Instituto Superior de Engenharia de Lisboa

Electronics, Telecommunications and Computers Engineering Department

Debugger BrightScript

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

outubro de 2016

Resume

BrigthScript is a programming language based on JavaScript and Visual Basic, created by Roku. Roku is a company who develops and sells boxes to watch movies and television. BrightScript is the language to develop applications for their boxes.

After some analysis, we could not found many development tools and they are not very functional. Roku offer’s 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 most used text editors, who make syntax highlighting.

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

The tool is a Visual Studio plugin for BrightScript language. Visual Studio is a development IDE created by Microsoft and it’s the main tool for develop Windows applications. This plugin will use language services provided by Visual Studio SDK.

The tool cloud be enhanced with a box simulator to run the applications on the development machine, which will be a future work and could be a great benefit.

The idea for this project comes from the author’s involvement on SkyStore Roku App development, for Sky UK Limited company.

1. Index

Resume i

1. Index ii

2. Introduction 1

   2.1. Compilers Theory 1

   2.2. Code Generation tools 1

   2.3. Implementation 2

3. Compilers Theory 3

   3.1. Lexical Analysis 4

   3.2. Parsing 5

   3.3. Abstract Syntax 7

4. Code Generation tools 8

   4.1. GPlex 8

   4.2. Gppg 9

   4.3. Irony 9

   4.4. JavaCC 10

   4.5. SableCC 10

   4.6. Conclusion 10

5. Compiler 11

   5.1. Lexer 11

   5.2. Parser 12

6. Debugger 14

   6.1. Telnet 15

   6.2. Deploy 19

   6.3. Remote 21

7. Visual Studio Plugin 23

   7.1. Project Type 23

   7.2. Builder/Deploy 25

   7.3. Editor Extension 25

7.4. Debugger integration 27
8. Conclusion 28
9. References 29
10. Figures Index 30

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 (Compilers Theory, s.d.), by Alex Aiken, a Professor of Computer Science in Stanford University and the book entitled Modern Compiler Implementation (Modern Compiler Implementation in Java, 2002) in Java. These two sources have a very similar approach of compilers theory, suggesting a modular implementation.

The BrightScript (BrightScript, s.d.) 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.

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 (YACC, s.d.)(which converts a grammar into a parsing program) and Lex (Lex, s.d.) (which converts a declarative specification into a lexical-analysis program). According to this we analyzed the following tools:

- GPlex (GPlex, s.d.)
- Gppg (Gppg, s.d.)

- Irony (Irony - .Net Language Implementation Kit, s.d.)
- JavaCC (JavaCC, s.d.)
- SableCC (SableCC, s.d.)

2.3. Implementation

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

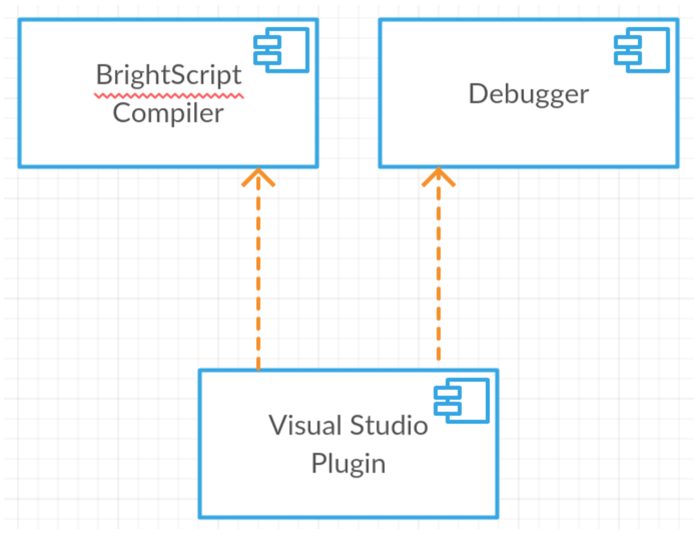


Figure 1 - Component diagram

The BrightScript Compiler is the generated code for syntax validation, it will prevent box compilation errors.

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

The Visual Studio Plugin will be based on some samples, developed by Microsoft, Python Tools (Python Tools, s.d.) and Visual Studio Extension for Lua (Visual Studio Extension for Lua, s.d.). Python Tools is an extension for Visual Studio that adds support for python language, Visual Studio Extension for Lua is a most simpler implementation for Lua language.

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 special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.

A compiler is a large software system, for much easy understand and implement, it should be modularized. For module communication should be defined interfaces between them. Figure 2 shows this models and interfaces.

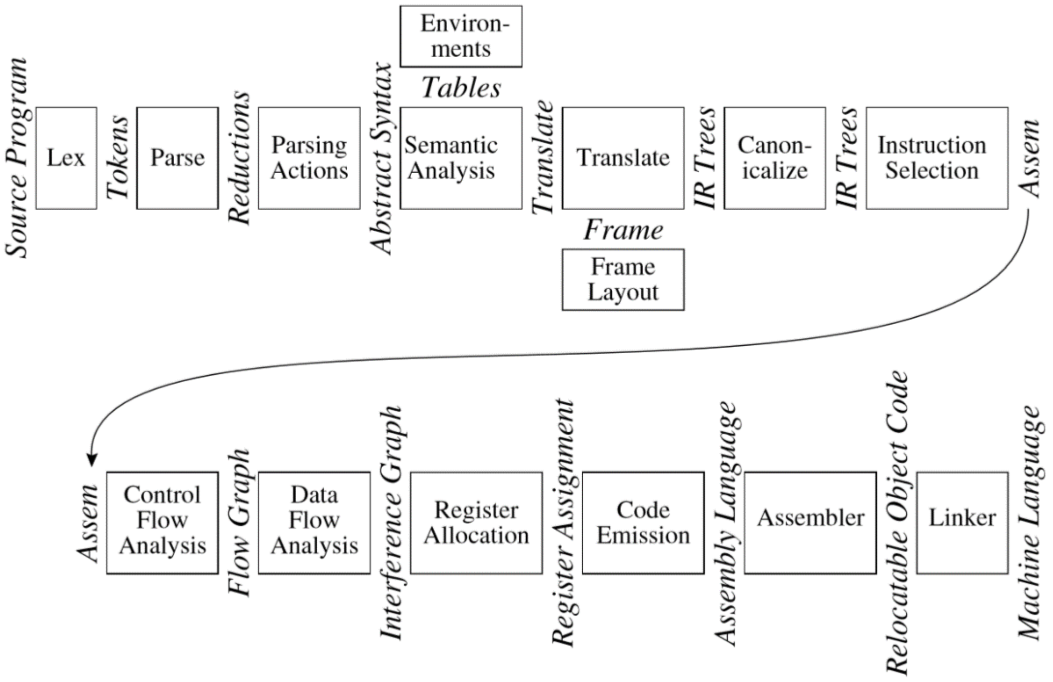


Figure 2 - Phases of a compiler, and interfaces between them

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

Phase	**Description**
Lex	Break the source file into individual words or tokens.
Parse	Analyze the phrase structure of the program.
Semantic Actions	Build a piece of abstract syntax tree corresponding to each phrase.
Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
Frame Layout	Place variables, function-parameters, etc. into activation

	records (stack frames) in a machine-dependent way.
Translate	Produce intermediate representation trees (IR trees), a notation that is not tied to any particular source language or target-machine architecture.
Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for the convenience of the next phases.
Instruction Selection	Group the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
Control Flow Analysis	Analyze 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	Gather information about the flow of information through variables of the program; for example, liveness analysis calculates the places where each program variable holds a still-needed value (is live).
Register Allocation	Choose 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	Replace the temporary names in each machine instruction with machine registers.

