

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](#_Toc462095864)

[2. Index ii](#_Toc462095865)

[3. Introduction 1](#_Toc462095866)

[3.1. Compilers Theory 1](#_Toc462095867)

[3.2. Code Generation tools 1](#_Toc462095868)

[3.3. Implementation 2](#_Toc462095869)

[4. Compilador 4](#_Toc462095870)

[4.1. Scanner 4](#_Toc462095871)

[4.2. Parser 4](#_Toc462095872)

[4.3. Plugin Visual Studio 5](#_Toc462095873)

[5. Debugger 6](#_Toc462095874)

[5.1. Deploy 6](#_Toc462095875)

[5.2. Telnet 7](#_Toc462095876)

[5.3. Remote Http 10](#_Toc462095877)

[5.4. Visual Studio 11](#_Toc462095878)

[6. Bibliography 12](#_Toc462095879)

[7. Figures Index 13](#_Toc462095880)

# 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.

# Compilador

O compilador é composto por três componentes, o analisador léxico (scanner), o Parser e a componente de integração com o Visual Studio. O objetivo das duas primeiras componentes é processar os ficheiros de código de forma a detetar erros de compilação e disponibilizar dados para o intellisense.

## Scanner

O analisador léxico tem como responsabilidade gerar tokens para serem usados no Parser e validar se o código obedece ao léxico definido para a linguagem.

O léxico define o formato dos tokens que compõem a linguagem, o formato é definido através dum conjunto de regras e as regras são definidas através de expressões regulares.

Um analisador léxico é uma máquina de estados que tenta encontrar os tokens com as maiores dimensões. O processamento é feito carater a carater, enquanto existir possibilidade de encontrar um token maior, ao encontrar o token a string é retirada do input e é emitido o token.

O Scanner é gerado utilizando o Gplex, que gera uma classe em C# com a implementação. O código gerado é obtido de três fontes, da estrutura base da classe, motor de reconhecimento das patterns e os decoders/buffer de leitura. O motor de reconhecimento é composto por tabelas que definem a máquina de estados (FSA - finite state automaton). Estas tabelas são geradas através do ficheiro de especificação (\*.lex).

Foram gerados dois scanners, um mais simples para sintax highlighting com um conjunto reduzido de tokens e outro mais completo para usar com o Parser.

## Parser

O Parser tem como objetivo analisar a estrutura gramatical da linguagem, as frases. Valida se a ordem pela qual se apresentam os tokens é valida. Essa análise permite estruturar o código numa arvore de tokens. O output do Parser é uma árvore abstrata da estrutura do código (AST - Abstract Syntax Tree).

A gramatica é constituía por um conjunto de regras, que determinam a ordem dos símbolos nas sequências validas.

O Parser é gerado utilizando o Gppg, que gera uma classe em C# com a implementação. O código gerado implementa o algoritmo Shift-Reduce para gerar a AST (Abstract Syntax Tree).

O Parser implementado está a fazer a validação gramatical, mas ainda não está a gerar a AST, será implementado na segunda fase do projeto.

## Plugin Visual Studio

O Visual Studio utiliza o Lexer para atribuir diferentes cores ao tokens e o Parser para carregar os dados para o Intellisense. São ainda usados os dois componentes para validar o código, evitando os erros de compilação na box.

Para além destas funcionalidades será criado um novo tipo de projeto para gerir as configurações e associar os templates para criação de novos ficheiros de código e outros tipos de ficheiros.

A implementação do plugin será efetuada na segunda fase do projeto. Para simular as chamadas ao compilador foi criado um projeto para testar o processo de compilação. Esse projeto cria um scanner e um parser para compilar um ficheiro especifico.

# Debugger

O debuger é uma tool que permite fazer deploy da aplicação para as boxs e fazer o interface entre o Visual Studio e o porto Tenet da box. Para além do porto Telnet a box disponibiliza um porto http, que permite simular ações do comando e iniciar aplicações.



Figura 2 - Aplicação Debug

## Deploy

O deploy consiste em gerar um zip com todos os ficheiros da aplicação e fazer o upload para a box, utilizando o porto http.

