

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

# 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, I 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.

This project is to implement an integrated tool for application development who makes more easy application development and debugging. This tool will support 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 complemented with a box simulator to run the applications on development machine, this will be an optional implementation and could be a great benefit.

The idea of this project comes from my participation on Sky Store Roku App development, for Sky UK Limited company.

# Index

[1. Resume i](#_Toc462698248)

[2. Index ii](#_Toc462698249)

[3. Introduction 1](#_Toc462698250)

[3.1. Compilers Theory 1](#_Toc462698251)

[3.2. Code Generation tools 1](#_Toc462698252)

[3.3. Implementation 2](#_Toc462698253)

[4. Compiler 4](#_Toc462698254)

[4.1. Lexer 4](#_Toc462698255)

[4.2. Parser 5](#_Toc462698256)

[5. Debugger 7](#_Toc462698257)

[5.1. Deploy 7](#_Toc462698258)

[5.2. Telnet 10](#_Toc462698259)

[5.3. Http 13](#_Toc462698260)

[6. Visual Studio Plugin 15](#_Toc462698261)

[6.1. Project Type 15](#_Toc462698262)

[6.2. Builder/Deploy 17](#_Toc462698263)

[6.3. Editor Extension 17](#_Toc462698264)

[6.4. Debugger integration 19](#_Toc462698265)

[7. Conclusion 20](#_Toc462698266)

[8. Bibliography 21](#_Toc462698267)

[9. Figures Index 22](#_Toc462698268)

# Introduction

The project is divided into three stages. First stage is investigation on compilers theory. The second stage is investigation on existing tools for generate compilers code, how it works and his benefits. The third stage is the plugin implementation.

## Compilers Theory

In first stage it was used the Compilers Theory videos (Aiken, s.d.), realized by Alex Aiken who is a professor of computer science in Stanford university and Modern Compiler Implementation (Appel, 2002) in Java, recommended by Advisor. This two sources have a very similar approach of compilers theory, suggesting a modular implementation.

The BrightScript is an interpreted language, according to this we don’t need to implement all compilation steps. The tool just has to implement the Lexer and Parser steps. If we implement the simulator we need to implement all compilation steps and we cloud generate MSIL (Microsoft Intermediate Language).

The Lexer read the code file and generate a list of tokens, also known as tokenizer.

The Parser receives the list of tokens, makes syntax validation and generate the abstract syntax tree.

## Code Generation tools

In second stage it was analyzed the use of tools for generate Lexer and Parser and they make much more easy to generate and maintain the repetitive code of Lexer and Parser.

The Visual Studio plugins needs to be written in C# or Visual Basic, according to this we select GPlex and Gppg code generation tools.

GPlex (GPlex, s.d.) is a Lexer generator, it generates a Lexer implementation in C#, based on specification file similar to Lex specification. The generated Lexer is based on finite state autómata algorithm.

Gppg (Gppg, s.d.) is a Parser generator, it generates a Parser with bottom-up approach, based on specification file similar to YACC specification. The Parser recognizes languages LALR(1) ( 1 Look-Ahead token, Left-to-Right - right most derivation).

The code generators were designed to work together but they can be used isolated, on implementation the generated Parser will use the generated Lexer to get the tokens. They were designed to integrate with Visual Studio giving some options to generate code for integration.

In addition to the code generators, was analyzed the Visual Studio SDK, for language and debug extensions. There’s a large and a bit confusing documentation. The implementation will be based on three samples (Python Tools, s.d.), (Visual Studio Extension for Lua, s.d.), (PowerShell Tools, s.d.).

## Implementation

The third stage is the plugin implementation. The plugin is divided in three components, illustrated on following diagram.



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 and a web page for deploy the apps and a telnet port to receive box output and send debug commands.

The plugin will be based on 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.

# Compiler

The Compiler is composed by three components, the Lexer (also called Scanner), the Parser and Visual Studio integration component. The purpose of the first two components is to process code files generating compilation errors and build the abstract syntax tree for intelisense functionality.



Figure 2 - Compiler Components

## Lexer

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

The lexical define the format of the tokens of the language, the format is defined by regular expressions. Each token corresponds to a regular expression.

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. The code is generated based in three sources, a base class with generic implementation, the definition file and the decoders/read buffer. GPlex reads the definition file (\*.lex) with the regular expressions and generates the finite state automata (FSA) tables.



