

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, 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.

# Index

[Resume i](#_Toc462957745)

[1. Index ii](#_Toc462957746)

[2. Introduction 1](#_Toc462957747)

[2.1. Compilers Theory 1](#_Toc462957748)

[2.2. Code Generation tools 1](#_Toc462957749)

[2.3. Implementation 2](#_Toc462957750)

[3. Compilers Theory 3](#_Toc462957751)

[3.1. Lexical Analysis 4](#_Toc462957752)

[3.2. Parsing 5](#_Toc462957753)

[3.3. Abstract Syntax 6](#_Toc462957754)

[4. Code Generation tools 8](#_Toc462957755)

[4.1. GPlex 8](#_Toc462957756)

[4.2. Gppg 9](#_Toc462957757)

[4.3. Irony 9](#_Toc462957758)

[4.4. JavaCC 10](#_Toc462957759)

[4.5. SableCC 10](#_Toc462957760)

[4.6. Conclusion 10](#_Toc462957761)

[5. Compiler 11](#_Toc462957762)

[5.1. Lexer 11](#_Toc462957763)

[5.2. Parser 12](#_Toc462957764)

[6. Debugger 14](#_Toc462957765)

[6.1. Deploy 14](#_Toc462957766)

[6.2. Telnet 17](#_Toc462957767)

[6.3. Http 20](#_Toc462957768)

[7. Visual Studio Plugin 22](#_Toc462957769)

[7.1. Project Type 22](#_Toc462957770)

[7.2. Builder/Deploy 24](#_Toc462957771)

[7.3. Editor Extension 24](#_Toc462957772)

[7.4. Debugger integration 26](#_Toc462957773)

[8. Conclusion 27](#_Toc462957774)

[9. References 28](#_Toc462957775)

[10. Figures Index 29](#_Toc462957776)

# 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.

## 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.

## Code Generation tools

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

## 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 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.

# 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 the models and interfaces.



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

Each model corresponds to a phase.

|  |  |
| --- | --- |
| **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. |

## 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.

The lexical analyzer generates a list of token, to be consumed by the Parser.

## 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 for a grammar.

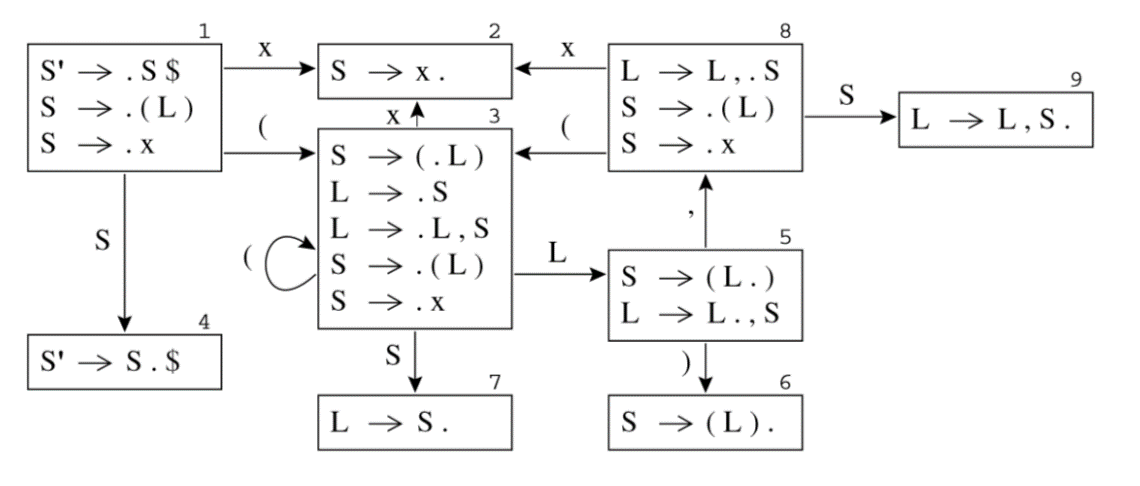


Figure 4 - LR(0) states for Grammar

The parser generates the Abstract Syntax Tree.

## Abstract Syntax

Abstract Syntax is disassociated from any speciﬁc instance.

