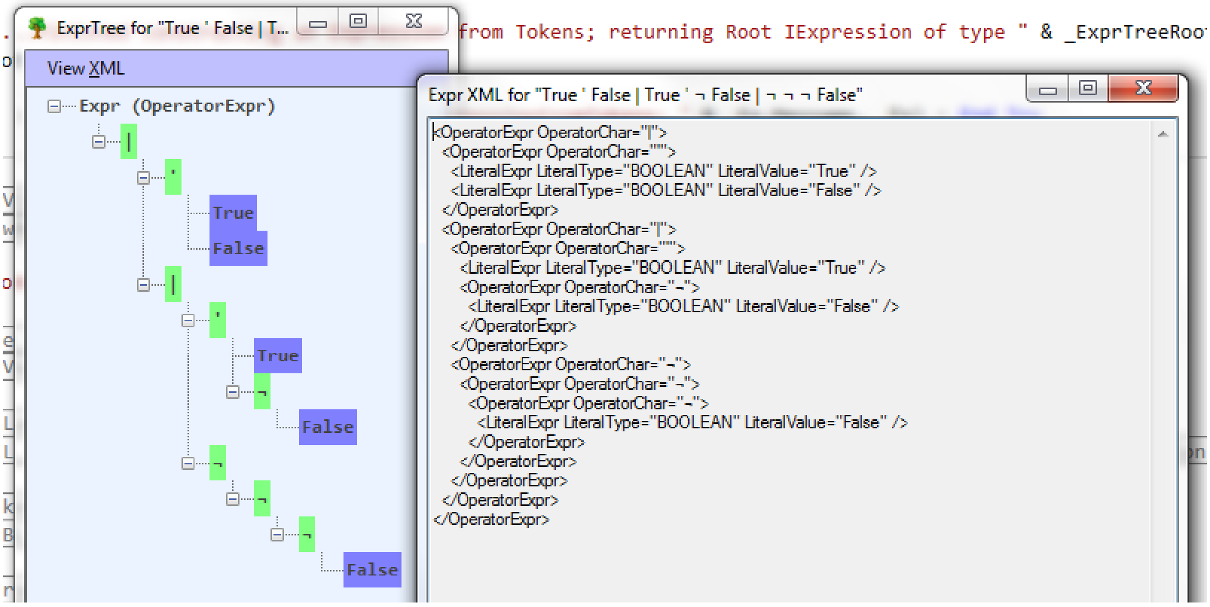
```
<OperatorExpr OperatorChar="*">
  <LiteralExpr LiteralType="NUMBER" LiteralValue="6" />
  <LiteralExpr LiteralType="NUMBER" LiteralValue="4" />
</OperatorExpr>
```

```
DS_String_ToUpper("Hello, " & Name)
```


ExprTree for "DS_String_ToU...
View XML
Expr (FunctionCallExpr)
DS_String_ToUpper(...)
&
"Hello, "
Name
Removes LBL-Only IExprs and forms the Expr. Tree and
Private Function CollapseToIOT_(ByVal _ValidatedLBL As
Expr XML for "DS_String_ToUpper("Hello, " & Name)"

```
<FunctionCallExpr Identifier="DS_String_ToUpper">
  <OperatorExpr OperatorChar="&amp;">
    <LiteralExpr LiteralType="STRING" LiteralValue="&quot;Hello, &quot;" />
    <VariableExpr Identifier="Name" />
  </OperatorExpr>
</FunctionCallExpr>
```

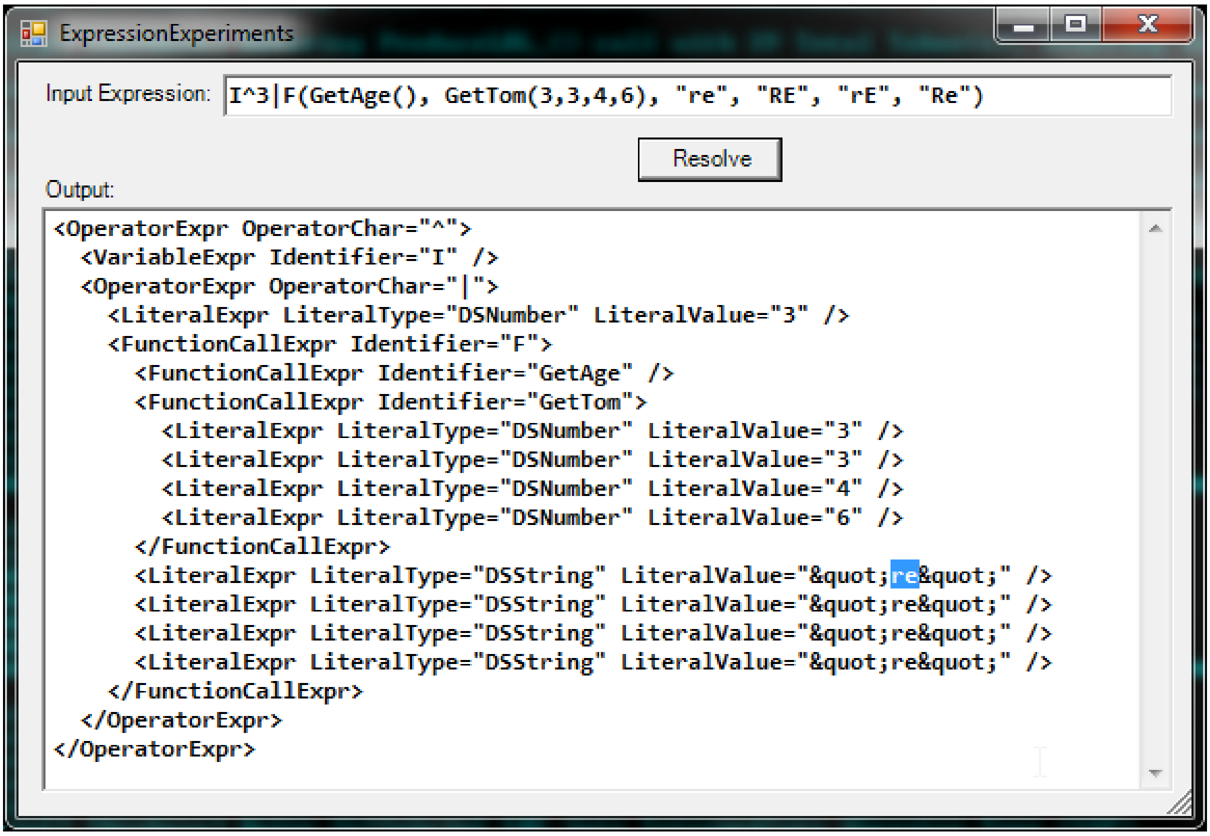
```
True ' False | [True ' ¬False | ¬¬¬False]
```


ExprTree for "True ' False | T...
View XML
Expr (OperatorExpr)
|
'
True
False
|
'
True
¬
False
¬
¬
¬
False
from Tokens; returning Root IExpression of type " & _ExprTreeRoo
Expr XML for "True ' False | True ' ¬ False | ¬ ¬ ¬ False"

```
<OperatorExpr OperatorChar="|">
  <OperatorExpr OperatorChar="'">
    <LiteralExpr LiteralType="BOOLEAN" LiteralValue="True" />
    <LiteralExpr LiteralType="BOOLEAN" LiteralValue="False" />
  </OperatorExpr>
  <OperatorExpr OperatorChar="|">
    <OperatorExpr OperatorChar="'">
      <LiteralExpr LiteralType="BOOLEAN" LiteralValue="True" />
      <OperatorExpr OperatorChar="¬">
        <LiteralExpr LiteralType="BOOLEAN" LiteralValue="False" />
      </OperatorExpr>
    </OperatorExpr>
    <OperatorExpr OperatorChar="¬">
      <OperatorExpr OperatorChar="¬">
        <OperatorExpr OperatorChar="¬">
          <LiteralExpr LiteralType="BOOLEAN" LiteralValue="False" />
        </OperatorExpr>
      </OperatorExpr>
    </OperatorExpr>
  </OperatorExpr>
</OperatorExpr>
```

Debugging

I identified a bug whereby the case of string literals was not being preserved. As can be seen here, I have 4 case-variations of the string literal "RE", but in the resultant Expression Tree, they are all appearing in lower-case!



```
I^3|F(GetAge(), GetTom(3,3,4,6), "re", "RE", "rE", "Re")
```

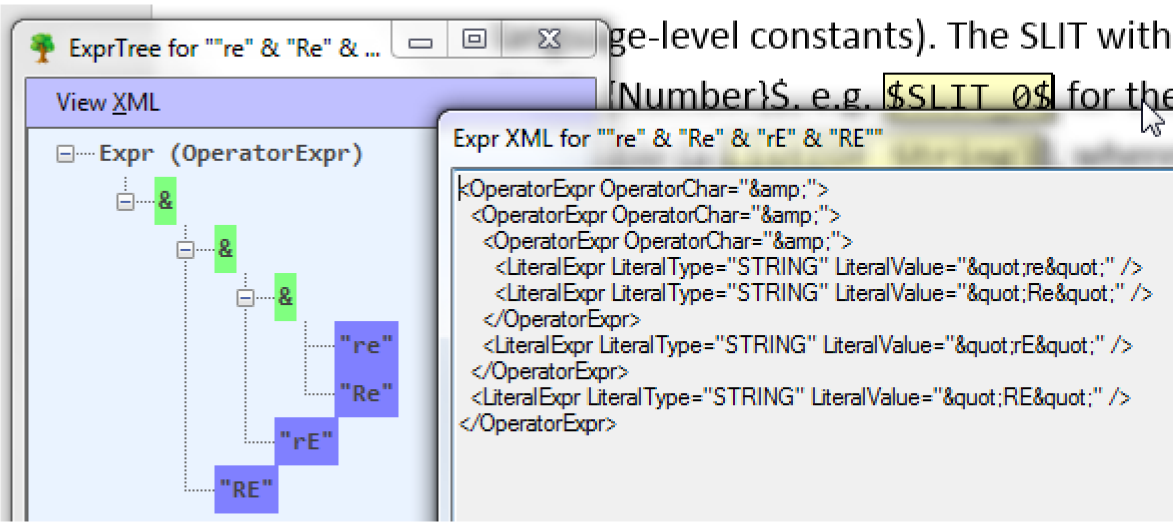
```
<OperatorExpr OperatorChar="^">
  <VariableExpr Identifier="I" />
  <OperatorExpr OperatorChar="|">
    <LiteralExpr LiteralType="DSNumber" LiteralValue="3" />
    <FunctionCallExpr Identifier="F">
      <FunctionCallExpr Identifier="GetAge" />
      <FunctionCallExpr Identifier="GetTom">
        <LiteralExpr LiteralType="DSNumber" LiteralValue="3" />
        <LiteralExpr LiteralType="DSNumber" LiteralValue="3" />
        <LiteralExpr LiteralType="DSNumber" LiteralValue="4" />
        <LiteralExpr LiteralType="DSNumber" LiteralValue="6" />
      </FunctionCallExpr>
      <LiteralExpr LiteralType="DSString" LiteralValue="&quot;re&quot;" />
      <LiteralExpr LiteralType="DSString" LiteralValue="&quot;re&quot;" />
      <LiteralExpr LiteralType="DSString" LiteralValue="&quot;re&quot;" />
      <LiteralExpr LiteralType="DSString" LiteralValue="&quot;re&quot;" />
    </FunctionCallExpr>
  </OperatorExpr>
</OperatorExpr>
```

My initial conclusion, was that I must have been unwittingly `ToLower()`-ing the string literal's value somewhere in the chain of functions it was being passed around. However, on stepping through (with F8 in Visual Studio), I realised that what was actually happening, was this:

- One of the very first things the Parser does, is to replace any string literals, with String-Literal-Indication-Tokens (SLITs). This is done via a regular expression which matches any character sequence starting and ending with the `StringLiteralStartEndChar` (as defined in the language-level constants). The SLIT with which the string literals are replaced takes the form $SLIT_{Number}$, e.g. `$SLIT_0$` for the first string-literal. The original strings are stored in the SLIT-Table (a `List(Of String)`), where the index of each String-Literal value is its SLIT {Number}.
- Much later on during parsing, when it comes to substituting the SLITs for the String-Literals again, the `Microsoft.VisualBasic.Val()` function is used, in order to extract an Int32 SLIT-Table Index, from the SLIT itself. My understanding of this function was as follows: It looks at the input string, ignores any non-digit characters, and then parses an Int32 from those remaining digit chars. However, after some quick testing in the Immediate Window in Visual Studio, I realised that it behaves a lot more inconsistently than I was hoping for. Therefore, I replaces this call to `Val(_SLIT)` with the simple LINQ-expression `Convert.ToInt32(New [String](_SLIT.Where(AddressOf [Char].IsDigit)))` which I should really have used in the first place.

- Lesson: *BEWARE THE ARCHAIC VB6 FUNCTIONS*!

Exchanging the Visual Basic 6 Function for a .NET replacement fixed the problem, so now, the correct Expression Tree is produced! ↓

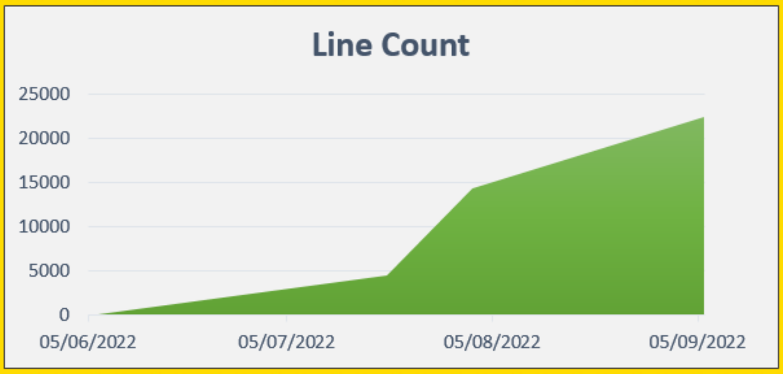

ExprTree for ""re" & "Re" & ...
View XML
Expr (OperatorExpr)
&
&
&
"re"
"Re"
"rE"
"RE"
ge-level constants). The SLIT with
Number}$. e.g. $SLIT_0$ for the
Expr XML for ""re" & "Re" & "rE" & "RE""

```
<OperatorExpr OperatorChar="&amp;">
  <OperatorExpr OperatorChar="&amp;">
    <OperatorExpr OperatorChar="&amp;">
      <LiteralExpr LiteralType="STRING" LiteralValue="&quot;re&quot;" />
      <LiteralExpr LiteralType="STRING" LiteralValue="&quot;Re&quot;" />
    </OperatorExpr>
    <LiteralExpr LiteralType="STRING" LiteralValue="&quot;rE&quot;" />
  </OperatorExpr>
  <LiteralExpr LiteralType="STRING" LiteralValue="&quot;RE&quot;" />
</OperatorExpr>
```

The extra line required for that fix accrues to the total of…

Progress Check

At this point in the development, I have written 22419 Lines of code…



Instruction Classes

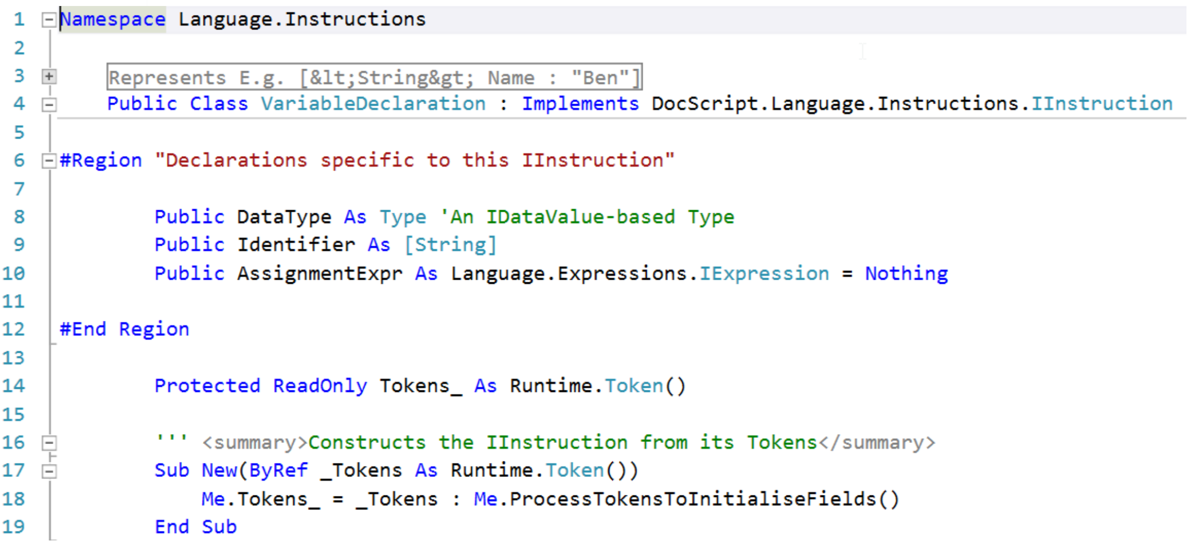
Of the eight `IInstruction`-derived classes, four of them ONLY implement `IInstruction`, and do not implement `IStatement`. In other words, `VariableDeclaration`, `VariableAssignment`, `ReturnToCaller`, and `FunctionCall` are just `IInstruction`s, whereas `IfStatement`, `WhileStatement`, `LoopStatement`, and `DSFunction` implement `IStatement` (and thereby implicitly also `IInstruction`) and therefore contain their own child Instructions.

Terminal Instructions

Because the terminal Instructions are the simpler subset, I shall begin with them.

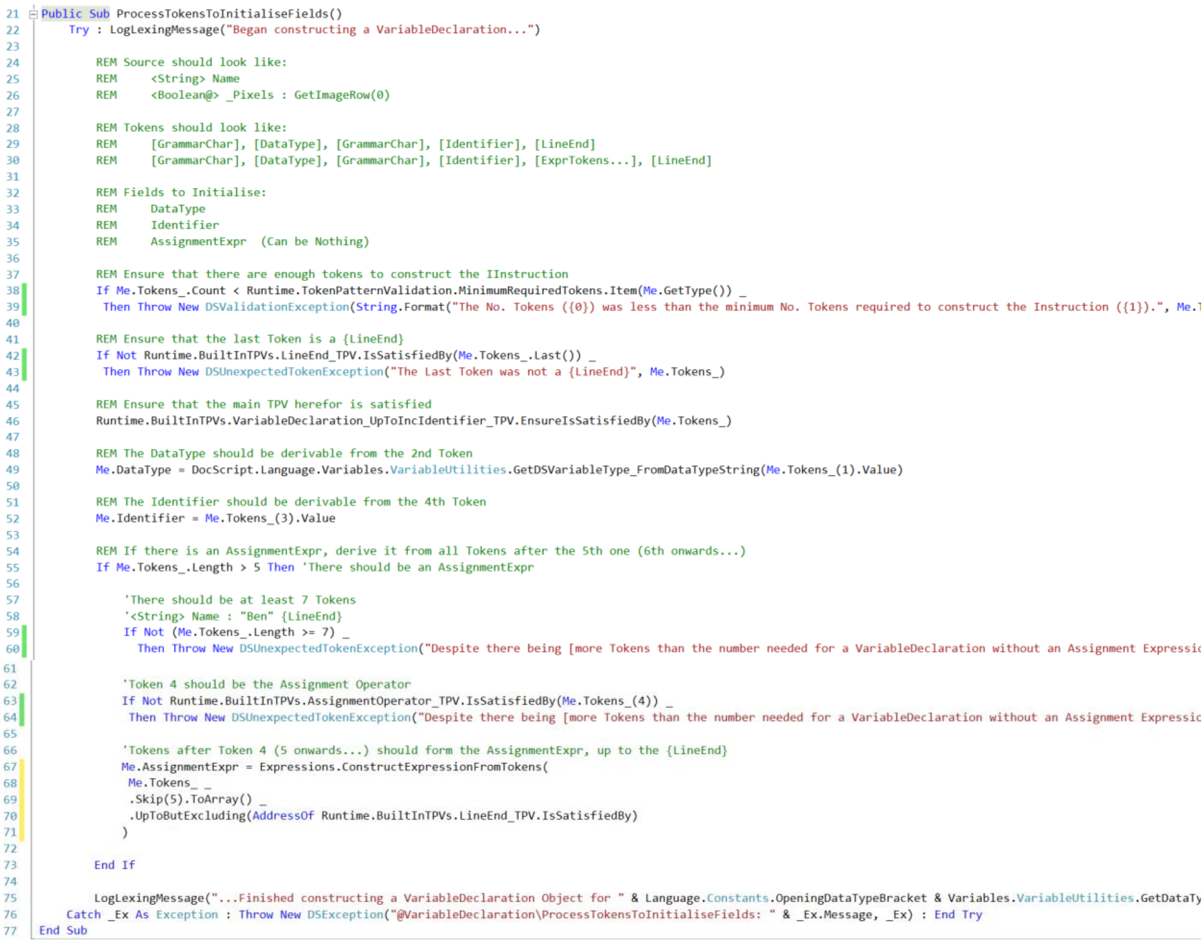
VariableDeclaration

The first is the `VariableDeclaration`, whose task it is to store a DataType, Identifier, and (optionally) AssignmentExpression, which will be used during Execution. I implemented the Class as follows…



```
Namespace Language.Instructions

    Represents E.g. [&lt;String&gt; Name : "Ben"]
    Public Class VariableDeclaration : Implements DocScript.Language.Instructions.IInstruction

#Region "Declarations specific to this IInstruction"

        Public DataType As Type 'An IDataValue-based Type
        Public Identifier As [String]
        Public AssignmentExpr As Language.Expressions.IExpression = Nothing

#End Region

        Protected ReadOnly Tokens_ As Runtime.Token()

        ''' <summary>Constructs the IInstruction from its Tokens</summary>
        Sub New(ByRef _Tokens As Runtime.Token())
            Me.Tokens_ = _Tokens : Me.ProcessTokensToInitialiseFields()
        End Sub
```

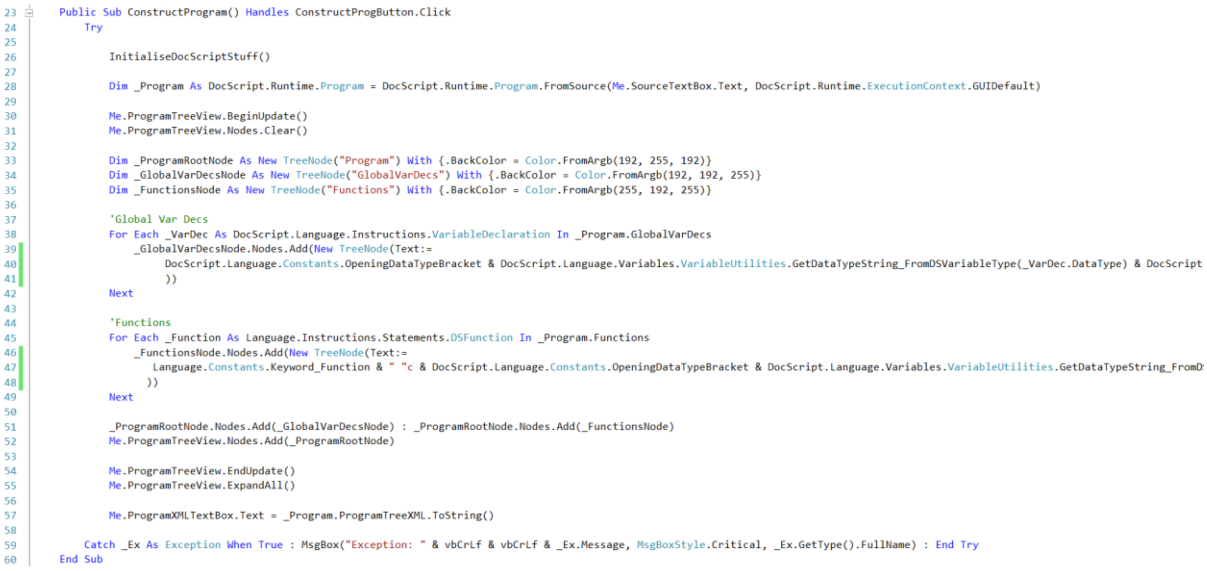
…And that ProcessTokensToInitialiseFields() Method looks like this:



```
Public Sub ProcessTokensToInitialiseFields()
    Try : LogLexingMessage("Began constructing a VariableDeclaration...")

        REM Source should look like:
        REM     <String> Name
        REM     <Boolean@> _Pixels : GetImageRow(0)

        REM Tokens should look like:
        REM     [GrammarChar], [DataType], [GrammarChar], [Identifier], [LineEnd]
        REM     [GrammarChar], [DataType], [GrammarChar], [Identifier], [ExprTokens...], [LineEnd]

        REM Fields to Initialise:
        REM     DataType
        REM     Identifier
        REM     AssignmentExpr  (Can be Nothing)

        REM Ensure that there are enough tokens to construct the IInstruction
        If Me.Tokens_.Count < Runtime.TokenPatternValidation.MinimumRequiredTokens.Item(Me.GetType()) _
         Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) was less than the minimum No. Tokens required to construct the Instruction ({1}).", Me.

        REM Ensure that the last Token is a {LineEnd}
        If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) _
         Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

        REM Ensure that the main TPV herefor is satisfied
        Runtime.BuiltInTPVs.VariableDeclaration_UpToIncIdentifier_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

        REM The DataType should be derivable from the 2nd Token
        Me.DataType = DocScript.Language.Variables.VariableUtilities.GetDSVariableType_FromDataTypeString(Me.Tokens_(1).Value)

        REM The Identifier should be derivable from the 4th Token
        Me.Identifier = Me.Tokens_(3).Value

        REM If there is an AssignmentExpr, derive it from all Tokens after the 5th one (6th onwards...)
        If Me.Tokens_.Length > 5 Then 'There should be an AssignmentExpr

            'There should be at least 7 Tokens
            '<String> Name : "Ben" {LineEnd}
            If Not (Me.Tokens_.Length >= 7) _
              Then Throw New DSUnexpectedTokenException("Despite there being [more Tokens than the number needed for a VariableDeclaration without an Expressi

            'Token 4 should be the Assignment Operator
            If Not Runtime.BuiltInTPVs.AssignmentOperator_TPV.IsSatisfiedBy(Me.Tokens_(4)) _
             Then Throw New DSUnexpectedTokenException("Despite there being [more Tokens than the number needed for a VariableDeclaration without an Assignment Expressi

            'Tokens after Token 4 (5 onwards...) should form the AssignmentExpr, up to the {LineEnd}
            Me.AssignmentExpr = Expressions.ConstructExpressionFromTokens(
             Me.Tokens_ _
             .Skip(5).ToArray() _
             .UpToButExcluding(AddressOf Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy)
            )

        End If

        LogLexingMessage("...Finished constructing a VariableDeclaration Object for " & Language.Constants.OpeningDataTypeBracket & Variables.VariableUtilities.GetDataTy
    Catch _Ex As Exception : Throw New DSException("@VariableDeclaration\ProcessTokensToInitialiseFields: " & _Ex.Message, _Ex) : End Try
End Sub
```

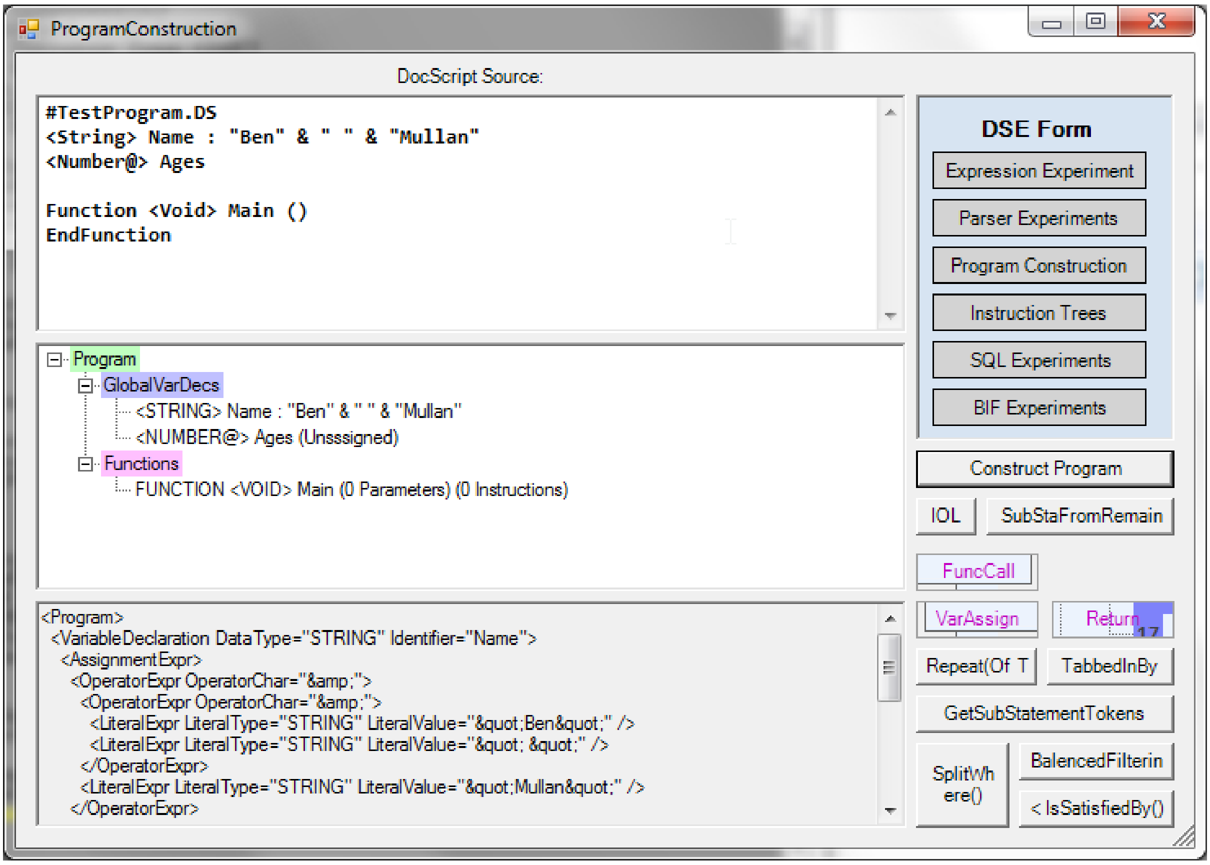
Testing

Because both Functions and Global-Variable-Declarations can appear at the top-statement (program) level, I can test this VariableDeclaration class, by writing a simple Program implementation:



```
Public Sub ConstructProgram() Handles ConstructProgButton.Click
    Try

        InitialiseDocScriptStuff()

        Dim _Program As DocScript.Runtime.Program = DocScript.Runtime.Program.FromSource(Me.SourceTextBox.Text, DocScript.Runtime.ExecutionContext.GUIDefault)

        Me.ProgramTreeView.BeginUpdate()
        Me.ProgramTreeView.Nodes.Clear()

        Dim _ProgramRootNode As New TreeNode("Program") With {.BackColor = Color.FromArgb(192, 255, 192)}
        Dim _GlobalVarDecsNode As New TreeNode("GlobalVarDecs") With {.BackColor = Color.FromArgb(192, 192, 255)}
        Dim _FunctionsNode As New TreeNode("Functions") With {.BackColor = Color.FromArgb(255, 192, 255)}

        'Global Var Decs
        For Each _VarDec As DocScript.Language.Instructions.VariableDeclaration In _Program.GlobalVarDecs
            _GlobalVarDecsNode.Nodes.Add(New TreeNode(Text:=
                 DocScript.Language.Constants.OpeningDataTypeBracket & DocScript.Language.Variables.VariableUtilities.GetDataTypeString_FromDSVariableType(_VarDec.DataType) & DocScript
                 ))
        Next

        'Functions
        For Each _Function As Language.Instructions.Statements.DSFunction In _Program.Functions
            _FunctionsNode.Nodes.Add(New TreeNode(Text:=
               Language.Constants.Keyword_Function & " "c & DocScript.Language.Constants.OpeningDataTypeBracket & DocScript.Language.Variables.VariableUtilities.GetDataTypeString_FromD
              ))
        Next

        _ProgramRootNode.Nodes.Add(_GlobalVarDecsNode) : _ProgramRootNode.Nodes.Add(_FunctionsNode)
        Me.ProgramTreeView.Nodes.Add(_ProgramRootNode)

        Me.ProgramTreeView.EndUpdate()
        Me.ProgramTreeView.ExpandAll()

        Me.ProgramXMLTextBox.Text = _Program.ProgramTreeXML.ToString()

    Catch _Ex As Exception When True : MsgBox("Exception: " & vbCrLf & vbCrLf & _Ex.Message, MsgBoxStyle.Critical, _Ex.GetType().FullName) : End Try
End Sub
```

This – along with a simple testing window in the DocScript.Experimentation project, allows me to see that Program Construction is working correctly, with just Global VariableDeclarations:



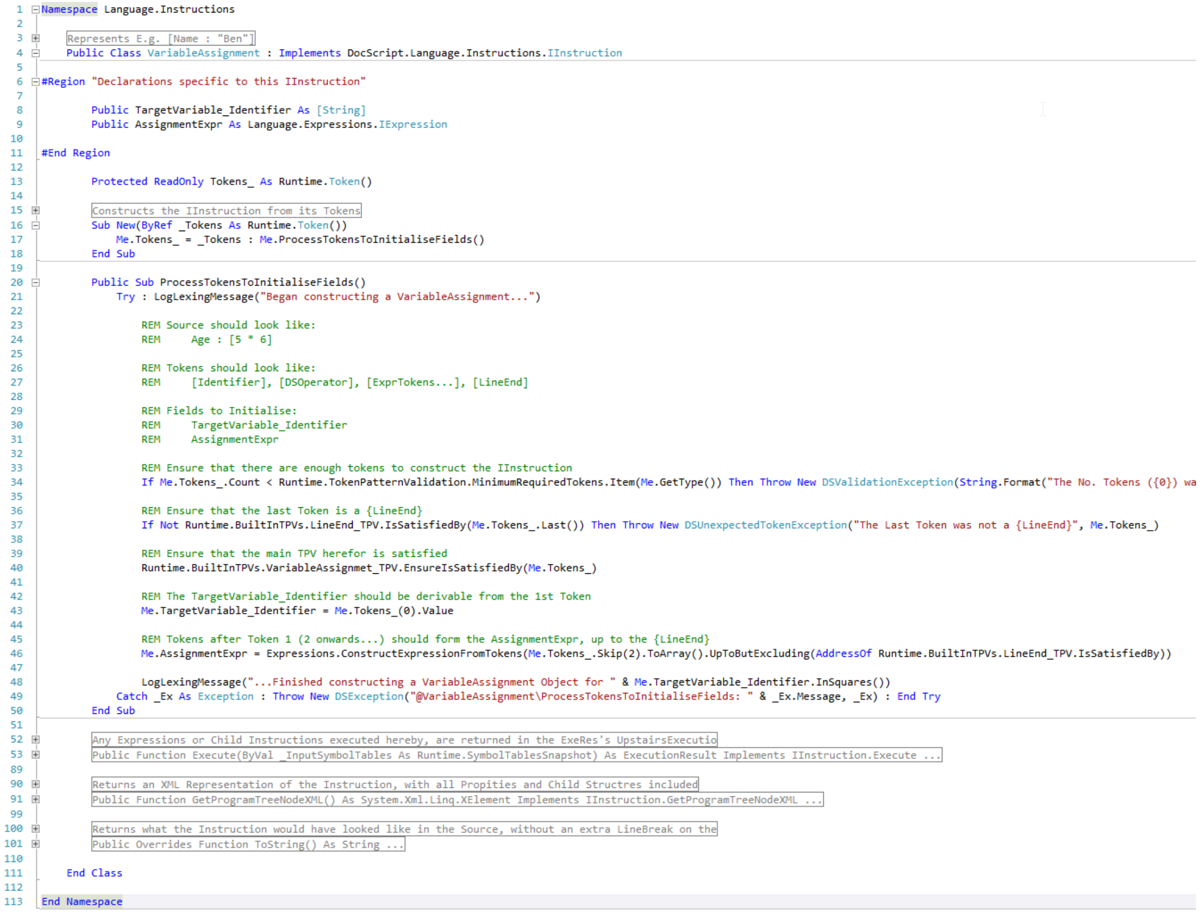
```
#TestProgram.DS
<String> Name : "Ben" & " " & "Mullan"
<Number@> Ages

Function <Void> Main ()
EndFunction
```

```
Program
    GlobalVarDecs
        <STRING> Name : "Ben" & " " & "Mullan"
        <NUMBER@> Ages (Unssigned)
    Functions
        FUNCTION <VOID> Main (0 Parameters) (0 Instructions)
```

```
<Program>
 <VariableDeclaration DataType="STRING" Identifier="Name">
  <AssignmentExpr>
   <OperatorExpr OperatorChar="&amp;">
    <OperatorExpr OperatorChar="&amp;">
     <LiteralExpr LiteralType="STRING" LiteralValue="&quot;Ben&quot;" />
     <LiteralExpr LiteralType="STRING" LiteralValue="&quot; &quot;" />
    </OperatorExpr>
    <LiteralExpr LiteralType="STRING" LiteralValue="&quot;Mullan&quot;" />
   </OperatorExpr>
```

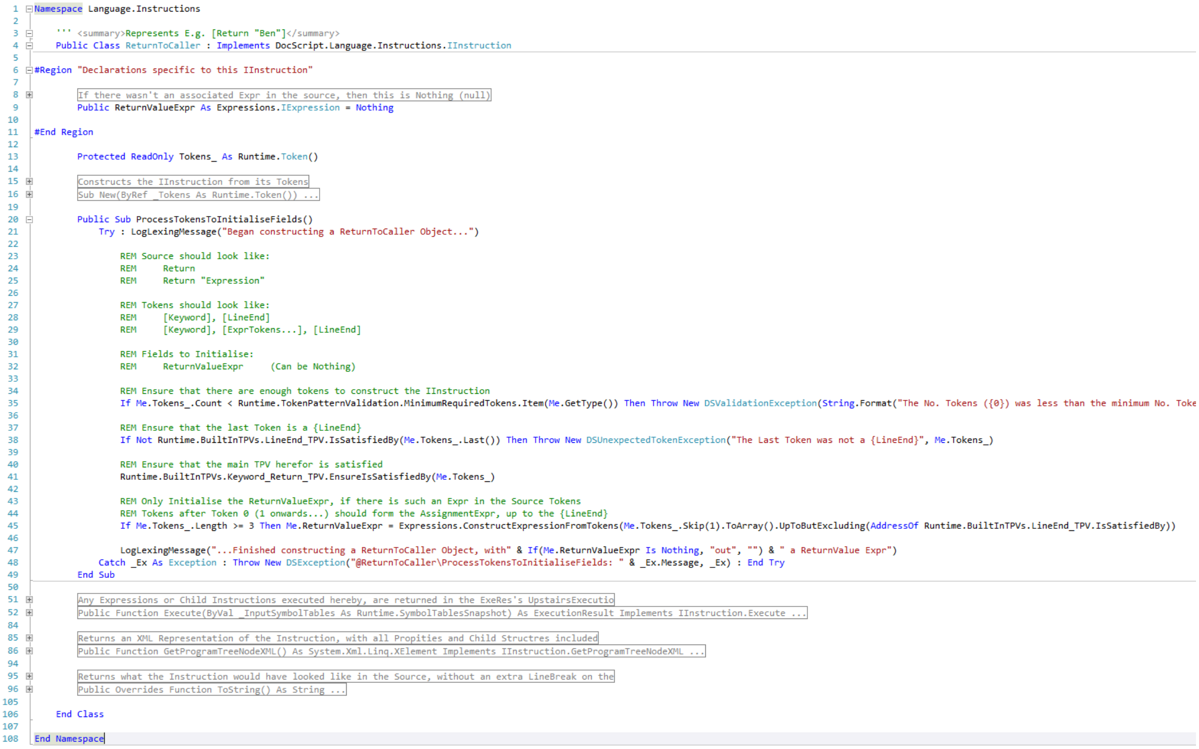
VariableAssignment, ReturnToCaller, and FunctionCall

The other three terminal instructions are fairly simple and similar; they accept Tokens to their constructors, and call an internal ProcessTokensToInitialiseFields() method. The classes look like this:



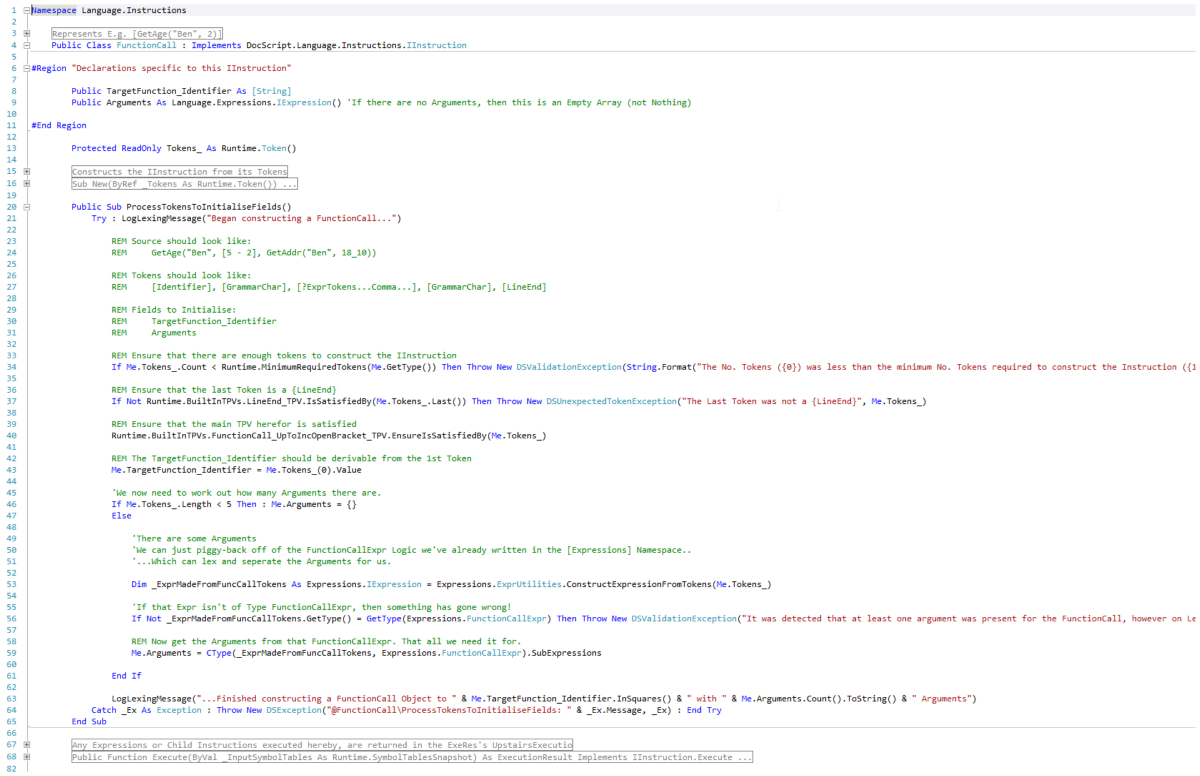
```
Namespace Language.Instructions

    Represents E.g. [Name : "Ben"]
    Public Class VariableAssignment : Implements DocScript.Language.Instructions.IInstruction

#Region "Declarations specific to this IInstruction"

        Public TargetVariable_Identifier As [String]
        Public AssignmentExpr As Language.Expressions.IExpression

#End Region

        Protected ReadOnly Tokens_ As Runtime.Token()

        Constructs the IInstruction from its Tokens
        Sub New(ByRef _Tokens As Runtime.Token())
            Me.Tokens_ = _Tokens : Me.ProcessTokensToInitialiseFields()
        End Sub

        Public Sub ProcessTokensToInitialiseFields()
            Try : LogLexingMessage("Began constructing a VariableAssignment...")

                REM Source should look like:
                REM     Age : [5 * 6]

                REM Tokens should look like:
                REM     [Identifier], [DSOperator], [ExprTokens...], [LineEnd]

                REM Fields to Initialise:
                REM     TargetVariable_Identifier
                REM     AssignmentExpr

                REM Ensure that there are enough tokens to construct the IInstruction
                If Me.Tokens_.Count < Runtime.TokenPatternValidation.MinimumRequiredTokens.Item(Me.GetType()) Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) wa

                REM Ensure that the last Token is a {LineEnd}
                If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

                REM Ensure that the main TPV herefor is satisfied
                Runtime.BuiltInTPVs.VariableAssignmet_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

                REM The TargetVariable_Identifier should be derivable from the 1st Token
                Me.TargetVariable_Identifier = Me.Tokens_(0).Value

                REM Tokens after Token 1 (2 onwards...) should form the AssignmentExpr, up to the {LineEnd}
                Me.AssignmentExpr = Expressions.ConstructExpressionFromTokens(Me.Tokens_.Skip(2).ToArray().UpToButExcluding(AddressOf Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy))

                LogLexingMessage("...Finished constructing a VariableAssignment Object for " & Me.TargetVariable_Identifier.InSquares())
            Catch _Ex As Exception : Throw New DSException("@VariableAssignment\ProcessTokensToInitialiseFields: " & _Ex.Message, _Ex) : End Try
        End Sub

        Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutio
        Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute ...

        Returns an XML Representation of the Instruction, with all Propities and Child Structres included
        Public Function GetProgramTreeNodeXML() As System.Xml.Linq.XElement Implements IInstruction.GetProgramTreeNodeXML ...

        Returns what the Instruction would have looked like in the Source, without an extra LineBreak on the
        Public Overrides Function ToString() As String ...

    End Class

End Namespace
```

[Above] Class: VariableAssignment



```
Namespace Language.Instructions

    ''' <summary>Represents E.g. [Return "Ben"]</summary>
    Public Class ReturnToCaller : Implements DocScript.Language.Instructions.IInstruction

#Region "Declarations specific to this IInstruction"

        If there wasn't an associated Expr in the source, then this is Nothing (null)
        Public ReturnValueExpr As Expressions.IExpression = Nothing

#End Region

        Protected ReadOnly Tokens_ As Runtime.Token()

        Constructs the IInstruction from its Tokens
        Sub New(ByRef _Tokens As Runtime.Token()) ...

        Public Sub ProcessTokensToInitialiseFields()
            Try : LogLexingMessage("Began constructing a ReturnToCaller Object...")

                REM Source should look like:
                REM     Return
                REM     Return "Expression"

                REM Tokens should look like:
                REM     [Keyword], [LineEnd]
                REM     [Keyword], [ExprTokens...], [LineEnd]

                REM Fields to Initialise:
                REM     ReturnValueExpr     (Can be Nothing)

                REM Ensure that there are enough tokens to construct the IInstruction
                If Me.Tokens_.Count < Runtime.TokenPatternValidation.MinimumRequiredTokens.Item(Me.GetType()) Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) was less than the minimum No. Toke

                REM Ensure that the last Token is a {LineEnd}
                If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

                REM Ensure that the main TPV herefor is satisfied
                Runtime.BuiltInTPVs.Keyword_Return_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

                REM Only Initialise the ReturnValueExpr, if there is such an Expr in the Source Tokens
                REM Tokens after Token 0 (1 onwards...) should form the AssignmentExpr, up to the {LineEnd}
                If Me.Tokens_.Length >= 3 Then Me.ReturnValueExpr = Expressions.ConstructExpressionFromTokens(Me.Tokens_.Skip(1).ToArray().UpToButExcluding(AddressOf Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy))

                LogLexingMessage("...Finished constructing a ReturnToCaller Object, with" & If(Me.ReturnValueExpr Is Nothing, "out", "") & " a ReturnValue Expr")
            Catch _Ex As Exception : Throw New DSException("@ReturnToCaller\ProcessTokensToInitialiseFields: " & _Ex.Message, _Ex) : End Try
        End Sub

        Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutio
        Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute ...

        Returns an XML Representation of the Instruction, with all Propities and Child Structres included
        Public Function GetProgramTreeNodeXML() As System.Xml.Linq.XElement Implements IInstruction.GetProgramTreeNodeXML ...

        Returns what the Instruction would have looked like in the Source, without an extra LineBreak on the
        Public Overrides Function ToString() As String ...

    End Class

End Namespace
```

[Above] Class: ReturnToCaller



```
Namespace Language.Instructions

    Represents E.g. [GetAge("Ben", 2)]
    Public Class FunctionCall : Implements DocScript.Language.Instructions.IInstruction

#Region "Declarations specific to this IInstruction"

        Public TargetFunction_Identifier As [String]
        Public Arguments As Language.Expressions.IExpression() 'If there are no Arguments, then this is an Empty Array (not Nothing)

#End Region

        Protected ReadOnly Tokens_ As Runtime.Token()

        Constructs the IInstruction from its Tokens
        Sub New(ByRef _Tokens As Runtime.Token()) ...

        Public Sub ProcessTokensToInitialiseFields()
            Try : LogLexingMessage("Began constructing a FunctionCall...")

                REM Source should look like:
                REM     GetAge("Ben", [5 - 2], GetAddr("Ben", 18_10))

                REM Tokens should look like:
                REM     [Identifier], [GrammarChar], [?ExprTokens...Comma...], [GrammarChar], [LineEnd]

                REM Fields to Initialise:
                REM     TargetFunction_Identifier
                REM     Arguments

                REM Ensure that there are enough tokens to construct the IInstruction
                If Me.Tokens_.Count < Runtime.MinimumRequiredTokens(Me.GetType()) Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) was less than the minimum No. Tokens required to construct the Instruction ({1

                REM Ensure that the last Token is a {LineEnd}
                If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

                REM Ensure that the main TPV herefor is satisfied
                Runtime.BuiltInTPVs.FunctionCall_UpToIncOpenBracket_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

                REM The TargetFunction_Identifier should be derivable from the 1st Token
                Me.TargetFunction_Identifier = Me.Tokens_(0).Value

                'We now need to work out how many Arguments there are.
                If Me.Tokens_.Length < 5 Then : Me.Arguments = {}
                Else

                    'There are some Arguments
                    'We can just piggy-back off of the FunctionCallExpr Logic we've already written in the [Expressions] Namespace..
                    '...Which can lex and seperate the Arguments for us.

                    Dim _ExprMadeFromFuncCallTokens As Expressions.IExpression = Expressions.ExprUtilities.ConstructExpressionFromTokens(Me.Tokens_)

                    'If that Expr isn't of Type FunctionCallExpr, then something has gone wrong!
                    If Not _ExprMadeFromFuncCallTokens.GetType() = GetType(Expressions.FunctionCallExpr) Then Throw New DSValidationException("It was detected that at least one argument was present for the FunctionCall, however on Le

                    REM Now get the Arguments from that FunctionCallExpr. That all we need it for.
                    Me.Arguments = CType(_ExprMadeFromFuncCallTokens, Expressions.FunctionCallExpr).SubExpressions

                End If

                LogLexingMessage("...Finished constructing a FunctionCall Object to " & Me.TargetFunction_Identifier.InSquares() & " with " & Me.Arguments.Count().ToString() & " Arguments")
            Catch _Ex As Exception : Throw New DSException("@FunctionCall\ProcessTokensToInitialiseFields: " & _Ex.Message, _Ex) : End Try
        End Sub

        Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutio
        Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute ...
```

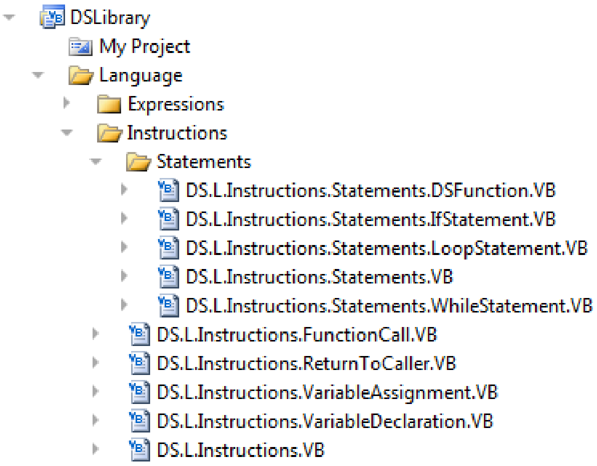
[Above] Class: FunctionCall

I will only be able to test these three latter terminal `IInstruction`-Types, when the four `IStatement`-Classes have been written (at which point, I shall test ***whole programs***).

Statement Instructions

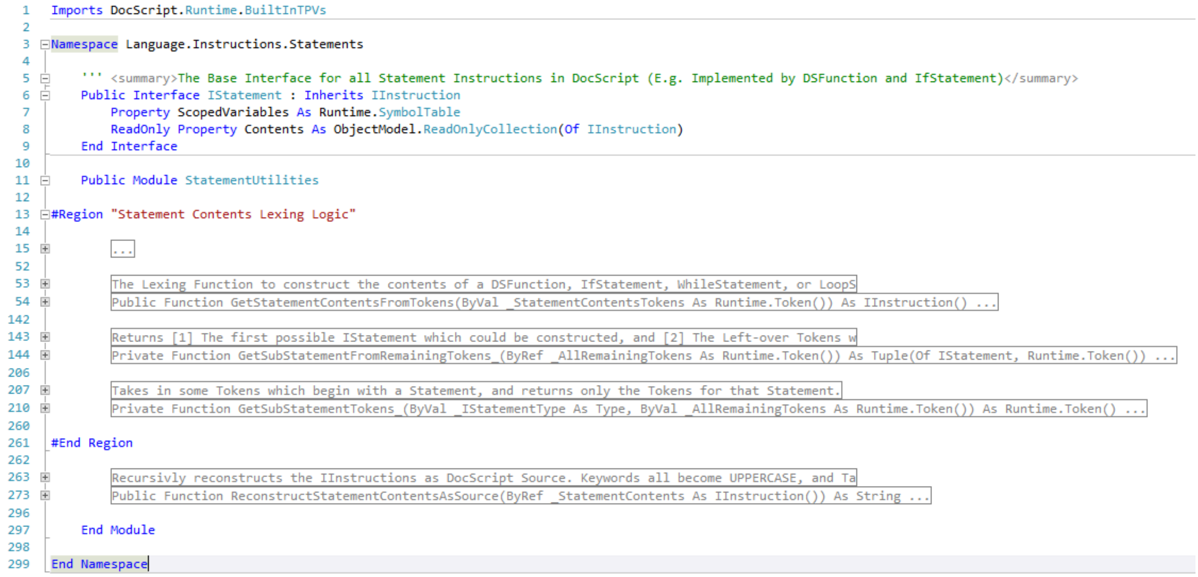
The principal distinction between the terminal- and statement-Instructions is that the Statement-Instructions contain a Contents member which represents all Instructions between that start of that Statement, and its corresponding `End{StatementType}` Token.

This is how I implemented the four `IStatement` classes:



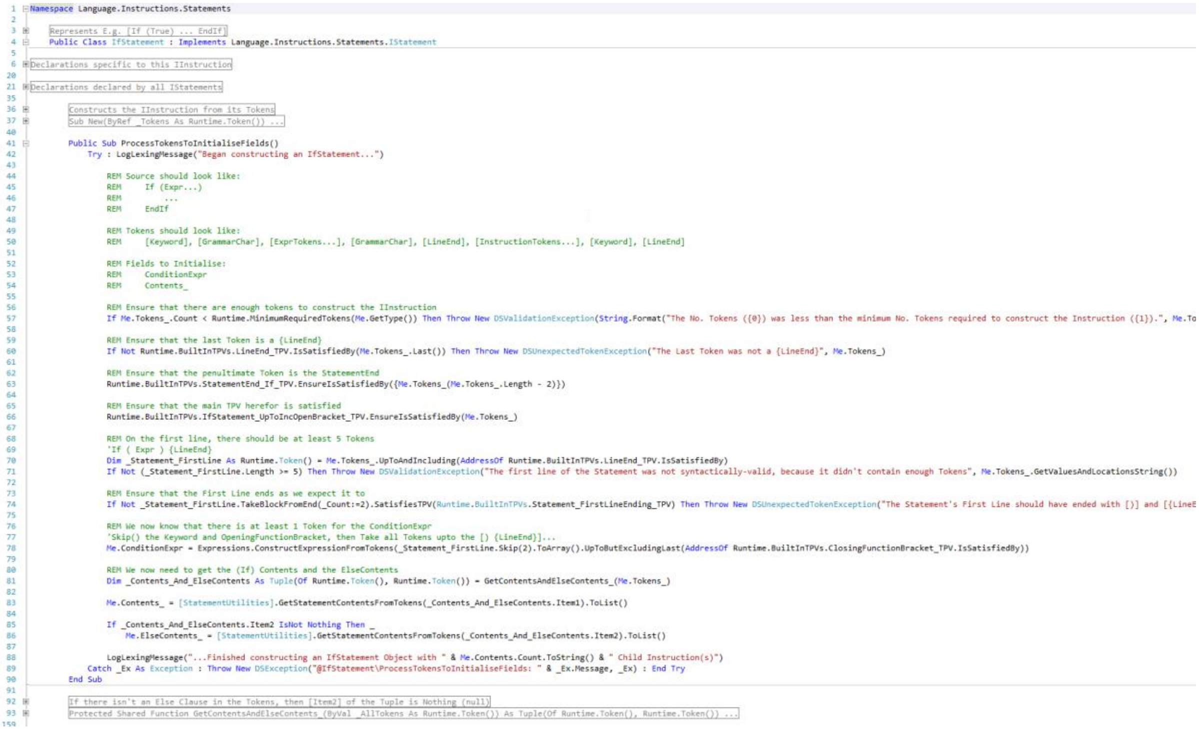
```
DSLibrary
    My Project
    Language
        Expressions
        Instructions
            Statements
                DS.L.Instructions.Statements.DSFunction.VB
                DS.L.Instructions.Statements.IfStatement.VB
                DS.L.Instructions.Statements.LoopStatement.VB
                DS.L.Instructions.Statements.VB
                DS.L.Instructions.Statements.WhileStatement.VB
            DS.L.Instructions.FunctionCall.VB
            DS.L.Instructions.ReturnToCaller.VB
            DS.L.Instructions.VariableAssignment.VB
            DS.L.Instructions.VariableDeclaration.VB
            DS.L.Instructions.VB
```

The Namespace Structure →



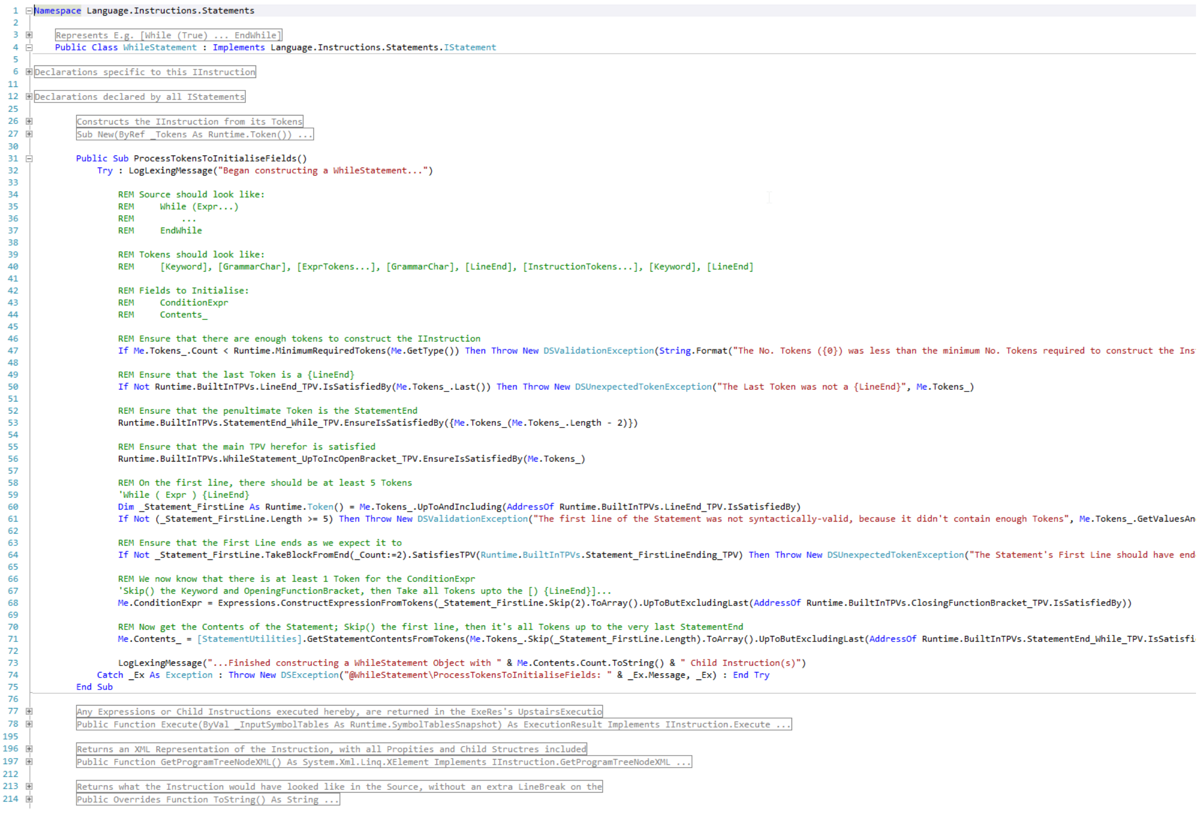
```
Imports DocScript.Runtime.BuiltInTPVs

Namespace Language.Instructions.Statements

    ''' <summary>The Base Interface for all Statement Instructions in DocScript (E.g. Implemented by DSFunction and IfStatement)</summary>
    Public Interface IStatement : Inherits IInstruction
        Property ScopedVariables As Runtime.SymbolTable
        ReadOnly Property Contents As ObjectModel.ReadOnlyCollection(Of IInstruction)
    End Interface

    Public Module StatementUtilities

#Region "Statement Contents Lexing Logic"

        ...

        The Lexing Function to construct the contents of a DSFunction, IfStatement, WhileStatement, or LoopS
        Public Function GetStatementContentsFromTokens(ByVal _StatementContentsTokens As Runtime.Token()) As IInstruction() ...

        Returns [1] The first possible IStatement which could be constructed, and [2] The Left-over Tokens w
        Private Function GetSubStatementFromRemainingTokens_(ByRef _AllRemainingTokens As Runtime.Token()) As Tuple(Of IStatement, Runtime.Token()) ...

        Takes in some Tokens which begin with a Statement, and returns only the Tokens for that Statement.
        Private Function GetSubStatementTokens_(ByVal _IStatementType As Type, ByVal _AllRemainingTokens As Runtime.Token()) As Runtime.Token() ...

#End Region

        Recursivly reconstructs the IInstructions as DocScript Source. Keywords all become UPPERCASE, and Ta
        Public Function ReconstructStatementContentsAsSource(ByRef _StatementContents As IInstruction()) As String ...

    End Module

End Namespace
```

