INDEPENDENT WORK REPORT, SPRING 2014

CritTer2: A New C Style Checking Tool

Author:

Alisa KROUTIKOVA

Advisor:

Dr. Robert DONDERO

May 6, 2014

This paper represents my own work in accordance with University regulations.

May 6, 2014 Alisa Kroutikova

CritTer2: A C Style Checking Tool

Alisa Kroutikova

advised by Dr. Robert Dondero

Abstract

Stylistic errors can be difficult to detect but detrimental to the readibility of the code. The current tool used by graders in Princeton University's COS 217: Introduction to Programming Systems to detect stylistic errors in C code submitted by students, CritTer, is fragile and can crash without warning or reason when encountering certain input files, threatening students' faith in the tool. With the help of CETUS, a source-to-source compiler infrastructure that includes a parse tree, I designed and implemented a new version of CritTer, CritTer2, which matched the performance of CritTer in detecting errors while remaining stable and correctly processing any valid C90 file. CritTer2 is also designed to be more easily customizable, with error checks easy to disable and re-enable.

1. Introduction

1.1. CritTer

One of the hardest habits to learn when first learning programming is writing stylistically good code. Badly written code can be difficult to read and debug, which means that teaching good code style is incredibly important. Unfortunately, manually grading code for stylistic flaws is tedious, repetitive and error-prone, as many stylistic flaws are small and easy to miss. Automating the task saves a great deal of time for instructors and provides students with a way to style-check their own code before submitting assignments.

To fill the void of customizable, automated C style checkers, Erin Rosenbaum created a tool called CritTer (Critique from the Terminal) in 2011, currently used in Princeton University's Introduction to Programming Systems (COS 217) [2]. Her tool succeeded in providing a radical improvement in detecting stylistic flaws in C code, with a recall of 98.1% and precision of 77.2% when tested [2]. However, CritTer has many issues. It was built using Flex as the lexical analyzer and Bison as a parser generator and written in fragile, difficult to understand code, making it not as customizable as it might need to be should course standards for COS 217 change. Making changes to the errors that CritTer detects runs a constant risk of breaking the tool entirely.

A larger problem with CritTer is that while it does correctly identify nearly all the stylistic flaws in input code, certain kinds of input can cause it to crash without warning or output bizarre and cryptic error messages. Because CritTer only analyzes user code and not standard header files, it fails to fully understand types defines in standard libraries, and therefore misses checks related to those types. Backslashes in user code, when used to continue a line of code onto another physical line, confuse CritTer's lexical parser, resulting in an error message and a failure to continue running checks on the rest of the code. Certain programs cause CritTer to fail with segmentation faults or asserts. Benjamin Parks, a former COS 217 student, discovered a valid, two-line program that causes CritTer to crash without any explanation. However, CritTer correctly runs on the same program if whitespace is added to the beginning of the second line.

Program handled by CritTer	**Program that crashes CritTer**
`int main(void){if((double)2>` `1){}return 1;}`	`int main(void){if((double)2>` `1){}return 1;}`

Table 1: The two-line program submitted by Benjamin Parks that causes CritTer to crash if there is no whitespace preceeding the second line.

When CritTer crashes without any obvious reason and instructors have no way of explaining to students why this code causes such a reaction from CritTer, students lose faith in CritTer as an accurate tool for detecting stylistic flaws. This discourages students from using CritTer, which diminishes its value as an instructional tool.

1.2. CETUS

Dr. Dondero, through work with another student, Andrew Morrison, and through discussions with Prof. David Walker, discovered the existence of a C compiler infrastructure that included an extensive parse tree framework, called CETUS [1]. CETUS reads C90 source code and generates a parse tree that is easily traversable, with nodes representing different parts of the program. If a new version of CritTer were to be designed using this more robust framework that correctly parsed all valid C90 programs and provided simple ways to traverse the C program in Java code, this could solve the current problems of CritTer. This was what Dr. Dondero and I set out to explore: the possibility of creating a new version of CritTer, designed in a completely new fashion using CETUS, that would match the performance of CritTer without its current fragility.

2. Functionality

After a semester of work, I succeeded in implementing CritTer2 to have the same functionality as the original CritTer. It is used in much the same way as CritTer and contains the same checks as the original CritTer, with some minor differences. The following is an in-depth analysis of the functionality of CritTer2, particularly as it relates to that of the original CritTer.

2.1. Users

CritTer2 should be used in much the same fashion as the original CritTer from the user side. It runs through a command from a Unix shell. After following installation instructions from the administrators, go to the working directory from the command line and type "`critTer2 *.c`" to check all the C source code in the directory or specify a particular filename in place of the "`*.c`".

2.2. Administrators

Broadly, the use of CritTer2 for administrators is essentially the same as that of the original CritTer in that it should be used as a grading aid for academic purposes, reducing the time it takes to grade student code by catching stylistic errors. However, CritTer2 expands upon CritTer's adaptability

and customization aspects. Administrators can very easily change the code of CritTer2 (located in the file `Critter.java`) to change the checks that CritTer2 runs. Checks that specify minimum or maximum lengths or counts of parts of code can be customized by changing the variables that define those minimum or maximum lengths or counts in the beginning of the file. Furthermore, existing checks can be easily disabled and reenabled. The `main()` function of `Critter.java` calls each check separately, so an administrator can easily choose to comment out a check to disable it. Finally, new checks can be easily added by writing new functions, allowing new course standards to be introduced.

2.3. Comparison to Original CritTer

Overall, the functionality of CritTer2 is the same as that of the original CritTer. Table 2 details all the stylistic flaws that CritTer and CritTer2 catch. One important difference is that two checks in the original CritTer were merged into one in CriTer2: it was only possible to catch CritTer's check `useEnumNotDefine` through the check `checkMagicNumbers` because of the way the parsing functions in CETUS operated. CETUS begins by calling a C preprocessor on the input, which implements `#define`. This means that macros are not present in the CETUS parse tree, having all been replaced by what appear to be hard-coded numbers, making it impossible to distinguish magic numbers and macros.

There are also some minor differences in how CritTer2 handles certain stylistic flaws, specifically those that involve comments. For the check `structFieldsHaveComments`, CritTer was satisfied if there were the same number of comments inside a struct as fields, whereas CritTer2 requires each field to have a comment on the line preceding it. Similarly, the checks `globalHasComment` and `isFunctionCommentValid` in CritTer can check comments that consist of multiple comment blocks but requires one of those blocks to be no more than two lines above the line of code it refers to, whereas CritTer2 can only check one block of comments but will accept any comment that precedes a line of code, no matter how many blank lines are in between them.

A final difference between the functionality of CritTer and CritTer2 is that there were cer-

Check in CritTer	Corresponding Check in CritTer2	Purpose
isFileTooLong	checkFileLength	Warns if file exceeds a maximum length
isFunctionTooLong-ByLines	checkFunctionLength-ByLines	Warns if function exceeds a linecount
tooManyParameters	checkFunctionParams	Warns if function has too many parameters
switchHasDefault	checkSwitchHasDefault	Warns if a switch is missing a default case
switchCasesHaveBreaks	checkSwitchCases	Warns if a switch case is missing a break statement
isTooDeeplyNested	checkNesting	Warns if a region of code is too deeply nested
useEnumNotDefine	checkMagicNumbers	Warns against using #define in code
isMagicNumber	checkMagicNumbers	Warns against using magic numbers outside declarations
neverUseGotos	checkGoTos	Warns against using goto statements
isVariableNameTooShort	checkVariableName	Warns if variable names are too short
globalHasComment	checkGlobalHas-Comment	Warns if global variables are missing comments
isLoopTooLong	checkLoop	Warns if loop exceeds a maximum length
isCompoundStatement-Empty	checkEmptyCompound	Warns against empty compound statements
tooManyFunctionsInFile	checkFunctionNumber	Warns if number of functions in a file exceeds maximum number
isFunctionCommentValid	checkFunctionComment-Valid	Warns if function comment fails to mention each parameter by name or explain what the function returns
arePointerParameters-Validated	checkAsserts	Warns if pointer parameters not validated by an assert
doFunctionsHave-CommonPrefix	checkFunctionNaming	Warns if not all functions have a common prefix [1]
functionHasEnough-LocalComments	checkFunctionHas-EnoughComments	Warns if functions don't have enough local comments
structFieldsHave-Comments	checkStructHas-Comment	Warns if a field in a struct lacks comments
hasComment	checkBeginning-Comment	Warns if there is no comment in the beginning of each file

Table 2: A comparison of the checks in CritTer and CritTer2.

tain checks deemed unnecessary to implement in CritTer2. These checks are `hasBraces`, `isFunctionTooLongByStatements`, `neverUseCPlusPlusComments`, and

[1] Princeton University's COS 217 requires all functions in non-main modules to have a common prefix, separated by an underscore. CritTer2 reflects this by specifically looking for prefixes in function names that are separated by an underscore and comparing them to ensure they are the same.

`isIfElsePlacementValid`. The check `neverUseCPlusPlusComments` was removed because it is not a stylistic check, as a C90 program with C++ comments would fail to compile. The checks `hasBraces` and `isIfElsePlacementValid`, both referring to stylistic rules of if/else statements, were deemed unnecessary after discussion with Dr. Robert Dondero, because they enforced unnecessarily strict guidelines on if/else statements. Finally, the check `isFunctionTooLongByStatements` was removed because it was redundant (in nearly every case, if a function is too long by statements it will also be too long by lines) and therefore not worth the complicated code required to implement it.

3. Design of CritTer2

Although CritTer2 implements the same checks that CritTer does, and is created to be used in the same fashion, the fundamental design of CritTer2 is radically different from that of the original CritTer, which handled parsing the given C90 code itself. CritTer2 calls CETUS to build a parse tree and traverses that parse tree to look for stylistic flaws.

3.1. CETUS

CETUS is a complier infrastructure for the source to source transformation of C programs. It is publicly available on the internet. It currently only supports the ANSI C89/ISO C90 standard, which is the standard used in Princeton University's Introduction to Programming Systems (COS 217). However, should the course structure ever change to include the ANSI/ISO C99 standard, CritTer2 would no longer be able to parse student submissions. However, this is also a limitation of the original CritTer, which can also only parse C90 code.