Table 1 - Compiler phases

3.1. Lexical Analysis

The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks. It discards white space and comments between the tokens. The parser doesn’t have to account for possible white space and comments at every possible point. This is the main reason for separating lexical analysis from parsing.

A lexical token is a sequence of characters that can be treated as a unit in the grammar 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 some of these (possibly infinite) languages with finite descriptions, we will use the notation of regular expressions. Each regular expression stands for a set of strings.

Regular expressions are convenient for specifying lexical tokens, but we need a formalism that can be implemented as a computer program.

For this we can 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.

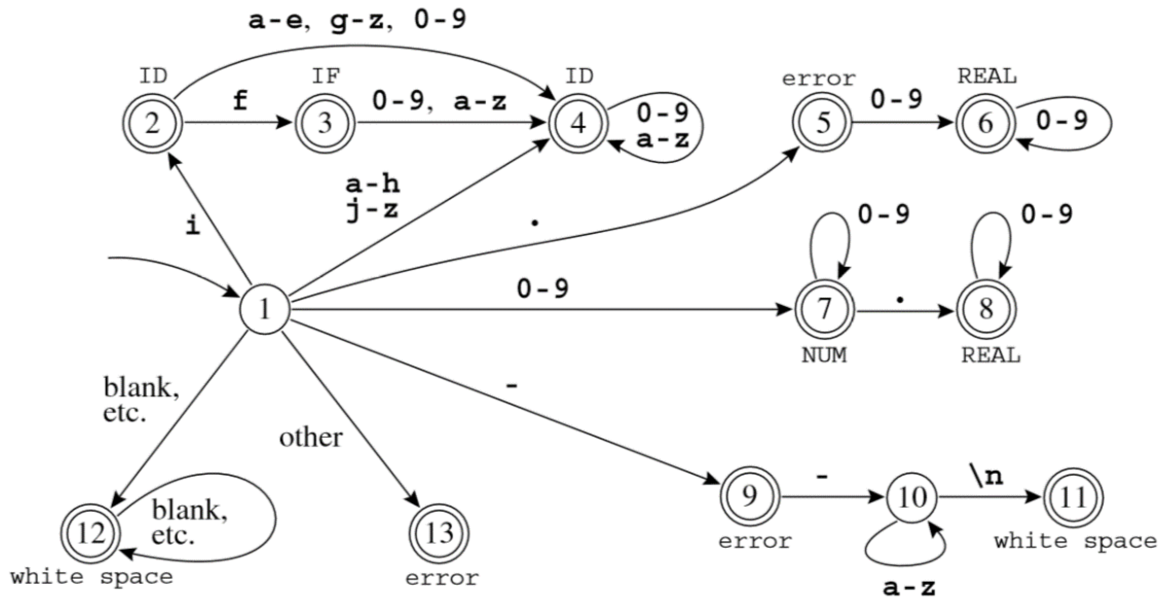


Figure 3 - Combined finite automaton

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 way in which words are put together to form phrases, clauses, or sentences.

For syntax analysis we use a simple notation called context-free grammars. Just as regular expressions can be used to define lexical structure in a static, declarative way, grammars define syntactic structure declaratively. But we will 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 is either 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. No token can ever appear on the left-hand side of a production. Finally, one of the non-terminals is distinguished as 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.

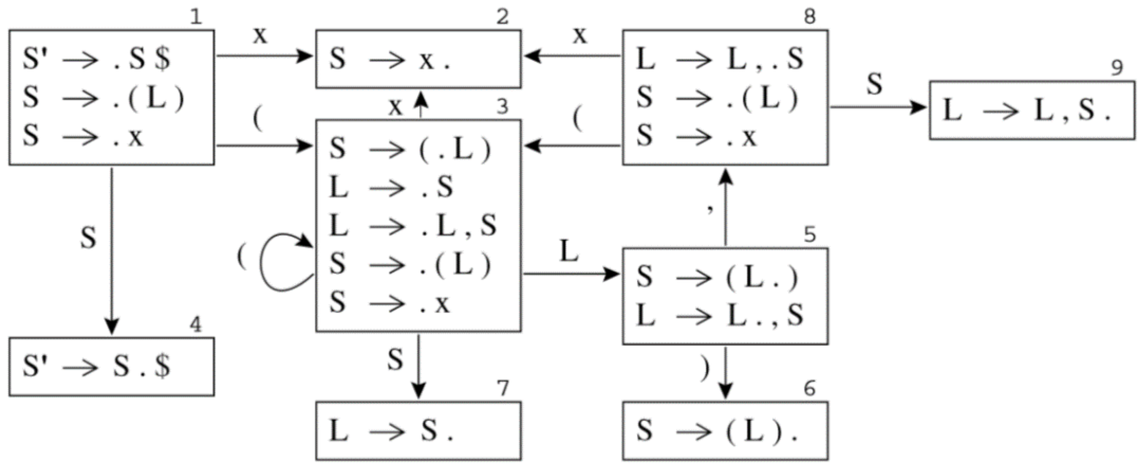


Figure 4 - LR(0) states for Grammar

A compiler must do more than recognize whether a sentence belongs to the language of a grammar, it must do something useful with that sentence. The semantic actions of a parser can do useful things with the phrases that are parsed. In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing 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 result of syntax analysis phase.

3.3. Abstract Syntax

Abstract syntax trees are data structures widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler. Figure 5 shows a syntax tree sample.