[Above] Module: StatementUtilities



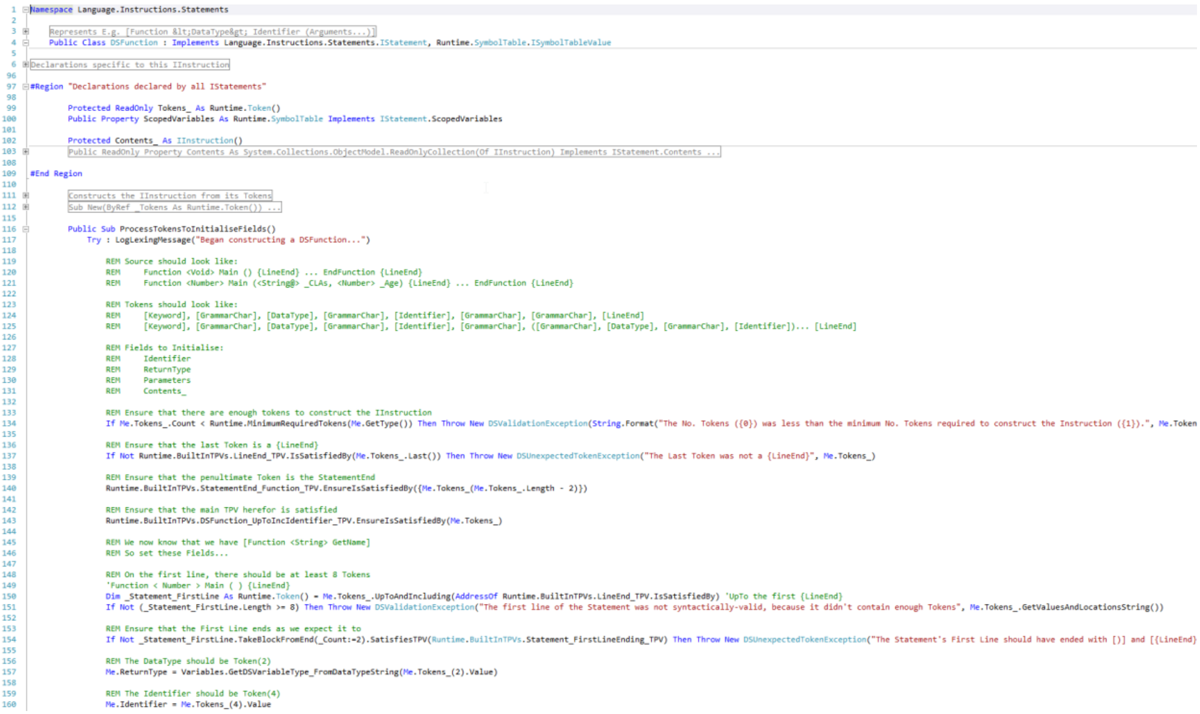
```
Namespace Language.Instructions.Statements

    Represents E.g. [If (True) ... EndIf]
    Public Class IfStatement : Implements Language.Instructions.Statements.IStatement

Declarations specific to this IInstruction

Declarations declared by all IStatements

        Constructs the IInstruction from its Tokens
        Sub New(ByRef _Tokens As Runtime.Token()) ...

        Public Sub ProcessTokensToInitialiseFields()
            Try : LogLexingMessage("Began constructing an IfStatement...")

                REM Source should look like:
                REM     If (Expr...)
                REM         ...
                REM     EndIf

                REM Tokens should look like:
                REM     [Keyword], [GrammarChar], [ExprTokens...], [GrammarChar], [LineEnd], [InstructionTokens...], [Keyword], [LineEnd]

                REM Fields to Initialise:
                REM     ConditionExpr
                REM     Contents_

                REM Ensure that there are enough tokens to construct the IInstruction
                If Me.Tokens_.Count < Runtime.MinimumRequiredTokens(Me.GetType()) Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) was less than the minimum No. Tokens required to construct the Instruction ({1}).", Me.To

                REM Ensure that the last Token is a {LineEnd}
                If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

                REM Ensure that the penultimate Token is the StatementEnd
                Runtime.BuiltInTPVs.StatementEnd_If_TPV.EnsureIsSatisfiedBy({Me.Tokens_(Me.Tokens_.Length - 2)})

                REM Ensure that the main TPV herefor is satisfied
                Runtime.BuiltInTPVs.IfStatement_UpToIncOpenBracket_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

                REM On the first line, there should be at least 5 Tokens
                'If ( Expr ) {LineEnd}
                Dim _Statement_FirstLine As Runtime.Token() = Me.Tokens_.UpToAndIncluding(AddressOf Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy)
                If Not (_Statement_FirstLine.Length >= 5) Then Throw New DSValidationException("The first line of the Statement was not syntactically-valid, because it didn't contain enough Tokens", Me.Tokens_.GetValuesAndLocationsString())

                REM Ensure that the First Line ends as we expect it to
                If Not _Statement_FirstLine.TakeBlockFromEnd(_Count:=2).SatisfiesTPV(Runtime.BuiltInTPVs.Statement_FirstLineEnding_TPV) Then Throw New DSUnexpectedTokenException("The Statement's First Line should have ended with [)] and [{LineE

                REM We now know that there is at least 1 Token for the ConditionExpr
                'Skip() the Keyword and OpeningFunctionBracket, then Take all Tokens upto the [) {LineEnd}]...
                Me.ConditionExpr = Expressions.ConstructExpressionFromTokens(_Statement_FirstLine.Skip(2).ToArray().UpToButExcludingLast(AddressOf Runtime.BuiltInTPVs.ClosingFunctionBracket_TPV.IsSatisfiedBy))

                REM We now need to get the (If) Contents and the ElseContents
                Dim _Contents_And_ElseContents As Tuple(Of Runtime.Token(), Runtime.Token()) = GetContentsAndElseContents_(Me.Tokens_)

                Me.Contents_ = [StatementUtilities].GetStatementContentsFromTokens(_Contents_And_ElseContents.Item1).ToList()

                If _Contents_And_ElseContents.Item2 IsNot Nothing Then _
                    Me.ElseContents_ = [StatementUtilities].GetStatementContentsFromTokens(_Contents_And_ElseContents.Item2).ToList()

                LogLexingMessage("...Finished constructing an IfStatement Object with " & Me.Contents.Count.ToString() & " Child Instruction(s)")
            Catch _Ex As Exception : Throw New DSException("@IfStatement\ProcessTokensToInitialiseFields: " & _Ex.Message, _Ex) : End Try
        End Sub

        If there isn't an Else Clause in the Tokens, then [Item2] of the Tuple is Nothing (null)
        Protected Shared Function GetContentsAndElseContents_(ByVal _AllTokens As Runtime.Token()) As Tuple(Of Runtime.Token(), Runtime.Token()) ...
```

[Above] Class: IfStatement



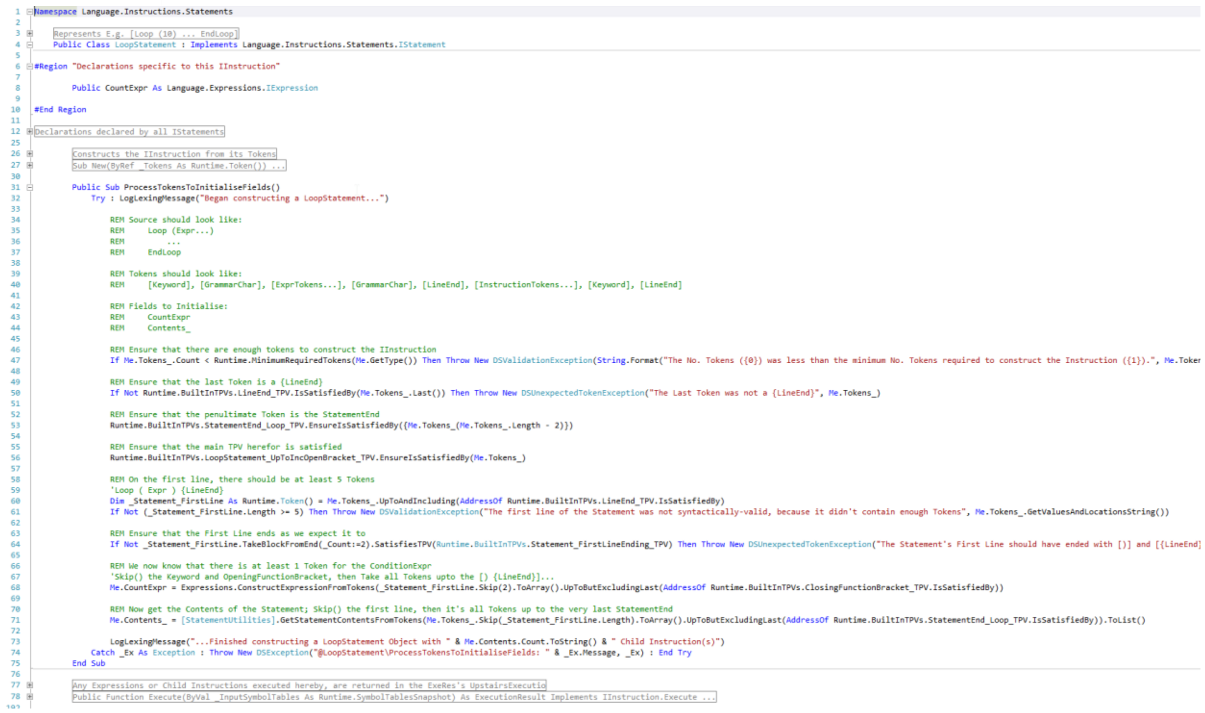
```
Namespace Language.Instructions.Statements

    Represents E.g. [While (True) ... EndWhile]
    Public Class WhileStatement : Implements Language.Instructions.Statements.IStatement

Declarations specific to this IInstruction

Declarations declared by all IStatements

        Constructs the IInstruction from its Tokens
        Sub New(ByRef _Tokens As Runtime.Token()) ...

        Public Sub ProcessTokensToInitialiseFields()
            Try : LogLexingMessage("Began constructing a WhileStatement...")

                REM Source should look like:
                REM     While (Expr...)
                REM         ...
                REM     EndWhile

                REM Tokens should look like:
                REM     [Keyword], [GrammarChar], [ExprTokens...], [GrammarChar], [LineEnd], [InstructionTokens...], [Keyword], [LineEnd]

                REM Fields to Initialise:
                REM     ConditionExpr
                REM     Contents_

                REM Ensure that there are enough tokens to construct the IInstruction
                If Me.Tokens_.Count < Runtime.MinimumRequiredTokens(Me.GetType()) Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) was less than the minimum No. Tokens required to construct the Ins

                REM Ensure that the last Token is a {LineEnd}
                If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

                REM Ensure that the penultimate Token is the StatementEnd
                Runtime.BuiltInTPVs.StatementEnd_While_TPV.EnsureIsSatisfiedBy({Me.Tokens_(Me.Tokens_.Length - 2)})

                REM Ensure that the main TPV herefor is satisfied
                Runtime.BuiltInTPVs.WhileStatement_UpToIncOpenBracket_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

                REM On the first line, there should be at least 5 Tokens
                'While ( Expr ) {LineEnd}
                Dim _Statement_FirstLine As Runtime.Token() = Me.Tokens_.UpToAndIncluding(AddressOf Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy)
                If Not (_Statement_FirstLine.Length >= 5) Then Throw New DSValidationException("The first line of the Statement was not syntactically-valid, because it didn't contain enough Tokens", Me.Tokens_.GetValuesAn

                REM Ensure that the First Line ends as we expect it to
                If Not _Statement_FirstLine.TakeBlockFromEnd(_Count:=2).SatisfiesTPV(Runtime.BuiltInTPVs.Statement_FirstLineEnding_TPV) Then Throw New DSUnexpectedTokenException("The Statement's First Line should have end

                REM We now know that there is at least 1 Token for the ConditionExpr
                'Skip() the Keyword and OpeningFunctionBracket, then Take all Tokens upto the [) {LineEnd}]...
                Me.ConditionExpr = Expressions.ConstructExpressionFromTokens(_Statement_FirstLine.Skip(2).ToArray().UpToButExcludingLast(AddressOf Runtime.BuiltInTPVs.ClosingFunctionBracket_TPV.IsSatisfiedBy))

                REM Now get the Contents of the Statement; Skip() the first line, then it's all Tokens up to the very last StatementEnd
                Me.Contents_ = [StatementUtilities].GetStatementContentsFromTokens(Me.Tokens_.Skip(_Statement_FirstLine.Length).ToArray().UpToButExcludingLast(AddressOf Runtime.BuiltInTPVs.StatementEnd_While_TPV.IsSatisfi

                LogLexingMessage("...Finished constructing a WhileStatement Object with " & Me.Contents.Count.ToString() & " Child Instruction(s)")
            Catch _Ex As Exception : Throw New DSException("@WhileStatement\ProcessTokensToInitialiseFields: " & _Ex.Message, _Ex) : End Try
        End Sub

        Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutio
        Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute ...

        Returns an XML Representation of the Instruction, with all Propities and Child Structres included
        Public Function GetProgramTreeNodeXML() As System.Xml.Linq.XElement Implements IInstruction.GetProgramTreeNodeXML ...

        Returns what the Instruction would have looked like in the Source, without an extra LineBreak on the
        Public Overrides Function ToString() As String ...
```

[Above] Class: WhileStatement



```
Namespace Language.Instructions.Statements

    Represents E.g. [Function &lt;DataType&gt; Identifier (Arguments...)]
    Public Class DSFunction : Implements Language.Instructions.Statements.IStatement, Runtime.SymbolTable.ISymbolTableValue

Declarations specific to this IInstruction

#Region "Declarations declared by all IStatements"

        Protected ReadOnly Tokens_ As Runtime.Token()
        Public Property ScopedVariables As Runtime.SymbolTable Implements IStatement.ScopedVariables

        Protected Contents_ As IInstruction()
        Public ReadOnly Property Contents As System.Collections.ObjectModel.ReadOnlyCollection(Of IInstruction) Implements IStatement.Contents ...

#End Region

        Constructs the IInstruction from its Tokens
        Sub New(ByRef _Tokens As Runtime.Token()) ...

        Public Sub ProcessTokensToInitialiseFields()
            Try : LogLexingMessage("Began constructing a DSFunction...")

                REM Source should look like:
                REM     Function <Void> Main () {LineEnd} ... EndFunction {LineEnd}
                REM     Function <Number> Main (<String@> _CLAs, <Number> _Age) {LineEnd} ... EndFunction {LineEnd}

                REM Tokens should look like:
                REM     [Keyword], [GrammarChar], [DataType], [GrammarChar], [Identifier], [GrammarChar], [GrammarChar], [LineEnd]
                REM     [Keyword], [GrammarChar], [DataType], [GrammarChar], [Identifier], [GrammarChar], ([GrammarChar], [DataType], [GrammarChar], [Identifier])... [LineEnd]

                REM Fields to Initialise:
                REM     Identifier
                REM     ReturnType
                REM     Parameters
                REM     Contents_

                REM Ensure that there are enough tokens to construct the IInstruction
                If Me.Tokens_.Count < Runtime.MinimumRequiredTokens(Me.GetType()) Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) was less than the minimum No. Tokens required to construct the Instruction ({1}).", Me.Token

                REM Ensure that the last Token is a {LineEnd}
                If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

                REM Ensure that the penultimate Token is the StatementEnd
                Runtime.BuiltInTPVs.StatementEnd_Function_TPV.EnsureIsSatisfiedBy({Me.Tokens_(Me.Tokens_.Length - 2)})

                REM Ensure that the main TPV herefor is satisfied
                Runtime.BuiltInTPVs.DSFunction_UpToIncIdentifier_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

                REM We now know that we have [Function <String> GetName]
                REM So set these Fields...

                REM On the first line, there should be at least 8 Tokens
                'Function < Number > Main ( ) {LineEnd}
                Dim _Statement_FirstLine As Runtime.Token() = Me.Tokens_.UpToAndIncluding(AddressOf Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy) 'UpTo the first {LineEnd}
                If Not (_Statement_FirstLine.Length >= 8) Then Throw New DSValidationException("The first line of the Statement was not syntactically-valid, because it didn't contain enough Tokens", Me.Tokens_.GetValuesAndLocationsString())

                REM Ensure that the First Line ends as we expect it to
                If Not _Statement_FirstLine.TakeBlockFromEnd(_Count:=2).SatisfiesTPV(Runtime.BuiltInTPVs.Statement_FirstLineEnding_TPV) Then Throw New DSUnexpectedTokenException("The Statement's First Line should have ended with [)] and [{LineEnd}

                REM The DataType should be Token(2)
                Me.ReturnType = Variables.GetDSVariableType_FromDataTypeString(Me.Tokens_(2).Value)

                REM The Identifier should be Token(4)
                Me.Identifier = Me.Tokens_(4).Value
```

[Above] Class: DSFunction



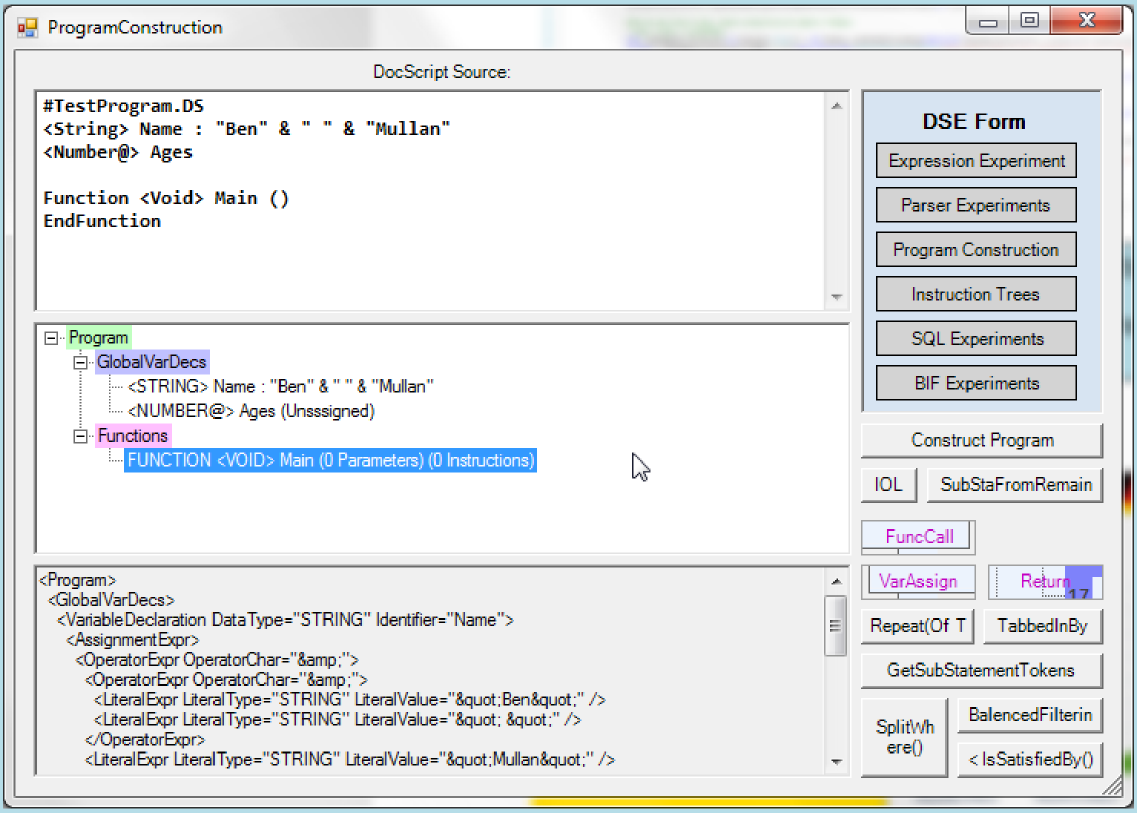
```
Namespace Language.Instructions.Statements

    Represents E.g. [Loop (10) ... EndLoop]
    Public Class LoopStatement : Implements Language.Instructions.Statements.IStatement

#Region "Declarations specific to this IInstruction"

        Public CountExpr As Language.Expressions.IExpression

#End Region

Declarations declared by all IStatements

        Constructs the IInstruction from its Tokens
        Sub New(ByRef _Tokens As Runtime.Token()) ...

        Public Sub ProcessTokensToInitialiseFields()
            Try : LogLexingMessage("Began constructing a LoopStatement...")

                REM Source should look like:
                REM     Loop (Expr...)
                REM         ...
                REM     EndLoop

                REM Tokens should look like:
                REM     [Keyword], [GrammarChar], [ExprTokens...], [GrammarChar], [LineEnd], [InstructionTokens...], [Keyword], [LineEnd]

                REM Fields to Initialise:
                REM     CountExpr
                REM     Contents_

                REM Ensure that there are enough tokens to construct the IInstruction
                If Me.Tokens_.Count < Runtime.MinimumRequiredTokens(Me.GetType()) Then Throw New DSValidationException(String.Format("The No. Tokens ({0}) was less than the minimum No. Tokens required to construct the Instruction ({1}).", Me.Toker

                REM Ensure that the last Token is a {LineEnd}
                If Not Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy(Me.Tokens_.Last()) Then Throw New DSUnexpectedTokenException("The Last Token was not a {LineEnd}", Me.Tokens_)

                REM Ensure that the penultimate Token is the StatementEnd
                Runtime.BuiltInTPVs.StatementEnd_Loop_TPV.EnsureIsSatisfiedBy({Me.Tokens_(Me.Tokens_.Length - 2)})

                REM Ensure that the main TPV herefor is satisfied
                Runtime.BuiltInTPVs.LoopStatement_UpToIncOpenBracket_TPV.EnsureIsSatisfiedBy(Me.Tokens_)

                REM On the first line, there should be at least 5 Tokens
                'Loop ( Expr ) {LineEnd}
                Dim _Statement_FirstLine As Runtime.Token() = Me.Tokens_.UpToAndIncluding(AddressOf Runtime.BuiltInTPVs.LineEnd_TPV.IsSatisfiedBy)
                If Not (_Statement_FirstLine.Length >= 5) Then Throw New DSValidationException("The first line of the Statement was not syntactically-valid, because it didn't contain enough Tokens", Me.Tokens_.GetValuesAndLocationsString())

                REM Ensure that the First Line ends as we expect it to
                If Not _Statement_FirstLine.TakeBlockFromEnd(_Count:=2).SatisfiesTPV(Runtime.BuiltInTPVs.Statement_FirstLineEnding_TPV) Then Throw New DSUnexpectedTokenException("The Statement's First Line should have ended with [)] and [{LineEnd]

                REM We now know that there is at least 1 Token for the ConditionExpr
                'Skip() the Keyword and OpeningFunctionBracket, then Take all Tokens upto the [) {LineEnd}]...
                Me.CountExpr = Expressions.ConstructExpressionFromTokens(_Statement_FirstLine.Skip(2).ToArray().UpToButExcludingLast(AddressOf Runtime.BuiltInTPVs.ClosingFunctionBracket_TPV.IsSatisfiedBy))

                REM Now get the Contents of the Statement; Skip() the first line, then it's all Tokens up to the very last StatementEnd
                Me.Contents_ = [StatementUtilities].GetStatementContentsFromTokens(Me.Tokens_.Skip(_Statement_FirstLine.Length).ToArray().UpToButExcludingLast(AddressOf Runtime.BuiltInTPVs.StatementEnd_Loop_TPV.IsSatisfiedBy)).ToList()

                LogLexingMessage("...Finished constructing a LoopStatement Object with " & Me.Contents.Count.ToString() & " Child Instruction(s)")
            Catch _Ex As Exception : Throw New DSException("@LoopStatement\ProcessTokensToInitialiseFields: " & _Ex.Message, _Ex) : End Try
        End Sub

        Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutio
        Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute ...
```

[Above] Class: LoopStatement

With the 8 different IInstruction-implementing Classes written, I ought to be able to construct a Program (Instruction) Tree, from some raw source:

Prototype: This point marks a milestone in the product, which is now capable of…

- Using a set of DocScript Tokens (piped in from the Parser) to construct a Program object. This is how that's implemented in the `DocScript.Experimentation` project:



```
#TestProgram.DS
<String> Name : "Ben" & " " & "Mullan"
<Number@> Ages

Function <Void> Main ()
EndFunction
```

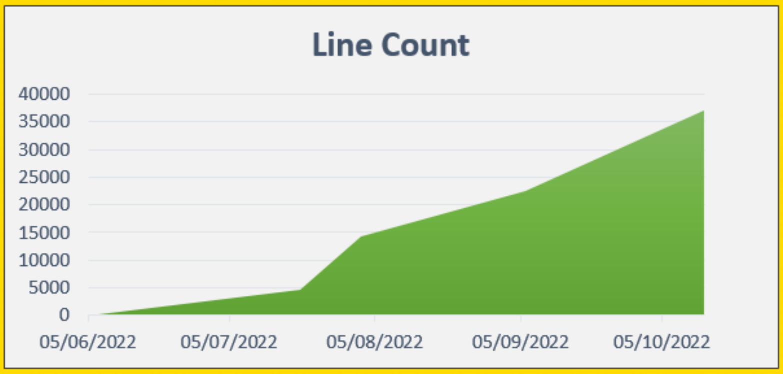
```
Program
    GlobalVarDecs
        <STRING> Name : "Ben" & " " & "Mullan"
        <NUMBER@> Ages (Unsssigned)
    Functions
        FUNCTION <VOID> Main (0 Parameters) (0 Instructions)
```

```
<Program>
 <GlobalVarDecs>
  <VariableDeclaration DataType="STRING" Identifier="Name">
   <AssignmentExpr>
    <OperatorExpr OperatorChar="&amp;">
     <OperatorExpr OperatorChar="&amp;">
      <LiteralExpr LiteralType="STRING" LiteralValue="&quot;Ben&quot;" />
      <LiteralExpr LiteralType="STRING" LiteralValue="&quot; &quot;" />
     </OperatorExpr>
     <LiteralExpr LiteralType="STRING" LiteralValue="&quot;Mullan&quot;" />
```

- Identifying the `DSFunction`s and Global-`VariableDeclaration`s of the Program.
- Ensuring that there is exactly one EntryPoint Function (Main) in the DocScript Program.

Progress Check

At this point in the development, I have written 36990 Lines of code...

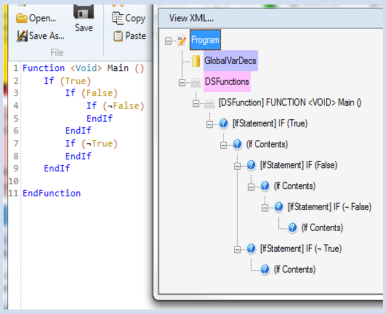


Testing Table: Does this component function in accordance with the stipulated criteria?
I will now test the Prototype Lexing System against criteria from the §Design, and some new criteria.

Does the Lexing System operate reliably, speedily, and consistently?
*Have all **edge-cases** been accounted-for?*

<table>
<tr><th>Test</th><th>☑ Passed?</th></tr>
<tr><td>A hierarchical Instruction Tree (<i>Program Tree</i>) is produced from a linear stream of Tokens.</td><td>Yes; see previous screenshots</td></tr>
<tr><td>The Lexer can correctly identify where one statement ends and another one ends.</td><td>Yes; see previous screenshots</td></tr>
<tr><td>Where Statements are nested, the lexer unambiguously determines which tokens correspond to which statement openings and closings.

For instance, this source...
<pre>
If (True)
    If (False)
        If (¬False)
        EndIf
    EndIf
    If (¬True)
    EndIf
EndIf
</pre>
...ought to produce 4 <code>IfStatement</code>s, in the structure of one parent containing two children, and one grandchild.</td><td>Yes;



<pre>
Function <Void> Main ()
    If (True)
        If (False)
            If (¬False)
            EndIf
        EndIf
        If (¬True)
        EndIf
    EndIf

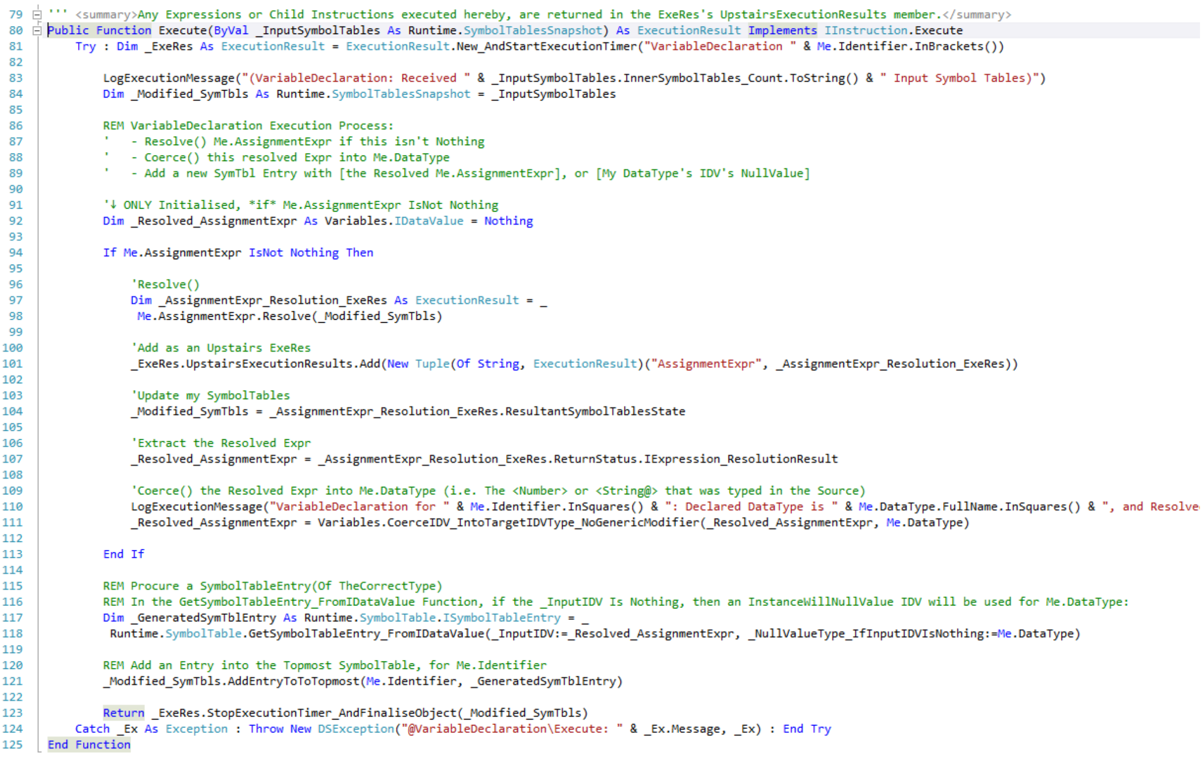
EndFunction
</pre>
<pre>
Program
    GlobalVarDecs
    DSFunctions
        [DSFunction] FUNCTION <VOID> Main ()
            [IfStatement] IF (True)
                (If Contents)
                    [IfStatement] IF (False)
                        (If Contents)
                            [IfStatement] IF (¬ False)
                                (If Contents)
                    [IfStatement] IF (¬ True)
                        (If Contents)
</pre>
(There are four If-Statements there)</td></tr>
</table>

Execution

The *third* of the three stages of interpretation is Execution, wherein the constructed `Program` Object is executed.

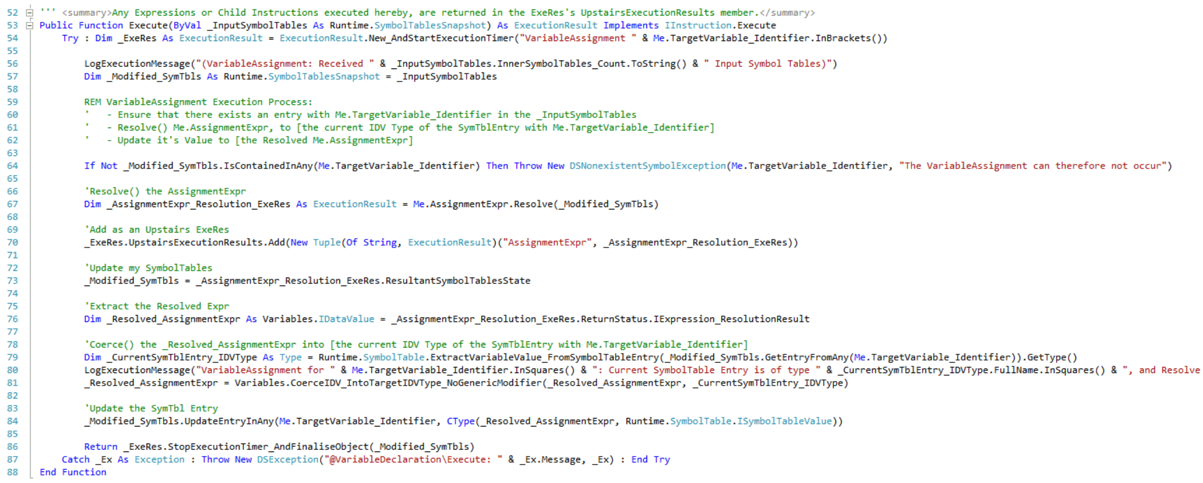
IInstruction.Execute()

The first set of methods to work on, are the implementations of `Execute(ByVal _InputSymTblsState As DocScript.Runtime.SymbolTablesSnapshot) As ExecutionResult`. They look like this:



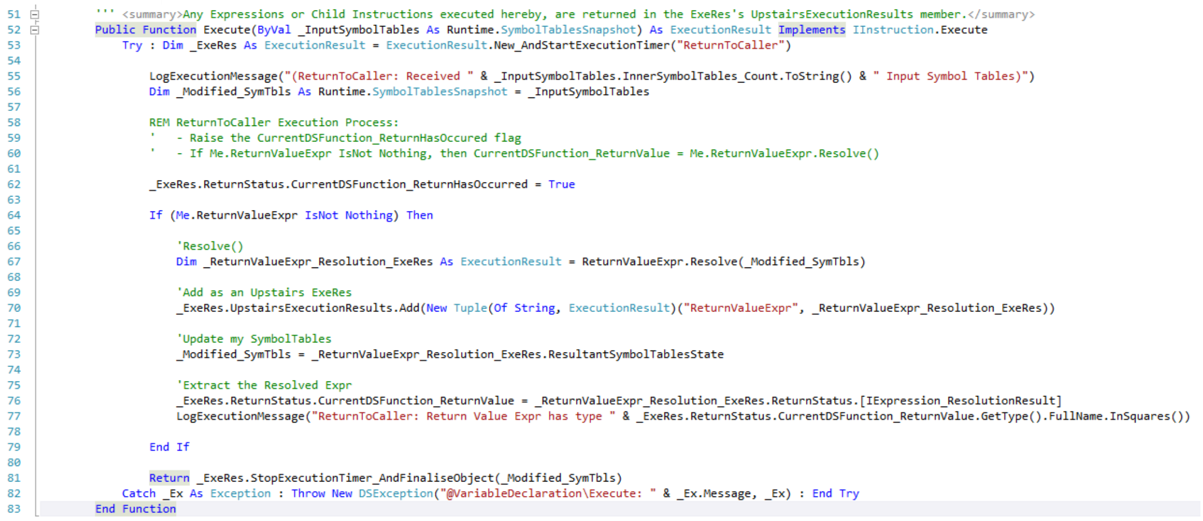
```
''' <summary>Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutionResults member.</summary>
Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute
    Try : Dim _ExeRes As ExecutionResult = ExecutionResult.New_AndStartExecutionTimer("VariableDeclaration " & Me.Identifier.InBrackets())

        LogExecutionMessage("(VariableDeclaration: Received " & _InputSymbolTables.InnerSymbolTables_Count.ToString() & " Input Symbol Tables)")
        Dim _Modified_SymTbls As Runtime.SymbolTablesSnapshot = _InputSymbolTables

        REM VariableDeclaration Execution Process:
        '   - Resolve() Me.AssignmentExpr if this isn't Nothing
        '   - Coerce() this resolved Expr into Me.DataType
        '   - Add a new SymTbl Entry with [the Resolved Me.AssignmentExpr], or [My DataType's IDV's NullValue]

        '↓ ONLY Initialised, *if* Me.AssignmentExpr IsNot Nothing
        Dim _Resolved_AssignmentExpr As Variables.IDataValue = Nothing

        If Me.AssignmentExpr IsNot Nothing Then

            'Resolve()
            Dim _AssignmentExpr_Resolution_ExeRes As ExecutionResult = _
             Me.AssignmentExpr.Resolve(_Modified_SymTbls)

            'Add as an Upstairs ExeRes
            _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, ExecutionResult)("AssignmentExpr", _AssignmentExpr_Resolution_ExeRes))

            'Update my SymbolTables
            _Modified_SymTbls = _AssignmentExpr_Resolution_ExeRes.ResultantSymbolTablesState

            'Extract the Resolved Expr
            _Resolved_AssignmentExpr = _AssignmentExpr_Resolution_ExeRes.ReturnStatus.IExpression_ResolutionResult

            'Coerce() the Resolved Expr into Me.DataType (i.e. The <Number> or <String@> that was typed in the Source)
            LogExecutionMessage("VariableDeclaration for " & Me.Identifier.InSquares() & ": Declared DataType is " & Me.DataType.FullName.InSquares() & ", and Resolve
            _Resolved_AssignmentExpr = Variables.CoerceIDV_IntoTargetIDVType_NoGenericModifier(_Resolved_AssignmentExpr, Me.DataType)

        End If

        REM Procure a SymbolTableEntry(Of TheCorrectType)
        REM In the GetSymbolTableEntry_FromIDataValue Function, if the _InputIDV Is Nothing, then an InstanceWillNullValue IDV will be used for Me.DataType:
        Dim _GeneratedSymTblEntry As Runtime.SymbolTable.ISymbolTableEntry = _
         Runtime.SymbolTable.GetSymbolTableEntry_FromIDataValue(_InputIDV:=_Resolved_AssignmentExpr, _NullValueType_IfInputIDVIsNothing:=Me.DataType)

        REM Add an Entry into the Topmost SymbolTable, for Me.Identifier
        _Modified_SymTbls.AddEntryToToTopmost(Me.Identifier, _GeneratedSymTblEntry)

        Return _ExeRes.StopExecutionTimer_AndFinaliseObject(_Modified_SymTbls)
    Catch _Ex As Exception : Throw New DSException("@VariableDeclaration\Execute: " & _Ex.Message, _Ex) : End Try
End Function
```

[Above] Method: VariableDeclaration.Execute()



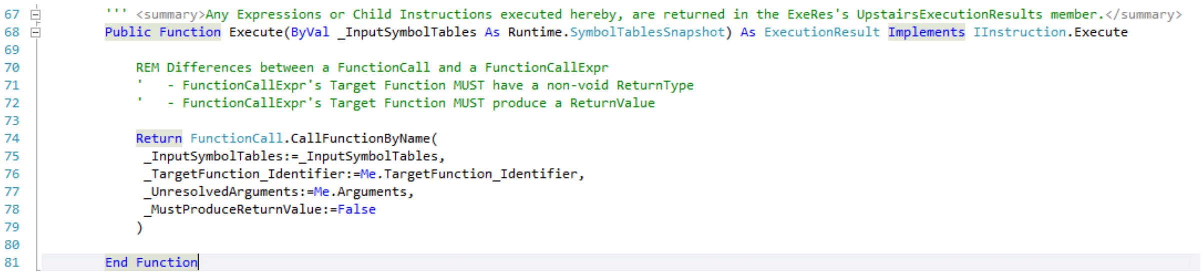
```
''' <summary>Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutionResults member.</summary>
Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute
    Try : Dim _ExeRes As ExecutionResult = ExecutionResult.New_AndStartExecutionTimer("VariableAssignment " & Me.TargetVariable_Identifier.InBrackets())

        LogExecutionMessage("(VariableAssignment: Received " & _InputSymbolTables.InnerSymbolTables_Count.ToString() & " Input Symbol Tables)")
        Dim _Modified_SymTbls As Runtime.SymbolTablesSnapshot = _InputSymbolTables

        REM VariableAssignment Execution Process:
        '   - Ensure that there exists an entry with Me.TargetVariable_Identifier in the _InputSymbolTables
        '   - Resolve() Me.AssignmentExpr, to [the current IDV Type of the SymTblEntry with Me.TargetVariable_Identifier]
        '   - Update it's Value to [the Resolved Me.AssignmentExpr]

        If Not _Modified_SymTbls.IsContainedInAny(Me.TargetVariable_Identifier) Then Throw New DSNonexistentSymbolException(Me.TargetVariable_Identifier, "The VariableAssignment can therefore not occur")

        'Resolve() the AssignmentExpr
        Dim _AssignmentExpr_Resolution_ExeRes As ExecutionResult = Me.AssignmentExpr.Resolve(_Modified_SymTbls)

        'Add as an Upstairs ExeRes
        _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, ExecutionResult)("AssignmentExpr", _AssignmentExpr_Resolution_ExeRes))

        'Update my SymbolTables
        _Modified_SymTbls = _AssignmentExpr_Resolution_ExeRes.ResultantSymbolTablesState

        'Extract the Resolved Expr
        Dim _Resolved_AssignmentExpr As Variables.IDataValue = _AssignmentExpr_Resolution_ExeRes.ReturnStatus.IExpression_ResolutionResult

        'Coerce() the _Resolved_AssignmentExpr into [the current IDV Type of the SymTblEntry with Me.TargetVariable_Identifier]
        Dim _CurrentSymTblEntry_IDVType As Type = Runtime.SymbolTable.ExtractVariableValue_FromSymbolTableEntry(_Modified_SymTbls.GetEntryFromAny(Me.TargetVariable_Identifier)).GetType()
        LogExecutionMessage("VariableAssignment for " & Me.TargetVariable_Identifier.InSquares() & ": Current SymbolTable Entry is of type " & _CurrentSymTblEntry_IDVType.FullName.InSquares() & ", and Resolve
        _Resolved_AssignmentExpr = Variables.CoerceIDV_IntoTargetIDVType_NoGenericModifier(_Resolved_AssignmentExpr, _CurrentSymTblEntry_IDVType)

        'Update the SymTbl Entry
        _Modified_SymTbls.UpdateEntryInAny(Me.TargetVariable_Identifier, CType(_Resolved_AssignmentExpr, Runtime.SymbolTable.ISymbolTableValue))

        Return _ExeRes.StopExecutionTimer_AndFinaliseObject(_Modified_SymTbls)
    Catch _Ex As Exception : Throw New DSException("@VariableDeclaration\Execute: " & _Ex.Message, _Ex) : End Try
End Function
```

[Above] Method: VariableAssignment.Execute()



```vb
''' <summary>Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutionResults member.</summary>
Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute
    Try : Dim _ExeRes As ExecutionResult = ExecutionResult.New_AndStartExecutionTimer("ReturnToCaller")

        LogExecutionMessage("(ReturnToCaller: Received " & _InputSymbolTables.InnerSymbolTables_Count.ToString() & " Input Symbol Tables)")
        Dim _Modified_SymTbls As Runtime.SymbolTablesSnapshot = _InputSymbolTables

        REM ReturnToCaller Execution Process:
        '   - Raise the CurrentDSFunction_ReturnHasOccured flag
        '   - If Me.ReturnValueExpr IsNot Nothing, then CurrentDSFunction_ReturnValue = Me.ReturnValueExpr.Resolve()

        _ExeRes.ReturnStatus.CurrentDSFunction_ReturnHasOccurred = True

        If (Me.ReturnValueExpr IsNot Nothing) Then

            'Resolve()
            Dim _ReturnValueExpr_Resolution_ExeRes As ExecutionResult = ReturnValueExpr.Resolve(_Modified_SymTbls)

            'Add as an Upstairs ExeRes
            _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, ExecutionResult)("ReturnValueExpr", _ReturnValueExpr_Resolution_ExeRes))

            'Update my SymbolTables
            _Modified_SymTbls = _ReturnValueExpr_Resolution_ExeRes.ResultantSymbolTablesState

            'Extract the Resolved Expr
            _ExeRes.ReturnStatus.CurrentDSFunction_ReturnValue = _ReturnValueExpr_Resolution_ExeRes.ReturnStatus.[IExpression_ResolutionResult]
            LogExecutionMessage("ReturnToCaller: Return Value Expr has type " & _ExeRes.ReturnStatus.CurrentDSFunction_ReturnValue.GetType().FullName.InSquares())

        End If

        Return _ExeRes.StopExecutionTimer_AndFinaliseObject(_Modified_SymTbls)
    Catch _Ex As Exception : Throw New DSException("@VariableDeclaration\Execute: " & _Ex.Message, _Ex) : End Try
End Function
```

[Above] Method: ReturnToCaller.Execute()



```vb
''' <summary>Any Expressions or Child Instructions executed hereby, are returned in the ExeRes's UpstairsExecutionResults member.</summary>
Public Function Execute(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As ExecutionResult Implements IInstruction.Execute

    REM Differences between a FunctionCall and a FunctionCallExpr
    '   - FunctionCallExpr's Target Function MUST have a non-void ReturnType
    '   - FunctionCallExpr's Target Function MUST produce a ReturnValue

    Return FunctionCall.CallFunctionByName(
     _InputSymbolTables:=_InputSymbolTables,
     _TargetFunction_Identifier:=Me.TargetFunction_Identifier,
     _UnresolvedArguments:=Me.Arguments,
     _MustProduceReturnValue:=False
    )

End Function
```

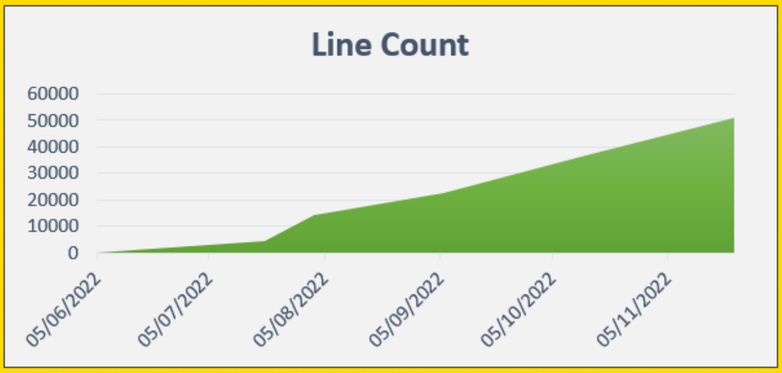
[Above] Method: FunctionCall.Execute()

The last of these `.Execute()` methods, in `FunctionCall`, piggy-backs off of the `CallFunctionByName()` Method (sounds oddly Win32-like, dosen't it), which provides the following advantages:

- The same code dosen't have to be re-written for FunctionCallExpr
- The `.Execute()` methods don't have to care about the difference between calling a DSFunction, and a Built-in Function; this is all handled by `CallFunctionByName()`.

Progress Check

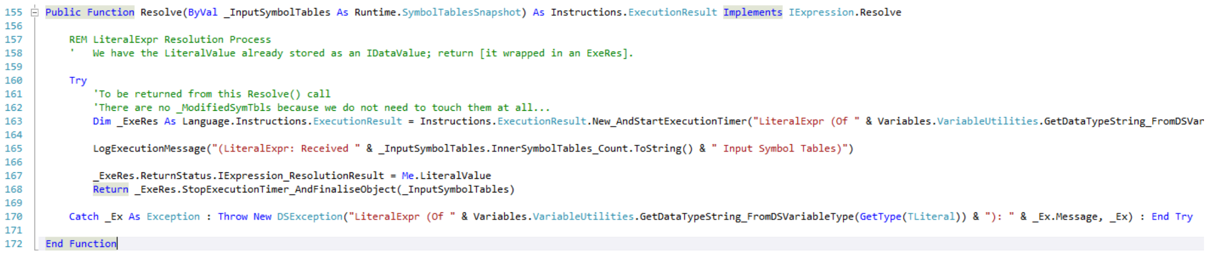
At this point in the development,
I have written 50996 Lines of
code...



IExpression.Resolve()

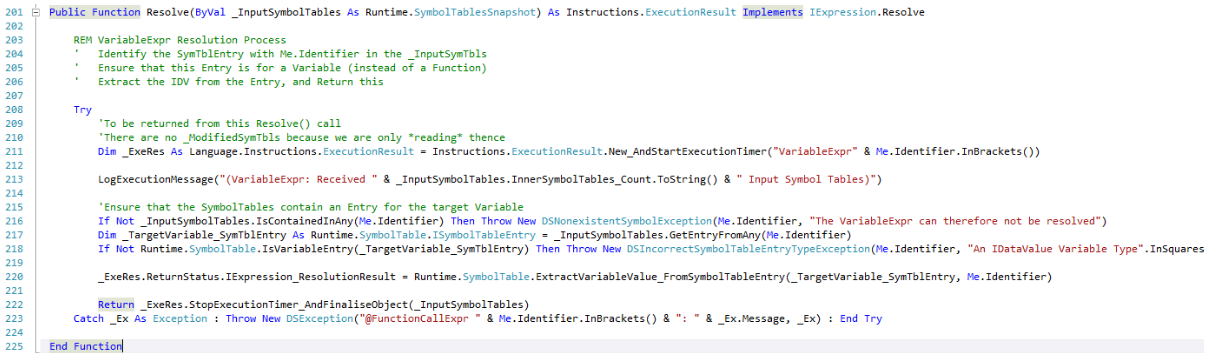
The IExpression-implementing classes do not have a `.Execute()` method, because they are not really Instructions which can be executed on their own. In DocScript, it is not valid to have an Expression sitting by itself on a line. It must be a component of an Instruction, such as a VariableAssignment or FunctionCall.

Therefore, the `.Resolve()` method is used instead. It still returns an ExecutionResult; `Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult`. I implemented the Resolve() methods thusly:



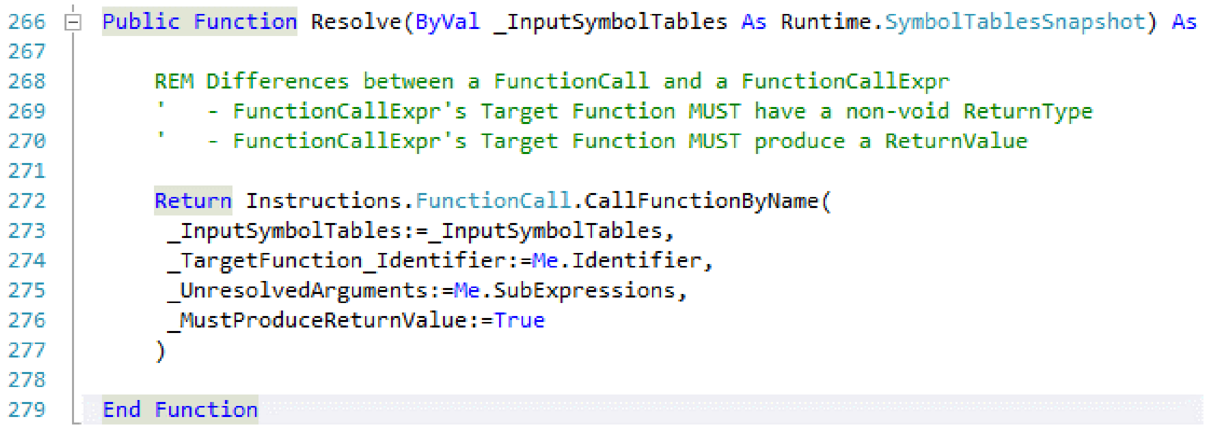
```
Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult Implements IExpression.Resolve

    REM LiteralExpr Resolution Process
    '   We have the LiteralValue already stored as an IDataValue; return [it wrapped in an ExeRes].

    Try
        'To be returned from this Resolve() call
        'There are no _ModifiedSymTbls because we do not need to touch them at all...
        Dim _ExeRes As Language.Instructions.ExecutionResult = Instructions.ExecutionResult.New_AndStartExecutionTimer("LiteralExpr (Of " & Variables.VariableUtilities.GetDataTypeString_FromDSVar

        LogExecutionMessage("(LiteralExpr: Received " & _InputSymbolTables.InnerSymbolTables_Count.ToString() & " Input Symbol Tables)")

        _ExeRes.ReturnStatus.IExpression_ResolutionResult = Me.LiteralValue
        Return _ExeRes.StopExecutionTimer_AndFinaliseObject(_InputSymbolTables)

    Catch _Ex As Exception : Throw New DSException("LiteralExpr (Of " & Variables.VariableUtilities.GetDataTypeString_FromDSVariableType(GetType(TLiteral)) & "): " & _Ex.Message, _Ex) : End Try

End Function
```

[Above] Method: LiteralExpr.Resolve()



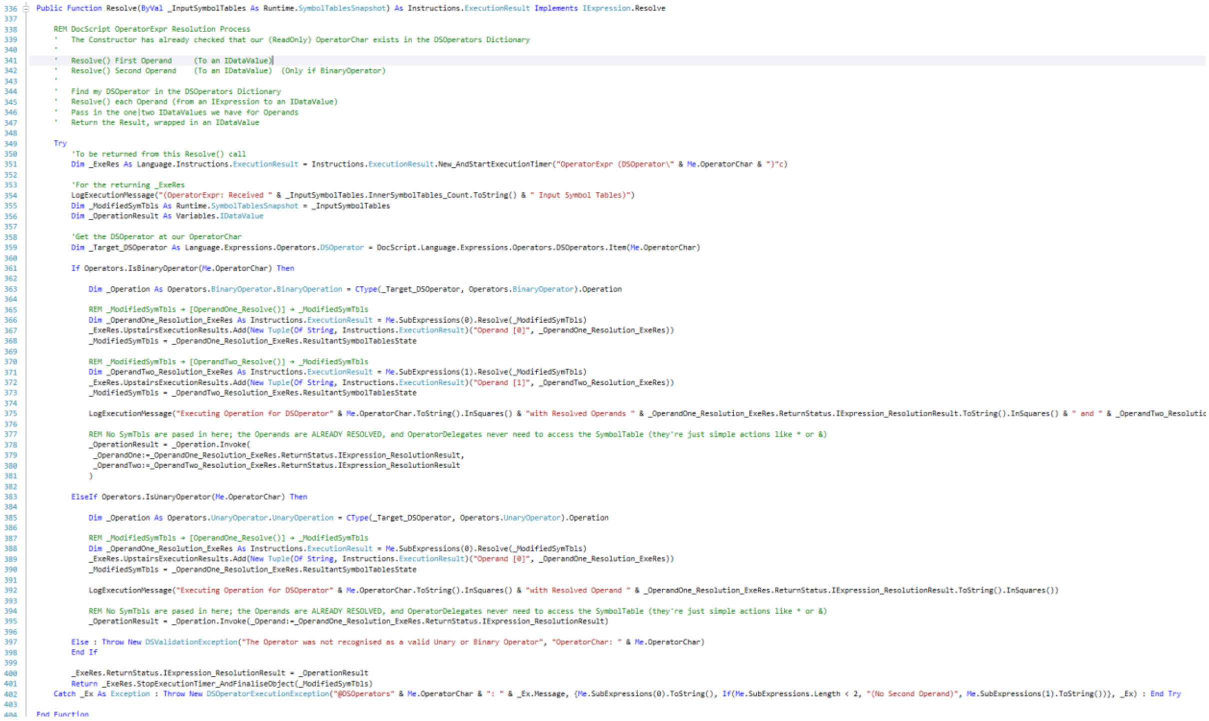
```
Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult Implements IExpression.Resolve

    REM VariableExpr Resolution Process
    '   Identify the SymTblEntry with Me.Identifier in the _InputSymTbls
    '   Ensure that this Entry is for a Variable (instead of a Function)
    '   Extract the IDV from the Entry, and Return this

    Try
        'To be returned from this Resolve() call
        'There are no _ModifiedSymTbls because we are only *reading* thence
        Dim _ExeRes As Language.Instructions.ExecutionResult = Instructions.ExecutionResult.New_AndStartExecutionTimer("VariableExpr" & Me.Identifier.InBrackets())

        LogExecutionMessage("(VariableExpr: Received " & _InputSymbolTables.InnerSymbolTables_Count.ToString() & " Input Symbol Tables)")

        'Ensure that the SymbolTables contain an Entry for the target Variable
        If Not _InputSymbolTables.IsContainedInAny(Me.Identifier) Then Throw New DSNonexistentSymbolException(Me.Identifier, "The VariableExpr can therefore not be resolved")
        Dim _TargetVariable_SymTblEntry As Runtime.SymbolTable.ISymbolTableEntry = _InputSymbolTables.GetEntryFromAny(Me.Identifier)
        If Not Runtime.SymbolTable.IsVariableEntry(_TargetVariable_SymTblEntry) Then Throw New DSIncorrectSymbolTableEntryTypeException(Me.Identifier, "An IDataValue Variable Type".InSquares

        _ExeRes.ReturnStatus.IExpression_ResolutionResult = Runtime.SymbolTable.ExtractVariableValue_FromSymbolTableEntry(_TargetVariable_SymTblEntry, Me.Identifier)

        Return _ExeRes.StopExecutionTimer_AndFinaliseObject(_InputSymbolTables)
    Catch _Ex As Exception : Throw New DSException("@FunctionCallExpr " & Me.Identifier.InBrackets() & ": " & _Ex.Message, _Ex) : End Try

End Function
```

[Above] Method: VariableExpr.Resolve()



```
Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As

    REM Differences between a FunctionCall and a FunctionCallExpr
    '   - FunctionCallExpr's Target Function MUST have a non-void ReturnType
    '   - FunctionCallExpr's Target Function MUST produce a ReturnValue

    Return Instructions.FunctionCall.CallFunctionByName(
     _InputSymbolTables:=_InputSymbolTables,
     _TargetFunction_Identifier:=Me.Identifier,
     _UnresolvedArguments:=Me.SubExpressions,
     _MustProduceReturnValue:=True
    )

End Function
```

[Above] Method: FunctionCallExpr.Resolve()



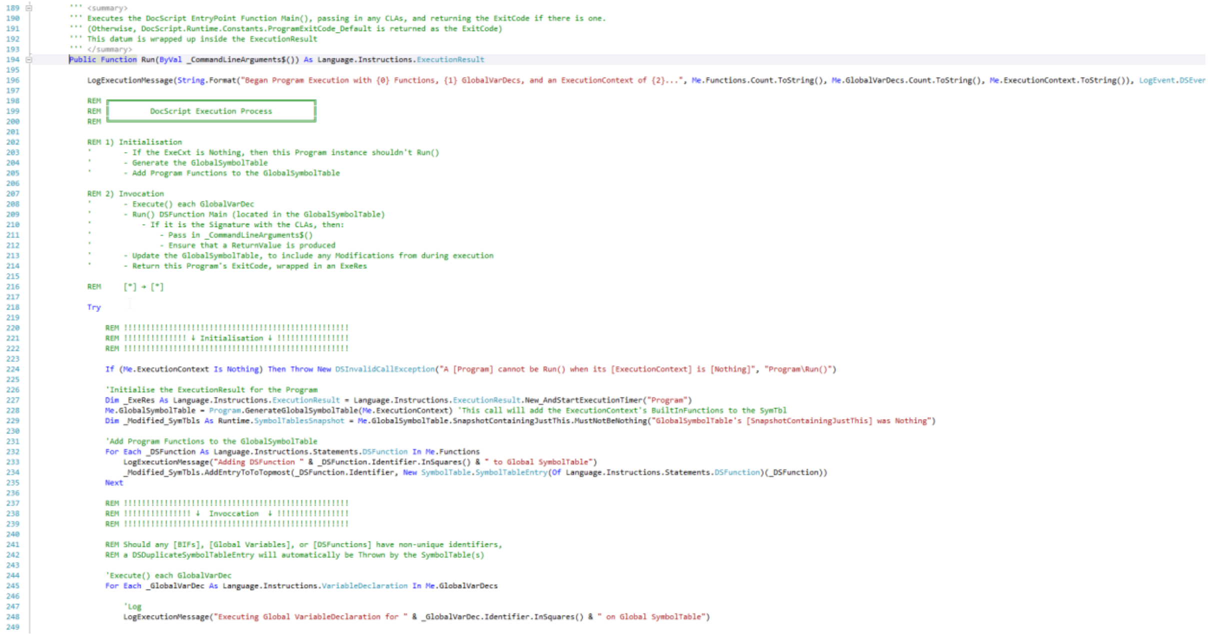
```
Public Function Resolve(ByVal _InputSymbolTables As Runtime.SymbolTablesSnapshot) As Instructions.ExecutionResult Implements IExpression.Resolve

    REM DocScript OperatorExpr Resolution Process
    '   The Constructor has already checked that our (ReadOnly) OperatorChar exists in the DSOperators Dictionary
    '
    '   Resolve() First Operand     (To an IDataValue)
    '   Resolve() Second Operand    (To an IDataValue)  (Only if BinaryOperator)
    '
    '   Find my DSOperator in the DSOperators Dictionary
    '   Resolve() each Operand (from an IExpression to an IDataValue)
    '   Pass in the one|two IDataValues we have for Operands
    '   Return the Result, wrapped in an IDataValue

    Try
        'To be returned from this Resolve() call
        Dim _ExeRes As Language.Instructions.ExecutionResult = Instructions.ExecutionResult.New_AndStartExecutionTimer("OperatorExpr (DSOperator\" & Me.OperatorChar & ")"c)

        'For the returning _ExeRes
        LogExecutionMessage("(OperatorExpr: Received " & _InputSymbolTables.InnerSymbolTables_Count.ToString() & " Input Symbol Tables)")
        Dim _ModifiedSymTbls As Runtime.SymbolTablesSnapshot = _InputSymbolTables
        Dim _OperationResult As Variables.IDataValue

        'Get the DSOperator at our OperatorChar
        Dim _Target_DSOperator As Language.Expressions.Operators.DSOperator = DocScript.Language.Expressions.Operators.DSOperators.Item(Me.OperatorChar)

        If Operators.IsBinaryOperator(Me.OperatorChar) Then

            Dim _Operation As Operators.BinaryOperator.BinaryOperation = CType(_Target_DSOperator, Operators.BinaryOperator).Operation

            REM _ModifiedSymTbls → [OperandOne_Resolve()] → _ModifiedSymTbls
            Dim _OperandOne_Resolution_ExeRes As Instructions.ExecutionResult = Me.SubExpressions(0).Resolve(_ModifiedSymTbls)
            _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, Instructions.ExecutionResult)("Operand [0]", _OperandOne_Resolution_ExeRes))
            _ModifiedSymTbls = _OperandOne_Resolution_ExeRes.ResultantSymbolTablesState

            REM _ModifiedSymTbls → [OperandTwo_Resolve()] → _ModifiedSymTbls
            Dim _OperandTwo_Resolution_ExeRes As Instructions.ExecutionResult = Me.SubExpressions(1).Resolve(_ModifiedSymTbls)
            _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, Instructions.ExecutionResult)("Operand [1]", _OperandTwo_Resolution_ExeRes))
            _ModifiedSymTbls = _OperandTwo_Resolution_ExeRes.ResultantSymbolTablesState

            LogExecutionMessage("Executing Operation for DSOperator" & Me.OperatorChar.ToString().InSquares() & "with Resolved Operands " & _OperandOne_Resolution_ExeRes.ReturnStatus.IExpression_ResolutionResult.ToString().InSquares() & " and " & _OperandTwo_Resolutic

            REM No SymTbls are pased in here; the Operands are ALREADY RESOLVED, and OperatorDelegates never need to access the SymbolTable (they're just simple actions like * or &)
            _OperationResult = _Operation.Invoke(
             _OperandOne:=_OperandOne_Resolution_ExeRes.ReturnStatus.IExpression_ResolutionResult,
             _OperandTwo:=_OperandTwo_Resolution_ExeRes.ReturnStatus.IExpression_ResolutionResult
            )

        ElseIf Operators.IsUnaryOperator(Me.OperatorChar) Then

            Dim _Operation As Operators.UnaryOperator.UnaryOperation = CType(_Target_DSOperator, Operators.UnaryOperator).Operation

            REM _ModifiedSymTbls → [OperandOne_Resolve()] → _ModifiedSymTbls
            Dim _OperandOne_Resolution_ExeRes As Instructions.ExecutionResult = Me.SubExpressions(0).Resolve(_ModifiedSymTbls)
            _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, Instructions.ExecutionResult)("Operand [0]", _OperandOne_Resolution_ExeRes))
            _ModifiedSymTbls = _OperandOne_Resolution_ExeRes.ResultantSymbolTablesState

            LogExecutionMessage("Executing Operation for DSOperator" & Me.OperatorChar.ToString().InSquares() & "with Resolved Operand " & _OperandOne_Resolution_ExeRes.ReturnStatus.IExpression_ResolutionResult.ToString().InSquares())

            REM No SymTbls are pased in here; the Operands are ALREADY RESOLVED, and OperatorDelegates never need to access the SymbolTable (they're just simple actions like * or &)
            _OperationResult = _Operation.Invoke(_Operand:=_OperandOne_Resolution_ExeRes.ReturnStatus.IExpression_ResolutionResult)

        Else : Throw New DSValidationException("The Operator was not recognised as a valid Unary or Binary Operator", "OperatorChar: " & Me.OperatorChar)
        End If

        _ExeRes.ReturnStatus.IExpression_ResolutionResult = _OperationResult
        Return _ExeRes.StopExecutionTimer_AndFinaliseObject(_ModifiedSymTbls)
    Catch _Ex As Exception : Throw New DSOperatorExecutionException("@DSOperators" & Me.OperatorChar & ": " & _Ex.Message, {Me.SubExpressions(0).ToString(), If(Me.SubExpressions.Length < 2, "(No Second Operand)", Me.SubExpressions(1).ToString())}, _Ex) : End Try

End Function
```

[Above] Method: OperatorExpr.Resolve()

Program.Run()

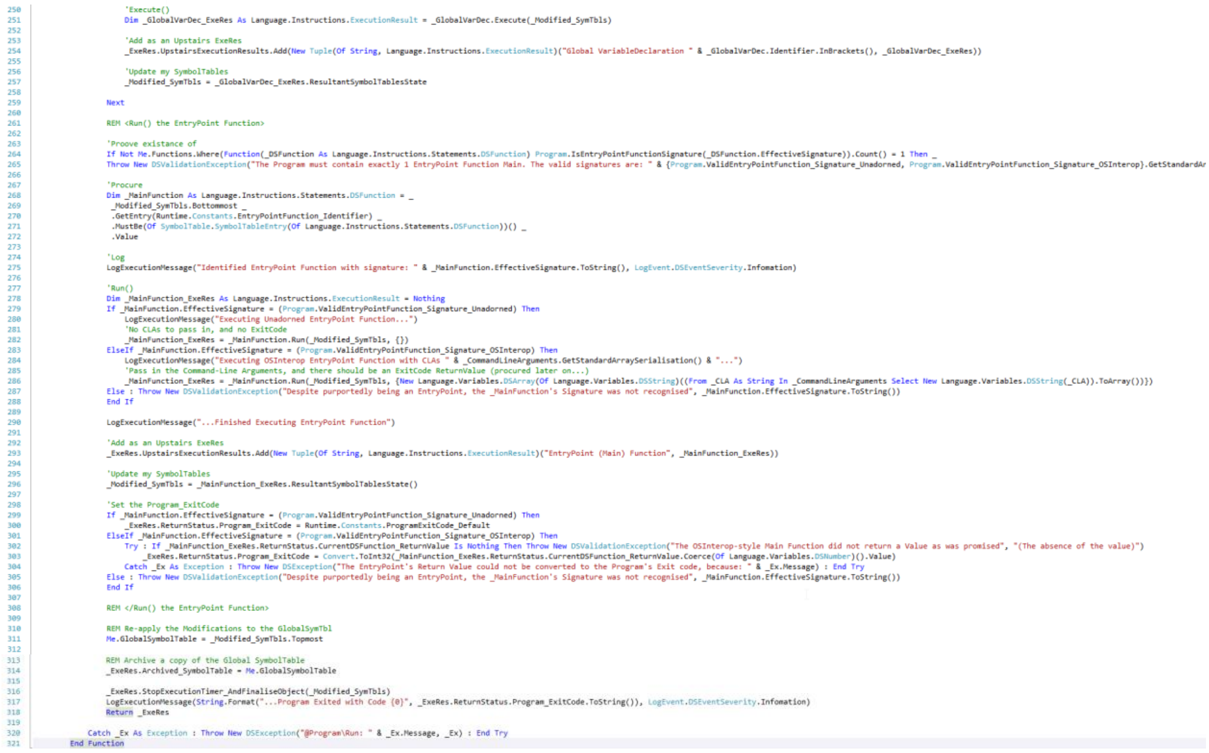
Finally – with all the components *inside* a DocScript `Program` having had *their* `Execute()`, `Resolve()`, and `Run()` methods implemented – I can now write the function that actually runs an entire DocScript Program.