Figure 3 - FSA - Finite State Autómata

There were generated two Lexers one most simpler for syntax highlighting and another most complex for use with the Parser. The first one will be used without Paser, we only need to generate the tokens.

## Parser

The Parser has the purpose of analyze the grammatical structure of the language. It validates the sentences, if the tokens are in the right order. This analyzes allow the Paser to structure the code in a tokens tree. That tree, the abstract syntax tree (AST), is the output of the Parser.

The grammatical structure is defined by a set of rules, defined in the YACC file, this rules define the tokens order.

The Paser is generated by Gppg, that generates a C# file with the implementation of the Parser. The generated code implements the Shift-Reduce algorithm and generates the AST.



Figure 4 - Shift reduce table



Figure 5 - Abstract Syntax Tree Sample

# Debugger

The Debugger is a tool that allows to deploy the app to the box and manage the interface between the telnet port and the Visual Studio. The box eposes a http port that allows to simulate the remote input.

It was created one application for test purposes, this uses all debugger functionalities.



Figure 6 - Debugger Application

The app divided in three components who provide different functionalities. The deploy generate the package and send it to the box. The telnet manages the telnet ports communication allowing to send debug commands and to receive debug output, the debug output is parsed to get current variables value and call stack. The http allows to send remote commands and capture screen.

## 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 allows:

* 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 is composed in four steps:

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

The process need a several configurations and it was created a page for manage this.



Figure 7 - Configurations Window

The upload needs the user and password for access box http port. The optimize will remove comments, empty lines and extra spaces. The includes is the sub folders to include in the zip. The exclude is the sub folders to delete (for unit tests). The extra configs is to use in replaces. The replaces is to replace code on files (allows to inject build configurations).

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.