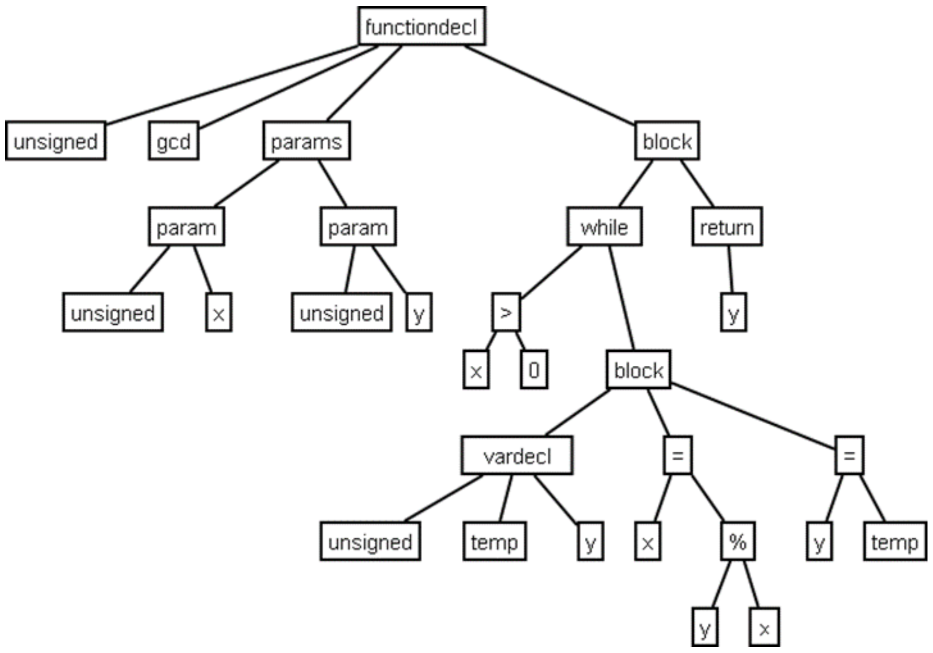


Figure 5 - Abstract Syntax Tree Sample

4. Code Generation tools

The code generation tools could be used to perform simple tasks, that can be automatic. 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 by two components, the Lexer (also called Scanner) and the Lexer. The purpose of the compiler is to process code files generating compilation errors and build the abstract syntax tree for intellisense functionality. Figure 6 shows components diagram of the compiler.

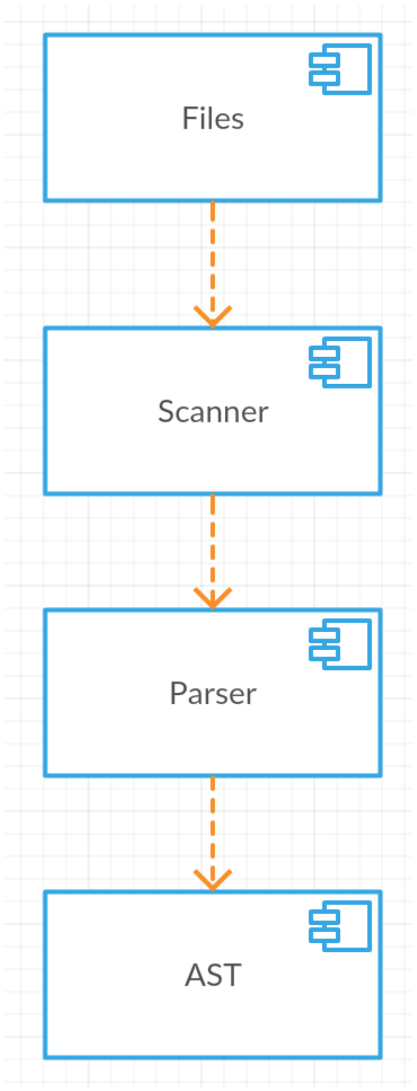


Figure 6 - Compiler Components

5.1. Lexer

The lexical analyzer makes lexical validation and generates tokens for Parser. This validation consists in verify 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.

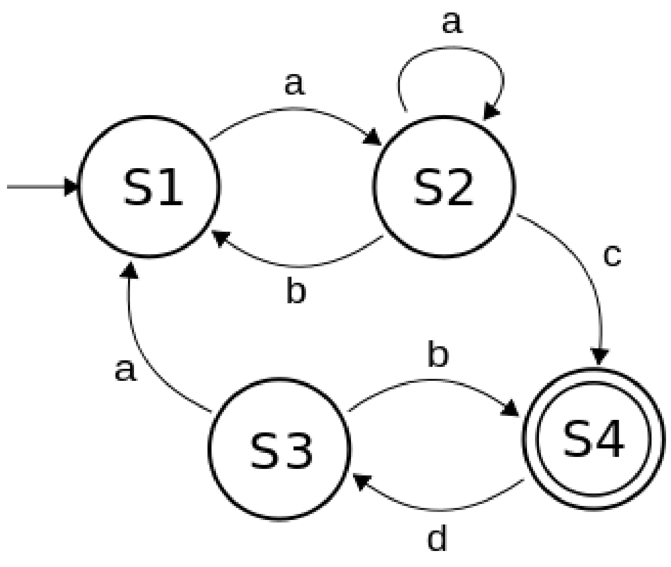


Figure 7 - FSA - Finite State Automata

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 analyzes allow the Parser to generate, the abstract syntax tree (AST), which is the output of the Parser.

The grammatical structure is defined by a set of rules (context-free grammars), defined in the Yacc file, this 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. Figure 8 shows a shift reduce sample table.

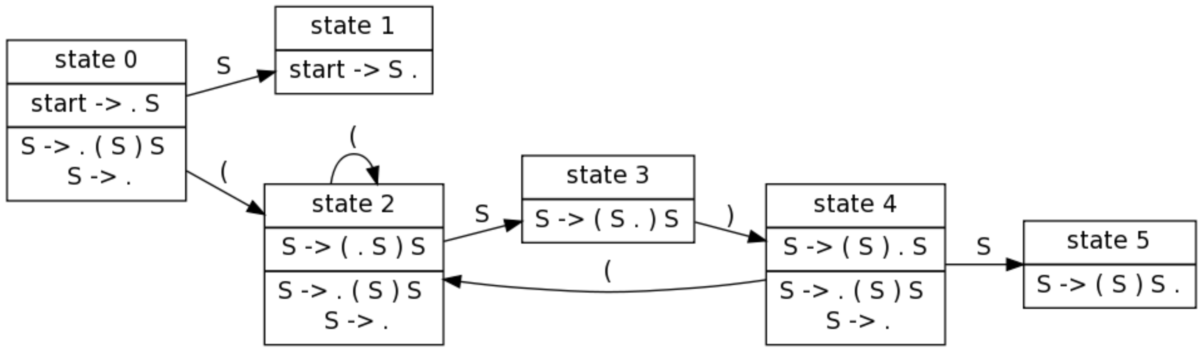


Figure 8 - Shift reduce table

6. Debugger

The Debugger is the tool that allows to deploy the app to the box and manage the interface between the telnet port and the Visual Studio.

The debugger has three main components, the Telnet, the Deploy and the Remote, illustrated in figure 9.