The primary function of CETUS is to allow optimizations of parallel programs written in C [1]. Although the input to CETUS must be C source code, CETUS itself is written in Java. CETUS was appealing for use in creating a C style-checking tool because it first creates a parse tree of the given C code, which I could then traverse to find particular regions of code to analyze. For example, the implementation of the check `checkFunctionParams`, which determines if a function has too many parameters, involves traversing the CETUS parse tree and closely examining the

nodes corresponding to the function definitions. CETUS has classes describing many types of regions of code, which allowed me to choose to either examine or ignore certain types of code in my traversals. Some of these classes most frequently used in CritTer2 are `ForLoop`, representing a `for` loop, `Procedure`, representing a function, and `VariableDeclaration`, representing a variable declaration.

3.2. CritTer2 Code

CritTer2 is structured using two files: a `Critter.java` file (see Appendix A) that includes all the checks and writes all the warning messages, and the `critTer2` Python script (see Appendix B) that prepares the input files to be sent to `Critter.java`.

3.2.1. Critter.java The checks that CritTer runs are all in one file, `Critter.java`. Each check is a separate function that is called by the `main()`, which enables administrators to easily deactivate and reactivate checks. Roughly, each check is structured in the same way: through a traversal of the parse tree representing the inputted code. Each traversal is handled in a way unique to the type of check, with some traversals focusing on just certain types of nodes or skipping over certain types of nodes. Some checks involve constants that can be set by the administrator by changing the global variables pertaining to those constants, such as the maximum allowed length of a file, or the minimum variable name allowed.

`Critter.java` calls on classes and functions inside of CETUS from the same position as the highest level main module of CETUS, `Driver.java`. I replaced `Driver.java` with my own version, `Critter.java`, which runs the checks on the input. Because `Critter.java` is at the position of the highest level module, I have access to all of the features of CETUS. This allows my code to fully explore the parse tree with the help of built-in tree iterators that can be specified to iterate only over certain types of nodes, shortening the time it takes for CritTer2 to run. However, despite this, CritTer2 still runs slower than the original CritTer because it encompasses so many tree traversals. The longer runtime is not significant enough to be an impediment when run on just a few C files in a directory, however.

One of the more difficult checks to implement in `Critter.java` was the nesting depth check, which is designed to analyze the depth of each region of code by traversing back through the parse tree to count the number of parents that are loops or bodies of if/else statements. However, because CETUS considers the parent of an else clause to be the if clause it is a part of, simply counting the number of parents that are loops or if statements would result in a large number of false reports of nesting depth. A series of if/else if/else if clauses are not necessarily nesting within each other, as demonstrated in Table 3. However CETUS views both as the same in its parse tree. Because the stylistically prefered version, shown on the left of Table 3, is used much more often than the flawed version on the right, I chose to design CritTer2 to not count if statements whose parent is an else statement as a level of nesting. This does mean that CritTer2 can miss some regions of deep nesting. However, cases when students nest an if statement inside an else statement are usually symptomatic of heavy nesting throughout that region of code, so in most cases there will also be deep nesting a level above the nested if statement, which CritTer2 will report correctly. The original CritTer did not have this problem because it was designed to keep track of the number of unclosed braces at any given point in the code. This also means that CritTer is able to suppress reports of deep nesting within a region of code that has already been reported to be nested too deep, thereby limiting the number of deep nesting warnings generated. Implementing that in CritTer2 runs the risk of skipping over too many nodes in the parse tree and generating too few warnings, so I chose to allow CritTer2 to report every instance of nesting depth it comes across. This difference in design accounts for many discrepancies in the number of stylistic flaws that CritTer reports and the number that CritTer2 reports.

3.2.2. CritTer2 Script CritTer2 is run through a Python script, heavily based off of one created by Andrew Morrison, that prepares inputted files before sending them through `Critter.java`. One of the limitations of CETUS is that it fails to preserve original line numbers for the code. Since original line numbers are so important to pinpoint the location of stylistic errors, this was a major issue that needed to be addressed. This was done with the help of Morrison, who was also working with CETUS, and suggested that the easiest way to preserve line numbers in files was to run through

```
if (i > x) {
      x++;
}
else if (i > y) {
      y++;
}
```

```
if (i > x) {
     x++;
}
else {
     if (i > y) {
          y++;
     }
}
```

Table 3: Different nesting depths of code that are indistinguishable in the CETUS parse tree.

the inputted .c files and annotate them, having each line be preceded with a pragma annotation revealing both the line number and the file name, as so: `#pragma critTer:1:hello.c:`. Because CritTer2 also runs checks on header files included in the .c files, the same annotations were added to all the .h files in the working directory. The annotated copies of all the files are placed in a temporary directory to avoid disrupting the original files, and only then is `Critter.java` run on the annotated files. When a warning message is printed to standard error, the line number and file name corresponding to the location of the stylistic flaw are taken from the pragma annotation immediately preceding the flawed line of code.

The CritTer2 script also uses pragmas to annotate the beginning and end of all included files it encounters (in the given .c files and in all the .h files in the working directory). This was necessary because when the source code is run through CETUS, the code of the included files is automatically inserted in the appropriate place of the parse tree, making it impossible to tell if a node in the tree is from the original file, a standard header file, or a user header file. Because CritTer2 is designed to run checks on only user-created header files, I had to make some distinction between the two. Therefore, whenever the CritTer2 script encounters a standard include, it adds an annotation right before, of the form `#pragma critTer:startStdInclude:`, and an annotation right after, of the form `#pragma critTer:endStdInclude:`, which allows the check functions in `Critter.java` to skip over them when traversing the CETUS parse tree. To allow CritTer2 to run checks on user-created header files, the CritTer2 script adds similar annotations before and after included user-created header files, of the form `#pragma critTer:startStudentInclude:` and `#pragma critTer:endStudentInclude:`.

4. Testing

Testing of CritTer2 was done in three parts. The first part consisted of testing each stylistic flaw one by one by writing small C90 programs, each containing a single flaw to test a specific check in CritTer2. The second part of testing was running CritTer2 on one long program that had every possible error that CritTer2 detects. The last part of testing was a comparison between the performance of the original CritTer and CritTer2, the goal of which was ensuring that CritTer2 performed at least as well as, if not better than, the original CritTer.

4.1. Testing Parts One and Two: Stylistic Flaw Catching

To test that CritTer2 finds every stylistic flaw it checks for, if that flaw is there, I created small C90 test files that individually tested each check contained in CritTer2. Running each program through CritTer2 revealed that every error was caught and reported correctly. For the next part of testing, I took an flaw-free submission for the final assignment, a Unix shell, of Princeton University's Introduction to Programming Systems (COS 217) course, and introduced every possible stylistic flaw into the files. The flaws introduced into the main module of the shell were:

- File exceeds the maximum length
- Function exceeds a maximum line count
- Function has too many parameters
- Switch statement without a default case
- Switch case without a break statement
- Region of code with deep nesting
- Variable definition using `#define`
- Magic numbers used outside declarations
- Goto statement used
- Variable names too short
- Global variable lacks a comment
- Loop exceeds a maximum length

- Empty compound statements included
- Number of functions in a file exceeds the maximum number allowed
- Function fails to have a comment
- Function comment fails to mention parameter by name and fails to state what the function returns
- Pointer parameter not validated by an assert
- Functions have too few local comments proportional to statements in function
- Field in a struct lacks comments
- No beginning comment

In addition, another flaw-free file that was not the main module of the program was used to test the check `checkFunctionNaming` by introducing a function whose name did not start with the same prefix as all the other functions.

Upon running CritTer2 on both of these files, the output did in fact show that CritTer2 caught every one of those introduced flaws, accurately documenting their location in the file.

4.2. Testing Part Three: Comparison with CritTer

To compare the performance of CritTer2 with the performance of the original CritTer, I examined the final assignment submissions from the Fall 2013 semester of Princeton University's Introduction to Programming Systems. All the submissions were anonymized before used in the testing. Out of a total of 219 submissions, I found six that had a large number of flaws when they were run through the original CritTer, and four that caused the original CritTer to crash, and ran them against CritTer2 to compare outputs. The following is an analysis of the warnings generated for five of the test files, with a side-by-side comparison of the output of CritTer and CritTer2.

4.2.1. Test File 1 CritTer reported twelve stylistic flaws in this test file. CritTer2 reported eighteen stylistic flaws.

CritTer error	CritTer2 error
command.c:80.7-80.17: medium priority: Do you want to validate 'pcNewStdin' through an assert? command.c:84.7-84.18: medium priority: Do you want to validate 'pcNewStdout' through an assert?	command.c: line 68: medium priority: Do you want to validate 'pcNewStdin' through an assert? command.c: line 68: medium priority: Do you want to validate 'pcNewStdout' through an assert?

Both CritTer and CritTer2 catch the failure to use an assert in the test file. The discrepancy in line numbers only indicates a difference in the definition of where the stylistic flaw occurs. The original CritTer reports the line number of where it would expect there to be an assert, while CritTer2 reports the line number of the parameter declaration.

CritTer error	CritTer2 error
command.c:167.13-169.14: low priority: This area is deeply nested, consider refactoring command.c:243.13-245.14: low priority: This area is deeply nested, consider refactoring	command.c: line 167: low priority: This area is deeply nested at level 4, consider refactoring command.c: line 243: low priority: This area is deeply nested at level 5, consider refactoring

CritTer threw 4 additional deep nesting warnings, and CritTer2 threw 10 additional deep nesting warnings. As described previously in Section 3.2.1, the deep nesting check is implemented very differently in CritTer2 than in CritTer, which accounts for the discrepancy. For this program, both CritTer and CritTer2 accurately report the nesting depth flaws.

In the nesting depth check, CritTer2 has the additional functionality of reporting how deep the nesting is.

CritTer error	CritTer2 error
command.c:149.4-275.5: low priority: A loop should consists of fewer than 35 lines; this loop consists of 127 lines; consider refactoring	command.c: line 150: low priority: A loop should consist of fewer than 35 lines; this loop consists of 130 lines; consider refactoring

In the loop length check, both CritTer2 and CritTer correctly generate the warning, differing only

in their definition of where the loop starts (either at the line where the opening brace begins or at the first line of code inside the loop).