In essence, all it needs to do is to Execute each of the Global `VariableDeclaration`s, and then `Run()` the Main (EntryPoint) Function…



```
    ''' <summary>
    ''' Executes the DocScript EntryPoint Function Main(), passing in any CLAs, and returning the ExitCode if there is one.
    ''' (Otherwise, DocScript.Runtime.Constants.ProgramExitCode_Default is returned as the ExitCode)
    ''' This datum is wrapped up inside the ExecutionResult
    ''' </summary>
    Public Function Run(ByVal _CommandLineArguments$()) As Language.Instructions.ExecutionResult

        LogExecutionMessage(String.Format("Began Program Execution with {0} Functions, {1} GlobalVarDecs, and an ExecutionContext of {2}...", Me.Functions.Count.ToString(), Me.GlobalVarDecs.Count.ToString(), Me.ExecutionContext.ToString()), LogEvent.DSEver

        REM
        REM         DocScript Execution Process
        REM

        REM 1) Initialisation
        '       - If the ExeCxt is Nothing, then this Program instance shouldn't Run()
        '       - Generate the GlobalSymbolTable
        '       - Add Program Functions to the GlobalSymbolTable

        REM 2) Invocation
        '       - Execute() each GlobalVarDec
        '       - Run() DSFunction Main (located in the GlobalSymbolTable)
        '           - If it is the Signature with the CLAs, then:
        '               - Pass in _CommandLineArguments$()
        '               - Ensure that a ReturnValue is produced
        '       - Update the GlobalSymbolTable, to include any Modifications from during execution
        '       - Return this Program's ExitCode, wrapped in an ExeRes

        REM    [*] → [*]

        Try

            REM !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            REM !!!!!!!!!!!!!! ↓ Initialisation ↓ !!!!!!!!!!!!!!!
            REM !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

            If (Me.ExecutionContext Is Nothing) Then Throw New DSInvalidCallException("A [Program] cannot be Run() when its [ExecutionContext] is [Nothing]", "Program\Run()")

            'Initialise the ExecutionResult for the Program
            Dim _ExeRes As Language.Instructions.ExecutionResult = Language.Instructions.ExecutionResult.New_AndStartExecutionTimer("Program")
            Me.GlobalSymbolTable = Program.GenerateGlobalSymbolTable(Me.ExecutionContext) 'This call will add the ExecutionContext's BuiltInFunctions to the SymTbl
            Dim _Modified_SymTbls As Runtime.SymbolTablesSnapshot = Me.GlobalSymbolTable.SnapshotContainingJustThis.MustNotBeNothing("GlobalSymbolTable's [SnapshotContainingJustThis] was Nothing")

            'Add Program Functions to the GlobalSymbolTable
            For Each _DSFunction As Language.Instructions.Statements.DSFunction In Me.Functions
                LogExecutionMessage("Adding DSFunction " & _DSFunction.Identifier.InSquares() & " to Global SymbolTable")
                _Modified_SymTbls.AddEntryToToTopmost(_DSFunction.Identifier, New SymbolTable.SymbolTableEntry(Of Language.Instructions.Statements.DSFunction)(_DSFunction))
            Next

            REM !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            REM !!!!!!!!!!!!!!! ↓  Invoccation  ↓ !!!!!!!!!!!!!!!
            REM !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

            REM Should any [BIFs], [Global Variables], or [DSFunctions] have non-unique identifiers,
            REM a DSDuplicateSymbolTableEntry will automatically be Thrown by the SymbolTable(s)

            'Execute() each GlobalVarDec
            For Each _GlobalVarDec As Language.Instructions.VariableDeclaration In Me.GlobalVarDecs

                'Log
                LogExecutionMessage("Executing Global VariableDeclaration for " & _GlobalVarDec.Identifier.InSquares() & " on Global SymbolTable")

```

[Above] Method: Program.Run() – *Part 1*

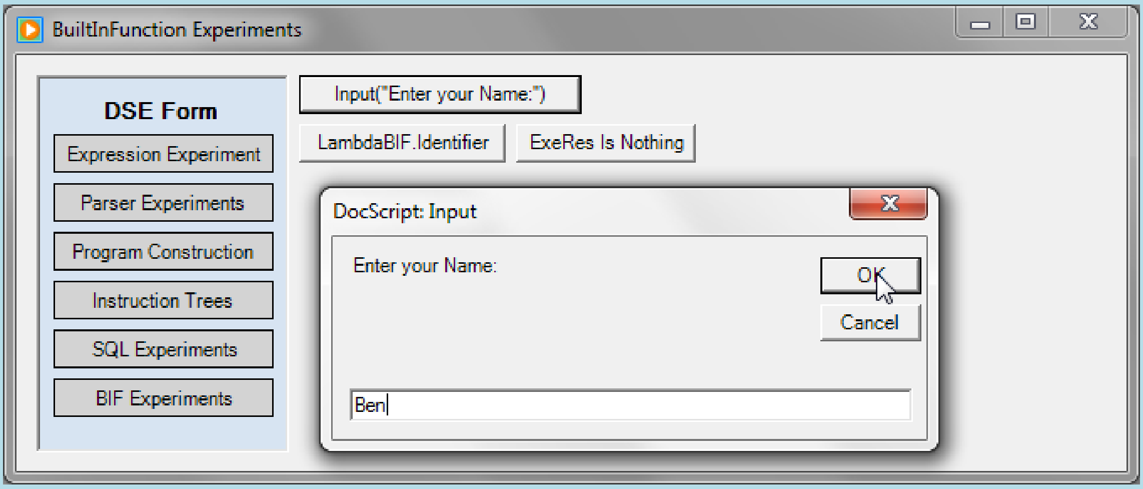


```
            'Execute()
            Dim _GlobalVarDec_ExeRes As Language.Instructions.ExecutionResult = _GlobalVarDec.Execute(_Modified_SymTbls)

            'Add as an Upstairs ExeRes
            _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, Language.Instructions.ExecutionResult)("Global VariableDeclaration " & _GlobalVarDec.Identifier.InBrackets(), _GlobalVarDec_ExeRes))

            'Update my SymbolTables
            _Modified_SymTbls = _GlobalVarDec_ExeRes.ResultantSymbolTablesState

        Next

        REM <Run() the EntryPoint Function>

        'Proove existance of
        If Not Me.Functions.Where(Function(_DSFunction As Language.Instructions.Statements.DSFunction) Program.IsEntryPointFunctionSignature(_DSFunction.EffectiveSignature)).Count() = 1 Then _
        Throw New DSValidationException("The Program must contain exactly 1 EntryPoint Function Main. The valid signatures are: " & {Program.ValidEntryPointFunction_Signature_Unadorned, Program.ValidEntryPointFunction_Signature_OSInterop}.GetStandardAr

        'Procure
        Dim _MainFunction As Language.Instructions.Statements.DSFunction = _
         _Modified_SymTbls.Bottommost _
         .GetEntry(Runtime.Constants.EntryPointFunction_Identifier) _
         .MustBe(Of SymbolTable.SymbolTableEntry(Of Language.Instructions.Statements.DSFunction))() _
         .Value

        'Log
        LogExecutionMessage("Identified EntryPoint Function with signature: " & _MainFunction.EffectiveSignature.ToString(), LogEvent.DSEventSeverity.Infomation)

        'Run()
        Dim _MainFunction_ExeRes As Language.Instructions.ExecutionResult = Nothing
        If _MainFunction.EffectiveSignature = (Program.ValidEntryPointFunction_Signature_Unadorned) Then
            LogExecutionMessage("Executing Unadorned EntryPoint Function...")
            'No CLAs to pass in, and no ExitCode
            _MainFunction_ExeRes = _MainFunction.Run(_Modified_SymTbls, {})
        ElseIf _MainFunction.EffectiveSignature = (Program.ValidEntryPointFunction_Signature_OSInterop) Then
            LogExecutionMessage("Executing OSInterop EntryPoint Function with CLAs " & _CommandLineArguments.GetStandardArraySerialisation() & "...")
            'Pass in the Command-Line Arguments, and there should be an ExitCode ReturnValue (procured later on...)
            _MainFunction_ExeRes = _MainFunction.Run(_Modified_SymTbls, {New Language.Variables.DSArray(Of Language.Variables.DSString)((From _CLA As String In _CommandLineArguments Select New Language.Variables.DSString(_CLA)).ToArray())})
        Else : Throw New DSValidationException("Despite purportedly being an EntryPoint, the _MainFunction's Signature was not recognised", _MainFunction.EffectiveSignature.ToString())
        End If

        LogExecutionMessage("...Finished Executing EntryPoint Function")

        'Add as an Upstairs ExeRes
        _ExeRes.UpstairsExecutionResults.Add(New Tuple(Of String, Language.Instructions.ExecutionResult)("EntryPoint (Main) Function", _MainFunction_ExeRes))

        'Update my SymbolTables
        _Modified_SymTbls = _MainFunction_ExeRes.ResultantSymbolTablesState()

        'Set the Program_ExitCode
        If _MainFunction.EffectiveSignature = (Program.ValidEntryPointFunction_Signature_Unadorned) Then
          _ExeRes.ReturnStatus.Program_ExitCode = Runtime.Constants.ProgramExitCode_Default
        ElseIf _MainFunction.EffectiveSignature = (Program.ValidEntryPointFunction_Signature_OSInterop) Then
            Try : If _MainFunction_ExeRes.ReturnStatus.CurrentDSFunction_ReturnValue Is Nothing Then Throw New DSValidationException("The OSInterop-style Main Function did not return a Value as was promised", "(The absence of the value)")
                _ExeRes.ReturnStatus.Program_ExitCode = Convert.ToInt32(_MainFunction_ExeRes.ReturnStatus.CurrentDSFunction_ReturnValue.Coerce(Of Language.Variables.DSNumber)().Value)
            Catch _Ex As Exception : Throw New DSException("The EntryPoint's Return Value could not be converted to the Program's Exit code, because: " & _Ex.Message) : End Try
        Else : Throw New DSValidationException("Despite purportedly being an EntryPoint, the _MainFunction's Signature was not recognised", _MainFunction.EffectiveSignature.ToString())
        End If

        REM </Run() the EntryPoint Function>

        REM Re-apply the Modifications to the GlobalSymTbl
        Me.GlobalSymbolTable = _Modified_SymTbls.Topmost

        REM Archive a copy of the Global SymbolTable
        _ExeRes.Archived_SymbolTable = Me.GlobalSymbolTable

        _ExeRes.StopExecutionTimer_AndFinaliseObject(_Modified_SymTbls)
        LogExecutionMessage(String.Format("...Program Exited with Code {0}", _ExeRes.ReturnStatus.Program_ExitCode.ToString()), LogEvent.DSEventSeverity.Infomation)
        Return _ExeRes

    Catch _Ex As Exception : Throw New DSException("@Program\Run: " & _Ex.Message, _Ex) : End Try
End Function
```

[Above] Method: Program.Run() – *Part 2*

Prototype: This point marks a milestone in the product, which is now capable of…

- Executing a Program Object (piped in from the lexing system), including executing each Global-`VariableDeclaration`, and then the `DSFunction` Main.
- Creating a Global `SymbolTable` at the beginning of execution, and passing the stack of symbol tables to each statement in the tree of the program, as it is executed.
- Distinguishing between `DSFunction`s defined in the DocScript Program, and `BuiltInFunction`s, defined by DocScript Runtime. This is the Experimentation Form I used to test the BIFs:



DocScript: Input

Enter your Name:

Ben

Progress Recap: Where am I in the development plan? *[Review]*

- **Done**: I have just written the DocScript Library DLL, which contains the logic required to interpret DocScript `Program`s.
- **Next**: I will implement this DLL into the first of three implementations, as was described in detail within §Design. This first DocScript Interpreter implementation is a command-line one.

[Stage 2]
Command-line Interpreter (**DSCLI.EXE**) Development

This is getting very exciting! I am nearly able to run the first ever DocScript Program. Before the logic in the DLL can be used to do this, however, I must in fact ***implement*** the logic into something that uses it.

> *Analogically speaking, what I have just made is like a **cassette tape**. Now – in order to enjoy the must on the tape – I must create the **boombox** that plays it.*

Writing the CLA-Manager

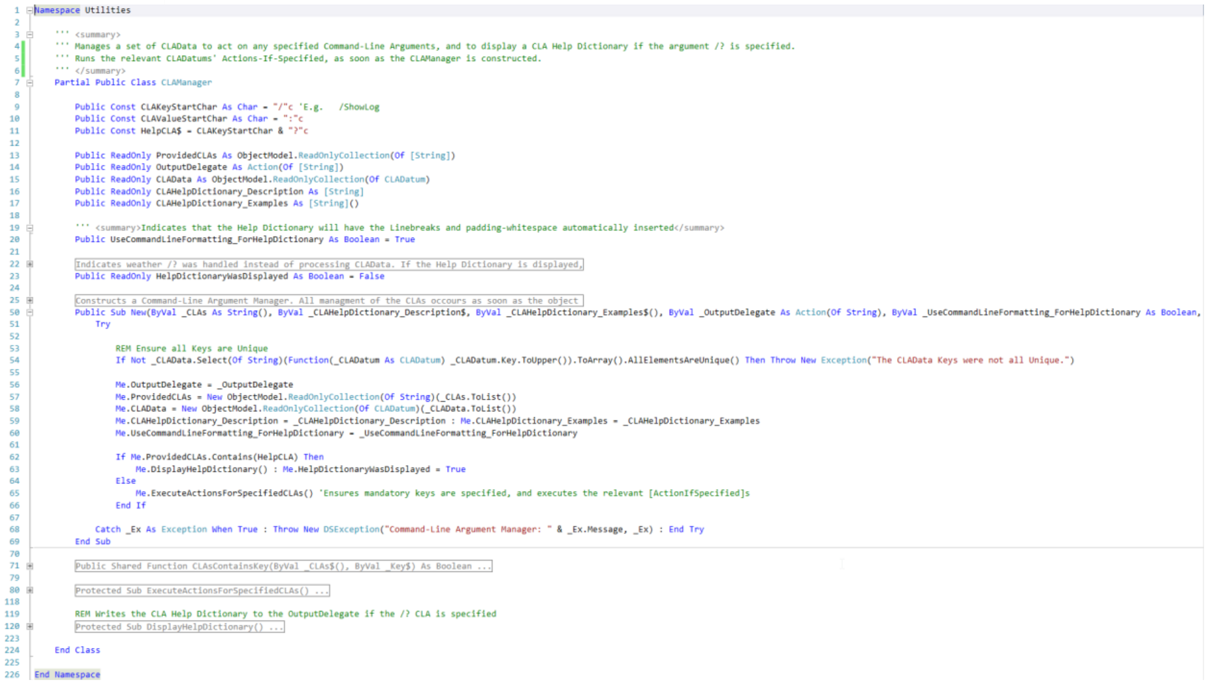
To make the process of understanding command-line input to the DSCLI program *significantly* easier, I shall develop a quick command-line argument Manager. It can be found in the `DocScript.Utilities` namespace.

It takes in a Key in the syntax: `/Key` or `/Key:Value` or `/Key:"Value"`. The constructor to a CLAManager then takes in a series of CLA Data, which each correspond to a given (case-insensitive) Key, and map this key to an Action, to be run if the Key is specified…

Structure and Modulatory: This Implementation is modular and multi-purpose, because…

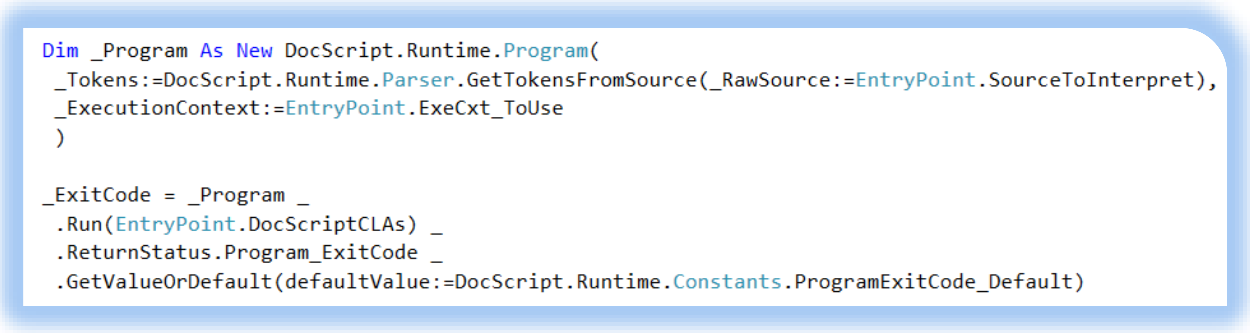
- The DSCLI binary (an *.exe* file) can perform different actions depending on the command-line arguments specified. With `/Run`, it performs all three stages of interpretation, whereas with `/GetProgramTree`, only the first two (parsing and lexing) occur.
- This means that program fits in well with operating system interoperability systems, such as command-line piping in Win32. The program's output can be passed through different subsequent programs, and this output can differ depending on the CLAs to DSCLI.exe.
- For example, running the command `DSCLI /Run /SourceFile:"HelloWorld.DS" | clip` provides different functionality from the DSCLI program, than in the command `dscli /Run /SourceFile:" HelloWorld.DS" /LogToConsole | more.com` (*.com* is an old Win32 executable format left over from MS-DOS, before *.exe* became standard.)
- This is convenient because a wide variety of functionality is provided from a singular executable, which is more portable and manageable than having many different .exe files to all perform slightly different tasks.

The Surprisingly-simple Implementation



```
Namespace Utilities

    ''' <summary>
    ''' Manages a set of CLAData to act on any specified Command-Line Arguments, and to display a CLA Help Dictionary if the argument /? is specified.
    ''' Runs the relevant CLADatums' Actions-If-Specified, as soon as the CLAManager is constructed.
    ''' </summary>
    Partial Public Class CLAManager

        Public Const CLAKeyStartChar As Char = "/"c 'E.g.   /ShowLog
        Public Const CLAValueStartChar As Char = ":"c
        Public Const HelpCLA$ = CLAKeyStartChar & "?"c

        Public ReadOnly ProvidedCLAs As ObjectModel.ReadOnlyCollection(Of [String])
        Public ReadOnly OutputDelegate As Action(Of [String])
        Public ReadOnly CLAData As ObjectModel.ReadOnlyCollection(Of CLADatum)
        Public ReadOnly CLAHelpDictionary_Description As [String]
        Public ReadOnly CLAHelpDictionary_Examples As [String]()

        ''' <summary>Indicates that the Help Dictionary will have the Linebreaks and padding-whitespace automatically inserted</summary>
        Public UseCommandLineFormatting_ForHelpDictionary As Boolean = True

        Indicates weather /? was handled instead of processing CLAData. If the Help Dictionary is displayed,
        Public ReadOnly HelpDictionaryWasDisplayed As Boolean = False

        Constructs a Command-Line Argument Manager. All managment of the CLAs occours as soon as the object
        Public Sub New(ByVal _CLAs As String(), ByVal _CLAHelpDictionary_Description$, ByVal _CLAHelpDictionary_Examples$(), ByVal _OutputDelegate As Action(Of String), ByVal _UseCommandLineFormatting_ForHelpDictionary As Boolean,
            Try

                REM Ensure all Keys are Unique
                If Not _CLAData.Select(Of String)(Function(_CLADatum As CLADatum) _CLADatum.Key.ToUpper()).ToArray().AllElementsAreUnique() Then Throw New Exception("The CLAData Keys were not all Unique.")

                Me.OutputDelegate = _OutputDelegate
                Me.ProvidedCLAs = New ObjectModel.ReadOnlyCollection(Of String)(_CLAs.ToList())
                Me.CLAData = New ObjectModel.ReadOnlyCollection(Of CLADatum)(_CLAData.ToList())
                Me.CLAHelpDictionary_Description = _CLAHelpDictionary_Description : Me.CLAHelpDictionary_Examples = _CLAHelpDictionary_Examples
                Me.UseCommandLineFormatting_ForHelpDictionary = _UseCommandLineFormatting_ForHelpDictionary

                If Me.ProvidedCLAs.Contains(HelpCLA) Then
                    Me.DisplayHelpDictionary() : Me.HelpDictionaryWasDisplayed = True
                Else
                    Me.ExecuteActionsForSpecifiedCLAs() 'Ensures mandatory keys are specified, and executes the relevant [ActionIfSpecified]s
                End If

            Catch _Ex As Exception When True : Throw New DSException("Command-Line Argument Manager: " & _Ex.Message, _Ex) : End Try
        End Sub

        Public Shared Function CLAsContainsKey(ByVal _CLAs$(), ByVal _Key$) As Boolean ...

        Protected Sub ExecuteActionsForSpecifiedCLAs() ...

        REM Writes the CLA Help Dictionary to the OutputDelegate if the /? CLA is specified
        Protected Sub DisplayHelpDictionary() ...

    End Class

End Namespace
```

The *meat* of the CLA-Interpreter comes down to these simple few lines:

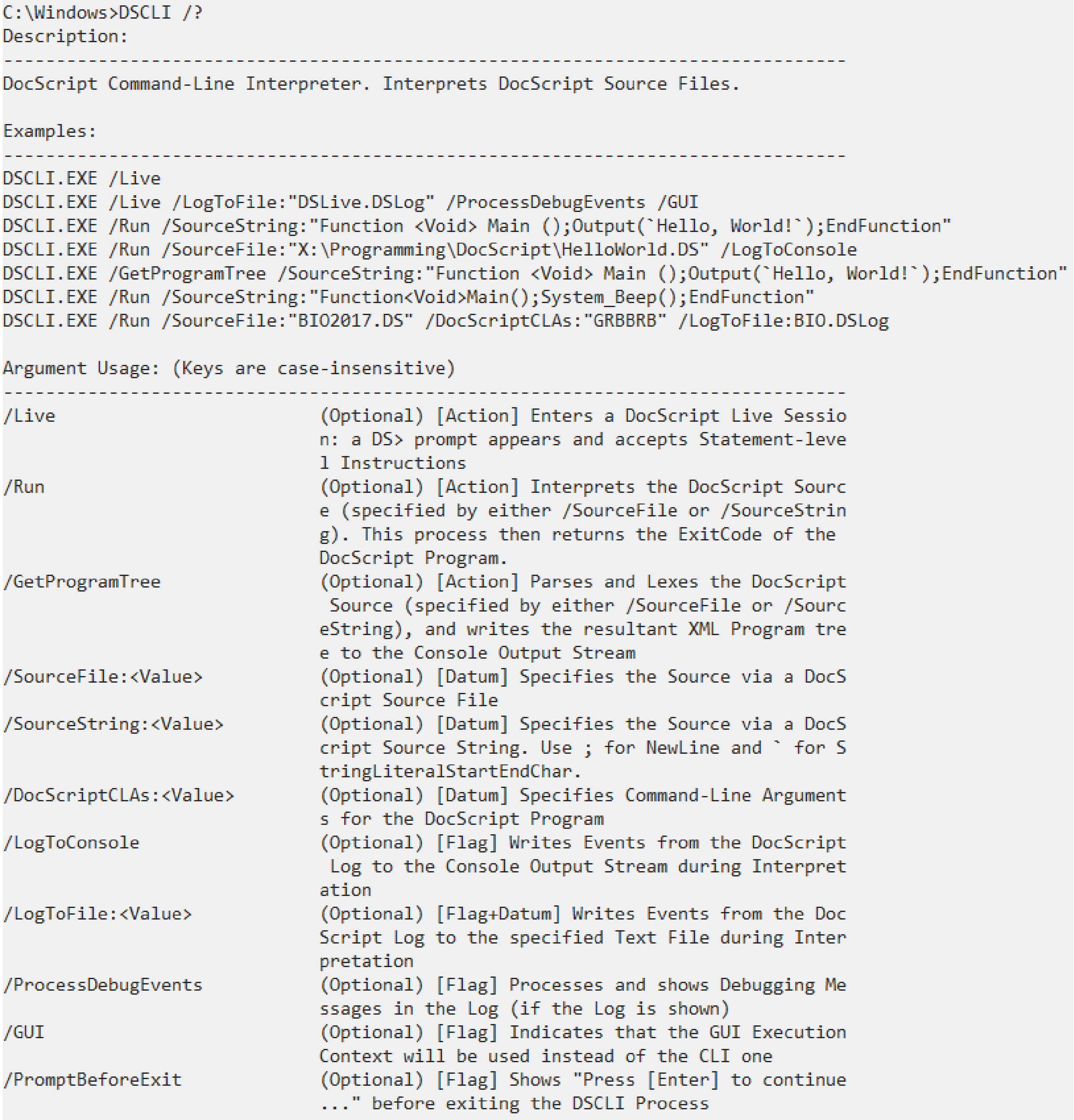


```
Dim _Program As New DocScript.Runtime.Program(
 _Tokens:=DocScript.Runtime.Parser.GetTokensFromSource(_RawSource:=EntryPoint.SourceToInterpret),
 _ExecutionContext:=EntryPoint.ExeCxt_ToUse
 )

_ExitCode = _Program _
 .Run(EntryPoint.DocScriptCLAs) _
 .ReturnStatus.Program_ExitCode _
 .GetValueOrDefault(defaultValue:=DocScript.Runtime.Constants.ProgramExitCode_Default)
```

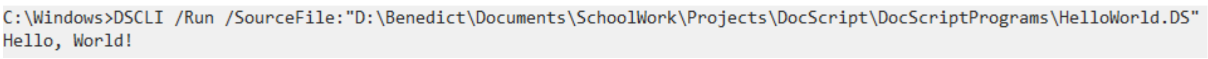
{That}, is how easy it becomes to run a DocScript Program, because of all the heavy-lifting being abstracted into the DLL.

Using the Interpreter

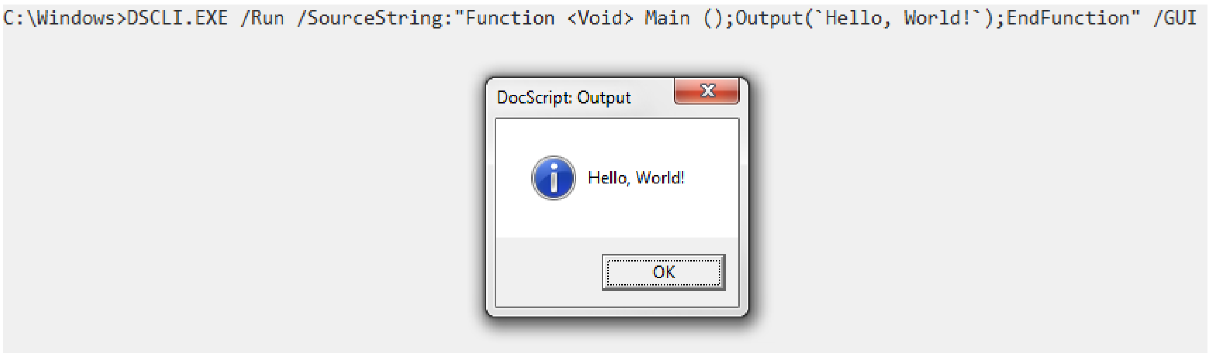
Here are some examples of DSCLI in use…



```
C:\Windows>DSCLI /?
Description:
--------------------------------------------------------------------------------
DocScript Command-Line Interpreter. Interprets DocScript Source Files.

Examples:
--------------------------------------------------------------------------------
DSCLI.EXE /Live
DSCLI.EXE /Live /LogToFile:"DSLive.DSLog" /ProcessDebugEvents /GUI
DSCLI.EXE /Run /SourceString:"Function <Void> Main ();Output(`Hello, World!`);EndFunction"
DSCLI.EXE /Run /SourceFile:"X:\Programming\DocScript\HelloWorld.DS" /LogToConsole
DSCLI.EXE /GetProgramTree /SourceString:"Function <Void> Main ();Output(`Hello, World!`);EndFunction"
DSCLI.EXE /Run /SourceString:"Function<Void>Main();System_Beep();EndFunction"
DSCLI.EXE /Run /SourceFile:"BIO2017.DS" /DocScriptCLAs:"GRBBRB" /LogToFile:BIO.DSLog

Argument Usage: (Keys are case-insensitive)
--------------------------------------------------------------------------------
/Live                          (Optional) [Action] Enters a DocScript Live Sessio
                               n: a DS> prompt appears and accepts Statement-leve
                               l Instructions
/Run                           (Optional) [Action] Interprets the DocScript Sourc
                               e (specified by either /SourceFile or /SourceStrin
                               g). This process then returns the ExitCode of the
                               DocScript Program.
/GetProgramTree                (Optional) [Action] Parses and Lexes the DocScript
                                Source (specified by either /SourceFile or /Sourc
                               eString), and writes the resultant XML Program tre
                               e to the Console Output Stream
/SourceFile:<Value>            (Optional) [Datum] Specifies the Source via a DocS
                               cript Source File
/SourceString:<Value>          (Optional) [Datum] Specifies the Source via a DocS
                               cript Source String. Use ; for NewLine and ` for S
                               tringLiteralStartEndChar.
/DocScriptCLAs:<Value>         (Optional) [Datum] Specifies Command-Line Argument
                               s for the DocScript Program
/LogToConsole                  (Optional) [Flag] Writes Events from the DocScript
                                Log to the Console Output Stream during Interpret
                               ation
/LogToFile:<Value>             (Optional) [Flag+Datum] Writes Events from the Doc
                               Script Log to the specified Text File during Inter
                               pretation
/ProcessDebugEvents            (Optional) [Flag] Processes and shows Debugging Me
                               ssages in the Log (if the Log is shown)
/GUI                           (Optional) [Flag] Indicates that the GUI Execution
                               Context will be used instead of the CLI one
/PromptBeforeExit              (Optional) [Flag] Shows "Press [Enter] to continue
                               ..." before exiting the DSCLI Process
```

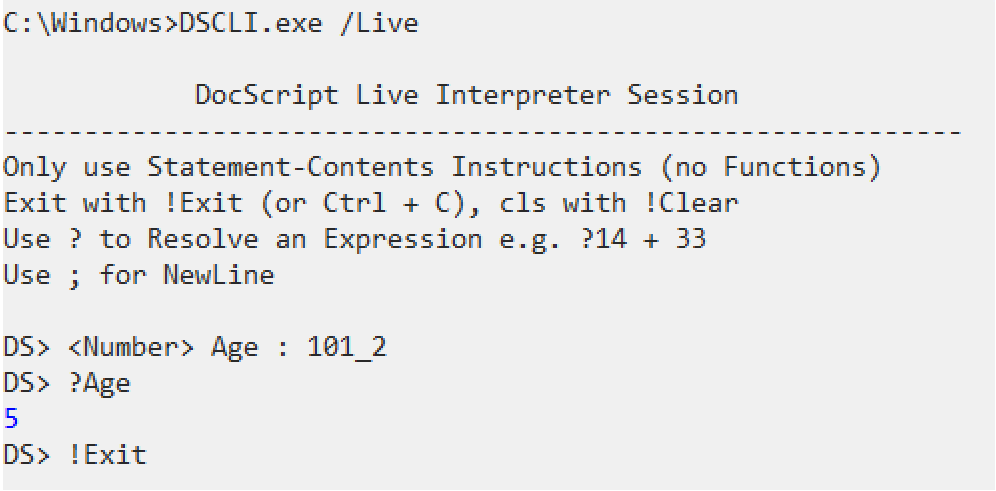


```
C:\Windows>DSCLI /Run /SourceFile:"D:\Benedict\Documents\SchoolWork\Projects\DocScript\DocScriptPrograms\HelloWorld.DS"
Hello, World!
```



```
C:\Windows>DSCLI.EXE /Run /SourceString:"Function <Void> Main ();Output(`Hello, World!`);EndFunction" /GUI
```

Hello, World!



```
C:\Windows>DSCLI.exe /Live

            DocScript Live Interpreter Session
-------------------------------------------------------------
Only use Statement-Contents Instructions (no Functions)
Exit with !Exit (or Ctrl + C), cls with !Clear
Use ? to Resolve an Expression e.g. ?14 + 33
Use ; for NewLine

DS> <Number> Age : 101_2
DS> ?Age
5
DS> !Exit
```

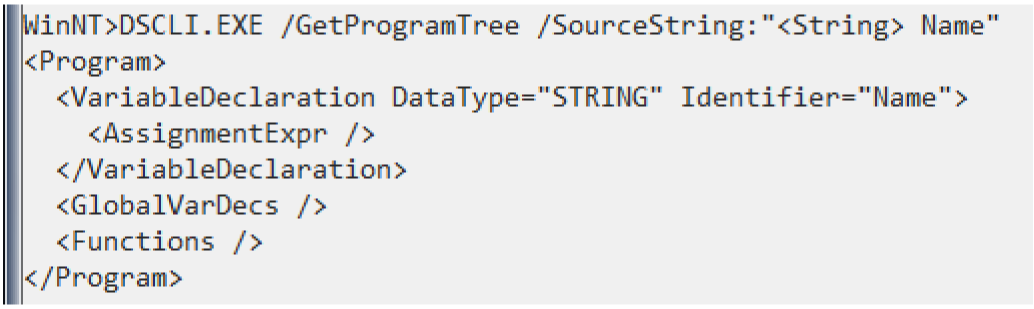
[Above] Prototype 1 of DSCLI's /Live Mode

Debugging

Before `DSCLI.exe` behaved as it should, and produced the above screenshots, there were a number of bugs which I had to correct:

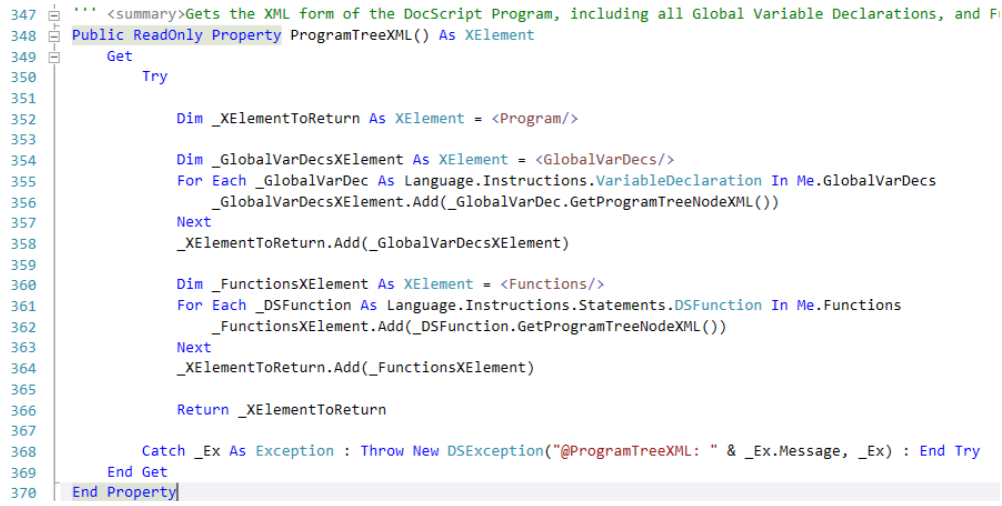
Incorrect Program XML Serialisation

When using the `/GetProgramTree` CLA, I noticed that the XML tree was not correctly formed; notice how the `<VariableDeclaration>` is not within the `<GlobalVarDecs>` Node...



```
WinNT>DSCLI.EXE /GetProgramTree /SourceString:"<String> Name"
<Program>
  <VariableDeclaration DataType="STRING" Identifier="Name">
    <AssignmentExpr />
  </VariableDeclaration>
  <GlobalVarDecs />
  <Functions />
</Program>
```

To mend this, I simply needed to add the GlobalVarDec nodes to `<GlobalVarDecs/>`, instead of `<Program/>`...

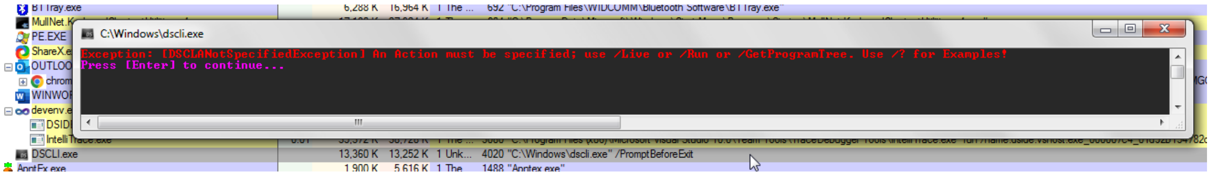


```
''' <summary>Gets the XML form of the DocScript Program, including all Global Variable Declarations, and F
Public ReadOnly Property ProgramTreeXML() As XElement
    Get
        Try

            Dim _XElementToReturn As XElement = <Program/>

            Dim _GlobalVarDecsXElement As XElement = <GlobalVarDecs/>
            For Each _GlobalVarDec As Language.Instructions.VariableDeclaration In Me.GlobalVarDecs
                _GlobalVarDecsXElement.Add(_GlobalVarDec.GetProgramTreeNodeXML())
            Next
            _XElementToReturn.Add(_GlobalVarDecsXElement)

            Dim _FunctionsXElement As XElement = <Functions/>
            For Each _DSFunction As Language.Instructions.Statements.DSFunction In Me.Functions
                _FunctionsXElement.Add(_DSFunction.GetProgramTreeNodeXML())
            Next
            _XElementToReturn.Add(_FunctionsXElement)

            Return _XElementToReturn

        Catch _Ex As Exception : Throw New DSException("@ProgramTreeXML: " & _Ex.Message, _Ex) : End Try
    End Get
End Property
```

Console closing immediately

When creating an .lnk file to the DSCLI.exe binary, with a set of command-line arguments to run a *.DS file, it was annoying that the console window would close, before I (and therefore any future user of the application) would have a chance to see what the output from DSCLI – and thereby also the DocScript Program – actually was.

To fix this, I have added a `/PromptBeforeExit` CLA switch (flag). This causes the interpreter to wait for the press of the [Enter] key, before exiting and therefore dismissing the Console window:



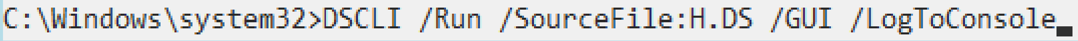
```
C:\Windows\dscli.exe
Exception: [DSCLANotSpecifiedException] An Action must be specified; use /Live or /Run or /GetProgramTree. Use /? for Examples!
Press [Enter] to continue...
```

```
DSCLI.exe   13,360 K   13,252 K   1 Unk...   4020 "C:\Windows\dscli.exe" /PromptBeforeExit
```

As can be thence seen, this requests the keypress, ↑ ***even if there is an Exception*** ↑.

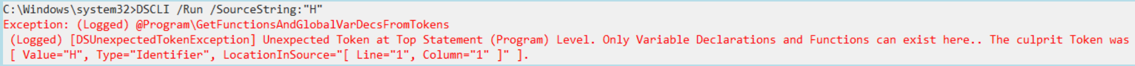
Prototype: This point marks a milestone in the product, which is now capable of…

- Parsing and understanding a series of Command-Line Arguments to the DSCLI binary, and performing one of several different actions, depending on the CLA. For example:



```
C:\Windows\system32>DSCLI /Run /SourceFile:H.DS /GUI /LogToConsole_
```

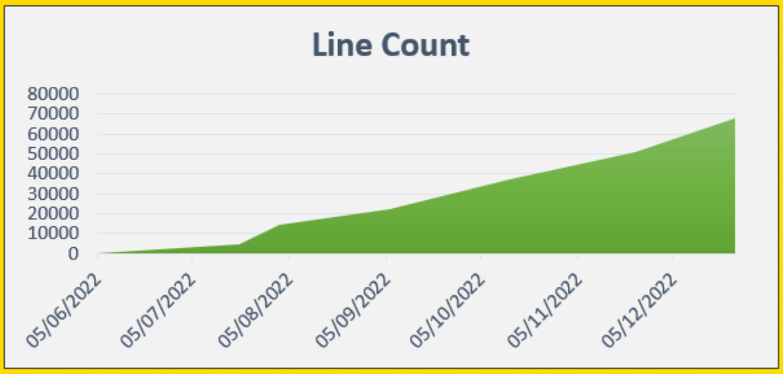
- Executing, or generating the XML-Program-Tree for, a DocScript Program – including requesting Input and providing Output, via a Win32 Console.
- Displaying clear error messages, resulting from Exceptions raised during the Interpretation process. For instance:



```
C:\Windows\system32>DSCLI /Run /SourceString:"H"
Exception: (Logged) @Program\GetFunctionsAndGlobalVarDecsFromTokens
 (Logged) [DSUnexpectedTokenException] Unexpected Token at Top Statement (Program) Level. Only Variable Declarations and Functions can exist here.. The culprit Token was
 [ Value="H", Type="Identifier", LocationInSource="[ Line="1", Column="1" ]" ].
```

Progress Check

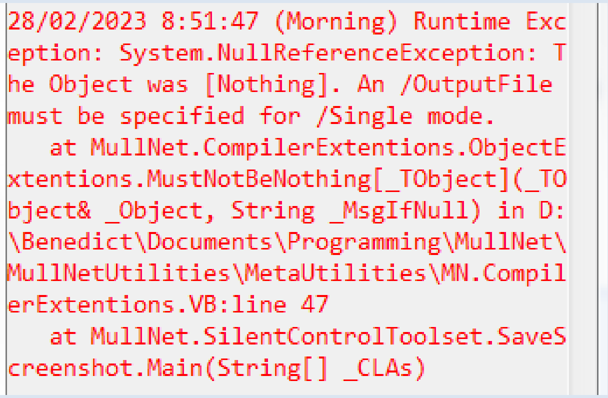
At this point in the development,
I have written 68031 Lines of
code…



Iterative Testing

Within §Analysis, I listed a number of types of test, along with example testing data, which I could use to test the product once implemented. I shall now make use of this testing plan…

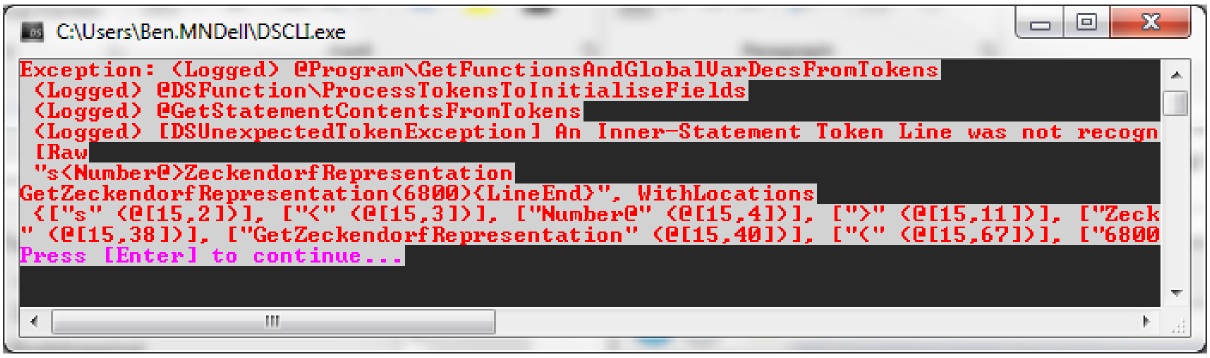
Assert-style Test	☑ Passed?
Input and Output: Erroneous DocScript source can be taken in, but the interpreter determines that there is something wrong, instead of continuing and crashing.	Yes; see previous screenshots
Input and Output: Standalone Expressions can be taken in in the `DSCLI /Live` mode, and are evaluated using the current Symbol Tables.	Yes; see previous screenshots

Usability: A high degree of verbosity is present in the error messages, leading to an unambiguous prognosis of whence the error came.	No; having conducted this test, I realise that it would be nice to see a StackTrace in addition to the Exception Message. Therefore, I shall change from using `Exception.Message`, to `Exception.ToString()`, which includes the StakTrace, like this:
	
	`28/02/2023 8:51:47 (Morning) Runtime Exception: System.NullReferenceException: The Object was [Nothing]. An /OutputFile must be specified for /Single mode.`
	`   at MullNet.CompilerExtentions.ObjectExtentions.MustNotBeNothing[_TObject](_TObject& _Object, String _MsgIfNull) in D:\Benedict\Documents\Programming\MullNet\MullNetUtilities\MetaUtilities\MN.CompilerExtentions.VB:line 47`
	`   at MullNet.SilentControlToolset.SaveScreenshot.Main(String[] _CLAs)`
	(The `at *()` strings on the bottom represent the call stack, and its functions)

All of the "**Standalone Expressions**" and "**Command-Line Arguments**" from the *Testing* section of *§Design*, can be run successfully.	Yes; see previous screenshots

Justifications for Actions Taken: By printing the entire stake trace, it is significantly easier to see exactly *where* the problem occurred in the code, and what state the Win32 Process was in when the

Exception was Thrown. Here is an example of a more widescreen StackTrace:



```
Exception: (Logged) @Program\GetFunctionsAndGlobalVarDecsFromTokens
 (Logged) @DSFunction\ProcessTokensToInitialiseFields
 (Logged) @GetStatementContentsFromTokens
 (Logged) [DSUnexpectedTokenException] An Inner-Statement Token Line was not recogn
 [Raw
 "s<Number@>ZeckendorfRepresentation
GetZeckendorfRepresentation(6800){LineEnd}", WithLocations
 {["s" (@[15,2])], ["<" (@[15,3])], ["Number@" (@[15,4])], [">" (@[15,11])], ["Zeck
" (@[15,38])], ["GetZeckendorfRepresentation" (@[15,40])], ["(" (@[15,67])], ["6800
Press [Enter] to continue...
```

[Stage 3]

Windows IDE (**DSIDE.EXE**) Development

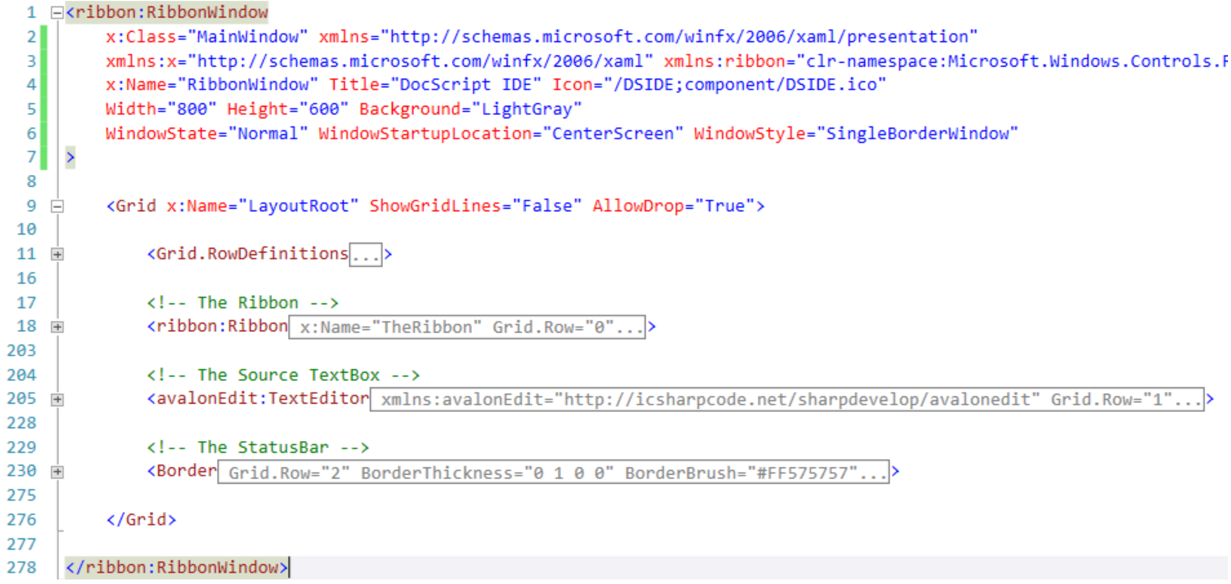
Progress Recap: Where am I in the development plan? *[Review]*

- **Done**: I have now written the first Interpreter Implementation, `DSCLI.EXE`. Initial testing has also been performed on it.
- **Next**: I will implement this DLL into the second of the three implementations, as was described in detail within §Design. This implementation is a graphical Windows Program.

WPF and XAML

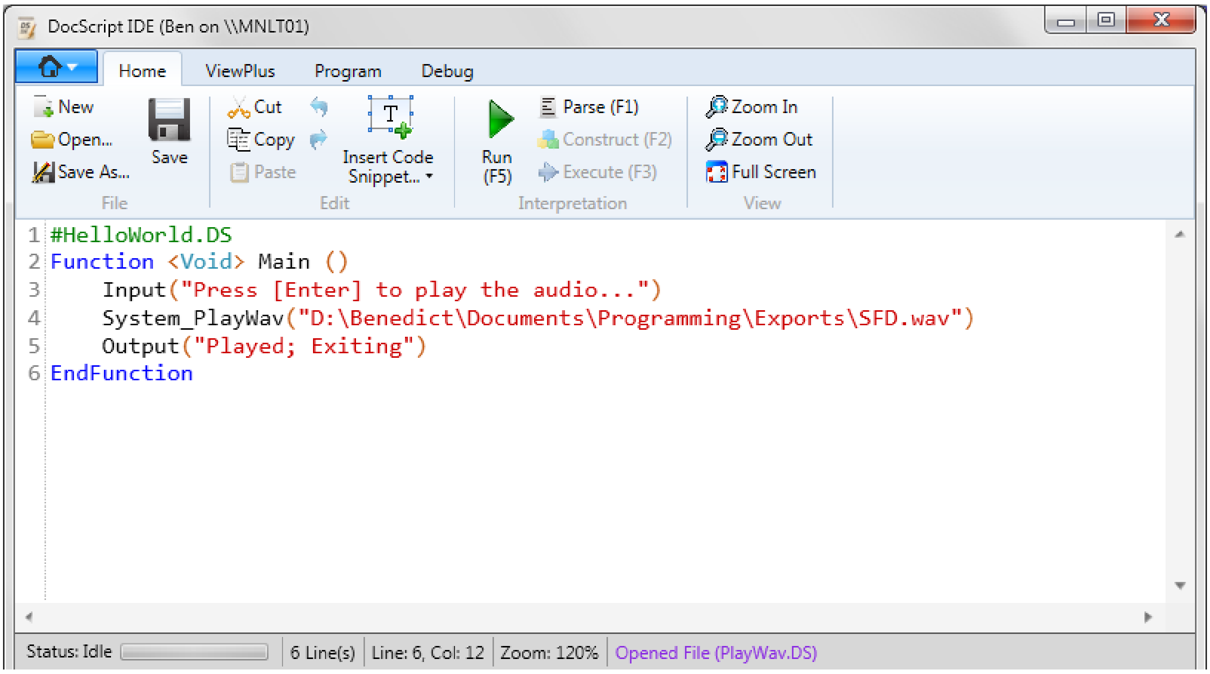
The moderately complex design requirements of the Windows IDE mean that I am best off using WPF instead of Windows Forms for this exe. This requires creating the markup for the User-Interface in an almost HTML-like derivative of XML, called **XAML** (*"Zamol"*).

This is an overview of the ↓ Main Window's XAML ↓ ...



```
<ribbon:RibbonWindow
        x:Class="MainWindow" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:ribbon="clr-namespace:Microsoft.Windows.Controls.F
        x:Name="RibbonWindow" Title="DocScript IDE" Icon="/DSIDE;component/DSIDE.ico"
        Width="800" Height="600" Background="LightGray"
        WindowState="Normal" WindowStartupLocation="CenterScreen" WindowStyle="SingleBorderWindow"
>

    <Grid x:Name="LayoutRoot" ShowGridLines="False" AllowDrop="True">

        <Grid.RowDefinitions...>

        <!-- The Ribbon -->
        <ribbon:Ribbon x:Name="TheRibbon" Grid.Row="0"...>

        <!-- The Source TextBox -->
        <avalonEdit:TextEditor xmlns:avalonEdit="http://icsharpcode.net/sharpdevelop/avalonedit" Grid.Row="1"...>

        <!-- The StatusBar -->
        <Border Grid.Row="2" BorderThickness="0 1 0 0" BorderBrush="#FF575757"...>

    </Grid>

</ribbon:RibbonWindow>
```

...Which produces ↓ this ↓ component:

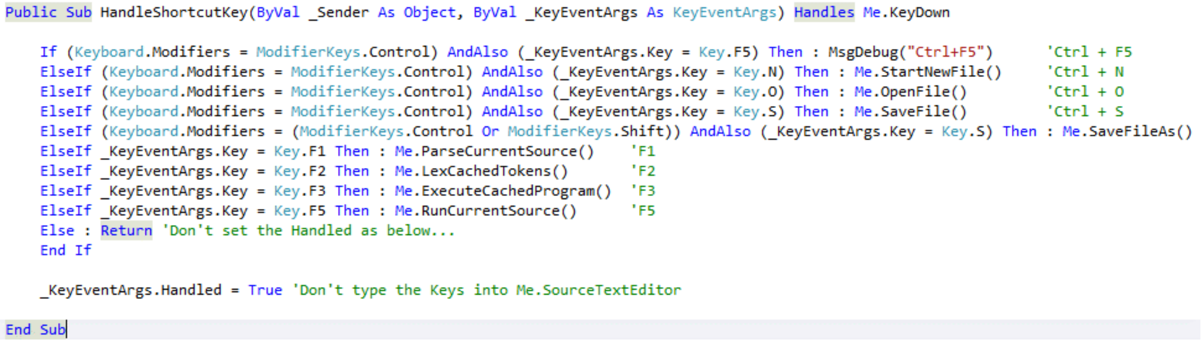


```
#HelloWorld.DS
Function <Void> Main ()
    Input("Press [Enter] to play the audio...")
    System_PlayWav("D:\Benedict\Documents\Programming\Exports\SFD.wav")
    Output("Played; Exiting")
EndFunction
```

Status: Idle | 6 Line(s) | Line: 6, Col: 12 | Zoom: 120% | Opened File (PlayWav.DS)

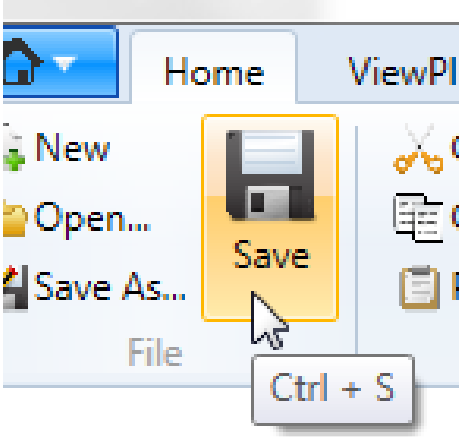
Keyboard Shortcuts

To aid in the usability of the application, I have added the following Keyboard Shortcuts:



```
Public Sub HandleShortcutKey(ByVal _Sender As Object, ByVal _KeyEventArgs As KeyEventArgs) Handles Me.KeyDown

    If (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.F5) Then : MsgDebug("Ctrl+F5")      'Ctrl + F5
    ElseIf (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.N) Then : Me.StartNewFile()     'Ctrl + N
    ElseIf (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.O) Then : Me.OpenFile()         'Ctrl + O
    ElseIf (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.S) Then : Me.SaveFile()         'Ctrl + S
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.S) Then : Me.SaveFileAs()
    ElseIf _KeyEventArgs.Key = Key.F1 Then : Me.ParseCurrentSource()     'F1
    ElseIf _KeyEventArgs.Key = Key.F2 Then : Me.LexCachedTokens()        'F2
    ElseIf _KeyEventArgs.Key = Key.F3 Then : Me.ExecuteCachedProgram()   'F3
    ElseIf _KeyEventArgs.Key = Key.F5 Then : Me.RunCurrentSource()       'F5
    Else : Return 'Don't set the Handled as below...
    End If

    _KeyEventArgs.Handled = True 'Don't type the Keys into Me.SourceTextEditor

End Sub
```

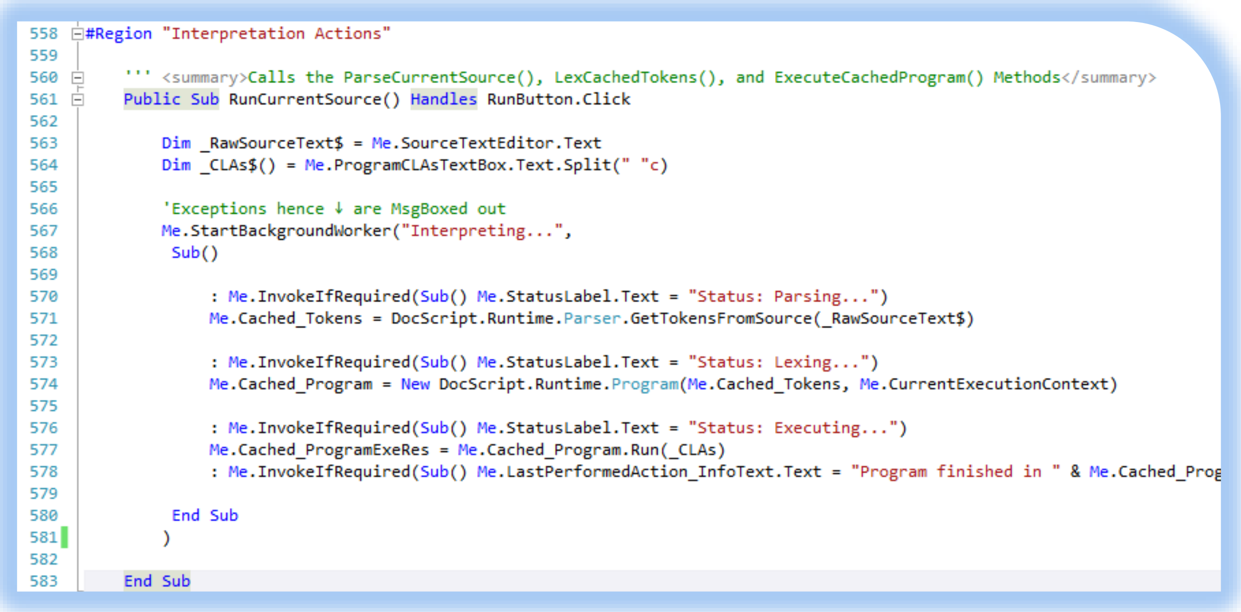
I have also added ToolTips to the Buttons which have corresponding Keyboard-Shortcuts, like so:



← The ToolTip label clearly shows the shortcut-key

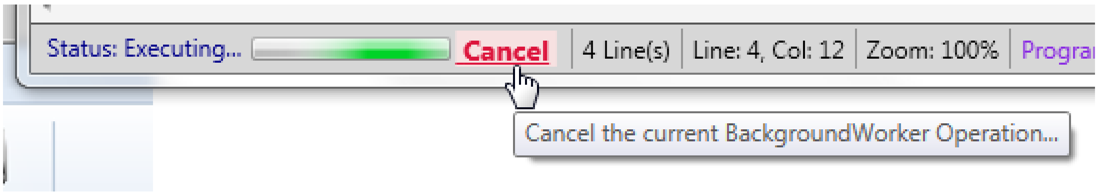
Implementing the DocScript DLL

The code-behind for the [Run (F5)] Button looks like this:



```
#Region "Interpretation Actions"

    ''' <summary>Calls the ParseCurrentSource(), LexCachedTokens(), and ExecuteCachedProgram() Methods</summary>
    Public Sub RunCurrentSource() Handles RunButton.Click

        Dim _RawSourceText$ = Me.SourceTextEditor.Text
        Dim _CLAs$() = Me.ProgramCLAsTextBox.Text.Split(" "c)

        'Exceptions hence ↓ are MsgBoxed out
        Me.StartBackgroundWorker("Interpreting...",
         Sub()

            : Me.InvokeIfRequired(Sub() Me.StatusLabel.Text = "Status: Parsing...")
            Me.Cached_Tokens = DocScript.Runtime.Parser.GetTokensFromSource(_RawSourceText$)

            : Me.InvokeIfRequired(Sub() Me.StatusLabel.Text = "Status: Lexing...")
            Me.Cached_Program = New DocScript.Runtime.Program(Me.Cached_Tokens, Me.CurrentExecutionContext)

            : Me.InvokeIfRequired(Sub() Me.StatusLabel.Text = "Status: Executing...")
            Me.Cached_ProgramExeRes = Me.Cached_Program.Run(_CLAs)
            : Me.InvokeIfRequired(Sub() Me.LastPerformedAction_InfoText.Text = "Program finished in " & Me.Cached_Prog

         End Sub
        )

    End Sub
```

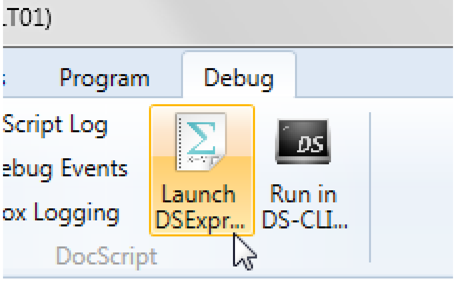
Notably, this is done in a separate, **background Thread**. This means that the user-interface remains responsive throughout interpretation (and a number of other operations). In addition, the operation can be cancelled at any time, by means of the [Cancel] Button, which appears only during a Background Operation:



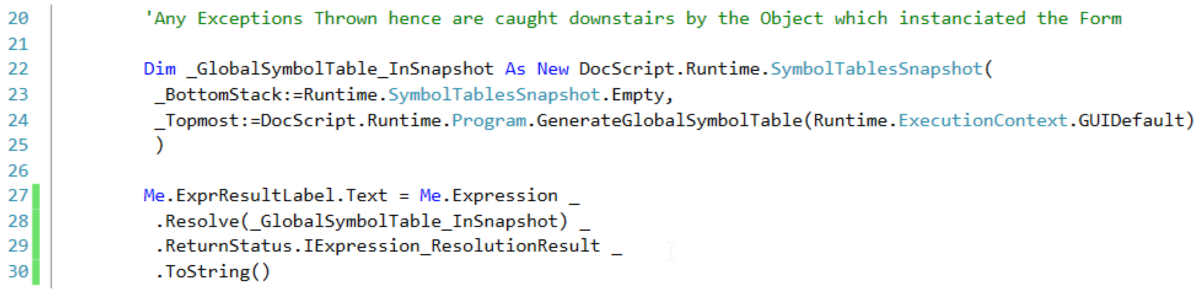
Cancel the current BackgroundWorker Operation...

