Instituto Superior de Engenharia de Lisboa

Electronics, Telecommunications and Computers Engineering Department

BrightScript Debugger

Celso de Almeida Fernandes

(Graduated in Computer Science and Computer Engineering)

Dissertation for Master's Degree in Computer Science and Computer Engineering

Advisors:	Eng. Paulo Pereira
	Eng. Pedro Pereira
	Jury:
President:	Eng. Manuel Barata

novembro de 2016

Abstract

BrigthScript is a programming language based on JavaScript and Visual Basic, created by Roku. Roku is a company who produces and commercializes devices that are used to watch movies and television. These devices are usually connected to a TV and are commonly known as TV boxes. BrightScript is the language on which applications for their boxes are developed.

After some analysis, we could not found many development tools and the existing ones are not functional enough. Roku provides an Eclipse plugin and the boxes expose a telnet port for basic debugging. The Eclipse plugin only makes syntax validation and exports application code to the box. There are several open source plugins for the most popular text editors, that perform syntax highlighting.

The goal of this project is therefore to implement an integrated tool for application development thereby simplifying the development process, that includes debugging. This tool supports syntax validation, code compilation, intellisense and graphical debug interaction.

Visual Studio is an Integrated Development Environment created by Microsoft and it is the main tool for the development of Windows applications. The tool is materialized as a Visual Studio plugin for the BrightScript language. This plugin uses the language services provided by Visual Studio SDK.

The tool could be enhanced with a box simulator that would be used to run the applications on the development machine, thereby further improving the development cycle; a goal identified as future work.

The idea for this project was a consequence of the author's involvement on SkyStore Roku App development, for Sky UK Limited Company.

1. Index

2. Introduction

The project is divided into three stages. The first stage focuses the study off compilers theory. The second stage is about the investigation of existing tools for generating compilers code, how they work and their benefits. The third stage focuses the plugin implementation.

2.1. Compilers Theory

In the first stage it was used the set of Compilers Theory videos [1], by Alex Aiken, a Professor of Computer Science in Stanford University and the book entitled Modern Compiler Implementation in Java [2]. These two sources have a very similar approach of compilers theory, suggesting a modular implementation.

BrightScript [3] is an interpreted language and therefore there is no need to implement all the customary compilation steps. In particular, the tool only has to contain the Lexer and Parser steps. If, on the other hand, we implement the simulator, then we need to implement all the compilation steps.

Regarding the two compilation steps to be implemented in the tool, the Lexer reads the code file and generates a list of tokens (also known as tokenizer) and the Parser receives the list of tokens, performs syntax validation and generates the corresponding abstract syntax tree.

2.2. Code Generation tools

In the second stage we analyzed the use of tools to generate compiler code.

Two of the most useful abstractions used in modern compilers are context-free grammars, for parsing, and regular expressions, for lexical analysis. To make the best use of these abstractions it is helpful to have special tools, such as Yacc [4], which converts a grammar into a parsing program, and Lex [5], which converts a declarative specification into a lexical-analysis program. According to this we analyzed the following tools:

- GPlex [6]
- Gppg [7]
- Irony [8]

- JavaCC [9]
- SableCC [10]

2.3. Implementation

The third stage comprises the plugin implementation. The plugin is divided in three components, illustrated in Figure 1.

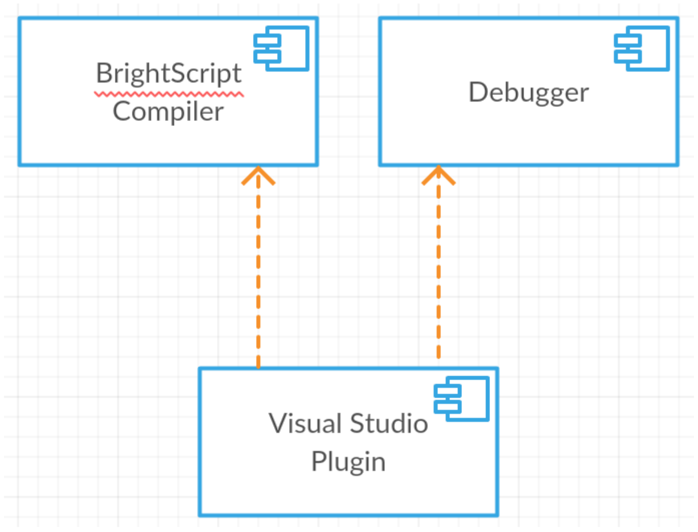


Figure 1 - Component diagram

The BrightScript Compiler is the generated code for syntax validation, therefore preventing compilation errors from occurring when deploying the code to the box.

The Debugger manages the connection with the box, using the telnet and HTTP ports. The box exposes an HTTP port to emulate remote inputs, a web page to deploy the apps and a telnet port to receive box output and send debug commands.

The Visual Studio Plugin is based on some samples, developed by Microsoft, Python Tools [11], Visual Studio Extension for Lua [12], and Visual Studio MI Debug Engine [13]. Python Tools is an extension for Visual Studio that adds Python support; Visual Studio Extension for Lua is a basic implementation for the Lua language; Visual Studio MI Debug Engine is a debugger implementation how supports the gdb Machine Interface ("MI") specification such as GDB, LLDB, and CLRDBG.

The plugin will use the compiler for syntax highlighting, syntax analysis and intellisense generation and uses the Debugger to interact with the box.

3. Compilers Theory

A compiler is a computer program that processes source code, written in a specific language, and generates machine code that computers can run.

A compiler is a very complex software. To simplify its understanding and implementation, it's modularized. Figure 2 shows compiler's components.

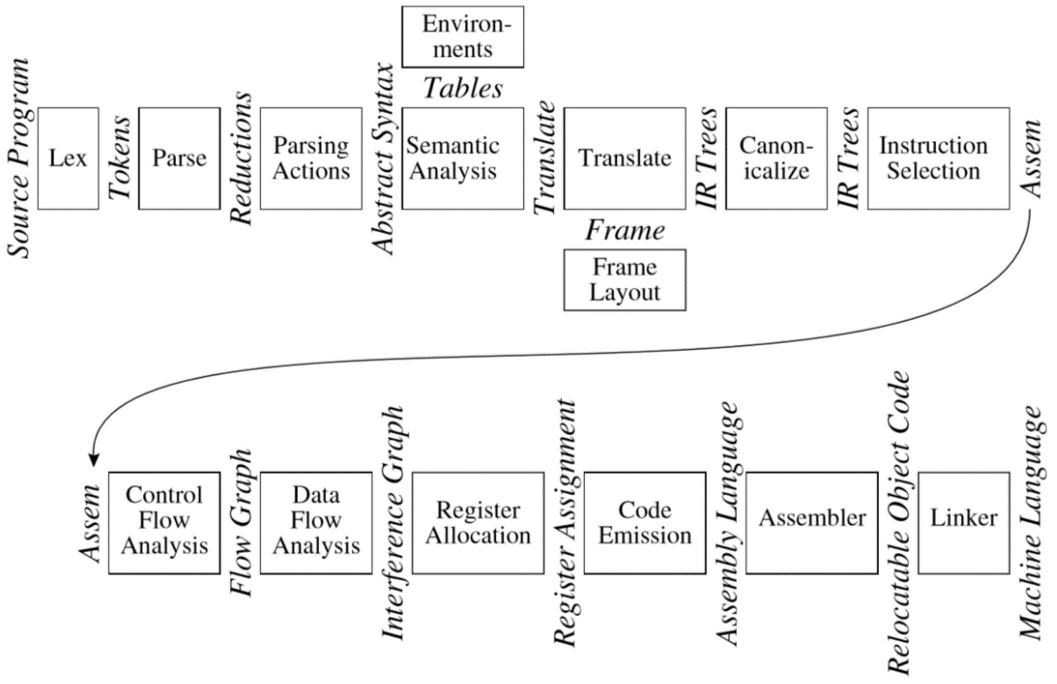


Figure 2 - Phases of a compiler, and interfaces between them, extracted from [2]

Each of the above models corresponds to a compiler phase, described in table 1.

Phase	Description
Lex	Breaks the source file into individual words or tokens.
Parse	Analyzes the phrase structure of the program.
Semantic Actions	Builds a piece of abstract syntax tree corresponding to each phrase.
Semantic Analysis	Determines what each phrase means, relates uses of variables to their definitions, checks types of expressions, requests translation of each phrase.
Frame Layout	Places variables, function-parameters, etc. into activation records (stack frames) in a machine-dependent way.
Translate	Produces intermediate representation trees (IR trees), a

	notation that is not tied to any particular source language or target-machine architecture.
Canonicalize	Hoists side effects out of expressions, and cleans up conditional branches, for the convenience of the next phases.
Instruction Selection	Groups the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
Control Flow Analysis	Analyzes the sequence of instructions into a control flow graph that shows all the possible flows of control the program might follow when it executes.
Dataflow Analysis	Gathers information about the flow of information through variables of the program; for example, liveliness analysis calculates the places where each program variable holds a still-needed value (is live).
Register Allocation	Chooses a register to hold each of the variables and temporary values used by the program; variables not live at the same time can share the same register.
Code Emission	Replaces the temporary names in each machine instruction with machine registers.

Table 1 - Compiler phases, extracted from [2]

3.1. Lexical Analysis

The lexical analyzer takes a stream of characters and generates a stream of tokens. It discards white spaces and comments between the tokens. The parser doesn't have to handle possible white space and comments at every possible point.

A lexical token is a sequence of characters that is a grammar unit of a programming language. A programming language classifies lexical tokens into a finite set of token types.

A language is a set of strings and a string is a finite sequence of symbols. The symbols themselves are taken from a finite alphabet.

To specify the language alphabet, we use regular expressions. Each regular expression represents a set of strings.

Regular expressions can specify lexical tokens, but we need an algorithm that can be implemented as a computer program.

For this we use finite automata. A finite automaton has a finite set of states, edges lead from one state to another, and each edge is labeled with a symbol.

