California University of Pennsylvania
Department of Mathematics, Computer Science and Information Systems

Code Analysis Software Package (Project C.A.S.P.) User Manual

Ryan Tedeschi
Dylan Carson

Instructor Comments/Evaluation

Table of Contents

1 Introduction

1.1 Project Overview

The goal of the Code Analysis Software Package (Project C.A.S.P.) is to enhance throughput and output of software developers through automating important tasks in the typical development workflow. Using experience in the field and metrics of the costliest processes in development, the software package will include tools to effectively reduce the amount of time and money squandered on these tasks, and also increase quality and functionality of final software products. The core deliverable will be a set of flexible command-line tools, primarily targeting Windows and Unix platforms, written in C++. A secondary deliverable will include a user interface application, written in C#, to consume the tools in the way the project was envisioned. The core deliverable will demonstrate the team's understanding of software development techniques, while the secondary objective is meant to demonstrate the team's knowledge of flexible modular development.

1.2 Problem

Small mistakes such as an exponentially growing or infinite loop or a poorly written module could lead to large amounts of money and time wasted. Retraining developers to utilize new unfamiliar languages can be expensive as well. Fault detection and outlining can be time consuming processes as well, taking up time to learn that particular languages in's and out's.

1.3 Solution

The idea of the Code Analyzer software package was conceived to reduce the amount of time and money spent on the expensive tasks of fault detection, outlining, and training. The project will automate some of the most time consuming processes, including inter-developer pattern unification, performance analysis, developer training, and program flow outlining. It will work alongside the developer and developmental tools in order to create a more streamlined software implementation process.

1.4 Intended Audience

The targeted audience for this document includes any person who plays any role in the software development process, including developers, architects, and managers. In addition, this document is intended for Dr. Weifeng Chen of California University of Pennsylvania, as well as any faculty members that are involved in the evaluation process of Senior Projects in the Math and Computer Science department of the University.

1.5 Secondary Audience

In addition to the intended audience of C.A.S.P., others are likely to interact with this software. These include others such as software engineers, designers and people learning how to code. This software can be used to help others understand why their program is running so long, translating it to a different language, or seeing an actual flowchart of their code.

2 System Block Diagram

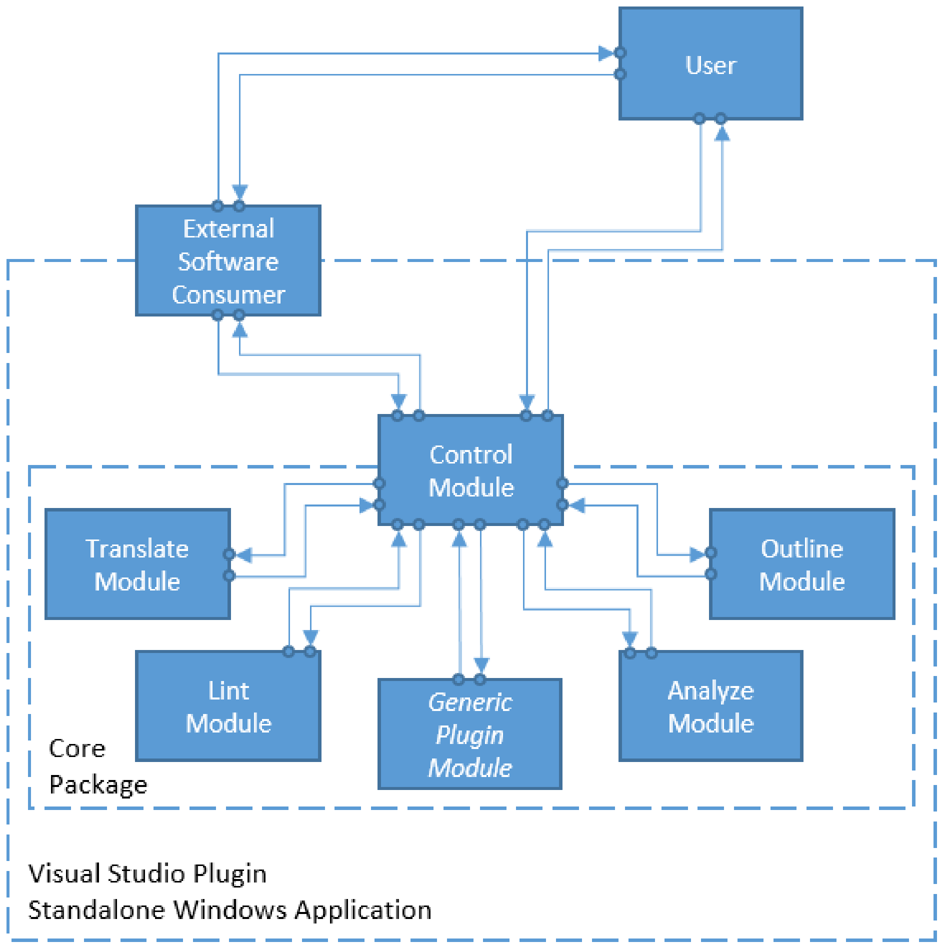


Figure 2.1 Block Diagram of the Code Analysis Software Package

Figure 2.1 depicts the interactions between components in the system. At the lowest functional level exists the core package. The Control module interacts with the plugin modules, one at a time, to achieve functional diversity. Included within the core package is the block labeled *Generic Plugin Module*. This block exists to demonstrate the ability to alter the functionality of the core package without the necessity of modifying existing code and depict how another plugin would integrate with the system.