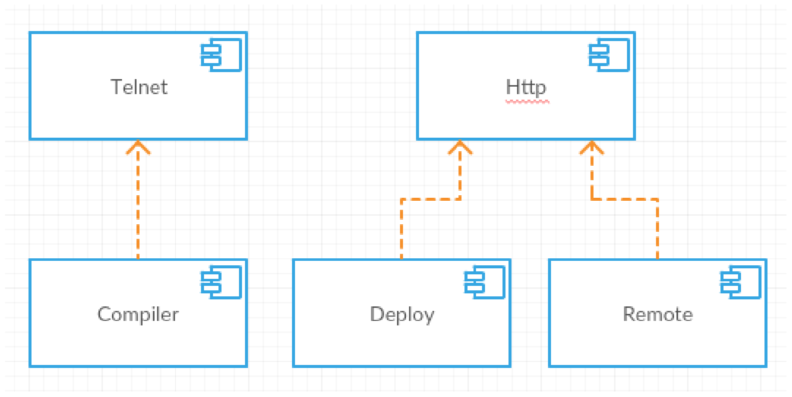
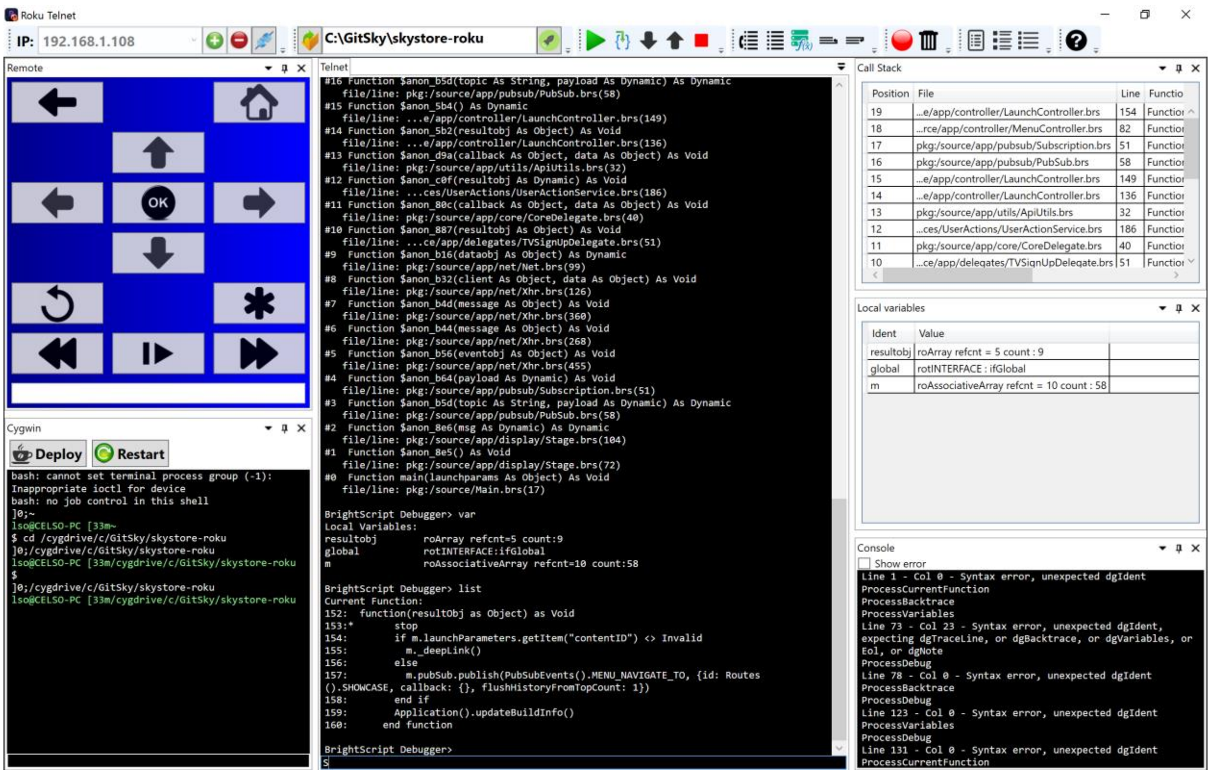


Figure 9 - Debuger diagram

Telnet uses a Compiler to parse the telnet output and generates debug context. The deploy generates the package and uploads it to the box using http. The remote uses http to emulate remote input.

The debugger will be integrated in Visual Studio Plugin, but should independent of Visual Studio SDK, for reuse purposes. To implement and test the functionalities we created an Application that can be used isolated. Figure 10 shows the debugger application.



```
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
```

```
#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
```

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 debugger diagram.

6.1. Telnet

The telnet component has two functionalities, show output of the box and send debug commands.

The box output is made using `print` or stop keywords in code files. Print writes to the output, is similar to `printf` function in C language. `stop` makes the box enter in debug mode, in this mode it allows to send the debug commands 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