*Standalone Expression Resolution Utility (**DSExpr.EXE**)*

I also wrote the DSExpr executable to accompany the IDE:

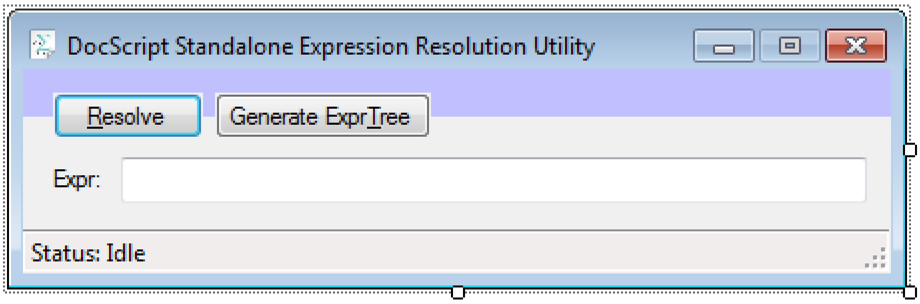


It does not call `Program.Run()`, but rather, `IExpression.Resolve()`:



```
'Any Exceptions Thrown hence are caught downstairs by the Object which instanciated the Form

Dim _GlobalSymbolTable_InSnapshot As New DocScript.Runtime.SymbolTablesSnapshot(
 _BottomStack:=Runtime.SymbolTablesSnapshot.Empty,
 _Topmost:=DocScript.Runtime.Program.GenerateGlobalSymbolTable(Runtime.ExecutionContext.GUIDefault)
 )

Me.ExprResultLabel.Text = Me.Expression _
 .Resolve(_GlobalSymbolTable_InSnapshot) _
 .ReturnStatus.IExpression_ResolutionResult _
 .ToString()
```

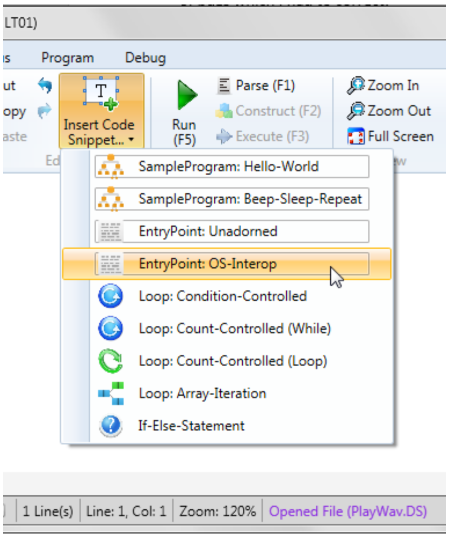
It is a simple Windows Forms program, and the Main Window looks like this:

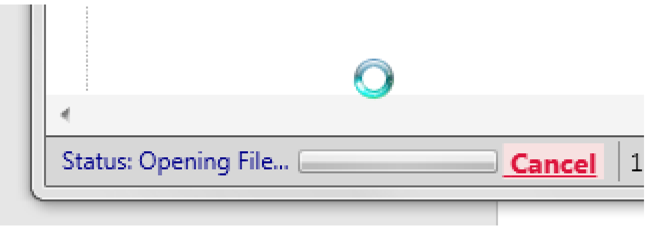


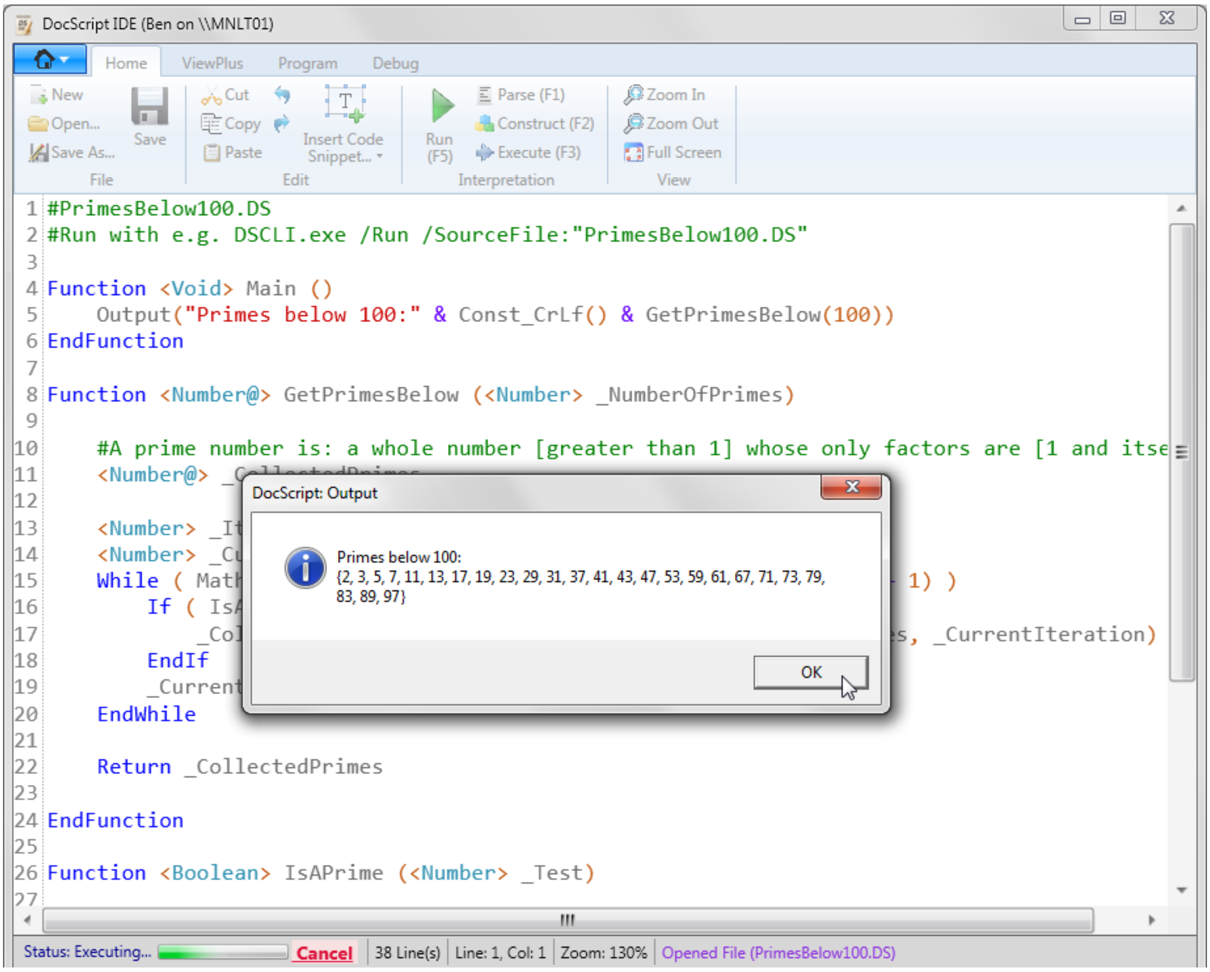
Using the Interpreter

Here are some examples of DSIDE in-use…

(…After the implementation of AvalonEdit as described below…)


LT01)
Program
Debug
Insert Code Snippet...
Run (F5)
Parse (F1)
Construct (F2)
Execute (F3)
Zoom In
Zoom Out
Full Screen
SampleProgram: Hello-World
SampleProgram: Beep-Sleep-Repeat
EntryPoint: Unadorned
EntryPoint: OS-Interop
Loop: Condition-Controlled
Loop: Count-Controlled (While)
Loop: Count-Controlled (Loop)
Loop: Array-Iteration
If-Else-Statement
1 Line(s) | Line: 1, Col: 1 | Zoom: 120% | Opened File (PlayWav.DS)





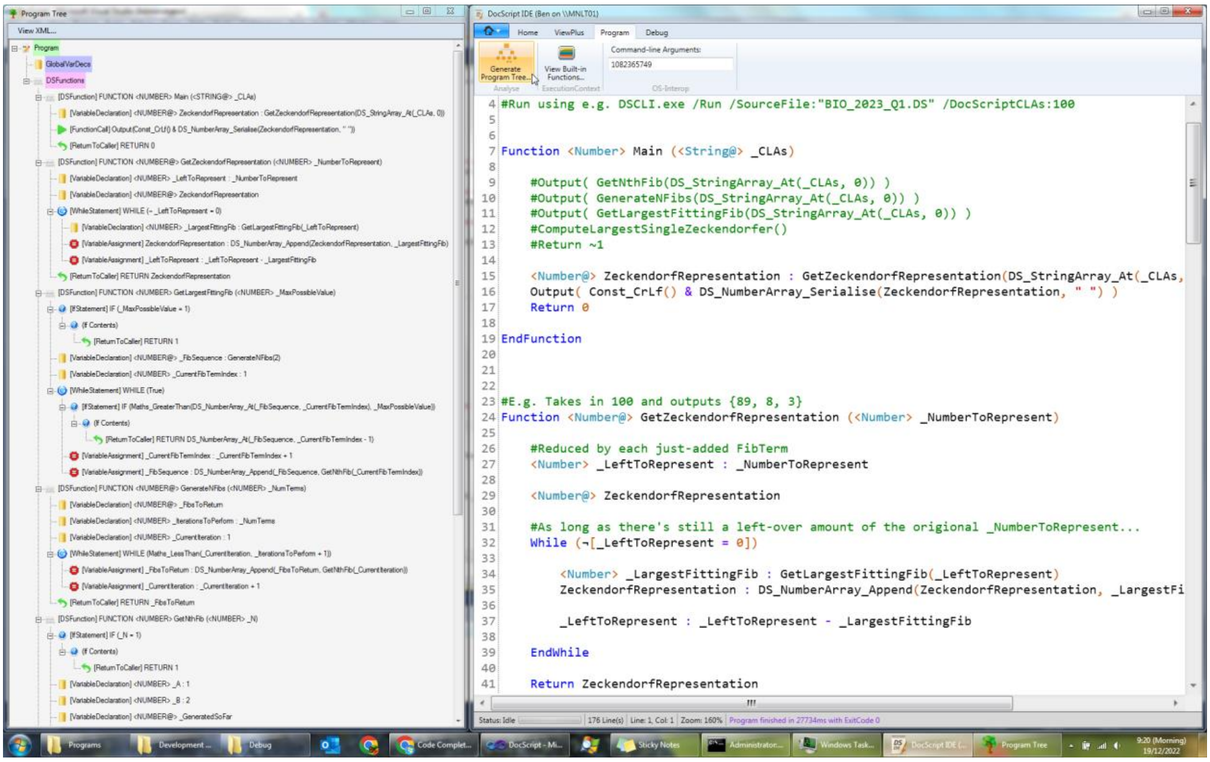
```
#PrimesBelow100.DS
#Run with e.g. DSCLI.exe /Run /SourceFile:"PrimesBelow100.DS"

Function <Void> Main ()
    Output("Primes below 100:" & Const_CrLf() & GetPrimesBelow(100))
EndFunction

Function <Number@> GetPrimesBelow (<Number> _NumberOfPrimes)

    #A prime number is: a whole number [greater than 1] whose only factors are [1 and itse
    <Number@> _CollectedPrimes

    <Number> _It
    <Number> _Cu
    While ( Math                                                        1) )
        If ( IsA
            _Col                                                       s, _CurrentIteration)
        EndIf
        _Current
    EndWhile

    Return _CollectedPrimes

EndFunction

Function <Boolean> IsAPrime (<Number> _Test)
```

DocScript: Output

```
Primes below 100:
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97}
```

Status: Executing... Cancel | 38 Line(s) | Line: 1, Col: 1 | Zoom: 130% | Opened File (PrimesBelow100.DS)

[Above] DSIDE running a mathematical program to compute Primes!

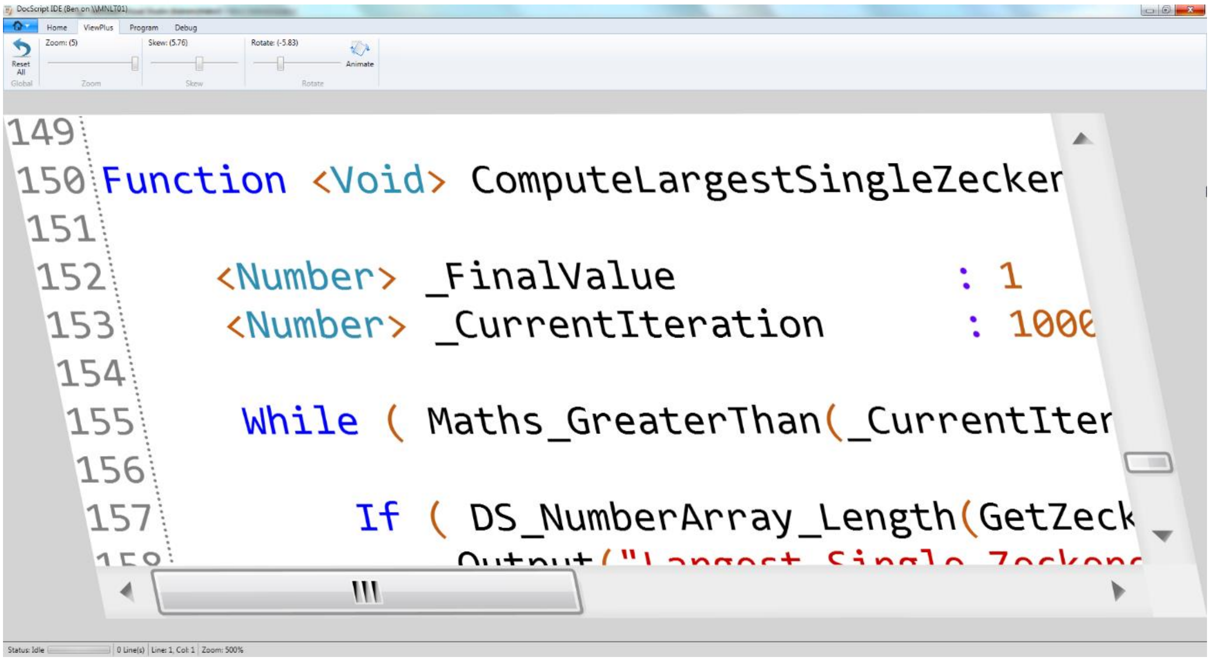


```
Program
    GlobalVarDecs
    DSFunctions
        [DSFunction] FUNCTION <NUMBER> Main (<STRING@> _CLAs)
            [VariableDeclaration] <NUMBER@> ZeckendorfRepresentation : GetZeckendorfRepresentation(DS_StringArray_At(_CLAs, 0))
            [FunctionCall] Output(Const_CrLf() & DS_NumberArray_Serialise(ZeckendorfRepresentation, " "))
            [ReturnToCaller] RETURN 0
        [DSFunction] FUNCTION <NUMBER@> GetZeckendorfRepresentation (<NUMBER> _NumberToRepresent)
            [VariableDeclaration] <NUMBER> _LeftToRepresent : _NumberToRepresent
            [VariableDeclaration] <NUMBER@> ZeckendorfRepresentation
            [WhileStatement] WHILE (¬ _LeftToRepresent = 0)
                [VariableDeclaration] <NUMBER> _LargestFittingFib : GetLargestFittingFib(_LeftToRepresent)
                [VariableAssignment] ZeckendorfRepresentation : DS_NumberArray_Append(ZeckendorfRepresentation, _LargestFittingFib)
                [VariableAssignment] _LeftToRepresent : _LeftToRepresent - _LargestFittingFib
            [ReturnToCaller] RETURN ZeckendorfRepresentation
        [DSFunction] FUNCTION <NUMBER> GetLargestFittingFib (<NUMBER> _MaxPossibleValue)
            [IfStatement] IF (_MaxPossibleValue = 1)
                (If Contents)
                    [ReturnToCaller] RETURN 1
            [VariableDeclaration] <NUMBER@> _FibSequence : GenerateNFibs(2)
            [VariableDeclaration] <NUMBER> _CurrentFibTermIndex : 1
            [WhileStatement] WHILE (True)
                [IfStatement] IF (Maths_GreaterThan(DS_NumberArray_At(_FibSequence, _CurrentFibTermIndex), _MaxPossibleValue))
                    (If Contents)
                        [ReturnToCaller] RETURN DS_NumberArray_At(_FibSequence, _CurrentFibTermIndex - 1)
                [VariableAssignment] _CurrentFibTermIndex : _CurrentFibTermIndex + 1
                [VariableAssignment] _FibSequence : DS_NumberArray_Append(_FibSequence, GetNthFib(_CurrentFibTermIndex))
        [DSFunction] FUNCTION <NUMBER@> GenerateNFibs (<NUMBER> _NumTerms)
            [VariableDeclaration] <NUMBER@> _FibsToReturn
            [VariableDeclaration] <NUMBER> _IterationsToPerform : _NumTerms
            [VariableDeclaration] <NUMBER> _CurrentIteration : 1
            [WhileStatement] WHILE (Maths_LessThan(_CurrentIteration, _IterationsToPerform + 1))
                [VariableAssignment] _FibsToReturn : DS_NumberArray_Append(_FibsToReturn, GetNthFib(_CurrentIteration))
                [VariableAssignment] _CurrentIteration : _CurrentIteration + 1
            [ReturnToCaller] RETURN _FibsToReturn
        [DSFunction] FUNCTION <NUMBER> GetNthFib (<NUMBER> _N)
            [IfStatement] IF (_N = 1)
                (If Contents)
                    [ReturnToCaller] RETURN 1
            [VariableDeclaration] <NUMBER> _A : 1
            [VariableDeclaration] <NUMBER> _B : 2
            [VariableDeclaration] <NUMBER@> _GeneratedSoFar
```

Command-line Arguments: 1082365749

```
#Run using e.g. DSCLI.exe /Run /SourceFile:"BIO_2023_Q1.DS" /DocScriptCLAs:100


Function <Number> Main (<String@> _CLAs)

    #Output( GetNthFib(DS_StringArray_At(_CLAs, 0)) )
    #Output( GenerateNFibs(DS_StringArray_At(_CLAs, 0)) )
    #Output( GetLargestFittingFib(DS_StringArray_At(_CLAs, 0)) )
    #ComputeLargestSingleZeckendorfer()
    #Return ~1

    <Number@> ZeckendorfRepresentation : GetZeckendorfRepresentation(DS_StringArray_At(_CLAs,
    Output( Const_CrLf() & DS_NumberArray_Serialise(ZeckendorfRepresentation, " ") )
    Return 0

EndFunction



#E.g. Takes in 100 and outputs {89, 8, 3}
Function <Number@> GetZeckendorfRepresentation (<Number> _NumberToRepresent)

    #Reduced by each just-added FibTerm
    <Number> _LeftToRepresent : _NumberToRepresent

    <Number@> ZeckendorfRepresentation

    #As long as there's still a left-over amount of the origional _NumberToRepresent...
    While (¬[_LeftToRepresent = 0])

        <Number> _LargestFittingFib : GetLargestFittingFib(_LeftToRepresent)
        ZeckendorfRepresentation : DS_NumberArray_Append(ZeckendorfRepresentation, _LargestFi

        _LeftToRepresent : _LeftToRepresent - _LargestFittingFib

    EndWhile

    Return ZeckendorfRepresentation
```

Status: Idle | 176 Line(s) | Line: 1, Col: 1 | Zoom: 160% | Program finished in 27734ms with ExitCode 0



```
Function <Void> ComputeLargestSingleZecker

    <Number> _FinalValue              : 1
    <Number> _CurrentIteration        : 1000

    While ( Maths_GreaterThan(_CurrentIter

        If ( DS_NumberArray_Length(GetZeck
            Output("Largest Single Zeckend
```

[Above] The DSIDE ViewPlus Features

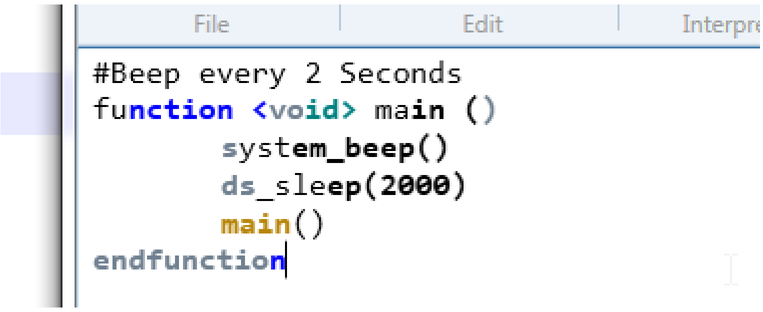
Debugging

Before DSIDE.exe behaved as it should, and produced the above screenshots, there were a number of bugs which I had to correct:

Syntax-highlighting Nightmares

Using simply a WPF RichTextBox gave me no end of problems with attempting to highlight certain selections of the text ("*Runs*", as they're called) in a specified colour. This was in part because of the insertion of paragraph breaks instead of line breaks whenever the enter key was pressed, and in part because the RichTextBox was doing nothing to prevent any form of rich text from being inserted into the document. Images, COM Objects, and even Adobe Photoshop documents could be pasted-in, and these would rather severely mess-up the highlighting offsets.

I frequently ended-up with aberrations like this:



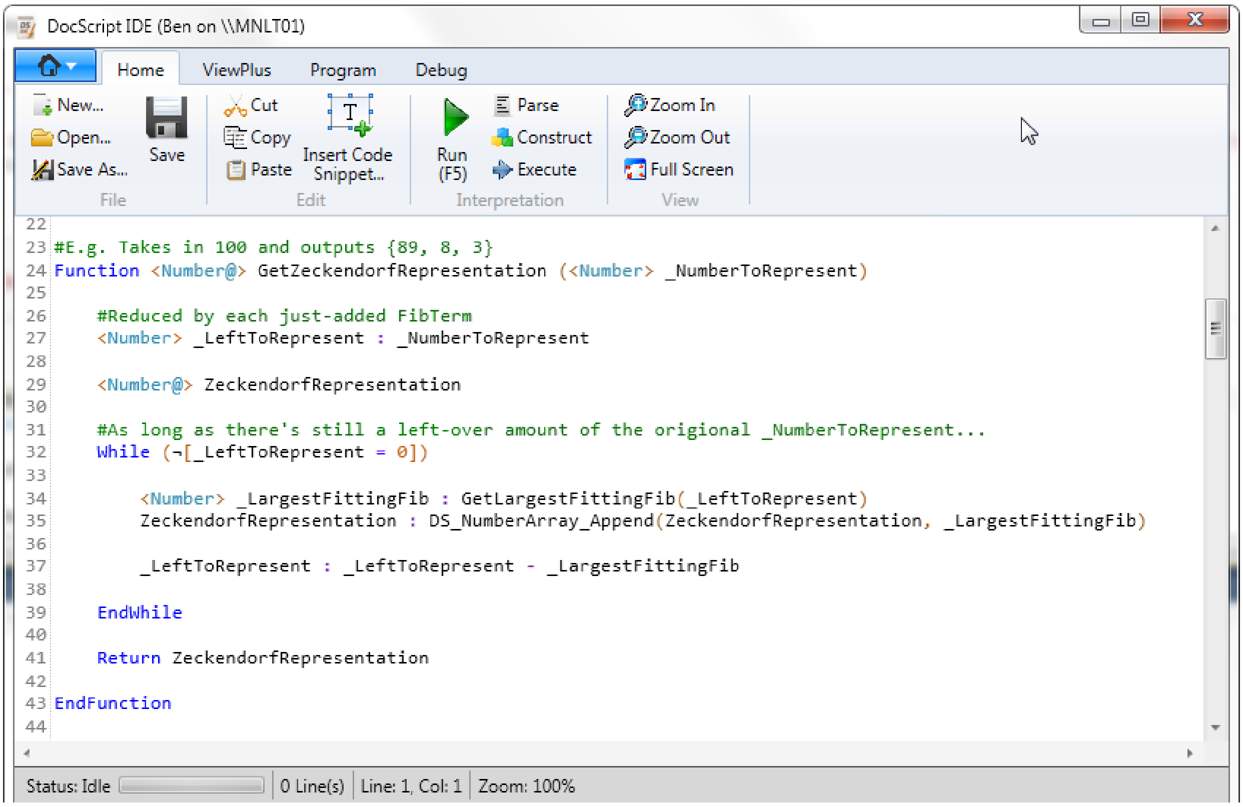
```
#Beep every 2 Seconds
function <void> main ()
        system_beep()
        ds_sleep(2000)
        main()
endfunction
```

I was getting quite *frustrated* and wasn't making progress, so reached the following conclusion:

> *The purpose of this project wasn't to implement a syntax-highlighting source editor from scratch; it was to build a Programming Language. I will struggle to rationalise dedicating so much time and energy to something that isn't meeting*

the initial goal directly. So: I don't need to ***reinvent the wheel****; I'm just going to use an open-source extensible RichTextBox Control for WPF, with Syntax-highlighting, Line Numbers, and indentation folding built-in;* ***AvalonEdit****. These features will better meet the Stakeholder requirements, and save me time!* ***It's a Win-Win.***

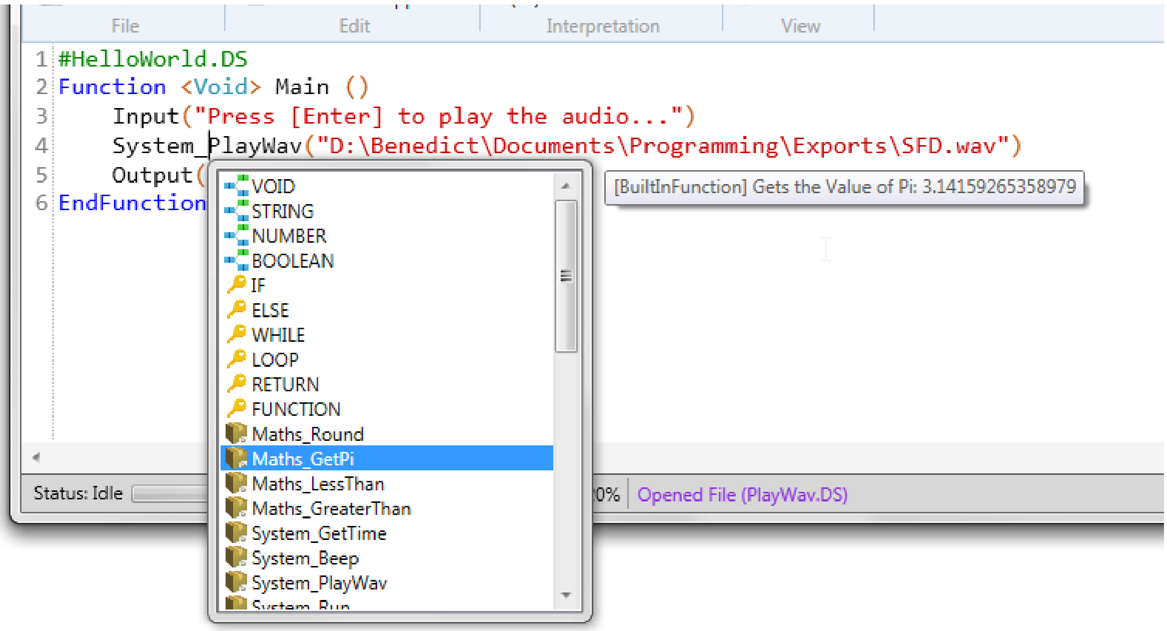
This new Text-Editing Control *(AvalonEdit's TextEditor)* looks like this:



```

#E.g. Takes in 100 and outputs {89, 8, 3}
Function <Number@> GetZeckendorfRepresentation (<Number> _NumberToRepresent)

    #Reduced by each just-added FibTerm
    <Number> _LeftToRepresent : _NumberToRepresent

    <Number@> ZeckendorfRepresentation

    #As long as there's still a left-over amount of the origional _NumberToRepresent...
    While (¬[_LeftToRepresent = 0])

        <Number> _LargestFittingFib : GetLargestFittingFib(_LeftToRepresent)
        ZeckendorfRepresentation : DS_NumberArray_Append(ZeckendorfRepresentation, _LargestFittingFib)

        _LeftToRepresent : _LeftToRepresent - _LargestFittingFib

    EndWhile

    Return ZeckendorfRepresentation

EndFunction

```

Status: Idle | 0 Line(s) | Line: 1, Col: 1 | Zoom: 100%

The IDE now has Line-Numbers, reliable syntax-highlighting, and even a ↓ mini-intellisense ↓...



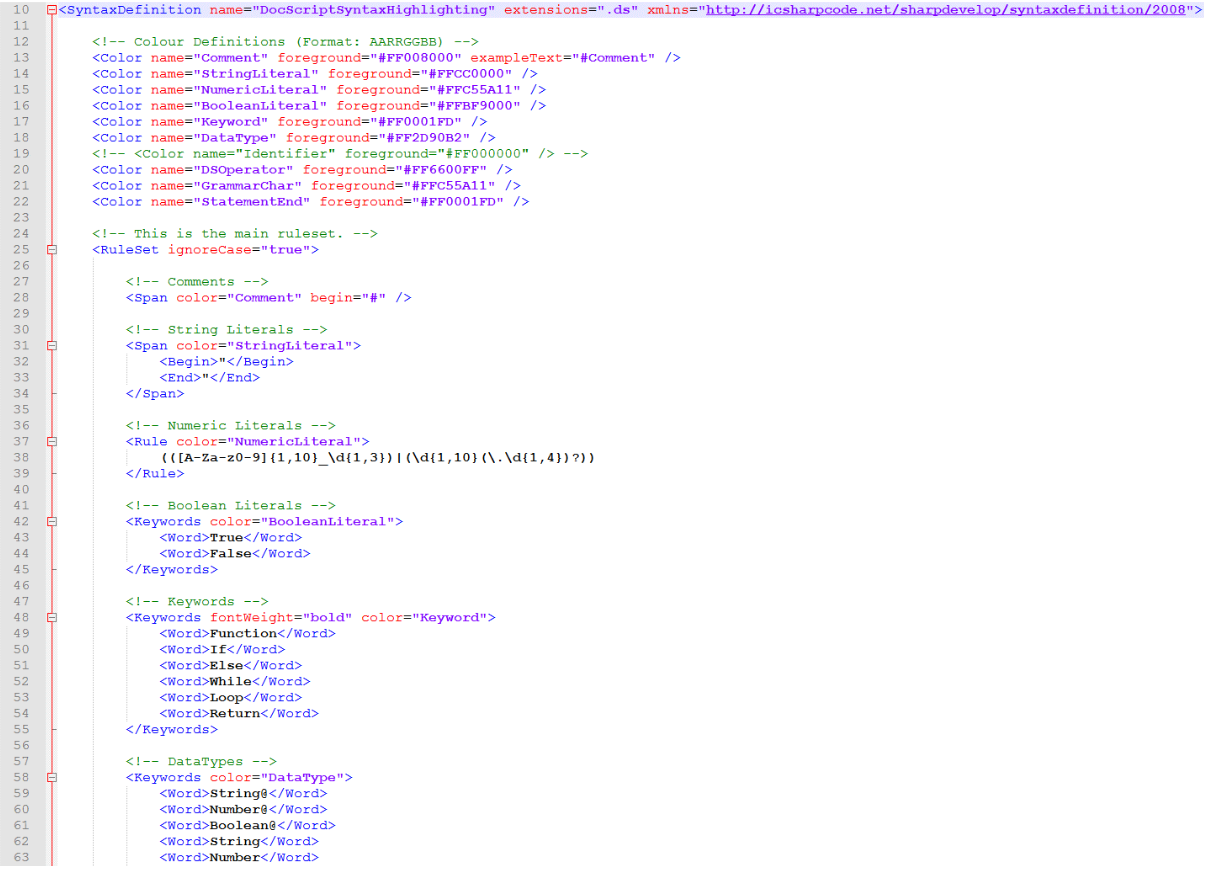
```
#HelloWorld.DS
Function <Void> Main ()
    Input("Press [Enter] to play the audio...")
    System_PlayWav("D:\Benedict\Documents\Programming\Exports\SFD.wav")
    Output(
EndFunction
```

[BuiltInFunction] Gets the Value of Pi: 3.14159265358979

Status: Idle

0% Opened File (PlayWav.DS)

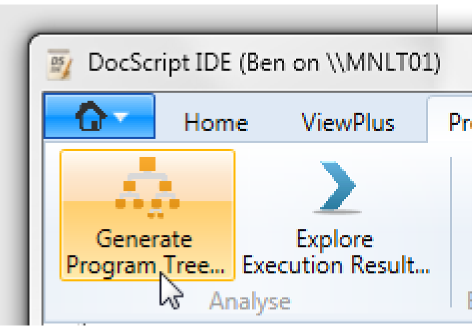
I implemented the syntax-highlighting through an XSHD (eXtensible Syntax Highlighting Definition) file, which looks like this:



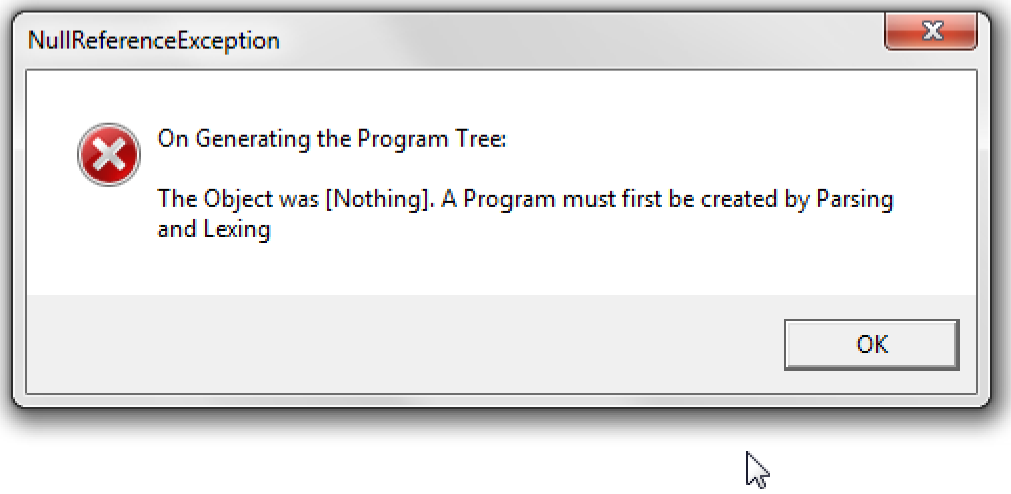
```
<SyntaxDefinition name="DocScriptSyntaxHighlighting" extensions=".ds" xmlns="http://icsharpcode.net/sharpdevelop/syntaxdefinition/2008">

    <!-- Colour Definitions (Format: AARRGGBB) -->
    <Color name="Comment" foreground="#FF008000" exampleText="#Comment" />
    <Color name="StringLiteral" foreground="#FFCC0000" />
    <Color name="NumericLiteral" foreground="#FFC55A11" />
    <Color name="BooleanLiteral" foreground="#FFBF9000" />
    <Color name="Keyword" foreground="#FF0001FD" />
    <Color name="DataType" foreground="#FF2D90B2" />
    <!-- <Color name="Identifier" foreground="#FF000000" /> -->
    <Color name="DSOperator" foreground="#FF6600FF" />
    <Color name="GrammarChar" foreground="#FFC55A11" />
    <Color name="StatementEnd" foreground="#FF0001FD" />

    <!-- This is the main ruleset. -->
    <RuleSet ignoreCase="true">

        <!-- Comments -->
        <Span color="Comment" begin="#" />

        <!-- String Literals -->
        <Span color="StringLiteral">
            <Begin>"</Begin>
            <End>"</End>
        </Span>

        <!-- Numeric Literals -->
        <Rule color="NumericLiteral">
            (([A-Za-z0-9]{1,10}_\d{1,3})|(\d{1,10}(\.\d{1,4})?))
        </Rule>

        <!-- Boolean Literals -->
        <Keywords color="BooleanLiteral">
            <Word>True</Word>
            <Word>False</Word>
        </Keywords>

        <!-- Keywords -->
        <Keywords fontWeight="bold" color="Keyword">
            <Word>Function</Word>
            <Word>If</Word>
            <Word>Else</Word>
            <Word>While</Word>
            <Word>Loop</Word>
            <Word>Return</Word>
        </Keywords>

        <!-- DataTypes -->
        <Keywords color="DataType">
            <Word>String@</Word>
            <Word>Number@</Word>
            <Word>Boolean@</Word>
            <Word>String</Word>
            <Word>Number</Word>
```

Null-Program Trees

Before a Program has been constructed during the lexing stage, the [Generate Program Tree] Button is still visible:



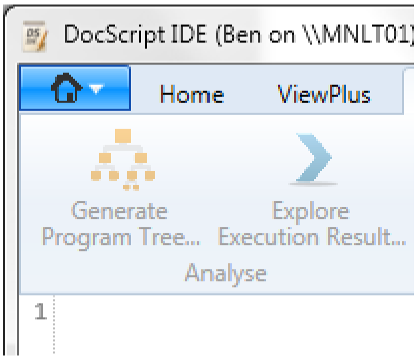
This means that when the button is clicked, an Exception is Thrown, informing the user that accessing the Cached-Program Object resulted in a `NullReferenceException`:


NullReferenceException

On Generating the Program Tree:

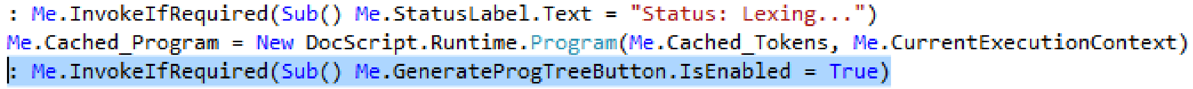
The Object was [Nothing]. A Program must first be created by Parsing and Lexing

Although this is admissible, it would be better to prevent this scenario from occurring, ***by design***. To this end, I shall alter the DSIDE program to disable the [Generate Program Tree] Button, until a Program has actually been cached:



← The Button is now disabled.

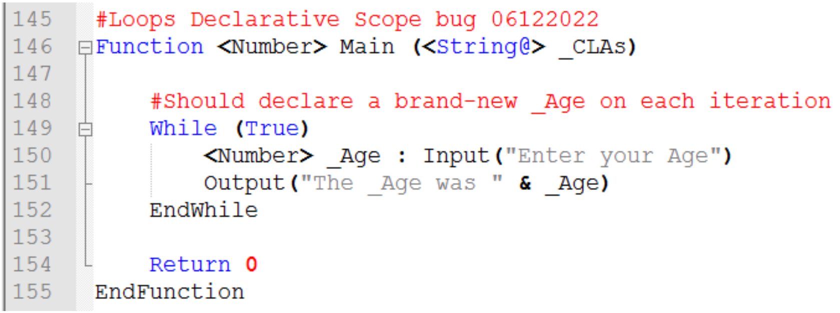
This is the line that enables the button, after a successful call to `Program.New()`:



```
: Me.InvokeIfRequired(Sub() Me.StatusLabel.Text = "Status: Lexing...")
Me.Cached_Program = New DocScript.Runtime.Program(Me.Cached_Tokens, Me.CurrentExecutionContext)
: Me.InvokeIfRequired(Sub() Me.GenerateProgTreeButton.IsEnabled = True)
```

While Loop Variable Scope

I then discovered another problem: In all programming languages I've ever used, each iteration of a while loop has its own **declarative scope**, in DocScript called a SymbolTable. This means that when running the program…



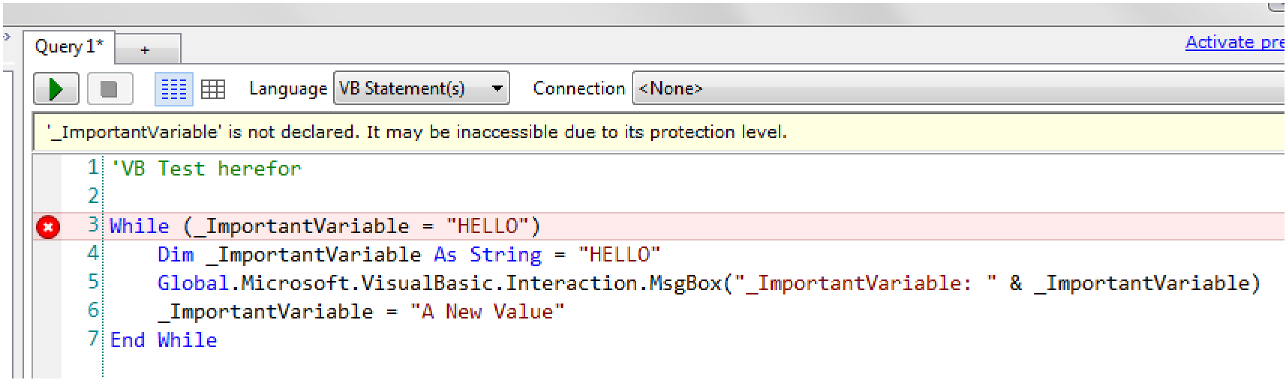
```
#Loops Declarative Scope bug 06122022
Function <Number> Main (<String@> _CLAs)

    #Should declare a brand-new _Age on each iteration
    While (True)
        <Number> _Age : Input("Enter your Age")
        Output("The _Age was " & _Age)
    EndWhile

    Return 0
EndFunction
```

…One would expect to infinitely be asked "Enter your Age", and for the _Age to be `Output()` again.

However, because I've never built a full procedural Programming Language before, I didn't actually cogently conceptualise that **each iteration of a while loop needs to start with a blank SymbolTable**. At the moment, a Statement-local SymbolTable is created at the start of the `WhileStatement`'s `Execute()` call. I need to add in a line to **reset** – as it were – this SymbolTable after each Iteration!

(…This also applies to the LoopStatement)

Embarrassingly, I had to remind myself of whether or not a Variable Declared inside a While Statement is visible from the expression of the While Loop, in normal Programming languages. To this end, I wrote a little test in Visual BASIC .NET:

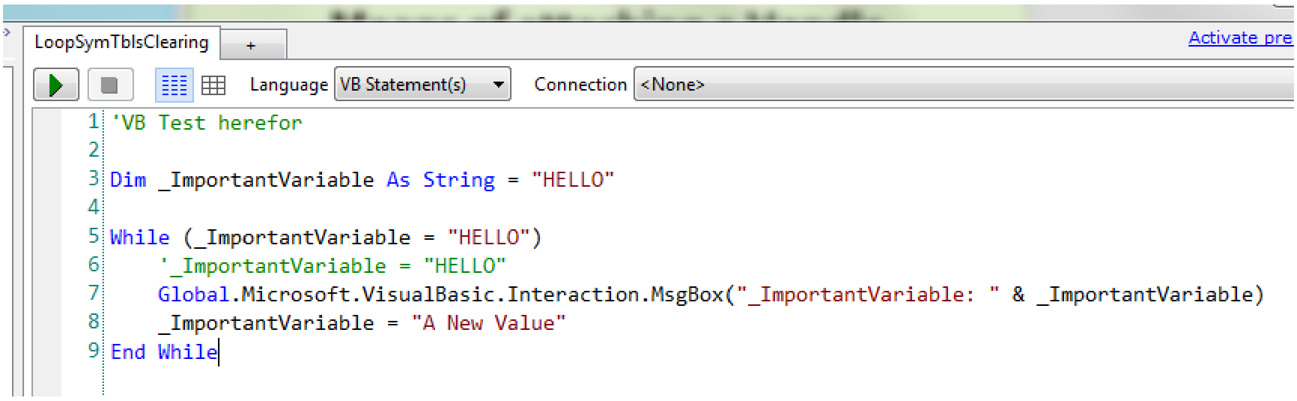


Language VB Statement(s) Connection <None>

'_ImportantVariable' is not declared. It may be inaccessible due to its protection level.

```
'VB Test herefor

While (_ImportantVariable = "HELLO")
    Dim _ImportantVariable As String = "HELLO"
    Global.Microsoft.VisualBasic.Interaction.MsgBox("_ImportantVariable: " & _ImportantVariable)
    _ImportantVariable = "A New Value"
End While
```

From this test, I am concluding that the SymbolTable… …Actually wait; I need to do one more test:



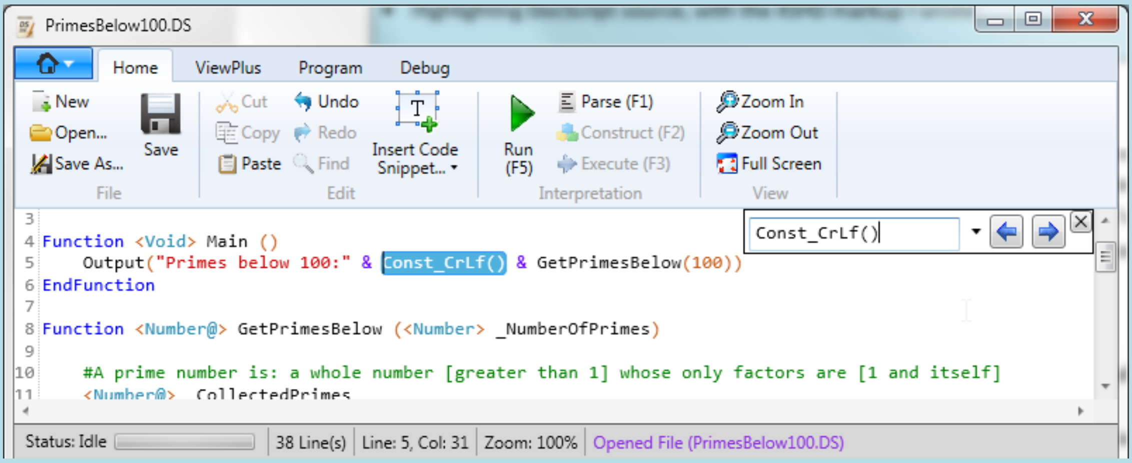
Language VB Statement(s) Connection <None>

```
'VB Test herefor

Dim _ImportantVariable As String = "HELLO"

While (_ImportantVariable = "HELLO")
    '_ImportantVariable = "HELLO"
    Global.Microsoft.VisualBasic.Interaction.MsgBox("_ImportantVariable: " & _ImportantVariable)
    _ImportantVariable = "A New Value"
End While
```

Okay; now I'm saying definitively that the WhileStatement (or IfStatement)'s local SymbolTable needs to be reset **after the condition is resolved** and **before the contents are executed**.
I implemented these changes accordingly in the DocScript Library DLL.

Prototype: This point marks a milestone in the product, which is now capable of...

- Highlighting DocScript source with the XSHD markup I wrote, and performing standard text-editing tasks including {New, Open, Save, Save-As, Cut, Copy, Paste, Undo, Redo, Find, Zoom-In/-Out}:

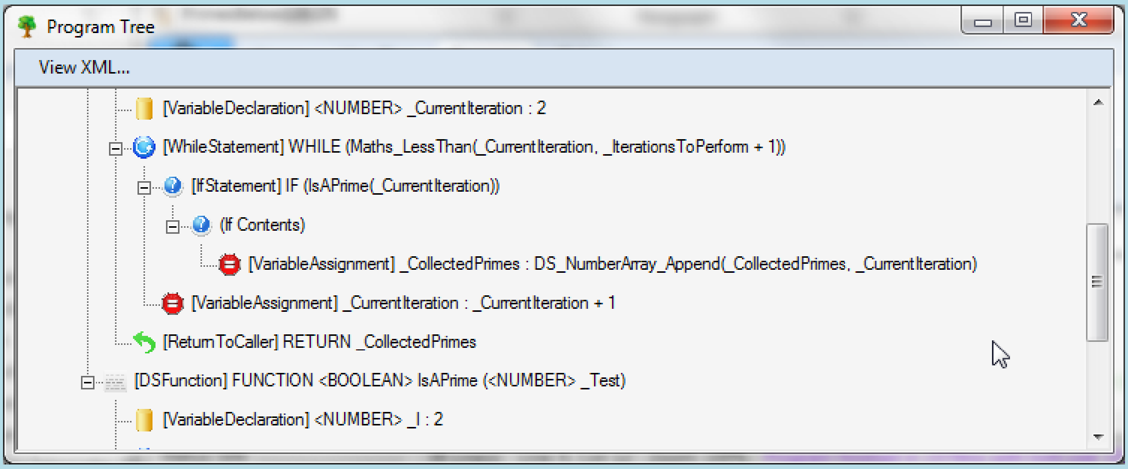


Const_CrLf()

```
Function <Void> Main ()
    Output("Primes below 100:" & Const_CrLf() & GetPrimesBelow(100))
EndFunction

Function <Number@> GetPrimesBelow (<Number> _NumberOfPrimes)

    #A prime number is: a whole number [greater than 1] whose only factors are [1 and itself]
    <Number@>  CollectedPrimes
```

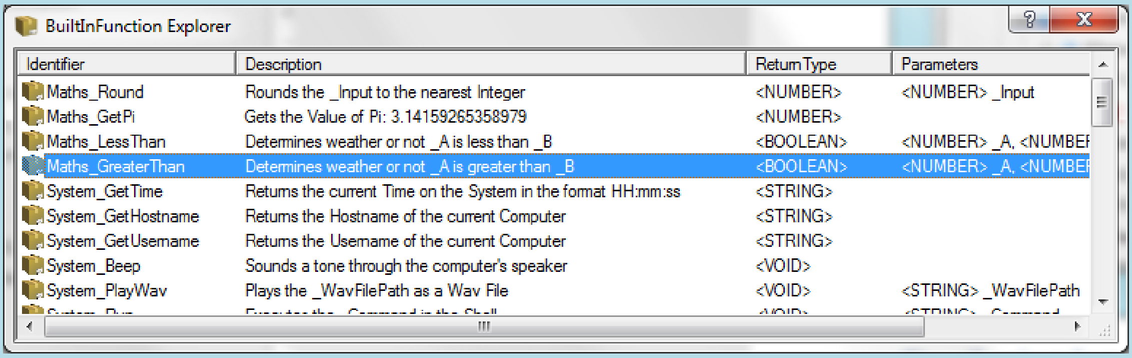
Status: Idle | 38 Line(s) | Line: 5, Col: 31 | Zoom: 100% | Opened File (PrimesBelow100.DS)

- Performing DocScript Program Analysis tasks, such as displaying a Program Tree



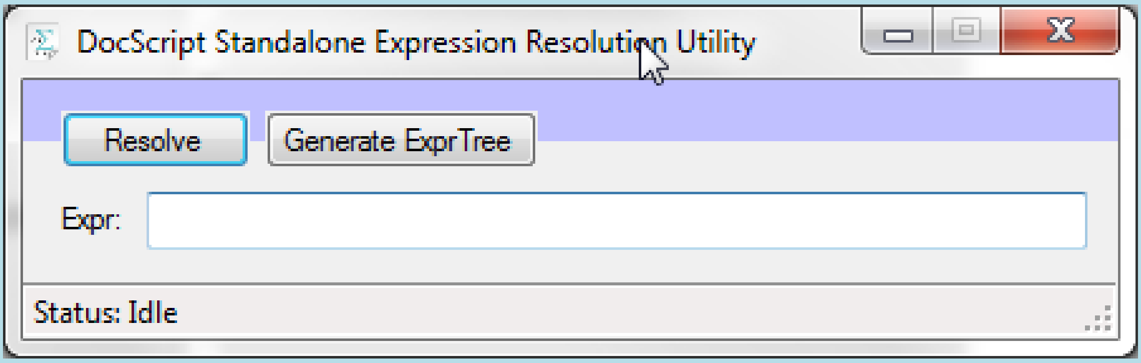
```
[VariableDeclaration] <NUMBER> _CurrentIteration : 2
[WhileStatement] WHILE (Maths_LessThan(_CurrentIteration, _IterationsToPerform + 1))
    [IfStatement] IF (IsAPrime(_CurrentIteration))
        (If Contents)
            [VariableAssignment] _CollectedPrimes : DS_NumberArray_Append(_CollectedPrimes, _CurrentIteration)
    [VariableAssignment] _CurrentIteration : _CurrentIteration + 1
[ReturnToCaller] RETURN _CollectedPrimes
[DSFunction] FUNCTION <BOOLEAN> IsAPrime (<NUMBER> _Test)
    [VariableDeclaration] <NUMBER> _I : 2
```

- Providing helpful development features, such as DocScript-Intellisense, and the BIF Explorer:



Identifier	Description	Return Type	Parameters
Maths_Round	Rounds the _Input to the nearest Integer	<NUMBER>	<NUMBER> _Input
Maths_GetPi	Gets the Value of Pi: 3.14159265358979	<NUMBER>	
Maths_LessThan	Determines weather or not _A is less than _B	<BOOLEAN>	<NUMBER> _A, <NUMBEF
Maths_GreaterThan	Determines weather or not _A is greater than _B	<BOOLEAN>	<NUMBER> _A, <NUMBEF
System_GetTime	Returns the current Time on the System in the format HH:mm:ss	<STRING>	
System_GetHostname	Returns the Hostname of the current Computer	<STRING>	
System_GetUsername	Returns the Username of the current Computer	<STRING>	
System_Beep	Sounds a tone through the computer's speaker	<VOID>	
System_PlayWav	Plays the _WavFilePath as a Wav File	<VOID>	<STRING> _WavFilePath
System_Run	Executes the _Command in the Shell	<VOID>	<STRING> _Command

- Being able to launch the DSExpr.exe utility, for easy standalone expression resolution:


DocScript Standalone Expression Resolution Utility

Status: Idle

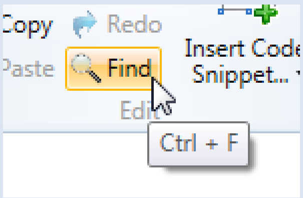
Progress Recap: Where am I in the development plan? *[Review]*

- **Done**: I have just written the Windows IDE implementation of DocScript.
- **Next**: I will implement this DLL into the third of the three implementations, as was described in detail within §Design. This is the web-based interpreter system.

Iterative Testing

During the §Analysis, I listed a number of types of test, along with example testing data, which I could use to test the product once implemented. I shall now make use of this testing plan…

Assert-style Test	☑ Passed?
Input and Output: Erroneous DocScript source can be taken in, but the interpreter determines that there is something wrong, instead of continuing and crashing.	Yes; see previous screenshots
Input and Output: Standalone Expressions can be taken in in the DSExpr.exe utility, and are evaluated using the current Symbol Tables.	Yes; see previous screenshots
Usability: Common tasks can be performed quickly via Keyboard shortcuts, and it is clear to discover which keyboard shortcuts are available in the product.	No; Although there are some keyboard shortcuts available for basic tasks such as Run (F5), most of the buttons still have to be clicked with a mouse. In addition, the Ribbon used in the WPF window *does* support Key Tips (the sort seen in MS Word when the [Alt] Key is pressed), but I have not made use of these. Therefore, I will now add a more comprehensive implementation of shortcuts for all common functions, including the launching of the Program Analysis dialogs (ExeRes Explorer, and ProgTree Visualiser). In addition – to enable easy discovery of which shortcuts are linked to which buttons, I have added ToolTips to the buttons, thusly:

	 Copy Redo Paste Find Edit Insert Code Snippet... Ctrl + F
All of the "**DocScript Source**" from the *Testing* section of the *§Design*, can be run successfully.	**Yes**; see previous screenshots. This included the "**Simplest-Possible DocScript Program**", which returned the Default Exit Code of 101, because there is no Exit-Code provided by the DocScript Program.

Justifications for Actions Taken: By adding a greater number of Keyboard Shortcuts, the application becomes easier to use, as these constitute a usability feature. In addition, by clearly and consistently labelling which Shortcuts can be used for each Button, the user is able to rapidly learn them.
Some of the shortcuts are – I feel – so important, that they ought to be even more overtly shown to the user. For example, the [Run (F5)] Button has its shortcut hard-coded onto the Button Text, because this has to be considered the most-clicked button of the entire application, so it can therefore save the most time by having a Shortcut.

[Stage 4]
DocScript Interactive (**DSI**) Development

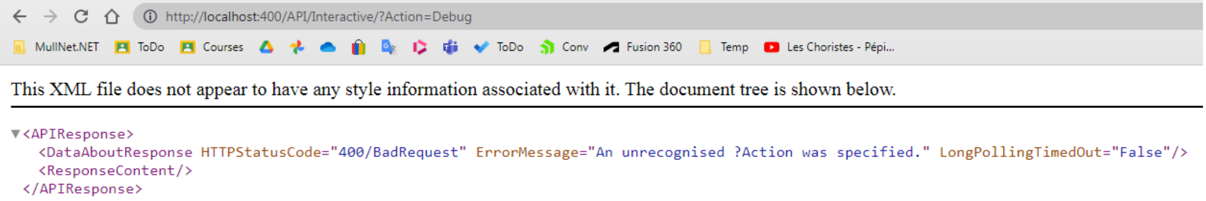
Interactive, is the DocScript component which can host real-time, multi-client execution sessions for a DS Program. Several clients can *tune-in* to the execution session, with each client being able to see outputs and LogEvents. When Input() is required by the DocScript Program, *all* clients have the chance to provide an input response, and the first client to do so, has their response accepted by the session.

Writing the API

The Server-side API will use XML, and will follow the API-Specification which I delineated in the §Design.

Explanation & Justification: XML is a commonplace and standardised serialisation format, and befits this project far better than JSON or CSV/TSV. In addition, XML-Literals are built-in to the Visual BASIC .NET Programming language, in which I am writing the server-side API. This makes it quick and terse and efficient to prepare API responses. JSON is designed to represent Key-Value-Pairs, and does not support Namespaces, Schemas, Comments, or tree-based data representation; it would be a poor choice for any project, but especially one that necessitates the analysis of individual layers of the development stack (e.g. exploring raw, plaintext API responses), such as DocScript.

Here is the style of API-Response I have decoded to use:

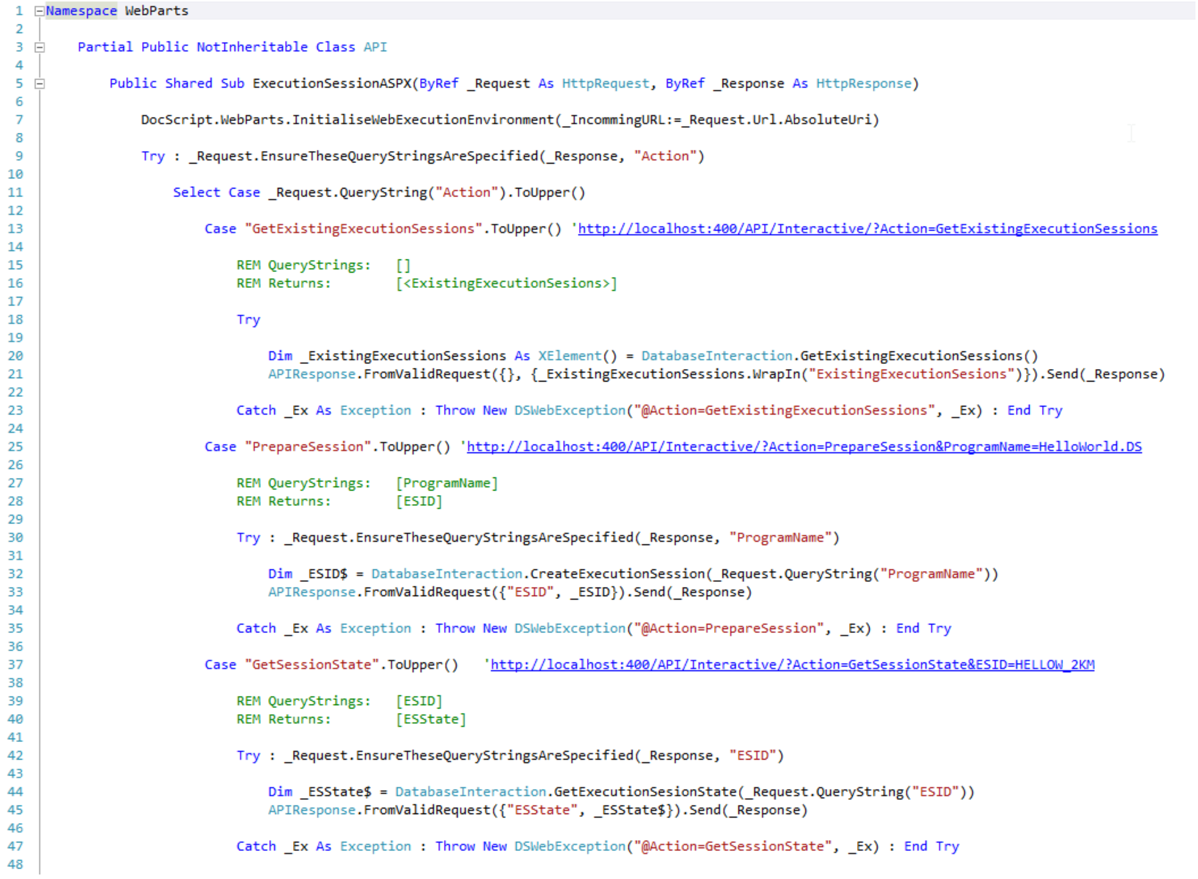


This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<APIResponse>
   <DataAboutResponse HTTPStatusCode="400/BadRequest" ErrorMessage="An unrecognised ?Action was specified." LongPollingTimedOut="False"/>
   <ResponseContent/>
 </APIResponse>
```

↑ **Explanation & Justification:** This API schema provides the HTTP status code not only in the HTTP response headers, but also as PlainText in the XML. This helps with debugging!

At this point, it is just a matter of following the API specification from §Design, and implementing each of the EndPoints…



```
Namespace WebParts

    Partial Public NotInheritable Class API

        Public Shared Sub ExecutionSessionASPX(ByRef _Request As HttpRequest, ByRef _Response As HttpResponse)

            DocScript.WebParts.InitialiseWebExecutionEnvironment(_IncommingURL:=_Request.Url.AbsoluteUri)

            Try : _Request.EnsureTheseQueryStringsAreSpecified(_Response, "Action")

                Select Case _Request.QueryString("Action").ToUpper()

                    Case "GetExistingExecutionSessions".ToUpper() 'http://localhost:400/API/Interactive/?Action=GetExistingExecutionSessions

                        REM QueryStrings:   []
                        REM Returns:        [<ExistingExecutionSesions>]

                        Try

                            Dim _ExistingExecutionSessions As XElement() = DatabaseInteraction.GetExistingExecutionSessions()
                            APIResponse.FromValidRequest({}, {_ExistingExecutionSessions.WrapIn("ExistingExecutionSesions")}).Send(_Response)

                        Catch _Ex As Exception : Throw New DSWebException("@Action=GetExistingExecutionSessions", _Ex) : End Try

                    Case "PrepareSession".ToUpper() 'http://localhost:400/API/Interactive/?Action=PrepareSession&ProgramName=HelloWorld.DS

                        REM QueryStrings:   [ProgramName]
                        REM Returns:        [ESID]

                        Try : _Request.EnsureTheseQueryStringsAreSpecified(_Response, "ProgramName")

                            Dim _ESID$ = DatabaseInteraction.CreateExecutionSession(_Request.QueryString("ProgramName"))
                            APIResponse.FromValidRequest({"ESID", _ESID}).Send(_Response)

                        Catch _Ex As Exception : Throw New DSWebException("@Action=PrepareSession", _Ex) : End Try

                    Case "GetSessionState".ToUpper()   'http://localhost:400/API/Interactive/?Action=GetSessionState&ESID=HELLOW_2KM

                        REM QueryStrings:   [ESID]
                        REM Returns:        [ESState]

                        Try : _Request.EnsureTheseQueryStringsAreSpecified(_Response, "ESID")

                            Dim _ESState$ = DatabaseInteraction.GetExecutionSesionState(_Request.QueryString("ESID"))
                            APIResponse.FromValidRequest({"ESState", _ESState$}).Send(_Response)

                        Catch _Ex As Exception : Throw New DSWebException("@Action=GetSessionState", _Ex) : End Try

```

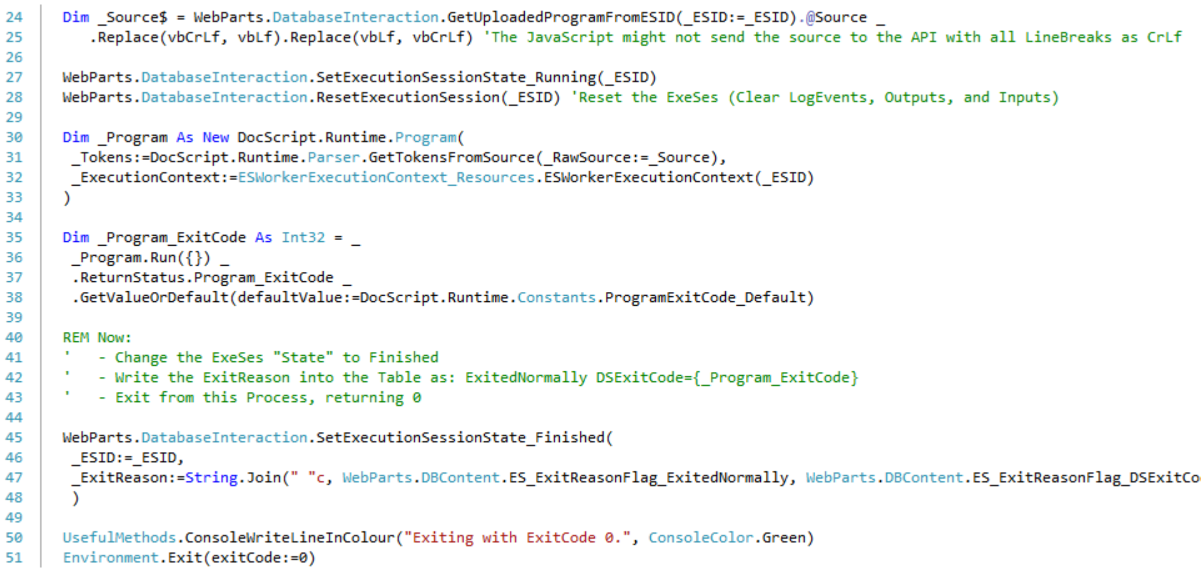
↑ These are the first 3 of **15 EndPoints** in the DocScript Interactive API.

Validation: The following features ensure that API requests are valid…

- Required URL QueryStrings are checked-for with my `EnsureTheseQueryStringsAreSpecified()` Function.
- The Syntax for the XML Response is made uniform and consistent, by routing the response data through an instance of the `APIResponse` Class, and then calling `.Send()` thereon.
- QueryString Arguments such as the `?ESID` are validated against regular-expressions, so that no peculiar characters make it past the API and into deeper parts of the Back-End such as the DataBase.

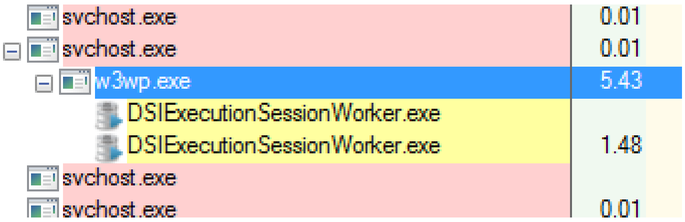
Implementing the DocScript DLL

This call to logic in the DLL occurs within the DSIExecutionSessionWorker assembly.