One state is the start state, and certain of the states are distinguished as final states. Figure 3 shows a finite automaton sample.

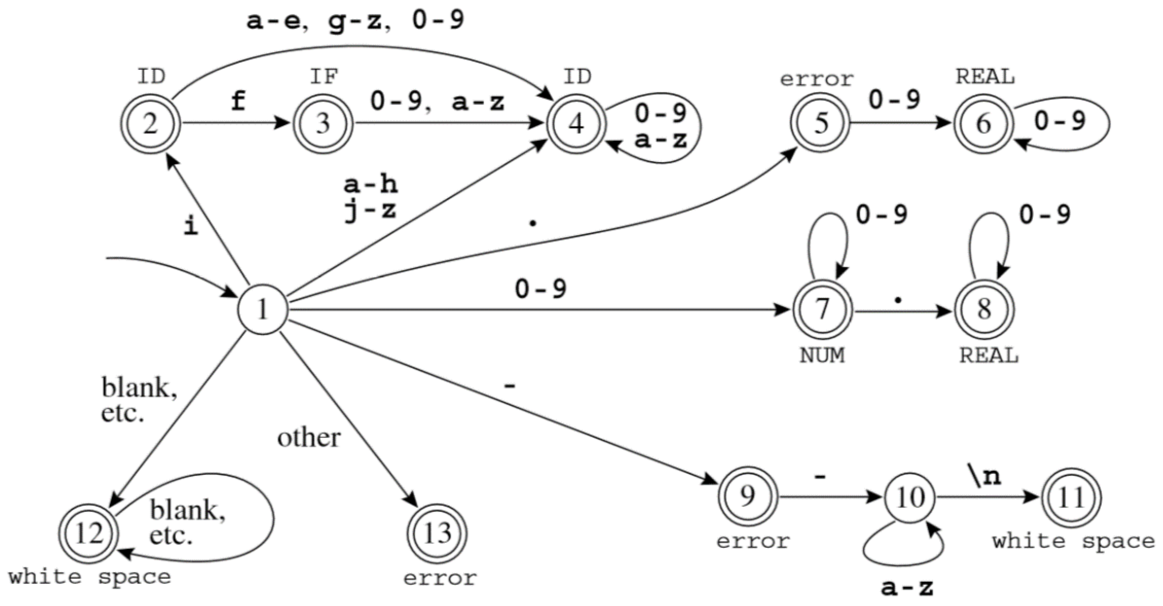


Figure 3 - Combined finite automaton, extracted from [2]

Lexical analyzer needs to find the longest matches, using the finite automata we just need to remember the last finite state. Each match corresponds to a token, after find a token the automaton reinitializes.

As result of lexical analysis phase, the lexical analyzer generates a list of tokens, to be consumed by the Parser.

3.2. Parsing

The parser makes the syntax analysis, validates the order ofwords and the way they are put together to form phrases, clauses, or sentences.

For syntax analysis we use context-free grammars. Grammars define syntactic structure declaratively. We need something more powerful than finite automata to parse languages described by grammars.

A language is a set of strings, each string is a finite sequence of symbols taken from a finite alphabet. For parsing, the strings are source programs, the symbols are lexical tokens, and the alphabet is the set of token-types returned by the lexical analyzer. A context-free grammar describes a language. A grammar has a set of productions of the form symbol → symbol symbol ···symbol

where there are zero or more symbols on the right-hand side. Each symbol could be terminal, meaning that it is a token from the alphabet of strings in the language, or non-terminal, meaning that it appears on the left-hand side of some production. Tokens cannot appear on the left-hand side of a production. One of the non-terminals is the start symbol of the grammar.

To show that this sentence is in the language of the grammar, we can perform a derivation: Start with the start symbol, then repeatedly replace any nonterminal by one of its right-hand sides. There are many different derivations of the same sentence. A leftmost derivation is one in which the leftmost nonterminal symbol is always the one expanded, in a rightmost derivation, the rightmost nonterminal is always the next to be expanded. Figure 4 shows a state table sample for a grammar.

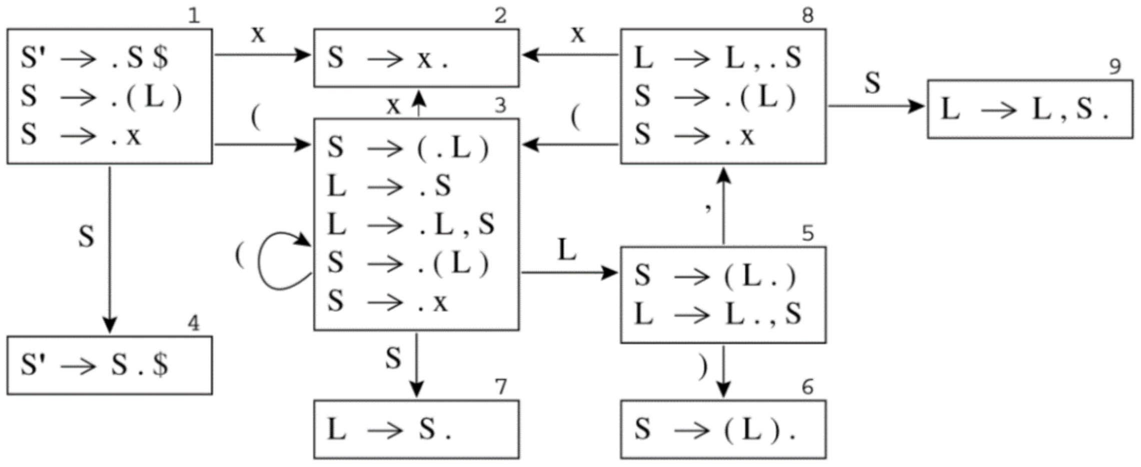


Figure 4 - LR(0) states for Grammar, extracted from [2]

A compiler does more than recognize if a sentence belongs to the language of a grammar, it performs semantic actions. In a parser implementation, semantic actions are fragments of code attached to grammar productions that generates the syntax tree.

The Abstract Syntax Tree (AST) is the output of syntax analysis phase.

3.3. Abstract Syntax Tree

Abstract syntax trees are data structures used in compilers, they can represent the structure of program code. An AST is the result of the syntax analysis phase of a compiler. It is an intermediate representation of the program that can be used in several stages that the compiler, and has a strong impact on the final output of the compiler. Figure 5 shows a syntax tree sample.

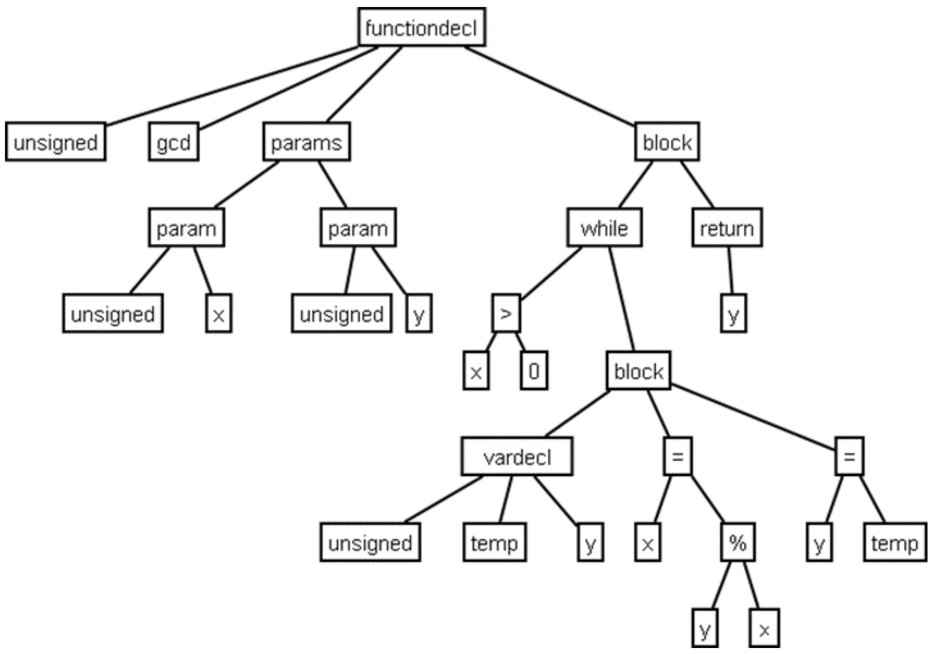


Figure 5 - Abstract Syntax Tree Sample, extracted from [2]

4. Code Generation tools

Code generation tools are used to perform simple tasks that can be automatic. Deterministic Finite Automaton (DFA) construction is a mechanical task, so it makes sense to have an automatic lexical-analyzer generator to translate regular expressions into a DFA. The task of constructing a parser can be automated too.

4.1. GPlex

Gardens Point LEX (GPlex) generates scanners based on finite state automata. The generated automata have the number of states minimized by default, and have a large number of options for table compression. The default compression scheme is chosen depending on the input alphabet cardinality, and almost always gives a reasonable result. However, a large number of options are available for the user to tune the behavior if necessary. The tool implements many of the FLEX extensions, including such things as start state stacks. The generated scanners are designed to interface cleanly with bottom-up parsers generated by Gardens Point Parser Generator (gppg).

GPlex is a scanner generator which accepts a “LEX-like” specification, and produces a C# output file. The tool does not attempt to implement the whole of the POSIX specification for LEX, however the program moves beyond LEX in some areas, such as support for Unicode.

The scanners produce by GPlex are thread safe, in that all scanner state is carried with in the scanner instance. The variables that are global in traditional LEX are instance variables of the scanner object. Most are accessed through properties which expose only a getter.

There are two main ways in which GPlex is used. In the most common case the scanner implements or extends certain types that are defined by the parser on whose behalf it works. Scanners may also be produced that are independent of any parser, and perform pattern matching on character streams. In this “stand-alone” case the GPlex tool inserts the required super type definitions into the scanner source file.