CritTer error	CritTer2 error
command.c:94.1-279.2: low priority: A function definition should consist of fewer than 140 lines; this function definition consists of 186 lines command.c:94.1-279.2: low priority: A function definition should consist of fewer than 50 statements; this function definition consists of 79 statements	command.c: line 95: low priority: A function should consist of fewer than 140 lines; this function consists of 183 lines; consider refactoring

The redundant check of the function length by statements is not present in CritTer2. Otherwise, CritTer2 and CritTer both correctly perceive the length of the function, differing only slightly in reporting the length. CritTer reports the length of the function counting the function declaration and the starting and ending braces, whereas CritTer2 reports the length of the body of the function.

CritTer error	CritTer2 error
command.c:94.1-279.2: low priority: This function definition probably needs more local comments	command.c: line 95: low priority: This function definition probably needs more local comments

Both CritTer and CritTer2 correctly report the flaw of too few local comments in a function.

CritTer error	CritTer2 error
	command.c: line 95: medium priority: A function's prefix should match the module name; Command and CommandConvert do not match

Here, CritTer fails to report a flaw in function naming that CritTer2 catches. The style guidelines enforced by CritTer and used in COS 217 require function prefixes to be separates by underscores. In this function, the underscore is missing, which is overlooked by CritTer but caught by CritTer2.

4.2.2. Test File 2 CritTer detected four stylistic flaws in this test file before encountering a vald C90 construct that it failed to parse. CritTer reported this as a syntax error and ceased parsing the file. In contrast, CritTer2 had no problems parsing this file and reported six stylistic flaws.

CritTer error	CritTer2 error
symtablehash.c:35.57-35.66: high priority: syntax error, unexpected TYPE_NAME, expecting '{'. Please make sure that there aren't any '\'s in your code.	symtablehash.c: line 17: high priority: A comment should appear above each global variable. symtablehash.c: line 18: medium priority: A comment should appear above each field in a struct. symtablehash.c: line 26: medium priority: A comment should appear above each field in a struct.

The syntax error thrown by CritTer indicates the confusing error messages that the original CritTer can output while failing to properly check the style of the rest of the code, which results in a failure to report the three errors that CritTer2 catches: a global variable without a comment, and two struct fields without comments.

CritTer error	CritTer2 error
symtablehash.c:93.22-93.29: medium priority: Do you want to validate 'pvValue' through an assert?	symtablehash.c: line 81: medium priority: Do you want to validate 'pvValue' through an assert?

Both CritTer and CritTer2 throw two more similar assert warnings here. As before, both CritTer and CritTer2 correctly report the lack of validation of pointer parameters through asserts.

4.2.3. Test File 3 This test file had fifteen warnings genereated by CritTer and eleven warnings generated by CritTer2.

CritTer error	CritTer2 error
ish.c:246.13-250.14: low priority: This area is deeply nested, consider refactoring	

CritTer generated seven additional deep nesting warnings, whereas CritTer2 generated none. However, the deep nesting warning is a subjective test, and as before, since CritTer and CritTer2 have different definitions of nesting depth, this discrepancy was deemed acceptable.

CritTer error	CritTer2 error
ish.c:128.1-318.2: low priority: A function definition should consist of fewer than 140 lines; this function definition consists of 191 lines ish.c:367.1-533.2: low priority: A function definition should consist of fewer than 140 lines; this function definition consists of 167 lines	ish.c: line 129: low priority: A function should consist of fewer than 140 lines; this function consists of 188 lines; consider refactoring ish.c: line 368: low priority: A function should consist of fewer than 140 lines; this function consists of 163 lines; consider refactoring

Both CritTer and CritTer2 correctly report functions that exceed a maximum length, as before.

CritTer error	CritTer2 error
ish.c:450.7-487.8: low priority: A loop should consists of fewer than 35 lines; this loop consists of 38 lines; consider refactoring ish.c:492.4-529.5: low priority: A loop should consists of fewer than 35 lines; this loop consists of 38 lines; consider refactoring	ish.c: line 451: low priority: A loop should consist of fewer than 35 lines; this loop consists of 38 lines; consider refactoring ish.c: line 493: low priority: A loop should consist of fewer than 35 lines; this loop consists of 38 lines; consider refactoring

Both CritTer and CritTer2 correctly report loops that exceed a maximum length, with the discrepancy in flaw location explained by different means of determining the beginning of a loop (either with the opening brace or the first line of code in the body of the loop).

CritTer error	CritTer2 error

`ish.c:534.2-534.3: low priority:`
`A source code file should contain`
`fewer than 500 lines;`
`this file contains 534 lines`
	`ish.c: low priority:`
	`A source code file should contain`
	`fewer than 500 lines;`
	`this file contains 533 lines`

Both CritTer and CritTer2 correctly report that the file is too long.

CritTer error	CritTer2 error

	`ish.c: line 332: high priority:`
	`Use of magic number (3), which should`
	`be given a meaningful name, or a`
	`#define, which should be replaced`
	`with an enum (unless it's the result of`
	`a #define in a standard C header file)`

CritTer generates no warnings here, but CritTer2 generateds five more similar warnings. One of the limitations in using CETUS to aid in style checking is that it calls a preprocessor that automatically replaces macros with their corresponding value throughout the code, including if the macro is defined in a standard C header file, resulting in warnings like the one shown here. However, the warning message clearly explains this, which will avoid confusion among users. Should a user see this warning message but check for the number 3 in his or her code only to discover that line of code uses a variable defined in a standard C header file, the reason behind this warning message becomes clear.

4.2.4. Test File 4 This test file has thirty-six warnings generated by CritTer and seventy-seven warnings generated by CritTer2.

CritTer error	CritTer2 error

`ish.c:77.4-77.68: low priority:`
`When using braces with an if`
`statement, use multiple lines`	

CritTer generates fourteen additional warnings to use multiple lines when using braces with an if statement. CritTer2 does not generate any. Upon discussion with Dr. Robert Dondero, CritTer2's failure to even look for this type of stylistic flaw was deemed acceptable because placing braces on

multiple lines does not always necessarily increase the readability of code, and the lines of code that CritTer questioned were deemed stylistically acceptable by Dr. Dondero.

CritTer error	CritTer2 error

`ish.c:203.55-205.19: low priority:`
`This area is deeply nested, consider`
`refactoring`
	`ish.c: line 205: low priority:`
	`This area is deeply nested at`
	`level 5, consider refactoring`

CritTer generates sixteen more warnings of deep nesting, while CritTer2 generates sixty-five additional warnings of deep nesting, ranging in depth from 4 to 8. This discrepancy can be explained by the different methods the two programs use to determine nesting depth.

CritTer error	CritTer2 error

`ish.c:159.12-366.13: low priority:`
`A loop should consists of fewer than`
`35 lines;`
`this loop consists of 208 lines;`
	`ish.c: line 160: low priority:`
	`A loop should consist of fewer than`
	`35 lines;`
	` this loop consists of 209 lines;`
	`consider refactoring`

Both CritTer and CritTer2 correctly determine that the loop is too long.

CritTer error	CritTer2 error

`ish.c:113.1-556.2: low priority:`
`A function definition should consist`
`of fewer than 140 lines;`
`this function definition consists of`
`444 lines`
	`ish.c: line 114: low priority:`
	`A function should consist of fewer`
	`than 140 lines;`
	`this function consists of 436 lines;`
	`consider refactoring`

Both CritTer and CritTer2 correctly determine that the function is too long, with the discrepancies in exactly length corresponding to the different ways that CritTer and CritTer2 define the start and end of the function (CritTer includes the function definition and the start and end braces, CritTer2 includes only the body of the function).

CritTer error	CritTer2 error

`ish.c:113.1-556.2: low priority:`
`This function definition probably`
`needs more local comments`
	`ish.c: line 114: low priority:`
	`This function definition probably`
	`needs more local comments`

Both CritTer and CritTer2 correctly report that the function has too few local comments.

CritTer error	CritTer2 error

`ish.c:557.2-557.3: low priority:`
`A source code file should contain`
`fewer than 500 lines;`
`this file contains 557 lines`
	`ish.c: low priority:`
	`A source code file should contain`
	`fewer than 500 lines;`
	`this file contains 556 lines`

Both CritTer and CritTer2 correctly report that the file is too long.

CritTer error	CritTer2 error

	`ish.c: line 76: high priority:`
	`Use of magic number (3), which should`
	`be given a meaningful name, or a`
	`#define, which should be replaced`
	`with an enum (unless it's the result of`
	`a #define in a standard C header file)`

CritTer generates no such warnings, but CritTer2 generates seven similar warnings. As in Test File 3, these warnings reflect variables defined using `#define` in standard header files.

4.2.5. Test File 5 This test file cause CritTer to generate seven warnings before aborting without explanation. CritTer2 reported seventeen stylistic flaws in this test file.

CritTer error	CritTer2 error

`synparser.c:40.9-40.15: medium priority:`
`Do you want to validate 'pcName'`
`through an assert?`
	`synparser.c: line 31: medium priority:`
	`Do you want to validate 'pcName'`
	`through an assert?`

Both CritTer and CritTer2 also caught five similar assert flaws, all correctly reported.

CritTer error	CritTer2 error
`synparser.c:173.27-177.17: low priority: This area is deeply nested, consider refactoring`	`synparser.c: line 173: low priority: This area is deeply nested at level 4, consider refactoring`

CritTer found only one deep nesting flaw while CritTer2 found three more. However, because the depth of nesting is a subjective check, both programs were correct in reporting these flaws.

CritTer error	CritTer2 error
`critTer: dynarray.c:228: DynArray_removeAt: Assertion 'iIndex < oDynArray->iLength' failed. Aborted`	

Here, CritTer terminated with an abort, failing to detect any further stylistic flaws. However, CritTer2 correctly continued to generate seven more warnings.

CritTer error	CritTer2 error

	`synparser.c: line 31: low priority: This function definition probably needs more local comments`
	
	`synparser.c: line 144: low priority: This function definition probably needs more local comments`

Both areas in the code were low on comments, as CritTer2 correctly reported.

CritTer error	CritTer2 error
	`synparser.c: line 162: low priority: A loop should consist of fewer than 35 lines; this loop consists of 54 lines; consider refactoring`

CritTer2 correctly determined that this file included a loop that was much longer than the allowed maximum loop length.

CritTer error	CritTer2 error
	`synparser.c: line 77: medium priority: A function's prefix should match the module name; newCommand and freeArguments do not match`

CritTer2 found three other function naming flaws, which were all reported correctly.