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

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

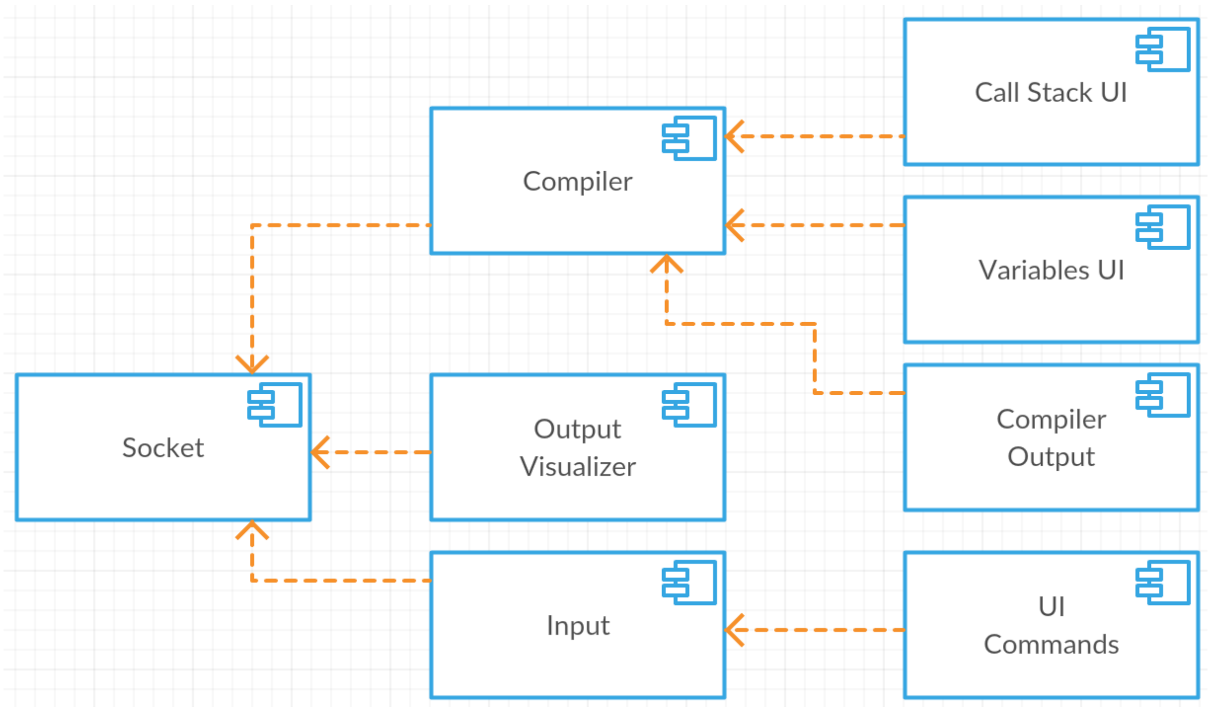
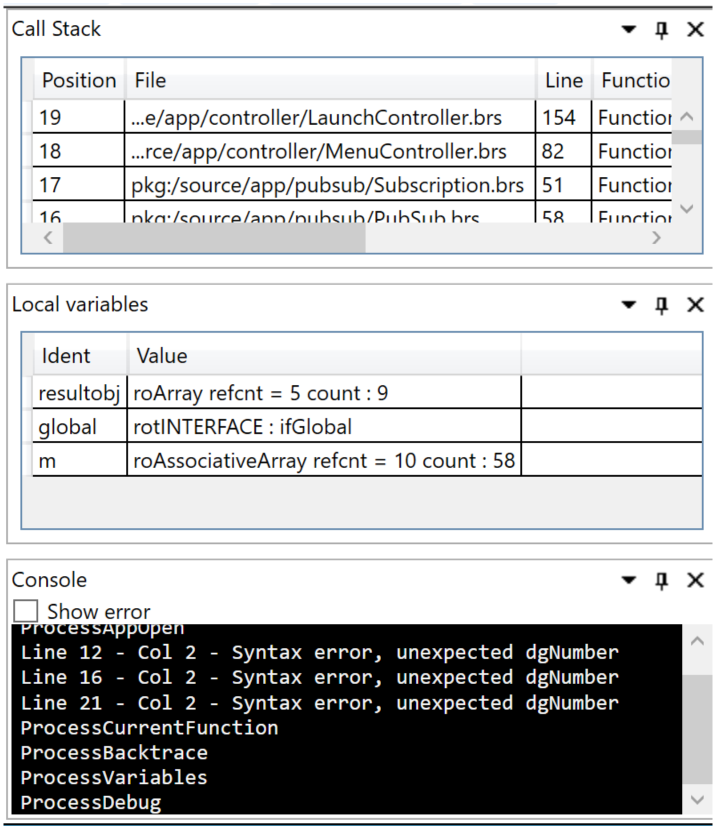


Figure 11 - Telnet component diagram

The compiler recognizes the call stack and local variables patterns and parse it, this information is shown on Call Stack and Local Variables windows. The output of compiler is shown on Console window. Figure 12 shows the compiler's output windows.


Call Stack

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

Local variables

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

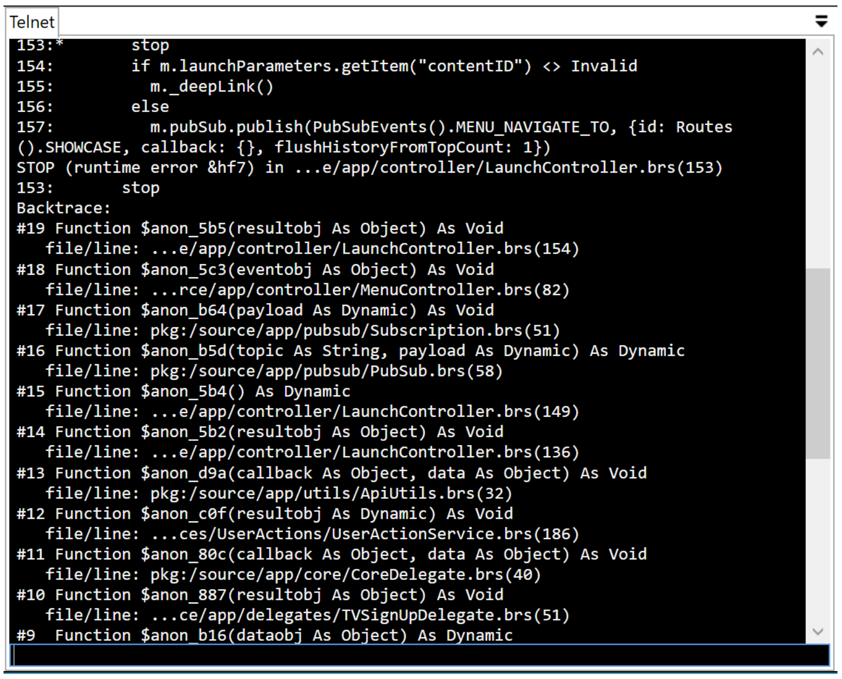
Console

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

The output visualizer uses the socket output to show the text and send debug commands. 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

The tool bar contains icons that correspond to debug commands and makes more easy to use the debugger. Figure 14 shows app tool bar.

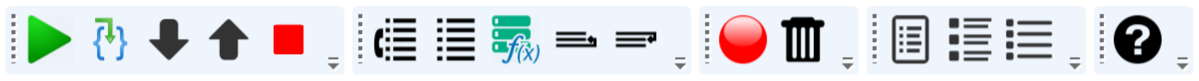


Figure 14 - UI Commands

6.2. Deploy

The deploy process consists in generate a zip file with the code files and upload it to the box, using http port.

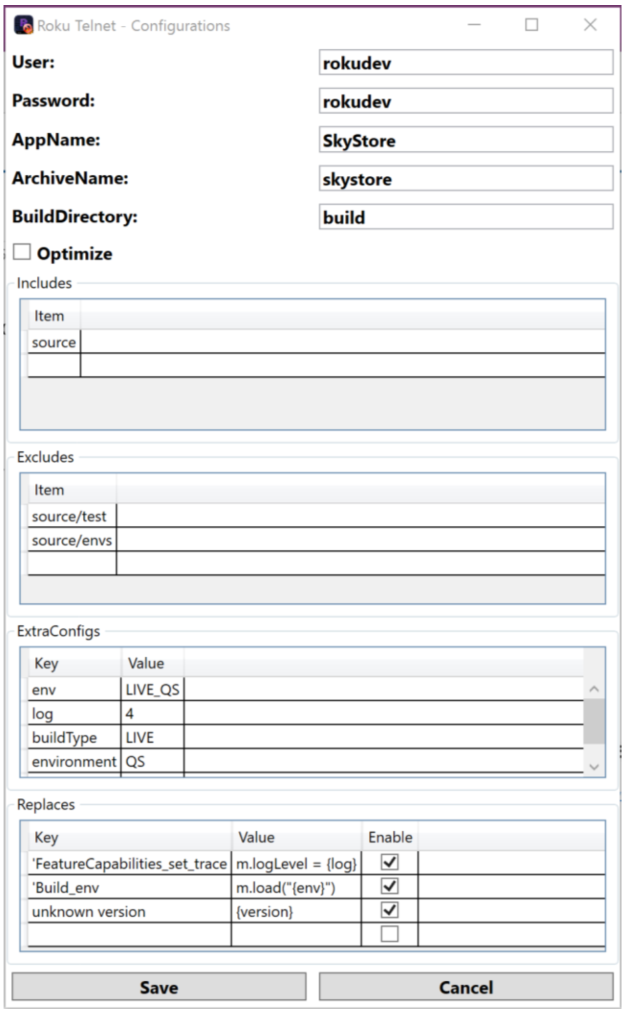
The process is configurable and should allow to:

- Select the folders to send
- Inject break points
- Remove specific parts of code
- Parameterize package, injecting code
- Execute unit tests
- Generate/Edit manifest file

The deploy needs to implement four steps:

1. Copy files to specific folder
2. Automatic edition, for configurations
3. Zip generation
4. Upload zip to the box

The process uses a several configurations, to support this was created a page for manage deploy configurations. Figure 15 shows the configuration window.



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

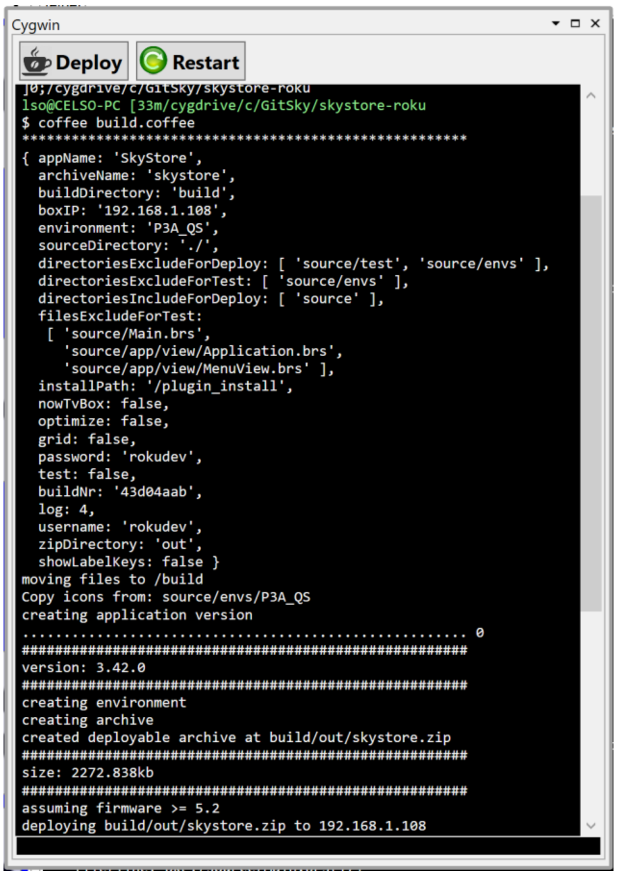
Replaces

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

Figure 15 - Configurations Window

The upload step needs the developer user and password of the box to access http port. The optimize configuration will add an extra task that removes comments, empty lines and extra spaces. The includes section is the list of sub folders to include in the zip. The exclude section is the list of sub folders to delete (for unit tests purposes). The extra configs section is for use in replaces. The replaces is to replace code on files (allows to inject build configurations, replacing code).

SkyStore App is deployed using Coffee Script, to maintain compatibility with SkyStore and other Roku apps that is deployed using make files or other command line tools, was created a graphical component that allows to use Cygwin console. Figure 16 shows the Cygwin console window.



```
]0;/cygdrive/c/GitSky/skystore-roku
lso@CELSO-PC [33m/cygdrive/c/GitSky/skystore-roku
$ coffee build.coffee
******************************************************
{ appName: 'SkyStore',
  archiveName: 'skystore',
  buildDirectory: 'build',
  boxIP: '192.168.1.108',
  environment: 'P3A_QS',
  sourceDirectory: './',
  directoriesExcludeForDeploy: [ 'source/test', 'source/envs' ],
  directoriesExcludeForTest: [ 'source/envs' ],
  directoriesIncludeForDeploy: [ 'source' ],
  filesExcludeForTest:
   [ 'source/Main.brs',
     'source/app/view/Application.brs',
     'source/app/view/MenuView.brs' ],
  installPath: '/plugin_install',
  nowTvBox: false,
  optimize: false,
  grid: false,
  password: 'rokudev',
  test: false,
  buildNr: '43d04aab',
  log: 4,
  username: 'rokudev',
  zipDirectory: 'out',
  showLabelKeys: false }
moving files to /build
Copy icons from: source/envs/P3A_QS
creating application version
...................................................... 0
######################################################
version: 3.42.0
######################################################
creating environment
creating archive
created deployable archive at build/out/skystore.zip
######################################################
size: 2272.838kb
######################################################
assuming firmware >= 5.2
deploying build/out/skystore.zip to 192.168.1.108
```

Figure 16 - Cygwin console

6.3. Remote

Roku boxes accept remote commands over http, using this feacture we build a remote component, that allows to control the box form the PC.

The remote has the corresponding buttons and a text box that allows to send text to the box. Figure 17 shows remote window.

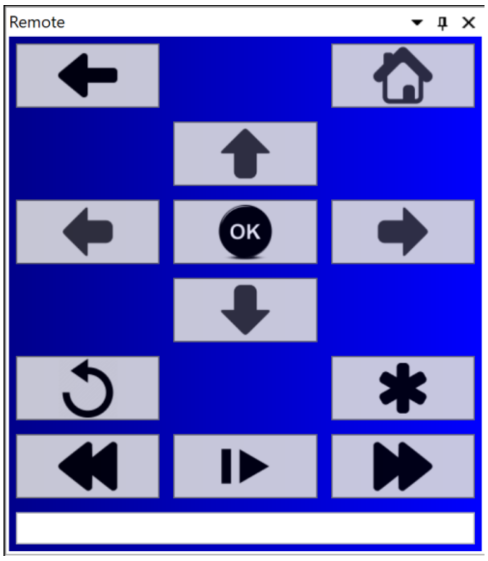


Figure 17 - Remote

7. Visual Studio Plugin

The Visual Studio has four different components, the project type, the builder/deploy, the editor extensions and the debugger.

Visual Studio has several ways of extend his behavior. We are using “MefComponent”, “VsPackage”, “ProjectTemplate” and “ItemTemplate”.