O processo de deploy deve ser configurável, permitindo:

* Selecionar as pastas a enviar
* Injetar break points
* Eliminar troços de código utilizados só para debug
* Passar parâmetros para a aplicação, através de injeção de código
* Executar testes unitários
* Gerar/Actualizar o ficheiro manifest

O deploy consiste em quatro passos:

1. Copiar ficheiros
2. Edição automatizada
3. Geração do ficheiro zip
4. Upload do ficheiro zip para a box

O processo de deploy será implementado na segunda fase do projeto, para efetuar o deploy são utilizados os scripts de coffee existentes no código da aplicação. Para facilitar a integração foi criado um componente gráfico que permite executar os scripts, recorrendo ao Cygwin.

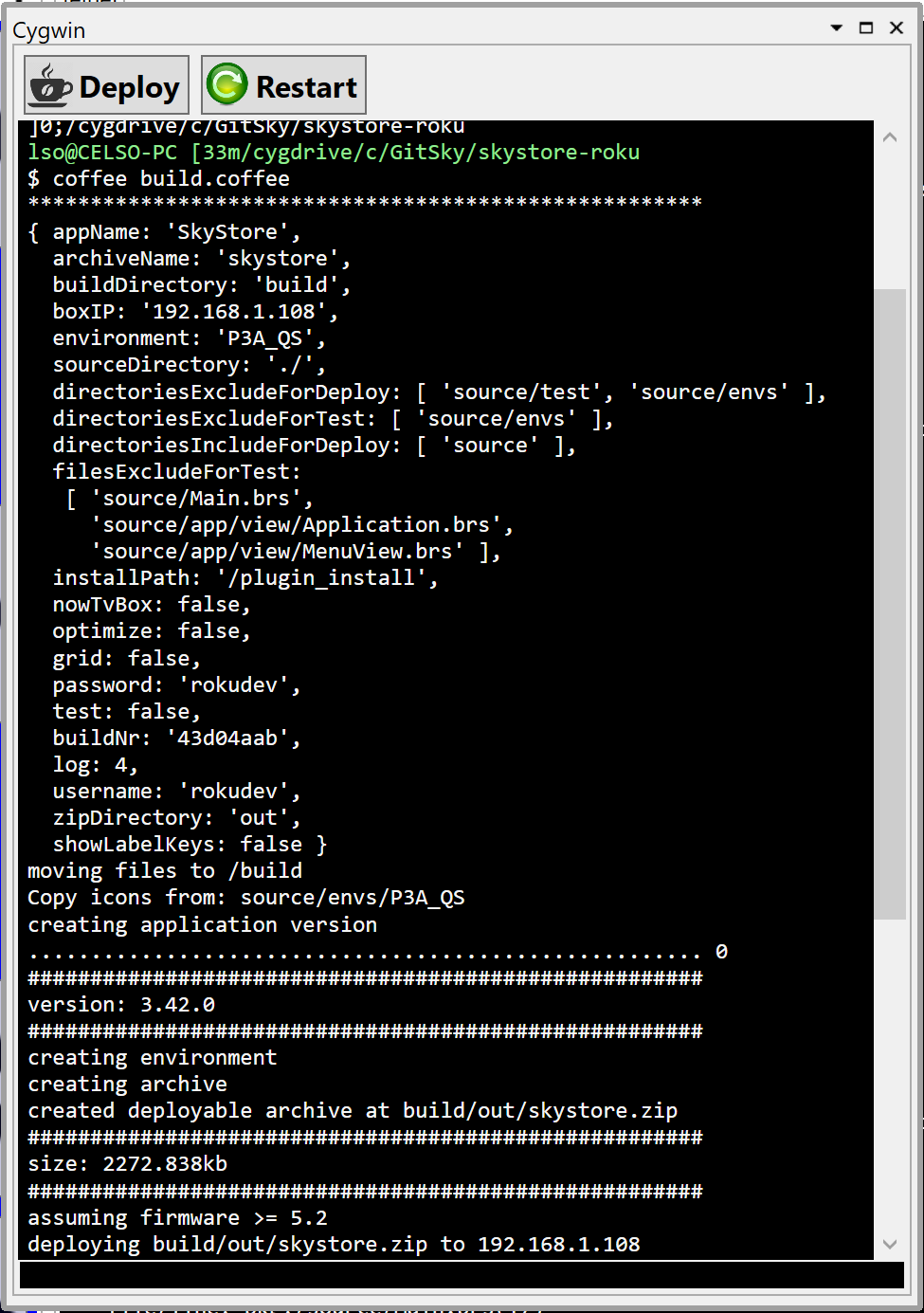


Figura 3 - Cliente Cygwin

## Telnet

O porto Telnet tem duas funcionalidades, mostrar o output da execução da aplicação e servir de terminal de debug.

O output é feito através da utilização da função “print” no código.

Para utilizar o terminal de debug utiliza-se a função “stop” no código. Quando a função for executada a aplicação para e é possível interagir com o debuger utilizando os seguintes comandos:

|  |  |
| --- | --- |
| **Comando** | **Descrição** |
| **bsc** | Mostra as instancias correntes |
| **bscs** | Mostra o sumario das instancias correntes |
| **brkd** | Para execução com mensagens não fatais |
| **bt** | Mostra o callstack |
| **classes** | Mostra as classes |
| **cont**or**c** | Continua execução do Script |
| **down** or **d** | Mover execução para baixo |
| **exit** | Sair |
| **gc** | Correr o garbage collector |
| **help** | Mostra a lista de comandos do sistema |
| **last** | Mostra a ultima linha executada |
| **list** | Mostra a função corrente |
| **next** | Mostra a proxima linha |
| **print, p,**or**?** | Imprime para o output |
| **step, s, or t** | Executa uma instrução |
| **up** or **u** | Mover execução para cima |
| **var** | Mostra variaveis locais e os seus tipos |
| Any Brightscript statement | Executar BrightScript |