4.3. Summary

CritTer2 and CritTer most frequently disagreed in reporting of subjective stylistic flaws such as nesting depth and amount of comments, which were deemed to be acceptable discrepancies since they are so easily amended with minor changes to CriTer2's code. If an administrator decides that CritTer2 outputs too many nesting depth flaws, for example, raising the allowed nesting depth would alleviate this problem. There were several instances in which CritTer2 outperformed the original CritTer, either by not crashing when encountering unusual input or by picking up on stylistic flaws that the original CritTer failed to catch. When CritTer was run against 219 final assignment submissions, it crashed with a segmentation fault in two cases and crashed with an assert in one case, not counting the times it outputed confusing syntax error messages because it was unable to deal with the input file. However, when CritTer2 was run against the 219 final assignment submissions, none of them caused CritTer2 to crash, which means that CritTer2 achieves the goal it was set out to accomplish: match the performance of the original CritTer without risk of crashing.

The biggest instance of unnecessary warnings generated by CritTer2 was the checkMagicNumbers warnings, which interpreted use of variables defined in standard header files using `#define` as magic numbers. However, these warnings are easily understood by the user when examining the location of the stylistic flaw. As detecting magic numbers when they are used is an important function of the original CritTer, and there is no way of detecting magic numbers in CritTer2 without throwing these unnecessary warnings, this check remains in place. However, because CritTer2 is so easily customizable, an administrator can easily choose to disable this check.

5. Evaluation

When the original CritTer was created, its performance was evaluated through a rigorous comparison of CritTer output with the results of manual style-checking done by Dr. Dondero [2]. The original CritTer was found to have a recall of 98.1% and precision of 77.2%, compared to Dr. Dondero's recall of 83.6% [2]. If the new version of CritTer, CritTer2, can match the performance of CritTer, then CritTer2's performance has already been evaluated. When testing CritTer2, it was determined that CritTer2 did in fact match the performance of the original CritTer, even outperforming CritTer in some instances. Since CritTer2 does at least as well as the original CritTer, there was no need to evaluate CritTer2's perfomance compared to manual style-checking.

6. Conclusion

After a semester of intense exploration of the inner workings of the CETUS parse tree, I emerged with a working tool: a new and improved version of CritTer that matched the performance of the original while using a much more stable framework that allowed CritTer2 to correctly report the stylistic flaws of all valid C90 code. CritTer2 is a success, having been tested against CritTer on real student code as well as fabricated error-filled programs. CritTer2 is a tool that can now be used instructionally, and adapted to fit different course standards as courses evolve.

There are still some limitations to CritTer2, however. CETUS can only parse C90 programs and not C99, which limits the type of courses that CritTer2 could be used in. There were small stylistic flaws that were impossible to detect because of the way CETUS handled input files, such as errors in the spacing of if-then statements with braces and the difference between a magic number error and a variable defined using `#define`. These flaws were deemed to be unimportant for reasons discussed earlier, but they are still flaws in the tool that perhaps could be fixed in future work.

However, this new version of CritTer is robust enough to be used not just in Princeton University's Introduction to Programming Systems, but even outside academia. Companies in industry could enforce a coding style within the company with the help of CritTer2, encouraging well-written and readable code, thanks to the customizability of the tool.

7. Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Robert Dondero. I would not have been able to accomplish any of this without his constant guidance at our weekly meetings and his tireless encouragement and support in this endeavor. I would also like to thank my Dad for answering questions I thought were too stupid to ask anyone else, and for teaching me everything I needed to know to start this project. I would also like to thank Andrew Morrison for giving me an introduction to the confusing and poorly documented world that is CETUS. I also could not have accomplished this semester of work without support from my friends and family. A special thank you goes to my roommate, Sydney Montgomery, for putting up with my late hours typing away in our room, and for supporting me in anything I take on. And of course I would like to thank my parents for always pushing me to do my very best and be my very best.

References

[1] Chirag Dave Sang-Ik Lee Seyong Lee Hansang Bae Leonardo Bachega Chirag Dave Sang-Ik Lee Seyong Lee Seung-Jai Min Rudolf Eigenmann Hansang Bae, Leonardo Bachega and Samuel Midki. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42(12), 2009.

[2] Erin Rosenbaum. Critter: Critique from the terminal: Customizable style checking for c programs, 2011.