The next level higher is the external software consumer. The proposed Microsoft Visual Studio plugin and implemented standalone Windows application implementations fall into this category. At this level, a software cannot directly manipulate the core package other than to request it to perform a predefined operation. Third party implementations of our software package would utilize this method of access.

At the highest level exists the human user. The user can access the Control module directly, or through an external software interface, but is limited to the data that either provides. The user cannot directly control the inner workings of either implementation.

3 Project Implementation Details

3.1 Differences from the Design Document

In the Design Document, the team set goals for the project for the desired final product before any implementation was done. The team discussed how the control and other modules would operate within the core implementation, how the different implementations would function, and proposed methods of interoperation between the core implementation and other implementations.

For the Core implementation, the team had proposed functionality for the control module for coalescing multiple input source code to one large source. This was not implemented, as it was decided to only accept one source file or snippet to reduce complexity of the software. Returning data from the core implementation was also changed as well. Instead of returning data from the application, the team printed return data to the standard output in the form of JSON (JavaScript Object Notation).

The team also altered how the Outline module would work from the initial proposition. It was to take the snippet of code provided by the user and create a flowchart of it. The implemented functionality differs slightly from the design document because it was not expected to be able to format the flowchart as well as was implemented.

Next, some logic in the design document for the Analyze module was not implemented. The proposed logic for this module included thorough analysis of source code to produce the complexity analysis. The implemented logic was not as thorough as the team had proposed, as the logic was very complex and the team did not enough time to implement and test a full set of rules. These problems are discussed in section 3.2.4.

For the Standalone Application, implementation went exactly as planned. The only minor difference regarding this was that it was scheduled to be completed in the last phase of the implementation. It was found that completing it earlier was useful because it made the output format neater and easier to read, and was therefore implemented directly after completion of the Outline module.

This Gantt chart shown in figure 3.1 gave the team a schedule at the beginning of this project. The last difference from the design document is in this. The team gave the time schedule broad time slots as to make sure there was sufficient time for each module. This was to cover any errors or other interruptions to workflow during implementation. It was found that the control and analyze modules took slightly longer than anticipated. This was offset however by the fact that the standalone, outline, and translate modules took less time than expected to complete.

During the implementation phase of the project, it also became more clear what needed to be worked on each week and the team was able to better address implementation needs than during the creation of the Design Document. The team also dropped the requirement of implementing the Lint module and Visual Studio Plugin Implementation due to time restraints.

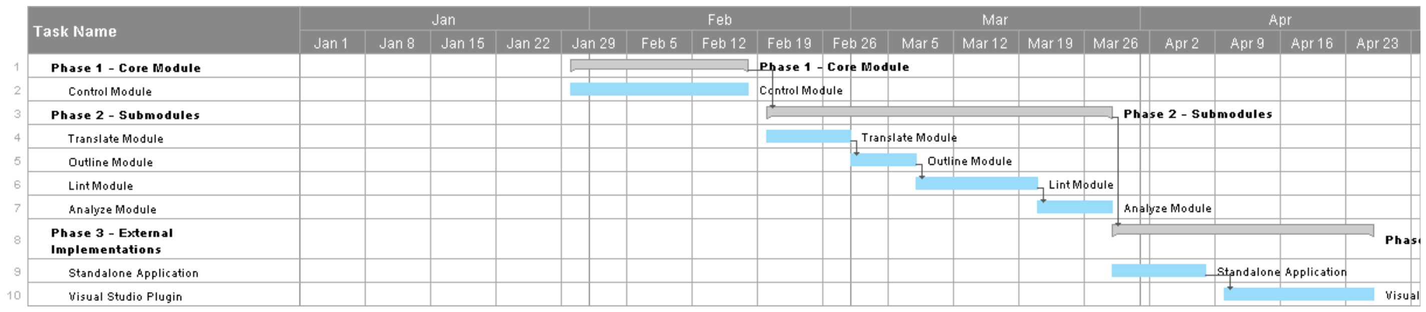


Figure 3.1. Gantt Chart

3.2 Challenges During Implementation

3.2.1 Control

Because the Control module controls all other modules, it was necessary for the team to return to this module periodically to update its functionality. Implementation of the Analyze and Outline modules in particular required attention to be returned to this module. During implementation of the Outline Module, it was found that the method of returning data would have to be updated. During implementation of the Analyze module, the parsing algorithms needed to be updated in order to incorporate syntactic action routines. Aside from these setbacks, implementation of this module was straightforward.

