FLEX AND BISON IMPLEMENTATION OF A C STYLE CHECKER

by

Bruce Edward Dearing

Acadia University
December 2013

This project by Bruce Edward Dearing
is accepted in its present form by the
Department of Computer Science
as satisfying the requirements for the degree of
Bachelor of Computer Science

Approved by the project Supervisor

(Dr. Diamond)	Date

Approved by the Head of the Department

(Dr. Benoit)	Date

I, Bruce Edward Dearing, grant permission to the University Librarian at Acadia University to reproduce, loan, or distribute copies of my project in micro form, paper or electronic formats on a non-profit basis. I, however, retain the copyright in my project.

_____

Signature of Author

_____

Date

Contents

Abstract **vii**

Acknowledgements **viii**

1 Introduction **1**

2 Description of tools **2**

2.1 Flex 2

2.1.1 Flex Input File 2

Options and Declarations 3

Token Action Section 3

C Code Section 5

Flex Summary 5

2.2 Bison 5

2.2.1 Bison Input File 6

Options and Declarations 6

Rule Declarations 7

C Code Section 8

Bison Summary 8

2.3 Indent 8

Indent Summary 9

2.4 Vim 9

Vim Summary 9

3 Problem Description **10**

4 Solution Description **11**

5 Implementation Description **12**

5.1 Comments 12

5.1.1 Modification of Comments by *Indent* 12

5.1.2 Check Comments Module 14

5.2 Indentation 14

5.2.1 Check Indent module 15

5.3 Common Errors and Style 16

5.4 Composite Check 16

5.4.1 Ignored Statements 16

5.4.2 Typedef Statements 16

5.4.3 Recurring Variables 17

5.4.4 Grammar Modifications and Extensions 17

5.5 Format and White Space 17

5.6 Web Based Interface 17

6 Possible Extensions **19**

6.1 Preprocessor 19

6.2 Additional Productions 19

6.3 Replace Indent and Vim 20

6.4 Javascript 20

6.5 Comments Extension 20

7 Summary and Conclusions **21**

A File Listing **22**

B User Manual **24**

B.1 Usage 24

B.1.1 Command line 24

B.1.2 Web based 24

C C Coding Style Notes **25**

D Available Checks **34**

E Required Software **37**

F cinoptions **38**

Bibliography **50**

Abstract

Students enrolled in second year computer programming at Acadia University are required to format their program code according to a set of style guidelines. In previous years there was no C style checker freely available to the students. This occasionally resulted in students submitting improperly formatted code. The solution to this problem required that a program be constructed that maintained flexibility while providing enough functionality to adequately improve the quality of code formatting, and produce a reduction in the number of style errors present in students submitted code. Ideally, the program would be used as the back end for a web page, so that students would be able to submit their program and the web server would return either a confirmation that the style meets the guidelines or a list of non-conforming constructs. My omnibus solution to this problem combines four translator-based programs together, along with various functions of additional programs in order to provide a number of style checks on files submitted by students.

Acknowledgements

I would like to acknowledge the help and guidance provided by Dr. Diamond.
Dr. Diamond provided me with very valuable advice, guidance, and useful critiques during the preparation of my final year project.

Chapter 1

Introduction

Despite the seemingly endless supply of code beautifiers available for use, I was unable to locate an open source solution that provided a method of identifying errors to the user. Code beautifiers or formatters are typically designed to only re-format the code that they are supplied with. The initial problem with having students use a code beautifier to format their code, is that by design code beautifiers are vastly configurable and could not provide us with a method of enforcing a strict coding standard. Therefore the only reasonable solution to the unavailability of a C style checker would be to construct one based on existing models for other languages. The model which my program was blueprinted from is checkstyle. Checkstyle is a development tool designed to help programmers write Java code that adheres to a coding standard by identifying errors in their code and reporting them to the user.

The main objective of the project was to provide a C style checker similar to checkstyle that students could use to verify their code before submission. The program will check student code against a set of available checks. The program will then generate a report of the errors for the student to action. The feedback provided by the program to the user will allow the student to learn from their mistakes and provide a method of enforcing a uniform coding standard.

In the remainder of this report I will discuss the tools that I used to construct a C style checker program, my implementation, and any extensions that would prove beneficial.

Chapter 2

Description of tools

2.1 Flex

The **F**ast **Lex**ical analyzer generator is a tool for generating scanners. A *scanner* is a program which recognizes regular expression patterns in text and then performs an action. The operation of Flex can be described as in [4]:

> "The Flex program reads user-specified input files for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates a C source file named, 'lex.yy.c', which defines the function yylex(). The file 'lex.yy.c' can be compiled and linked to produce an executable. When the executable is run, it analyzes its input for occurrences of text matching the regular expressions for each rule. Whenever it finds a match, it executes the corresponding C code."

2.1.1 Flex Input File

A Flex scanner description file consist of three main parts, which are separated from each other by %% lines.

Options and Declarations

```
\%e 1019  /* number of parsed tree nodes */
\%p 2807  /* number of positions */
\%n 371   /* number of states /*
\%k 284   /* number of packed character classes */
\%a 1213  /* number of transitions */
\%o 1117  /* size of output array */
O [0-7]
D [0-9]
NZ [1-9]
L [a-zA-Z_]
A [a-zA-Z_0-9]
H [a-fA-F0-9]
ES (\\([’"\?\\abfnrtv]|[0-7]{1,3}|x[a-fA-F0-9]+))
WS [ \t\v\f]
\%option yylineno

/* START Exclusive Start State definitions. */
\%x COMMENT
/* END Exclusive Start State definitions. */

/* START Inclusive Start State definitions. */
\%s TYPE\_DEF
/* END Inclusive Start State definitions. */

%{
#define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;
#include "check_comments.tab.h"

int yycolumn = 1;

void add_entry(int line_num, int flag, char* str);

%}
```

This is where we can define the options that we want Flex to include in the scanner. Flex offers several hundred options; most can be written as
%option name
This section is also where we would define any inclusive or exclusive start states that we might require in the scanner, as well as any regular expression simplifications. We can also include any header files or macro definitions inside %{ }% blocks.