Appendices

A. Critter.java

```
/*
 * Critter.java
 *
 * Created by Alice Kroutikova '15, based on the Driver.java code of
 * CETUS.
 *
 * May 6, 2014
 *
 */


package cetus.exec;

import cetus.analysis.*;
import cetus.base.grammars.CetusCParser;
import cetus.codegen.CodeGenPass;
import cetus.codegen.ompGen;
import cetus.hir.Annotatable;
import cetus.hir.Annotation;
import cetus.hir.AnnotationDeclaration;
import cetus.hir.AnnotationStatement;
import cetus.hir.BreadthFirstIterator;
import cetus.hir.Case;
import cetus.hir.ClassDeclaration;
import cetus.hir.CodeAnnotation;
import cetus.hir.CommentAnnotation;
import cetus.hir.CompoundStatement;
import cetus.hir.DFIterator;
import cetus.hir.Declaration;
import cetus.hir.Declarator;
import cetus.hir.Default;
import cetus.hir.DepthFirstIterator;
import cetus.hir.Enumeration;
import cetus.hir.Expression;
import cetus.hir.FlatIterator;
import cetus.hir.FloatLiteral;
import cetus.hir.ForLoop;
import cetus.hir.GotoStatement;
import cetus.hir.IDExpression;
import cetus.hir.IfStatement;
```

```
import cetus.hir.InlineAnnotation;
import cetus.hir.IntegerLiteral;
import cetus.hir.Literal;
import cetus.hir.Loop;
import cetus.hir.NestedDeclarator;
import cetus.hir.PragmaAnnotation;
import cetus.hir.PreAnnotation;
import cetus.hir.PrintTools;
import cetus.hir.Procedure;
import cetus.hir.ProcedureDeclarator;
import cetus.hir.Program;
import cetus.hir.SimpleExpression;
import cetus.hir.Statement;
import cetus.hir.StatementExpression;
import cetus.hir.SwitchStatement;
import cetus.hir.SymbolTools;
import cetus.hir.Tools;
import cetus.hir.TranslationUnit;
import cetus.hir.Traversable;
import cetus.hir.VariableDeclaration;
import cetus.hir.VariableDeclarator;
import cetus.hir.WhileLoop;
import cetus.transforms.*;

import java.io.*;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.ArrayList;
import java.util.Set;

/**
* Implements the command line parser and controls pass ordering.
* Users may extend this class by overriding runPasses
* (which provides a default sequence of passes).  The derived
* class should pass an instance of itself to the run method.
* Derived classes have access to a protected {@link Program Program}
* object.
*/
public class Critter {

    /**
    * A mapping from option names to option values.
    */
    protected static CommandLineOptionSet options =
     new CommandLineOptionSet();
```

```
    /**
    * Override runPasses to do something with this object.
    * It will contain a valid program when runPasses is called.
    */
    protected Program program;

    /** The filenames supplied on the command line. */
    protected List<String> filenames;

    // COS217 maximum loop length
    private int MAX_LOOP_LENGTH = 35;
    // COS217 maximum function length
    private int MAX_FUNCTION_LENGTH = 140;
    // COS217 maximum function length by statements
    private int MAX_FUNCTION_STATEMENT_LENGTH = 50;
    // COS217 maximum nesting level
    private int MAX_NESTING = 3;
    // COS217 maximum function number per file
    private int MAX_FUNCTION_NUMBER = 15;
    // COS217 maximum number of parameters per function
    private int MAX_PARAMETER_NUMBER = 7;
    // COS217 maximum discrepancy between number of local comments
    // and the number of elements that should have comments
    private int MAX_LOCAL_COMMENT_DISCREPANCY = 5;
    // COS217 maximum file length in lines.
    private int MAX_FILE_LENGTH = 500;
    // COS217 acceptable variable names that are shorter than
    // MIN_VAR_NAME_LENGTH
// the empty "" variable name is there to account for void parameters
    // in functions that technically have empty variable names.
    private String[] VAR_NAMES = {
           "c", "pc", "c1", "c2", "uc", "ac",
           "s", "ps", "s1", "s2", "us", "as",
           "i", "pi", "i1", "i2", "ui", "ai",
           "l", "pl", "l1", "l2", "ul", "al",
           "f", "pf", "f1", "f2", "af",
           "d", "pd", "d1", "d2", "ad",
           "pv",
           "o", "po", "ao",
           "j", "k", "n", "m", "",
        };

    private List<String> ACCEPTABLE_VAR_NAMES = Arrays.asList(VAR_NAMES);

    // COS217 minimum variable name length
    private int MIN_VAR_NAME_LENGTH = 3;
```

```
private String currentFilename;

/**
* Constructor used by derived classes.
*/
protected Critter() {
 Driver.registerOptions();
    options.add(options.UTILITY, "parser",
            "cetus.base.grammars.CetusCParser", "parsername",
            "Name of parser to be used for parsing source file");
    options.add(options.UTILITY, "outdir", "cetus_output", "dirname",
            "Set the output directory name (default is cetus_output)");
    options.add(options.UTILITY, "preprocessor", "cpp -C -I.",
     "command",
            "Set the preprocessor command to use");
    options.add(options.UTILITY, "verbosity", "0", "N",
            "Degree of status messages (0-4) that you wish to see " +
            "(default is 0)");
    options.add(options.UTILITY, "expand-user-header",
            "Expand user (non-standard) header file #includes into " +
            "code");
    options.add(options.UTILITY, "expand-all-header", null,
            "Expand all header file #includes into code");
    Driver.setOptionValue("expand-all-header", null);
}

/**
* Returns the value of the given key or null * if the value is not
* set.
* Key values are set on the command line as <b>-option_name=value</b>.
*
* @param key The key to search
* @return the value of the given key or null if the value is not set.
*/
public static String getOptionValue(String key) {
    return options.getValue(key);
}

/**
* Returns the set of  procedure names that should be excluded from
* transformations. These procedure names are specified with the
* skip-procedures command line option by providing a comma-separated list
* of names.
* @return set of procedure names that should be excluded from
*         transformations
*/
public static HashSet getSkipProcedureSet() {
```

```
    HashSet<String> proc_skip_set = new HashSet<String>();
    String s = getOptionValue("skip-procedures");
    if (s != null) {
        String[] proc_names = s.split(",");
        proc_skip_set.addAll(Arrays.asList(proc_names));
    }
    return proc_skip_set;
}

protected void parseOption(String opt) {
    opt = opt.trim();
    // empty line
    if (opt.length() < 2) {
        return;
    }
    int eq = opt.indexOf('=');
    if (eq == -1) { // if value is not set
        // registered option
        if (options.contains(opt)) {
            // no value on the option line, so set it to null
            setOptionValue(opt, null);
        } else {
            System.err.println("ignoring unrecognized option " + opt);
        }
    } else { // if value is set
        String option_name = opt.substring(0, eq);
        if (options.contains(option_name)) {
            if (option_name.equals("preprocessor")) {
                setOptionValue(option_name,
                           opt.substring(eq + 1).replace("\"", ""));
            } else {
                // use the value from the command line
                setOptionValue(option_name, opt.substring(eq + 1));
            }
        } else {
            System.err.println("ignoring unrecognized option "
                               + option_name);
        }
    }
}

/**
* Parses command line options to Cetus.
*
* @param args The String array passed to main by the system.
*/
protected void parseCommandLine(String[] args) {
```

```
/* print a useful message if there are no arguments */
if (args.length == 0) {
    printUsage();
    Tools.exit(1);
}
// keeps track of dangling preprocessor values
// e.g., args[n] = -preprocessor="cpp
//       args[n+1] = -EP"
boolean preprocessor = false;
int i; /* used after loop; don't put inside for loop */
for (i = 0; i < args.length; ++i) {
    String opt = args[i];
    // options start with "-"
    if (opt.charAt(0) != '-') {
        /* not an option -- skip to handling options and
         * filenames */
        break;
    }
    int eq = opt.indexOf('=');
    if (eq == -1) { // if value is not set
        String option_name = opt.substring(1);
        if (options.contains(option_name)) { // registered option
            preprocessor = false;
            // no value on the command line, so just set it to "1"
            // setValue(name) will search for predefined value
            // --> see setValue(String) for more information.
            options.setValue(option_name);
        } else if (preprocessor) {
            // found dangling preprocessor option
            setOptionValue("preprocessor",
                           getOptionValue("preprocessor")
                           + " " + opt.replace("\"",""));
        } else {
            System.err.println("ignoring unrecognized option " +
                               option_name);
        }
    } else { // if value is set
        String option_name = opt.substring(1, eq);
        if (options.contains(option_name)) {
            if (option_name.equals("preprocessor")) {
                preprocessor = true;
                setOptionValue(option_name,
                          opt.substring(eq + 1).replace("\"",""));
            } else {
                preprocessor = false;
                // use the value from the command line
                setOptionValue(option_name, opt.substring(eq + 1));
```

```
            }
        } else if (preprocessor) {
            setOptionValue("preprocessor",
                           getOptionValue("preprocessor")
                           + " " + opt.replace("\"",""));
        } else {
            System.err.println("ignoring unrecognized option " +
                               option_name);
        }
    }
    if (getOptionValue("help") != null ||
        getOptionValue("usage") != null) {
        printUsage();
        Tools.exit(0);
    }
    if (getOptionValue("dump-options") != null) {
        setOptionValue("dump-options", null);
        dumpOptionsFile();
        Tools.exit(0);
    }
    if (getOptionValue("dump-system-options") != null) {
        setOptionValue("dump-system-options", null);
        dumpSystemOptionsFile();
        Tools.exit(0);
    }
    // load options file and then proceed with rest of command line
    // options
    if (getOptionValue("load-options") != null) {
        // load options should not be set in options file
        setOptionValue("load-options", null);
        loadOptionsFile();
        // prevent reentering this handler
        setOptionValue("load-options", null);
    }
}
// end of arguments without a file name
if (i >= args.length) {
    System.err.println("No input files!");
    Tools.exit(1);
}
// The purpose of this wildcard expansion is to ease the use of
// IDE environment which usually doesn't handle wildcards.
int num_file_args = args.length-i;
filenames = new ArrayList<String>(num_file_args);
for (int j = 0; j < num_file_args; ++j, ++i) {
    if (args[i].contains("*") || args[i].contains("?")) {
        File parent =
```

```
                    (new File(args[i])).getAbsoluteFile().getParentFile();
            for (File file : parent.listFiles(new RegexFilter(args[i]))) {
                filenames.add(file.getAbsolutePath());
            }
        } else {
            filenames.add(args[i]);
        }
    }
    if (filenames.isEmpty()) {
        System.err.println("No input files!");
        Tools.exit(1);
    }
}

/**
* Parses all of the files listed in <var>filenames</var>
* and creates a {@link Program Program} object.
*/
@SuppressWarnings({"unchecked", "cast"})
protected void parseFiles() {
    program = new Program();
    Class class_parser;
    try {
        class_parser = getClass().getClassLoader().loadClass(
                       getOptionValue("parser"));
        //CetusParser cparser =
         //       (CetusParser)class_parser.getConstructor().newInstance();
        String dir = (new File(filenames.get(0))).getParent();
        CetusParser cparser = new CetusCParser(dir);
        for (String file : filenames) {
         TranslationUnit tu = cparser.parseFile(file, options);
            program.addTranslationUnit(tu);
            String[] f = file.split("/");
            currentFilename = f[f.length - 1];
        }
    } catch (ClassNotFoundException e) {
        System.err.println("Failed to load parser: " +
                           getOptionValue("parser"));
        Tools.exit(1);
    } catch(Exception e) {
        System.err.println("Failed to initialize parser");
        Tools.exit(1);
    }

    // It is more natural to include these two steps in this method.
    // Link IDExpression => Symbol object for faster future access.
    SymbolTools.linkSymbol(program);
```

```
    // Convert the IR to a new one with improved annotation support
    TransformPass.run(new AnnotationParser(program));
}

/**
* Prints the list of options that Cetus accepts.
*/
public void printUsage() {
    String usage = "\ncetus.exec.Critter [option]... [file]...\n";
    usage += options.getUsage();
    System.err.println(usage);
}

/**
* dump default options to file options.cetus in working directory
* do not overwrite if file already exists.
*/
public void dumpOptionsFile()  {
    // check for options.cetus in working directory
    // registerOptions();
    File optionsFile = new File("options.cetus");
    // create file options.cetus
    try {
        if (optionsFile.createNewFile()) {
            // populate options.cetus
            FileOutputStream fo = new FileOutputStream(optionsFile);
            PrintStream ps = new PrintStream(fo);
            ps.println(options.dumpOptions().trim());
            ps.close();
            fo.close();
        }
    } catch (IOException e) {
        System.err.println("Error: Failed to dump options.cetus");
    }
}

public void dumpSystemOptionsFile()  {
    // check for options.cetus in working directory
    // registerOptions();
    String homePath = System.getProperty("user.home");
    File optionsFile = new File(homePath,"options.cetus");
    // create file options.cetus
    try {
        if (optionsFile.createNewFile()) {
            // populate options.cetus
            FileOutputStream fo = new FileOutputStream(optionsFile);
            PrintStream ps = new PrintStream(fo);
```

```
            ps.println(options.dumpOptions().trim());
            ps.close();
            fo.close();
        }
    } catch (IOException e) {
        System.err.println(
                "Error: Failed to dump system wide options.cetus");
    }
}

/**
* load options.cetus
* search order is working directory and then home directory
*/
public void loadOptionsFile() {
    // check working directory for options.cetus
    // check home directory for options.cetus
    File optionsFile = new File("options.cetus");
    //dumpOptionsFile();
    if (!optionsFile.exists()) {
        String homePath = System.getProperty("user.home");
        optionsFile = new File(homePath,"options.cetus");
    }
    if (!optionsFile.exists()) {
        System.err.println("Error: Failed to load options.cetus");
        System.err.println(
            "Use option -dump-options or -dump-system-options"
            + " to create options.cetus with default values");
        Tools.exit(1);
    }