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. Figure 5 shows a syntax tree sample.



Figure 5 - Abstract Syntax Tree Sample

# 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.

## 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).

Gardens Point LEX (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.

## 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#.

## 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.

## JavaCC

Java Compiler Compiler tm (JavaCC tm) 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.

## 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.

## Conclusion

After analyzing this tools and the Visual Studio SDK, we conclude the following:

* 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

The Gplex and Gppg can be used without any problem, according to this we will use them.

# Compiler

The Compiler is composed by two components, the Lexer (also called Scanner) and the. 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 compiler components diagram.



Figure 6 - 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 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 7 shows a FSA sample.



Figure 7 - FSA - Finite State Automata

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 Parser, 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 Parser 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 Parser 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 8 shows a shift reduce sample table.



Figure 8 - Shift reduce table

# 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.

Was created one application for test purposes, that implements all debugger functionalities. All functionalities will be reused on Visual Studio plugin. Figure 9 shows the debugger application.



Figure 9 - 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 10 shows the configuration window.



Figure 10 - 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 11 shows the Cygwin console window.

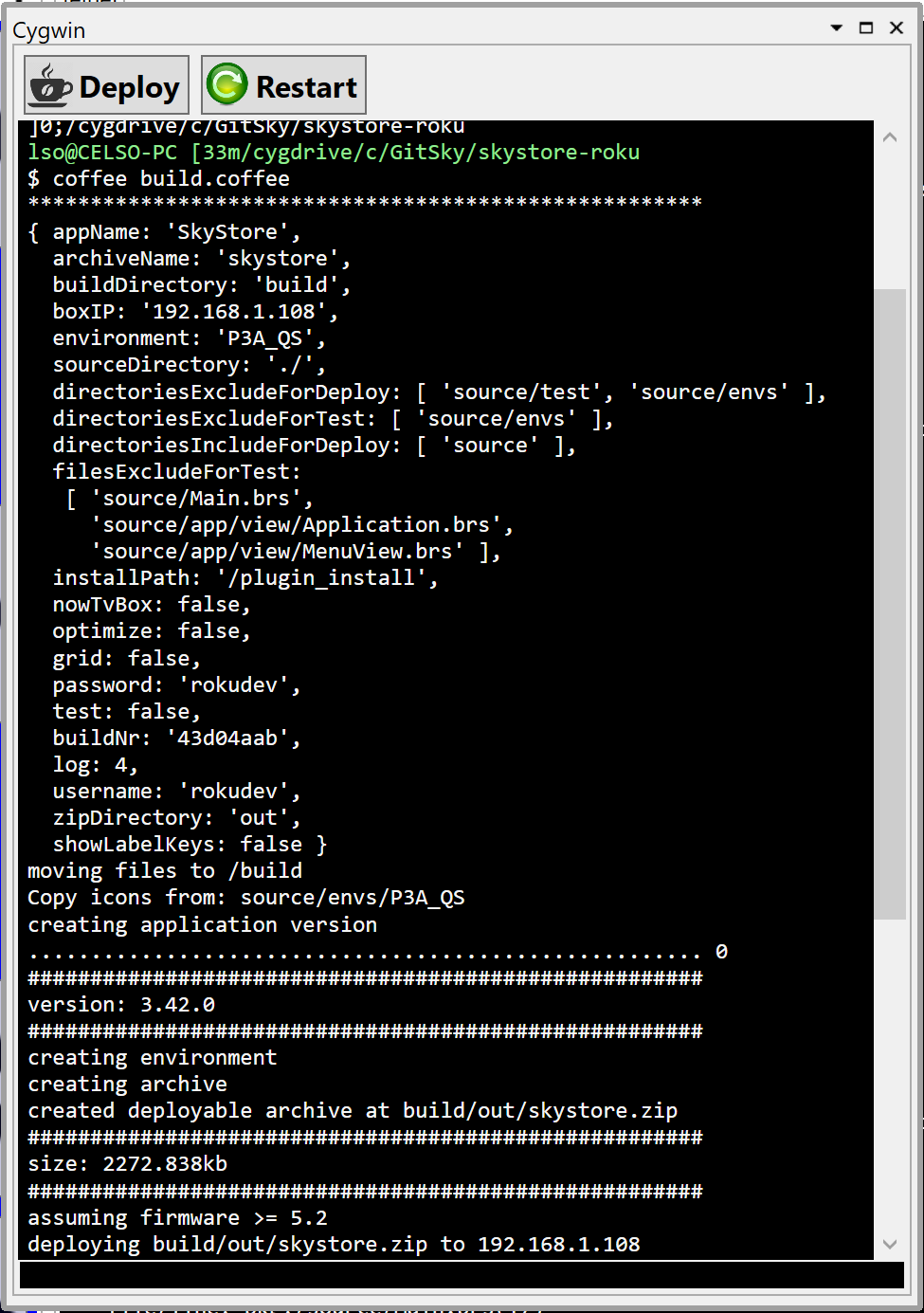


Figure 11 - Cygwin console

## 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. Figure 12 shows the telnet component diagram.



Figure 12 - 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. Figure 13 shows the compiler output windows.



Figure 13 - Compiler output windows

The output visualizer uses the socket output to show the text and send debug commands. Figure 14 show telnet output visualizer.



Figure 14 - Output visualizer

The tool bar contains icons that correspond to debug commands and makes more easy to use the debugger. Figure 15 shows app tool bar.



Figure 15 - 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 16 shows remote window.



Figure 16 - 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.

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