Token Action Section

The second section is where we construct our list of regular expression and action pairs.

```
"/*"                              { BEGIN(COMMENT);
                                    return(START_COMMENT);
                                  }
"//".*                            { /* consume //-comment */ }
<COMMENT>"*/"                     { BEGIN(INITIAL); return(END_COMMENT);}
<COMMENT>\n                       { }
<COMMENT>^{BG}{F}{WS}+{FN}        { return(FILE_LBL);}
<COMMENT>^{BG}{A}{TH}{WS}+{BN}    { return(AUTHOR); }
<COMMENT>^{BG}{V}                 { return(VERSION); }
<COMMENT>^{BG}{D}{WS}+{DT}        { return(DATE); }
<COMMENT>^{BG}{P}                 { return(PURPOSE);}
<COMMENT>^{BG}{N}{ME}             { return(NAME); }
<COMMENT>^{BG}{A}{RG}             { return(ARGUMENTS); }
<COMMENT>^{BG}{O}                 { return(OUTPUT); }
<COMMENT>^{BG}{M}                 { return(MODIFIES); }
<COMMENT>^{BG}{R}                 { return(RETURNS); }
<COMMENT>^{BG}{A}{SS}             { return(ASSUMPTIONS); }
<COMMENT>^{BG}{B}                 { return(BUGS); }
<COMMENT>^{BG}{N}{OT}             { return(NOTES); }
<COMMENT>[^*\n]|.                 { }
{WS}                              { /* white space separates tokens */ }
.                                 { /* discard bad characters */ }
```

We have to be careful in the construction and ordering of our regular expressions in this portion, as Flex uses a greedy match, and any regular expressions below a larger match will never receive token matches. Fortunately Flex is intelligent enough to let us know when we make a mistake like this.

```
check_comments.l:142: warning, rule cannot be matched
```

C Code Section

The third section is C code that is copied to the generated scanner. In the declaration section, code inside of %{ and %} is copied near the beginning of the generated C source file.

```
void
add_entry(int line_num, int flag, char* str)
{
    char *rtnstr = fix_string(str);
}
```

In this section we can place small subroutines that may be required as part of the action on a regular expression match.

Flex Summary

By utilizing Flex's ability to match regular expressions paired with exclusive and inclusive start states we can easily construct a highly configurable scanner for use in a C style checker.

2.2 Bison

GNU Bison parser generator is a tool for generating LR or GLR parsers. A *parser* is a program which takes input provided to it from a scanner and takes that input and constructs an abstract syntax tree and checks the syntax based on supplied rules. The operation of Flex can be described as in [3]:

> "Bison is the gnu general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser."

2.2.1 Bison Input File

Options and Declarations

Similar to a Flex scanner description file a Bison grammar rule file consists of three main parts, which are separated from each other by %% lines.

```
%locations
%glr-parser
%expect 2
%expect-rr 0

%{

/* START Include files. */
#include <stdio.h>
#include <string.h>
/* END Include files. */

/* START Program variables */
extern char *yytext;
/* End Program variables */



/* START Definitions. */
#define YYERROR_VERBOSE
#define MAX_TRYS 1000
/* End Definitions. */
```

```
/* START Function Definitions */
void yyerror(const char *str);
/* START Function Definitions */
%}

%token BIDENTIFIER IDENTIFIER I_CONSTANT F_CONSTANT STRING_LITERAL
%start program_body
```

This is where we can define the additional options, tokens and starting token that we want Bison to include in the parser. This section is also where we would include any header files or macro definitions inside %{ }% blocks.

Rule Declarations

The second section contains simplified BNF rules.

```
generic_selection
    : GENERIC '(' assignment_expression ',' generic_assoc_list ')'
    ;

generic_assoc_list
    : generic_association
    | generic_assoc_list ',' generic_association
    ;

generic_association
    : type_name ':' assignment_expression
    | DEFAULT ':' assignment_expression { check_list(); }
    ;
```

This is where we supply the rules for productions and reductions of program input. C code can also be supplied inside the structure of a rule inside curly brackets.

C Code Section

The third section is C code that is copied to the generated parser. In the declaration section, code inside of %{ and %} is copied near the beginning of the generated C source file.

```
void
add_entry(int line_num, int flag, char* str)
{
    char *rtnstr = fix_string(str);
}
```

In this section we can place small subroutines that may be required as part of the action on a regular expression match.

Bison Summary

A Bison grammar input file can be carefully constructed to allow parsing of incredibly complex languages. The combination of Flex and Bison generator tools provide the required functionality to implement a C style checker.

2.3 Indent

GNU *indent* is a code formatter which modifies C code by inserting or deleting white space, based on a set of flags supplied by the user on the command line. For a reference on available flags see [8]. Constructing a grammar which parses a complex language like C would quickly become too complex if we required it to handle all the possible combinations of white space tokens. Therefore *indent* is employed to handle the required white space changes within the C style checker.

Indent Summary

The *indent* portion of the program originally proved very useful in formatting code to the required settings, however, as the project progressed *indent* was more of a hindrance than a help. The *indent* program proved to be inflexible in it's flag selection. When choosing a particular style if two forms of styling were acceptable, there was no clear way to allow both. It was impossible to turn off code modification for a particular style, selection of one style was enforced. As well *indent* modified code without the option to select a particular style. For example `char* string` or `char * string` would become `char *string` with no ability to select a style. In future designs I hope to improve *indent'*s source code or replace it with an alternate solution.

2.4 Vim

As the name suggests gnu *indent* can handle indentation, however, it is not very flexible in implementing different levels of indentation. *Vim* or Vi IMproved is a non-toy editor which has numerous useful features. One of them being *cindent*, with it's ability to flexibly re-indent code according to a set of flags supplied to it through a variable *cinoptions*. See Appendix F for details.