Utilizando o debugger é possível visualizar o valor corrente das variáveis, verificar em que ficheiro e em que linha está a execução e controlar a execução do código.

A implementação do componente de telnet utiliza um socket para se ligar ao porto telnet, um compilador para tratar o output. Para facilitar a interface visual, foram criados alguns componentes visuais, como mostra o diagrama seguinte.



Figura 4 - Diagrama de componentes do telnet

O compilador identifica os padrões do output para reconhecer o call stack e o valor das variáveis locais. O Compiler Output mostra o output do compilador.



Figura 5 - UI do compilador

O output visualizer e o input usam diretamente o socket para enviar comandos e mostrar o output da box.



Figura 6 - Output visualizer

O UI commands é um conjunto de botões que correspondem aos disponibilizados para debug.



Figura 7 - UI Commands

## Remote Http

Utilizando o porto http foi implementado um comando para controlar a box através do PC.

O comando é composto pelos botões do comando e por uma textbox que permite a introdução de texto.



Figura 8 - Remote

## Visual Studio

A componente de integração do debugger com o Visual Studio, vai permitir utilizar a interface visual de debug de forma semelhante às aplicações .Net. Permitindo definir breakpoint’s, controlar a execução do código e visualizar o valor corrente das variáveis. O ponto de execução vai ser visualizado nos ficheiros de código através da funcionalidade de consultar o ponto de execução.

# 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/

*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

# Figures Index

[Figure 1 – Component diagram 2](#_Toc462095850)

[Figura 2 - Aplicação Debug 6](#_Toc462095851)

[Figura 3 - Cliente Cygwin 7](#_Toc462095852)

[Figura 4 - Diagrama de componentes do telnet 9](#_Toc462095853)

[Figura 5 - UI do compilador 9](#_Toc462095854)

[Figura 6 - Output visualizer 10](#_Toc462095855)

[Figura 7 - UI Commands 10](#_Toc462095856)

[Figura 8 - Remote 11](#_Toc462095857)