```
Dim _Source$ = WebParts.DatabaseInteraction.GetUploadedProgramFromESID(_ESID:=_ESID).@Source _
   .Replace(vbCrLf, vbLf).Replace(vbLf, vbCrLf) 'The JavaScript might not send the source to the API with all LineBreaks as CrLf

WebParts.DatabaseInteraction.SetExecutionSessionState_Running(_ESID)
WebParts.DatabaseInteraction.ResetExecutionSession(_ESID) 'Reset the ExeSes (Clear LogEvents, Outputs, and Inputs)

Dim _Program As New DocScript.Runtime.Program(
 _Tokens:=DocScript.Runtime.Parser.GetTokensFromSource(_RawSource:=_Source),
 _ExecutionContext:=ESWorkerExecutionContext_Resources.ESWorkerExecutionContext(_ESID)
)

Dim _Program_ExitCode As Int32 = _
 _Program.Run({}) _
 .ReturnStatus.Program_ExitCode _
 .GetValueOrDefault(defaultValue:=DocScript.Runtime.Constants.ProgramExitCode_Default)

REM Now:
'   - Change the ExeSes "State" to Finished
'   - Write the ExitReason into the Table as: ExitedNormally DSExitCode={_Program_ExitCode}
'   - Exit from this Process, returning 0

WebParts.DatabaseInteraction.SetExecutionSessionState_Finished(
 _ESID:=_ESID,
 _ExitReason:=String.Join(" "c, WebParts.DBContent.ES_ExitReasonFlag_ExitedNormally, WebParts.DBContent.ES_ExitReasonFlag_DSExitCo
 )

UsefulMethods.ConsoleWriteLineInColour("Exiting with ExitCode 0.", ConsoleColor.Green)
Environment.Exit(exitCode:=0)
```

This assembly is the DSIExecutionSessionWorker.exe file, which is instanciated by the Server-Side when the `InitiateSession` API-EndPoint is called. The Server then spins-up an ESWorker executable, like this:



```
   svchost.exe                          0.01
⊟  svchost.exe                          0.01
  ⊟  w3wp.exe                           5.43
        DSIExecutionSessionWorker.exe
        DSIExecutionSessionWorker.exe   1.48
   svchost.exe
   svchost.exe                          0.01
```

← One ESWorker per DSI-Session

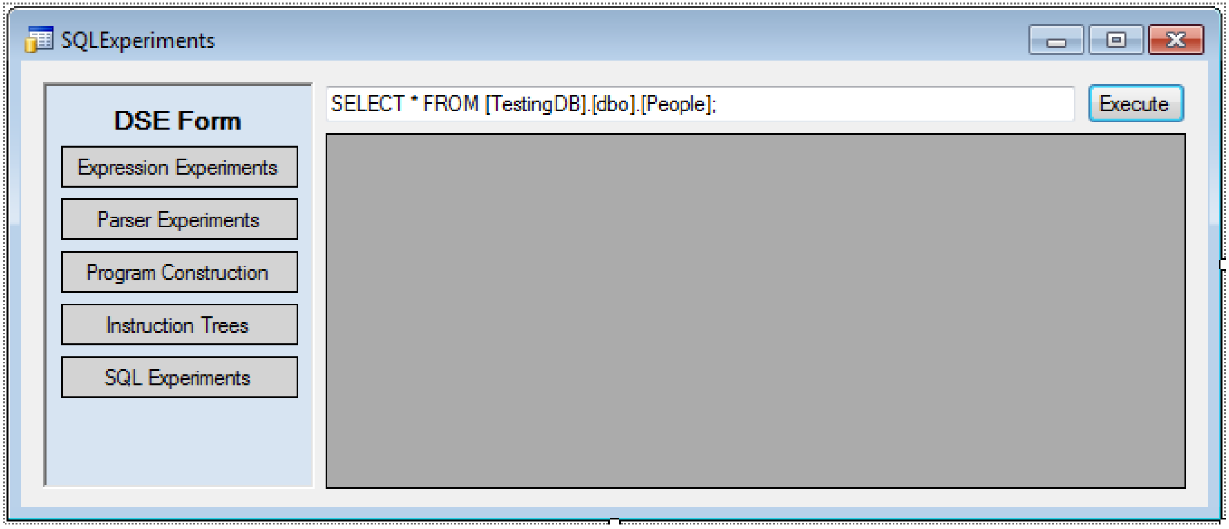
In the command-line arguments to an ESWorker, the ESID is passed in the form `/ESID:"*"`. This uses the same CLA-Manager which I wrote for DSCLI earlier.

Prototype: This point marks a milestone in the product, which is now capable of…

- Accepting a call to `/API/Interactive/?Action=InitiateSession&ESID=*` to start an Execution Session being hosted on the DSI Server.
- Instantiating a new ESWorker Executable as a child-process of the IIS Worker Process `w3wp.exe`.
- Responding to the client requesting the initiation, with a well-formed API Response and HTTP status code.

SQL Server Interaction

I wrote a Class called SQLQueryRunner, to simplify my use of SQL. It is called like this:

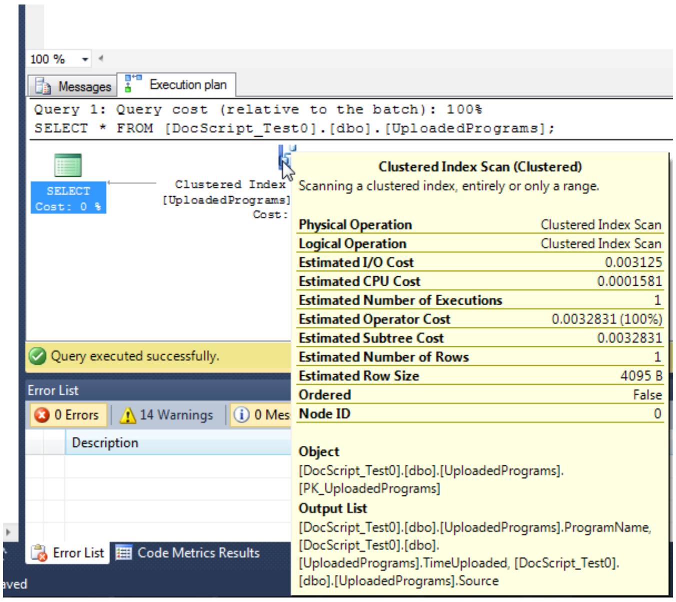


SELECT * FROM [TestingDB].[dbo].[People];

...With the Code-Behind...

```
Private Sub ExecQueryButton_Click() Handles ExecQueryButton.Click

    'Log via the Default LogWindow
    DocScript.Logging.LogUtilities.CurrentLogEventHandler = DocScript.Logging.BuiltInLogEv

    'Instanciat the SQL Runner
    Dim _SQLQueryRunner As New DocScript.Utilities.SQLQueryRunner("MNLT01\SQLEXPRESS")
    _SQLQueryRunner.ExecuteQuery(Me.SQLQueryTextBox.Text)

    'Copy the SQL Query output to the DataGridView on the Form
    Me.SQLOutput_DataGridView.DataSource = _SQLQueryRunner.QueryOutputDataTable

End Sub
```

This is the Execution Plan for the SQL SELECT Statement I am using:



```
Query 1: Query cost (relative to the batch): 100%
SELECT * FROM [DocScript_Test0].[dbo].[UploadedPrograms];
```

Clustered Index Scan (Clustered)

Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Number of Executions	1
Estimated Operator Cost	0.0032831 (100%)
Estimated Subtree Cost	0.0032831
Estimated Number of Rows	1
Estimated Row Size	4095 B
Ordered	False
Node ID	0

Object

[DocScript_Test0].[dbo].[UploadedPrograms].[PK_UploadedPrograms]

Output List

[DocScript_Test0].[dbo].[UploadedPrograms].ProgramName, [DocScript_Test0].[dbo].[UploadedPrograms].TimeUploaded, [DocScript_Test0].[dbo].[UploadedPrograms].Source

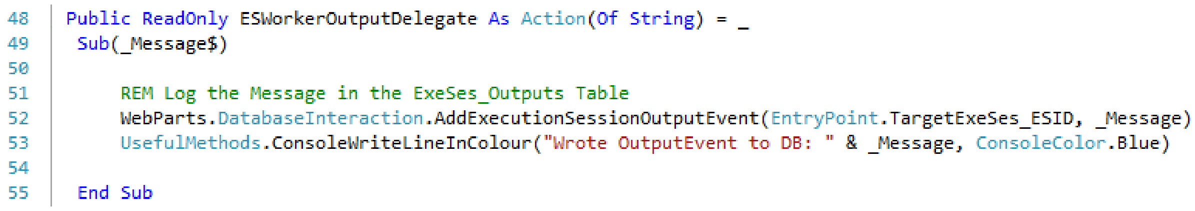
Query executed successfully.

Prototype: This point marks a milestone in the product, which is now capable of…

- Dependably and reliably running an SQL Query on an SQL Server (for the testing, this was simply a copy of SQL Server Express 2008 R2 on my development workstation)
- Retrieving the results from the database in a `DataTable`, and outputting these onto a user-interface.
- Completing all of this in under a millisecond (*Thanks x86!*)

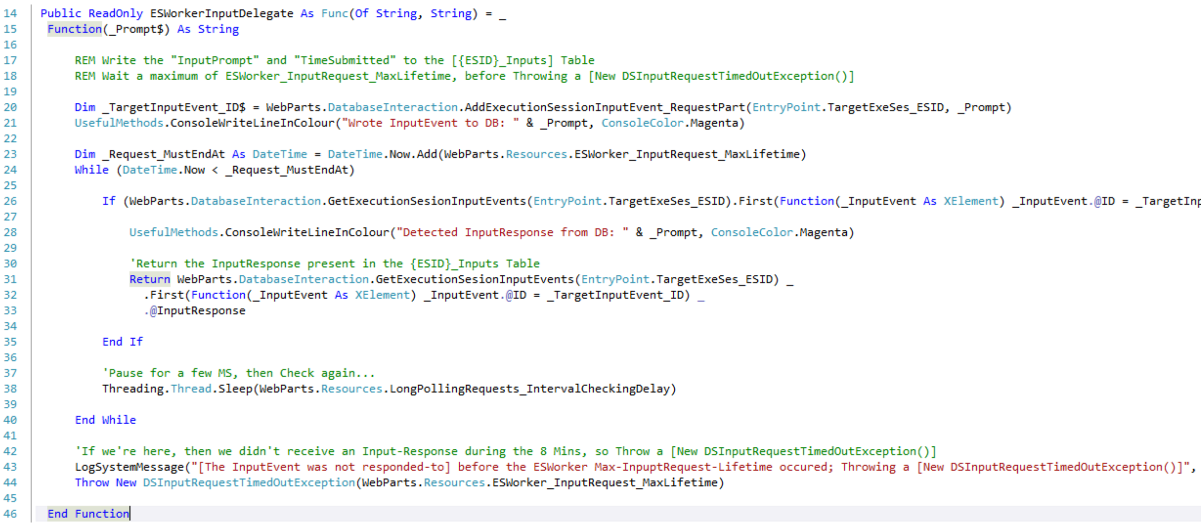
The **DSIExecutionSessionWorker.exe** program implements this same `SQLQueryRunner` Class, and uses it to write Output-Events to the Database, and also to write Input-Prompt to the Database. It then waits a finite length of time for an Input-Response to be inserted into the database (submitted by a DSI Client via `/API/Interactive/?Action=ProvideInputResponse`), and then reads this response value, and uses it for the continued execution of the DocScript program.

Here is how that's implemented:



```
Public ReadOnly ESWorkerOutputDelegate As Action(Of String) = _
 Sub(_Message$)

     REM Log the Message in the ExeSes_Outputs Table
     WebParts.DatabaseInteraction.AddExecutionSessionOutputEvent(EntryPoint.TargetExeSes_ESID, _Message)
     UsefulMethods.ConsoleWriteLineInColour("Wrote OutputEvent to DB: " & _Message, ConsoleColor.Blue)

 End Sub
```

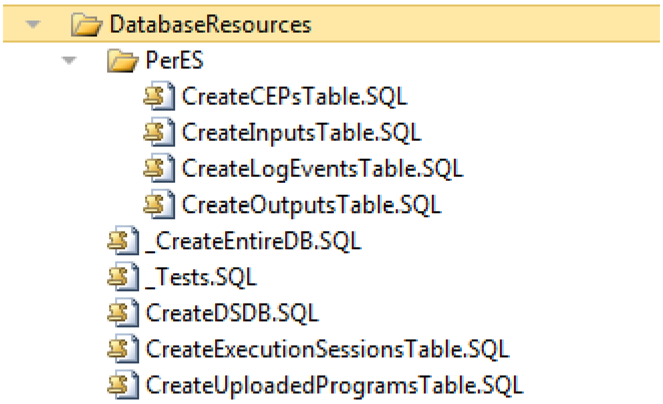
…And…



```
Public ReadOnly ESWorkerInputDelegate As Func(Of String, String) = _
 Function(_Prompt$) As String

    REM Write the "InputPrompt" and "TimeSubmitted" to the [{ESID}_Inputs] Table
    REM Wait a maximum of ESWorker_InputRequest_MaxLifetime, before Throwing a [New DSInputRequestTimedOutException()]

    Dim _TargetInputEvent_ID$ = WebParts.DatabaseInteraction.AddExecutionSessionInputEvent_RequestPart(EntryPoint.TargetExeSes_ESID, _Prompt)
    UsefulMethods.ConsoleWriteLineInColour("Wrote InputEvent to DB: " & _Prompt, ConsoleColor.Magenta)

    Dim _Request_MustEndAt As DateTime = DateTime.Now.Add(WebParts.Resources.ESWorker_InputRequest_MaxLifetime)
    While (DateTime.Now < _Request_MustEndAt)

        If (WebParts.DatabaseInteraction.GetExecutionSesionInputEvents(EntryPoint.TargetExeSes_ESID).First(Function(_InputEvent As XElement) _InputEvent.@ID = _TargetInp

            UsefulMethods.ConsoleWriteLineInColour("Detected InputResponse from DB: " & _Prompt, ConsoleColor.Magenta)

            'Return the InputResponse present in the {ESID}_Inputs Table
            Return WebParts.DatabaseInteraction.GetExecutionSesionInputEvents(EntryPoint.TargetExeSes_ESID) _
              .First(Function(_InputEvent As XElement) _InputEvent.@ID = _TargetInputEvent_ID) _
              .@InputResponse

        End If

        'Pause for a few MS, then Check again...
        Threading.Thread.Sleep(WebParts.Resources.LongPollingRequests_IntervalCheckingDelay)

    End While

    'If we're here, then we didn't receive an Input-Response during the 8 Mins, so Throw a [New DSInputRequestTimedOutException()]
    LogSystemMessage("[The InputEvent was not responded-to] before the ESWorker Max-InpuptRequest-Lifetime occured; Throwing a [New DSInputRequestTimedOutException()]",
    Throw New DSInputRequestTimedOutException(WebParts.Resources.ESWorker_InputRequest_MaxLifetime)

 End Function
```

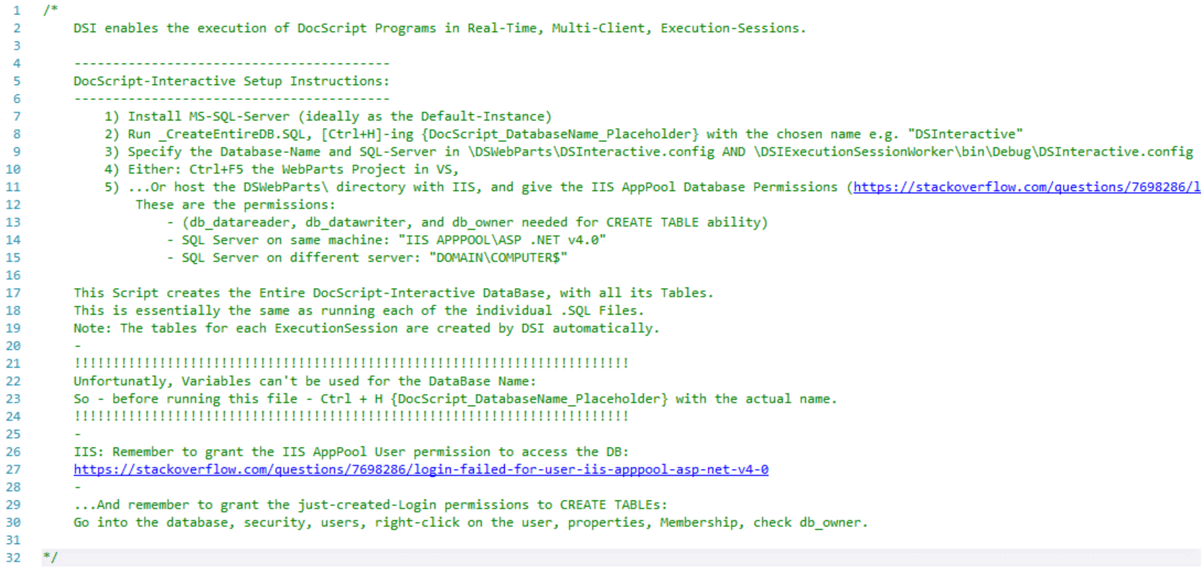
Database Structure

I wrote these SQL files to build the DocScript Interactive Database…

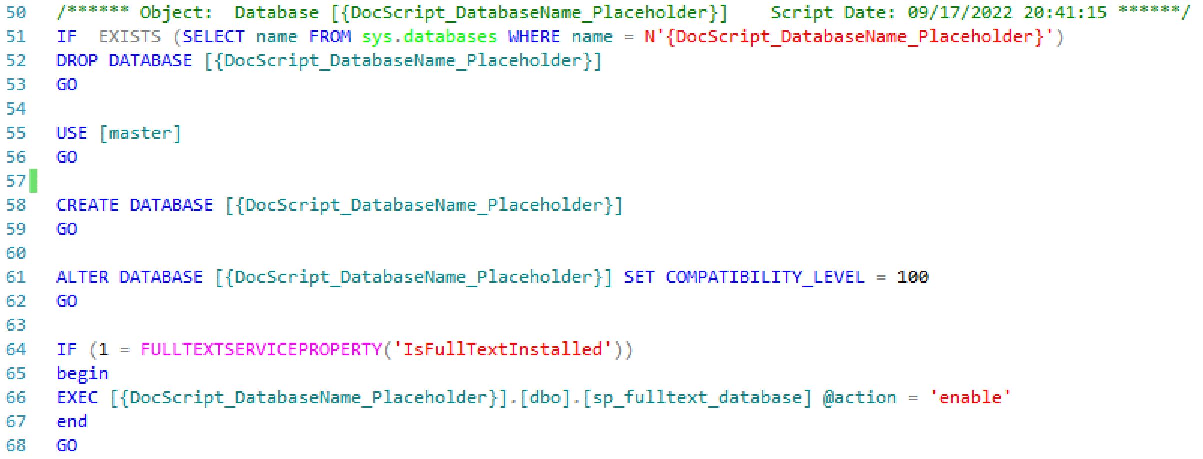


```
DatabaseResources
    PerES
        CreateCEPsTable.SQL
        CreateInputsTable.SQL
        CreateLogEventsTable.SQL
        CreateOutputsTable.SQL
    _CreateEntireDB.SQL
    _Tests.SQL
    CreateDSDB.SQL
    CreateExecutionSessionsTable.SQL
    CreateUploadedProgramsTable.SQL
```

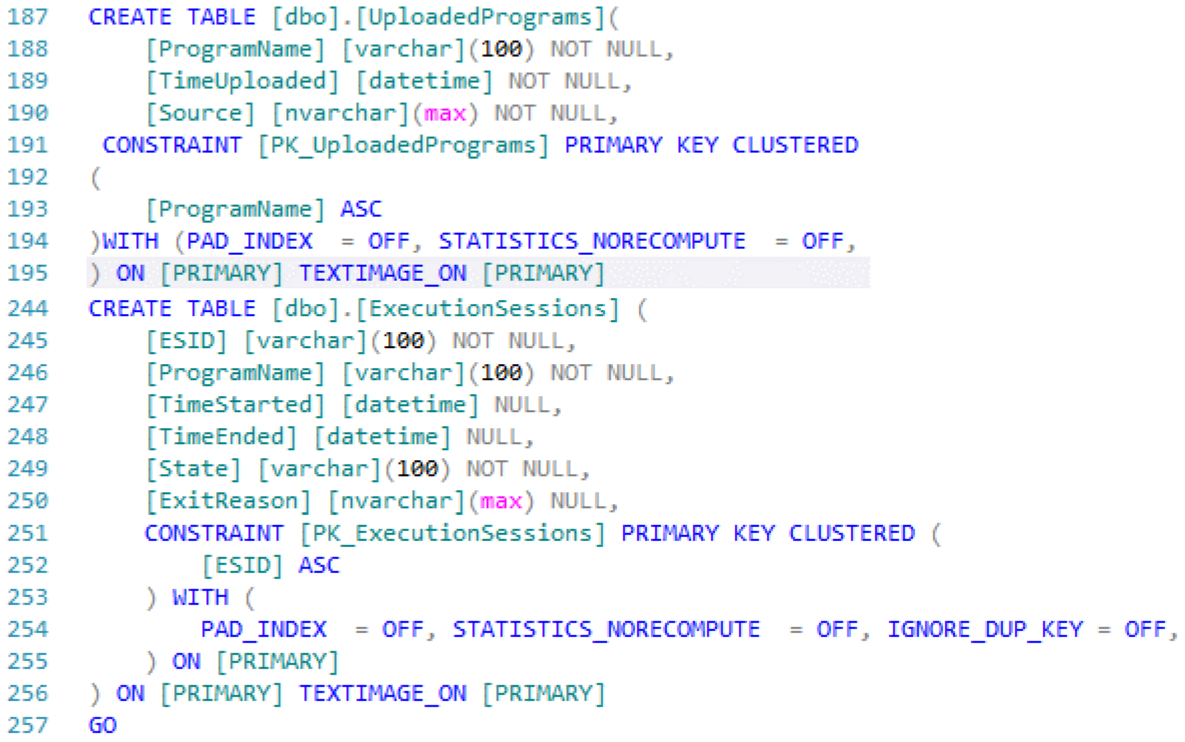
The `_CreateEntireDB.SQL` file automatically creates all the required (initial) tables, and the database. It also contains instructions for DSI Setup:



```sql
/*
    DSI enables the execution of DocScript Programs in Real-Time, Multi-Client, Execution-Sessions.

    ----------------------------------------
    DocScript-Interactive Setup Instructions:
    ----------------------------------------
        1) Install MS-SQL-Server (ideally as the Default-Instance)
        2) Run _CreateEntireDB.SQL, [Ctrl+H]-ing {DocScript_DatabaseName_Placeholder} with the chosen name e.g. "DSInteractive"
        3) Specify the Database-Name and SQL-Server in \DSWebParts\DSInteractive.config AND \DSIExecutionSessionWorker\bin\Debug\DSInteractive.config
        4) Either: Ctrl+F5 the WebParts Project in VS,
        5) ...Or host the DSWebParts\ directory with IIS, and give the IIS AppPool Database Permissions (https://stackoverflow.com/questions/7698286/l
            These are the permissions:
                - (db_datareader, db_datawriter, and db_owner needed for CREATE TABLE ability)
                - SQL Server on same machine: "IIS APPPOOL\ASP .NET v4.0"
                - SQL Server on different server: "DOMAIN\COMPUTER$"

    This Script creates the Entire DocScript-Interactive DataBase, with all its Tables.
    This is essentially the same as running each of the individual .SQL Files.
    Note: The tables for each ExecutionSession are created by DSI automatically.
    -
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    Unfortunatly, Variables can't be used for the DataBase Name:
    So - before running this file - Ctrl + H {DocScript_DatabaseName_Placeholder} with the actual name.
    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    -
    IIS: Remember to grant the IIS AppPool User permission to access the DB:
    https://stackoverflow.com/questions/7698286/login-failed-for-user-iis-apppool-asp-net-v4-0
    -
    ...And remember to grant the just-created-Login permissions to CREATE TABLEs:
    Go into the database, security, users, right-click on the user, properties, Membership, check db_owner.

*/
```



```sql
/****** Object:  Database [{DocScript_DatabaseName_Placeholder}]    Script Date: 09/17/2022 20:41:15 ******/
IF  EXISTS (SELECT name FROM sys.databases WHERE name = N'{DocScript_DatabaseName_Placeholder}')
DROP DATABASE [{DocScript_DatabaseName_Placeholder}]
GO

USE [master]
GO

CREATE DATABASE [{DocScript_DatabaseName_Placeholder}]
GO

ALTER DATABASE [{DocScript_DatabaseName_Placeholder}] SET COMPATIBILITY_LEVEL = 100
GO

IF (1 = FULLTEXTSERVICEPROPERTY('IsFullTextInstalled'))
begin
EXEC [{DocScript_DatabaseName_Placeholder}].[dbo].[sp_fulltext_database] @action = 'enable'
end
GO
```

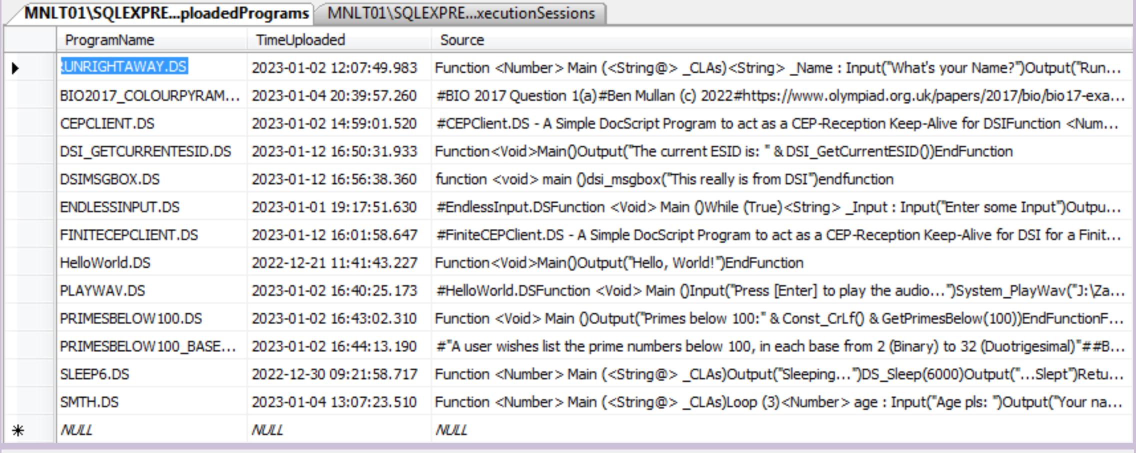


```
CREATE TABLE [dbo].[UploadedPrograms](
    [ProgramName] [varchar](100) NOT NULL,
    [TimeUploaded] [datetime] NOT NULL,
    [Source] [nvarchar](max) NOT NULL,
 CONSTRAINT [PK_UploadedPrograms] PRIMARY KEY CLUSTERED
(
    [ProgramName] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
CREATE TABLE [dbo].[ExecutionSessions] (
    [ESID] [varchar](100) NOT NULL,
    [ProgramName] [varchar](100) NOT NULL,
    [TimeStarted] [datetime] NULL,
    [TimeEnded] [datetime] NULL,
    [State] [varchar](100) NOT NULL,
    [ExitReason] [nvarchar](max) NULL,
    CONSTRAINT [PK_ExecutionSessions] PRIMARY KEY CLUSTERED (
        [ESID] ASC
    ) WITH (
        PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
    ) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
```

[Above] _CreateEntireDB.SQL

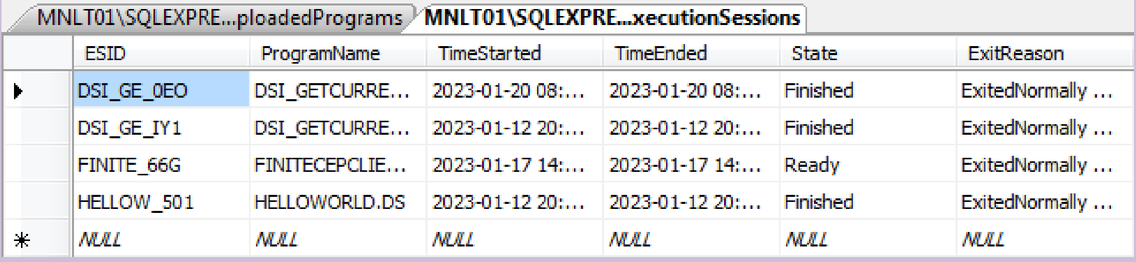
Structure and Modulatory: This DSI Database is well-structured, because…

- The tables comply with 1-NF and 2-NF standards; the `ProgramName` of the UploadedPrograms Table appears as a foreign key in the ExecutionSessions Table:



MNLT01\SQLEXPRE...ploadedPrograms
MNLT01\SQLEXPRE...xecutionSessions

	ProgramName	TimeUploaded	Source
▶	:UNRIGHTAWAY.DS	2023-01-02 12:07:49.983	Function <Number> Main (<String@> _CLAs)<String> _Name : Input("What's your Name?")Output("Run…
	BIO2017_COLOURPYRAM…	2023-01-04 20:39:57.260	#BIO 2017 Question 1(a)#Ben Mullan (c) 2022#https://www.olympiad.org.uk/papers/2017/bio/bio17-exa…
	CEPCLIENT.DS	2023-01-02 14:59:01.520	#CEPClient.DS - A Simple DocScript Program to act as a CEP-Reception Keep-Alive for DSIFunction <Num…
	DSI_GETCURRENTESID.DS	2023-01-12 16:50:31.933	Function<Void>Main()Output("The current ESID is: " & DSI_GetCurrentESID())EndFunction
	DSIMSGBOX.DS	2023-01-12 16:56:38.360	function <void> main ()dsi_msgbox("This really is from DSI")endfunction
	ENDLESSINPUT.DS	2023-01-01 19:17:51.630	#EndlessInput.DSFunction <Void> Main ()While (True)<String> _Input : Input("Enter some Input")Outpu…
	FINITECEPCLIENT.DS	2023-01-12 16:01:58.647	#FiniteCEPClient.DS - A Simple DocScript Program to act as a CEP-Reception Keep-Alive for DSI for a Finit…
	HelloWorld.DS	2022-12-21 11:41:43.227	Function<Void>Main()Output("Hello, World!")EndFunction
	PLAYWAV.DS	2023-01-02 16:40:25.173	#HelloWorld.DSFunction <Void> Main ()Input("Press [Enter] to play the audio…")System_PlayWav("J:\Za…
	PRIMESBELOW100.DS	2023-01-02 16:43:02.310	Function <Void> Main ()Output("Primes below 100:" & Const_CrLf() & GetPrimesBelow(100))EndFunctionF…
	PRIMESBELOW100_BASE…	2023-01-02 16:44:13.190	#"A user wishes list the prime numbers below 100, in each base from 2 (Binary) to 32 (Duotrigesimal)"##B…
	SLEEP6.DS	2022-12-30 09:21:58.717	Function <Number> Main (<String@> _CLAs)Output("Sleeping…")DS_Sleep(6000)Output("…Slept")Retu…
	SMTH.DS	2023-01-04 13:07:23.510	Function <Number> Main (<String@> _CLAs)Loop (3)<Number> age : Input("Age pls: ")Output("Your na…
✱	NULL	NULL	NULL

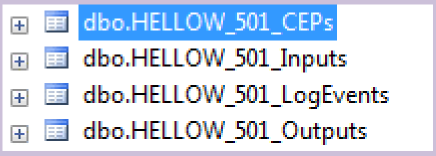


MNLT01\SQLEXPRE...ploadedPrograms
MNLT01\SQLEXPRE...xecutionSessions

	ESID	ProgramName	TimeStarted	TimeEnded	State	ExitReason
▶	DSI_GE_0EO	DSI_GETCURRE…	2023-01-20 08:…	2023-01-20 08:…	Finished	ExitedNormally …
	DSI_GE_IY1	DSI_GETCURRE…	2023-01-12 20:…	2023-01-12 20:…	Finished	ExitedNormally …
	FINITE_66G	FINITECEPCLIE…	2023-01-17 14:…	2023-01-17 14:…	Ready	ExitedNormally …
	HELLOW_501	HELLOWORLD.DS	2023-01-12 20:…	2023-01-12 20:…	Finished	ExitedNormally …
✱	NULL	NULL	NULL	NULL	NULL	NULL

- It is easy to perform a conditional selection on either the SQL Server, or the Web Server, via either TSQL or .NET respectively.

- One table is created for each type of Event, for each Execution-Session:



dbo.HELLOW_501_CEPs
dbo.HELLOW_501_Inputs
dbo.HELLOW_501_LogEvents
dbo.HELLOW_501_Outputs

This means that the data are neatly segmented into different tables, and can be easily selected individually or in groups.

I have also enforced a validity and consistency across the database, simply in my choice of DataTypes for the different fields…

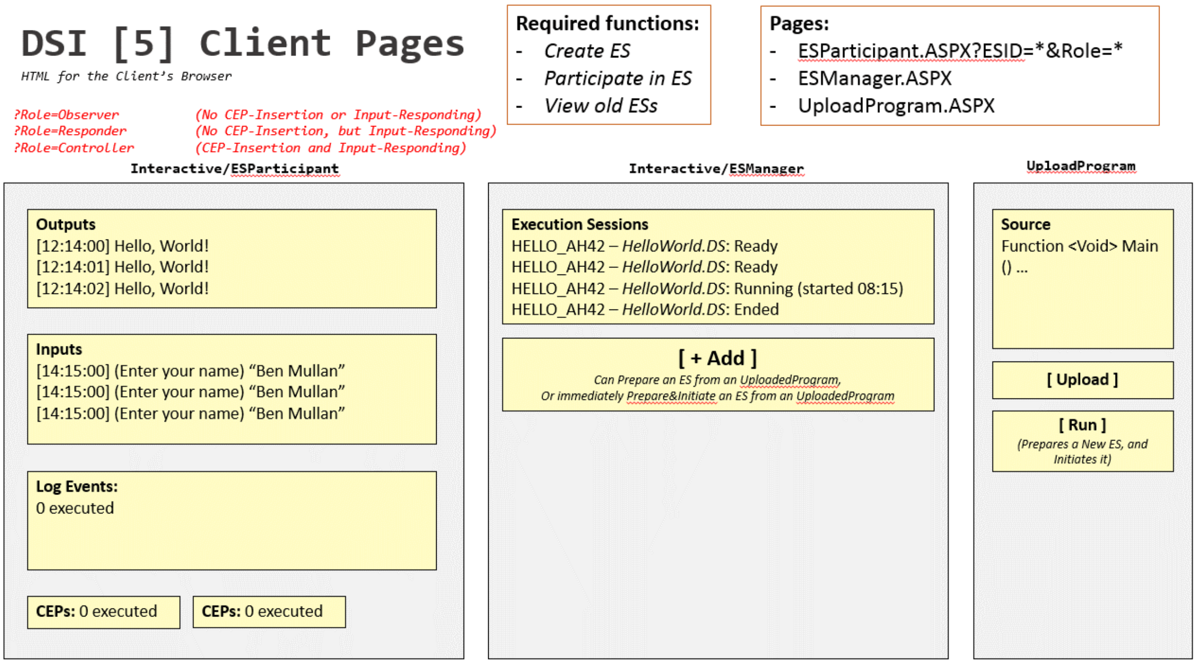
Validation: The following ensure that data of the correct variety is present in the SQL Tables…

- UploadedPrograms\ProgramName is `varchar(100)`
- ExecutionSessions\ESID is `varchar(100)`
- ExecutionSessions\State is `varchar(100)`
- ExecutionSessions\ProgramName is `varchar(100)`
- {ESID}_LogEvents\Severity is `varchar(100)`
- {ESID}_LogEvents\Category is `varchar(100)`
- The relevant fields in these tables are declared `NOT NULL`, to protect against NullReferenceExceptions.
- `VARCHAR(100)` means that **ONLY ASCII** characters are valid, and only 100 of them at that. Therefore, just in the choice of datatype, I have already protected the system against strange Unicode escape characters, or unbreakable spaces, or the backspace character-code etc.

Client-side Pages and Scripts

The client's web browser – of course – will never actually see any of the ~8,000 lines of script forming the API and Database. I will now write client-side pages (HTML and CSS) and scripts (JavaScript), which the client's browser will render.

I slightly re-thought how the input system ought to work on the client-side:



DSI [5] Client Pages

HTML for the Client's Browser

?Role=Observer (No CEP-Insertion or Input-Responding)
?Role=Responder (No CEP-Insertion, but Input-Responding)
?Role=Controller (CEP-Insertion and Input-Responding)

Required functions:
- *Create ES*
- *Participate in ES*
- *View old ESs*

Pages:
- ESParticipant.ASPX?ESID=*&Role=*
- ESManager.ASPX
- UploadProgram.ASPX

Interactive/ESParticipant

Outputs
[12:14:00] Hello, World!
[12:14:01] Hello, World!
[12:14:02] Hello, World!

Inputs
[14:15:00] (Enter your name) "Ben Mullan"
[14:15:00] (Enter your name) "Ben Mullan"
[14:15:00] (Enter your name) "Ben Mullan"

Log Events:
0 executed

CEPs: 0 executed

CEPs: 0 executed

Interactive/ESManager

Execution Sessions
HELLO_AH42 – *HelloWorld.DS*: Ready
HELLO_AH42 – *HelloWorld.DS*: Ready
HELLO_AH42 – *HelloWorld.DS*: Running (started 08:15)
HELLO_AH42 – *HelloWorld.DS*: Ended

[+ Add]
Can Prepare an ES from an UploadedProgram,
Or immediately Prepare&Initiate an ES from an UploadedProgram

UploadProgram

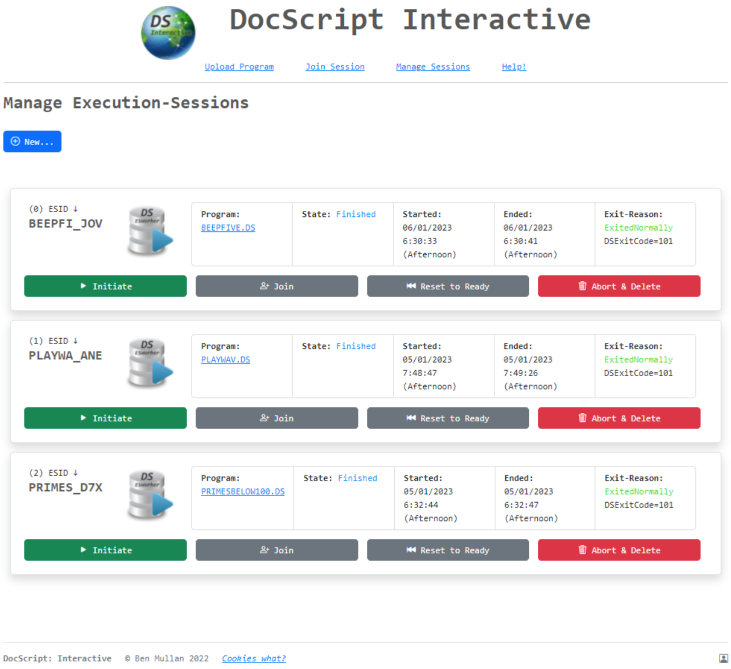
Source
Function <Void> Main () …

[Upload]

[Run]
(Prepares a New ES, and Initiates it)

I finalised my choice on frameworks used to style the webpages, and have decided to use **Bootstrap**, because is very widely-documented, and has good cross-browser compatibility.

This is what client-facing pages look like after writing the markup herefor…



DocScript Interactive

Upload Program Join Session Manage Sessions Help!

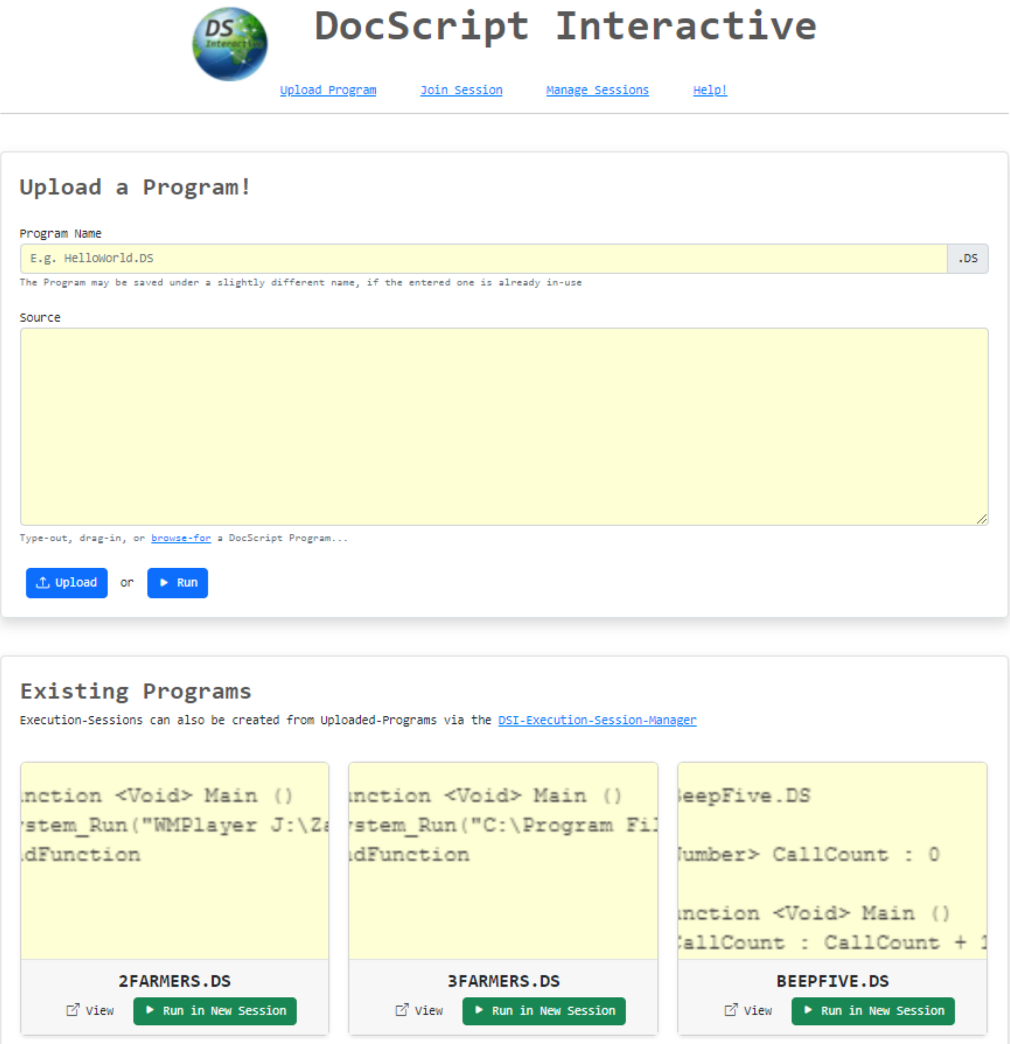
Manage Execution-Sessions

(0) ESID ↓
BEEPFI_JOV
Program: BEEPFIVE.DS
State: Finished
Started: 06/01/2023 6:30:33 (Afternoon)
Ended: 06/01/2023 6:30:41 (Afternoon)
Exit-Reason: ExitedNormally DSExitCode=101

(1) ESID ↓
PLAYWA_ANE
Program: PLAYWAV.DS
State: Finished
Started: 05/01/2023 7:48:47 (Afternoon)
Ended: 05/01/2023 7:49:26 (Afternoon)
Exit-Reason: ExitedNormally DSExitCode=101

(2) ESID ↓
PRIMES_D7X
Program: PRIMESBELOW100.DS
State: Finished
Started: 05/01/2023 6:32:44 (Afternoon)
Ended: 05/01/2023 6:32:47 (Afternoon)
Exit-Reason: ExitedNormally DSExitCode=101

DocScript: Interactive © Ben Mullan 2022 Cookies what?



DocScript Interactive

Upload a Program!

Program Name

E.g. HelloWorld.DS .DS

The Program may be saved under a slightly different name, if the entered one is already in-use

Source

Type-out, drag-in, or browse-for a DocScript Program...

Existing Programs

Execution-Sessions can also be created from Uploaded-Programs via the DSI-Execution-Session-Manager

```
nction <Void> Main ()
stem_Run("WMPlayer J:\Za
dFunction
```

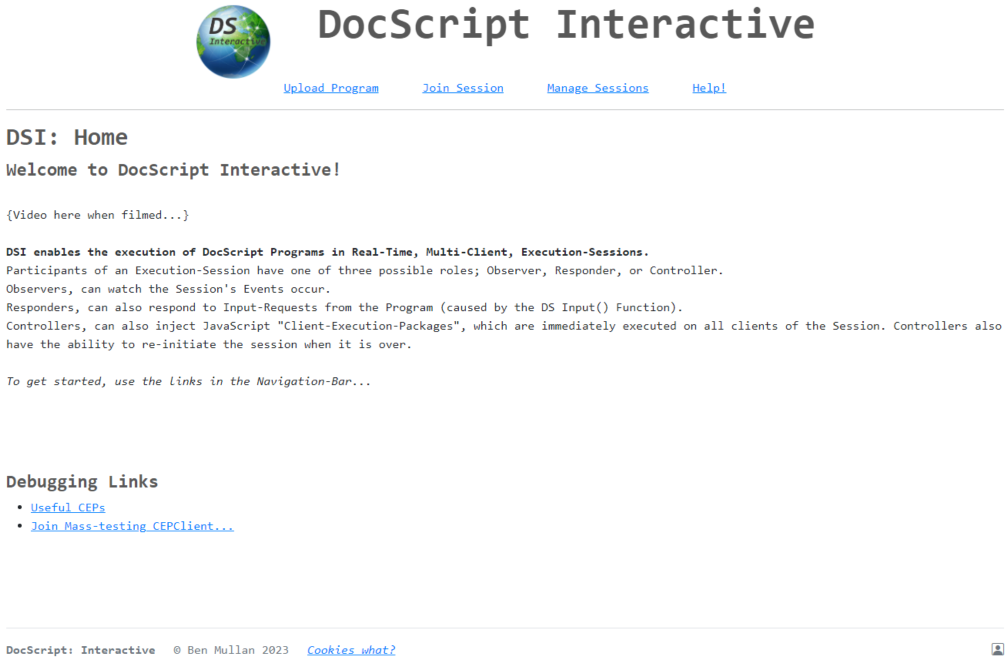
2FARMERS.DS

```
nction <Void> Main ()
stem_Run("C:\Program Fi
dFunction
```

3FARMERS.DS

```
eepFive.DS

umber> CallCount : 0

nction <Void> Main ()
allCount : CallCount + 1
```

BEEPFIVE.DS



DocScript Interactive

DSI: Home

Welcome to DocScript Interactive!

{Video here when filmed...}

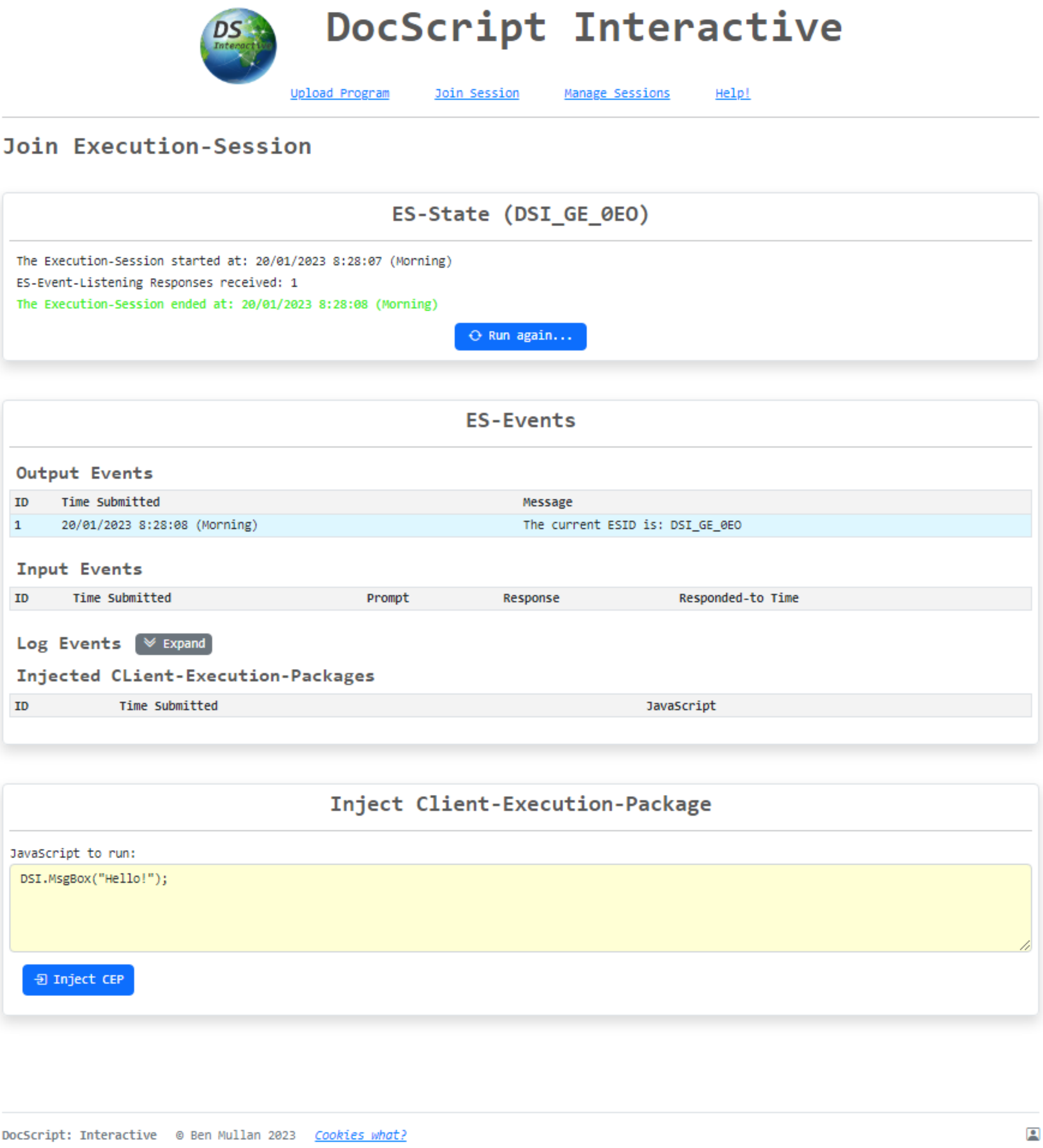
DSI enables the execution of DocScript Programs in Real-Time, Multi-Client, Execution-Sessions.
Participants of an Execution-Session have one of three possible roles; Observer, Responder, or Controller.
Observers, can watch the Session's Events occur.
Responders, can also respond to Input-Requests from the Program (caused by the DS Input() Function).
Controllers, can also inject JavaScript "Client-Execution-Packages", which are immediately executed on all clients of the Session. Controllers also have the ability to re-initiate the session when it is over.

To get started, use the links in the Navigation-Bar...

Debugging Links

- Useful CEPs
- Join Mass-testing CEPClient...

DocScript: Interactive © Ben Mullan 2023 *Cookies what?*



Join Execution-Session

ES-State (DSI_GE_0EO)

The Execution-Session started at: 20/01/2023 8:28:07 (Morning)

ES-Event-Listening Responses received: 1

The Execution-Session ended at: 20/01/2023 8:28:08 (Morning)

ES-Events

Output Events

ID	Time Submitted	Message
1	20/01/2023 8:28:08 (Morning)	The current ESID is: DSI_GE_0EO

Input Events

ID	Time Submitted	Prompt	Response	Responded-to Time

Log Events

Injected CLient-Execution-Packages

ID	Time Submitted	JavaScript

Inject Client-Execution-Package

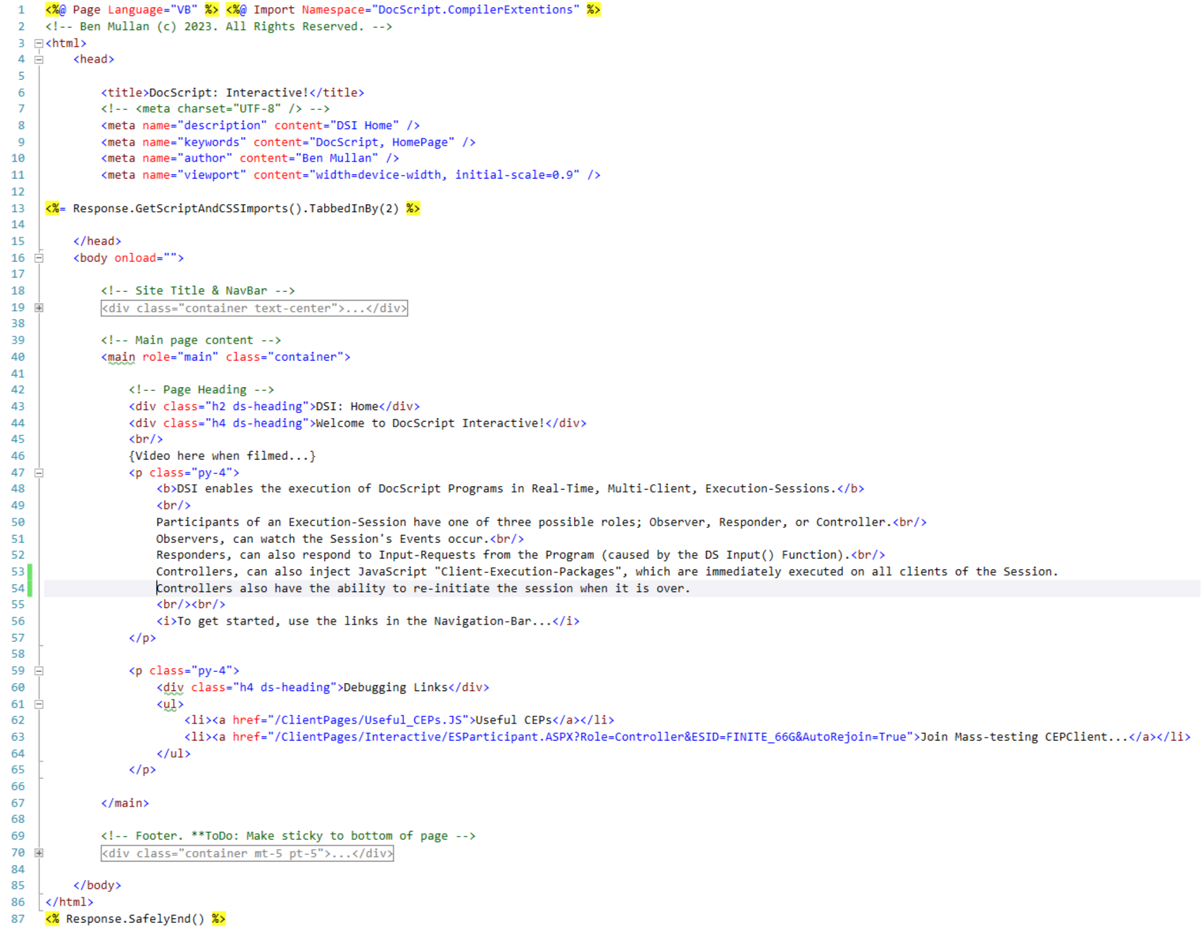
JavaScript to run:

```
DSI.MsgBox("Hello!");
```

DocScript: Interactive © Ben Mullan 2023

[Above] The DSInteractive ESParticipant Client page

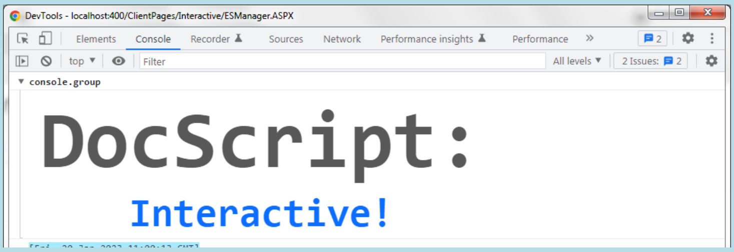
This is what the Code-Behind looks like for my HTML's general structure:



```
<%@ Page Language="VB" %> <%@ Import Namespace="DocScript.CompilerExtentions" %>
<!-- Ben Mullan (c) 2023. All Rights Reserved. -->
<html>
    <head>

        <title>DocScript: Interactive!</title>
        <!-- <meta charset="UTF-8" /> -->
        <meta name="description" content="DSI Home" />
        <meta name="keywords" content="DocScript, HomePage" />
        <meta name="author" content="Ben Mullan" />
        <meta name="viewport" content="width=device-width, initial-scale=0.9" />

<%= Response.GetScriptAndCSSImports().TabbedInBy(2) %>

    </head>
    <body onload="">

        <!-- Site Title & NavBar -->
        <div class="container text-center">...</div>

        <!-- Main page content -->
        <main role="main" class="container">

            <!-- Page Heading -->
            <div class="h2 ds-heading">DSI: Home</div>
            <div class="h4 ds-heading">Welcome to DocScript Interactive!</div>
            <br/>
            {Video here when filmed...}
            <p class="py-4">
                <b>DSI enables the execution of DocScript Programs in Real-Time, Multi-Client, Execution-Sessions.</b>
                <br/>
                Participants of an Execution-Session have one of three possible roles; Observer, Responder, or Controller.<br/>
                Observers, can watch the Session's Events occur.<br/>
                Responders, can also respond to Input-Requests from the Program (caused by the DS Input() Function).<br/>
                Controllers, can also inject JavaScript "Client-Execution-Packages", which are immediately executed on all clients of the Session.
                Controllers also have the ability to re-initiate the session when it is over.
                <br/><br/>
                <i>To get started, use the links in the Navigation-Bar...</i>
            </p>

            <p class="py-4">
                <div class="h4 ds-heading">Debugging Links</div>
                <ul>
                    <li><a href="/ClientPages/Useful_CEPs.JS">Useful CEPs</a></li>
                    <li><a href="/ClientPages/Interactive/ESParticipant.ASPX?Role=Controller&ESID=FINITE_66G&AutoRejoin=True">Join Mass-testing CEPClient...</a></li>
                </ul>
            </p>

        </main>

        <!-- Footer. **ToDo: Make sticky to bottom of page -->
        <div class="container mt-5 pt-5">...</div>

    </body>
</html>
<% Response.SafelyEnd() %>
```

<u>Prototype</u><u>: This point marks a milestone in the product, which is now capable of…</u>

- Displaying a user interface to the user, with the correct buttons and labels for each component.
- Running JavaScript initiated by components of the page via HTML `onclick=""` or `onload=""` attributes; this is the start-up Console-Banner for instance:



```
DocScript:
        Interactive!
```

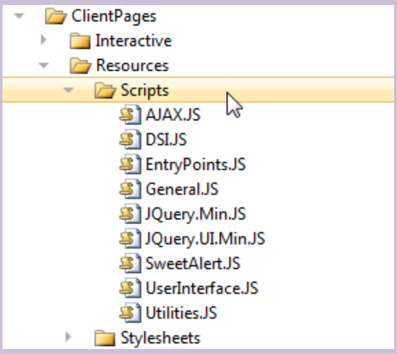
- Responsively scaling to resizing of the browser window, and displaying well and clearly on mobile devices whose displays are taller than they are wide.
- Performing server-side HTML-via-ASPX compilation for the pages which require QueryStrings. For example, ESParticipant.ASPX must be passed a `?ESID=*` QueryString.

Scripting: Linking the Back- to the Front-end

The JavaScript has the role of making requests to the API, from the client- to the server-side. These scripts are never actually seen by the user, but control the application web page, including programmatic user-interface functionality.

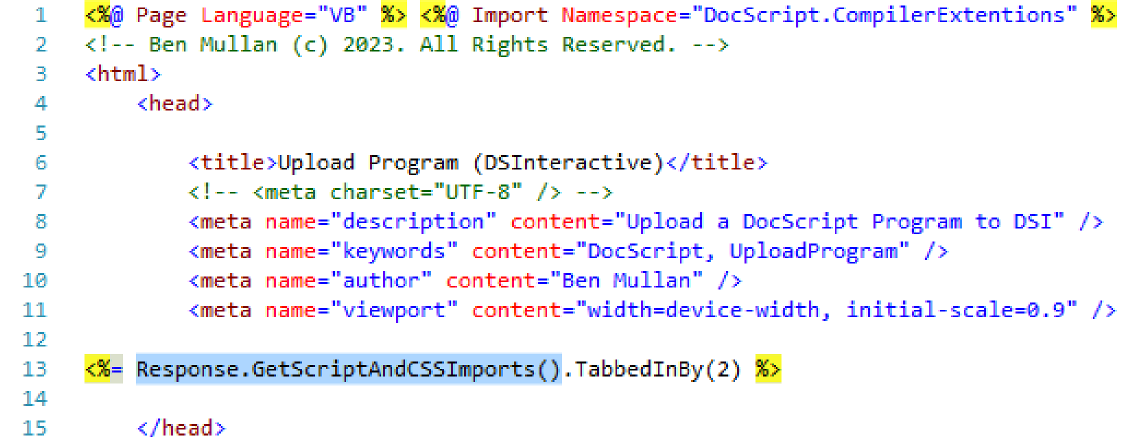
Structure and Modulatory: This Client-side JavaScript is well-structured, because…

- I have segmented the logic into a number of different *.JS Files. Some of these (including jQuery and SweetAlert, are not my source code, but rather, standard Webpage JavaScript Libraries).



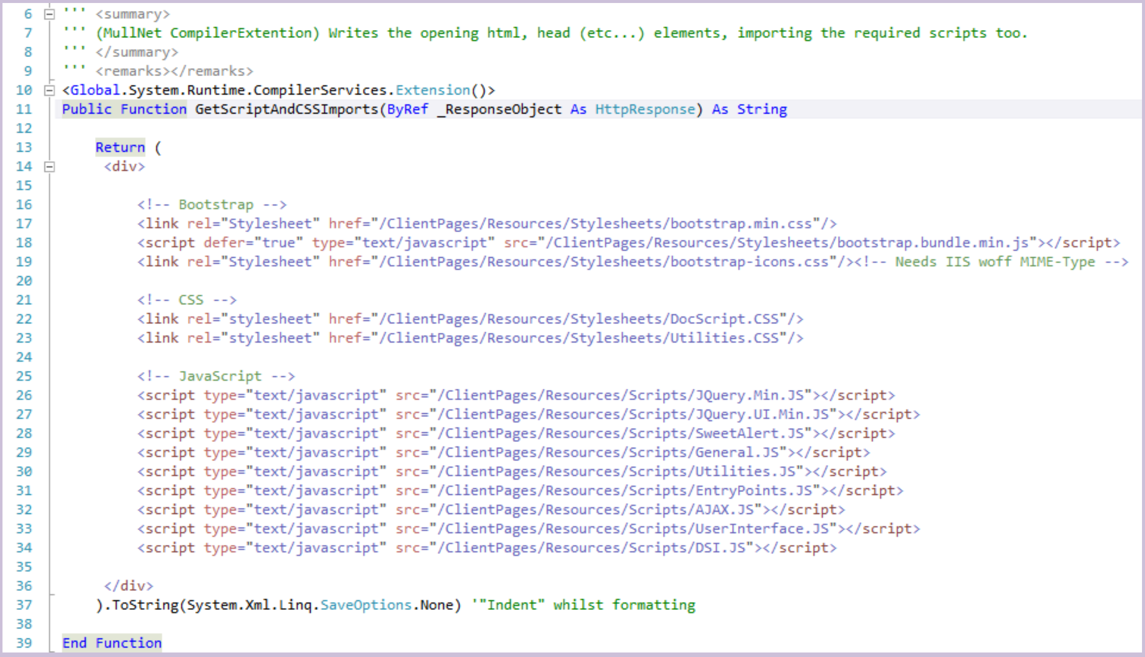
```
ClientPages
    Interactive
    Resources
        Scripts
            AJAX.JS
            DSI.JS
            EntryPoints.JS
            General.JS
            JQuery.Min.JS
            JQuery.UI.Min.JS
            SweetAlert.JS
            UserInterface.JS
            Utilities.JS
        Stylesheets
```

- I am injecting the script files into each HTML page, by a Server-Side Compiler-Extension call to `Response.GetScriptAndCSSImports()`:



```
<%@ Page Language="VB" %> <%@ Import Namespace="DocScript.CompilerExtentions" %>
<!-- Ben Mullan (c) 2023. All Rights Reserved. -->
<html>
    <head>

        <title>Upload Program (DSInteractive)</title>
        <!-- <meta charset="UTF-8" /> -->
        <meta name="description" content="Upload a DocScript Program to DSI" />
        <meta name="keywords" content="DocScript, UploadProgram" />
        <meta name="author" content="Ben Mullan" />
        <meta name="viewport" content="width=device-width, initial-scale=0.9" />

<%= Response.GetScriptAndCSSImports().TabbedInBy(2) %>

    </head>
```

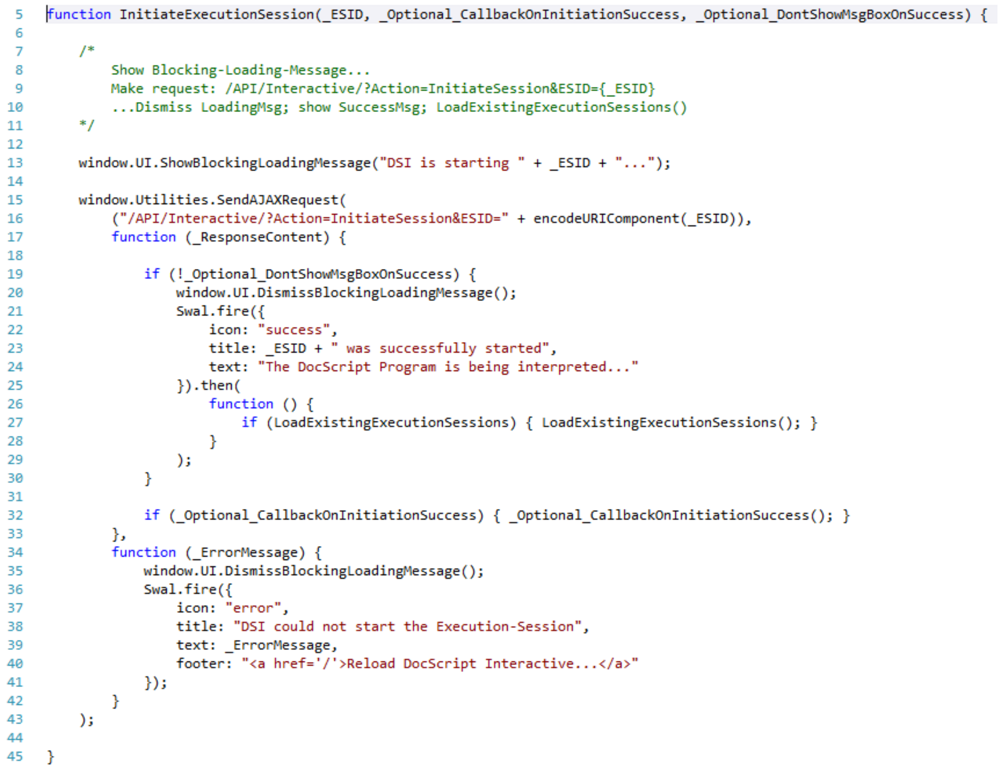
I implemented that Function thusly, using an XML Literal:



```
''' <summary>
''' (MullNet CompilerExtention) Writes the opening html, head (etc...) elements, importing the required scripts too.
''' </summary>
''' <remarks></remarks>
<Global.System.Runtime.CompilerServices.Extension()>
Public Function GetScriptAndCSSImports(ByRef _ResponseObject As HttpResponse) As String

    Return (
     <div>

         <!-- Bootstrap -->
         <link rel="Stylesheet" href="/ClientPages/Resources/Stylesheets/bootstrap.min.css"/>
         <script defer="true" type="text/javascript" src="/ClientPages/Resources/Stylesheets/bootstrap.bundle.min.js"></script>
         <link rel="Stylesheet" href="/ClientPages/Resources/Stylesheets/bootstrap-icons.css"/><!-- Needs IIS woff MIME-Type -->

         <!-- CSS -->
         <link rel="stylesheet" href="/ClientPages/Resources/Stylesheets/DocScript.CSS"/>
         <link rel="stylesheet" href="/ClientPages/Resources/Stylesheets/Utilities.CSS"/>

         <!-- JavaScript -->
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/JQuery.Min.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/JQuery.UI.Min.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/SweetAlert.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/General.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/Utilities.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/EntryPoints.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/AJAX.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/UserInterface.JS"></script>
         <script type="text/javascript" src="/ClientPages/Resources/Scripts/DSI.JS"></script>

     </div>
    ).ToString(System.Xml.Linq.SaveOptions.None) '"Indent" whilst formatting

End Function
```

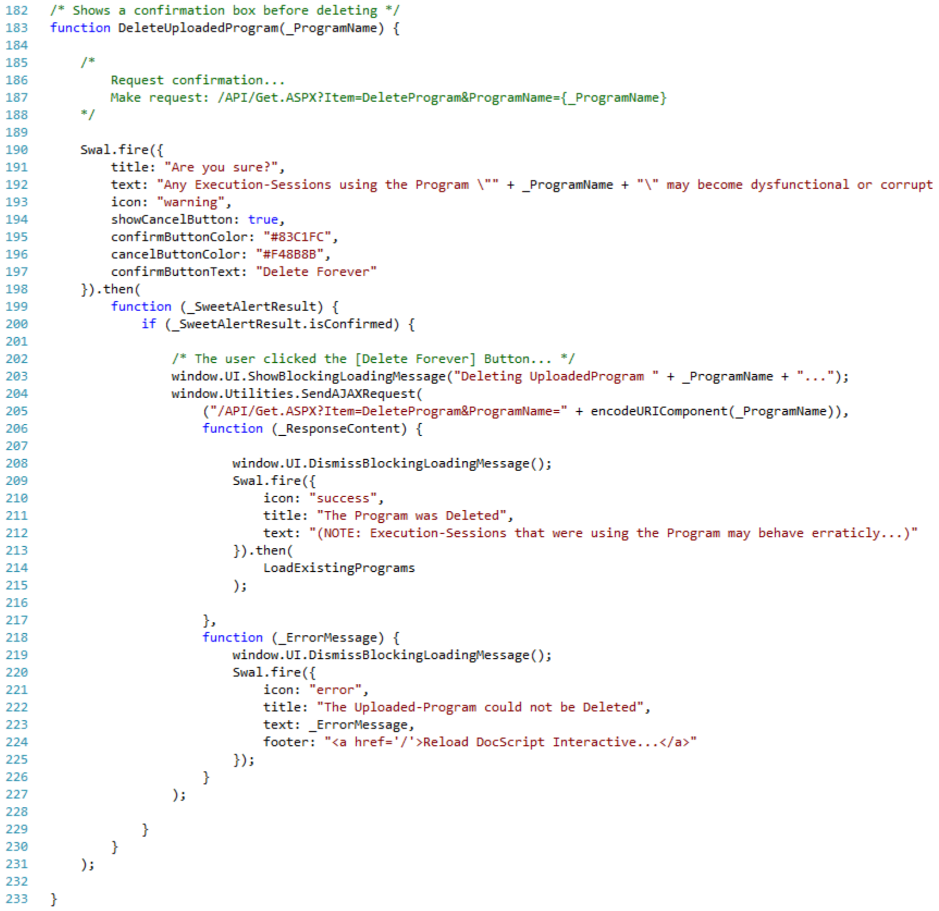
- I have written my own Utilities.JS JavaScript Library file, which is structured in a modular and reusable fashion; the file defined a JSON object `window.Utilities`, whose members are functions.

The bulk of the scripting can be found in the `AJAX.JS` File, which contains functions which perform requests for *Asynchronous-JavaScript-And-XML*…



```
function InitiateExecutionSession(_ESID, _Optional_CallbackOnInitiationSuccess, _Optional_DontShowMsgBoxOnSuccess) {

    /*
        Show Blocking-Loading-Message...
        Make request: /API/Interactive/?Action=InitiateSession&ESID={_ESID}
        ...Dismiss LoadingMsg; show SuccessMsg; LoadExistingExecutionSessions()
    */

    window.UI.ShowBlockingLoadingMessage("DSI is starting " + _ESID + "...");

    window.Utilities.SendAJAXRequest(
        ("/API/Interactive/?Action=InitiateSession&ESID=" + encodeURIComponent(_ESID)),
        function (_ResponseContent) {

            if (!_Optional_DontShowMsgBoxOnSuccess) {
                window.UI.DismissBlockingLoadingMessage();
                Swal.fire({
                    icon: "success",
                    title: _ESID + " was successfully started",
                    text: "The DocScript Program is being interpreted..."
                }).then(
                    function () {
                        if (LoadExistingExecutionSessions) { LoadExistingExecutionSessions(); }
                    }
                );
            }

            if (_Optional_CallbackOnInitiationSuccess) { _Optional_CallbackOnInitiationSuccess(); }
        },
        function (_ErrorMessage) {
            window.UI.DismissBlockingLoadingMessage();
            Swal.fire({
                icon: "error",
                title: "DSI could not start the Execution-Session",
                text: _ErrorMessage,
                footer: "<a href='/'>Reload DocScript Interactive...</a>"
            });
        }
    );

}
```

…And further down the file…



```
/* Shows a confirmation box before deleting */
function DeleteUploadedProgram(_ProgramName) {

    /*
        Request confirmation...
        Make request: /API/Get.ASPX?Item=DeleteProgram&ProgramName={_ProgramName}
    */

    Swal.fire({
        title: "Are you sure?",
        text: "Any Execution-Sessions using the Program \"" + _ProgramName + "\" may become dysfunctional or corrupt
        icon: "warning",
        showCancelButton: true,
        confirmButtonColor: "#83C1FC",
        cancelButtonColor: "#F48B8B",
        confirmButtonText: "Delete Forever"
    }).then(
        function (_SweetAlertResult) {
            if (_SweetAlertResult.isConfirmed) {

                /* The user clicked the [Delete Forever] Button... */
                window.UI.ShowBlockingLoadingMessage("Deleting UploadedProgram " + _ProgramName + "...");
                window.Utilities.SendAJAXRequest(
                    ("/API/Get.ASPX?Item=DeleteProgram&ProgramName=" + encodeURIComponent(_ProgramName)),
                    function (_ResponseContent) {

                        window.UI.DismissBlockingLoadingMessage();
                        Swal.fire({
                            icon: "success",
                            title: "The Program was Deleted",
                            text: "(NOTE: Execution-Sessions that were using the Program may behave erraticly...)"
                        }).then(
                            LoadExistingPrograms
                        );

                    },
                    function (_ErrorMessage) {
                        window.UI.DismissBlockingLoadingMessage();
                        Swal.fire({
                            icon: "error",
                            title: "The Uploaded-Program could not be Deleted",
                            text: _ErrorMessage,
                            footer: "<a href='/'>Reload DocScript Interactive...</a>"
                        });
                    }
                );

            }
        }
    );

}
```

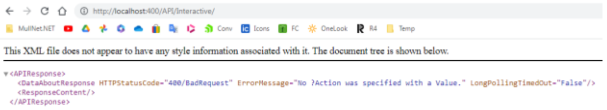
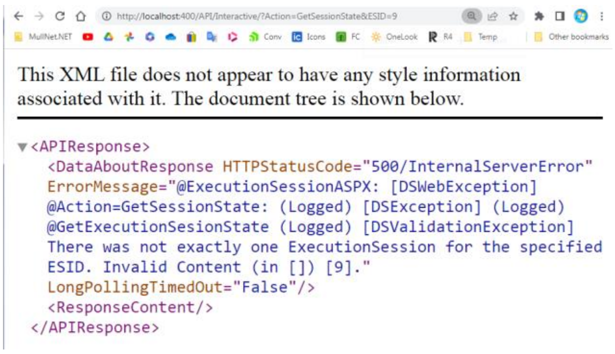
Progress Recap: Where am I in the development plan? *[Review]*

- **Done**: I have just written the Web-Based (*Interactive*) implementation of DocScript, which supports real-time, multi-client execution sessions.
  - **Next**: I will perform some more thorough testing of different DocScript programs. This is now possible, because I have implementations of the Interpreter which can be executed directly. (*I have the EXEs which invoke the logic within the DLL*)

Iterative Testing

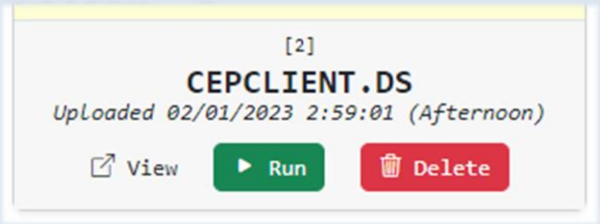
During §Analysis, I listed a number of types of test, along with example testing data, which I could use to test the product once implemented. I shall now make use of this testing plan…

Assert-style Test	☑ Passed?
Input and Output: Erroneous DocScript source can be taken in, but the interpreter determines that there is something wrong, instead of continuing and crashing.	Yes; see previous screenshots

Input and Output: API Calls can be accepted by the server, and they are validated syntactically and logically for consistency and sense. This includes the	Yes;
	
	http://localhost:400/API/Interactive/
	This XML file does not appear to have any style information associated with it. The document tree is shown below.
	`<APIResponse>`
	`<DataAboutResponse HTTPStatusCode="400/BadRequest" ErrorMessage="No ?Action was specified with a Value." LongPollingTimedOut="False"/>`
	`<ResponseContent/>`
	`</APIResponse>`
	
	http://localhost:400/API/Interactive/?Action=GetSessionState&ESID=9
	This XML file does not appear to have any style information associated with it. The document tree is shown below.
	`<APIResponse>`
	`<DataAboutResponse HTTPStatusCode="500/InternalServerError" ErrorMessage="@ExecutionSessionASPX: [DSWebException] @Action=GetSessionState: (Logged) [DSException] (Logged) @GetExecutionSesionState (Logged) [DSValidationException] There was not exactly one ExecutionSession for the specified ESID. Invalid Content (in []) [9]." LongPollingTimedOut="False"/>`
	`<ResponseContent/>`
	`</APIResponse>`

Usability: The database can be managed via the web application, instead of having to perform operations via `ssms` (SQL Server Management Studio). (i.e. there should be a button in the web interface for deleting an Execution-Session, instead of having to delete the records and tables from the database directly.)	No; There is not a comprehensive array of buttons available for every reasonably-expectable database interaction function yet. Because of this, I will now implement these extra buttons: • [Delete] for an UploadedProgram • [View] for an UploadedProgram • [Reset to Ready] for an ExecutionSession

<table>
<tr>
<td></td>
<td>



[2]

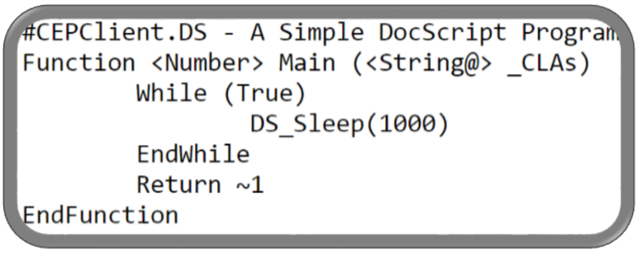
CEPCLIENT.DS

Uploaded 02/01/2023 2:59:01 (Afternoon)

</td>
</tr>
<tr>
<td>All of the "<b>DocScript Source</b>" from the <i>Testing</i> section of <i>§Design</i>, can be run successfully.</td>
<td>

Yes; see previous screenshots.

This included the "**CEPClient.DS**" Sample Program, which acts as a keep-alive, so that incoming packages of JavaScript can be executed, for as long as the session is needed.



```
#CEPClient.DS - A Simple DocScript Program
Function <Number> Main (<String@> _CLAs)
        While (True)
                DS_Sleep(1000)
        EndWhile
        Return ~1
EndFunction
```

This is incidentally how the Mass-Bluescreen demonstration from the DS3Min Video was orchestrated.

</td>
</tr>
</table>

Justifications for Actions Taken: By adding these additional Buttons to the Web Interface, the user can more quickly delete the ExecutionSessions and UploadedPrograms. Whilst using ssms directly does offer more *control*, it is not necessarily an approachable tool for newcomers, and teachers hosting DSI in their classrooms.

[Stage 5...]

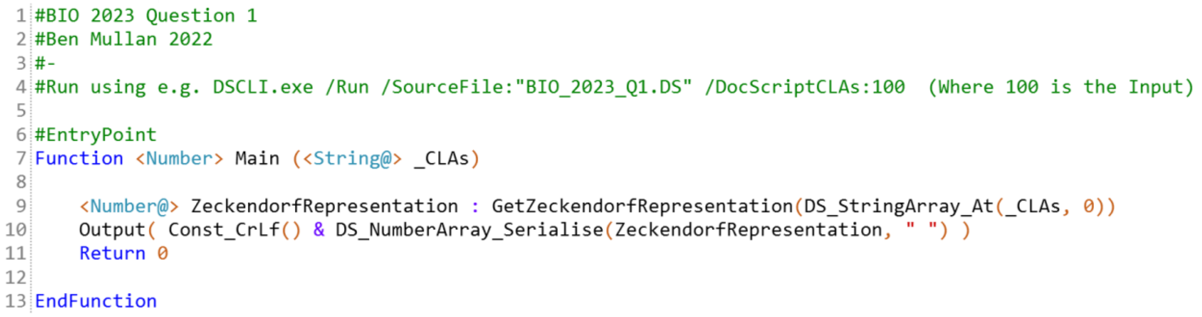
Whole-Program Testing

With all the **DocScript Implementations** themselves roughly at an RC0 stage (*Release Candidate Zero*), I shall now test the DocScript Interpretation Engine with some real-world programs!

As I make changes to the Core Interpreter DLL (e.g. because I discover a bug, or wish to add a feature), they will instantly be reflected in each of the Implementations too, because building any of the implementation projects will first compile the Library DLL.

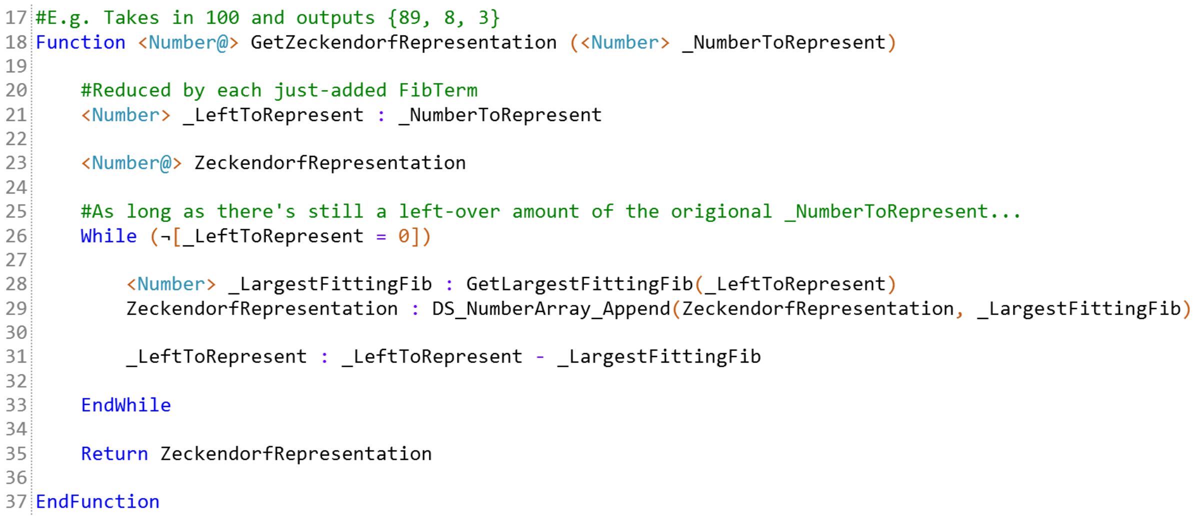
British Informatics Olympiad Question

Because this year's BIO is only a week away, I thought I'd set myself the challenge of using DocScript to compete in it. This would go some way to proving that the language isn't just a toy or gimmick, but can be used for important algorithmic problem-solving too! This was my entry:



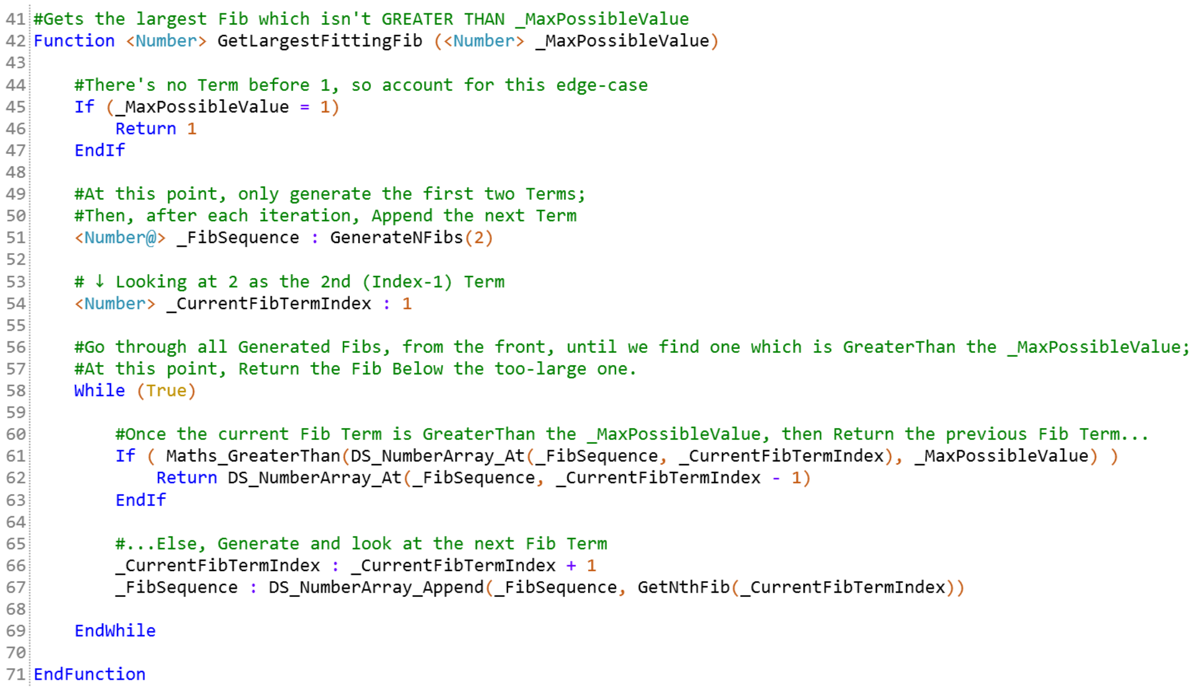
```
#BIO 2023 Question 1
#Ben Mullan 2022
#-
#Run using e.g. DSCLI.exe /Run /SourceFile:"BIO_2023_Q1.DS" /DocScriptCLAs:100  (Where 100 is the Input)

#EntryPoint
Function <Number> Main (<String@> _CLAs)

    <Number@> ZeckendorfRepresentation : GetZeckendorfRepresentation(DS_StringArray_At(_CLAs, 0))
    Output( Const_CrLf() & DS_NumberArray_Serialise(ZeckendorfRepresentation, " ") )
    Return 0

EndFunction
```

[Above] DocScript Function Main()



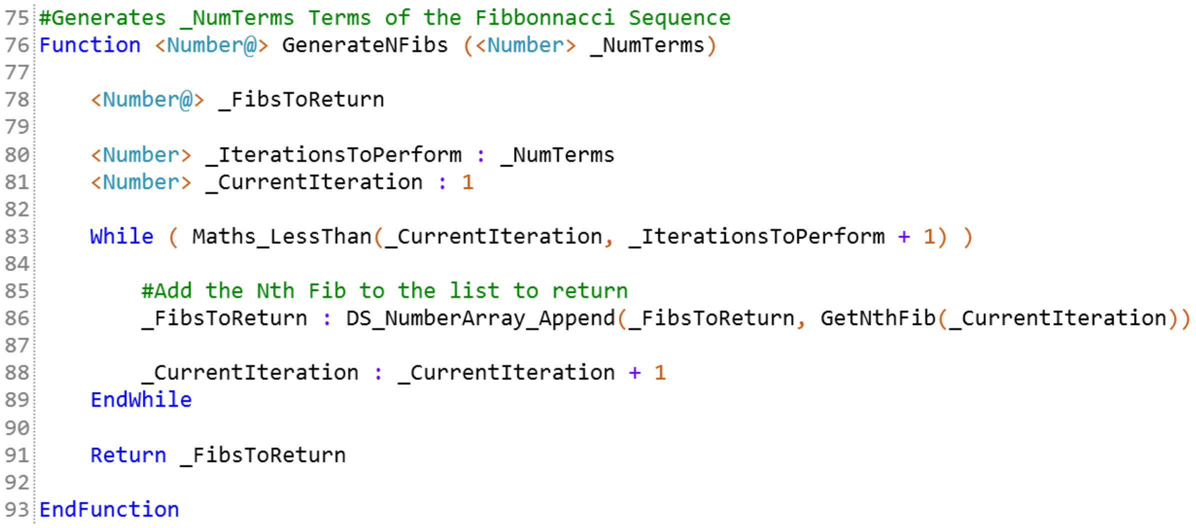
```
#E.g. Takes in 100 and outputs {89, 8, 3}
Function <Number@> GetZeckendorfRepresentation (<Number> _NumberToRepresent)

    #Reduced by each just-added FibTerm
    <Number> _LeftToRepresent : _NumberToRepresent

    <Number@> ZeckendorfRepresentation

    #As long as there's still a left-over amount of the origional _NumberToRepresent...
    While (¬[_LeftToRepresent = 0])

        <Number> _LargestFittingFib : GetLargestFittingFib(_LeftToRepresent)
        ZeckendorfRepresentation : DS_NumberArray_Append(ZeckendorfRepresentation, _LargestFittingFib)

        _LeftToRepresent : _LeftToRepresent - _LargestFittingFib

    EndWhile

    Return ZeckendorfRepresentation

EndFunction
```

[Above] DocScript Function GetZeckendorfRepresentation()



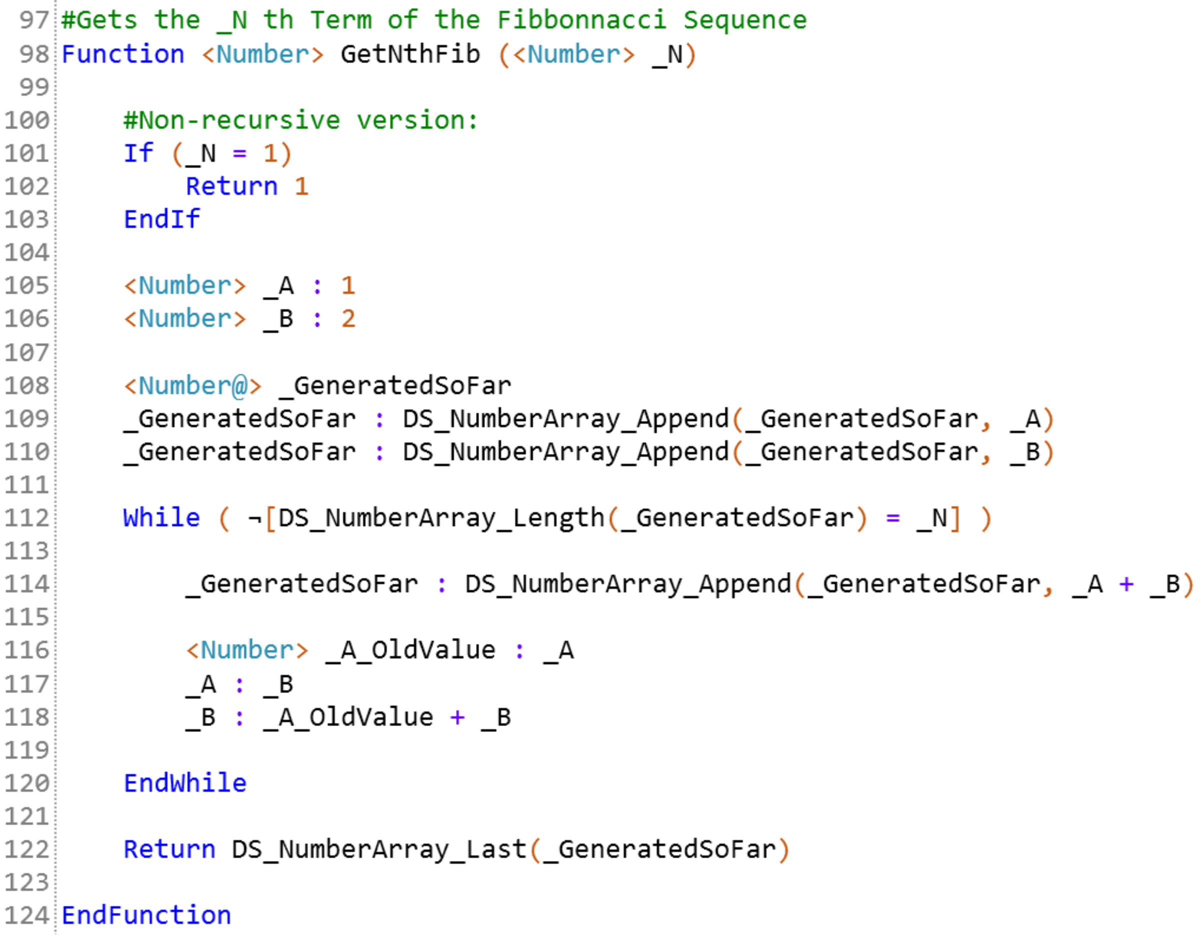
```
#Gets the largest Fib which isn't GREATER THAN _MaxPossibleValue
Function <Number> GetLargestFittingFib (<Number> _MaxPossibleValue)

    #There's no Term before 1, so account for this edge-case
    If (_MaxPossibleValue = 1)
        Return 1
    EndIf

    #At this point, only generate the first two Terms;
    #Then, after each iteration, Append the next Term
    <Number@> _FibSequence : GenerateNFibs(2)

    # ↓ Looking at 2 as the 2nd (Index-1) Term
    <Number> _CurrentFibTermIndex : 1

    #Go through all Generated Fibs, from the front, until we find one which is GreaterThan the _MaxPossibleValue;
    #At this point, Return the Fib Below the too-large one.
    While (True)

        #Once the current Fib Term is GreaterThan the _MaxPossibleValue, then Return the previous Fib Term...
        If ( Maths_GreaterThan(DS_NumberArray_At(_FibSequence, _CurrentFibTermIndex), _MaxPossibleValue) )
            Return DS_NumberArray_At(_FibSequence, _CurrentFibTermIndex - 1)
        EndIf

        #...Else, Generate and look at the next Fib Term
        _CurrentFibTermIndex : _CurrentFibTermIndex + 1
        _FibSequence : DS_NumberArray_Append(_FibSequence, GetNthFib(_CurrentFibTermIndex))

    EndWhile

EndFunction
```

[Above] DocScript Function GetLargestFittingFib()



```
#Generates _NumTerms Terms of the Fibbonnacci Sequence
Function <Number@> GenerateNFibs (<Number> _NumTerms)

    <Number@> _FibsToReturn

    <Number> _IterationsToPerform : _NumTerms
    <Number> _CurrentIteration : 1

    While ( Maths_LessThan(_CurrentIteration, _IterationsToPerform + 1) )

        #Add the Nth Fib to the list to return
        _FibsToReturn : DS_NumberArray_Append(_FibsToReturn, GetNthFib(_CurrentIteration))

        _CurrentIteration : _CurrentIteration + 1
    EndWhile

    Return _FibsToReturn

EndFunction
```

[Above] DocScript Function GenerateNFibs()



```
#Gets the _N th Term of the Fibbonnacci Sequence
Function <Number> GetNthFib (<Number> _N)

    #Non-recursive version:
    If (_N = 1)
        Return 1
    EndIf

    <Number> _A : 1
    <Number> _B : 2

    <Number@> _GeneratedSoFar
    _GeneratedSoFar : DS_NumberArray_Append(_GeneratedSoFar, _A)
    _GeneratedSoFar : DS_NumberArray_Append(_GeneratedSoFar, _B)

    While ( ¬[DS_NumberArray_Length(_GeneratedSoFar) = _N] )

        _GeneratedSoFar : DS_NumberArray_Append(_GeneratedSoFar, _A + _B)

        <Number> _A_OldValue : _A
        _A : _B
        _B : _A_OldValue + _B

    EndWhile

    Return DS_NumberArray_Last(_GeneratedSoFar)

EndFunction
```

[Above] DocScript Function GetNthFib()

(These screenshots are all from the DocScript Windows IDE, so also serve to show that the syntax-highlighting thereof is effective…)

Review

For this question in the BIO, I miraculously managed to score full-marks with this DocScript Program! What's more:

- All the test-cases were executed in **under a second** (∴ the language is fast)
- The interpreter **never crashed** (∴ the language is stable)
- Tests could be run **with ease**, and in quick succession, owing to the inputs being provided via command-line arguments; on the press of the up-arrow in the console window, I could instantly run exactly the same script, just with different CLAs (∴ the language is easy-to-use)

Primes-Below-100-Base-2-To-32 Example

At the very start of §Analysis, I proposed that the programming language being developed herein ought to be able to tackle the following problem:

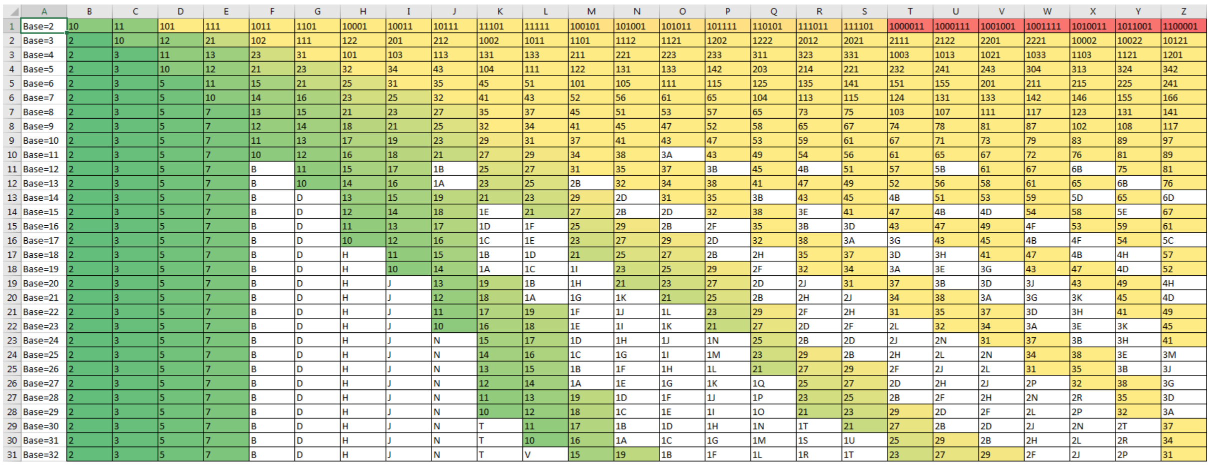
> *" A user wishes list the prime numbers below 100, in each base from 2 (Binary) to 32 (Duotrigesimal)"*

To resolve this project in a gratifying, cyclical manner, I will therefore use DocScript to actually write this program. Here it is:

```
#PrimesBelow100_Base2To32.DS
#Brief: "A user wishes list the prime numbers below 100, in each base from 2 to 32"

Function <Number> Main (<String@> _CLAs)

    #To be saved to an Output File:
    <String> _AllPrimes_InAllBases

    #Up to 97:
    <Number@> _Primes : GetPrimesBelow(100)

    #For each Base {2...32}
    <Number> _HighestBase : 32_10
    <Number> _CurrentBase : 2_10
    While ( Maths_LessThan(_CurrentBase, _HighestBase + 1) )

        _AllPrimes_InAllBases : _AllPrimes_InAllBases & "Base=" & _CurrentBase & ","

        #For each PrimeTerm {0...100}
        <Number> _HighestPrimeTerm : DS_NumberArray_Length(_Primes)
        <Number> _CurrentPrimeTerm : 0
        While ( Maths_LessThan(_CurrentPrimeTerm, _HighestPrimeTerm) )
            #Output e.g.    "2,"
            _AllPrimes_InAllBases : _AllPrimes_InAllBases & [DS_Number_ToBase(DS_NumberArra
y_At(_Primes, _CurrentPrimeTerm), _CurrentBase) & ","]
            _CurrentPrimeTerm : _CurrentPrimeTerm + 1
        EndWhile

        _AllPrimes_InAllBases : _AllPrimes_InAllBases & Const_CrLf()
        _CurrentBase : _CurrentBase + 1
    EndWhile

    File_WriteText("PrimesBelow100_Base2To32.CSV", _AllPrimes_InAllBases, False)
    Return 0
```

```
EndFunction

Function <Number@> GetPrimesBelow (<Number> _NumberOfPrimes)

    #A prime number is: a whole number [greater than 1] whose only factors are [1 and
itself]
    <Number@> _CollectedPrimes

    <Number> _IterationsToPerform : _NumberOfPrimes
    <Number> _CurrentIteration : 2
    While ( Maths_LessThan(_CurrentIteration, _IterationsToPerform + 1) )
        If ( IsAPrime(_CurrentIteration) )
            _CollectedPrimes : DS_NumberArray_Append(_CollectedPrimes, _CurrentIteration)
        EndIf
        _CurrentIteration : _CurrentIteration + 1
    EndWhile

    Return _CollectedPrimes

EndFunction

Function <Boolean> IsAPrime (<Number> _Test)

    <Number> _I : 2
    While ( Maths_LessThan(_I * _I, _Test) | [[_I * _I] = _Test] )
        If ([_Test % _I] = 0)
            Return False
        EndIf
        _I : _I + 1
    EndWhile

    Return True

EndFunction
```

That program generates a simple .csv file, which, when **viewed in *Excel* with some *conditional* *formatting***, produced this rather fascinating result:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	Base=2	10	11	101	111	1011	1101	10001	10011	10111	11101	11111	100101	101001	101011	101111	110101	111011	111101	1000011	1000111	1001001	1001111	1010011	1011001	1100001
2	Base=3	2	10	12	21	102	111	122	201	212	1002	1011	1101	1112	1121	1202	1222	2012	2021	2111	2122	2201	2221	10002	10022	10121
3	Base=4	2	3	11	13	23	31	101	103	113	131	133	211	221	223	233	311	323	331	1003	1013	1021	1033	1103	1121	1201
4	Base=5	2	3	10	12	21	23	32	34	43	104	111	122	131	133	142	203	214	221	232	241	243	304	313	324	342
5	Base=6	2	3	5	11	15	21	25	31	35	45	51	101	105	111	115	125	135	141	151	155	201	211	215	225	241
6	Base=7	2	3	5	10	14	16	23	25	32	41	43	52	56	61	65	104	113	115	124	131	133	142	146	155	166
7	Base=8	2	3	5	7	13	15	21	23	27	35	37	45	51	53	57	65	73	75	103	107	111	117	123	131	141
8	Base=9	2	3	5	7	12	14	18	21	25	32	34	41	45	47	52	58	65	67	74	78	81	87	102	108	117
9	Base=10	2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89	97
10	Base=11	2	3	5	7	10	12	16	18	21	27	29	34	38	3A	43	49	54	56	61	65	67	72	76	81	89
11	Base=12	2	3	5	7	B	11	15	17	1B	25	27	31	35	37	3B	45	4B	51	57	5B	61	67	6B	75	81
12	Base=13	2	3	5	7	B	10	14	16	1A	23	25	2B	32	34	38	41	47	49	52	56	58	61	65	6B	76
13	Base=14	2	3	5	7	B	D	13	15	19	21	23	29	2D	31	35	3B	43	45	4B	51	53	59	5D	65	6D
14	Base=15	2	3	5	7	B	D	12	14	18	1E	21	27	2B	2D	32	38	3E	41	47	4B	4D	54	58	5E	67
15	Base=16	2	3	5	7	B	D	11	13	17	1D	1F	25	29	2B	2F	35	3B	3D	43	47	49	4F	53	59	61
16	Base=17	2	3	5	7	B	D	10	12	16	1C	1E	23	27	29	2D	32	38	3A	3G	43	45	4B	4F	54	5C
17	Base=18	2	3	5	7	B	D	H	11	15	1B	1D	21	25	27	2B	2H	35	37	3D	3H	41	47	4B	4H	57
18	Base=19	2	3	5	7	B	D	H	10	14	1A	1C	1I	23	25	29	2F	32	34	3A	3E	3G	43	47	4D	52
19	Base=20	2	3	5	7	B	D	H	J	13	19	1B	1H	21	23	27	2D	2J	31	37	3B	3D	3J	43	49	4H
20	Base=21	2	3	5	7	B	D	H	J	12	18	1A	1G	1K	21	25	2B	2H	2J	34	38	3A	3G	3K	45	4D
21	Base=22	2	3	5	7	B	D	H	J	11	17	19	1F	1J	1L	23	29	2F	2H	31	35	37	3D	3H	41	49
22	Base=23	2	3	5	7	B	D	H	J	10	16	18	1E	1I	1K	21	27	2D	2F	2L	32	34	3A	3E	3K	45
23	Base=24	2	3	5	7	B	D	H	J	N	15	17	1D	1H	1J	1N	25	2B	2D	2J	2N	31	37	3B	3H	41
24	Base=25	2	3	5	7	B	D	H	J	N	14	16	1C	1G	1I	1M	23	29	2B	2H	2L	2N	34	38	3E	3M
25	Base=26	2	3	5	7	B	D	H	J	N	13	15	1B	1F	1H	1L	21	27	29	2F	2J	2L	31	35	3B	3J
26	Base=27	2	3	5	7	B	D	H	J	N	12	14	1A	1E	1G	1K	1Q	25	27	2D	2H	2J	2P	32	38	3G
27	Base=28	2	3	5	7	B	D	H	J	N	11	13	19	1D	1F	1J	1P	23	25	2B	2F	2H	2N	2R	35	3D
28	Base=29	2	3	5	7	B	D	H	J	N	10	12	18	1C	1E	1I	1O	21	23	29	2D	2F	2L	2P	32	3A
29	Base=30	2	3	5	7	B	D	H	J	N	T	11	17	1B	1D	1H	1N	1T	21	27	2B	2D	2J	2N	2T	37
30	Base=31	2	3	5	7	B	D	H	J	N	T	10	16	1A	1C	1G	1M	1S	1U	25	29	2B	2H	2L	2R	34
31	Base=32	2	3	5	7	B	D	H	J	N	T	V	15	19	1B	1F	1L	1R	1T	23	27	29	2F	2J	2P	31

[↑ DocScript produced these data ↑]

Review

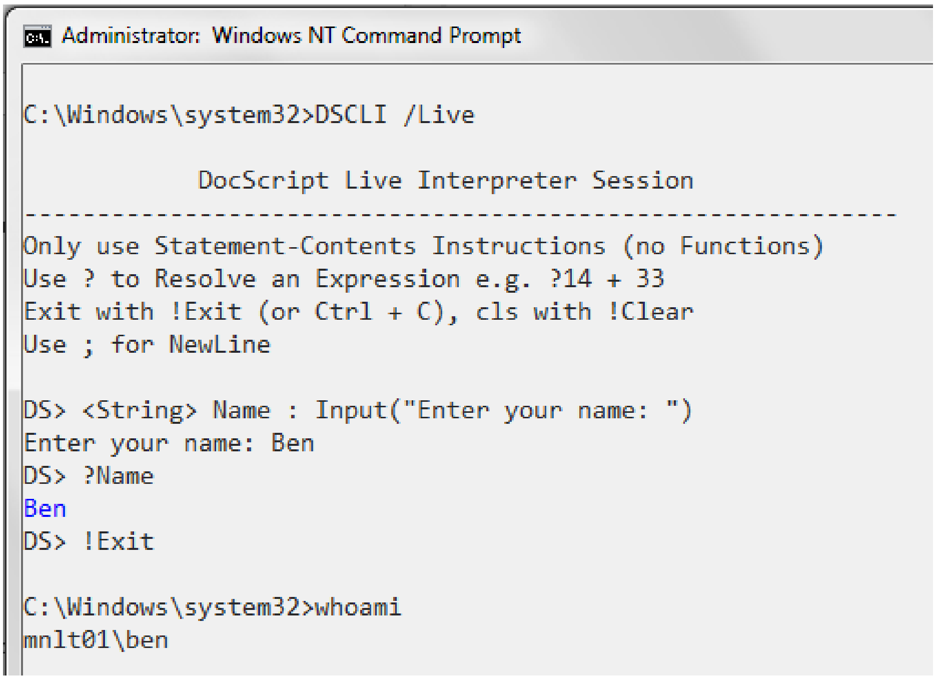
What that graphic essentially shows, is that the value compositions in essence cascade down and along the prime numbers. More importantly though: **what this *test* and example program have shown**, is that DocScript can reliably, consistently, straightforwardly, and (*perhaps surprisingly*) rapidly interpret real programs to produce real, palpable results. The test has therefore been a success, and it is almost something of a shame that there wasn't another calamitous – albeit entertaining – failure, for me to write about herein!

Improvements to make, revealed by the Tests

Although the programs ran faultlessly, I did notice that it would be worth making the following improvements and additions:

- A greater number of `BuiltInFunction`s would have been useful in some instances. For example, instead of having to write `Maths_GreaterThan(4, 5) | [4 = 5]`, it would admittedly have been easier if there were a singular BuiltInFunction available to act as a `>=` operator; `Maths_GreaterThanOrEqualTo(4, 5)`. Therefore, I am adding a new series of BIFs, to make common tasks even more easy. These include: `DS_*Array_Last()`, `DS_*Array_First()`, `File_Create()`, `File_Delete()`, `Maths_GreaterThanOrEqualTo()` and `Maths_LessThanOrEqualTo()`.

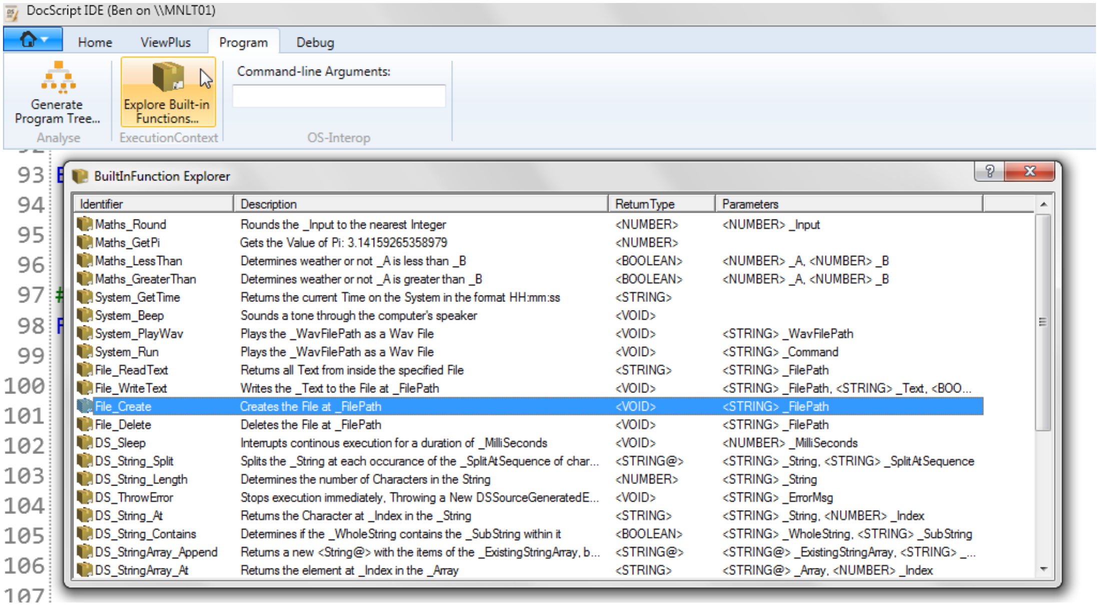
- A "*Live*" mode for the command-line interpreter (DSCLI.EXE) would also have been useful, to enable me to test single lines of DocScript, during the development of a larger Program. Interpreted languages such as Python (which – it behoves me to say – I *abhor*) have such a "*Live*" feature, in the form of "interactive mode" (not to be confused with DocScript Interactive, the web-based system, which is something quite different).
  Because I have written DocScript to be extensible, adding this ***Live*** feature to DSCLI is not too difficult at all; the executable shall simply take in an additional optional command-line argument `/Live`, which calls `EnterDSLiveSession()`, which infinitely loops, asking for a line of DocScript, executing it, and re-applying the resultant symbol-tables-state to a local variable herefor. I managed to implement the ***Live*** mode in *under 100 lines*. Here it is in-action:



```
C:\Windows\system32>DSCLI /Live

           DocScript Live Interpreter Session
------------------------------------------------------------
Only use Statement-Contents Instructions (no Functions)
Use ? to Resolve an Expression e.g. ?14 + 33
Exit with !Exit (or Ctrl + C), cls with !Clear
Use ; for NewLine

DS> <String> Name : Input("Enter your name: ")
Enter your name: Ben
DS> ?Name
Ben
DS> !Exit

C:\Windows\system32>whoami
mnlt01\ben
```

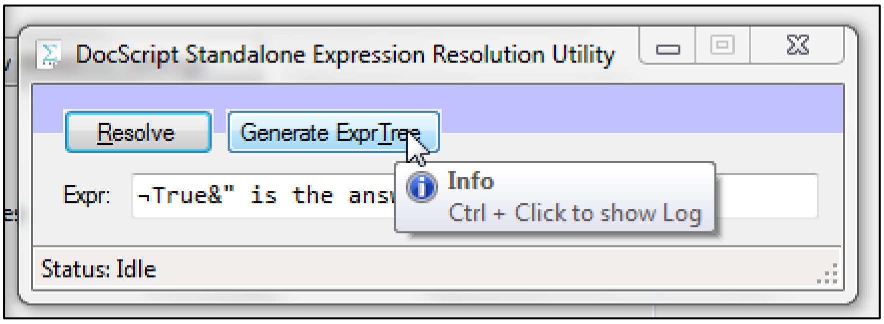
(As can be seen by the introduction text for DSLive, I also added `?{Expr}` expression-resolution, and `!{Meta}` meta-commends, to make the ***Live*** feature more user-friendly and direct.)

- I could have done with a means of viewing which **Built-In-Functions were available** in the current `ExecutionContext`. I am therefore adding a "Explore Built-in Functions" button to DSIDE. It brings up the following window:



Identifier	Description	ReturnType	Parameters
Maths_Round	Rounds the _Input to the nearest Integer	<NUMBER>	<NUMBER> _Input
Maths_GetPi	Gets the Value of Pi: 3.14159265358979	<NUMBER>	
Maths_LessThan	Determines weather or not _A is less than _B	<BOOLEAN>	<NUMBER> _A, <NUMBER> _B
Maths_GreaterThan	Determines weather or not _A is greater than _B	<BOOLEAN>	<NUMBER> _A, <NUMBER> _B
System_GetTime	Returns the current Time on the System in the format HH:mm:ss	<STRING>	
System_Beep	Sounds a tone through the computer's speaker	<VOID>	
System_PlayWav	Plays the _WavFilePath as a Wav File	<VOID>	<STRING> _WavFilePath
System_Run	Plays the _WavFilePath as a Wav File	<VOID>	<STRING> _Command
File_ReadText	Returns all Text from inside the specified File	<STRING>	<STRING> _FilePath
File_WriteText	Writes the _Text to the File at _FilePath	<VOID>	<STRING> _FilePath, <STRING> _Text, <BOO...
File_Create	Creates the File at _FilePath	<VOID>	<STRING> _FilePath
File_Delete	Deletes the File at _FilePath	<VOID>	<STRING> _FilePath
DS_Sleep	Interrupts continous execution for a duration of _MilliSeconds	<VOID>	<NUMBER> _MilliSeconds
DS_String_Split	Splits the _String at each occurance of the _SplitAtSequence of char...	<STRING@>	<STRING> _String, <STRING> _SplitAtSequence
DS_String_Length	Determines the number of Characters in the String	<NUMBER>	<STRING> _String
DS_ThrowError	Stops execution immediately, Throwing a New DSSourceGeneratedE...	<VOID>	<STRING> _ErrorMsg
DS_String_At	Returns the Character at _Index in the _String	<STRING>	<STRING> _String, <NUMBER> _Index
DS_String_Contains	Determines if the _WholeString contains the _SubString within it	<BOOLEAN>	<STRING> _WholeString, <STRING> _SubString
DS_StringArray_Append	Returns a new <String@> with the items of the _ExistingStringArray, b...	<STRING@>	<STRING@> _ExistingStringArray, <STRING> _...
DS_StringArray_At	Returns the element at _Index in the _Array	<STRING>	<STRING@> _Array, <NUMBER> _Index

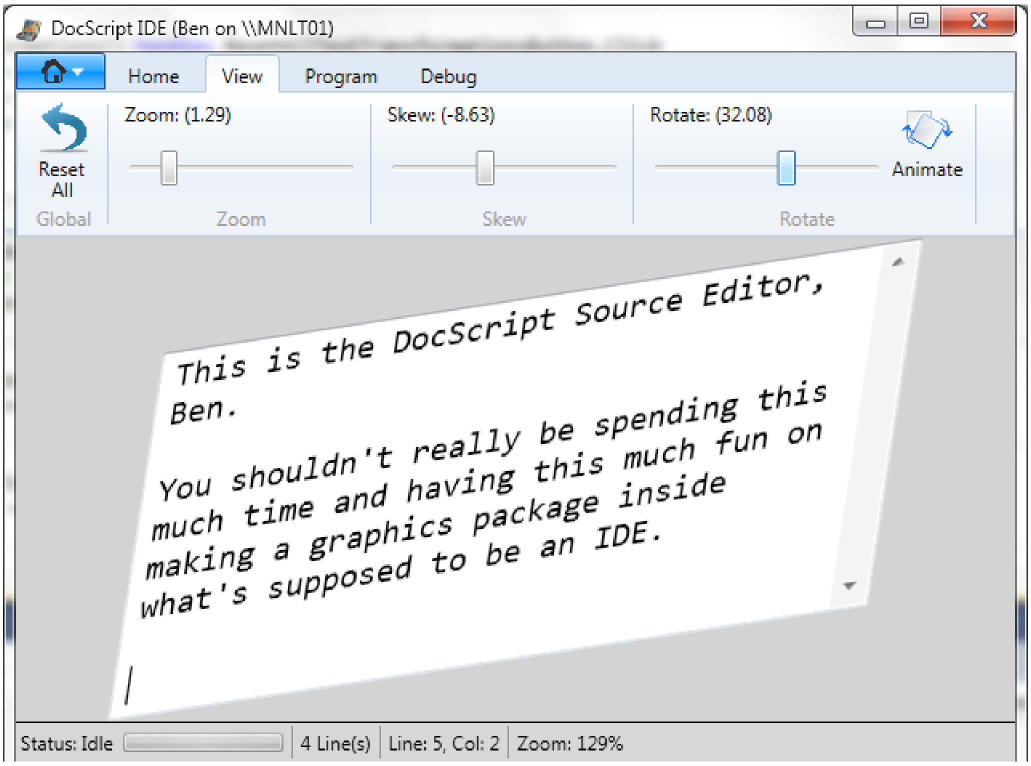
- There are also a few user-interface components to touch up; I shall add some ToolTips in the DSExpr (DocScript Standalone Expression Resolution Utility) program…



Expr: ¬True&" is the ansı

Status: Idle

…as well as some zoom and other graphics controls for the Windows IDE, using Matrix Multiplications:



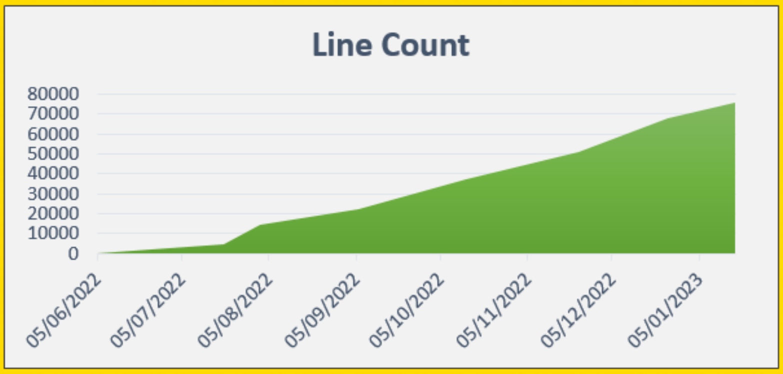
```
This is the DocScript Source Editor,
Ben.

You shouldn't really be spending this
much time and having this much fun on
making a graphics package inside
what's supposed to be an IDE.
```

Status: Idle | 4 Line(s) | Line: 5, Col: 2 | Zoom: 129%

These features are intended to improve usability.

Progress Check

At this point in the development, I have written 75726 Lines of code…



*(…Obviously not all of that code is here in this document; that would take **over 2 reams** of paper…)*

[...Stage 5...]

Scenario-based Testing

Here, I enact some of the stakeholders' scenarios discussed in **§Analysis**, and evaluate whether or not the software I have built fares well when used in these situations.

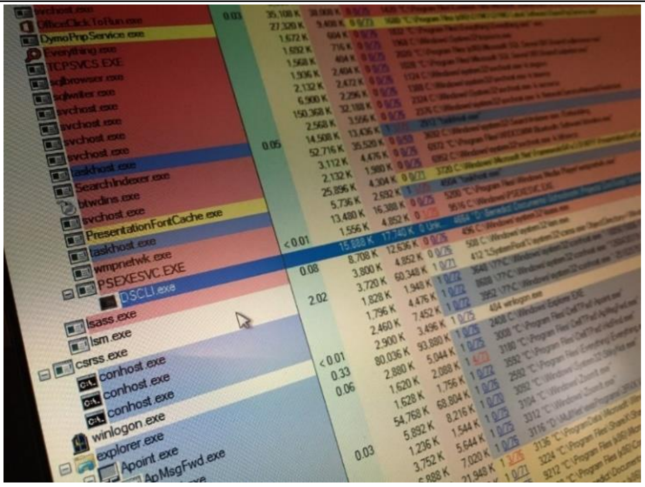
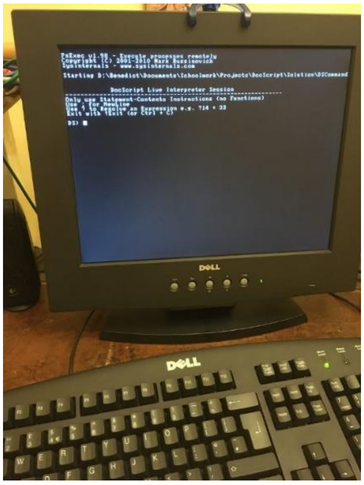
Remote Command-line Usage

Scenario: There is a terminal server and a client workstation on the same LAN. The user of the workstation wishes to use DocScript (via the command-line) *running on the Server*, but *from the Workstation's Console*.

How it's done: Of the four common ways of getting a remote command-prompt to another computer (TELNET, PowerShell Remoting, PsExec, and SSH), PsExec can be considered the easiest, as it requires no setup on either the client or server. To initiate the remote DocScript session, the user at the Workstation types:

```
PsExec.exe \\ServerHostname -u Domain\Username -p Password DSCLI.EXE /Live
```

I tested this with the interpreter running on my Latitude (acting as a server), and connecting to it over PsExec. I was then able to enter DocScript Instructions at the remote client computer (in this instance running Windows 2000) and type commands which were executed on the Server. The resultant output text was then sent over the network back to the client, and displayed.

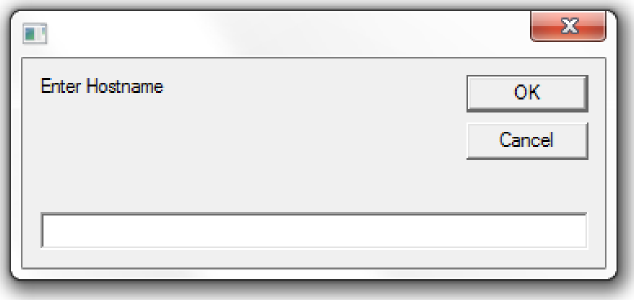
DSCLI.EXE running on the Latitude:	PsExec connecting to the Latitude Server:
	

```
PsExec v1.98 - Execute processes remotely
Copyright (C) 2001-2010 Mark Russinovich
Sysinternals - www.sysinternals.com

Starting D:\Benedict\Documents\Schoolwork\Projects\DocScript\Solution\DSCommand

              DocScript Live Interpreter Session
-------------------------------------------------------------
Only use Statement-Contents Instructions (no Functions)
Use ; for NewLine
Use ? to Resolve an Expression e.g. ?14 + 33
Exit with !Exit (or Ctrl + C)

DS> 
```

Review and Improvements Herefrom

- **IL-Merging**: It would be nice to not have to carry around *two* files, in order to use the supposedly simple command-line interpreter. The two files required at the moment are `DSCLI.EXE` and `DocScript.Library.DLL`. However, I can merge the underlying .NET MSIL Code from inside both binaries, into the same binary. This would mean that the user would only need to worry about the presence of a singular .EXE File. To achieve this, I simply run `ILMerge.exe` thusly:
  ```
  ILMerge.exe DSCLI.exe DocScript.Library.dll /out:ILM_DSCLI.exe
  ```
- **A pre-defined .VBS** file for easily initiating a remote DocScript session might be useful for some users who struggle to remember the syntax for PsExec. It could ask for the Server Hostname via

an InputBox, like this:

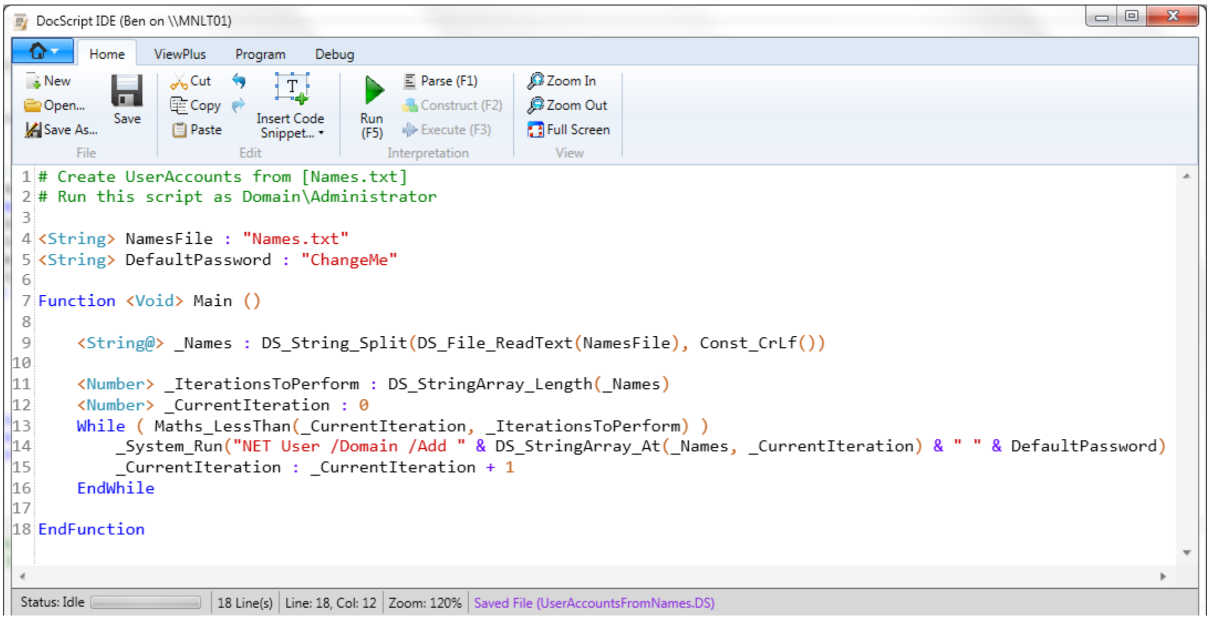


Enter Hostname

Administrative Scripting

Scenario: A network administrator wishes to create a quick script, to create 30 new user accounts on the domain, from a text file of names.

How it's done: The Windows IDE would be best-suited to this task, what with the Syntax-Highlighting, Program-Analysis features, and File-Interaction tools. The script might look something like this:

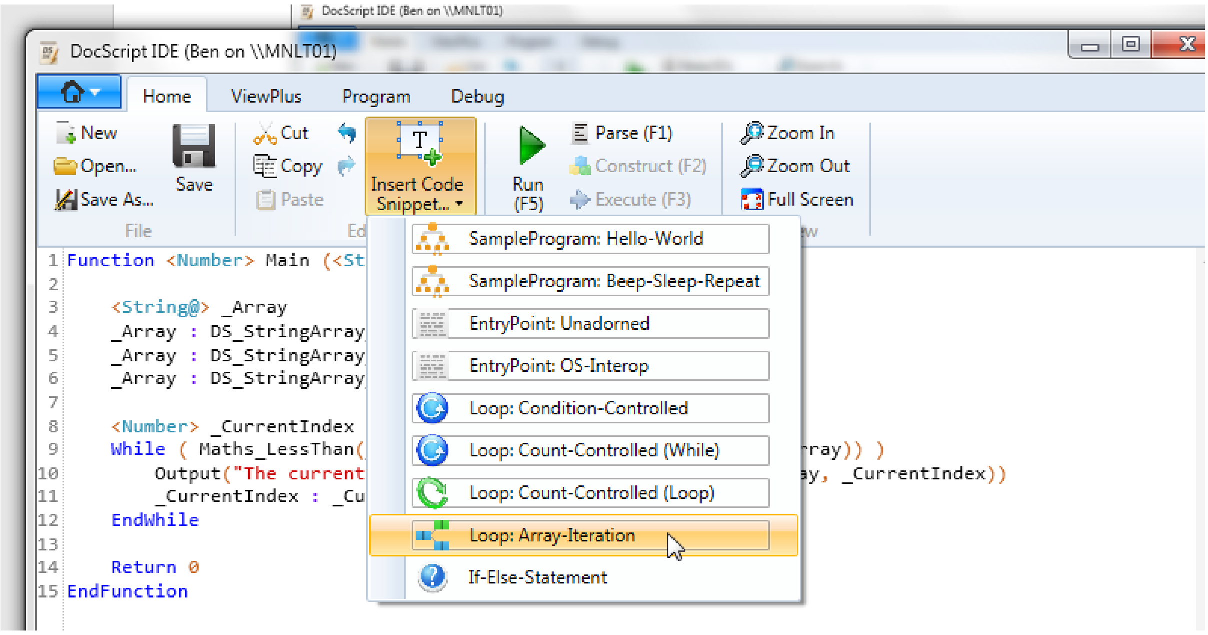


```
# Create UserAccounts from [Names.txt]
# Run this script as Domain\Administrator

<String> NamesFile : "Names.txt"
<String> DefaultPassword : "ChangeMe"

Function <Void> Main ()

    <String@> _Names : DS_String_Split(DS_File_ReadText(NamesFile), Const_CrLf())

    <Number> _IterationsToPerform : DS_StringArray_Length(_Names)
    <Number> _CurrentIteration : 0
    While ( Maths_LessThan(_CurrentIteration, _IterationsToPerform) )
        _System_Run("NET User /Domain /Add " & DS_StringArray_At(_Names, _CurrentIteration) & " " & DefaultPassword)
        _CurrentIteration : _CurrentIteration + 1
    EndWhile

EndFunction
```

Status: Idle | 18 Line(s) | Line: 18, Col: 12 | Zoom: 120% | Saved File (UserAccountsFromNames.DS)

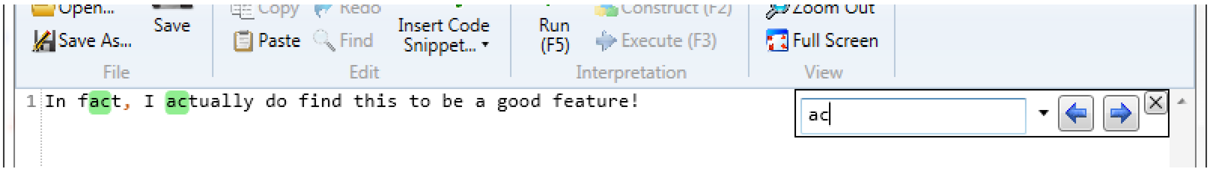
Review and Improvements Herefrom

- **Array Iteration**: It would have been useful to have an Array-Iteration Code-Snippet built-in to DSIDE, because it is a very common task. I shall therefore add this feature:



```
Function <Number> Main (<St

    <String@> _Array
    _Array : DS_StringArray
    _Array : DS_StringArray
    _Array : DS_StringArray

    <Number> _CurrentIndex
    While ( Maths_LessThan(                          ray)) )
        Output("The current                          y, _CurrentIndex))
        _CurrentIndex : _Cu
    EndWhile

    Return 0
EndFunction
```

- **A Find Dialog**: It would also be very useful to be able to have a "*Find*" dialog for the Text Editor, especially for highlighting identifiers which occur multiple times in the program. This feature is actually built-in to AvalonEdit, so I have just added a button to activate it:

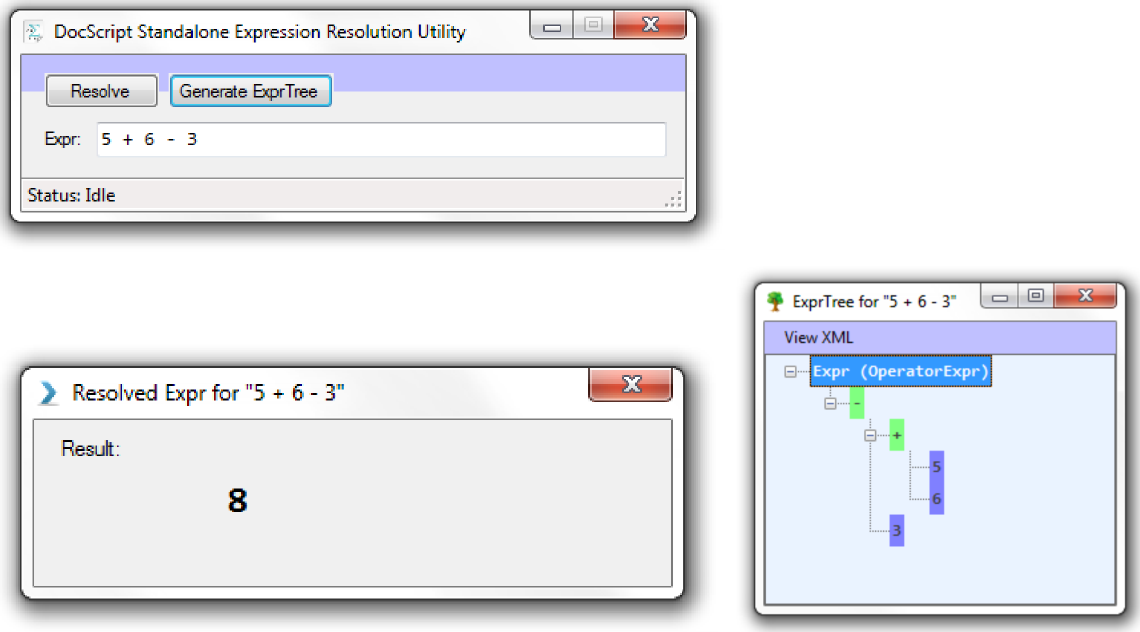


```
In fact, I actually do find this to be a good feature!
```

Mathematical-Expression Resolution

Scenario: An avid mathematician wishes to evaluate a series of mathematical expressions. He dosen't want to write an entire DocScript program, if he dosen't have to.

How it's done: DocScript supports two means of satisfying this need; *DocScript-Live*, and the *DocScript Standalone-Expression-Resolution-Utility*. Since the latter is written especially for this purpose, let's use it. Here's an example:



Expr: 5 + 6 - 3

Status: Idle

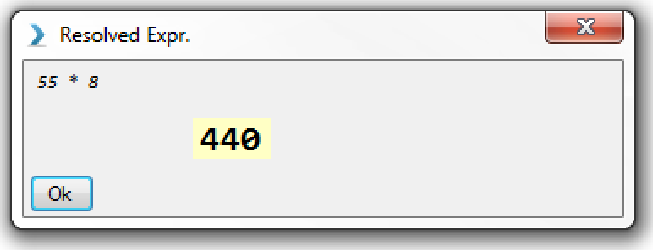
Result:

8

[Above] A Demonstration of DSExpr.exe

Review and Improvements Herefrom

Resolution Window Improvements: It would be useful to be able to quickly dismiss a resolved expression by simply pressing the enter key. The Result would also benefit from standing-out more from the background of the window. I have therefore altered the window thusly:



55 * 8

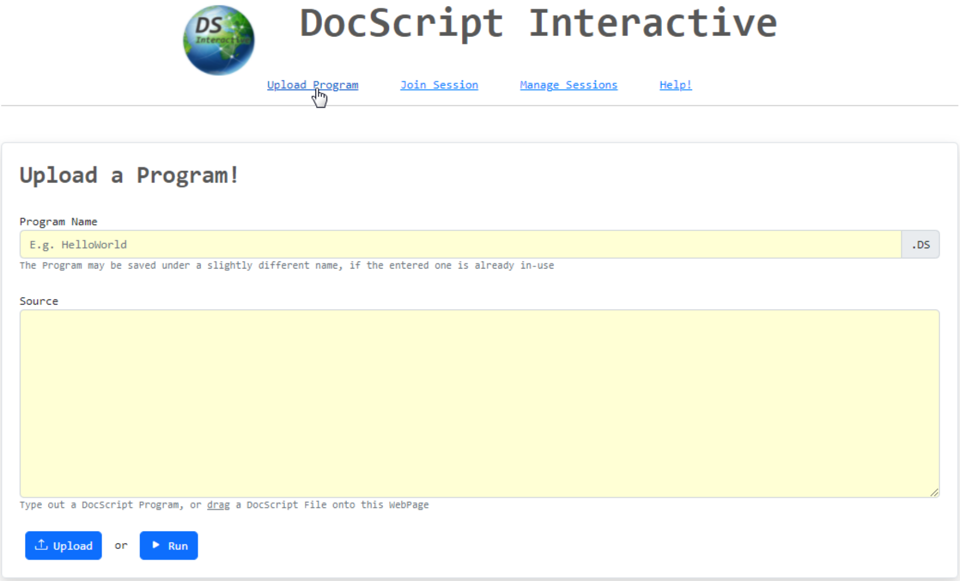
440

Interactive Multi-Client Execution

Scenario: A teacher has written a program, which she wishes to show her class. She wants all students to be able to participate in the execution of the program, in real-time. She wants specific students to enter input responses at given points during program execution.

How it's done: DocScript Interactive provides precisely this functionality. Here are the steps taken to configure and host the Execution-Session…

1. **Click "Upload Program", type-out or drag-in a DocScript Program, and click "Upload":**



Upload a Program!

Program Name

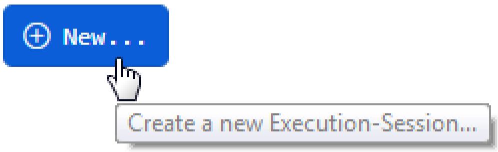
E.g. HelloWorld .DS

The Program may be saved under a slightly different name, if the entered one is already in-use

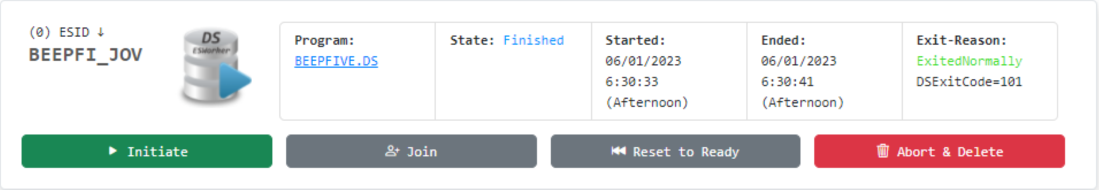
Source

Type out a DocScript Program, or drag a DocScript File onto this WebPage

2. **Create an Execution-Session for the Uploaded-Program with the *[+ New…]* button:**



3. **Get any clients (who want to participate in the session) to wait for initiation of the session on the landing-page, accessible via the *[+ Join…]* button:**



(0) ESID ↓
BEEPFI_JOV

Program: BEEPFIVE.DS

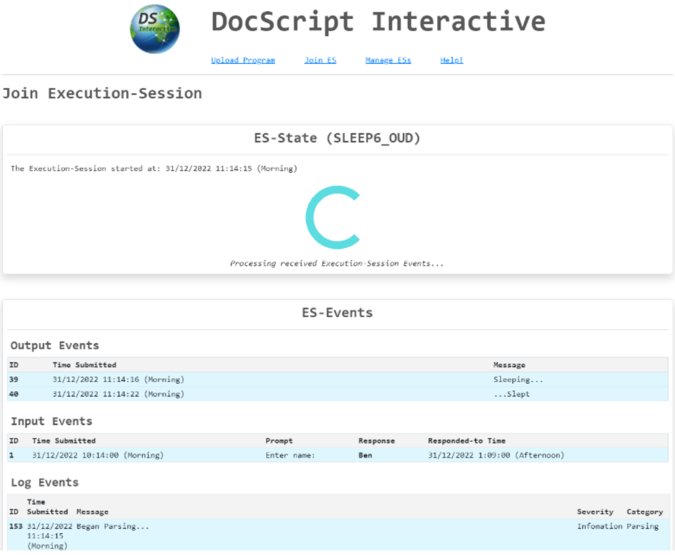
State: Finished

Started: 06/01/2023 6:30:33 (Afternoon)

Ended: 06/01/2023 6:30:41 (Afternoon)

Exit-Reason: ExitedNormally DSExitCode=101

4. **Initiate the session from the ESManager page from which it was created, and all waiting clients will immediately begin to receive Execution-Session Events:**



Join Execution-Session

ES-State (SLEEP6_OUD)

The Execution-Session started at: 31/12/2022 11:14:15 (Morning)

Processing received Execution-Session Events...

ES-Events

Output Events

ID	Time Submitted	Message
39	31/12/2022 11:14:16 (Morning)	Sleeping...
40	31/12/2022 11:14:22 (Morning)	...Slept

Input Events

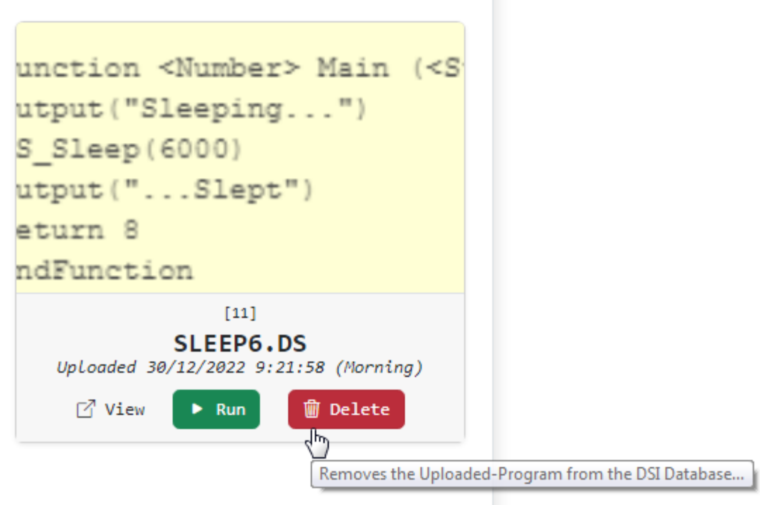
ID	Time Submitted	Prompt	Response	Responded-to Time
1	31/12/2022 10:14:00 (Morning)	Enter name:	Ben	31/12/2022 1:09:00 (Afternoon)

Log Events

ID	Time Submitted	Message	Severity	Category
153	31/12/2022 11:14:15 (Morning)	Began Parsing...	Infomation	Parsing

Review and Improvements Herefrom

CEP and Program Deletion: It would be useful to be able to delete Uploaded-Programs and CEPs, *from the web interface*, after they have been uploaded. At the moment, this has to be done from `ssms.exe`, directly in the database. Therefore, I have added the HTML components, JavaScript AJAX Functions, and Server-side API EndPoints necessary to perform the deletion of both of these types of DSI Objects, from the Database:



```
unction <Number> Main (<S
utput("Sleeping...")
S_Sleep(6000)
utput("...Slept")
eturn 8
ndFunction
```

[11]

SLEEP6.DS

Uploaded 30/12/2022 9:21:58 (Morning)

Removes the Uploaded-Program from the DSI Database...

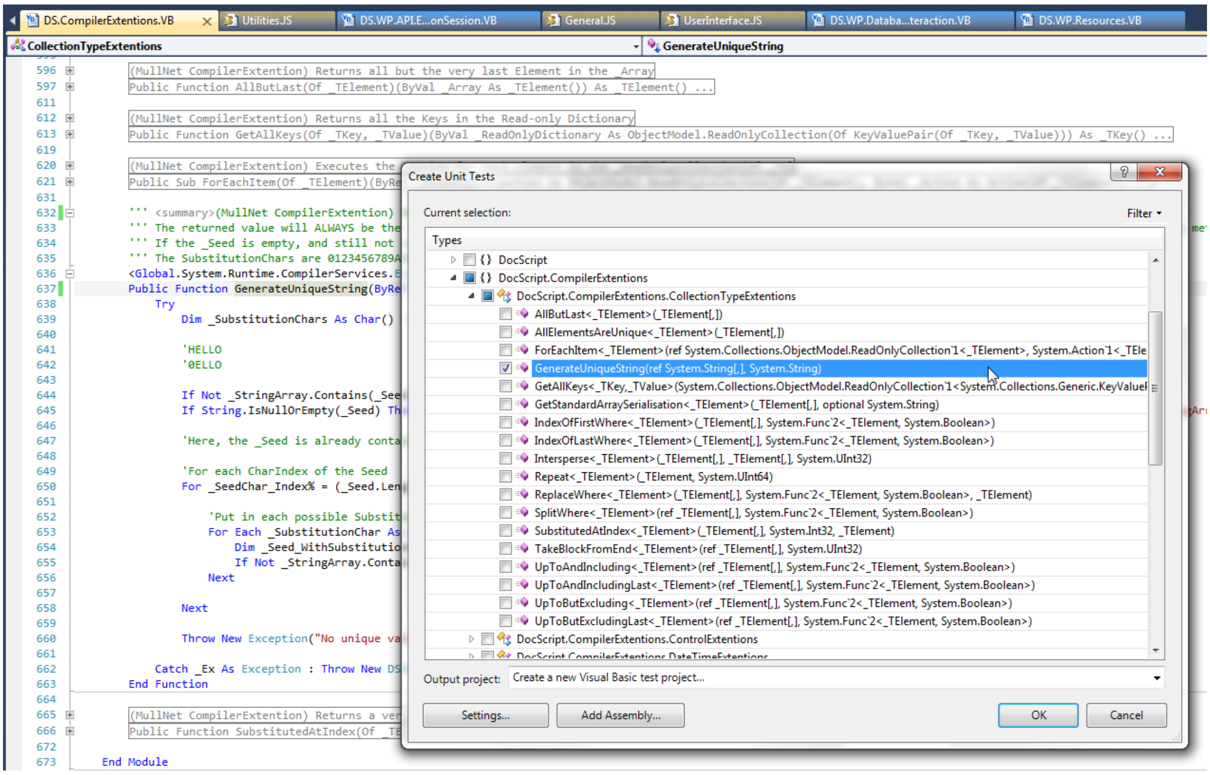
[...Stage 5...]

Unit-Testing

It would be more efficient to test many parts of the DocScript Solution via Unit Tests. These are more repeatable, consistent, and automatic, than the scenario-based or whole-program tests. If I cause an unintended side effect by changing one function, and this impairs the operation of another function, then the side-effect will clearly show up in the Unit Test results. Unit tests are particularly effective when there is a process which should take in a known input, in order to produce a known output. It would – for instance – be very difficult to write a Unit Test for a Function like `GetRandomNumber()`, but very easy for something like `Add(A, B)`. The unit tests for the entire solution can be run regularly, to ensure that no new additions and modifications have any side-effects on old parts of the solution, for which Unit Tests have been written.

As an example, the `GenerateUniqueString()` method (used to avoid name-collisions for Execution-Sessions in DocScript Interactive) should always produce the output `"PROGRA0"`, when given the Inputs `{"PROGRAM"}` and `"PROGRAM"`. (The parameters are [An array of already-taken strings] and [a seed, off of which to base the new, unique string].)

To create the Unit Test, I shall simply [Left-Click] → [Create Unit Tests…]



```
    (MullNet CompilerExtention) Returns all but the very last Element in the _Array
    Public Function AllButLast(Of _TElement)(ByVal _Array As _TElement()) As _TElement() ...

    (MullNet CompilerExtention) Returns all the Keys in the Read-only Dictionary
    Public Function GetAllKeys(Of _TKey, _TValue)(ByVal _ReadOnlyDictionary As ObjectModel.ReadOnlyCollection(Of KeyValuePair(Of _TKey, _TValue))) As _TKey() ...

    (MullNet CompilerExtention) Executes the
    Public Sub ForEachItem(Of _TElement)(ByRe

    ''' <summary>(MullNet CompilerExtention)
    ''' The returned value will ALWAYS be the
    ''' If the _Seed is empty, and still not
    ''' The SubstitutionChars are 0123456789A
    <Global.System.Runtime.CompilerServices.E
    Public Function GenerateUniqueString(ByRe
        Try
            Dim _SubstitutionChars As Char()

            'HELLO
            '0ELLO

            If Not _StringArray.Contains(_See
            If String.IsNullOrEmpty(_Seed) Th

            'Here, the _Seed is already conta

            'For each CharIndex of the Seed
            For _SeedChar_Index% = (_Seed.Len

                'Put in each possible Substit
                For Each _SubstitutionChar As
                    Dim _Seed_WithSubstitutio
                    If Not _StringArray.Conta
                Next

            Next

            Throw New Exception("No unique va

        Catch _Ex As Exception : Throw New DS
    End Function

    (MullNet CompilerExtention) Returns a ver
    Public Function SubstitutedAtIndex(Of _TE

End Module
```

Create Unit Tests

Current selection:

Types

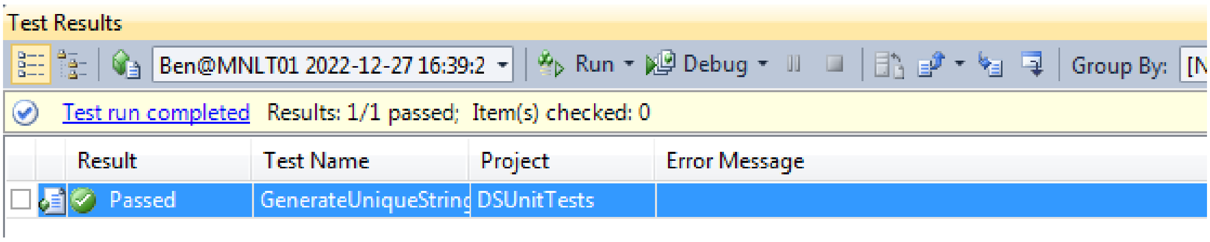
- □ {} DocScript
- ■ {} DocScript.CompilerExtentions
  - ■ DocScript.CompilerExtentions.CollectionTypeExtentions
    - □ AllButLast<_TElement>(_TElement[,])
    - □ AllElementsAreUnique<_TElement>(_TElement[,])
    - □ ForEachItem<_TElement>(ref System.Collections.ObjectModel.ReadOnlyCollection`1<_TElement>, System.Action`1<_TEle
    - ☑ GenerateUniqueString(ref System.String[,], System.String)
    - □ GetAllKeys<_TKey,_TValue>(System.Collections.ObjectModel.ReadOnlyCollection`1<System.Collections.Generic.KeyValueP
    - □ GetStandardArraySerialisation<_TElement>(_TElement[,], optional System.String)
    - □ IndexOfFirstWhere<_TElement>(_TElement[,], System.Func`2<_TElement, System.Boolean>)
    - □ IndexOfLastWhere<_TElement>(_TElement[,], System.Func`2<_TElement, System.Boolean>)
    - □ Intersperse<_TElement>(_TElement[,], _TElement[,], System.UInt32)
    - □ Repeat<_TElement>(_TElement, System.UInt64)
    - □ ReplaceWhere<_TElement>(_TElement[,], System.Func`2<_TElement, System.Boolean>, _TElement)
    - □ SplitWhere<_TElement>(ref _TElement[,], System.Func`2<_TElement, System.Boolean>)
    - □ SubstitutedAtIndex<_TElement>(_TElement[,], System.Int32, _TElement)
    - □ TakeBlockFromEnd<_TElement>(ref _TElement[,], System.UInt32)
    - □ UpToAndIncluding<_TElement>(ref _TElement[,], System.Func`2<_TElement, System.Boolean>)
    - □ UpToAndIncludingLast<_TElement>(ref _TElement[,], System.Func`2<_TElement, System.Boolean>)
    - □ UpToButExcluding<_TElement>(ref _TElement[,], System.Func`2<_TElement, System.Boolean>)
    - □ UpToButExcludingLast<_TElement>(ref _TElement[,], System.Func`2<_TElement, System.Boolean>)
  - □ DocScript.CompilerExtentions.ControlExtentions
  - □ DocScript.CompilerExtentions.DateTimeExtentions

Output project: Create a new Visual Basic test project...

This is what the Testing-Method looks like…

```
'''<summary>
'''A test for GenerateUniqueString()
'''</summary>
<TestMethod()> _
Public Sub GenerateUniqueStringTest()

    Dim _StringArray() As String = {"PROGRAM"}
    Dim _Seed As String = "PROGRAM"
    Dim expected As String = "PROGRA0"
    Dim actual As String = _
       DocScript.CompilerExtentions.CollectionTypeExtentions.GenerateUniqueString(_StringArray, _Seed)

    Assert.AreEqual(expected, actual)

End Sub
```

…And when I run the Test, I can see that the method works as I intend:



Test run completed Results: 1/1 passed; Item(s) checked: 0

	Result	Test Name	Project	Error Message
□	Passed	GenerateUniqueString	DSUnitTests	

[...Stage 5]

Stakeholder-Testing

Though, admittedly, I have to some extent *pandered to my own eccentric creative whims* in the development of this product, it is ultimately the Stakeholders, for whom this programming language system has been designed. I did a great deal of investigation at the beginning of the project, into what the needs of these stakeholders were.

To be rather summary, these needs were:

- Keep it simple – any good teaching tool ought to be easy to learn and use
- Enforce features found in more advanced higher-level languages, such as: DataTypes, Operators, Procedural Statements (while, if), functions or encapsulative units, built-in libraries or functions
- Having the ability to use the system on a wide variety of different sorts of computer systems (differing architectures and operating-systems)
- "Being able to get a lower-level view, of a high-level script you've written"; program analysis tools
- Being able to evaluate stand-alone expressions quickly and conveniently
- Interoperability with existing commonplace programming systems, such as the input of command-line arguments, and output of an exit code

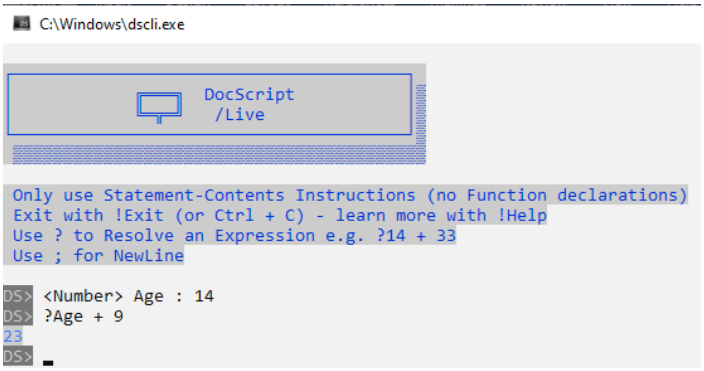
Stakeholder Feedback

I sent an Email (with the DocScript binaries attached) to the 4 primary stakeholders, asking the following of them:

- First impressions?
- Were you – without additional guidance – able to navigate through the program, and find the features you were looking for? *(Usability)*
- How stable was the software?
- How quickly were you able to get up-and-running?
- Improvements?

The gist of the responses was as follows: They generally found the product **easy-to-use**, **performant**, and **feature-rich**. Subject to particular adulation, were the following features:

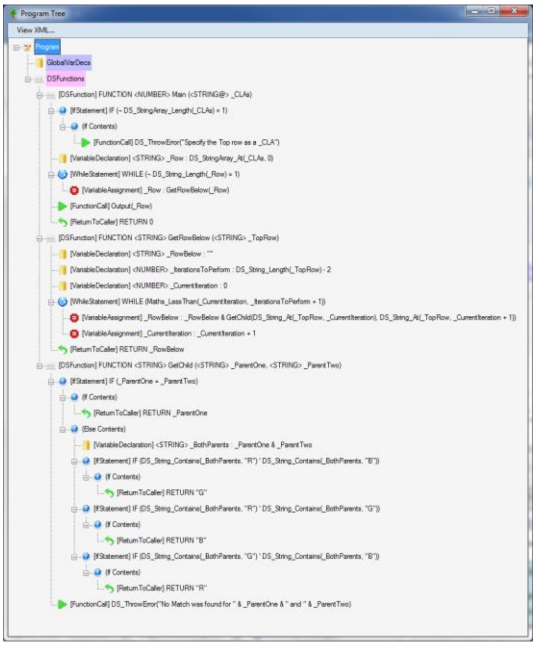
- The `DSCLI /Live` mode. Respondents actually indicated that this was probably more convenient for expression evaluation, than the `DSExpr` program which has been specifically designed for this purpose, owing to the inline and single-window nature of the console application:



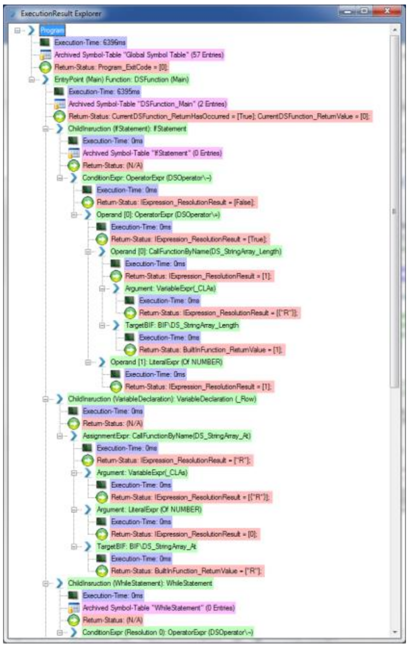
```
C:\Windows\dscli.exe

        DocScript
         /Live

Only use Statement-Contents Instructions (no Function declarations)
Exit with !Exit (or Ctrl + C) - learn more with !Help
Use ? to Resolve an Expression e.g. ?14 + 33
Use ; for NewLine

DS> <Number> Age : 14
DS> ?Age + 9
23
DS> _
```

↑ One of the stakeholders, decadently caught using Windows 10. (Fortunately, DSCLI still worked!)

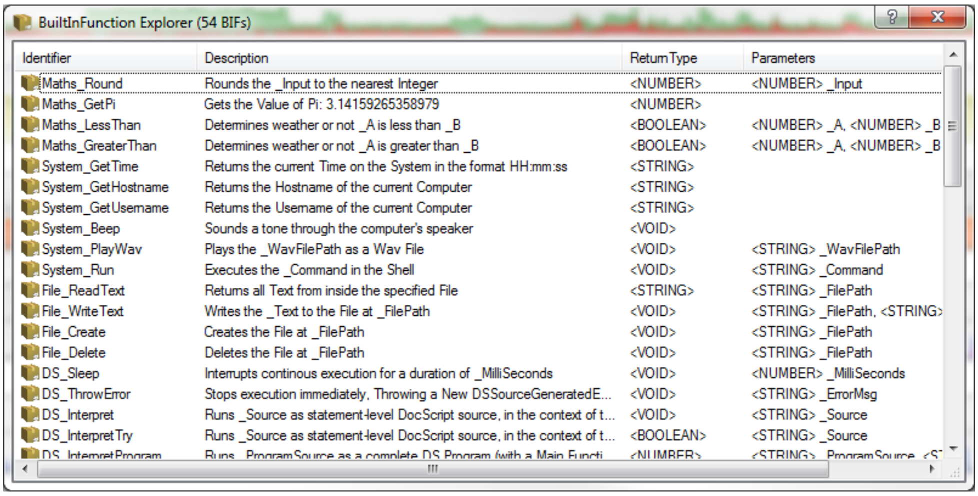
- The Program-Analysis dialogs in the IDE. These provided a more in-depth means of understanding what the interpretation engine actually sees, from the typed-out source:



```
Program
  GlobalVarDecs
  DSFunctions
    [DSFunction] FUNCTION <NUMBER> Main (<STRING@> _CLAs)
      [IfStatement] IF (~ DS_StringArray_Length(_CLAs) = 1)
        (If Contents)
          [FunctionCall] DS_ThrowError("Specify the Top row as a _CLA")
      [VariableDeclaration] <STRING> _Row : DS_StringArray_At(_CLAs, 0)
      [WhileStatement] WHILE (~ DS_String_Length(_Row) = 1)
        [VariableAssignment] _Row : GetRowBelow(_Row)
      [FunctionCall] Output(_Row)
      [ReturnToCaller] RETURN 0
    [DSFunction] FUNCTION <STRING> GetRowBelow (<STRING> _TopRow)
      [VariableDeclaration] <STRING> _RowBelow : ""
      [VariableDeclaration] <NUMBER> _IterationsToPerform : DS_String_Length(_TopRow) - 2
      [VariableDeclaration] <NUMBER> _CurrentIteration : 0
      [WhileStatement] WHILE (Maths_LessThan(_CurrentIteration, _IterationsToPerform + 1))
        [VariableAssignment] _RowBelow : _RowBelow & GetChild(DS_String_At(_TopRow, _CurrentIteration), DS_String_At(_TopRow, _CurrentIteration + 1))
        [VariableAssignment] _CurrentIteration : _CurrentIteration + 1
      [ReturnToCaller] RETURN _RowBelow
    [DSFunction] FUNCTION <STRING> GetChild (<STRING> _ParentOne, <STRING> _ParentTwo)
      [IfStatement] IF (_ParentOne = _ParentTwo)
        (If Contents)
          [ReturnToCaller] RETURN _ParentOne
        (Else Contents)
          [VariableDeclaration] <STRING> _BothParents : _ParentOne & _ParentTwo
          [IfStatement] IF (DS_String_Contains(_BothParents, "R") ' DS_String_Contains(_BothParents, "B"))
            (If Contents)
              [ReturnToCaller] RETURN "G"
          [IfStatement] IF (DS_String_Contains(_BothParents, "R") ' DS_String_Contains(_BothParents, "G"))
            (If Contents)
              [ReturnToCaller] RETURN "B"
          [IfStatement] IF (DS_String_Contains(_BothParents, "G") ' DS_String_Contains(_BothParents, "B"))
            (If Contents)
              [ReturnToCaller] RETURN "R"
      [FunctionCall] DS_ThrowError("No Match was found for " & _ParentOne & " and " & _ParentTwo)
```



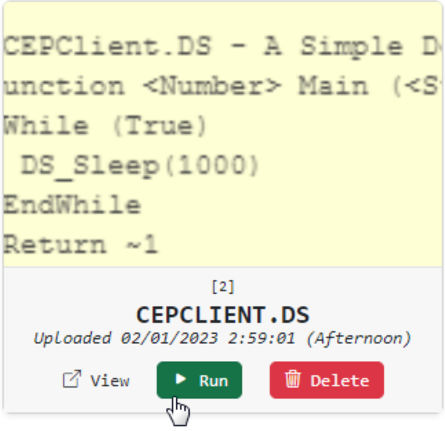
```
Program
  Execution-Time: 6396ms
  Archived Symbol-Table "Global Symbol Table" (57 Entries)
  Return-Status: Program_ExitCode = [0];
  EntryPoint (Main) Function: DSFunction (Main)
    Execution-Time: 6395ms
    Archived Symbol-Table "DSFunction_Main" (2 Entries)
    Return-Status: CurrentDSFunction_ReturnHasOccured = [True]; CurrentDSFunction_ReturnValue = [0];
    ChildInsruction (IfStatement): IfStatement
      Execution-Time: 0ms
      Archived Symbol-Table "IfStatement" (0 Entries)
      Return-Status: (N/A)
      ConditionExpr: OperatorExpr (DSOperator\~)
        Execution-Time: 0ms
        Return-Status: IExpression_ResolutionResult = [False];
        Operand [0]: OperatorExpr (DSOperator\=)
          Execution-Time: 0ms
          Return-Status: IExpression_ResolutionResult = [True];
          Operand [0]: CallFunctionByName(DS_StringArray_Length)
            Execution-Time: 0ms
            Return-Status: IExpression_ResolutionResult = [1];
            Argument: VariableExpr(_CLAs)
              Execution-Time: 0ms
              Return-Status: IExpression_ResolutionResult = [["R"]];
            TargetBIF: BIF\DS_StringArray_Length
              Execution-Time: 0ms
              Return-Status: BuiltInFunction_ReturnValue = [1];
          Operand [1]: LiteralExpr (Of NUMBER)
            Execution-Time: 0ms
            Return-Status: IExpression_ResolutionResult = [1];
    ChildInsruction (VariableDeclaration): VariableDeclaration (_Row)
      Execution-Time: 0ms
      Return-Status: (N/A)
      AssignmentExpr: CallFunctionByName(DS_StringArray_At)
        Execution-Time: 0ms
        Return-Status: IExpression_ResolutionResult = ["R"];
        Argument: VariableExpr(_CLAs)
          Execution-Time: 0ms
          Return-Status: IExpression_ResolutionResult = [["R"]];
        Argument: LiteralExpr (Of NUMBER)
          Execution-Time: 0ms
          Return-Status: IExpression_ResolutionResult = [0];
        TargetBIF: BIF\DS_StringArray_At
          Execution-Time: 0ms
          Return-Status: BuiltInFunction_ReturnValue = ["R"];
    ChildInsruction (WhileStatement): WhileStatement
      Execution-Time: 0ms
      Archived Symbol-Table "WhileStatement" (0 Entries)
      Return-Status: (N/A)
      ConditionExpr (Resolution 0): OperatorExpr (DSOperator\~)
```



BuiltInFunction Explorer (54 BIFs)

Identifier	Description	Return Type	Parameters
Maths_Round	Rounds the _Input to the nearest Integer	<NUMBER>	<NUMBER> _Input
Maths_GetPi	Gets the Value of Pi: 3.14159265358979	<NUMBER>	
Maths_LessThan	Determines weather or not _A is less than _B	<BOOLEAN>	<NUMBER> _A, <NUMBER> _B
Maths_GreaterThan	Determines weather or not _A is greater than _B	<BOOLEAN>	<NUMBER> _A, <NUMBER> _B
System_GetTime	Returns the current Time on the System in the format HH:mm:ss	<STRING>	
System_GetHostname	Returns the Hostname of the current Computer	<STRING>	
System_GetUsername	Returns the Username of the current Computer	<STRING>	
System_Beep	Sounds a tone through the computer's speaker	<VOID>	
System_PlayWav	Plays the _WavFilePath as a Wav File	<VOID>	<STRING> _WavFilePath
System_Run	Executes the _Command in the Shell	<VOID>	<STRING> _Command
File_ReadText	Returns all Text from inside the specified File	<STRING>	<STRING> _FilePath
File_WriteText	Writes the _Text to the File at _FilePath	<VOID>	<STRING> _FilePath, <STRING>
File_Create	Creates the File at _FilePath	<VOID>	<STRING> _FilePath
File_Delete	Deletes the File at _FilePath	<VOID>	<STRING> _FilePath
DS_Sleep	Interrupts continous execution for a duration of _MilliSeconds	<VOID>	<NUMBER> _MilliSeconds
DS_ThrowError	Stops execution immediately, Throwing a New DSSourceGeneratedE...	<VOID>	<STRING> _ErrorMsg
DS_Interpret	Runs _Source as statement-level DocScript source, in the context of t...	<VOID>	<STRING> _Source
DS_InterpretTry	Runs _Source as statement-level DocScript source, in the context of t...	<BOOLEAN>	<STRING> _Source
DS_InterpretProgram	Runs _ProgramSource as a complete DS Program (with a Main Functi	<NUMBER>	<STRING> _ProgramSource, <S

- The ability to create ad-hoc ExecutionSessions from the UploadProgram page in DSI:



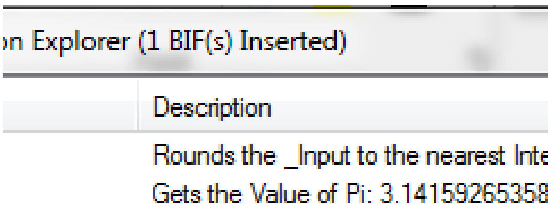
```
CEPClient.DS - A Simple D
unction <Number> Main (<S
While (True)
 DS_Sleep(1000)
EndWhile
Return ~1
```

[2]
CEPCLIENT.DS
Uploaded 02/01/2023 2:59:01 (Afternoon)

  (This is easier than manually creating an execution session from the ExecutionSession-Manager page.)

To desist from any more self-flattery however, it behoves me to enumerate – in some detail - the **shortcomings** pointed out by the stakeholders:

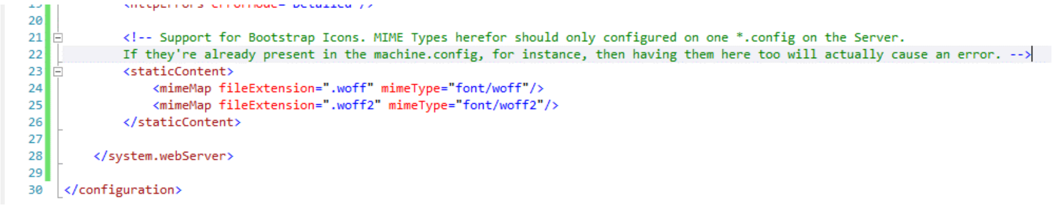
- **Oliver:** "*When I click on a Built-in Function in the BIF Explorer Window, it dosen't do anything. I would expect it to do something!*" **Remedial Actions Taken and Justification:** I added an EventHandler which inserts a TemplateCall of the selected BIFs, into the TextEditor of the Main IDE Window. The TemplateCall is a new Property on the BIF Class, included specifically for this problem. E.g. for the Output BIF, the TemplateCall would look like `Output(_Text)`. The BIFs Window also now reports how many BIFs have been inserted:



on Explorer (1 BIF(s) Inserted)

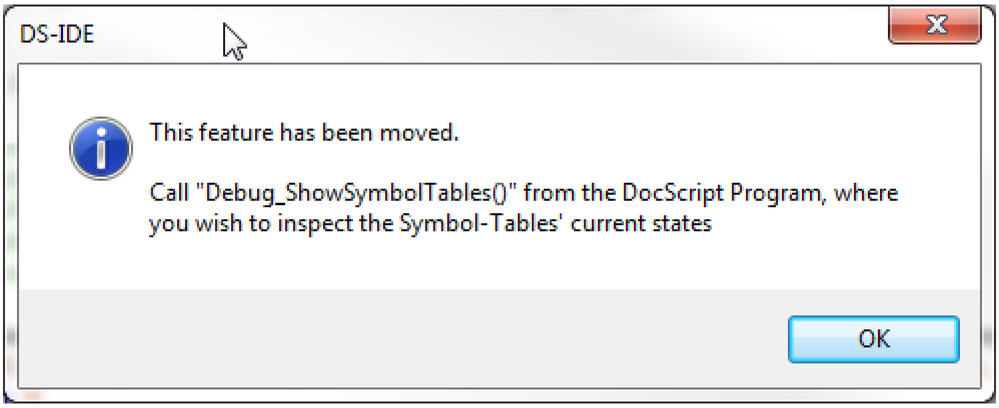
	Description
	Rounds the _Input to the nearest Inte
	Gets the Value of Pi: 3.14159265358

- **Kiran:** "*I wanted to try out hosting DSInteractive on my personal machine, but there was an IIS Web.config error with the WOFF MIME Types for the Bootstrap Fonts – what's that about?*" **Remedial Actions Taken and Justification:** This occurs because the Operating System I was developing on was running Server 2008 R2 – which *dosen't* include the WOFF MIME Type in the base-most Web.config – whereas this stakeholder was running Windows 11 Enterprise, which *does* include this as a default MIME Type. I can't think of any simple workaround for this problem, except from making an inline comment in the Web.config which ships with DocScript:



```
        <httpErrors errorMode="Detailed" />

        <!-- Support for Bootstrap Icons. MIME Types herefor should only configured on one *.config on the Server.
        If they're already present in the machine.config, for instance, then having them here too will actually cause an error. -->
        <staticContent>
            <mimeMap fileExtension=".woff" mimeType="font/woff"/>
            <mimeMap fileExtension=".woff2" mimeType="font/woff2"/>
        </staticContent>

    </system.webServer>

</configuration>
```

- **Joe:** "*I attempted to use the [View Symbol Tables...] Button in DSIDE, but it didn't do anything. What's it for?*" **Remedial Actions Taken and Justification:** I had moved the functionality of this button into a DocScript BuiltInFunction, accessible from the DS Source using `Debug_ShowSymbolTables()`. I had forgotten – however – to disable or remove this Button. I decided against removal however, on the grounds that users are unlikely to discover a feature – albeit a very useful one – which they cannot see. Instead, I made the Button show this MessageBox:

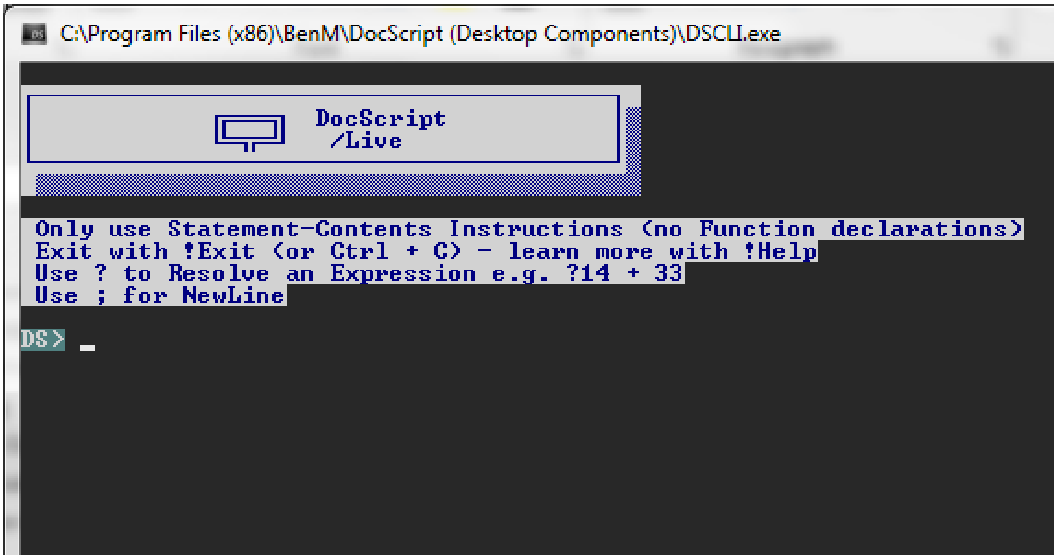


DS-IDE

This feature has been moved.

Call "Debug_ShowSymbolTables()" from the DocScript Program, where you wish to inspect the Symbol-Tables' current states

- **Klara:** "*Ich habe festgestellt, dass die Textausgabe von DSCLI manchmal unlesbar war, wenn es eine Standard-BackColour für die Konsole gab, die der Vordergrundfarbe des Ausgabetexts entsprach.*" **Remedial Actions Taken and Justification:** This was occurring because DSCLI – in an attempt to be more user-friendly – uses different console foreground colours where possible (in certain environments such as under PsExec, it is not supported). The problem was that the foreground text colour was the same as the console's Background Colour, meaning that the text was effectively unreadable. To solve this, I need to set the Background Colour too. That makes DSCLI look like this...



C:\Program Files (x86)\BenM\DocScript (Desktop Components)\DSCLI.exe

```
DocScript
 /Live

Only use Statement-Contents Instructions (no Function declarations)
Exit with !Exit (or Ctrl + C) - learn more with !Help
Use ? to Resolve an Expression e.g. ?14 + 33
Use ; for NewLine

DS> _
```

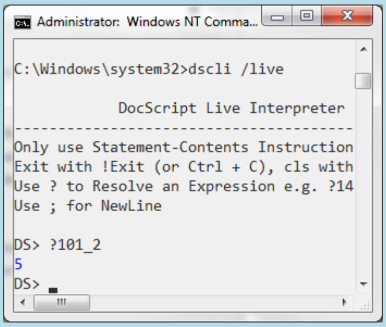
...a little odd on a darker console, but at least it will **always** be readable now.

Final Prototype

The Solution is now in a near-final state.

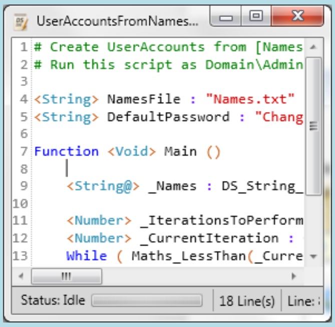
Prototype: This point marks ***Release Candidate 1*** of the product, whose noteworthy features are…

- Natively supporting Numeric-Literals of different bases in DocScript source. For example, `101_2` and `5_10` and `11_4` and `5` and `5.0000` all equate to the same numerical value, and are all valid DS Expressions.
- Interpreting DocScript Programs, in either a Command-Line, Windows-Program, or Web-Browser;



```
C:\Windows\system32>dscli /live

             DocScript Live Interpreter
-----------------------------------------------
Only use Statement-Contents Instruction
Exit with !Exit (or Ctrl + C), cls with
Use ? to Resolve an Expression e.g. ?14
Use ; for NewLine

DS> ?101_2
5
DS> _
```

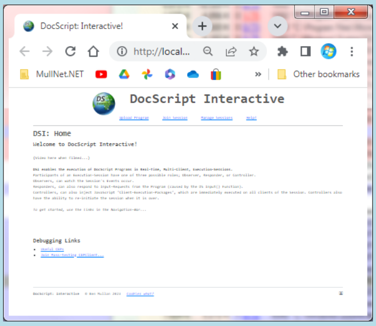
or



```
# Create UserAccounts from [Names
# Run this script as Domain\Admin

<String> NamesFile : "Names.txt"
<String> DefaultPassword : "Chang

Function <Void> Main ()

    <String@> _Names : DS_String_

    <Number> _IterationsToPerform
    <Number> _CurrentIteration :
    While ( Maths_LessThan(_Curre
```

Status: Idle | 18 Line(s) | Line:

or



DocScript Interactive

Upload Program | Join Session | Manage Sessions | Help!

DSI: Home

Welcome to DocScript Interactive!

{Video here when filmed...}

DSI enables the execution of DocScript Programs in Real-Time, Multi-Client, Execution-Sessions.
Participants of an Execution-Session have one of three possible roles; Observer, Responder, or Controller.
Observers, can watch the Session's Events occur.
Responders, can also respond to Input-Requests from the Program (caused by the DS Input() Function).
Controllers, can also inject JavaScript "Client-Execution-Packages", which are immediately executed on all clients of the session. Controllers also have the ability to re-initiate the session when it is over.

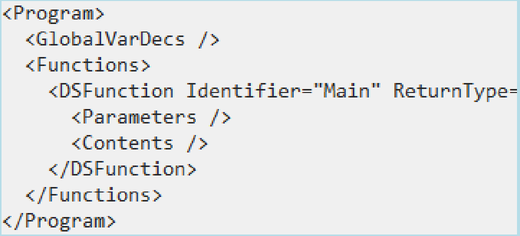
To get started, use the links in the Navigation-Bar...

Debugging Links

- Useful CEPs
- Join Mass-testing CEPClient...

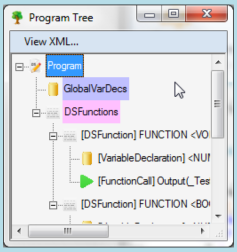
DocScript: Interactive © Ben Mullan 2023 Cookies what?

- Providing advanced program-analysis features, including Program-Tree generation (XML or GUI), and Execution-Result exploration;



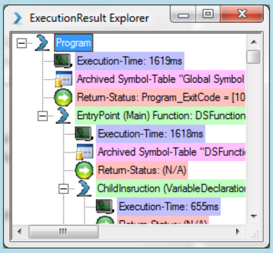
```
<Program>
  <GlobalVarDecs />
  <Functions>
    <DSFunction Identifier="Main" ReturnType=
      <Parameters />
      <Contents />
    </DSFunction>
  </Functions>
</Program>
```

or



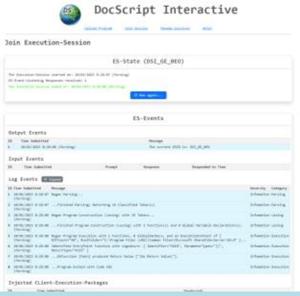
```
Program
  GlobalVarDecs
  DSFunctions
    [DSFunction] FUNCTION <VO
      [VariableDeclaration] <NU
      [FunctionCall] Output(_Tes
    [DSFunction] FUNCTION <BO
```

and

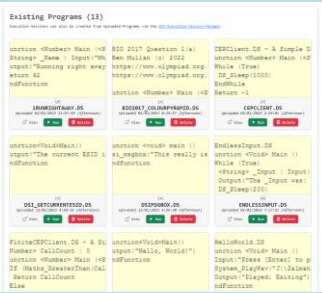


```
Program
  Execution-Time: 1619ms
  Archived Symbol-Table "Global Symbol
  Return-Status: Program_ExitCode = [10
  EntryPoint (Main) Function: DSFunction
    Execution-Time: 1618ms
    Archived Symbol-Table "DSFuncti
    Return-Status: (N/A)
    ChildInsruction (VariableDeclaratio
      Execution-Time: 655ms
```

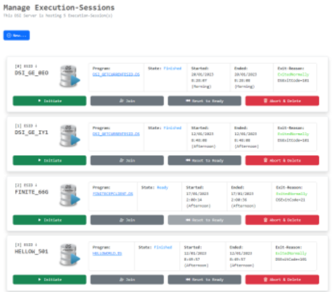
- Hosting real-time multi-client Execution-Sessions, for the collaborative execution of DocScript programs with multiple participating clients;



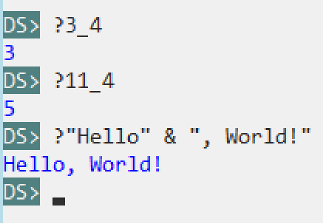
and



and

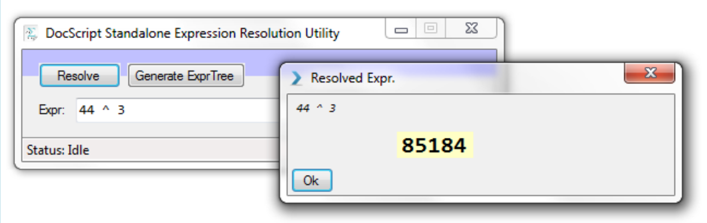


- Facilitating standalone expression-resolution in both CLI and GUI environments:



```
DS> ?3_4
3
DS> ?11_4
5
DS> ?"Hello" & ", World!"
Hello, World!
DS> _
```

and



Expr: 44 ^ 3

Status: Idle

44 ^ 3

85184

Development Review

Progress Recap: Where am I in the development plan? *[Review]*

- **Done**: I have written the DocScript Interpreter, and the three different implementations thereof.
- **Next**: There is more testing and evaluation to be done…

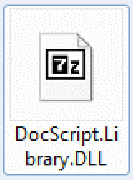
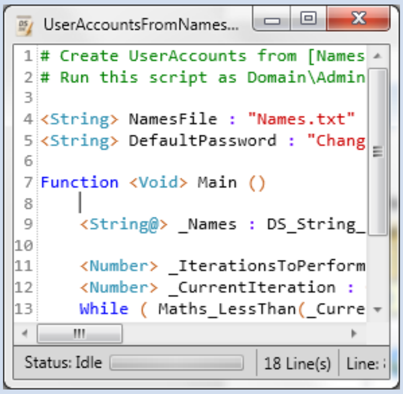
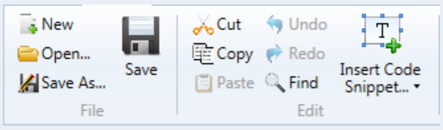
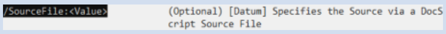
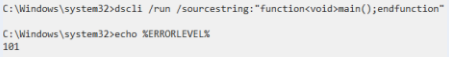
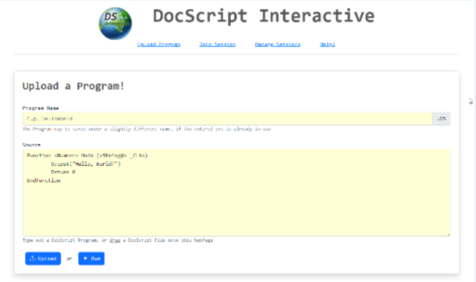
Revisiting the Requirements and Success Criteria Tables

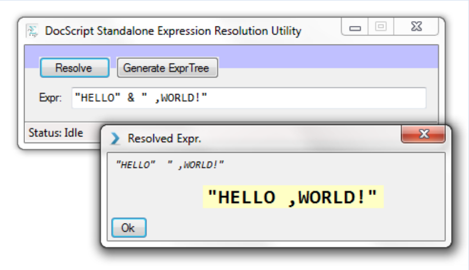
Testing Table: Does this component function in accordance with the stipulated criteria?
I will now test the Final Prototype against the initial testing criteria, which were delineated back in §Analysis.

Here, I have conflated the three Requirements and Criteria Tables from §Analysis, into one table.

Have the components I designed met the stakeholders' requirements?

Category	Related Criterium (Analysis)	☑ Passed?
(Language)	A Programming Language Formal Specification (*LangSpec*)	↓
(Language)	Serviceable **Language Features** including:	↓
(Language)	Data Types	Yes; the language includes this feature: `<String> Name : "Ben"`
(Language)	Predefined Functions (for common tasks)	Yes; the language includes this feature: `System_Run("CMD.EXE")`
(Language)	Encapsulative Units (E.g. Functions)	Yes; the language includes this feature: `Function <Void> Main ()…`
(Language)	Procedural Programming Constructs (While; If; etc…)	Yes; the language includes this feature: `While (Expr)…`
(Language)	OS Interoperability (CLAs & Exit Codes)	Yes; the language includes this feature: `_CLAs`
(Language)	Specialist **Mathematical Features**:	↓
(Language)	BasedNumber Manipulation	Yes; the language includes this feature: `<Number> Age : 101_2`
(Language)	Boolean Logic features	Yes; the language includes this feature: `<Boolean> B : A|B`
(Language)	Maths Expression Evaluation	Yes; the language includes this feature: `Age : 4+5/2^[~9]`
(Implementation)	A Programming Language Runtime (*engine*) which can execute Scripts	Yes:

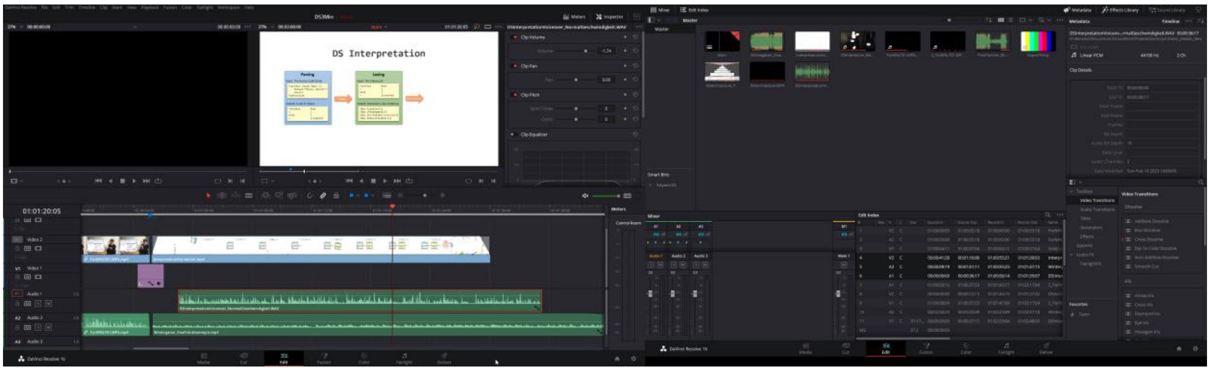
		
(Implementation)	[Windows GUI] An IDE with text-editing and script-running abilities	Yes:  # Create UserAccounts from [Names # Run this script as Domain\Admin <String> NamesFile : "Names.txt" <String> DefaultPassword : "Chang Function <Void> Main () <String@> _Names : DS_String_ <Number> _IterationsToPerform <Number> _CurrentIteration : While (Maths_LessThan(_Curre Status: Idle | 18 Line(s) | Line:
(Implementation)	[Windows GUI] A simple, familiar graphical design	Yes:  New Open... Save As... Save File Cut Copy Paste Undo Redo Find Insert Code Snippet... Edit
(Implementation)	[Windows CLI] Takes a Command-Line argument for the script to run	Yes:  /SourceFile:<Value> (Optional) [Datum] Specifies the Source via a DocS cript Source File
(Implementation)	[Windows CLI] Returns the Exit Code of the Script just run	Yes:  C:\Windows\system32>dscli /run /sourcestring:"function<void>main();endfunction" C:\Windows\system32>echo %ERRORLEVEL% 101
(Implementation)	[Web Client] Runs on a variety of different browsers on different devices	Yes: Google Chrome (~v50+) Apple Safari Moz://a Firefox The API itself can actually be used from any HTTP-compatible browser, including the likes of Internet Explorer 1.0
(Implementation)	[Web Client] Allows the user to enter source code into a text field and thereafter execute it	Yes:  DocScript Interactive Upload Program Join Session Manage Sessions Help! Upload a Program! Program Name E.g. HelloWorld .DS The Program may be saved under a slightly different name, if the entered one is already in use Source Function <Number> Main (<String@> _CLAs) Output("Hello, World!") Return 0 EndFunction Type out a DocScript Program, or drag a DocScript File onto this WebPage
(Usability)	Singular Expressions can be evaluated independently (without having to run a whole script)	Yes:

		 DocScript Standalone Expression Resolution Utility Resolve Generate ExprTree Resolved Expr. Ok Expr: "HELLO" & " ,WORLD!" Status: Idle "HELLO" " ,WORLD!" "HELLO ,WORLD!"
(Usability)	Instructions and Documentation is organised and easy to find	Partially: There is a page in DSI (the web system for DocScript) which contains help and basic instructions, and there is a help dialog in the IDE program. However: this is not really sufficient. Therefore, I require an easy-to-access, quick form of help, which enables users and newcomers to get to grips with DocScript. To this end, I have decided at this point to film a short "**DocScript in 3 Minutes**" video, explaining how the system works. In addition, on account of the stakeholders' comments, I shall implement a ***pictorial help*** feature in the DSIDE product, to more visually explain the basic DocScript concepts, required to start writing DS Programs.
(Usability)	The Language is easy to learn and use	{See forthcoming Stakeholder-Feedback section…}

DocScript in 3 Minutes

It was incidentally at this point, that the "*in 3 Minutes*" video was filmed and edited…

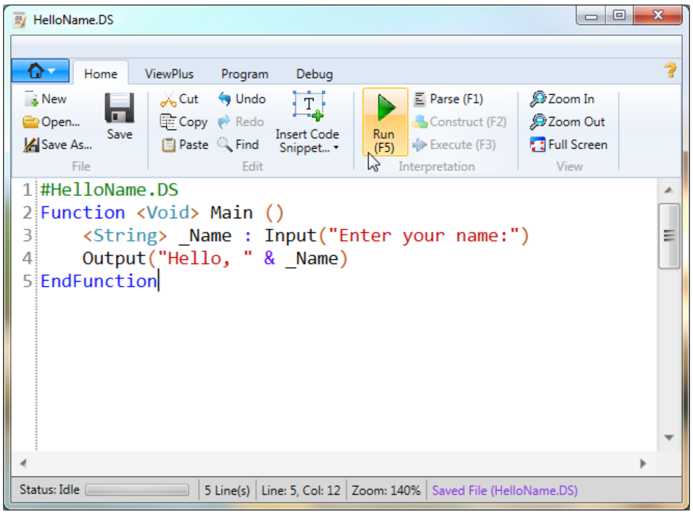
(*https://youtu.be/ybl5pVSJOOk*)



Testing and Evaluation

Overview

On completing *§Development and Testing*, I now have a functional set of software products, which – as has been proven by the testing-to-inform-development – go at least some of the way towards meeting the criteria and stakeholder needs.