Vim Summary

Vim has proven to be very useful strictly as a flexible indentation component of the style checker, however, during the course of looking for a replacement program for *indent*, a possible solution has been found that could effectively replace *Vim* and *indent*. Currently `Astyle` is under review for addition to the program as a replacement.

Chapter 3

Problem Description

As previously stated in the introduction in chapter 1, in previous years there had been no C style checker freely available to the students, which occasionally resulted in students submitting improperly formatted code. It had been mentioned that on occasion students may consistently repeat the same style mistakes throughout the year, this could be a case of apathy or possibly the lack of responsive feedback. A large number of interactive learning and feedback studies argue that immediate feedback facilitates learning better than delayed feedback in applied settings.[1] Without a method of providing immediate corrective feedback to the students they may not be effectively retaining the corrective measures required to maintain their code as it applies to style errors.

Chapter 4

Solution Description

The construction of a C style checker will provide a solution to a number of the problems identified in the problem description in chapter 3. It will provide a method of immediate feedback to the students which will optimistically help to reinforce the knowledge of the coding standard. Additionally by providing the style checker through a web based interface, the style checker can provide the additional functionality of presenting the student their code directly to the right of their list of errors. The code view will additionally provide content which is editable allowing the students to make the corrections instantly. Unfortunately the code view lacks any interactive web features and provides only the minimal amount of functionality to make it a useful asset to students.

Chapter 5

Implementation Description

The core components of the C style checker are composed of four translator-based programs, which with the help of *indent* and *Vim* provide the ability to perform a number of checks on submitted files for a list of available checks see Appendix D. The reasoning behind using translator-based programs is based on the fact that they are highly configurable and each module could be easily updated or extended by modifying the scanner description and parser grammar if required.

5.1 Comments

For code comments the C style checker is required to verify code comments conform to the style guide as well as verify that files minimally contain a header comment identifying all the required data for the file; for more information on acceptable comment style see Appendix C. The format and placement of comments is currently handled by *indent*. As in all parts related to *indent*, the file is passed through *indent* and then compared against the original file to display required changes to the user.

5.1.1 Modification of Comments by *Indent*

The program *indent* formats both C and C++ comments. The program also attempts to leave properly formatted *boxed comments* unmodified.

```
   --comment-delimiters-on-blank-lines
```

This option places the comment delimiters on blank lines. Thus, a single line comment like `/* Loving hug */` can be transformed into:

```
/*
   Loving hug
 */
```

```
   --star-left-side-of-comments
```

This options causes stars to be placed at the beginning of multi-line comments. Thus, the single-line comment above can be transformed into:

```
/*
 * Loving hug
 */
```

indent can also fix an existing comments style, transforming the comment we have here at the top to match the one at the bottom.

```
/*A different kind of scent,
            for a different kind
    of comment.*/


/*
 * A different kind of scent,
 * for a different kind of comment.
 */
```

While *indent* is capable of handling the structure of comments, we require an additional component to handle the contents of the comment and enforce that a header comment is present.

5.1.2 Check Comments Module

The check comments module is a combination scanner-parser. The scanner ignores all characters until it matches the initial `/*` indicating the start of a multi-line comment. The scanner then enters an exclusive start state in which it attempts to parse common comment labels which would be present in either header or function comment blocks. The scanner then passes off the tokens to the parser which verifies the syntax of the tokens and prints to stdout in the case where the tokens do not parse correctly. Upon matching `*/` while in the exclusive start state it will then return to the scanners initial start state.

```
   HEADER COMMENT
/*
 * File:     A2P1.c
 * Author:   My Name 100123456
 * Date:     2011/09/12
 * Version:  1.0
 *
 * Purpose:
 * ...
 */
```

```
FUNCTION COMMENT
 /*
  * Name:         my_func
  * Purpose:      ...
  * Arguments:    ...
  * Output:       ...
  * Modifies:     ...
  * Returns:      ...
  * Assumptions:  ...
  * Bugs:         ...
  * Notes:        ...
  */
```

5.2 Indentation

For code indentation the C style checker is required to check for correct indentation for many different levels of indentation. Unfortunately *indent* was not capable of this level of flexibility. Enter *Vim's cindent* to save the day. Using *Vim* in ex mode we are able to pass the file in and indent it against a set of values specified in *cinoptions*. The set of values that the C style checker uses to adhere to the C style guidelines are:

```
e0,n0,f0,{0,}0,:2,=2,l1,b0,t0,+4,c4,C1,(0,w1
```

for a full listing or description of how these flags effect indentation see Appendix F.

5.2.1 Check Indent module

After the file has been re-indented it is then compared against the original file using *diff* with a custom output format set.

```
diff \
    --old-line-format='%l
' \
    --new-line-format='%l
' \
    --old-group-format='%df%(f=l?:,%dl)d%dE
%<' \
    --new-group-format='%dea%dF%(F=L?:,%dL)
%>' \
    --changed-group-format='%df%(f=l?:,%dl)c%dF%(F=L?:,%dL)
%<---
%>$$$
' \
    --unchanged-group-format=""
```

The output of the comparison is passed through a scanner program `indentR`. The indentR program then calculates the indentation level of the original file and the indentation level of the re-indented file for each line that the two files differ. It then prints to stdout the line number that the difference occurred, or the range in the case where multiple lines are incorrectly indented. Additionally the program prints the code on the line that differs, the current level of indentation, and the proper level of indentation.

```
38:
    printf("do nothing\n");

Code at indentation level [4] not at correct indentation [8].
```

5.3 Common Errors and Style

In order to keep the composite checks scanner from becoming too complex, a separate module for common errors was created to provide checks for some common errors which are easily discovered with the use of a scanner program. The program checks conditional, loop and struct statements for white space after the token and left curly location on a separate line. It additionally checks for tab and carriage return characters on a line. Once errors are identified it prints to stdout the line number on which the error occurred, the code on that line, and the appropriate warning message.

5.4 Composite Check

The bulk of the complex checks are performed by the composite checks module which is a General LR parser. The code for the module is based on modified versions of the ANSI C lex specification, and the ANSI C yacc grammar description [5].