## 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 17 shows the project dialog window.



Figure 17 - 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 18 shows project structure.



Figure 18 - 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 19 shows new item dialog.



Figure 19 - New item dialog

Figure 20 shows create file.



Figure 20 - 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 21 shows syntax highlighting on edito.



Figure 21 - 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.

The editor errors are presented using a “*ITaggerProvider”* for “*ErrorTag”*. The Error tagger will expose the compiler errors. Figure 22 shows error underline.



Figure 22 - Editor error

To present errors on error window we need to create a “*ErrorListProvider”* to exposes the generated errors. Figure 23 shows the error on error window.



Figure 23 - Error window

For intellisense we use the Parser to generate the AST and implement “*ICompletionSourceProvider*” to generate the list of suggestions. We need to implement “*IVsTextViewCreationListener*” to register the command handler to show the list. Figure 24 shows intellisense popup.



Figure 24 - 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.

## Debugger integration

The debugger will use the code 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.

# 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.

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 intellisense
  + Debugger integration

# 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/

# Figures Index

[Figure 1 - Component diagram 2](#_Toc462957721)

[Figure 2 - Phases of a compiler, and interfaces between them 3](#_Toc462957722)

[Figure 3 - Combined ﬁnite automaton 5](#_Toc462957723)

[Figure 4 - LR(0) states for Grammar 6](#_Toc462957724)

[Figure 5 - Abstract Syntax Tree Sample 7](#_Toc462957725)

[Figure 6 - Compiler Components 11](#_Toc462957726)

[Figure 7 - FSA - Finite State Automata 12](#_Toc462957727)

[Figure 8 - Shift reduce table 13](#_Toc462957728)

[Figure 9 - Debugger Application 14](#_Toc462957729)

[Figure 10 - Configurations Window 16](#_Toc462957730)

[Figure 11 - Cygwin console 17](#_Toc462957731)

[Figure 12 - Telnet component diagram 19](#_Toc462957732)

[Figure 13 - Compiler output windows 19](#_Toc462957733)

[Figure 14 - Output visualizer 20](#_Toc462957734)

[Figure 15 - UI Commands 20](#_Toc462957735)

[Figure 16 - Remote 21](#_Toc462957736)

[Figure 17 - Visual Studio project dialog 23](#_Toc462957737)

[Figure 18 - Project created 23](#_Toc462957738)

[Figure 19 - New item dialog 24](#_Toc462957739)

[Figure 20 - Base code file 24](#_Toc462957740)

[Figure 21 - Syntax highlighting 25](#_Toc462957741)

[Figure 22 - Editor error 25](#_Toc462957742)

[Figure 23 - Error window 25](#_Toc462957743)

[Figure 24 - intellisense 26](#_Toc462957744)