The code of the scanner derives from three sources. There is invariant code which defines the class structure of the scanner, the machinery of the pattern recognition engine, and the decoding and buffering of the input stream. These parts are defined in a “frame” file and a “buffers” file each of which is an embedded resource of the GPlex executable. The tables which define the finite state machine that performs pattern recognition, and the semantic actions that are invoked when each pattern is recognized are interleaved with the code of the frame file. These tables are created by GPlex from the user-specified “*.lex” input file. Finally, user-specified code may be embedded in the input file. All such code is inserted in the main scanner class definition. Since the generated scanner class is declared partial it is also possible for the user to specify code for the scanner class in a C# file separate from the LEX specification.

4.2. Gppg

Gardens Point Parser Generator (Gppg) is a parser generator that produces parsers written in the C# language. Gppg generates bottom-up parsers. The generated parsers recognize languages that are LALR(1), with the traditional Yacc disambiguation’s. There are a number of extensions of the traditional input language that are necessary for correctness of the generated C# output files.

A particular feature of the tool is the optional generation of an html report file that allows easy navigation of the finite state automaton that recognizes the viable prefixes of the specified language. The report shows the production items, look ahead symbols and actions for each state of the automaton. It also optionally shows an example of a shortest input, and shortest FSA-path reaching each state.

Gppg parser generator accepts a “Yacc-like” specification, and produces a C# output file. Both the parser generator and the runtime components are implemented entirely in C#.

4.3. Irony

Irony is a development kit for implementing languages on .NET platform. Unlike most existing Yacc/Lex-style solutions, Irony does not employ any scanner or parser code generation from grammar specifications written in a specialized meta-language. In Irony, the target language grammar is coded directly in C# using operator overloading to express grammar constructs. Irony's scanner and parser modules use the grammar encoded as C# class to control the parsing process.

4.4. JavaCC

Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building and parsing actions.

4.5. SableCC

SableCC is a parser generator, which generates fully featured object-oriented frameworks for building compilers, interpreters and other text parsers. In particular, generated frameworks include intuitive strictly-typed abstract syntax trees and tree walkers. SableCC also keeps a clean separation between machine-generated code and user-written code which leads to a shorter development cycle.

4.6. Conclusion

After analyzing the presented tools and the Visual Studio SDK, we get the following conclusions:

- The Visual Studio plugins only use .Net code, the Parser and the Lexer should be in C# or Visual Studio code.
- JavaCC only generates Java Lexer's and Java Parser's.
- SableCC can't generate pure .Net code, only Java in C# style.
- Irony doesn't use Yacc or Lex common definitions, the definition can't be used in other implementations.
- GPlex generates a C# Lexer based on Lex definition.
- Gppg generates a C# Parser based on Yacc definition.

The GPlex and Gppg matches all requirements and are designed to work together. According to this we chose GPlex and Gppg.

5. Compiler

The Compiler is composed of two components: The Lexer (also called Scanner) and the Lexer. The purpose of the compiler is to process code files, generating compilation errors, if they exist, and build the abstract syntax tree for intellisense functionality. Figure 6 shows components diagram of the compiler.

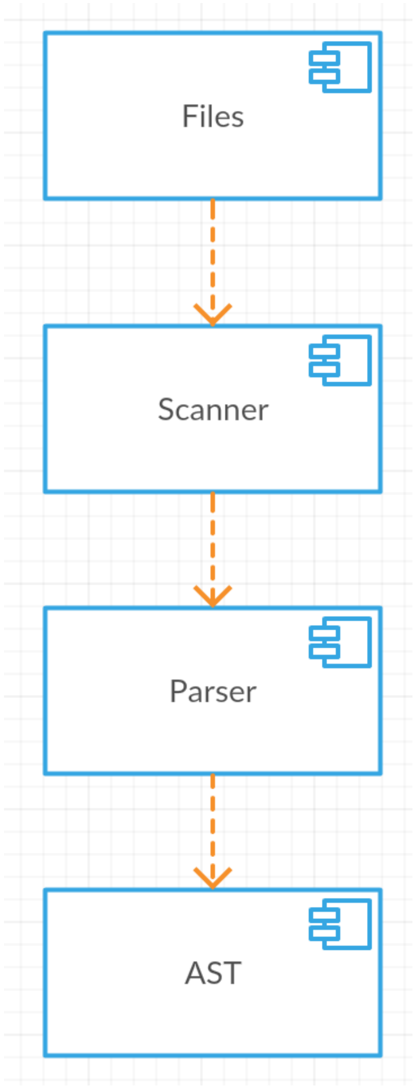


Figure 6 - Compiler Components

5.1. Lexer

The lexical analyzer makes lexical validation and generates tokens for the Parser. This validation consists in verifying if the code is according to the lexical definition for this language.

The lexical analyzer is a state machine that tries to find the longest tokens. It reads character by character changing to the possible states. When it founds a token, removes the string form the source and generates the token.

The Lexer is generated by GPlex, who generates a C# file with the state machine implementation. GPlex reads the definition file (*.lex) with the regular expressions and generates the Finite State Automata (FSA) tables. Figure 7 shows a FSA sample.

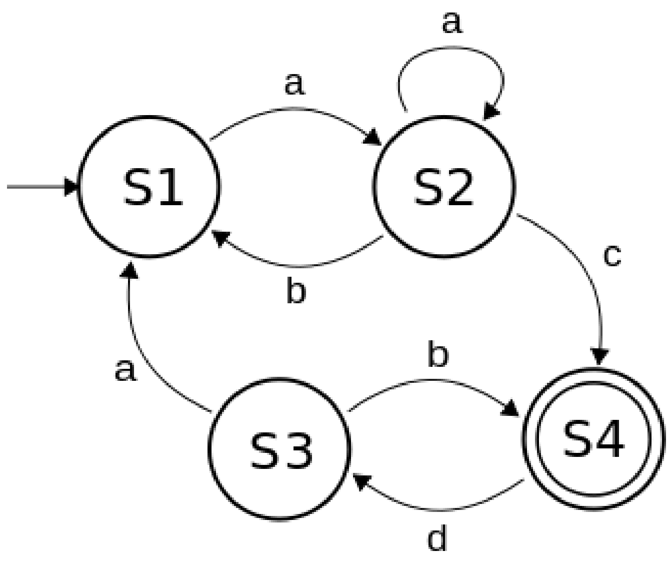


Figure 7 - FSA - Finite State Automata, extracted from [2]

The Visual Studio SDK uses the Lexer for two different purposes, the first is for syntax highlighting and the second is to generate tokens for Parser. The syntax highlighting needs less tokens then the parser, according to this we implement two Lexers one for each purpose.

5.2. Parser

The Parser analyzes the grammatical structure of the language. It validates the sentences, if the tokens are in the right order. This analysis allows the Parser to generate the abstract syntax tree (AST), which is its output.

The grammatical structure is defined by a set of rules, named context-free grammars, defined in the Yacc file. These rules define the tokens' order.

The Parser is generated by Gppg, which generates a C# file with the implementation of the Parser. The generated Parser implements the Shift-

Reduce algorithm and generates the AST by calling actions on reduce. Figure 8 shows a shift-reduce sample table.

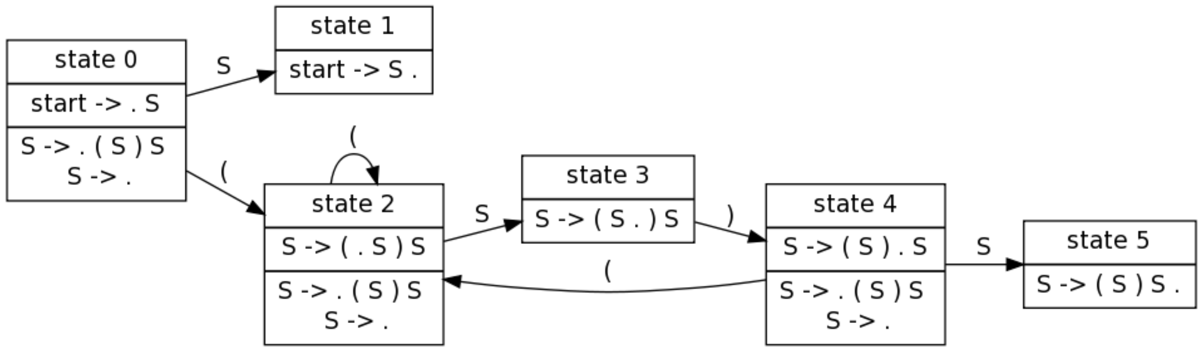


Figure 8 - Shift reduce table, extracted from [2]

Gppg allows to define actions written in C#, this actions returns the AST nodes and builds the AST using these nodes.

6. Debugger

The Debugger is the tool that performs the deployment of the app to the box, and manages the interface between the telnet port and Visual Studio.

The debugger has three main components, namely: Telnet, Deploy and Remote, as depicted in figure 9.