3.2.2 Standalone

During implementation of the Standalone Application, the team encountered the issue of running the console program effectively within the application. Due to delay of operation of the console program, the user interface froze due to background processing of the console program. The team resolved this issue by running the application on a separate thread, and allowing the user to terminate the thread at any time during operation without terminating the application to avoid the issue of having to wait for the console program to finish before executing again.

3.2.3 Translate

The Translate module was originally assigned to the team's third group member, however, was eventually completed by the other two. Unforeseen circumstances led to the third group member leaving the project after five weeks of being assigned the Translate module, leaving the rest of the group to pick it up with only a few weeks left of the implementation phase. The module was completed within two days of being reassigned to the other group members, and the team recovered fairly well from this setback. During implementation, the team also had to modify the existing grammar definitions for the C++ language and create and continually modify a grammar definition for the JavaScript language in order to get translations working properly.

3.2.4 Analyze

The Analyze module gave the team the most problems, mostly due to the limited amount of time remaining to implement it after completing the Translate module. The algorithms were not able to be completed enough, and the team had to improvise by making certain assumptions about the input code, such as the assumption that for-loop would always have a variable assignment, a conditional statement, and a post-loop incrementation. Another assumption that had to be made in order to have a working module is that any expression that did not resolve to an integer literal was a variable statement, and therefore always evaluated to a non-constant expression. Another issue encountered during implementation of this module was that expressions needed to be resolved in order to determine if they were variable or not. The team partially resolved this issue by implementing syntactic action routines and implementing value prediction and expression resolution action routines. Due to time constraints, a thorough implementation including while-loops, function resolutions, and algorithms for a more complete analysis were unable to be completed. While there will always be uncertain analyses, future implementation would yield more effective algorithms and a more complete analysis of code, as the team has logic planned to accomplish these tasks.

3.2.5 Outline

During implementation of the Standalone Application Analyze Module, the team rean into problems while trying to generate the flowchart edges. Due to the nature of a flowchart, nodes can be drawn anywhere and connected to any other node on the flowchart graph. The algorithm that the team constructed was able to connect nodes together, but due to time constraints and the understood complexity of the necessary algorithm, logic was not implemented for edges to avoid running through nodes. Therefore, edges are drawn over nodes in some cases.

3.2.6 Lint

Due to time requirements and issues within the team, the Lint module was not able to be completed. This module was meant to assert predefined rules on the user's source code. The rules were to be defined in the lint descriptor file. Given future implementation time, the team would be able to implement this using a method similar to XSD (XML Schema Definition) assertions on XML (Extensible Markup Language) files.

4 Uses of Software Engineering Principles

The team followed project construction standards discussed in Senior Project I (CSC 490) during the Fall semester of 2016 to create C.A.S.P. The project was broken down into requirements, specification, design, and implementation. Each phases was necessary in its own right.

The requirements phase was completed early in Senior Project I, yielding a document to provide an overview and refine the team's project idea. This document laid the foundation for the project and future documents. Certain ideas in this document were changed and omitted throughout the project's progression, but the general idea remained throughout all phases.

The next phase was the specification, completed mid-semester in Senior Project I. This phase was necessary to define each process and what they would do in the software. The resulting document gave insight to how exactly it would work and how each module would work with each other. Flowcharts and block diagrams were created to help visualize these processes.

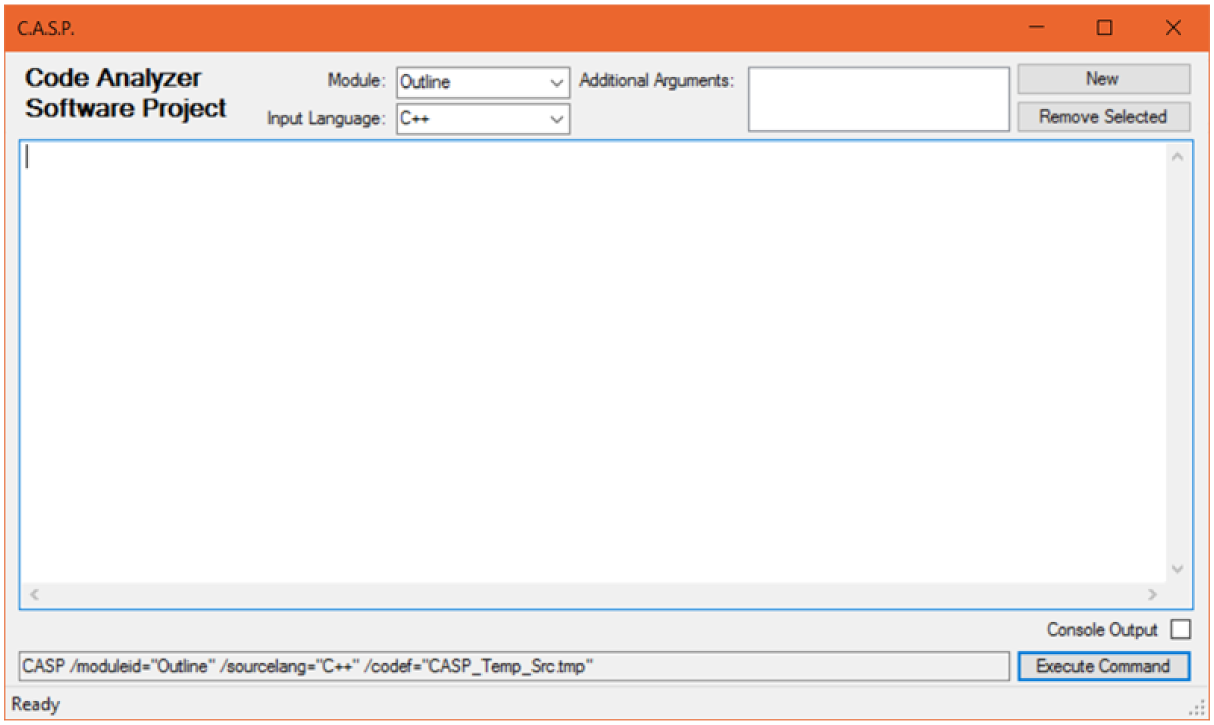
Completed at the end of Senior Project I was the design phase. This phase required the team to break down each module and go over what each process needed. This helped catch and handle future errors and improved the overall modular capability of the program.