5.4.1 Ignored Statements

The scanner specification has been modified to ignore preprocessor directives, include files, define statements, and comments. This greatly simplifies the parsing of the files and reduces the number of errors the parser encounters. The scanner is capable of ignoring complex statements by entering an exclusive start state when it matches a particular token.

5.4.2 Typedef Statements

When the scanner matches a typedef token the scanner enters an inclusive start state, this means all tokens outside of the inclusive state are still matched, however, tokens inside the inclusive state have priority. The inclusive state allows for typedef statements to consist of multiple identifier tokens without having to make another set of actions for those tokens. For example in the case of `typedef int banana` on matching typedef it enters the `<TYPEDEF>` inclusive start state, then upon matching

the `int` token it performs the action associated with `int`. The same method of using inclusive start states exists for struct, and enum statements.

5.4.3 Recurring Variables

When the scanner encounters variables that it will need to parse in the future like the name of a typedef, enumeration constant, or identifier name, it has to store the variable name locally so that it will be able to look it up if it occurs again and return the appropriate token to the scanner. This is accomplished by maintaining a hash table of symbol values, when a variable name is encountered the symbol table is checked for existence of the name and if it is not already in the table it is added.

5.4.4 Grammar Modifications and Extensions

The ANSI C yacc grammar description has been modified to include extra production rules or in some cases small C routines, to initiate checks for style errors within a C file.

5.5 Format and White Space

The format and white space portion of the program is currently handled by *indent*. As in all parts related to *indent*, the file is passed through *indent* and then compared against the original file to display required changes to the user. Although *indent* is inflexible it does perform well in correcting errors pertaining to white space and the format of the document.

5.6 Web Based Interface

The web based interface was a last minute upgrade to the project. It consists of three PHP files and one css stylesheet. The PHP files contain an uploader script which allows the student to upload a file to the server, a PHP class file line_sort.php which handles the sorting of the program output, and printing the output to the

screen. Finally the stylecheck.php handles moving the files to a temporary location and passing the files as input to scruffy.sh as well as handing the output of scruffy.sh to an instance of the line_sort class. The web interface provides a clean output of the reported errors and a code view portion to the right which displays the users code in an editable view pane.

Chapter 6

Possible Extensions

6.1 Preprocessor

A multipart extension idea would be to extend the web based uploader to handle multiple file uploads. This addition would allow the C style checker to be extended to have it's files run through the C preprocessor before parsing starts. This implementation will have the added benefit of no longer requiring a GLR parser for composite check. I made an initial attempt to implement preprocessing of the files before checking for errors. Running through the C preprocessor proved very useful in eliminating portions of the file which are difficult to parse without it. It can be implemented to replaces macro definitions and join broken strings and variables together. Original line position can be easily recalculated with the linemarkers the preprocessor inserts into the outfile. Where I ran into problems with implementation is when the C file included local header files. In order to implement the preprocessor portion a multiple file upload portion would be required.

6.2 Additional Productions

Continue constructing grammar productions to produce checks for additional style errors as they are identified.

6.3 Replace Indent and Vim

A program was identified that could potentially replace indent. The program *Astyle* on initial inspection appears to have the flexibility to be able to replace indent and possibly Vim's indentation capabilities as well.

6.4 Javascript

It may be possible to construct a more dynamic and user friendly web interface using jQuery or other javascript libraries that would provide the required functionality. Currently the line numbers displayed on the code view portion are loaded by counting the number of lines in the file and then running a for loop to print the numbers off. This means if the student inserts a newline into the text of their code file the line numbers do not dynamically adjust to match. The code view portion provides the ability to save the document after it has been edited, however, it is being saved as either a new file or opened in a text editor. It would be a beneficial feature if that when you pressed save it could save the changes to the original uploaded document. This may be asking to much of the system and the possibilities of accidentally deleting your original file might outweigh the benefits.

6.5 Comments Extension

The current version of the check comments module uses regular expression to parse header and function comments. The tokens the comments scanner provides should be separated out and the grammar file extended to parse the data more accurately. The comments portion could also be merged with the composite check module as the composite check is currently ignoring comments.