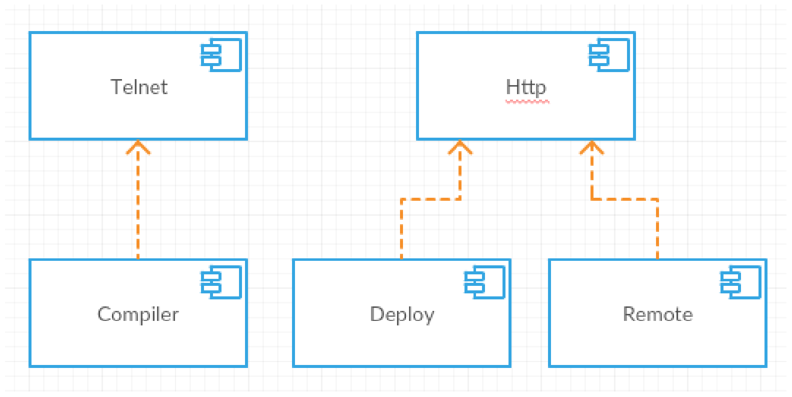
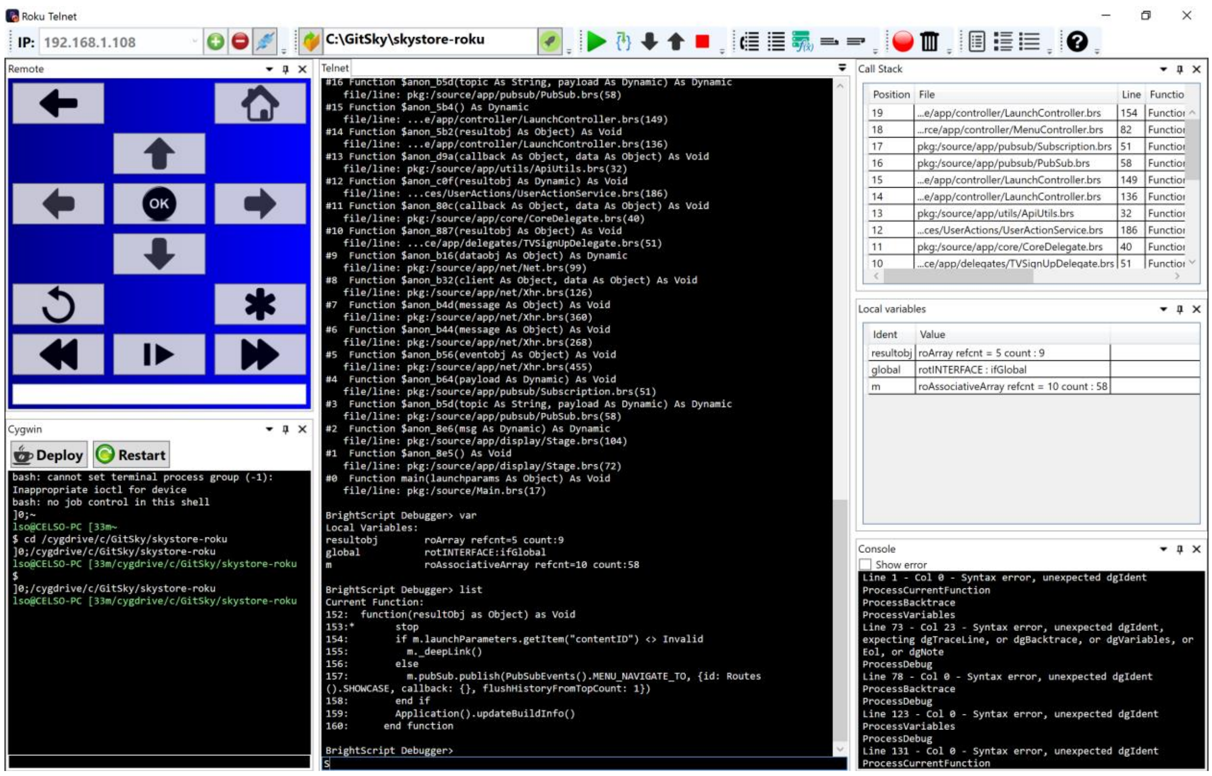


Figure 9 - Debuger diagram

The Telnet module uses a Compiler to parse the telnet output and generates debug context. The Deploy module generates the deployment package and uploads it to the box, using HTTP. The Remote module uses HTTP to emulate remote input.

The debugger will be integrated in Visual Studio Plugin, but should be independent without the Visual Studio SDK dependencies, for reuse purposes. To implement and test the functionalities we created. Figure 10 shows the debugger application.



```
#16 Function $anon_b5d(topic As String, payload As Dynamic) As Dynamic
   file/line: pkg:/source/app/pubsub/PubSub.brs(58)
#15 Function $anon_5b4() As Dynamic
   file/line: ...e/app/controller/LaunchController.brs(149)
#14 Function $anon_5b2(resultobj As Object) As Void
   file/line: ...e/app/controller/LaunchController.brs(136)
#13 Function $anon_d9a(callback As Object, data As Object) As Void
   file/line: pkg:/source/app/utils/ApiUtils.brs(32)
#12 Function $anon_c0f(resultobj As Dynamic) As Void
   file/line: ...ces/UserActions/UserActionService.brs(186)
#11 Function $anon_80c(callback As Object, data As Object) As Void
   file/line: pkg:/source/app/core/CoreDelegate.brs(40)
#10 Function $anon_887(resultobj As Object) As Void
   file/line: ...ce/app/delegates/TVSignUpDelegate.brs(51)
#9  Function $anon_b16(dataobj As Object) As Dynamic
   file/line: pkg:/source/app/net/Net.brs(99)
#8  Function $anon_b32(client As Object, data As Object) As Void
   file/line: pkg:/source/app/net/Xhr.brs(126)
#7  Function $anon_b4d(message As Object) As Void
   file/line: pkg:/source/app/net/Xhr.brs(360)
#6  Function $anon_b44(message As Object) As Void
   file/line: pkg:/source/app/net/Xhr.brs(268)
#5  Function $anon_b56(eventobj As Object) As Void
   file/line: pkg:/source/app/net/Xhr.brs(455)
#4  Function $anon_b64(payload As Dynamic) As Void
   file/line: pkg:/source/app/pubsub/Subscription.brs(51)
#3  Function $anon_b5d(topic As String, payload As Dynamic) As Dynamic
   file/line: pkg:/source/app/pubsub/PubSub.brs(58)
#2  Function $anon_8e6(msg As Dynamic) As Dynamic
   file/line: pkg:/source/app/display/Stage.brs(104)
#1  Function $anon_8e5() As Void
   file/line: pkg:/source/app/display/Stage.brs(72)
#0  Function main(launchparams As Object) As Void
   file/line: pkg:/source/Main.brs(17)

BrightScript Debugger> var
Local Variables:
resultobj        roArray refcnt=5 count:9
global           rotINTERFACE:ifGlobal
m                roAssociativeArray refcnt=10 count:58

BrightScript Debugger> list
Current Function:
152:  function(resultObj as Object) as Void
153:*       stop
154:        if m.launchParameters.getItem("contentID") <> Invalid
155:          m._deepLink()
156:        else
157:          m.pubSub.publish(PubSubEvents().MENU_NAVIGATE_TO, {id: Routes
().SHOWCASE, callback: {}, flushHistoryFromTopCount: 1})
158:        end if
159:        Application().updateBuildInfo()
160:      end function

BrightScript Debugger>
S
```

```
bash: cannot set terminal process group (-1):
Inappropriate ioctl for device
bash: no job control in this shell
]0;~
lso@CELSO-PC [33m~
$ cd /cygdrive/c/GitSky/skystore-roku
]0;/cygdrive/c/GitSky/skystore-roku
lso@CELSO-PC [33m/cygdrive/c/GitSky/skystore-roku
$
]0;/cygdrive/c/GitSky/skystore-roku
lso@CELSO-PC [33m/cygdrive/c/GitSky/skystore-roku
```

Position	File	Line	Functio
19	...e/app/controller/LaunchController.brs	154	Functio
18	...rce/app/controller/MenuController.brs	82	Functio
17	pkg:/source/app/pubsub/Subscription.brs	51	Functio
16	pkg:/source/app/pubsub/PubSub.brs	58	Functio
15	...e/app/controller/LaunchController.brs	149	Functio
14	...e/app/controller/LaunchController.brs	136	Functio
13	pkg:/source/app/utils/ApiUtils.brs	32	Functio
12	...ces/UserActions/UserActionService.brs	186	Functio
11	pkg:/source/app/core/CoreDelegate.brs	40	Functio
10	...ce/app/delegates/TVSignUpDelegate.brs	51	Functio

Ident	Value
resultobj	roArray refcnt = 5 count : 9
global	rotINTERFACE : ifGlobal
m	roAssociativeArray refcnt = 10 count : 58

```
Line 1 - Col 0 - Syntax error, unexpected dgIdent
ProcessCurrentFunction
ProcessBacktrace
ProcessVariables
Line 73 - Col 23 - Syntax error, unexpected dgIdent,
expecting dgTraceLine, or dgBacktrace, or dgVariables, or
Eol, or dgNote
ProcessDebug
Line 78 - Col 0 - Syntax error, unexpected dgIdent
ProcessBacktrace
ProcessDebug
Line 123 - Col 0 - Syntax error, unexpected dgIdent
ProcessVariables
ProcessDebug
Line 131 - Col 0 - Syntax error, unexpected dgIdent
ProcessCurrentFunction
```

Figure 10 - Debugger Application

The app is implemented according to the debugger diagram, depicted in Figure 9.

6.1. Telnet module

The Telnet module has two functionalities: to show the box output, and send it debug commands.

The box output is done using `print` or stop keywords in the source code files. The `print` keyword writes to the output, similarly to the `printf` function from C's standard library. The `stop` keyword makes puts the box in debug mode. In this mode, the box accepts debug commands, which are described in table 2.

Command	Description
bsc	Print current BrightScript Component instances
bscs	Print a summary of BrightScript component instance counts by component type.
brkd	Toggle whether BrightScript should break into the debugger after non-fatal diagnostic messages.
bt	Print backtrace of call function context frames

classes	Print Brightscript Component classes
cont or **c**	Continue Script Execution
down or **d**	Move down the function context chain one
exit	Exit shell
gc	Run garbage collector
help	Print the list of debugger commands
last	Print the last line that executed
list	List current function
next	Print the next line to execute
print, p, or **?**	Print a variable or expression
step, s, or t	Step one program statement
over	Step over function
out	Step out of a function
up or **u**	Move up the function context chain one
var	Print local variables and their types/values
Any Brightscript statement	Execute an arbitrary Brightscript statement

Table 2 - Debug commands, extracted from [3]

Using the debugger, it's possible to get the current variables' value and get information about the call stack.

The implementation of the component uses a socket to connect to the telnet port and a compiler that compiles the debugger output, to generate the debugger context. Figure 11 shows the Telnet component diagram.