    // load file contents
    try {
        FileReader fr = new FileReader(optionsFile);
        BufferedReader br = new BufferedReader(fr);
        String line;
        // Read lines
        while ((line=br.readLine()) != null) {
            // Remove comments
            if (line.startsWith("#"))
                continue;
            // load option
            parseOption(line);
        }
    } catch (Exception e) {
        System.err.println("Error while loading options file");
        Tools.exit(1);
    }
```

```
  }


  private long getLineNumber(Traversable element)
  {
      Traversable lastComment = getPrevious(element);

      while(!lastComment.toString().startsWith("#pragma critTer")
       || lastComment.toString().contains("Include"))
       lastComment = getPrevious(lastComment);

      String[] parts = lastComment.toString().split(":");
      return Long.parseLong(parts[1]);
  }

  private String getFilename(Traversable element) {
   Traversable lastComment = getPrevious(element);

      while(!lastComment.toString().startsWith("#pragma critTer")
       || lastComment.toString().contains("Include"))
       lastComment = getPrevious(lastComment);

      String[] parts = lastComment.toString().split(":");
      return parts[2];
  }

  /*
   * Check if loop length exceeds a maximum length (max_loop_length).
   */
  public void checkLoop() {
   DepthFirstIterator<Traversable> dfs =
   new DepthFirstIterator<Traversable>(program);

   while (dfs.hasNext()) {
   Traversable t = dfs.next();

   // skips all the standard included files
   if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
   while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
   t = dfs.next();
   }
   }

   else if (t instanceof Loop) {
   Statement s = ((Loop) t).getBody();
   DepthFirstIterator<Traversable> ldfs =
new DepthFirstIterator<Traversable>(s);
```

```
      int looplinecount = 0;
      while (ldfs.hasNext()) {
      Traversable st = ldfs.next();

      if (st instanceof AnnotationStatement) {
      looplinecount++;
      }
      }

      if (looplinecount > MAX_LOOP_LENGTH) {
      System.err.printf("\n%s: line %d: low priority: " +
      "\nA loop should consist of fewer than %d " +
      "lines;\n " +
      "this loop consists of %d lines; consider " +
      "refactoring\n",
      getFilename(t), getLineNumber(t),
      MAX_LOOP_LENGTH, looplinecount);
      }
      }
      }

    }

    /*
     * Check if all functions in non-main modules have the same prefix.
     */
    public void checkFunctionNaming() {
     DepthFirstIterator<Traversable> dfs =
     new DepthFirstIterator<Traversable>(program);

String commonPrefix = null;

// check if test needs to be done (check if there is a main
// function definition)
boolean hasMain = false;
while (dfs.hasNext()) {
Traversable t = dfs.next();

if (t instanceof Procedure) {
IDExpression n = ((Procedure) t).getName();
     String name = n.getName();
     if (name.compareTo("main") == 0)
     hasMain = true;
}
}
```

```
if (!hasMain) {
dfs = new DepthFirstIterator<Traversable>(program);

while (dfs.hasNext()) {
     Traversable t = dfs.next();

     // skips all the standard included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }


     if (t instanceof Procedure) {

     IDExpression n = ((Procedure) t).getName();
     String name = n.getName();
     String prefix = name.split("_")[0];
     if (commonPrefix == null &&
     prefix.compareTo("main") != 0)
     commonPrefix = prefix;
     else if (prefix.compareTo("main") != 0) {
     if (commonPrefix.compareTo(prefix) != 0) {
     System.err.printf("\n%s: line %d: medium priority: " +
     "\nA function's prefix should match the " +
     "module name; %s and %s do not match\n",
     getFilename(t), getLineNumber(t),
         commonPrefix, prefix);

     }
     }
     }
}

     }
    }

    /*
     * Checks if a function length exceeds a maximum length
     * (MAX_FUNCTION_LENGTH)
     */
    void checkFunctionLengthByLines() {
     DepthFirstIterator<Traversable> dfs =
     new DepthFirstIterator<Traversable>(program);

while (dfs.hasNext()) {
```

```
     Traversable t = dfs.next();

     // skips all standard included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     else if (t instanceof Procedure) {
     Statement s = ((Procedure) t).getBody();
     DepthFirstIterator<Traversable> ldfs =
     new DepthFirstIterator<Traversable>(s);

     int looplinecount = 0;
     while (ldfs.hasNext()) {
     Traversable st = ldfs.next();

     if (st.toString().startsWith("#pragma critTer")) {
     looplinecount++;
     }
     }

     if (looplinecount > MAX_FUNCTION_LENGTH) {
     System.err.printf("\n%s: line %d: low priority: " +
     "\nA function should consist of fewer than " +
     "%d lines;\n " +
     "this function consists of %d lines; " +
     "consider refactoring\n",
     getFilename(t), getLineNumber(t),
     MAX_FUNCTION_LENGTH, looplinecount);
     }

     }

     }
    }

    /*
     * Check if there are too many functions in a file
     * (MAX_FUNCTION_NUMBER).
     */
    public void checkFunctionNumber() {
     DepthFirstIterator<Traversable> dfs =
     new DepthFirstIterator<Traversable>(program);

int functioncount = 0;
```

```
     while (dfs.hasNext()) {
     Traversable t = dfs.next();

     // skips all the included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     if (t instanceof Procedure) {
     functioncount++;
     }

     }

     if (functioncount > MAX_FUNCTION_NUMBER) {
System.err.printf("\n%s: low priority: \nA file should " +
"contain no more than %d functions;\n " +
"this file contains %d functions\n",
currentFilename, MAX_FUNCTION_NUMBER, functioncount);
}
    }

    /*
     * Check if there are too many parameters in a function
     * (max_parameter_number).
     */
    public void checkFunctionParams() {
     DepthFirstIterator<Traversable> dfs =
     new DepthFirstIterator<Traversable>(program);

while (dfs.hasNext()) {
     Traversable t = dfs.next();

     // skips all included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     else if (t instanceof Procedure) {
     int paramNum = ((Procedure) t).getNumParameters();

     if (paramNum > MAX_PARAMETER_NUMBER) {
     System.err.printf("\n%s: line %d: medium priority: " +
```

```
 "\nA function should have no more than %d " +
 "parameters; this function has %d\n",
 getFilename(t), getLineNumber(t),
 MAX_PARAMETER_NUMBER, paramNum);
 }
 }
 }
}

/* Check if all functions have comments, and if the comment mentions
 * each parameter by name and what the function returns.
 */
public void checkFunctionCommentValid() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);


 while (dfs.hasNext()) {
 Traversable t = dfs.next();

 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof Procedure) {
 Procedure function = (Procedure) t;

 List list = function.getReturnType();
 List stringList = new ArrayList();

for (Object x : list) {
stringList.add(x.toString());
}

Traversable p = getPreviousNonPragma(function);
// main function doesn't need to be checked for parameter
// or return mentions in comment
if (function.getName().toString().compareTo("main") != 0) {

 if (p instanceof AnnotationDeclaration) {
 Traversable comment = (AnnotationDeclaration) p;

 // Checks if the function's comment refers to
 // parameters.
```

```
    for (int i = 0;
    i < function.getNumParameters();
    i++) {

    String paramName =
    function.getParameter(i).getDeclaredIDs().get(0).toString();
    if (!comment.toString().contains(paramName)) {
    System.err.printf("\n%s: line %d: high priority: " +
    "\nA function's comment should refer to " +
    "each parameter by name;\nyour comment " +
    "does not refer to '%s'\n",
     getFilename(comment), getLineNumber(comment), paramName);
    }
     }

     // Checks for explicitly stated return, only
    // for non-void function.
     if (!stringList.contains("void")) {
     if (!comment.toString().contains("return") &&
     !comment.toString().contains("Return")) {
    System.err.printf("\n%s: line %d: high priority: " +
    "\nA function's comment should state " +
                           "explicitly what the function returns\n",
getFilename(comment), getLineNumber(comment));
     }
     }
     }
    }

    if (!(p instanceof AnnotationDeclaration)) {
     System.err.printf("\n%s: line %d: high priority: " +
     "\nA function definition should have a comment\n",
     getFilename(function), getLineNumber(function));
     }
     }
     }

    }


    /* Assumes that if, while, for, do while and switch elements should
     * all have comments.
     * Checks the number of local comments with the number of those
     * elements, and throws a warning if the discrepancy between the two
     * is greater than maxLocalCommentDiscrepancy.
     */
    public void checkFunctionHasEnoughComments() {
```

```
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();

 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof Procedure) {
 DepthFirstIterator<Traversable> functiondfs =
 new DepthFirstIterator<Traversable>(t);
 int countElements = 0;
 int countComments = 0;

 while (functiondfs.hasNext()) {
 Traversable functiont = functiondfs.next();
 if (functiont instanceof Loop)
 countElements++;
 else if (functiont instanceof IfStatement)
 countElements++;
 else if (functiont instanceof SwitchStatement)
 countElements++;

 else if (functiont instanceof AnnotationStatement) {
 if (!functiont.toString().startsWith("#pragma"))
 countComments++;
 }
 }

 if ((countElements - countComments)
 > MAX_LOCAL_COMMENT_DISCREPANCY) {
 System.err.printf("\n%s: line %d: low priority: " +
 "\nThis function definition probably needs" +
 " more local comments\n",
 getFilename(t), getLineNumber(t));
 }

 }
 }
}

/* Checks if all global variables have comments. Comments must be
```

```
 * either on the line immediately previous the global variable, or
 * with at most one blank line between the comment and the global
 * variable.
 */
public void checkGlobalHasComment() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);
 dfs.pruneOn(Procedure.class); // skips all the functions
 dfs.pruneOn(ClassDeclaration.class); // skips all the structs

 while (dfs.hasNext()) {
 Traversable t = dfs.next();

 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof VariableDeclaration
 || t instanceof Enumeration
 || t instanceof ClassDeclaration) {
 if (!t.toString().startsWith("typedef int * __")) {
 Traversable p = getPreviousNonPragma(t);
 if (!(p instanceof AnnotationStatement)
 && !(p instanceof AnnotationDeclaration)) {
 if (t.getParent().getParent() != null) {
 if (!(t.getParent().getParent() instanceof VariableDeclaration)) {
 System.err.printf("\n%s: line %d: high priority: " +
 "\nA comment should appear above each " +
 "global variable.\n",
 getFilename(t), getLineNumber(t));
 }
 }
 }
 }
 }
 }
}

/*
 * Checks if the file begins with a comment.
 */
public void checkBeginningComment() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);
```

```
     Traversable t = dfs.next();

     while (!(t.toString().startsWith("#pragma critTer"))) {
     t = dfs.next();
     }
     Traversable first = dfs.next();

     if (first.toString().startsWith("#pragma")) {
     System.err.printf("\n%s: line %d: high priority: " +
     "\nA file should begin with a comment.\n",
currentFilename, getLineNumber(first));
     }

     if (!(first instanceof AnnotationDeclaration)) {
     System.err.printf("\n%s: line %d: high priority: " +
     "\nA file should begin with a comment.\n",
currentFilename, getLineNumber(first));
     }

     dfs = new DepthFirstIterator<Traversable>(program);

     // check all student's .h files
     while (dfs.hasNext()) {
     t = dfs.next();

     if (t.toString().startsWith("#pragma critTer:startStudentInclude")) {
     while (!(t.toString().startsWith("#pragma critTer:1:")))
          t = dfs.next();
     Traversable n = dfs.next();
     if (n.toString().startsWith("#pragma critTer")) {
     System.err.printf("\n%s: line %d: high priority: " +
     "\nA file should begin with a comment.\n",
     getFilename(n), getLineNumber(n));
     }
     if (!(n instanceof AnnotationDeclaration)) {
     System.err.printf("\n%s: line %d: high priority: " +
     "\nA file should begin with a comment.\n",
     getFilename(n), getLineNumber(n));
     }
     }

     }
    }

    /*
     * Checks that all switch statements have default cases.
```

```
 */
public void checkSwitchHasDefaultCase() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();

 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof SwitchStatement) {
 DepthFirstIterator<Traversable> sdfs =
 new DepthFirstIterator<Traversable>(t);
 boolean hasDefault = false;

 while (sdfs.hasNext()) {

 Traversable s = sdfs.next();
 if (s instanceof Default)
 hasDefault = true;
 }

 if (!hasDefault) {
 System.err.printf("\n%s: line %d: low priority: " +
 "\nA switch statement should have a default " +
 "case\n",
 getFilename(t), getLineNumber(t));
 }
 }

 }
}

/*
 * Checks that all switch cases have a break or return statement.
 */
public void checkSwitchCases() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();
```

```
 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof SwitchStatement) {

 Traversable body = ((SwitchStatement) t).getBody();

 List<Traversable> list = body.getChildren();

 boolean caseHasBreak = true;
 Traversable currentCase = null;

 for (Traversable i : list) {
 if (i.toString().startsWith("case")
 || i.toString().startsWith("default")) {
 if (caseHasBreak) {
 caseHasBreak = false;
 currentCase = i;
 }
 else {
 System.err.printf("\n%s: line %d: medium priority:" +
 " \nEach case/default in a switch statement " +
 "should have a break or return statement, " +
 "you're missing one here.\n",
 getFilename(currentCase),
 getLineNumber(currentCase));
 }
 }

 if (i.toString().startsWith("break")
 || i.toString().startsWith("return"))
 caseHasBreak = true;
 }
 }
 }
}


/*
 * Checks if the file is longer than MAX_FILELENGTH.
 */
public void checkFileLength() {
```

```
     DepthFirstIterator<Traversable> dfs =
     new DepthFirstIterator<Traversable>(program);

     long linecount = 0;

     while (dfs.hasNext()) {
     Traversable t = dfs.next();

     // skips all the standard included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     // deals with student's included files
     else if (t.toString().startsWith("#pragma critTer:endStudentInclude:")) {
     if (getLineNumber(t) > MAX_FILE_LENGTH) {
          System.err.printf("\n%s: low priority: \nA source " +
          "code file should contain fewer than %d " +
          "lines;\nthis file contains %d lines\n",
     getFilename(t), MAX_FILE_LENGTH, getLineNumber(t));
          }

     }

     else if (t.toString().startsWith("#pragma critTer")
     && !t.toString().contains("Include")) {
     String[] parts = t.toString().split(":");
         long currentline = Long.parseLong(parts[1]);
         if (currentline > linecount)
             linecount = currentline;
     }
     }

     if (linecount > MAX_FILE_LENGTH) {
     System.err.printf("\n%s: low priority: \nA source code " +
     "file should contain fewer than %d " +
     "lines;\nthis file contains %d lines\n",