Chapter 7

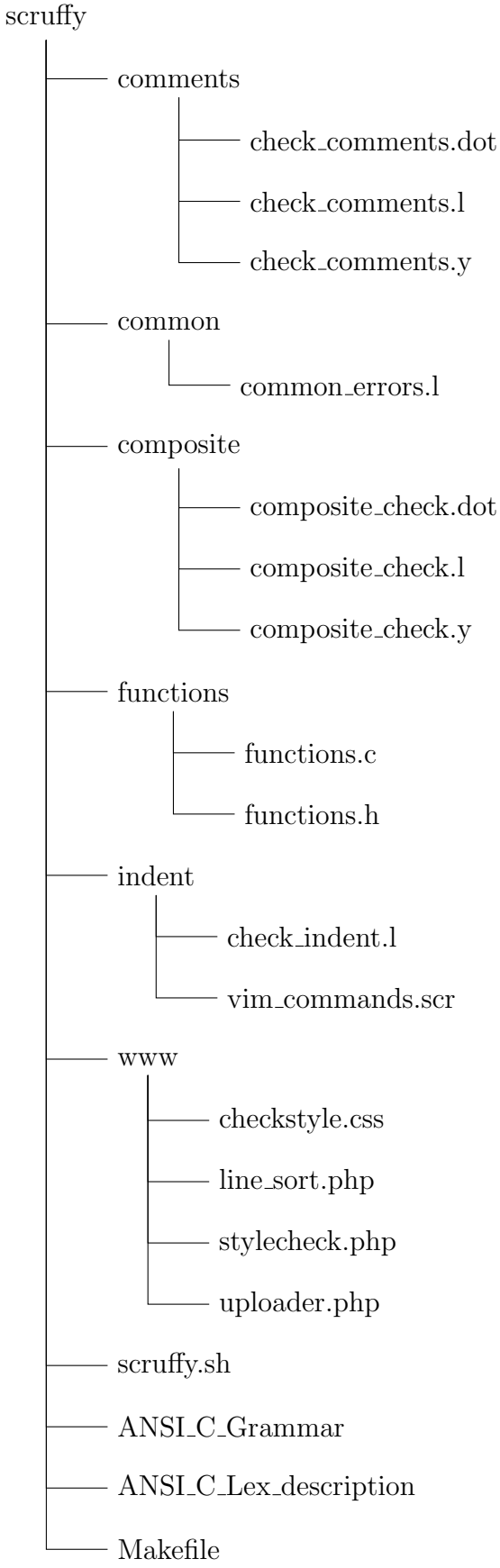
Summary and Conclusions

The main objective of the project was to provide a C style checker similar to checkstyle that students could use to verify their code before submission. The style checker has been in testing with Dr. Diamond's COMP 2103 students since September 29th 2013. Without the ability to log whether the program is being utilized it is currently undetermined if the project is performing as expected, however, TA's have reported fewer style errors recently. A long term goal for this project would be to continue logging the results of its use to see if the program aids in reducing the number of repetitive style errors of students. An additional objective of the project was to produce a flexible and extendable program, with the implementation of additional grammar productions as additional style errors are identified the program can be extended to incorporate the additional style errors.

In conclusion the C style checker will provide a method to enforce a uniform coding style throughout the COMP 2103 students which was previously unavailable. The C style checker has the potential to produce a reduction in the number of common style errors present in COMP 2103 student code. The additional ability to receive immediate corrective feedback may help to increase retention of the coding standards guide line.

Appendix A

File Listing


scruffy
comments
check_comments.dot
check_comments.l
check_comments.y
common
common_errors.l
composite
composite_check.dot
composite_check.l
composite_check.y
functions
functions.c
functions.h
indent
check_indent.l
vim_commands.scr
www
checkstyle.css
line_sort.php
stylecheck.php
uploader.php
scruffy.sh
ANSI_C_Grammar
ANSI_C_Lex_description
Makefile

Appendix B

User Manual

B.1 Usage

B.1.1 Command line

The command line usage of the components is simplified through the use of a shell script included with the program. usage: `sh scruffy.sh <filename>` where `<filename>` is the name of the file that should be inspected for errors.

B.1.2 Web based

The web based version is as simple as identifying a file to upload and clicking submit.

They C style checker performs best with C files, scanning files other than C code files is undocumented and unadvised.

Appendix C

C Coding Style Notes

This document gives the basic information about acceptable coding style for C programs in COMP 2103. Written by Jim Diamond, Acadia University Last updated October 2, 2013.

Header comment

Your program should start with a header comment similar to the following

```
/*
 * File:     A2P1.c
 * Author:   Jim Diamond   000123456
 * Date:     2011/09/12
 * Version:  1.0
 *
 * Purpose:
 * ...
 */
```

Note the following points:

(a) The '/*' and '*/' are on lines by themselves.

(b) There is (at least) one space between other '*'s and the following word(s), if any.

(c) You need ALL of the indicated items above, although, of course, you will use your own name and ID, and you will use appropriate values for the file name, the date, and the version.

(d) The purpose should give a reasonable concise description of the program. Not as much detail as the assignment question itself, but enough detail so that someone who knows nothing about the assignment or your program could get a very good idea of what the program's purpose is by reading the comment.

(e) The "File:" name is the actual name of the file. The example above is a C program file (as evidenced by the ".c"); a shell program would not normally have any file name extension.

Line length

Lines must not be longer than 79 characters. Should you need to break a "logical" line up into two or more "physical" lines in order to keep the line length from being too long,

(a) choose reasonable places in the line (from a syntactic point of view) to break the line, and

(b) indent the continuation lines an amount which makes it clear to the reader where the continued text "fits in".

For example:

```
if (some_long_variable_name != some_long_function_name(value)
    && short_var_nm == something_else)
{
    ...
}
```

Note that the "&&" is at the beginning of the continuation line, rather than at the end of the previous line, making it very obvious that that is a continuation line.

For functions with long argument lists, a common style (which I expect you to use) is to have the arguments on the second and subsequent lines lined up with the first argument on the first line.

```
printf("The number of snarfles is %d, %d, %d and sometimes %d\n",
       tuesday_snarfles, weekly_special_snarfles + 43,
       speciality_snarfles, other_snarfles);
```

Variable naming conventions

If you use any information that is a so-called "magic number", you should #define a macro whose name is composed of upper case letters and, optionally, the underscore character ('_'). For example,

```
#define   DAYS_IN_WEEK   7
```

"Ordinary" variables should have names composed of lower case letters and, optionally, digits and underscore. If you like variables where mixed case is used for readability, such as `numberOfEmployees` that is acceptable, but consider whether something like `number_of_employees` is more readable or not.

Variable declaration conventions

You may declare multiple variables on one line: int i, j, k; But if you initialize any variables in a declaration statement, the initialized variable must be the only declared variable in that statement. For example:

```
BAD   int i, j = 0, k;

GOOD  int i, k;
      int j = 0;
```

Indentation

Your code MUST be properly indented. I suggest (extremely strongly) that you use 4 spaces per indentation level. Further, 8 is really too many because with only a few levels of indentation, you may have difficulty writing lines that don't exceed 79 characters in length. Regardless of what you use, be consistent! Here is a sample of properly indented code:

```
if (answer == 3  ||  answer < 0)
{
    for (i = 0; i < n; i++)
    {
        switch (i % 2)
        {
          case 0:
            printf("%d is even\n", i);
            break;

          case 1:
            printf("%d is odd\n", i);
            break;

          default:
            printf("%d is extremely odd\n", i);
        }
    }
}
```

Note that the "case" labels are indented by only two spaces. This allows them to stick out noticeably without causing the code to get indented too far.

Tab characters

You may use tab characters to indent your code, but only if each tab character is equivalent to 8 characters. (More specifically, the character following a tab character is in column number 8n+1 for some n.)

This means that your indentation could consist of a mix of tab characters and spaces. But repeat after me: "Tab stops are every 8 spaces, as God intended."