The implementation phase was set up by the previous three phases, and completed over the duration of Senior Project II (CSC 492) during the Spring 2017 semester. It was important to refer to the previous documents when the team began implementation to assist us through each module. A Gantt chart was created to manage implementation time with each component. Each module was tested separately at its completion and then was added to the software.

5 User's Manual

The Code Analyzer Software Package was designed and developed with software engineers in mind. It utilizes various aspects of computer science to achieve this, including language translation, complexity analysis, and flowchart constructs. It is also a tool for learning new programming languages and saving time and money, and using the basic console application can be tedious and errors can occur if not used correctly. The console application is always available to use, but a standalone application was developed to provide an intuitive graphical window to the same powerful tools of the console application. This application allows users to easily interact with the console application and the resulting data.

5.1 Standalone Application



Module: Outline
Input Language: C++

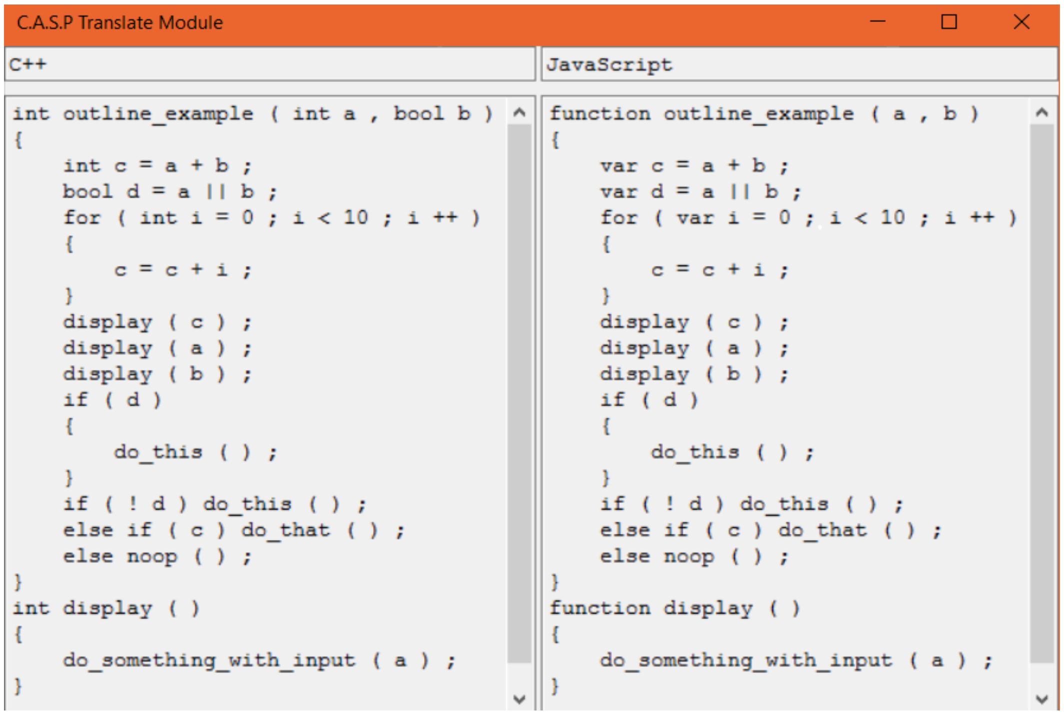
```
CASP /moduleid="Outline" /sourcelang="C++" /codef="CASP_Temp_Src.tmp"
```