currentFilename, MAX_FILE_LENGTH, linecount);
     }
    }

    /*
     * Checks if all fields in a struct have comments.
     */
    public void checkStructHasComment() {
```

```
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();

 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof ClassDeclaration) {
 DepthFirstIterator<Traversable> cdfs =
 new DepthFirstIterator<Traversable>(t);

 while (cdfs.hasNext()) {
 Traversable c = cdfs.next();

 if (c instanceof VariableDeclaration
 || t instanceof Enumeration) {

     Traversable p = getPreviousNonPragma(c.getParent());
     if (!(p instanceof PreAnnotation)) {

     System.err.printf("\n%s: line %d: medium priority:" +
     " \nA comment should appear above each " +
     "field in a struct.\n",
     getFilename(c), getLineNumber(c));
     }

 }
 }
     }
 }
}

/*
 * Warn against using GOTOs.
 */
public void checkGoTos() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();
```

```
     // skips all the standard included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     else if (t instanceof GotoStatement) {
     System.err.printf("\n%s: line %d: high priority: " +
     "\nNever use GOTO statements\n",
getFilename(t), getLineNumber(t));
     }
     }
    }

    /*
     * Warn against using magic numbers outside of a declaration
     * (except for 0, 1 and 2, except in case statements).
     */
    public void checkMagicNumbers() {
     DepthFirstIterator<Traversable> dfs =
     new DepthFirstIterator<Traversable>(program);
     dfs.pruneOn(VariableDeclaration.class); // don't check declarations
     dfs.pruneOn(Enumeration.class); // don't check magic numbers in enums

     while (dfs.hasNext()) {
     Traversable t = dfs.next();

     // skips all the standard included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     else {
     // handle cases here (no magic numbers whatsoever
     // inside case)
     if (t instanceof Case) {
     String c = ((Case) t).getExpression().toString();

     if (isNumeric(c)) {
     System.err.printf("\n%s: line %d: high priority: " +
     "\nUse of magic number (%s), which should " +
     "be given a meaningful name, " +
     "or a #define, which should be replaced " +
```

```
 "with an enum (unless it's the result of " +
 "a #define in a standard C header file)\n",
 getFilename(t), getLineNumber(t), c);
 }

 }

 if (t instanceof FloatLiteral) {
 FloatLiteral number = (FloatLiteral) t;
 if (!t.getParent().toString().startsWith("__")) {
 if (number.getValue() != 0 && number.getValue() != 1
 && number.getValue() != 2) {
 System.err.printf("\n%s: line %d: high priority:" +
 " \nUse of magic number (%s), which should" +
 " be given a meaningful name, " +
     "or a #define, which should be replaced " +
     "with an enum (unless it's the result of " +
     "a #define in a standard C header file)\n",
     getFilename(t), getLineNumber(t),
     t.toString());
 }
 }

 }

 if (t instanceof IntegerLiteral) {
 IntegerLiteral number = (IntegerLiteral) t;
 if (!t.getParent().toString().startsWith("__")) {
 if (number.getValue() != 0 && number.getValue() != 1
 && number.getValue() != 2) {
 System.err.printf("\n%s: line %d: high priority: " +
 "\nUse of magic number (%s), which should " +
 "be given a meaningful name, " +
     "or a #define, which should be replaced with " +
     "an enum (unless it's the result of a #define " +
     "in a standard C header file)\n",
     getFilename(t), getLineNumber(t),
     t.toString());
 }
 }
 }
 }
 }
}

/*
 * Checks that variable names are longer than the MIN_VAR_NAME_LENGTH,
```

```
 * with the exception of ACCEPTABLE_VAR_NAMES.
 */
public void checkVariableName() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();

 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof VariableDeclaration
 && !t.getParent().toString().startsWith("__")) {


 List<IDExpression> vars = ((VariableDeclaration) t).getDeclaredIDs();
 for (IDExpression x : vars) {
 String xName = x.toString();
 if (!ACCEPTABLE_VAR_NAMES.contains(xName)) {
 if (xName.length() < MIN_VAR_NAME_LENGTH) {
 System.err.printf("\n%s: line %d: medium priority:" +
 " \nVariable/function name '%s' " +
 "is too short\n",
         getFilename(t), getLineNumber(t), xName);
 }
 }
 }
 }

 else if (t instanceof Enumeration) {
 List<IDExpression> vars = ((Enumeration) t).getDeclaredIDs();
 for (IDExpression x : vars) {
 String xName = x.toString();
 if (!ACCEPTABLE_VAR_NAMES.contains(xName)) {
 if (xName.length() < MIN_VAR_NAME_LENGTH) {
 System.err.printf("\n%s: line %d: medium priority:" +
 " \nVariable/function name '%s' " +
 "is too short\n",
         getFilename(t), getLineNumber(t), xName);
 }
 }
 }
```

```
 }
 }
}

/*
 * Check if a function exceeds a maximum statement count.
 */
public void checkFunctionLengthByStatement() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();

 // skips all the standard included files
 if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
 while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
 t = dfs.next();
 }
 }

 else if (t instanceof Procedure) {
 CompoundStatement body = ((Procedure) t).getBody();

 int statementcount = countStatements(body);

 if (statementcount > MAX_FUNCTION_STATEMENT_LENGTH) {
 System.err.printf("\n%s: line %d: low priority: " +
 "\nA function definition should consist of " +
 "fewer than %d statements;\nthis function " +
 "definition consists of %d statements\n",
 getFilename(t), getLineNumber(t),
 MAX_FUNCTION_STATEMENT_LENGTH, statementcount);
 }
 }
 }
}

/*
 * Check if nesting is too deep.
 */
public void checkNesting() {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);

 while (dfs.hasNext()) {
 Traversable t = dfs.next();
```

```
// skips all the standard included files
if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
t = dfs.next();
}
}

else {
if (t instanceof Loop) {
Traversable body = ((Loop) t).getBody();
int nesting = 0;

Traversable parent = body.getParent();
while (parent != null) {
if (parent instanceof Loop
|| parent instanceof IfStatement
|| parent instanceof SwitchStatement)
nesting++;
parent = parent.getParent();
}

if (nesting > MAX_NESTING) {
System.err.printf("\n%s: line %d: low priority: " +
"\nThis area is deeply nested at level %d," +
" consider refactoring\n",
getFilename(body), getLineNumber(body),
nesting);
}
}
else if (t instanceof IfStatement) {
int nesting = 1;

Traversable parent = t.getParent();
Traversable parentsGrandChild = t.getChildren().get(0);
Traversable current = t;
while (parent != null) {
if (parent instanceof Loop
|| parent instanceof SwitchStatement)
nesting++;
if (parent instanceof IfStatement) {
if (parentsGrandChild instanceof IfStatement) {
Traversable thenStatement =
((IfStatement) parent).getThenStatement();

    if (thenStatement.toString().compareTo(current.toString()) == 0) {
    nesting++;
```

```
    }
 }
 else
 nesting++;

 }

 parentsGrandChild = parentsGrandChild.getParent();
 current = parent;
 parent = parent.getParent();
 }
 if (nesting > MAX_NESTING) {
 System.err.printf("\n%s: line %d: low priority: " +
 "\nThis area is deeply nested at level %d," +
 " consider refactoring\n",
 getFilename(t), getLineNumber(t), nesting);
 }
 }
 else if (t instanceof SwitchStatement) {
 Traversable body = ((SwitchStatement) t).getBody();
 int nesting = 0;

 Traversable parent = body.getParent();
 while (parent != null) {
 if (parent instanceof Loop
 || parent instanceof IfStatement
 || parent instanceof SwitchStatement)
 nesting++;
 parent = parent.getParent();
 }

 if (nesting > MAX_NESTING) {
 System.err.printf("\n%s: line %d: low priority: " +
 "\nThis area is deeply nested at level %d," +
 " consider refactoring\n",
 getFilename(body), getLineNumber(body),
 nesting);
 }
 }
 }

 }
}

/*
 * Checks if compound statement is empty.
 */
```

```
    public void checkEmptyCompound() {
     DepthFirstIterator<Traversable> dfs =
     new DepthFirstIterator<Traversable>(program);

     while (dfs.hasNext()) {
     Traversable t = dfs.next();

     // skips all the standard included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     else if (t.toString().compareTo("{\n\n}") == 0) {
     System.err.printf("\n%s: line %d: medium priority: " +
     "\nDo not use empty compound statements.\n",
getFilename(t), getLineNumber(t));
     }
     }
    }

    /*
     * Check if pointer parameters are checked by asserts.
     */
    public void checkAsserts() {
    DFIterator<Traversable> dfs = new DFIterator<Traversable>(program);

     while (dfs.hasNext()) {
     Traversable t = dfs.next();

     // skips all standard included files
     if (t.toString().startsWith("#pragma critTer:startStdInclude:")) {
     while (!(t.toString().startsWith("#pragma critTer:endStdInclude:"))) {
     t = dfs.next();
     }
     }

     if (t instanceof Procedure) {

     DepthFirstIterator<Traversable> functiondfs =
     new DepthFirstIterator<Traversable>(t);

     List<Declaration> params = ((Procedure) t).getParameters();
     List<String> paramNames = new ArrayList<String>();

     for (Declaration p : params) {
```

```
 List<IDExpression> declaredIDs = p.getDeclaredIDs();
 for (IDExpression parameter : declaredIDs) {
 if (parameter.getParent().toString().contains("[]")) {
 paramNames.add(parameter.toString());
 }
 // parameters formatted as pointers
 else if (parameter.getParent().toString().contains("*")) {
 paramNames.add(parameter.toString());
 }
 }

 }
 boolean[] hasAssert = new boolean[paramNames.size()];


 while (functiondfs.hasNext()) {
 Traversable t2 = functiondfs.next();

 if (t2.toString().startsWith("__assert")) {
for (int i = 0; i < paramNames.size(); i++) {
if (t2.toString().contains(paramNames.get(i)))
hasAssert[i] = true;
}

 }
 }

 // no need for asserts for argv
 for (int i = 0; i < hasAssert.length; i++) {
 if (paramNames.get(i).compareTo("argv") != 0) {
 if (!hasAssert[i]) {
System.err.printf("\n%s: line %d: medium priority:" +
" \nDo you want to validate '%s' " +
"through an assert?\n",
getFilename(t), getLineNumber(t),
paramNames.get(i));
}
 }
 }

 }
 }
}

private int countStatements(Traversable body) {
 FlatIterator<Traversable> flat = new FlatIterator<Traversable>(body);
```

```
int statementcount = 0;

while (flat.hasNext()) {
Traversable s = flat.next();
statementcount++;

if (s instanceof Loop) {
statementcount += countStatements(((Loop) s).getBody());
}

else if (s instanceof IfStatement) {
if (((IfStatement) s).getElseStatement() != null)
statementcount += countStatements(((IfStatement) s).getElseStatement());
if (((IfStatement) s).getThenStatement() != null)
statementcount += countStatements(((IfStatement) s).getThenStatement());
}

else if (s instanceof CompoundStatement) {
statementcount += countStatements(s);
statementcount--;
}

else if (s instanceof SwitchStatement) {
statementcount += countStatements(((SwitchStatement) s).getBody());
}

else if (s instanceof AnnotationStatement || s instanceof AnnotationDeclaration)
statementcount--;
}
return statementcount;
    }

    /*
     * Determines if a string input is numeric.
     */
    private boolean isNumeric(String input) {
        try {
            Double.parseDouble(input);
            return true;
        }
        catch( Exception e ) {
            return false;
        }
    }

    /*
     * Returns the previous node in the parse tree that is not a pragma
```

```
 * annotation
 */
private Traversable getPreviousNonPragma(Traversable current) {
 Traversable nonPragmaPrev = getPrevious(current);
 while (nonPragmaPrev.toString().startsWith("#pragma")) {
 // skip over .h files
 if (nonPragmaPrev.toString().startsWith("#pragma critTer:end")) {
 while(!nonPragmaPrev.toString().startsWith("#pragma critTer:start"))
 nonPragmaPrev = getPrevious(nonPragmaPrev);
 }
 nonPragmaPrev = getPrevious(nonPragmaPrev);
 }

 return nonPragmaPrev;
}

/*
 * Returns previous node in parse tree that is a pragma annotation
 */
private Traversable getPreviousPragma(Traversable current) {
 Traversable pragmaPrev = getPrevious(current);
 while (!pragmaPrev.toString().startsWith("#pragma"))
 pragmaPrev = getPrevious(pragmaPrev);
 return pragmaPrev;
}

/*
 * Returns the previous node in parse tree.
 */
private Traversable getPrevious(Traversable current) {
 DepthFirstIterator<Traversable> dfs =
 new DepthFirstIterator<Traversable>(program);
 DepthFirstIterator<Traversable> dfs2 =
 new DepthFirstIterator<Traversable>(program);

 // dfs2 is always walking one ahead of dfs
 dfs2.next();
 Traversable t = dfs2.next();

 Traversable prev = dfs.next();

 while (dfs2.hasNext()) {
 if (t == current)
 break;
 t = dfs2.next();
 prev = dfs.next();
 }
```

```
 return prev;
}


/**
* Sets the value of the option represented by <i>key</i> to
* <i>value</i>.
*
* @param key The option name.
* @param value The option value.
*/
public static void setOptionValue(String key, String value) {
    options.setValue(key, value);
}

public static boolean isIncluded(
        String name, String hir_type, String hir_name) {
    return options.isIncluded(name, hir_type, hir_name);
}

/**
* Entry point for Cetus; creates a new Driver object,
* and calls run on it with args.
*
* @param args Command line options.
*/
public static void main(String[] args) {
 Critter dt = new Critter();
    dt.parseCommandLine(args);
    dt.parseFiles();

    System.err.println("critTer2 warnings start here");
    System.err.println("----------------------------");
    System.err.println();

    // Checks begin here.
    dt.checkBeginningComment();
    dt.checkFunctionCommentValid();
    dt.checkFunctionHasEnoughComments();
    dt.checkGlobalHasComment();
    dt.checkLoop();
    dt.checkFunctionParams();
    dt.checkFunctionLengthByLines();
    dt.checkFunctionNumber();
    dt.checkFunctionNaming();
    dt.checkFileLength();
```

```
            dt.checkSwitchHasDefaultCase();
            dt.checkSwitchCases();
            dt.checkStructHasComment();
            dt.checkGoTos();
            dt.checkMagicNumbers();
            dt.checkVariableName();
            //dt.checkFunctionLengthByStatement();
            dt.checkNesting();
            dt.checkEmptyCompound();
            dt.checkAsserts();

            System.err.println();
            System.err.println("----------------------------");
            System.err.println("critTer2 warnings end here");

        }

        /**
        * Implementation of file filter for handling wild card character and other
        * special characters to generate regular expressions out of a string.
        */
        private static class RegexFilter implements FileFilter {
            /** Regular expression */
            private String regex;

            /**
            * Constructs a new filter with the given input string
            * @param str String to construct regular expression out of
            */
            public RegexFilter(String str) {
                regex = str.replaceAll("\\.", "\\\\.") // . => \.
                .replaceAll("\\?", ".")                // ? => .
                .replaceAll("\\*", ".*");              // * => .*
            }

            @Override
            public boolean accept(File f) {
                return f.getName().matches(regex);
            }
        }

    }
```