Syntactic elements and line structure

At Acadia we expect C and Java programs to have the open brace ('{') of a code block or function/method definition on a line by itself. The close brace must also be on its own line, except in these cases

(a) do-while statements:

```
do
{
    ...
} while (a < b);
```

(b) typedef'ing a struct:

```
typedef struct
{
    ...
} mytype_T;
```

(c) combining a struct definition with a variable declaration:

```
struct mystruct
{
    ...
} ms1, ms2;
```

Finally, these rules don't apply to braces when used to initialize arrays, such as in `int time_units[] = {60, 60, 24, 365};`

Comments

Aside from the header comment above, you should use comments when a few words of wisdom would enlighten a reasonably competent programmer who is reading your code. If you have done something particularly clever or devious, explain it. Also, if you have done something which makes use of some fact(s) that even a reasonably competent programmer wouldn't know, then explain it. But never, ever use comments like this:

```
/* set i to 0 */
 i = 0;
```

If you have multi-line comments, make sure that all the '*'s of the same comment line up vertically.

Note: gcc (in C90 mode) allows you to use // comments, but this is not portable to all C90 compilers, so if you do this, your program won't be as portable as if you use /* ... */ comments. Since we are using the C99 standard, you are welcome to use // for comments, but be aware that if you use a non-gcc C90 compiler, you may have to edit your code.

White space

The keywords "if", "while", "for" and so on must be followed by a space, before the open parenthesis. See the sample code above in "(4) Indentation" for examples of this style.

Use extra white space when you think it will make a statement easier to read. For example, if you compare

```
if (answer==3||answer<0)
```

to

```
if (answer == 3 || answer < 0)
```

or even

```
if (answer == 3  ||  answer < 0)
```

you might agree the white space makes it easier to separate the tokens.

Use blank lines between (relatively) unrelated blocks of code. For example, if some function (main() or other) has to do three things, but each one is more or less independent of the others, then separate those blocks of code with a blank line. For example:

```
some code to do the first thing
in a sequence of things to do
for some program or function.

some code which does some stuff that is, by itself,
more or less independent of the stuff above.

yet another block of code which does
stuff not directly related to either
of the above two blocks of code, except, of course,
that all three blocks are used inside the same program.
```

Having said that:

(a) don't use white space between a function name and the '(':

```
z = sqrt(y);  // Not   z = sqrt (y);
```

(b) don't use white space after the ')' when casting:

```
i = (int)f;   // Not   i = (int) f;
```

Functions

Using functions allows you to manage the complexity of your program, making it easier to write in the first place, easier to test and debug, and quicker to finish.

The body of a function should be indented. For example:

```
int
my_func(...args...)
{
    some code;
    some code;
    some code;
    ...
}
```

I like putting the return type of a function on the previous line like that, because I can then do a regexp search for `^my_func` which will take me to the definition of the function, not some place where it is used. But putting the type of the function on the same line is fine if you want.

Function comments

A function should be preceded by a short comment describing the purpose, the inputs (if any), the outputs (if any), side effects (if any), and any other information which would allow a reader to quickly understand the purpose and use of the function. Here is a comment which contains all of the items that are normally relevant. If you have something that you think should be said which isn't in one of these categories, go ahead and add it.

```
/*
 * Name:         my_func
 * Purpose:      add up the sizes, in bytes, of all the files given
 *               as arguments
 * Arguments:    zero or more filenames
 * Output:       sum of the sizes of those files
 * Modifies:     writes sum of sizes to stdout
 * Returns:      0 on success, 1 if there were files whose sizes
 *               could not be determined
 * Assumptions: none
 * Bugs:         none
 * Notes:        any interesting comments about this function,
 *               whether they deal with the arguments, the
 *               algorithm employed, or the format of the output.
 */
```

Appendix D

Available Checks

The C style checker provides the following checks:

Array Trailing Comma

Checks if array initialization contains optional trailing comma.

Type Name

Checks that type names conform to a format specified by the format property.

Default Comes Last

Check that the `default` case is after all the cases in a switch statement.

Empty Block

Checks for empty blocks.

Empty Statement

Detects empty statements (standalone ';').

Fall Through

Checks for fall through in `switch` statements Finds locations where a case contains code — but lacks a `break`, `return`, or `continue` statement.

File Length

Checks for long source files.

File Tab Character

Checks to see if a file contains a tab character.

Generic White space

Checks that the white space around the generic tokens < and > are correct to the typical convention.

Indentation

Checks correct indentation of C Code.

Left Curly

Checks the placement of left curly braces on types, functions and other blocks.

Line Length

Checks for long lines.

Local Variable Name

Checks that local, variable names conform to a format specified by the format property.

Magic Number

Checks for magic numbers.

Method Name

Checks that method names conform to a format specified by the format property.

Method Param Pad

Checks the padding between the identifier of a method definition and the left parenthesis of the parameter list.

Missing Switch Default

Checks that switch statement has `default` case.

Need Braces

Checks for braces around code blocks.

One Statement Per Line

Checks there is only one statement per line.

Parameter Name

Checks that parameter names conform to a format specified by the format property.

Paren Pad

Checks the padding of parentheses; that is whether a space is required after a left parenthesis and before a right parenthesis.

Right Curly

Checks the placement of right curly braces.

Type Name

Checks that type names conform to a format specified by the format property.

Type cast Paren Pad

Checks the padding of parentheses for typecasts.

Header Comment

Detects files missing or improperly formatted header comments.

White space After

Checks that a token is followed by white space.

White space Around

Checks that a token is surrounded by white space.

Multiple variable declarations with initialization

Checks that an initialization of a variable is on its own line.

Function Comment format

Checks that functions preceded by a short comment describing the function are correctly formatted.

Appendix E

Required Software

Certain software and particular versions of some software are required to properly compile and run the C style checker. Keep in mind that this is not a complete list of software, and that other packages or dependencies may have to be installed.