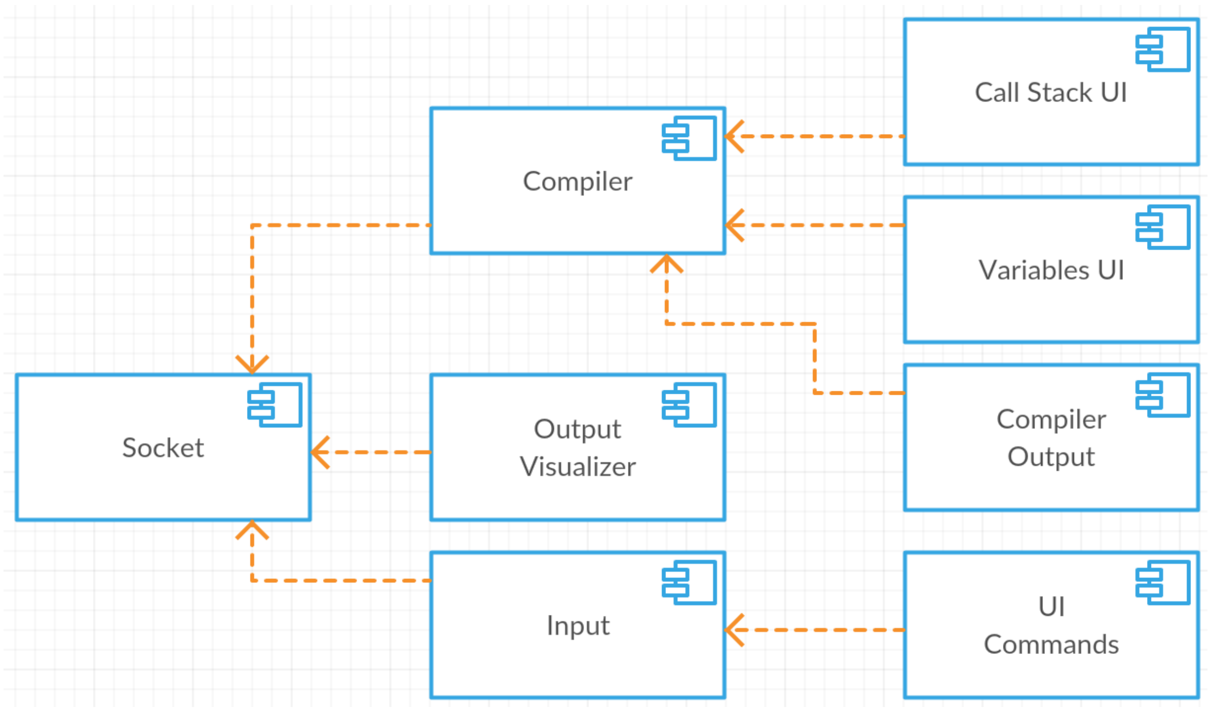
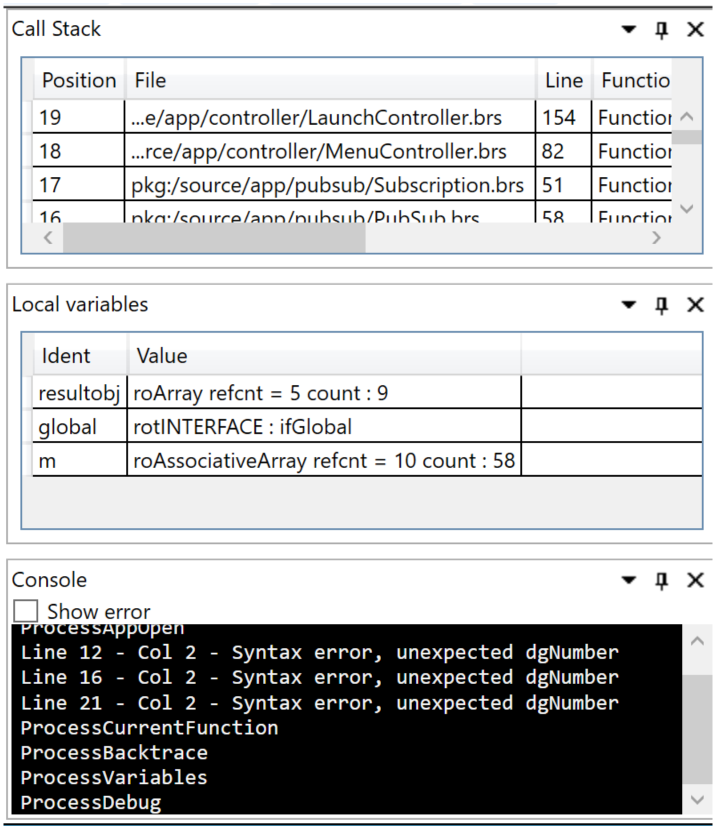


Figure 11 - Telnet component diagram

The Socket component represents the communication layer and is implemented using a socket that connects to telnet port of the box; the Compiler component uses the output of the socket to Parse this output and generates call stack and variables information; Call Stack UI and Variables UI shows the corresponding information to the user; Compiler Output shows debugging info of the Compiler; Output visualizer shows all the output of the Telnet port; Input allows the user to execute code or debug commands on the box, sending them to them through the Socket Component; UI Commands is a tool bar that exposes the debug commands as icons.

Figure 12 shows the compiler's output windows.


Call Stack

Position	File	Line	Functio
19	...e/app/controller/LaunchController.brs	154	Functior
18	...rce/app/controller/MenuController.brs	82	Functior
17	pkg:/source/app/pubsub/Subscription.brs	51	Functior
16	pkg:/source/app/pubsub/PubSub.brs	58	Functior

Local variables

Ident	Value
resultobj	roArray refcnt = 5 count : 9
global	rotINTERFACE : ifGlobal
m	roAssociativeArray refcnt = 10 count : 58

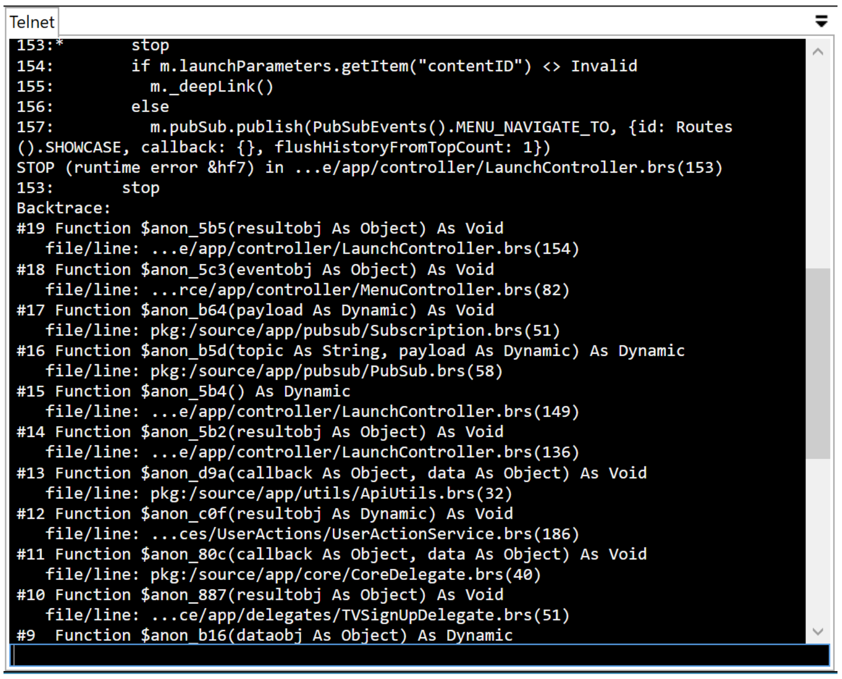
Console

Show error

```
ProcessAppOpen
Line 12 - Col 2 - Syntax error, unexpected dgNumber
Line 16 - Col 2 - Syntax error, unexpected dgNumber
Line 21 - Col 2 - Syntax error, unexpected dgNumber
ProcessCurrentFunction
ProcessBacktrace
ProcessVariables
ProcessDebug
```

Figure 12 - Compiler output windows

Figure 13 show telnet output visualizer.


Telnet

```
153:*       stop
154:        if m.launchParameters.getItem("contentID") <> Invalid
155:          m._deepLink()
156:        else
157:          m.pubSub.publish(PubSubEvents().MENU_NAVIGATE_TO, {id: Routes
().SHOWCASE, callback: {}, flushHistoryFromTopCount: 1})
STOP (runtime error &hf7) in ...e/app/controller/LaunchController.brs(153)
153:       stop
Backtrace:
#19 Function $anon_5b5(resultobj As Object) As Void
   file/line: ...e/app/controller/LaunchController.brs(154)
#18 Function $anon_5c3(eventobj As Object) As Void
   file/line: ...rce/app/controller/MenuController.brs(82)
#17 Function $anon_b64(payload As Dynamic) As Void
   file/line: pkg:/source/app/pubsub/Subscription.brs(51)
#16 Function $anon_b5d(topic As String, payload As Dynamic) As Dynamic
   file/line: pkg:/source/app/pubsub/PubSub.brs(58)
#15 Function $anon_5b4() As Dynamic
   file/line: ...e/app/controller/LaunchController.brs(149)
#14 Function $anon_5b2(resultobj As Object) As Void
   file/line: ...e/app/controller/LaunchController.brs(136)
#13 Function $anon_d9a(callback As Object, data As Object) As Void
   file/line: pkg:/source/app/utils/ApiUtils.brs(32)
#12 Function $anon_c0f(resultobj As Dynamic) As Void
   file/line: ...ces/UserActions/UserActionService.brs(186)
#11 Function $anon_80c(callback As Object, data As Object) As Void
   file/line: pkg:/source/app/core/CoreDelegate.brs(40)
#10 Function $anon_887(resultobj As Object) As Void
   file/line: ...ce/app/delegates/TVSignUpDelegate.brs(51)
#9  Function $anon_b16(dataobj As Object) As Dynamic
```

Figure 13 - Output visualizer

Figure 14 shows app tool bar.