B. CritTer2 script

```
#!/usr/bin/python
#--------------------------------------------------------------------------
# critTer
#--------------------------------------------------------------------------

import sys
import os
from subprocess import call
from shutil import rmtree

#--------------------------------------------------------------------------

CRITTER_DIR = '/u/akroutik/critTer2'
CRITTER_BIN_DIR = CRITTER_DIR + '/bin'
CRITTER_JAR_DIR = CRITTER_DIR + '/Jars'
TEMP_DIR = './critTer_bak'
CPATH = \
    CRITTER_JAR_DIR + '/cetus.jar' + ':' + \
    CRITTER_JAR_DIR + '/antlr-3.5.1-complete.jar' + ':' + \
    CRITTER_BIN_DIR

#--------------------------------------------------------------------------
#--------------------------------------------------------------------------

# Splice pragmas into file frompath which indicate line numbers, and
# which delimit includes of standard .h files.  Store the result in
# file topath.

def annotate(frompath, topath):
    num = 1
    with open(frompath, 'r') as student_code:
        with open(topath, 'w') as modded:
            modded.write('typedef int* __WAIT_STATUS;\n')
            for line in student_code:
                modded.write('#pragma critTer:' + str(num) + ':' + str(frompath) + ':\n')
                if (line.lstrip().startswith('#include')) \
                        and ('<' in line) and ('>' in line):
                    modded.write('#pragma critTer:startStdInclude:\n')
                    modded.write(line)
                    modded.write('#pragma critTer:endStdInclude:\n')
                elif (line.lstrip().startswith('#include')) \
                        and ('\"' in line):
                    modded.write('#pragma critTer:startStudentInclude:\n')
```

```
                    modded.write(line)
                    modded.write('#pragma critTer:endStudentInclude:\n')
                else:
                    modded.write(line)
                num += 1

#--------------------------------------------------------------------

def main():

    if (len(sys.argv) == 1):
        print "No arguments given!"
        sys.exit(1)

    # Start the string of command-line arguments to be passed to cetus.
    args = '-preprocessor="gcc217 -E -C -dD" '

    # If TEMP_DIR doesn't exist, then create it.
    if not os.path.exists(TEMP_DIR):
        os.makedirs(TEMP_DIR)

    # Populate TEMP_DIR with line-numbered versions of each .c file.
    for arg in sys.argv[1:]:
        if arg.endswith('.c'):
            cFile = arg
            annotatedCFile = TEMP_DIR + '/' + arg
            annotate(cFile, annotatedCFile)
            args += annotatedCFile + " "

    for hFile in os.listdir('.'):
        if hFile.endswith('.h'):
            annotatedHFile = TEMP_DIR + '/' + hFile
            annotate(hFile, annotatedHFile)

    # Execute Cetus to (1) parse the line-numbered .c files that are
    # in TEMP_DIR (placing the resulting files in the ./cetus_output
    # directory), and (2) write its critique to stdout.
    call('java -cp "' + CPATH + '" cetus.exec.Critter ' + args, shell=True)

    # Delete the temporary directories.
    rmtree('./critTer_bak')
    #rmtree('./cetus_output')

#--------------------------------------------------------------------

if __name__ == "__main__":
    main()
```