The “MefComponent” uses Mef (MEF - Managed Extensibility Framework, s.d.), MEF is a library for creating lightweight, extensible applications. It allows Visual Studio to discover and use extensions with no configuration required.

The “*VsPackage*” uses an implementation of “*Package*” to register extensions.

The “*ProjectTemplate*” exposes project templates to be used on project dialogs.

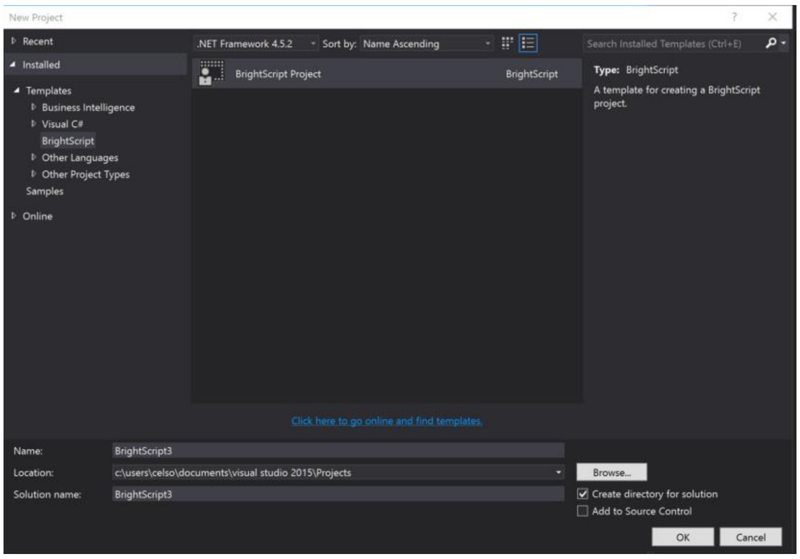
The “*ItemTemplate*” exposes templates to be used on new item dialog.

7.1. Project Type

The project type provides the templates to create the BrightScript project and for create the code files. The project type will provide the settings to show on UI and to be used by the other components.

In project type implementation we used VSProjectSystem (VSProjectSystem, s.d.), this base implementation has the most common code to create project types.

The project template defines the base “*bsproj*” that needs to be registered on plugin to appears on new project dialog. Figure 18 shows the project dialog window.



.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

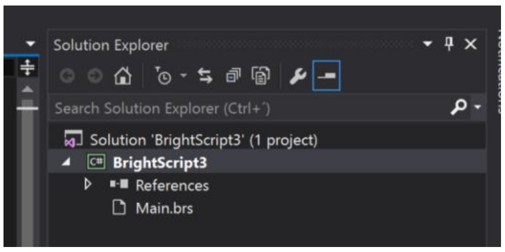
Solution name: BrightScript3

[✓] Create directory for solution

[ ] Add to Source Control

Figure 18 - Visual Studio project dialog

The project template should have the base code files to be added on project creation. We could define a project factory to inject code on templates. Figure 19 shows project structure, after being created.

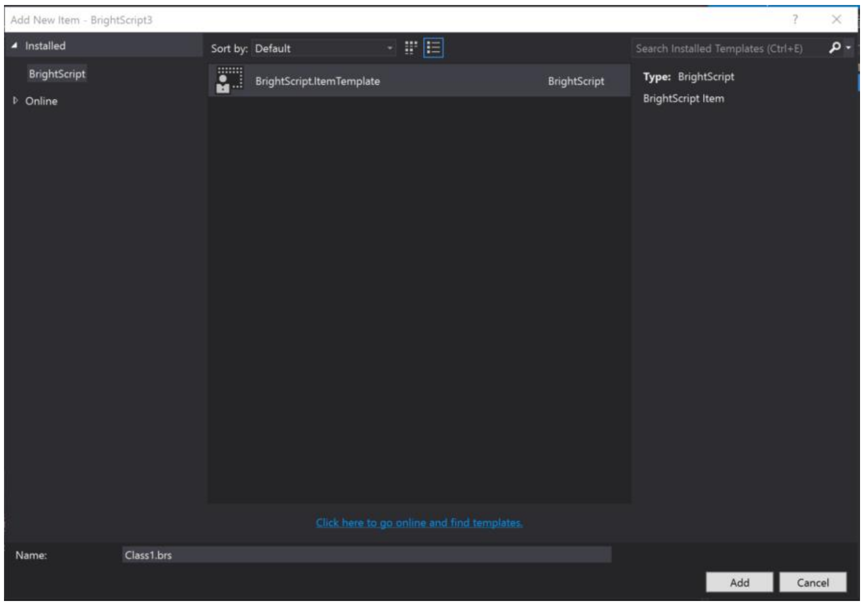


Solution 'BrightScript3' (1 project)
BrightScript3
References
Main.brs

Figure 19 - Project created

Using VSProjectSystem (VSProjectSystem, s.d.), we need to define all item templates and register them, for the code files appear in solution explorer.

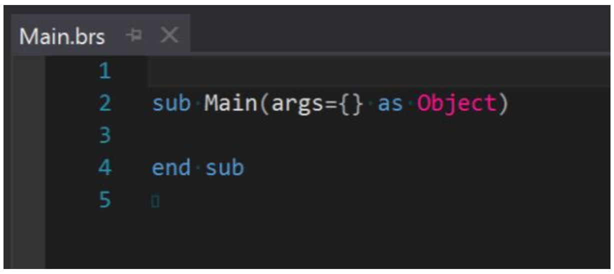
The item templates are base code files, that appears on new item dialog and has the base code structure. Figure 20 shows new item dialog.


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.
Add
Cancel

Name: Class1.brs

Figure 20 - New item dialog

Figure 21 shows create file, after being created.



```

sub Main(args={} as Object)

end sub

```

Figure 21 - Base code file

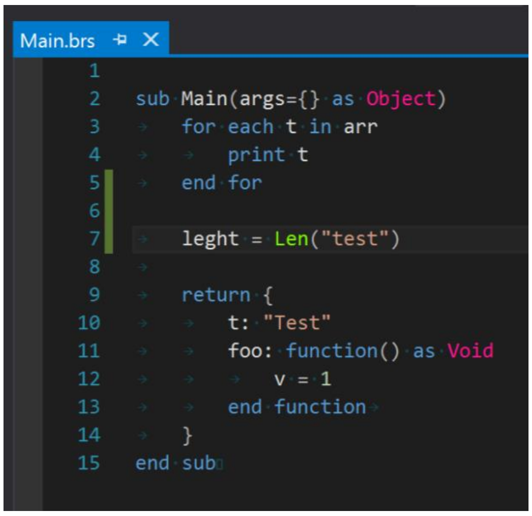
7.2. Builder/Deploy

The builder/deploy will implement the MSBuild tasks who compile all code files, prepare the package and send it to the box. This tasks will use the implementation created for debugger, to deploy the apps.