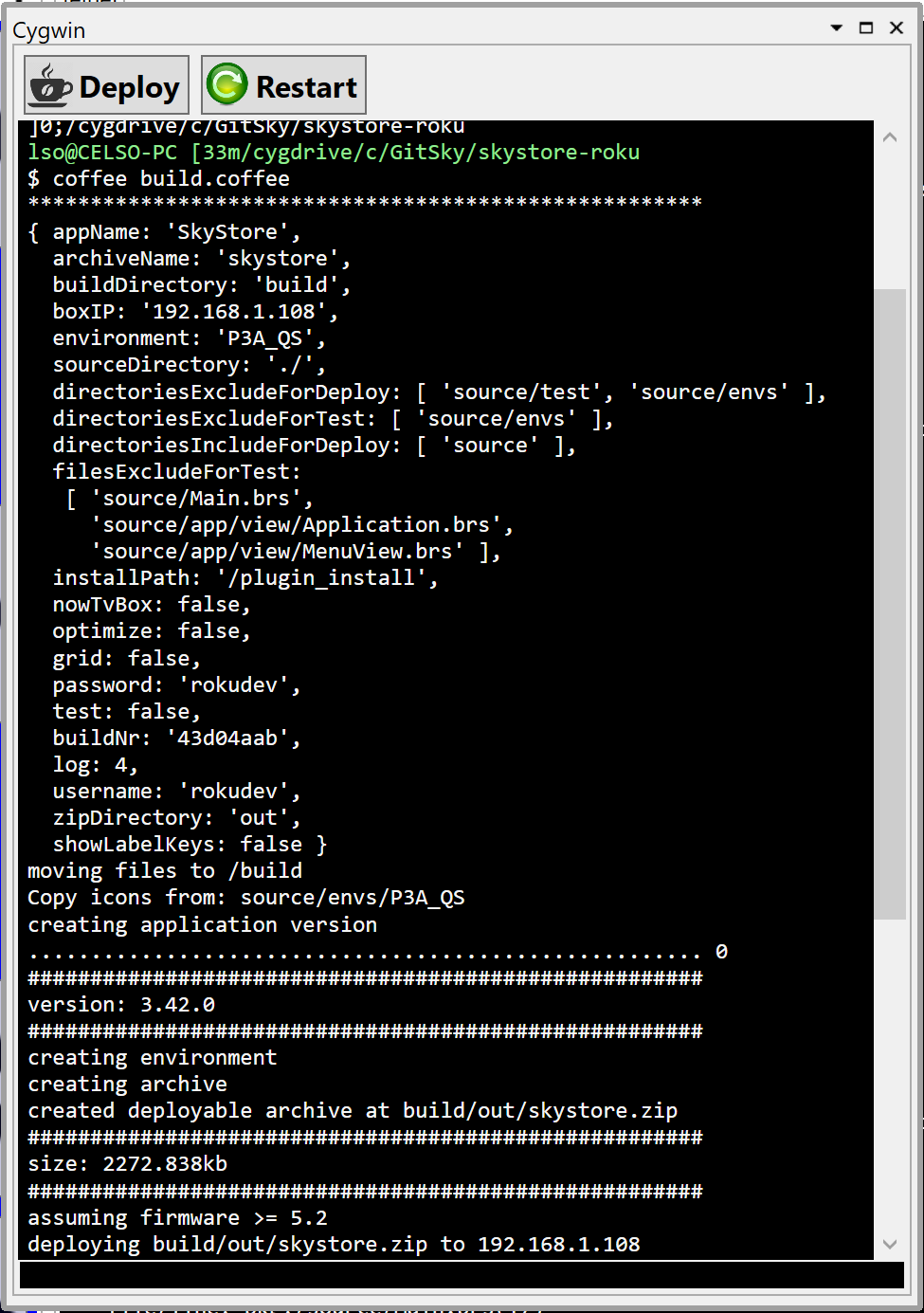


Figure 8 - Cygwin console

## Telnet

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

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

|  |  |
| --- | --- |
| **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 |

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 for debugger output. Like shown in following diagram.



Figure 9 - Telnet component diagram

The compiler recognizes the call stack and local variables patterns and parse it to show on following tables. The compiler output is shown too.



Figura 1 - UI do compilador

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



Figure 10 - Output visualizer

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



Figure 11 - UI Commands

## Http

Using http remote simulation, 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 12 - Remote

# 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 a project templates to be used on project dialogs

## 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, 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 13 - Visual Studio project dialog

The project template should have the base code files to be created on project creation. We could define a project factory to inject code on templates.



Figure 14- Project created

By using (VSProjectSystem, s.d.), we need to define all item templates and register them, to be shown on solution explorer.

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



Figure 15 - New item dialog



Figure 16 - Base code file

## 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 same code used on the debugger app for deploy task.

## Editor Extension

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

For syntax highlighting we use the most simpler Lexer, that generates tokens for syntax highlighting. In Visual Studio we need to create and export a class that implements *ITaggerProvider* interface.



Figure 17 – Syntax highlighting

For compiler errors we use the Parser to compile the code and generate the compiling errors. The compiling errors is shown on editor as underlining mark and are listed on error window.



Figure 18 - Editor error



Figure 19 - Error window

For intelisense we use the Parser to generate the AST and implement “*ICompletionSourceProvider*” to generate the list of sugestions. We need to implement to implement “*IVsTextViewCreationListener*” to register the command handler to show the list.



Figure 20 – Intelisense

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.

## Debugger integration

The debugger will use the telnet output compiler to receive box debugger state and use the telnet port to send debug commands to the box.

# 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 I didn’t have any knowledge of how compilers work or 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.

I’m using the debugger app on day to day work and it makes much more easy to debug the app I’m developing. 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 intelisense
  + Debugger integration

# Bibliography

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.

*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/

# Figures Index

[Figure 1 - Component diagram 2](#_Toc462698198)

[Figure 2 - Compiler Components 4](#_Toc462698199)

[Figure 3 - FSA - Finite State Autómata 5](#_Toc462698200)

[Figure 4 - Shift reduce table 6](#_Toc462698201)

[Figure 5 - Abstract Syntax Tree Sample 6](#_Toc462698202)

[Figure 6 - Debugger Application 7](#_Toc462698203)

[Figure 7 - Configurations Window 9](#_Toc462698204)

[Figure 8 - Cygwin console 10](#_Toc462698205)

[Figure 9 - Telnet component diagram 12](#_Toc462698206)

[Figure 10 - Output visualizer 13](#_Toc462698207)

[Figure 11 - UI Commands 13](#_Toc462698208)

[Figure 12 - Remote 14](#_Toc462698209)

[Figure 13 - Visual Studio project dialog 16](#_Toc462698210)

[Figure 14- Project created 16](#_Toc462698211)

[Figure 15 - New item dialog 17](#_Toc462698212)

[Figure 16 - Base code file 17](#_Toc462698213)

[Figure 17 – Syntax highlighting 18](#_Toc462698214)

[Figure 18 - Editor error 18](#_Toc462698215)

[Figure 19 - Error window 18](#_Toc462698216)

[Figure 20 – Intelisense 19](#_Toc462698217)