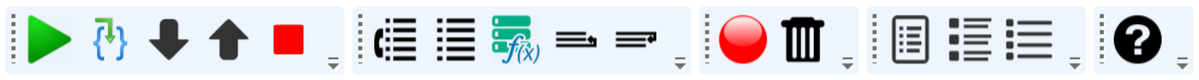


Figure 14 - UI Commands

6.2. Deploy

The deploy process consists in generating a zip file bearing the source files and uploading it to the box, using the HTTP port. The figure 15 shows the deploy steps.

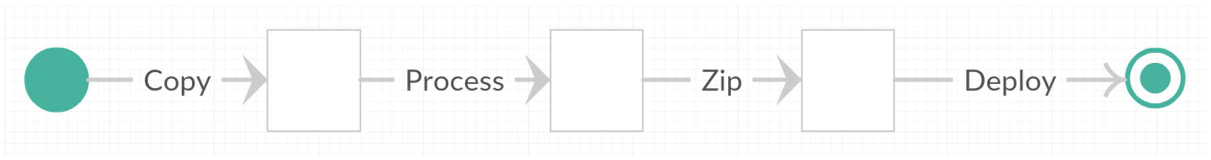
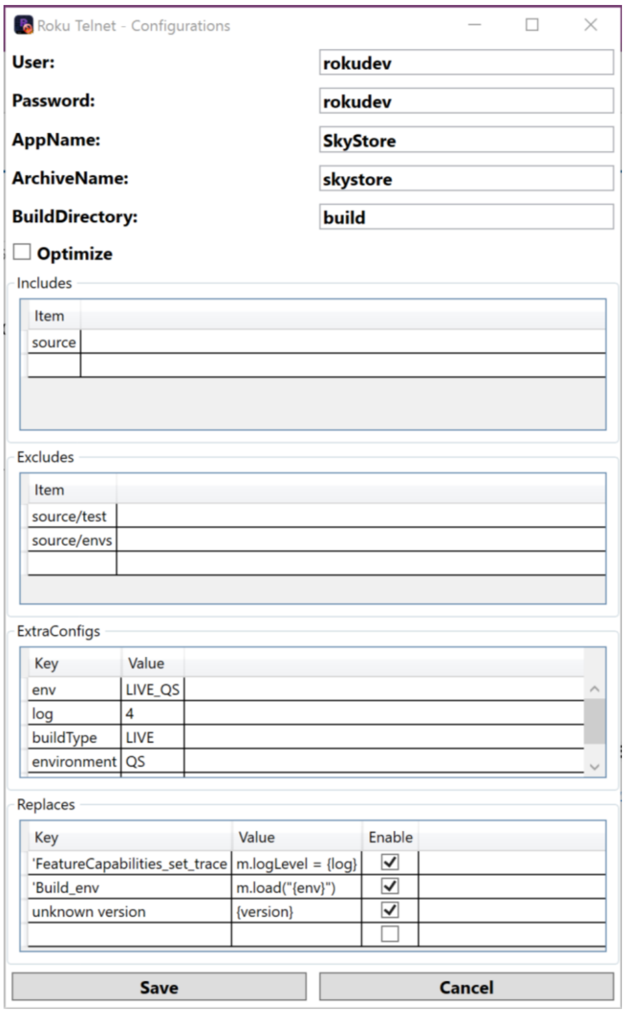


Figure 15 - Deploy process

To deploy the application, we copy all the source files to the output folder; process this files; generate a zip archive width all files; and upload it to the box.

The process step is introduced to allow the developer to pass build parameters to the deployed application. This is done by replacing specific blocks of code or removing source files from output folder. This allows to filter the folders to deploy; to remove blocks of code that was used for debug proposes; to inject code that is used to pass parameters.

The process uses several configurations. In order to support it, a page for managing the deploy configurations was created.. Figure 16 shows the configuration page.


Roku Telnet - Configurations

User:	rokudev
Password:	rokudev
AppName:	SkyStore
ArchiveName:	skystore
BuildDirectory:	build

☐ **Optimize**

Includes

Item
source

Excludes

Item
source/test
source/envs

ExtraConfigs

Key	Value
env	LIVE_QS
log	4
buildType	LIVE
environment	QS

Replaces

Key	Value	Enable
'FeatureCapabilities_set_trace	m.logLevel = {log}	✓
'Build_env	m.load("{env}")	✓
unknown version	{version}	✓
		☐

Figure 16 - Configurations Window

The upload step requires the developer’s credentials (i.e. username and password) to be used on accesses to HTTP port of the box. These credentials are specified at the top of the configurations window; The optimize configuration removes comments, empty lines and extra spaces from code files; The section named *includes* contains the list of sub folders to include in the zip file that will be uploaded; The *exclude* section contains the list of sub folders to exclude from deploy; The extra configs and the Replaces section allows the developer to pass custom parameters to the deployed application, this can be done just by define the replace key and the replace value. The replace key is the block of code that will be replaces and the value is the block of code that will replace it. The value of Replaces can use the extra configs, if the value of replace has a key of extra config between curly braces this block will be replaced for extra config value before the replace ware applied to source files.

6.3. Remote

The Remote component is a remote emulator that allows the developer to control the box using the PC. The component has common keys of Roku's remotes and has an additional text box that allows the developer to send text to box.

The implementation of this component uses the Roku External Control Service Commands over HTTP, that allows to send remote instructions.

Figure 17 shows remote window.

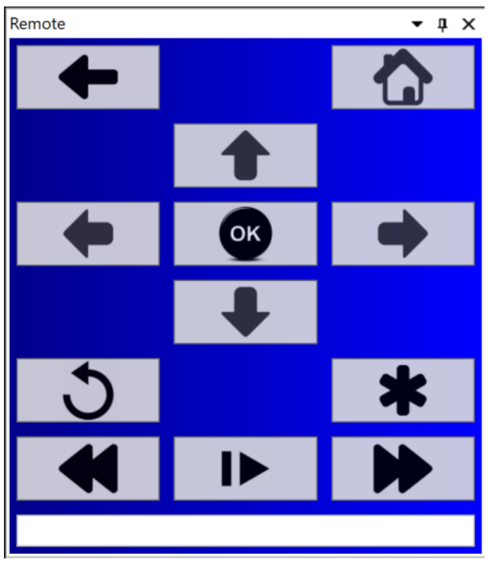


Figure 17 - Remote

Using this component the developer can debug and control the box using the same application.

7. Visual Studio Plugin

Visual Studio behavior can be extended in several ways. We are using “MefComponent”, “VsPackage”, “ProjectTemplate” and “ItemTemplate” extension points.

The “MefComponent” uses Mef [13], MEF is a library for creating lightweight, extensible applications. It allows Visual Studio to discover and use extensions with no configuration required.

The “*VsPackages*” [15] are software modules that extend the Visual Studio integrated development environment (IDE) by providing UI elements, services, projects, editors, and designers

The “*ProjectTemplates*” [16] are templates to create new project files, this templates can be selected when the developer creates a new project. The project file is a MSBuild that describes the project. The project description has all files of the project and the project configurations.

The “*ItemTemplates*” [17] are templates to create new items for the project, can be used when the developer creates a new file in the project.

The Visual Studio Plugin implementation is separated on four different components, the project type, the builder/deploy, the editor extensions and the debugger.

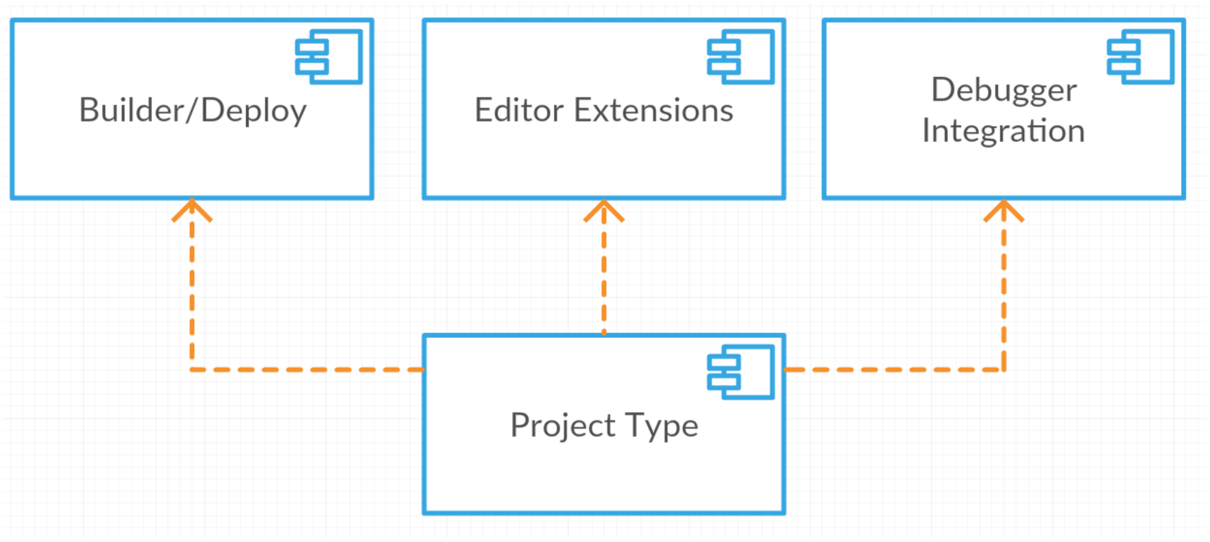


Figure 18 - Visual Studio Plugin diagram

Project Type component is the main component and generates the plugin installation. This component is also responsible to expose the project file template and the code files template, these templates will be used by the Visual Studio to create new projects and/or new code files.