Figure 5.1. Standalone Application interface

Figure 5.1 depicts the standalone application. This application was created for simplicity, and there are few requirements to use it. First, the desired module should be selected using the dropdown labeled *Module*. This is located at the top-center of figure 5.1, and current options include Outline, Analyze, Print, and Translate, representing all implemented modules. The next step is to select the source code input language using the dropdown labeled "Input Language". There are currently only options for C++ and JavaScript, however, there are plans to add more language definitions in the future. Next, any additional arguments for the selected module should be added using the controls located in the top right of the application. To add an argument, the *New* button can be clicked, which opens a dialog box prompting for argument information. Upon confirmation of the new argument, it will be added to the list directly to the left of the *New* button. To remove an argument simply click select the undesired argument and click the *Remove Selected* button. As a usage example example, if the desired functionality is to translate a snippet of code from C++ to JavaScript, the user should select the *Translate* option in the Module dropdown, *C++* in the Source Language dropdown, and add an argument *targetlang* with a value of *JavaScript*. The next control of interest is the user input box. This is located in the center of the window and is where desired code snippets to be processed will be entered.

Upon filling out all of the information described above, the user should then shift focus to the *Execute Command* button. This button invokes the console program, waits for completion, then processes and displays its output. The console program is run as a background task so the user can continue making modifications to source code or prepare for another execution of the program. Therefore, while the console program is executing in the background, the *Execute Command* button changes to a *Stop* button that can be pressed to cancel execution of the current running process. At the bottom of the application is a status area, which indicates either that the program is currently processing or that it is ready to process again, including the processing time of the last execution.

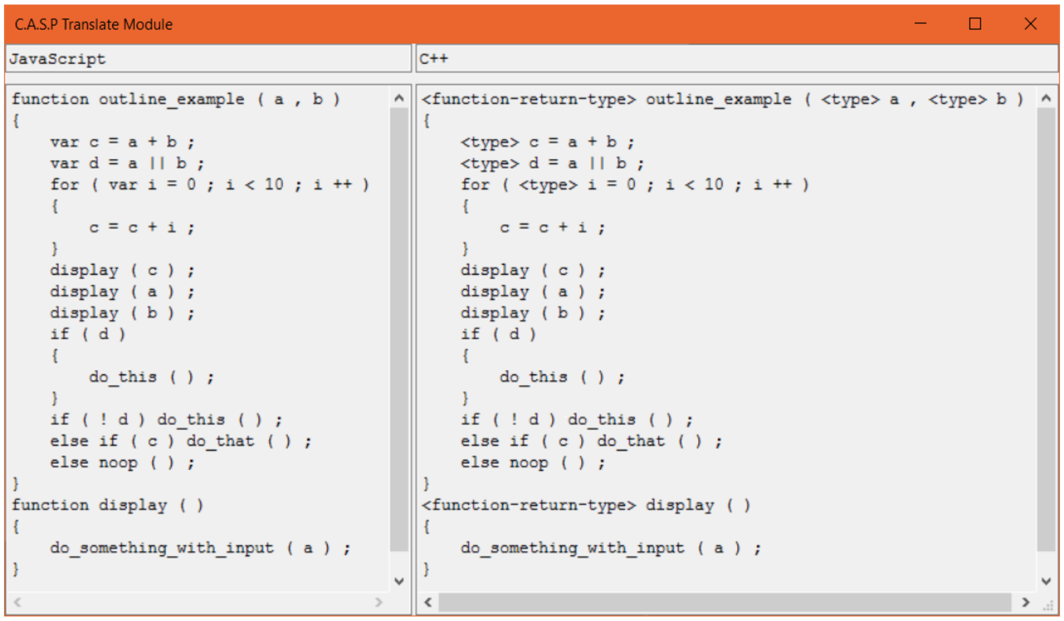
For development purposes, two additional controls were included in this implementation. The command preview area, located to the left of the Execute button, displays the command that is used to invoke the console program. Lastly, the Console Output checkbox, located above the Execute button, if checked shows the console output as it would be if run directly from the console. This feature is particularly useful for testing development on modules.



```
int outline_example ( int a , bool b )
{
    int c = a + b ;
    bool d = a || b ;
    for ( int i = 0 ; i < 10 ; i ++ )
    {
        c = c + i ;
    }
    display ( c ) ;
    display ( a ) ;
    display ( b ) ;
    if ( d )
    {
        do_this ( ) ;
    }
    if ( ! d ) do_this ( ) ;
    else if ( c ) do_that ( ) ;
    else noop ( ) ;
}
int display ( )
{
    do_something_with_input ( a ) ;
}
```

```
function outline_example ( a , b )
{
    var c = a + b ;
    var d = a || b ;
    for ( var i = 0 ; i < 10 ; i ++ )
    {
        c = c + i ;
    }
    display ( c ) ;
    display ( a ) ;
    display ( b ) ;
    if ( d )
    {
        do_this ( ) ;
    }
    if ( ! d ) do_this ( ) ;
    else if ( c ) do_that ( ) ;
    else noop ( ) ;
}
function display ( )
{
    do_something_with_input ( a ) ;
}
```