7.3. Editor Extension

The editor extensions provide syntax highlighting, compiler errors and intellisense.

The syntax highlighting is supported by the simpler Lexer, that generates tokens for syntax highlighting. To expose fuctionality we need to create and export a class that implements *"ITaggerProvider"* interface. Figure 22 shows syntax highlighting on editor.

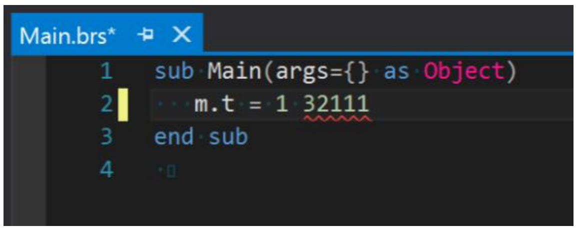


```
Main.brs

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 22 - Syntax highlighting

The compiler errors are supported by the Parser who compiles the code and generate the compiling errors. The compiling errors is shown on editor as underlining mark and are listed on error window.

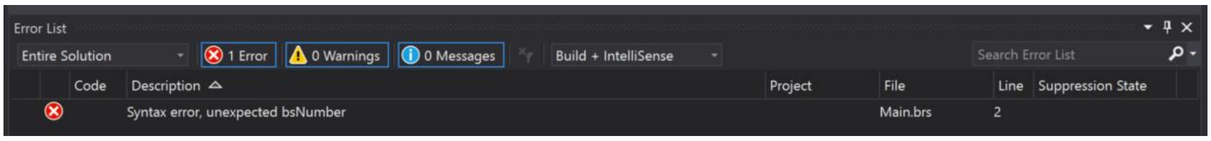
The editor errors are presented using a *"ITaggerProvider"* for *"ErrorTag"*. The Error tagger will expose the compiler errors. Figure 23 shows error underline.



```
Main.brs*

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

Figure 23 - Editor error

The error window uses an implementation of *"ErrorListProvider" class* to show the generated errors. Figure 24 shows the error on error window.



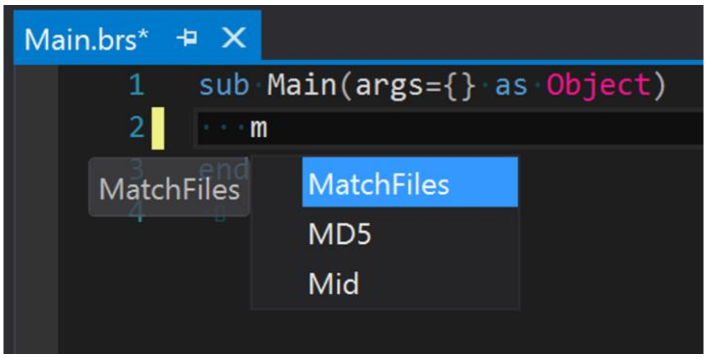
Error List

Entire Solution | 1 Error | 0 Warnings | 0 Messages | Build + IntelliSense | Search Error List

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

Figure 24 - Error window

The intellisense uses the Parser to generate the AST and needs to implement an instance of *"ICompletionSourceProvider"* interface to generate the list of suggestions. We need to implement *"IVsTextViewCreationListener"* to register the command handler to show the list. Figure 25 shows intellisense popup.



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

Figure 25 - intellisense

Visual Studio generates events on every file change, this would make the code to be compile in different points for the same code. To avoid this, we cache the result of the compilation.

7.4. Debugger integration

The debugger will use the implementation developed for Debugger App that manage the connection with the box and generates debug output. The debugger output and debugger commands will be integrated with Visual Studio debugger extension.

8. Conclusion

The implementation of this project was very useful for understand the way that compilers work and how it cloud be used in tools to make development more easy.

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 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. The all project will be much more useful.

The project is on an incomplete stage, it remains to implement the following features:

- Parser
    - Generate AST (terminating)
- Visual Studio
    - Build/Deploy tasks
    - Editor Extensions
        - Only use AST on intellisense
    - Debugger integration

9. References

(n.d.). Retrieved from Irony - .Net Language Implementation Kit: https://irony.codeplex.com/

(n.d.). Retrieved from JavaCC: https://javacc.java.net/

(n.d.). Retrieved from SableCC: http://www.sablecc.org/

(n.d.). Retrieved from YACC: http://dinosaur.compilertools.net/#yacc

(n.d.). Retrieved from Lex: http://dinosaur.compilertools.net/#lex

Aiken, A. (n.d.). *Compilers Theory*. Retrieved from https://www.youtube.com/playlist?list=PLLH73N9cB21VSVEX1aSRlNTufaLK1dTAI

Appel, A. W. (2002). *Modern Compiler Implementation in Java.* Cambrige University Press.

BrightScript. (n.d.). Retrieved from Roku SDK: https://sdkdocs.roku.com/display/sdkdoc/Roku+SDK+Documentation

GPlex. (n.d.). Retrieved from http://gplex.codeplex.com/

Gppg. (n.d.). Retrieved from https://gppg.codeplex.com/

MEF - Managed Extensibility Framework. (n.d.). Retrieved from https://msdn.microsoft.com/en-us/library/dd460648(v=vs.110).aspx

PowerShell Tools. (n.d.). Retrieved from https://github.com/adamdriscoll/poshtools

Python Tools. (n.d.). Retrieved from https://github.com/Microsoft/PTVS

Visual Studio Extension for Lua. (n.d.). Retrieved from https://github.com/Microsoft/VSLua

VSProjectSystem. (n.d.). Retrieved from GitHub: https://github.com/Microsoft/VSProjectSystem/

10. Figures Index

Figure 1 - Component diagram 2
Figure 2 - Phases of a compiler, and interfaces between them 3
Figure 3 - Combined finite automaton 5
Figure 4 - LR(0) states for Grammar 6
Figure 5 - Abstract Syntax Tree Sample 7
Figure 6 - Compiler Components 11
Figure 7 - FSA - Finite State Automata 12
Figure 8 - Shift reduce table 13
Figure 9 - Debuger diagram 14
Figure 10 - Debugger Application 15
Figure 11 - Telnet component diagram 17
Figure 12 - Compiler output windows 18
Figure 13 - Output visualizer 18
Figure 14 - UI Commands 19
Figure 15 - Configurations Window 20
Figure 16 - Cygwin console 21
Figure 17 - Remote 22
Figure 18 - Visual Studio project dialog 24
Figure 19 - Project created 24
Figure 20 - New item dialog 25
Figure 21 - Base code file 25
Figure 22 - Syntax highlighting 26
Figure 23 - Editor error 26
Figure 24 - Error window 26
Figure 25 - intellisense 27