Vim VIM - Vi IMproved - version 7.3.547

Flex flex 2.5.35

Bison bison (GNU Bison) 2.7.12-4996 (any version above 2.5).

Indent GNU indent 2.2.11

GCC gcc 4.7.2

diff diff (GNU diffutils) 3.2

Appendix F

cinoptions

The 'cinoptions' option sets how Vim performs indentation. In the list below, N represents a number of your choice (the number can be negative). When there is an 's' after the number, Vim multiplies the number by 'shiftwidth': 1s is 'shiftwidth' 2s is two times 'shiftwidth', etc. You can use a decimal point, too: "-0.5" is minus half a 'shiftwidth'. The examples below assume a 'shiftwidth' of 4.

>**N** Amount added for "normal" indent. Used after a line that should increase the indent (lines starting with "if", an opening brace, etc.) (default 'shiftwidth').

```
cino=            cino=>2          cino=>2s
  if (cond)        if (cond)        if (cond)
  {                {                {
      foo;             foo;             foo;
  }                }                }
```

eN Add N to the prevailing indent inside a set of braces if the opening brace at the End of the line (more precise: is not the first character in a line). This is useful if you want a different indent when the '{' is at the start of the line from when '{' is at the end of the line (default 0).

```
cino=             cino=e2          cino=e-2
  if (cond) {       if (cond) {      if (cond) {
      foo;                foo;         foo;
  }                 }                }
  else              else             else
  {                 {                {
      bar;              bar;             bar;
  }                 }                }
```

nN Add N to the prevailing indent for a statement after an "if", "while", etc., if it is NOT inside a set of braces. This is useful if you want a different indent when there is no '{' before the statement from when there is a '{' before it (default 0).

```
cino=           cino=n2           cino=n-2
  if (cond)       if (cond)         if (cond)
      foo;            foo;              foo;
  else              else              else
  {                 {                 {
      bar;              bar;              bar;
  }                 }                 }
```

fN Place the first opening brace of a function or other block in column N. This applies only for an opening brace that is not inside other braces and is at the start of the line. What comes after the brace is put relative to this brace (default 0).

```
cino=           cino=f.5s           cino=f1s
  func()          func()              func()
  {                 {                   {
      int foo;          int foo;            int foo;
```

{N Place opening braces N characters from the prevailing indent. This applies only for opening braces that are inside other braces (default 0).

```
cino=             cino={.5s             cino={1s
  if (cond)         if (cond)             if (cond)
  {                   {                     {
      foo;              foo;                  foo;
```

}N Place closing braces N characters from the matching opening brace (default 0).

```
cino=             cino={2,}-0.5s      cino=}2
  if (cond)         if (cond)           if (cond)
  {                   {                 {
      foo;              foo;                foo;
  }                 }                   }
```

^N Add N to the prevailing indent inside a set of braces if the opening brace is in column 0. This can specify a different indent for whole of a function (some may like to set it to a negative number) (default 0).

```
cino=                cino=^-2             cino=^-s
  func()               func()               func()
  {                    {                    {
      if (cond)          if (cond)          if (cond)
      {                  {                  {
          a = b;             a = b;             a = b;
      }                  }                  }
  }                    }                    }
```

LN Controls placement of jump labels. If N is negative, the label will be placed at column 1. If N is non-negative, the indent of the label will be the prevailing indent minus N (default -1).

```
cino=                cino=L2              cino=Ls
  func()               func()               func()
  {                    {                    {
      {                    {                    {
          stmt;                stmt;                stmt;
  LABEL:               LABEL:               LABEL:
      }                    }                    }
  }                    }                    }
```

:N Place case labels N characters from the indent of the switch() (default 'shiftwidth').

```
cino=                cino=:0
  switch (x)           switch(x)
  {                    {
      case 1:          case 1:
          a = b;           a = b;
      default:         default:
  }                    }
```

=N Place statements occurring after a case label N characters from the indent of the label (default 'shiftwidth').

```
cino=              cino==10
  case 11:           case 11:  a = a + 1;
      a = a + 1;                b = b + 1;
```

lN If N != 0 Vim will align with a case label instead of the statement after it in the same line.

```
cino=                      cino=l1
  switch (a) {               switch (a) {
      case 1: {                  case 1: {
                  break;                 break;
              }                          }
```

bN If N != 0 Vim will align a final "break" with the case label, so that case..break looks like a sort of block (default: 0). When using 1, consider adding "0=break" to cinkeys.

```
cino=                cino=b1
  switch (x)           switch(x)
  {                    {
      case 1:              case 1:
          a = b;               a = b;
          break;           break;

      default:             default:
          a = 0;               a = 0;
          break;           break;
  }                    }
```

gN Place C++ scope declarations N characters from the indent of the block they are in (default 'shiftwidth'). A scope declaration can be "public:", "protected:" or "private:".

```
cino=                cino=g0
  {                    {
      public:          public:
          a = b;           a = b;
      private:         private:
  }                    }
```

hN Place statements occurring after a C++ scope declaration N characters from the indent of the label (default 'shiftwidth').

```
cino=                cino=h10
   public:              public:   a = a + 1;
       a = a + 1;                b = b + 1;
```

pN Parameter declarations for K&R-style function declarations will be indented N characters from the margin (default 'shiftwidth').

```
cino=                cino=p0          cino=p2s
  func(a, b)           func(a, b)       func(a, b)
      int a;           int a;                   int a;
      char b;          char b;                  char b;
```

tN Indent a function return type declaration N characters from the margin (default 'shiftwidth').

```
cino=              cino=t0          cino=t7
       int           int                      int
  func()           func()           func()
```

iN Indent C++ base class declarations and constructor initializations, if they start in a new line (otherwise they are aligned at the right side of the ':') (default 'shiftwidth').

```
cino=                          cino=i0
  class MyClass :                class MyClass :
      public BaseClass           public BaseClass
  {}                             {}
  MyClass::MyClass() :           MyClass::MyClass() :
      BaseClass(3)               BaseClass(3)
  {}                             {}
```