Figure 5.2. Translate Module input/output (C++ to JavaScript)



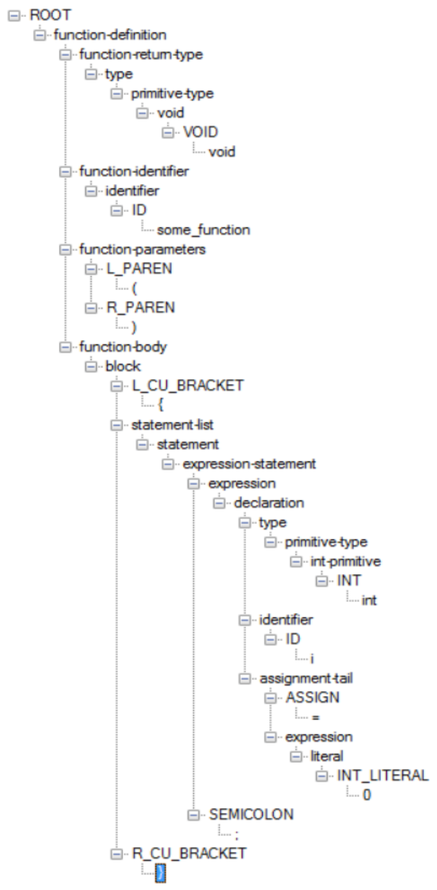
```
function outline_example ( a , b )
{
    var c = a + b ;
    var d = a || b ;
    for ( var i = 0 ; i < 10 ; i ++ )
    {
        c = c + i ;
    }
    display ( c ) ;
    display ( a ) ;
    display ( b ) ;
    if ( d )
    {
        do_this ( ) ;
    }
    if ( ! d ) do_this ( ) ;
    else if ( c ) do_that ( ) ;
    else noop ( ) ;
}
function display ( )
{
    do_something_with_input ( a ) ;
}
```

```
<function-return-type> outline_example ( <type> a , <type> b )
{
    <type> c = a + b ;
    <type> d = a || b ;
    for ( <type> i = 0 ; i < 10 ; i ++ )
    {
        c = c + i ;
    }
    display ( c ) ;
    display ( a ) ;
    display ( b ) ;
    if ( d )
    {
        do_this ( ) ;
    }
    if ( ! d ) do_this ( ) ;
    else if ( c ) do_that ( ) ;
    else noop ( ) ;
}
<function-return-type> display ( )
{
    do_something_with_input ( a ) ;
}
```

Figure 5.3. Translate Module input/output (JavaScript to C++)

Figures 5.2 and 5.3 depict a sample output of the Translate module. In order to use this functionality, the *Translate* module must be selected on the main form of the application, a source language must be selected, and the argument *targetlang* must be added to the additional arguments. Figure 5.2 shows a translation from C++ input (left side of image) to JavaScript output (right side of image). Figure 5.3 shows the JavaScript output from Figure 5.2 being translated back to C++.

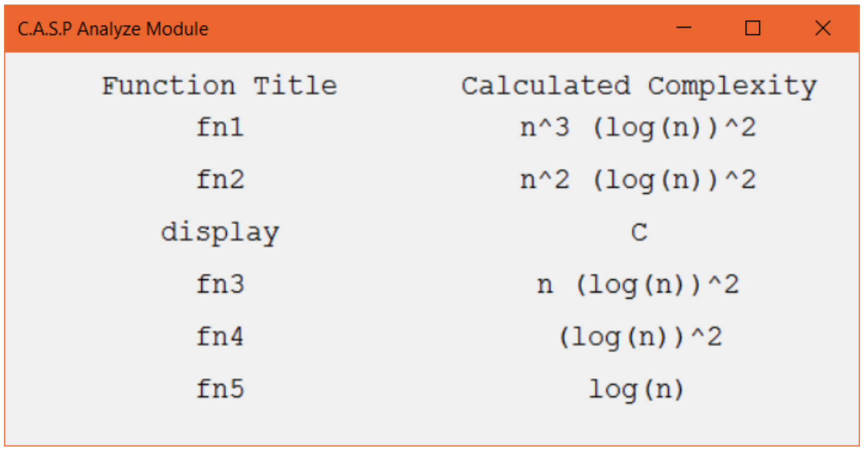
It should be noted here that certain languages have more or less general constructs than others. For example, when translating a declaration statement in C++ to its JavaScript equivalent, type information is lost due to JavaScript's generic nature (i.e. C++ statement *int i = 0;* translates to *var i = 0;* in JavaScript). Therefore, a translation from generic to specific constructs, such as a declaration statement from JavaScript to C++, type information cannot be included, and a placeholder for variable *type* is included in the resulting translation (i.e. JavaScript statement *var i = 0;* translates to <*type*> *i = 0;* in C++, indicating the missing type construct).



```
ROOT
  function-definition
    function-return-type
      type
        primitive-type
          void
            VOID
              void
    function-identifier
      identifier
        ID
          some_function
    function-parameters
      L_PAREN
        (
      R_PAREN
        )
    function-body
      block
        L_CU_BRACKET
          {
        statement-list
          statement
            expression-statement
              expression
                declaration
                  type
                    primitive-type
                      int-primitive
                        INT
                          int
                  identifier
                    ID
                      i
                  assignment-tail
                    ASSIGN
                      =
                    expression
                      literal
                        INT_LITERAL
                          0
              SEMICOLON
                ;
        R_CU_BRACKET
          }
```