Builder/Deploy Component implements the MSBuild tasks to use when the developer builds or runs the project.

Editor Extensions Component extends the Visual Studio editor adding syntax highlighting, error validation, and intellisense.

Debugger integration implements the Debugger Engine. This component receives debugger commands and send debug events to Visual Studio.

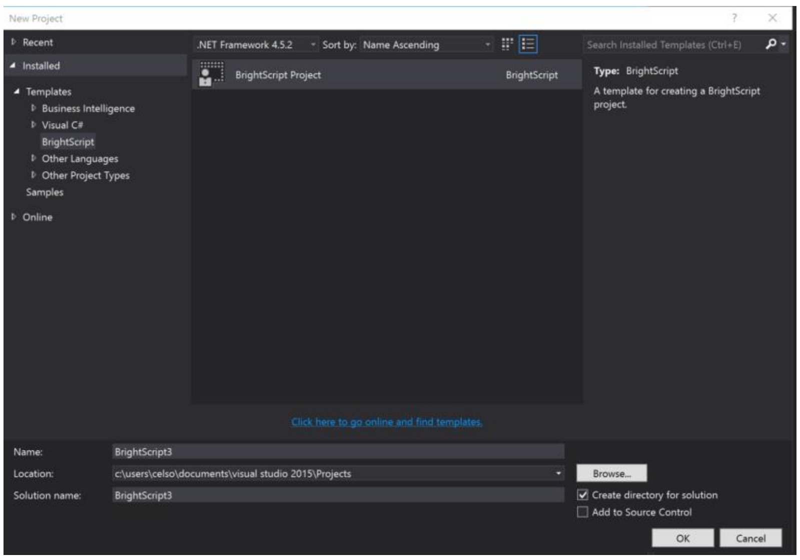
7.1. Project Type

The project type provides the templates to create the BrightScript project and for create the code files.

In project type implementation we used VSProjectSystem [14], this base implementation has the most common code to create project types.

The project template defines the base project file and the default items to be created on project creation. The project file is MSBuild project [19] file, written in XML. The MSBuild project defines the files to appear in project solution, the build action for each file, project configurations, and the workflow for build and deploy the app. The workflow uses the tasks described on Builder/Deploy section.

The project template needs to be registered on plugin to appears on new project dialog. Figure 19 shows the project dialog window width new project type.



.NET Framework 4.5.2 Sort by: Name Ascending

BrightScript Project BrightScript

Type: BrightScript

A template for creating a BrightScript project.

Name: BrightScript3

Location: c:\users\celso\documents\visual studio 2015\Projects

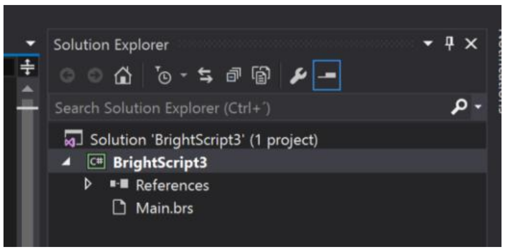
Solution name: BrightScript3

[✓] Create directory for solution

[ ] Add to Source Control

Figure 19 - Visual Studio project dialog

The project template contains the set of base code files that are added to the solution upon project creation. Figure 20 shows project structure, after being created.



Solution 'BrightScript3' (1 project)
BrightScript3
References
Main.brs

Figure 20 - Project created

Using VSProjectSystem [14], we need to define all item templates and register them, for the code files to appear in the solution explorer.

The item templates are the base set of code files, that appears on new item dialog and contains the base code structure. Figure 21 shows the new item dialog.

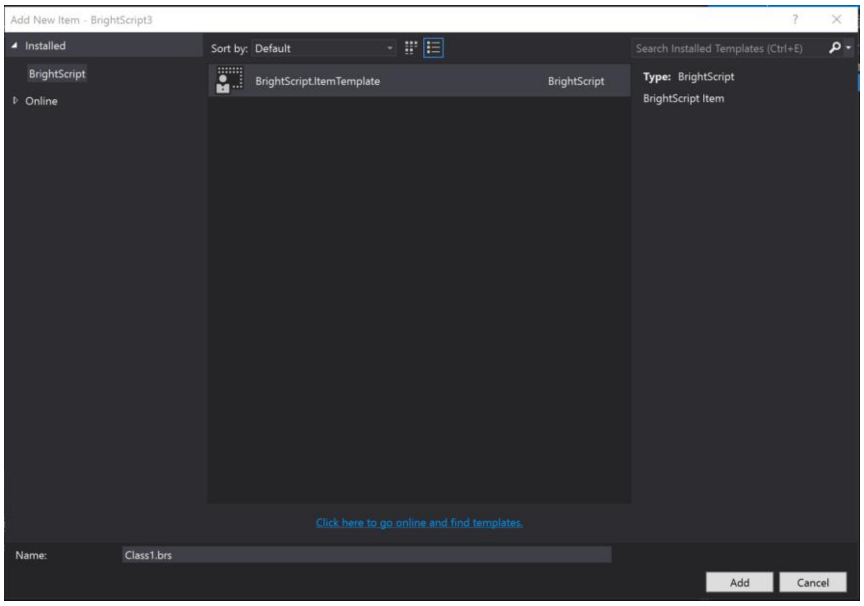
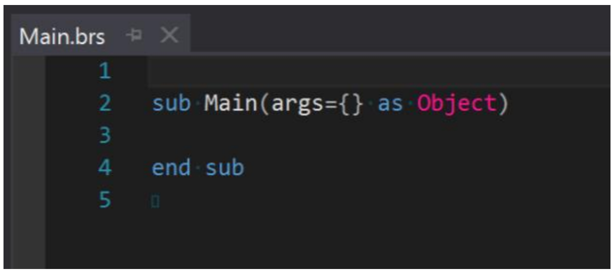

Add New Item - BrightScript3
Installed
BrightScript
Online
Sort by: Default
BrightScript.ItemTemplate
BrightScript
Search Installed Templates (Ctrl+E)
Type: BrightScript
BrightScript Item
Click here to go online and find templates.
Name:
Class1.brs
Add
Cancel

Figure 21 - New item dialog

Figure 22 shows create file, after being created.



```

sub Main(args={} as Object)

end sub

```

Figure 22 - Base code file

7.2. Builder/Deploy

The builder/deploy component implement's the tasks to be used on build process. The behavior is the same of described in Debugger section, but there's an additional *CompileTask* step, this task uses the compiler to validate the code before deploy.

Table 3 describes all the created tasks:

Task	**Description**
CompileTask	Uses the Compiler to validate the code before deploy.
CopyToOutputTask	Copy files to output path
GitVersionTask	Generates a output parameter width last tag from Git repository
ManifestTask	Edits manifest file replacing app name and version
ReplacesTask	Replace specific file contents. Used to pass parameters

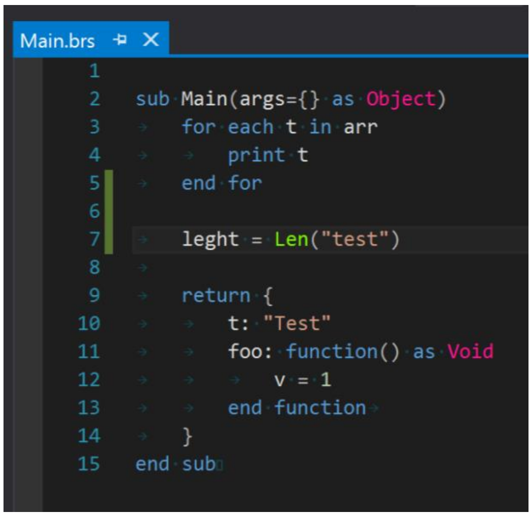
	for app
OptimizeTask	Remove following code: • Empty lines • Comments • Duplicated spaces • Code between '%--' and '--%' tags
ArchiveTask	Generates the zip file to be deployed
DeployTask	Uploads the zip file to the box

Table 3 – MSBuild tasks

7.3. Editor Extension

The editor extensions provide syntax highlighting, error validation and intellisense.

The syntax highlighting is supported by the Lexer, that generates tokens for syntax highlighting. To expose that functionality, we need to create and export a class that implements the *"ITaggerProvider"* interface. Figure 23 depicts the editor window displaying syntax highlighting for the BrightScript language.

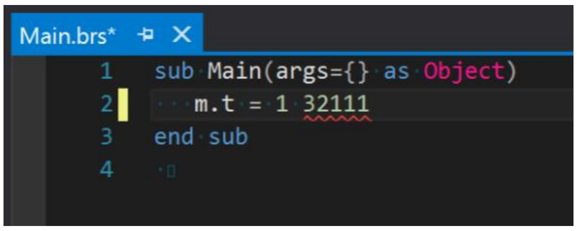


```

sub Main(args={} as Object)
    for each t in arr
        print t
    end for

    leght = Len("test")

    return {
        t: "Test"
        foo: function() as Void
            v = 1
        end function
    }
end sub
```

Figure 23 - Syntax highlighting

The compiler errors are supported by the Parser which compiles the source code and generates the existing compiling errors. Compiling errors are displayed on the editor as underlining mark and are listed on the errors window, as depicted in figure 24 and figure 25.