```
#HelloName.DS
Function <Void> Main ()
    <String> _Name : Input("Enter your name:")
    Output("Hello, " & _Name)
EndFunction
```

[Above] *DocScript^{IDE}* with an example program

What remains, is to…

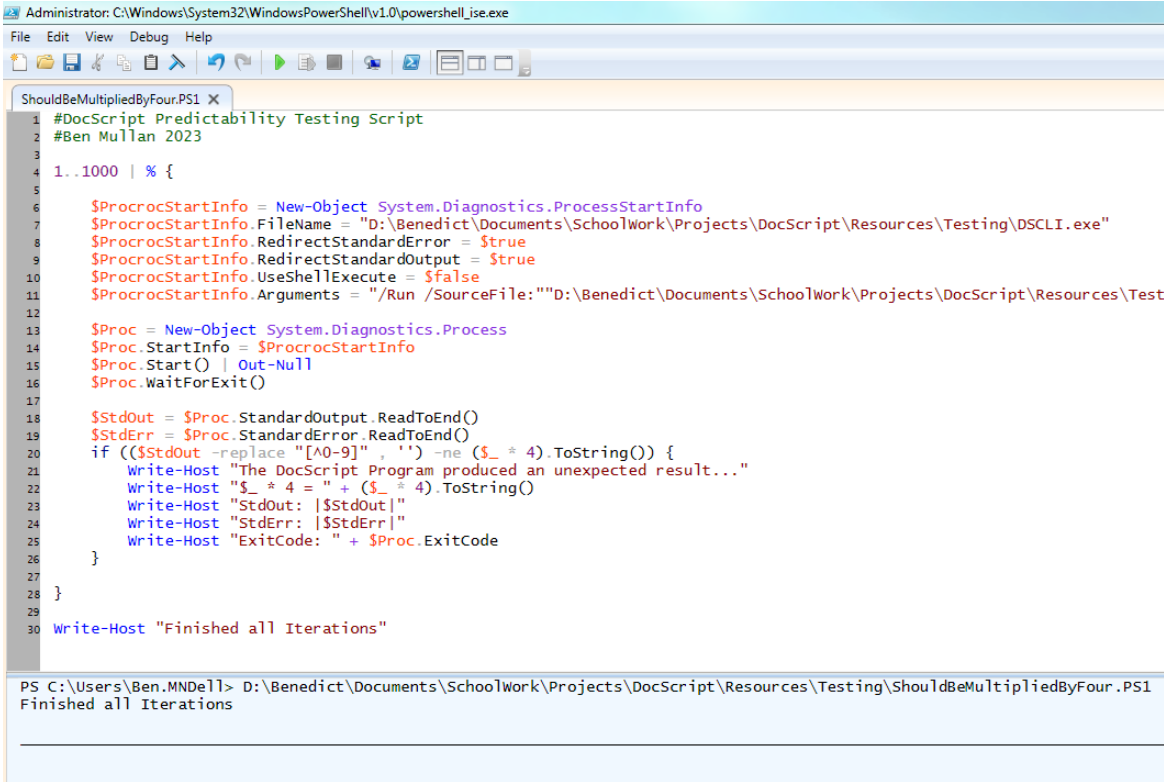
- …More thoroughly test the **reliability and robustness** of the DocScript software.
- …Assess the **usability** of *[a]* each individual component, and then *[b]* how effectually the components fit together. Then, I will evidence these usability features, *justifying* their success, and *commenting*, on how any as-yet unmet or incomplete features may be added in the future.
- …Determine – for each **success criterium** defined in the *Analysis* and *Design* stages – whether it can be proved to have been met, by collected evidence.
- …Remark on how any unmet criteria might be addressed, in the future.
- …Declare current issues with the **maintenance** of the solution.
- …Revisit the anticipated **limitations** from the *Design* stage, describing how they have been tackled, and how any as-yet insuperable limitations might be overcome in the future.
- …Check-in with the Stakeholders for the last time, ensuring that they are (largely) satisfied.

Reliability and Robustness

One of the primary stipulations of the stakeholders, had been that the DocScript system must necessarily function across a number of "*different platforms, architectures, operating systems, and environments*". Therefore, it is especially important that the software is dependable and reliable, because the environments under which it is to run, cannot be guaranteed to be consistent. To this end, I shall now test that this diversity of robust functionality does indeed work as I have described it.

Predictability

I wrote a quick script to run 1000 instances of a DocScript program, which takes in a command-line argument as an integer, multiplies it by 4, and outputs the result. The runner-script checks, that for each execution, the expected result is produced. That script looks like this:


Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell_ise.exe
File Edit View Debug Help
ShouldBeMultipliedByFour.PS1

```
#DocScript Predictability Testing Script
#Ben Mullan 2023

1..1000 | % {

    $ProcrocStartInfo = New-Object System.Diagnostics.ProcessStartInfo
    $ProcrocStartInfo.FileName = "D:\Benedict\Documents\SchoolWork\Projects\DocScript\Resources\Testing\DSCLI.exe"
    $ProcrocStartInfo.RedirectStandardError = $true
    $ProcrocStartInfo.RedirectStandardOutput = $true
    $ProcrocStartInfo.UseShellExecute = $false
    $ProcrocStartInfo.Arguments = "/Run /SourceFile:""D:\Benedict\Documents\SchoolWork\Projects\DocScript\Resources\Test

    $Proc = New-Object System.Diagnostics.Process
    $Proc.StartInfo = $ProcrocStartInfo
    $Proc.Start() | Out-Null
    $Proc.WaitForExit()

    $StdOut = $Proc.StandardOutput.ReadToEnd()
    $StdErr = $Proc.StandardError.ReadToEnd()
    if (($StdOut -replace "[^0-9]" , '') -ne ($_ * 4).ToString()) {
        Write-Host "The DocScript Program produced an unexpected result..."
        Write-Host "$_ * 4 = " + ($_ * 4).ToString()
        Write-Host "StdOut: |$StdOut|"
        Write-Host "StdErr: |$StdErr|"
        Write-Host "ExitCode: " + $Proc.ExitCode
    }

}

Write-Host "Finished all Iterations"
```

```
PS C:\Users\Ben.MNDell> D:\Benedict\Documents\SchoolWork\Projects\DocScript\Resources\Testing\ShouldBeMultipliedByFour.PS1
Finished all Iterations
```

Test: I executed `Powershell.exe ShouldBeMultipliedByFour.PS1`, watching out for any output besides the closure message.

Result: Success: There were no instances of the "unexpected result" message, so all 1000 interpretations of the program worked entirely consistently and predictably! This means that the language can rightly be called *reliable*.

Justification: The use of a script here, meant that I didn't manually have to execute the DS program 1000 times. This saved time, and meant that the testing occurred in an *automated*, more *consistent*, fashion. This was demonstrably a success.

Robustness: DSInteractive Mass-Testing

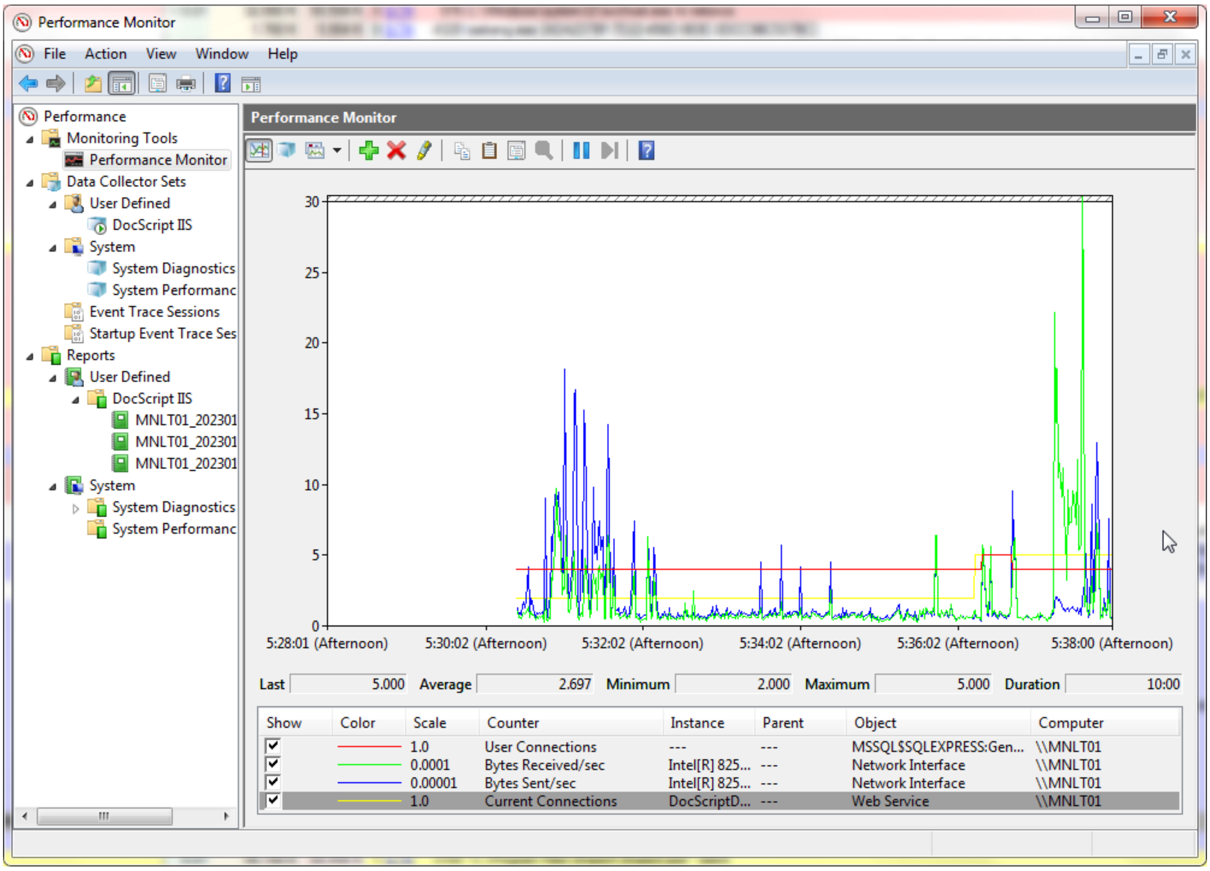
I set up a total of **12 different physical computers** on the same Local-Area Network, so that I could effectively test the performance of the DocScript Interactive system, when it is put under-stress.

Then, to demonstrate that DSInteractive has been designed with **extensibility** and **scalability** in-mind, I used this Rack of Enterprise Servers which I happened to have lying around, to host the DSI Services. I had one physical server dedicated to the DSI Database, which communicated over the network to another physical server, which was hosting the API and Client-Pages from IIS. This could in fact be made even more distributed via SQL Server and IIS Load-Balancing, but using 16 Logical (over 4 Physical) CPUs and 24 GB of RAM, to serve 12 computers, seemed sufficient to me.

← Hosting DSI **(SQL Server and IIS)**

↑ I also configured a local DNS record for the Web server, so that I wouldn't need to memorise the IP Addr. of this server for all 12+ clients. This ended up being "`DocScript.MullNet.NET.`", due to the connection-specific DNS suffix for the Active Directory Domain on which DSI was being hosted.

Justification: This made orchestrating the mass-testing easier and quicker.



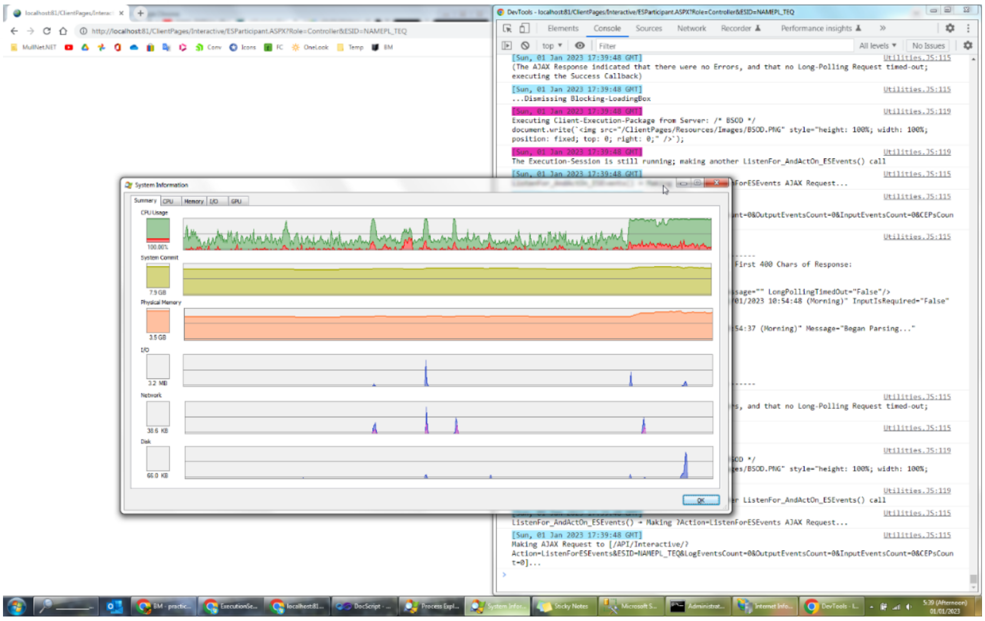
Last 5.000 Average 2.697 Minimum 2.000 Maximum 5.000 Duration 10:00

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
☑	🟥	1.0	User Connections	---	---	MSSQL$SQLEXPRESS:Gen...	\\MNLT01
☑	🟩	0.0001	Bytes Received/sec	Intel[R] 825...	---	Network Interface	\\MNLT01
☑	🟦	0.00001	Bytes Sent/sec	Intel[R] 825...	---	Network Interface	\\MNLT01
☑	🟨	1.0	Current Connections	DocScriptD...	---	Web Service	\\MNLT01

↑ Then, to *monitor the load* on the DSI Servers, I configured a custom PerfMon Tool, to graph the SQL Server and Network activity for me. As can be seen from the above graph, the connections and bytes sent/received do fluctuate a great deal, but this corresponds to the initiation or ending of an Execution-Session, whereat all clients make several AJAX requests and sometimes refresh their pages too.

Test: I ran the Mass-Testing in the described configuration for ~3 Hours, monitoring the graph, and for stability of connections.

Result: Partial-Success: The servers did cope with the load reasonably well, but I did have one problem with a specific CEP (bundle of JavaScript broadcast to DSI Clients during the Execution-Session), which removed all elements of the HTML Document, replacing them with a single image. This meant that the client would instantly attempt to fetch new Execution-Session Events, claiming that it had no current Events in its memory (because these had been deleted from the HTML). Then, the server would respond with **all** the ES Events (rather a large number of data), which the client would fail to add to the now-non-existent ES-Events-Tables. This caused an infinite loop of requests from the client, which instigated **100% CPU usage**:



```
[Sun, 01 Jan 2023 17:39:48 GMT]                                                          Utilities.JS:115
(The AJAX Response indicated that there were no Errors, and that no Long-Polling Request timed-out;
executing the Success Callback)
[Sun, 01 Jan 2023 17:39:48 GMT]                                                          Utilities.JS:115
...Dismissing Blocking-LoadingBox
[Sun, 01 Jan 2023 17:39:48 GMT]                                                          Utilities.JS:119
Executing Client-Execution-Package from Server: /* BSOD */
document.write(`<img src="/ClientPages/Resources/Images/BSOD.PNG" style="height: 100%; width: 100%;
position: fixed; top: 0; right: 0;" />`);
[Sun, 01 Jan 2023 17:39:48 GMT]                                                          Utilities.JS:119
The Execution-Session is still running; making another ListenFor_AndActOn_ESEvents() call
[Sun, 01 Jan 2023 17:39:48 GMT]                                                          Utilities.JS:115
ListenFor_AndActOn_ESEvents() → Making ?Action=ListenForESEvents AJAX Request...
[Sun, 01 Jan 2023 17:39:48 GMT]                                                          Utilities.JS:115
Making AJAX Request to [/API/Interactive/?
Action=ListenForESEvents&ESID=NAMEPL_TEQ&LogEventsCount=0&OutputEventsCount=0&InputEventsCount=0&CEPsCoun
t=0]...
```

The Server was similarly overwhelmed.

Remedial Actions Taken: To fix this, I added in a check before the ES-Events-Request is made by the client, which ensures that the HTML Events tables are still existent. If they aren't, the request isn't made, and the client realises that it is corrupted.

```
/*
    Some (dodgy) CEPs might document.write(), which causes Elements like #LogEventsTable to dissappear.
    This causes the ListenForESEvents request to say that it has a ?CEPsCount of 0, which sends all the
    CEPs for this ExeSes over to the client *again*, which causes the client to execute the same
    document.write CEP again.
    To prevent this, the below ↓ if-statement ensures that the Tables still exist.
*/

if (
    ["#LogEventsTable", "#OutputEventsTable", "#InputEventsTable", "#CEPsTable"].some(
        function (_ElementSelector) { return ($(_ElementSelector).length == 0); }
    )
) {

    /* The Element does not exist on the current [document].
    Not using Swal.fire(), because the resultant MsgBox might not be visible */
    window.alert("This ESParticipant will not listen for any further ES-Events, because at least one of tl
    document.title = "[CORRUPTED ES-Participant]";

} else {
```

The JavaScript `.some()` function is similar to the .NET `.Any()` LINQ method.

This solution fixes the issue!

Usability Features

These are the features designed to make the logical and algorithmic parts of a program, commandable from the point of view of the end user. They include **visual aids** (e.g. syntax highlighting, or button placement) and **control aids** (e.g. the ability to quickly delete an Execution-Session from the ESManager, instead of having to use ssms).

Stakeholder-Testing Meeting

I met with the four primary stakeholders, to show them the Progress I have made with the DocScript products so far.

Together, we reviewed the following **Usability Features**…

In the Language Specification

Operators

Test: "*Have the DocScript Operators been implemented in such a way as to meet the ease-of-use and standardisation requirements which you – the Stakeholders - stipulated?*"

Result & Comments: Positive: There is no operator overloading (thereby simplifying the learning experience), and all the operator characters can be found on a standard UK-ISO Keyboard, such as…

…meaning that there are no ALT-Codes required for any of the symbols, including `¬|&^%*/-+'~`

Keywords

Test: "*Are the DocScript Keywords clear, self-explanatory, and memorable?*"

Result & Comments: Positive: The Statement-Closing notation of *End{StatementType}* e.g. `EndFunction` means that effectively only *half* the number of keywords need to be remembered, since the statements all follow this pattern. In addition, the grand total of 6 Keywords, means that there isn't a litany of complicated abstract jargon to memorise, for users of this simple scripting language.

In the Implementations

Windows IDE

I discussed the following Usability Features of the DocScript© Windows™ IDE with the Stakeholders:

- **Coloured Icons** - **Result & Comments: Positive:** "*These greatly improve the ease and speed with which the software can be navigated visually; the eyes lock onto the colours in their locations, to form a frame-of-vision, which enables a rapid memorisation of where different features are.*"

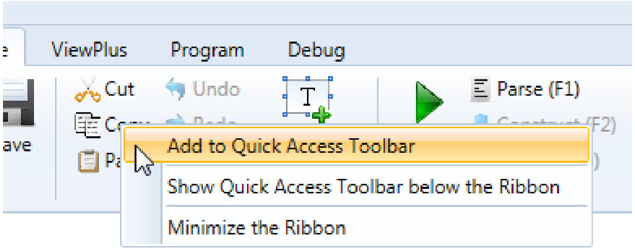
- **Zoom, Full-Screen, and ViewPlus Features** - **Result & Comments: Partially-Positive:** "*It is very useful to be able to zoom-in and -out, particularly for presentations and teaching and the like, although I'm not sure if the Skew and Rotate options were really necessary. Nevertheless, then don't impede the functionality of the product, just by being there!*"

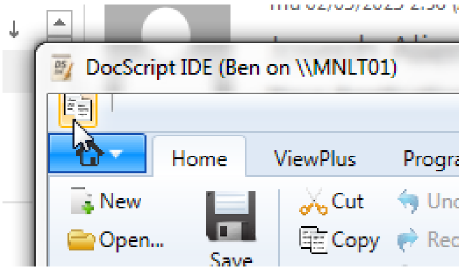
- **Syntax-Highlighting** - **Result & Comments: Positive:** "*This is a veritably useful feature; in a similar way to the coloured icons, it creates a frame in the mind's eye, which calibrates the glances around the source code and IDE. This makes decomposing the program visually, significantly easier.*"

- **The All-In-One Run (F5) Button** - **Result & Comments: Positive:** "*This is much easier than individually pressing F1, F2, and then F3.*"

- **Keyboard Shortcuts** - **Result & Comments: Positive:** "*Great! Once you learn them, it makes usage of the IDE faster than using the mouse for everything.*"

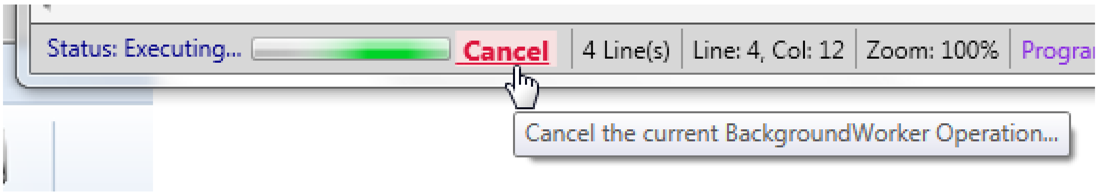
- **The QuickAccessToolbar** - **Result & Comments: Negative:** "*The Microsoft Ribbon SDK provides the ability to pin buttons to the QuickAccessToolbar, but in the DocScript IDE, this hasn't been enabled – why not?*"
  Seemingly, I had to add a XAML Attribute to the RibbonButtons, to permit them to be added to the QuickAccessToolbar. I quickly did this for some of the most frequently-used commands, so that they can now be pinned… ↓ Pinned "Copy" Button


ViewPlus
Program
Debug
Cut
Undo
Parse (F1)
ave
Add to Quick Access Toolbar
Show Quick Access Toolbar below the Ribbon
Minimize the Ribbon

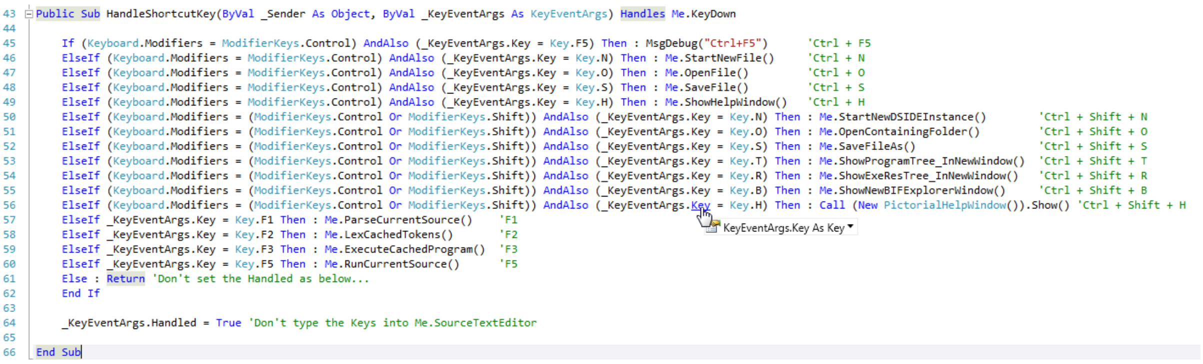


- **The BackgroundWorker for Interpretation Operators** - **Result & Comments: Partially-Positive:** "*It's good that the Interpretation takes place in a separate thread to that of the UI, however we – the Stakeholders – have noticed some odd behaviour with the [**Cancel**] Button not really working for certain operations. For instance, if the DocScript program starts playing some audio, and the Cancel button is pressed, then the audio continues to play, even after the ostensible cancellation.*

This isn't really a huge problem though, especially in the vast majority of simpler, I/O-Based Programs."



Parenthetically, the number of available Keyboard Shortcuts in DSIDE has been expanded since those mentioned in *§Design*; the `.KeyDown` Event Handler now looks like this:



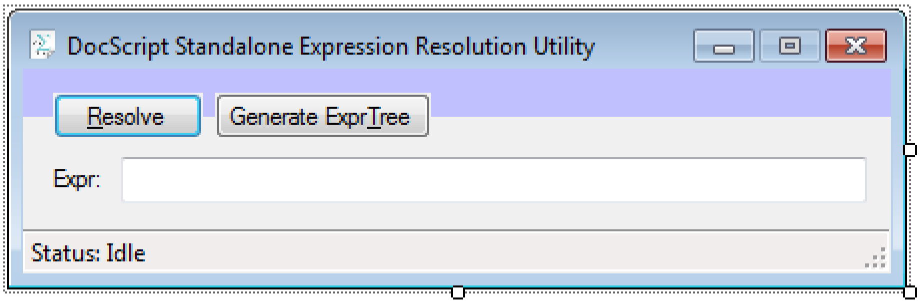
```
Public Sub HandleShortcutKey(ByVal _Sender As Object, ByVal _KeyEventArgs As KeyEventArgs) Handles Me.KeyDown

    If (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.F5) Then : MsgDebug("Ctrl+F5")      'Ctrl + F5
    ElseIf (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.N) Then : Me.StartNewFile()     'Ctrl + N
    ElseIf (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.O) Then : Me.OpenFile()         'Ctrl + O
    ElseIf (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.S) Then : Me.SaveFile()         'Ctrl + S
    ElseIf (Keyboard.Modifiers = ModifierKeys.Control) AndAlso (_KeyEventArgs.Key = Key.H) Then : Me.ShowHelpWindow()   'Ctrl + H
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.N) Then : Me.StartNewDSIDEInstance()       'Ctrl + Shift + N
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.O) Then : Me.OpenContainingFolder()       'Ctrl + Shift + O
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.S) Then : Me.SaveFileAs()                 'Ctrl + Shift + S
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.T) Then : Me.ShowProgramTree_InNewWindow() 'Ctrl + Shift + T
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.R) Then : Me.ShowExeResTree_InNewWindow()  'Ctrl + Shift + R
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.B) Then : Me.ShowNewBIFExplorerWindow()    'Ctrl + Shift + B
    ElseIf (Keyboard.Modifiers = (ModifierKeys.Control Or ModifierKeys.Shift)) AndAlso (_KeyEventArgs.Key = Key.H) Then : Call (New PictorialHelpWindow()).Show() 'Ctrl + Shift + H
    ElseIf _KeyEventArgs.Key = Key.F1 Then : Me.ParseCurrentSource()    'F1
    ElseIf _KeyEventArgs.Key = Key.F2 Then : Me.LexCachedTokens()       'F2
    ElseIf _KeyEventArgs.Key = Key.F3 Then : Me.ExecuteCachedProgram()  'F3
    ElseIf _KeyEventArgs.Key = Key.F5 Then : Me.RunCurrentSource()      'F5
    Else : Return 'Don't set the Handled as below...
    End If

    _KeyEventArgs.Handled = True 'Don't type the Keys into Me.SourceTextEditor

End Sub
```

Other Implementations

Besides the IDE – which necessitates and claims a preponderance of the usability features – are the other implementations...

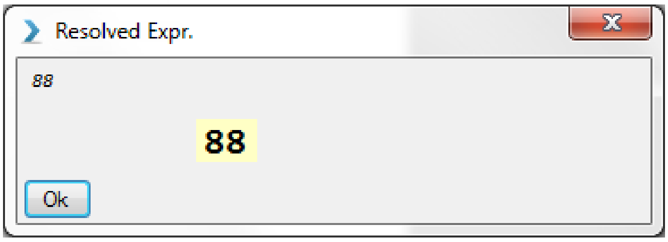
AcceptButtons

I have ensured that all the main Windows and Dialogs have their `AcceptButton` Properties set. This means that there is a *default* Button on the form, which is automatically clicked when the [Enter] key is pressed. This enables faster navigation of the various DocScript windows, because – like the Keyboard shortcuts – it minimises the level of mouse-interaction required.



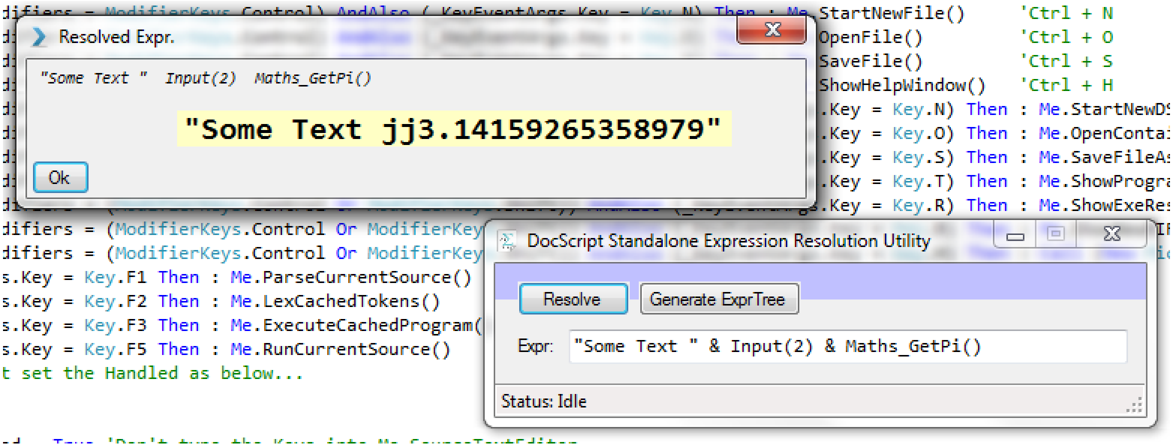
Test: Get the Stakeholders to attest to the benefit of having the `AcceptButton`s – Are they actually useful, of does the additional blue border highlighting make the User Interface more confusing?

Result & Comments: Positive: I sent a copy of `DSExpr.exe` to the Primary Stakeholder "*Kiran*", who commented that this is a useful feature, which expedites navigation and usage of the package; he could, for instance, easily close the ResultWindow dialogs by pressing enter, after having requested the evaluation of an expression:



```
88

88
```

Standalone Expression Evaluation



```
"Some Text "  Input(2)  Maths_GetPi()

"Some Text jj3.14159265358979"
```

Expr: `"Some Text " & Input(2) & Maths_GetPi()`

Status: Idle

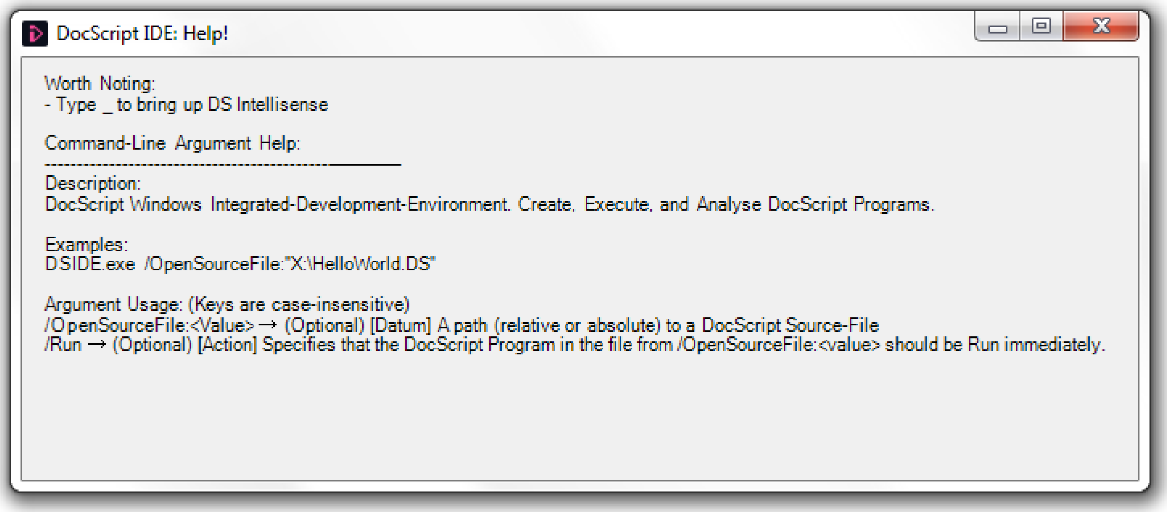
```
StartNewFile()     'Ctrl + N
OpenFile()         'Ctrl + O
SaveFile()         'Ctrl + S
ShowHelpWindow()   'Ctrl + H
.Key = Key.N) Then : Me.StartNewDS
.Key = Key.O) Then : Me.OpenConta
.Key = Key.S) Then : Me.SaveFileAs
.Key = Key.T) Then : Me.ShowProgra
.Key = Key.R) Then : Me.ShowExeRes
difiers = (ModifierKeys.Control Or ModifierKey
difiers = (ModifierKeys.Control Or ModifierKey
s.Key = Key.F1 Then : Me.ParseCurrentSource()
s.Key = Key.F2 Then : Me.LexCachedTokens()
s.Key = Key.F3 Then : Me.ExecuteCachedProgram(
s.Key = Key.F5 Then : Me.RunCurrentSource()
t set the Handled as below...
```

Test: Get the Stakeholders to solve some mathematical and Boolean-logic problems, using the DSEXPR.EXE program. Is it easy-to-use?

Result & Comments: **Partially-Positive:** "*The product is largely well-thought-out, and is useful for quick expression analysis. It complements the* `DSCLI's /Live` *mode quite well in fact. It would be nice, however, if the [Resolved Expr.] Window would automatically resize to whatever the length of the output string is. This is not a problem with the* `/Live` *mode, when using the* `?{Expr}` *feature.*"

Help Window

The IDE does include a Help Window, which – at the moment – looks like this:



Worth Noting:
- Type _ to bring up DS Intellisense

Command-Line Argument Help:

Description:
DocScript Windows Integrated-Development-Environment. Create, Execute, and Analyse DocScript Programs.

Examples:
DSIDE.exe /OpenSourceFile:"X:\HelloWorld.DS"

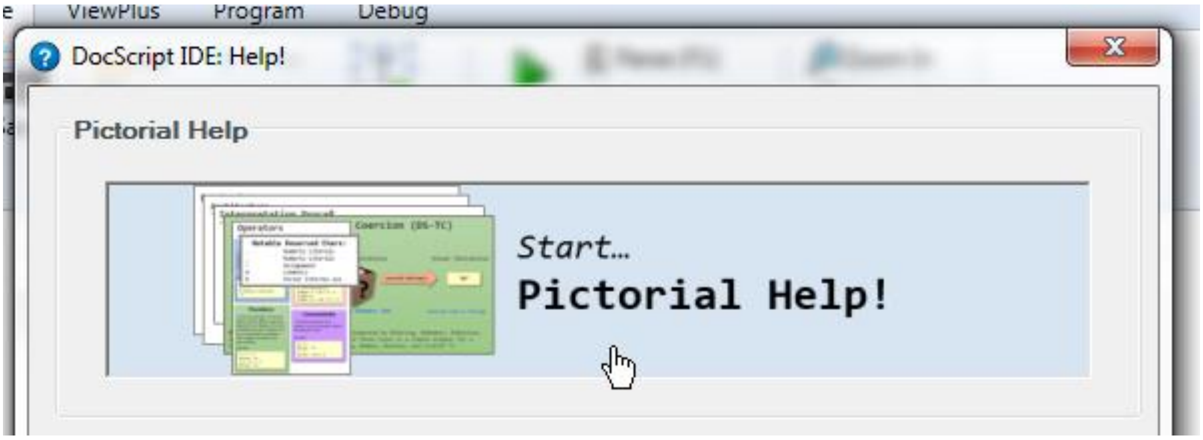
Argument Usage: (Keys are case-insensitive)
/OpenSourceFile:<Value> → (Optional) [Datum] A path (relative or absolute) to a DocScript Source-File
/Run → (Optional) [Action] Specifies that the DocScript Program in the file from /OpenSourceFile:<value> should be Run immediately.

Test: Get the Stakeholders to use the Help Window (Ctrl + H) – Is the information thereon useful?

Result & Comments: Partially-Positive: "This text-based help is of some use, and is *better than nothing*, but isn't particularly extensive, dosen't introduce concepts about the DS Language, and isn't visual."

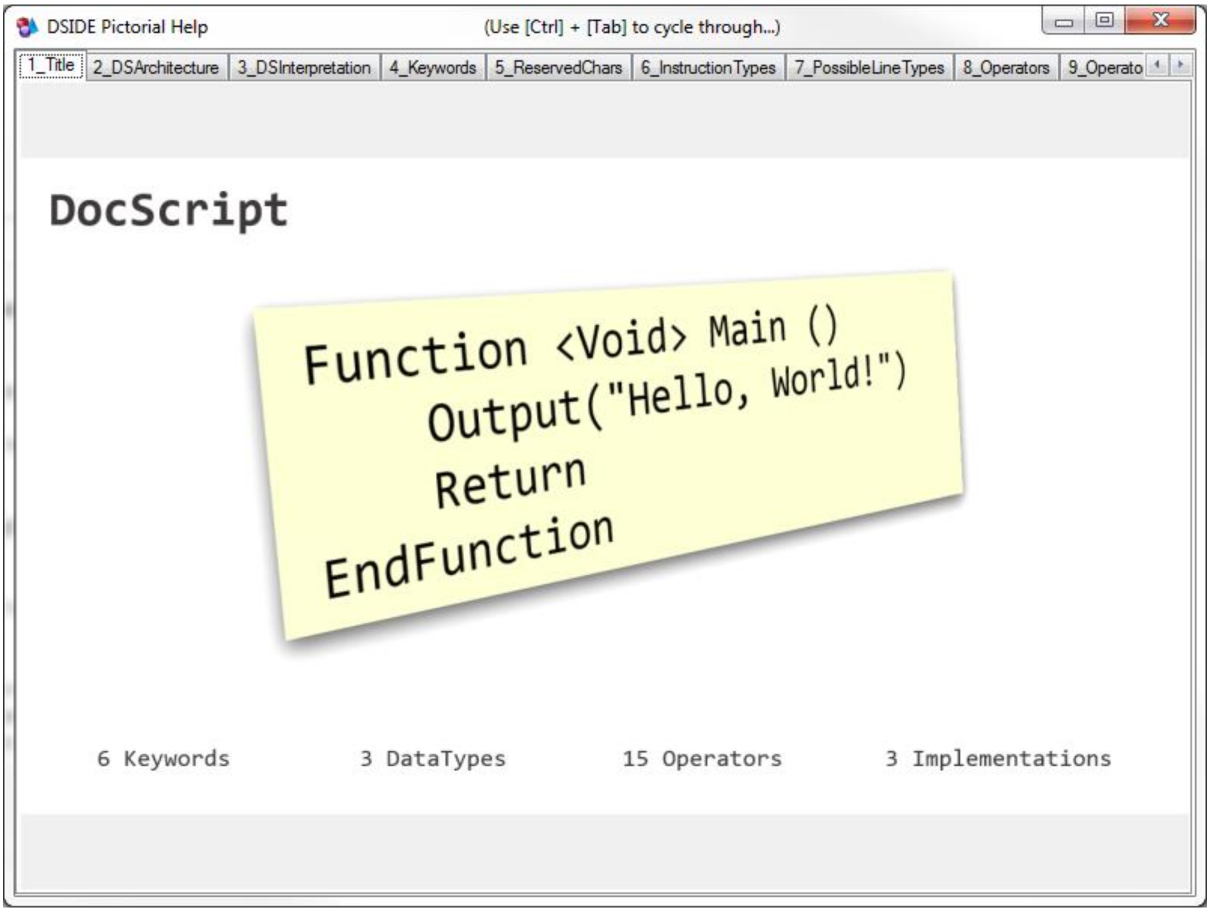
Remedial Actions Taken: I have now written a "Pictorial Help" system for DSIDE, using many of the same DS Architecture and system-mechanics diagrams from this document, and the DS-Specification-PowerPoint.

The pictorial help can be launched using the dedicated button…



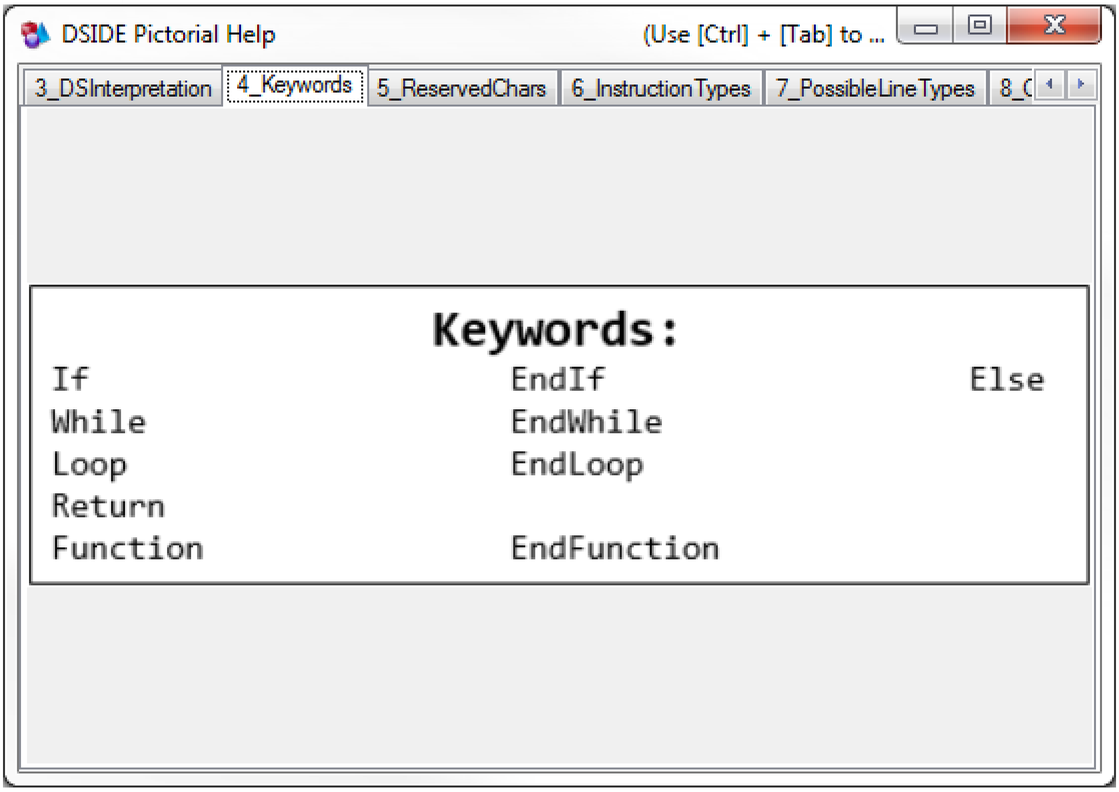
…or the KeyBoard ShortCut Ctrl + Shift + H.

It looks like this:



```
Function <Void> Main ()
    Output("Hello, World!")
    Return
EndFunction
```

[Above + Below] DSIDE's Pictorial-Help Feature



Keywords:

If	EndIf	Else
While	EndWhile	
Loop	EndLoop	
Return		
Function	EndFunction	

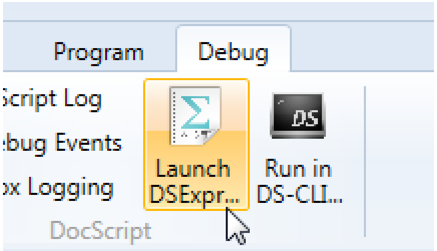
How the Implementations work together

I also need to consider how well the 3 (or four including DSExpr.exe) DocScript implementations work together.

Firstly, it would not make sense for the Command-line Implementation, DSCLI.exe, to call on features of the Windows IDE, DSIDE.exe. This is because using the command-line interpreter can sometimes occur in non-desktop environments – e.g. via a TELNET or PSEXEC session. Such environments would fail to invoke features of a graphical Windows application, and even if they were invocable, the end user (sitting on the other end or an 80-column ASCII text terminal) would not see any of the graphics reproduced.

Secondly, because the web-based client, DSInteractive, is designed to run on different operating systems, it cannot be relied upon for a particular subsystem to be present on the client. For example, were all clients known to be using Internet Explorer 9, with Silverlight enabled, then it may be worthwhile implementing a [Run in DSIDE] button, for easy transfer of execution to the Desktop Interpreter. However, because browser plug-ins are largely viewed pejoratively these days, no such consistency can be expected, and therefore no such feature can be implemented.

What does make sense – however – is to provide integration *from* the Windows IDE, *to* the other DocScript products. If a user is running DSIDE (from a desktop WINSTA Session), then it can reasonably be expected that other processes (such as



the DSCLI command-window, or a web-browser instance) can be launched, and that the user will see them. Therefore, the buttons in the adjacent image, exist within the DocScript® Windows™ IDE© to facilitate such integration.

Future Additions: Stakeholder-Suggested Usability Improvements

The Stakeholders did have some ideas for potential future improvements, aside from those mentioned during my questioning.

This is how any unmet criteria might be addressed, in the future…

- **[Esc] to close Dialogs** - "*It's slightly inconvenient that the Dialogs in the DocScript software products can't be closed by the escape key. This is a commonplace GUI feature, and it would be nice to have it here too.*" **Future Action to Take:** I will add an EventHandler to the `.KeyDown` Event of the Forms, which calls `.Close()` if the `.KeyCode` is equal to `Keys.Escape`.

- **An Installation Wizard** - "*It's good that the products don't require installation, however it would be useful if there were an InstallShield-style wizard to more **permanently** install the DocScript Software. This would be particularly useful for deploying DocScript across an AD-Domain using Group Policy, if the installation package were an* `.MSI` *file.*" **Future Action to Take:** I will create an MSI installer for the DocScript Desktop Components (everything except from DSInteractive, which requires an SQL Server Instance). Visual Studio supports the creation of MSI packages natively, and they include both a Graphical and a Silent means of installation.

- **An Animated SplashScreen** - "*The current SplashScreen for DSIDE is adequate, but it might be nice to be able to tell precisely what the Process is actually loading.*" **Future Action to Take:** I ought to be able to add a loading-bar or text-label to the SplashScreen, making it a Window, instead of just a PNG as it is at the moment. For example, the Photoshop CS5 SplashScreen has a loading label of the sort I mean.

- **Use of a VCS or Source Control System** - "*As the development of DocScript continues, you may wish to collaborate with other developers. This would necessitate a Version-Control-System, which can track changes and handle merging or altered lines in different parts of a file.*" **Future Action to Take:** Although it would be best to use VS-TFS for this, I must regrettably concede that – these days – the only realistic way to host the DocScript source-code to be freely-accessible, centralised, and have documentation in one place, does seem to be **GitHub**. There is a version of the Git Source Control System for VS 2010, which I have started to experiment with, and would have to admit that it's actually rather nice. This system *will* therefore be part of DocScript's future, for pragmatic reasons.

Microsoft® Visual Studio™ Team Foundation Server 2010 *Vs.* git

Future Additions: Personal Ideas

In Addition, I do have some personal thoughts about exciting features I'd like to integrate into DocScript in the future…

- **DocScript Remoting!** – The ability to remotely execute DocScript programs on a target computer of the user's choice. I think I'd build this in as a feature of DSIDE, so that if DSIDE.exe were to be running on two computers on the same Network, then they could use TCP Networking to communicate over a certain port, and one DSIDE instance would receive a program-to-execute from the other, which would be interpreted ad-hoc. **What would be really neat**, is if there didn't even need to be a DSIDE client running on the target computer though. I could use PsExec to connect to a machine (with supplied credentials), select a Logon Session, and instantly run a DocScript program thereon. (Rather a lot of fun could be had with this feature, I feel)

- **DocScript Compilation!** – At the moment, this in an Interpreted Programming Language. However: I have been thinking about how I could extend it to permit compilation of a DocScript program to a self-contained .exe file. I would do this by translating each line of the DocScript program (one of the 8 possible IInstruction Types) into a corresponding Visual Basic .NET line. The two languages are – by no coincidence – rather similar, so this translation would be quite doable. Then, I would save this generated Visual Basic source to a .VB file on disk, and automatically run `vbc.exe` (the Visual Basic .NET Compiler) to generate an .exe file. There would be a Compilation-Options dialog before this to allow the user to choose e.g. an Icon to use for the output .exe. To get the Built-In-Functions to work, I would have to include a copy of `DocScript.Library.DLL` next to the output .exe. However, I could solve this problem by automatically running ILMerge.exe, on the .exe and .dll.

- **DocScript Graphics!** – The ability to use DocScript to create basic GUI applications. At a basic level, I could take a Microsoft-SmallBasic-like approach, and have some Built-in Functions for managing a single GraphicsWindow, e.g. `Graphics_DrawText("Hello", 50, 50, "#0A0A0A")` and `Graphics_DrawImage("X:\DS\Client.PNG")`. Alternatively, I could even use some form of markup language to generate user interfaces from (e.g. a WinForms `.Designer.VB`, `.HTML`, `.XAML`, or `.DSGUI` file).

*It must be said, however, that there comes a point at which these advanced features can't really be used to their full potential; DocScript has been designed from-the-ground-up to be a **SIMPLE** and **UNPRETENTIOUS** system, and bolting-on advanced functionality is ultimately limited by the simplistic nature of the programming language itself.*

Testing Tables & Success Criteria

There were 3 tables from the [Analysis] stage, and 1 from the [Design] stage, which I need to officially sign-off. **Reminder:** The Analysis-stage tables have already been reviewed at the end of §Development.

Testing Checklist (Interpreter DLL)

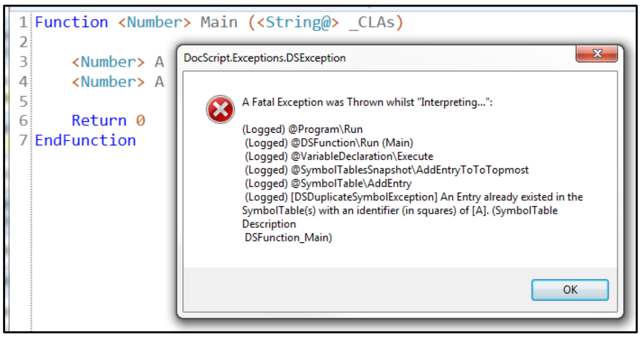
Now that the vast majority of the development is complete, *myself* and the *Stakeholders* shall perform each of the following tests, to ensure that the requirements have been met. This testing table was delineated back in §Design.

Assert-style Test	Passed?
An error is thrown if two variables are declared within the same (or relative downstream) scope, when both variables have the same identifier, or the identifiers differ only by case; the language is **not** case-sensitive	Yes
Variables can be declared in each of the 6 valid DataTypes (The 3 DataTypes, and their Array variants)	Yes
A high degree of verbosity is present in the error message resultant of an attempt to lexically analyse the expression (e.g.) `5 + + 4`	Yes
A Function can be declared with IInstruction-based contents statements inside, such as an IfStatement `If (Expr) {LineEnd} … EndIf`	Yes
A Global variable can be declared outside of any Functions (E.g. `<String> NameGlobal = "Ben Mullan; S7; Y13; U6;"`)	Yes
An error is thrown if a Program is written without an EntryPoint Function `Main`	Yes
A DocScript Program can be written to output "B" if Command-Line Argument [0] is "1", and "C" if it is "2" (Just an example, to prove that CLAs can affect programme output)	Yes
A DocScript program can be written to `Output()` "Hello, World!"	Yes
A DocScript program can be written to take `Input()` from the user	Yes
A Comment can be specified with `# Comment {LineEnd}`	Yes
The Expr `[5 + 3] * 9` resolves to 72 whereas `5 + [3 * 9]` resolves to 32 (proof that brackets work)	Yes
The numeric literals `10`, `10.0`, and `10_10` are all magnitudionally equivalent	Yes
All of the Test Data run successfully in the DocScript System & Implementations	Yes

Each of these tests was performed ***by the stakeholders***, on *their personal computers*. It is my hope, that the copious screenshots and other media evidence provided hereinbefore constitute a sufficient volume of evidence for these tests…

Extended Testing Evidence

…However, to pander deferentially to the mark scheme at whose mercy this document lies, I am providing this supplemental evidence of the products having passed these tests:

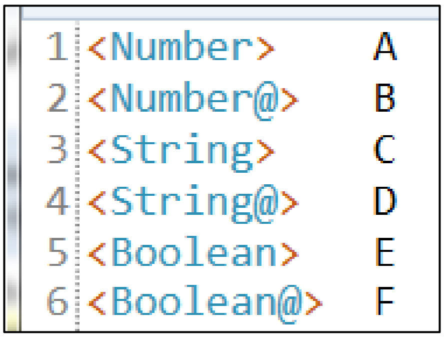


```
Function <Number> Main (<String@> _CLAs)

    <Number> A
    <Number> A

    Return 0
EndFunction
```

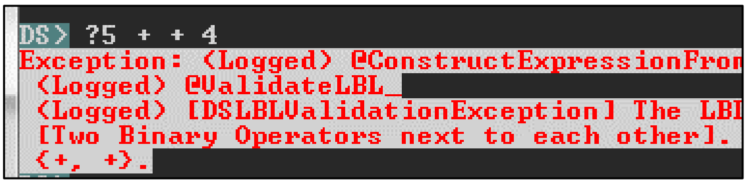
DocScript.Exceptions.DSException

A Fatal Exception was Thrown whilst "Interpreting...":

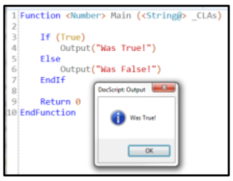
```
(Logged) @Program\Run
 (Logged) @DSFunction\Run (Main)
 (Logged) @VariableDeclaration\Execute
 (Logged) @SymbolTablesSnapshot\AddEntryToToTopmost
 (Logged) @SymbolTable\AddEntry
 (Logged) [DSDuplicateSymbolException] An Entry already existed in the
SymbolTable(s) with an identifier (in squares) of [A]. (SymbolTable
Description
 DSFunction_Main)
```



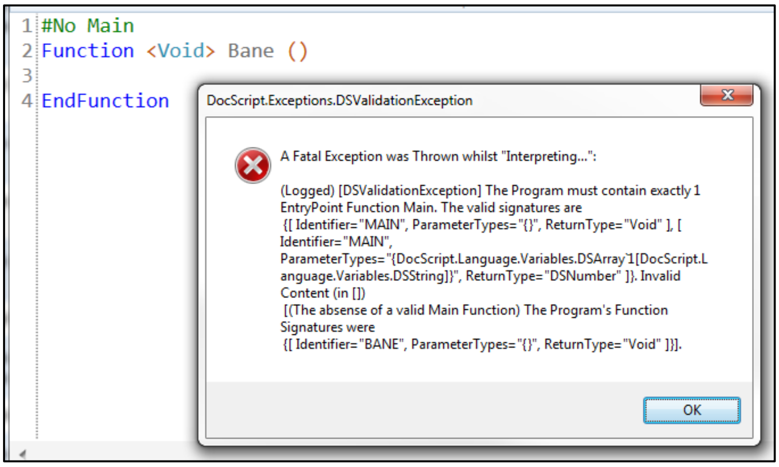
```
<Number>     A
<Number@>    B
<String>     C
<String@>    D
<Boolean>    E
<Boolean@>   F
```



```
DS> ?5 + + 4
Exception: <Logged> @ConstructExpressionFrom
 <Logged> @ValidateLBL_
 <Logged> [DSLBLValidationException] The LBL
 [Two Binary Operators next to each other].
 {+, +}.
```



```
Function <Number> Main (<String@> _CLAs)

    If (True)
        Output("Was True!")
    Else
        Output("Was False!")
    EndIf

    Return 0
EndFunction
```

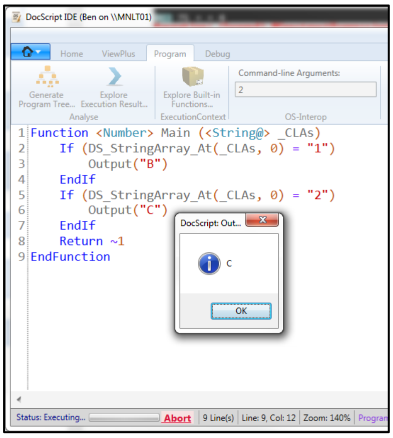


```
#No Main
Function <Void> Bane ()

EndFunction
```

DocScript.Exceptions.DSValidationException

A Fatal Exception was Thrown whilst "Interpreting...":

```
(Logged) [DSValidationException] The Program must contain exactly 1
EntryPoint Function Main. The valid signatures are
 {[ Identifier="MAIN", ParameterTypes="{}", ReturnType="Void" ], [
Identifier="MAIN",
ParameterTypes="{DocScript.Language.Variables.DSArray`1[DocScript.L
anguage.Variables.DSString]}", ReturnType="DSNumber" ]}. Invalid
Content (in [])
 [(The absense of a valid Main Function) The Program's Function
Signatures were
 {[ Identifier="BANE", ParameterTypes="{}", ReturnType="Void" ]}].
```

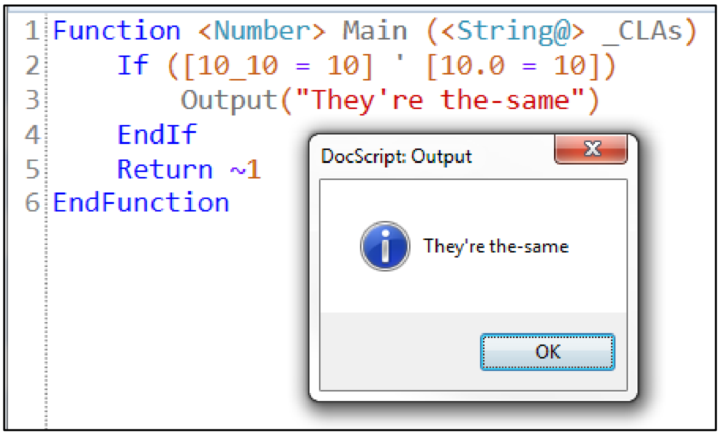


DocScript IDE (Ben on \\MNLT01)
Home
ViewPlus
Program
Debug
Generate Program Tree...
Explore Execution Result...
Analyse
Explore Built-in Functions...
ExecutionContext
OS-Interop

Command-line Arguments: 2

```
Function <Number> Main (<String@> _CLAs)
    If (DS_StringArray_At(_CLAs, 0) = "1")
        Output("B")
    EndIf
    If (DS_StringArray_At(_CLAs, 0) = "2")
        Output("C")
    EndIf
    Return ~1
EndFunction
```

Status: Executing... | 9 Line(s) | Line: 9, Col: 12 | Zoom: 140% | Program

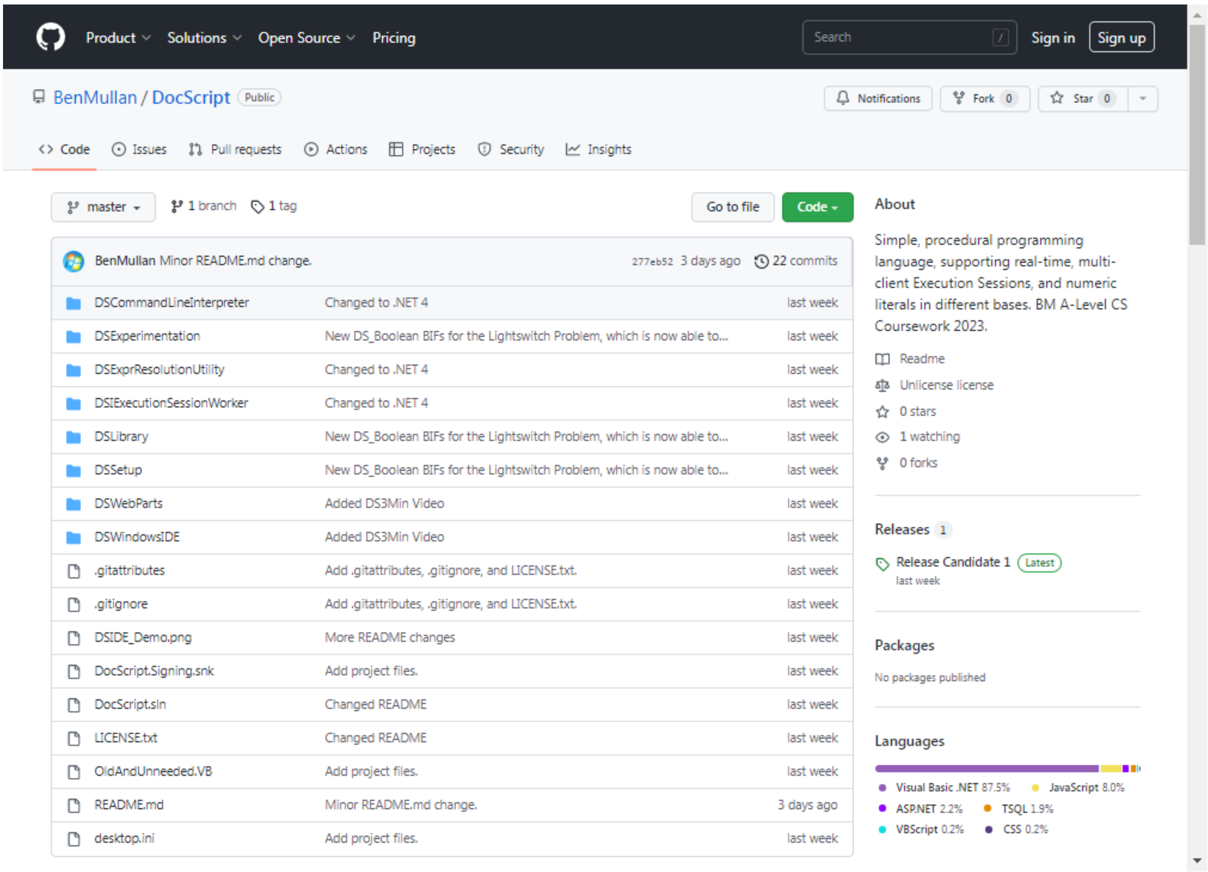


```
Function <Number> Main (<String@> _CLAs)
    If ([10_10 = 10] ' [10.0 = 10])
        Output("They're the-same")
    EndIf
    Return ~1
EndFunction
```

Future Maintenance

Further to the comments I made about the use of Source Control, the aspects of DocScript's Future, are as follows:

- **Accessible** – There is now a **GitHub** "Repository" for DocScript, which means that anybody can *PULL* down a copy of the entire solution, and see how DocScript works, or make changes and improvements. I do find many of the features of Git to be sloppy and hideously-inconsistent (such as the complete lack of uniformity in the filenames `README.md`, `.gitignore`, `LISCENSE.txt`, and `.gitattributes`), but there nevertheless don't seem to be many alternatives to "*GitHub*", so DocScript seems to be along for the ride, as it were. https://github.com/BenMullan/DocScript.git



BenMullan Minor README.md change. 277eb52 3 days ago 22 commits

DSCommandLineInterpreter	Changed to .NET 4	last week
DSExperimentation	New DS_Boolean BIFs for the Lightswitch Problem, which is now able to...	last week
DSExprResolutionUtility	Changed to .NET 4	last week
DSIExecutionSessionWorker	Changed to .NET 4	last week
DSLibrary	New DS_Boolean BIFs for the Lightswitch Problem, which is now able to...	last week
DSSetup	New DS_Boolean BIFs for the Lightswitch Problem, which is now able to...	last week
DSWebParts	Added DS3Min Video	last week
DSWindowsIDE	Added DS3Min Video	last week
.gitattributes	Add .gitattributes, .gitignore, and LICENSE.txt.	last week
.gitignore	Add .gitattributes, .gitignore, and LICENSE.txt.	last week
DSIDE_Demo.png	More README changes	last week
DocScript.Signing.snk	Add project files.	last week
DocScript.sln	Changed README	last week
LICENSE.txt	Changed README	last week
OldAndUnneeded.VB	Add project files.	last week
README.md	Minor README.md change.	3 days ago
desktop.ini	Add project files.	last week

About

Simple, procedural programming language, supporting real-time, multi-client Execution Sessions, and numeric literals in different bases. BM A-Level CS Coursework 2023.

Readme
Unlicense license
0 stars
1 watching
0 forks

Releases 1

Release Candidate 1 Latest
last week

Packages

No packages published

Languages

● Visual Basic .NET 87.5% ● JavaScript 8.0%
● ASP.NET 2.2% ● TSQL 1.9%
● VBScript 0.2% ● CSS 0.2%

- **Flexible** – The fact that the solution is extremely **modular and extensible**, means that sub-components of DocScript can easily be implemented into other products. For instance, if an application such as an image-editing package, required basic scripting capabilities, then DocScript could straightforwardly be embedded – using the Core Interpreter DLL – along with a few custom BuiltInFunctions, to permit automation of certain image-manipulation tasks. Many products use scripting languages such as Lua to achieve a similar goal.

- **Approachable** – The extensive **comments, inline-annotations, and specification diagrams** provided with the solution, mean that navigation of the project ought to be at-least tractable, if not almost enjoyable. The self-evident variable names and identifiers have a similar effect.

Limitations – Or were they?

Here, I revisit the limitations which – several months ago in §Analysis – I anticipated encountering throughout the development. Were they genuine concerns?

- **The Breath (complexity) of the Solution**: "How many built-in functions will be available? Will there be additional encapsulative features such as namespaces? How many complex operators can be used for the mathematical expressions?"
  Outcome: Once I got started, I rather quickly picked-up pace with the development (the solution is now at 82,634 lines), and was therefore able to implement more features than I had *thought* I was going to be able to. There is a multiplicity of Built-in Functions, and it's easy to add more. There aren't any namespaces yet, but that wasn't really required, because the BIFs use the `{Category}_{Name}` nomenclature, e.g. `System_Run()`. There are all the standard mathematical operators, and then some; many languages don't have exponentiation `^` or modulo `%` operators.
  The breadth therefore never became an effectuated limitation.

- **Excessive Complexity**: "…any worthwhile programming language must be sufficiently complex as to enable the programmer to develop at a reasonable pace, once familiar with the system. At the same time, *this* language has the paramount requirement that it be simple to use and learn."
  Outcome: Having seen the language being learnt for the first time by several stakeholders, I would argue that it is more approachable than most other languages, but acknowledge that there are nevertheless still a number of slightly more unconventional elements – such as the explicit separation of Parsing, Lexing, and Execution – which many newcomers won't instantly understand.
  Excessive Complexity therefore became a small but unavoidable problem.

- **Mandatory Use of a Keyboard/Mouse for Interaction**: "Having to interact with the programming language in a conventional, predominantly keyboard-based fashion (which was agreed upon by the stakeholders to be suitable) does of course mean that people with certain disabilities, which make it difficult for them to type, may be unable to make full use of the software."
  Outcome: I have not come across any stakeholders (so-far) who have been unable to use a Keyboard or Mouse to interact with the software. Even if such stakeholders were to make themselves known, I'm afraid that the problem isn't in the scope of this project.
  Keyboard usage therefore isn't a relevant limitation.

- **Difficulty of Development**: "Since I have not implemented any complex expression parsing in my programming before, this is something I shall have to learn much more about."
  Outcome: I initially researched Parsers and Abstract-Syntax-Trees from an old book from the 1980's (*COMPUTER SCEINCE, 4th Edition, C. S. French*), but progressed onto a Masters' Course on Compilers, from the university of Washington. After tens of A3 sheets of planning, and thinking about how to create such a complex system, I eventually came up with – what I haughtily considered to be – a watertight structure.

(I'm still somewhat surprised that I haven't discovered some gaping hole or structural problem somewhere in the logic (It may just be a matter of time…))
Hence, the Difficulty wasn't an insurmountable limitation.

- **Security**: "…since it is a system of such complexity, and because there are a very large number of edge cases and unaccounted-for pieces of input, certain scripts may potentially cause insecure behaviour such as buffer overflows or injection."
  Outcome: I have not come across any security flaws so far, and I do claim that if properly configured, there is nothing insecure about any of the DocScript products. However: The component most susceptible to security issues would be DocScript-Interactive (the web-based implementation). Misconfigurations of the system user accounts under which the Web Server and SQL Server run, could in-theory expose private areas of a computer.
  Although the Security hasn't been an issue hitherto, it's possible that it could lead to problems in the future. Therefore, to overcome this limitation, I could produce a more in-depth setup guide for DSInteractive, to reduce the likelihood of misconfigurations.

Conclusion

I led a final check-in with the Stakeholders, at the end of what has been a **12-month** project…

Stakeholder Comments

The Stakeholders made the following concluding comments:

- It's a very **convenient and portable** set of powerful and well-thought-out programs. In particular, the DSCLI /Live mode is very useful for expression evaluation.
- The products are effective as approachable pedagogical **teaching tools**. Specifically, DSInteractive is great for multi-client algorithmic demonstrations.
- It certainly would be exciting to see the **DocScript-Compilation**, **-Graphics**, and **-Remoting** concepts come to floriation.

Where's *All* the Code?

It's here, or at http://BenM.eu5.org/ under "**DocScript**".
The more decadent amongst you, may even wish to use https://github.com/BenMullan/DocScript/.

Appendix

Abbreviations

IR	Intermediate Representation (e.g. an AST or Instruction Tree)
LBL	Linear Bracketed Level (ExprTree Construction)
IOT	Intermediate Operator Tree (ExprTree Construction)
SCI	Scanned Component Indicator (ExprTree Construction)
ESID	ExecutionSession Identifier (DocScript Interactive)
TPV	Tokens to (Token)Patterns Validator (Lexing)
CEP	Client Execution Package (DocScript Interactive)
BIF	Built-In Function

Notation

No.	Number Of
UpToInc *	Up-to and Including *
UpToExc *	Up-to but Excluding *
<*>	Of the DataType * (DocScript Source)
_*	* is an Identifier for a Local Item
*_	* is an Identifier for a Private or Protected Item
*__	* is an identifier for a Friend Item
_*_	* is an Identifier for a Static (not *Shared*) Variable
T*	* is a Generic Type Specifier
I*	* is an Interface
Ensures *	Throws an Exception if * is not the case
DS*Exception	* is a DocScript DataType inheriting from System.Exception
[*]	WebParts: * will be returned by the API, as an XML Attribute
[...*]	WebParts: * will be returned by the API, when Long-Polling ends
[<*>]	WebParts: * will be returned as an XML-Child of <ResponseContent>

References

- https://courses.cs.washington.edu/courses/csep501/14sp/video/archive/html5/video.html?id=csep501_14sp_1
- https://greg4cr.github.io/courses/spring16csce747/Lectures/Spring16-Lecture23PostRelease.pdf
- https://www.codeproject.com/Articles/12335/Using-SqlDependency-for-data-change-events