Figure 5.4. Print Module

Figure 5.4 shows the Print module. This module was included during the implementation phase, mostly for development purposes, to show how the code input is parsed by the program. This module is accessed by selecting the *Print* module in the standalone application. This module shows the concrete syntax tree that the program generates from user input and provides to submodules for further processing.

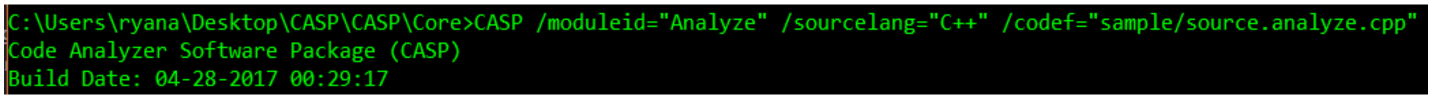


```
Function Title          Calculated Complexity
     fn1                  n^3 (log(n))^2

     fn2                  n^2 (log(n))^2

   display                      C

     fn3                   n (log(n))^2

     fn4                    (log(n))^2

     fn5                      log(n)
```

Figure 5.5 Analyze Module Output

In figure 5.5 shows the output of the Analyze module after processing some source code including six functions, as shown in the left column of the figure. The Analyze module analyzes each included function and provides an overall complexity. If no functions are included in the input, it analyzes provided code as if it were the body of a function. This module is accessed from the main form by selecting the *Analyze* module, a source language, and providing source code in the source input box. As shown in the figure, the application presents each function analysis by showing its title in the left column and the corresponding analysis in the right column.

5.2 Command Prompt



```
C:\Users\ryana\Desktop\CASP\CASP\Core>CASP /moduleid="Analyze" /sourcelang="C++" /codef="sample/source.analyze.cpp"
Code Analyzer Software Package (CASP)
Build Date: 04-28-2017 00:29:17
```

Figure 5.6 Command Prompt

Figure 5.6 is an example of a command prompt operation. To use the command prompt instead of the standalone application, arguments must be provided in the following format:

/<argument_id>="<argument_value>",

where <*argument_id*> should be replaced by the appropriate argument identifier and <*argument_value*> by the corresponding argument value. For example, there are three arguments provided to the program in Figure 5.6: *moduleid*, *sourcelang,* and *codef*. A list of all appropriate arguments for current modules is as follows:

- Argument ID: *moduleid*, Argument Value: Module name
  - Argument is always required
  - Value reflects the desired module
- Argument ID: *sourcelang*, Argument Value: Language name
  - Argument is always required
  - value reflects the language of input code
- Argument ID: *codef*, Argument Value: File location
  - Argument is required if */code* argument is not provided
  - Value reflects a file read as input source code
- Argument ID: *code*, Argument Value: A raw snippet of code
  - Argument is required if */codef* argument is not provided
  - Value reflects a raw snippet of code read as source code

- Argument ID: *targetlang*, Argument Value: Language Name
    - Argument is required if */moduleid* is *Translate*
    - Value reflects the desired language of output code

The program will print a JSON object in the standard output, beginning directly after the text *CASP_RETURN_DATA_START* and ending immediately before the text *CASP_RETURN_DATA_END*. Data returned by the program will be under this object's *Data* property, errors encountered by the program will be listed under the *Errors* property, and warnings provided by the program will be included in the *Warnings* property.

6 Installation Instructions

Not much of an installation process exists currently. The user must acquire a copy of the team's source directory and place it somewhere on their filesystem. The console application can be run by navigating to the Core directory under the master project directory in a command prompt, and executing the file CASP.exe with the arguments explained in the previous section. To execute the Standalone Windows application, the a link exists in the "Standalone Application" directory under the master project directory to begin execution.

7 Contacts

If there are any inquiries regarding installation or usage of the software, the creators of C.A.S.P. would be happy to answer any questions at the following email addresses.

Ryan Tedeschi
Ted4686@calu.edu

Dylan Carson
Car5921@calu.edu

8 Frequently Asked Questions

What is C.A.S.P.?

C.A.S.P. is development tool designed to help cut down on tedious development tasks. It does this by allowing the user to generate flowcharts, analysis, and translations of their code.

What can C.A.S.P. do?