+N Indent a continuation line (a line that spills onto the next) inside a function N additional characters (default 'shiftwidth'). Outside of a function, when the previous line ended in a backslash, the 2 * N is used.

```
cino=                    cino=+10
  a = b + 9 *              a = b + 9 *
      c;                           c;
```

cN Indent comment lines after the comment opener, when there is no other text with which to align, N characters from the comment opener (default 3).

```
cino=                  cino=c5
  /*                     /*
     text.                    text.
   */                     */
```

CN When N is non-zero, indent comment lines by the amount specified with the c flag above even if there is other text behind the comment opener (default 0).

```
cino=c0                cino=c0,C1
  /********             /********
    text.                text.
  ********/              ********/
```

(Example uses ":set comments & comments-=s1:/* comments^=s0:/*")

/N Indent comment lines N characters extra (default 0).

```
cino=                  cino=/4
  a = b;                a = b;
  /* comment */            /* comment */
  c = d;                c = d;
```

(N When in unclosed parentheses, indent N characters from the line with the unclosed parentheses. Add a 'shiftwidth' for every unclosed parentheses. When N is 0 or the unclosed parentheses is the first non-white character in its line, line up with the next non-white character after the unclosed parentheses (default 'shiftwidth' * 2).

```
cino=                       cino=(0
  if (c1 && (c2 ||            if (c1 && (c2 ||
             c3))                        c3))
      foo;                        foo;
  if (c1 &&                   if (c1 &&
        (c2 || c3))                 (c2 || c3))
     {                           {
```

uN Same as (N, but for one level deeper (default 'shiftwidth').

```
cino=                        cino=u2
  if (c123456789               if (c123456789
         && (c22345                       && (c22345
             || c3))                        || c3))
```

UN When N is non-zero, do not ignore the indenting specified by '(' or 'u' in case that the unclosed parentheses is the first non-white character in its line (default 0).

```
cino= or cino=(s      cino=(s,U1
  c = c1 &&             c = c1 &&
      (                     (
       c2 ||                     c2 ||
       c3                        c3
      ) && c4;              ) && c4;
```

wN When in unclosed parentheses and N is non-zero and either using "(0" or "u0", respectively, or using "U0" and the unclosed parentheses is the first non-white character in its line, line up with the character immediately after the unclosed parentheses rather than the first non-white character (default 0).

```
cino=(0                    cino=(0,w1
  if (   c1                  if (   c1
         && (   c2                 && (   c2
                || c3))                 || c3))
      foo;                       foo;
```

WN When in unclosed parentheses and N is non-zero and either using "(0" or "u0", respectively and the unclosed parentheses is the last non-white character in its line and it is not the closing parentheses, indent the following line N characters relative to the outer context (i.e. start of the line or the next unclosed parentheses) (default: 0).

```
cino=(0                      cino=(0,W4
  a_long_line(                 a_long_line(
              argument,            argument,
              argument);           argument);
  a_short_line(argument,     a_short_line(argument,
               argument);                 argument);
```

mN When N is non-zero, line up a line starting with a closing parentheses with the first character of the line with the matching opening parentheses (default 0).

```
cino=(s                 cino=(s,m1
  c = c1 && (             c = c1 && (
      c2 ||                   c2 ||
      c3                      c3
      ) && c4;              ) && c4;
  if (                    if (
      c1 && c2                c1 && c2
     )                      )
      foo;                    foo;
```

MN When N is non-zero, line up a line starting with a closing parentheses with the first character of the previous line (default 0).

```
cino=                    cino=M1
  if (cond1 &&             if (cond1 &&
         cond2                    cond2
     )                            )
```

)N Vim searches for unclosed parentheses at most N lines away. This limits the time needed to search for parentheses (default 20 lines).

***N** Vim searches for unclosed comments at most N lines away. This limits the time needed to search for the start of a comment (default 70 lines).

#N When N is non-zero recognize shell/Perl comments, starting with '#'. Default N is zero: don't recognizes '#' comments. Note that lines starting with '#' will still be seen as preprocessor lines.

The defaults, spelled out in full, are:

```
cinoptions=>s,e0,n0,f0,{0,}0,^0,L-1,:s,=s,l0,b0,gs,hs,ps,ts,is,+s,
        c3,C0,/0,(2s,us,U0,w0,W0,m0,j0,J0,)20,*70,#0
```

The options used to obtain correct indentation for '*scuffy*' are:

```
cinoptions=e0,n0,f0,{0,}0,:2,=2,l1,b0,t0,+4,c4,C1,(0,w1
```

Vim puts a line in column 1 if:

- It starts with '#' (preprocessor directives), if cinkeys contains '#'.
- It starts with a label (a keyword followed by ':', other than "case" and "default") and 'cinoptions' does not contain an 'L' entry with a positive value.
- Any combination of indentations causes the line to have less than 0 indentation.

Bibliography

[1] John V. Dempsey and Gregory C. Sales. Interactive instruction and feedback., 1993. ISSN 0-87778-260-1.

[2] Charles Donnelly and Richard Stallman. Bison. 1(1):294, 2009. ISSN 1-882114-45-0.

[3] Charles Donnelly and Richard Stallman. The yacc-compatible parser generator. GNU Public Licence, December 2013. URL `http://www.gnu.org/software/bison/manual/bison.pdf`.

[4] Flex. Fast lexical analyzer generator. Sourceforge, December 2013. URL `http://flex.sourceforge.net/`.

[5] Jeff Lee, Tom Stockfisch, and Jutta Degener. ANSI C lex specification and ANSI C yacc grammar description. Usenet, December 1985, 1987, 1995. URL `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`.

[6] John R. Levine. flex and bison. 1(1):294, 2009. ISSN 978-0-596-15597-1.

[7] Bram Moolenaar. Vim reference manual. 1(7.3):10, 2011.

[8] Carlo Wood, Joseph Arceneaux, Jim Kingdon, and David Ingamells. Indent - format c code. GNU Public Licence, December 2013. URL `http://www.gnu.org/software/indent/manual/indent.pdf`.