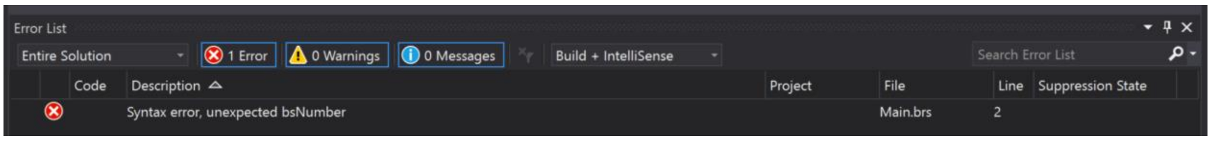
This functionality is supported by exporting a “*ITaggerProvider*” implementation of type “*ErrorTag*”. The Error tagger provides the existing compilation errors.



```
sub Main(args={} as Object)
    m.t = 1 32111
end sub
```

Figure 24 - Editor error

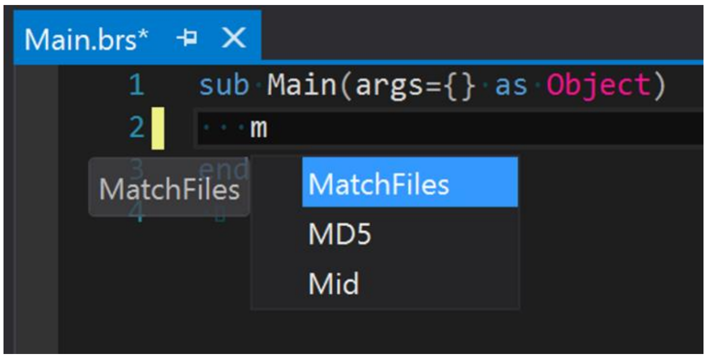
The error window uses “*ErrorListProvider” class* to show the generated errors. Figure 25 shows an example of an on the error on error window.



	Code	Description	Project	File	Line	Suppression State
⊗		Syntax error, unexpected bsNumber		Main.brs	2	

Figure 25 - Error window

The intellisense component uses the Parser to generate the AST and exports an implementation of “*ICompletionSourceProvider*” interface to generate the list of suggestions. We need to implement “*IVsTextViewCreationListener*” to register the command handler to show the list. Figure 26 shows the intellisense popup.



```
sub Main(args={} as Object)
    m
```

Figure 26 - intellisense

7.4. Debugger integration

The debugger was based on Visual Studio MI Debug Engine [13] and use the implementation developed for Debugger App, that manage the connection with the box and shows debug output. The debugger output and debugger commands will be integrated with Visual Studio debugger extension.

To implement a debugger, we need to perform three tasks. Implement Debug Engine, Register Debug Engine, and create a debug launcher.

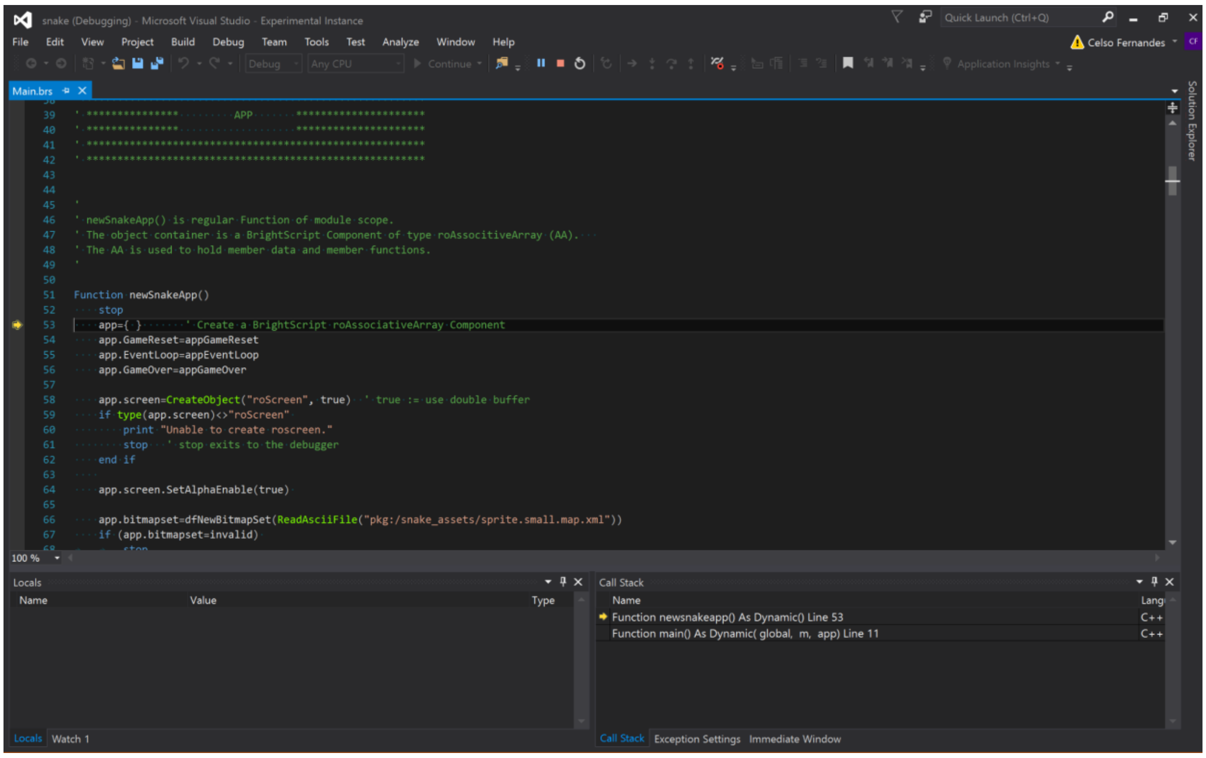
Debug engine component manages the interaction between the Visual Studio and our debugger. The implementation is based on following componentes: “AD7Engine”, “AD7Thread”, “AD7StackFrame”, “AD7DocumentContext”, “AD7Events”, “Structures”.

“AD7Engine” exposes methods for Launch, Resume, Suspend, Step, and Terminate debugger, manages the main interactions width Visual Studio “AD7Thread” represents the current thread on debugging app, exposes thread id, thread name, and current stack list. “AD7StackFrame” represents a frame of stack list. “AD7DocumentContext” represents the document and line of specific stack frame.“AD7Events” implements the debugger events, to be send to Visual Studio. Using these events, we can pass current state of debugger to Visual Studio. For example, when program hits a break point. “Structures” are Structures used to pass data on events.

Debugger engine needs to be registered in Windows register to be used, this is implemented using a registration attribute, used on Visual Studio package.

Visual Studio needs a Launcher implementation to specify witch Debug Engine to create.

Figure 27 shows Visual Studio in debug mode.



```
' ***************         APP      ********************
' ***************                  ********************
' ******************************************************
' ******************************************************


'
' newSnakeApp() is regular Function of module scope.
' The object container is a BrightScript Component of type roAssocitiveArray (AA).
' The AA is used to hold member data and member functions.
'

Function newSnakeApp()
    stop
    app={ }       ' Create a BrightScript roAssociativeArray Component
    app.GameReset=appGameReset
    app.EventLoop=appEventLoop
    app.GameOver=appGameOver

    app.screen=CreateObject("roScreen", true)   ' true := use double buffer
    if type(app.screen)<>"roScreen"
        print "Unable to create roscreen."
        stop   ' stop exits to the debugger
    end if

    app.screen.SetAlphaEnable(true)

    app.bitmapset=dfNewBitmapSet(ReadAsciiFile("pkg:/snake_assets/sprite.small.map.xml"))
    if (app.bitmapset=invalid)
```

Locals

Name	Value	Type

Call Stack

Name	Lang
Function newsnakeapp() As Dynamic() Line 53	C++
Function main() As Dynamic(global, m, app) Line 11	C++

Figure 27 - Debugging BrightScript

8. Conclusion

When the project starts we need to collect all the knowledge of theory and how to make IDEs extension. It was a long research to get all knowledge and it needs to be refresh along the implementation. When we join all the pieces it was very gratefully to see the result.

The final project will be an integrated tool for Roku development that contains all the features on other tools, and supports code editing, syntax highlighting, error validation on development time, intellisense, build and deploy the application, and integrated debug.

The Visual Studio plugin isn't yet usable, but the debugger app is a very useful tool on day to day work and it makes much more easy to debug the apps. As future work to make this tool useful we identify the following future work:

- Fix debugger issues
- Add support to all code statements
    - Some code blocks aren't yet supported
    - Use reserved words as function identifiers
- Integrate Scene Graph framework
    - Scene Graph Parser
    - Debug additional threads

9. References

[1] A. Aiken, "Compilers Theory," [Online]. Available: https://www.youtube.com/playlist?list=PLLH73N9cB21VSVEX1aSRlNTufaLK1dTAI.

[2] A. W. Appel, Modern Compiler Implementation in Java, Cambrige University Press, 2002.

[3] "BrightScript," [Online]. Available: https://sdkdocs.roku.com/display/sdkdoc/Roku+SDK+Documentation.

[4] [Online]. Available: http://dinosaur.compilertools.net/#yacc.

[5] [Online]. Available: http://dinosaur.compilertools.net/#lex.

[6] "GPlex," [Online]. Available: http://gplex.codeplex.com/.

[7] "Gppg," [Online]. Available: https://gppg.codeplex.com/.

[8] [Online]. Available: https://irony.codeplex.com/.

[9] [Online]. Available: https://javacc.java.net/.

[10] [Online]. Available: http://www.sablecc.org/.

[11] "Python Tools," [Online]. Available: https://github.com/Microsoft/PTVS.

[12] "Visual Studio Extension for Lua," [Online]. Available: https://github.com/Microsoft/VSLua.

[13] Microsoft, "Visual Studio MI Debug Engine," [Online]. Available: https://github.com/Microsoft/MIEngine.

[14] "MEF - Managed Extensibility Framework," [Online]. Available: https://msdn.microsoft.com/en-us/library/dd460648(v=vs.110).aspx.

[15] "VSPackages," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/bb166424.aspx.

[16] "Creating Project and Item Templates," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/s365byhx(v=vs.100).aspx.

[17] "How to: Create Item Templates," Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/tsyyf0yh.aspx.

[18] "VSProjectSystem," [Online]. Available: https://github.com/Microsoft/VSProjectSystem/.

[19] "MSBuild Project File Schema Reference," [Online]. Available: https://msdn.microsoft.com/en-us/library/5dy88c2e.aspx.

[20] "PowerShell Tools," [Online]. Available: https://github.com/adamdriscoll/poshtools.

10. Figures Index