Currently, C.A.S.P. can outline and provide a flowchart of the source code, analyze code for complexity, and translate the code to a different language. The team has only implemented language definitions for the C++ and JavaScript languages, but implementation of new languages is relatively simple. Also, additional modules can easily be developed and included in the project to add functionality.

Why is C.A.S.P. useful?

The software is intended to simplify or eliminate tedious development tasks. From learning a new language, to laying out cryptic code, to performance analysis, C.A.S.P. is designed to take monotonous work out of development and leave more resources for architecture and planning.

Any plans for future implementations?

As stated before, this project was limited due to the time constraints. However, plans for future implementations are quite extensive. A Lint module would be added to provide users the set rules for the code being analyzed. More extensive logic for the Translate module may be implemented to allow for more languages and possibly even semantic translations instead of just syntactic translations. Additional and more thorough logic to the Analyze module is planned to provide more complete complexity analysis.

9 Appendix

A Glossary

JSON - JavaScript Object Notation, is a lightweight data interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language.

Syntactic Action Routines - Routines invoked by a compiler to perform some predefined action based on the parsed code.

Command Line - An interface between the user and a computer where the user types a single command at a time to invoke some functionality.

Consumer - A person or third-party software that interacts with the software.

C++ - A programming language developed at Bell Labs in 1979 and standardized in 1998. The language features imperative and object oriented abilities.

C# - A programming language conceived by Microsoft in 1999, influenced by C++. The language features imperative, object oriented, and functional abilities.

Javascript - an object-oriented computer programming language commonly used to create interactive effects within web browsers.

Dynamic Link Library - (DLL) is a module that contains functions and data that can be used by another module. A DLL can define two kinds of functions: exported and internal. the exported functions are intended to be called by other modules, as well as from within the DLL where they are defined.

IDE - Integrated Development Environment. This is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools, and a debugger. Most modern IDEs have intelligent code completion.

Implementation- a realization of a technical specification or algorithm as a program, software component, or other computer system through computer programming and deployment. Many implementations may exist for a given specification or standard.

Linting- Lint was the term originally given to a particular program that flagged some suspicious and non-portable constructs in C language source code. The term is now applied generically to tools that flag suspicious usage in software written in any language. In the context of this project, suspicious means that code is out of the ordinary and not following any recognizable guidelines.

Module - In software, a module is a part of a program. Programs are composed of one or more independently developed modules that are not combined until the program is linked. A single module can contain one or several routines.

Performance Analysis - The generalization of the performance of a computer program as compared to a mathematical function, such as $f(x)=n^2$ or $f(x)=n\times log(n)$. Performance analysis uses relational operators - less than [or equal to], greater than [or equal to], and equal to - to describe the relationship between program complexity and a mathematical rate of growth.

Plugin - a software component that adds a specific feature to an existing computer program.

Language Translator- a computer program that performs the translation of a program written in a given programming language into a functionally equivalent program in a different computer language, without losing the functional or logical structure of the original code.

Machine Code- a computer programming language consisting of binary or hexadecimal instructions that a computer can respond to directly.

B Team Details and Contributions

The work herein was evenly distributed by the participants of the Code Analyzer Software Package, Ryan Tedeschi and Dylan Carson. The preparation of this document was started in the final phases of the project and completed alongside the implementation of the software.

Due to the loss of the third team member, the remainder of the team had to adjust appropriately. Additional work for each group member was significantly increased, and each member had to work around their already daunting schedule for education and work alike. Because of this load on each of the remaining members, the team made the difficult decision that certain proposed functionalities of the software had to be omitted in order to reach a functional final product.

For each module, the team had a meeting and discussed the corresponding layout and pseudocode. This helped detect any errors left unknown by the Design document that could occur during implementation.

Specifically, Ryan worked on and completed the Control module and the Outline module. Dylan worked on and completed the Analyze module and worked on the weekly reports and presentations, as well as much of this document. Both members picked up some pieces of the Translate module upon loss of the third team member.

C References

C++ Language Reference:
"C and C reference." Cppreference.com. N.p., n.d. Web. 28 Apr. 2017.

Functionality of a Compiler:
Pyzdrowski, Anthony. "Language Translation" California University of Pennsylvania, Eberly, Spring 2017. Lecture.

Finite State Machines and Context Free Grammars:
Chen, Weifang. "Theory of Languages" California University of Pennsylvania, Eberly, Spring 2017. Lecture.

Definitions
Wikipedia. Wikimedia Foundation. Web.

Listing file generation:
Source code beautifier / syntax highlighter – convert code snippets to HTML « hilite.me. N.p., n.d. Web. 28 Apr. 2017.

D Workflow Authentication

I, Ryan Tedeschi, confirm that I have performed the work documented herein

Signature:____________________________________ Date:

I, Dylan Carson, confirm that I have performed the work documented herein

Signature:____________________________________ Date:

E Code

This Page is intentionally blank. The following pages include source code of each file created by the team.