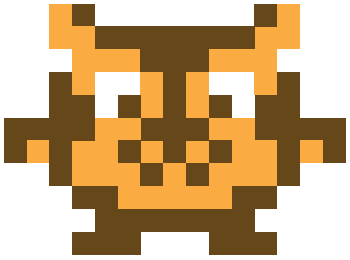
CritTer
Critique From the Terminal

Customizable Style Checking for C Programs

Author:
Erin Rosenbaum

Advisor:
Dr. Robert Dondero



Senior Thesis

Submitted to Princeton University
Department of Computer Science
In Partial Fulfillment of the Requirements for the A.B. Degree

April 15, 2011

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

Erin Rosenbaum
April 15, 2011

Abstract

Stylistic errors are a symptom of poorly written code. Sadly, relatively few tools have implemented automated stylistic error checking and even fewer are customizable or written for C. CritTer (Critique from the Terminal) fills this void. It provides a tool to check for administrator-defined stylistic errors in C code. CritTer uses a SAX style of event-based programming to perform checks and produces warnings as the code is being read. Administrators can use predefined checks or create their own to enforce coding standards. Additionally, they can use CritTer to help grade and teach "good style" to students. To test CritTer's abilities, I ran it over a series of graded student submissions and compared CritTer's performance to the grader's. The results proved that CritTer is both helpful and reliable. Not only did CritTer find 98.1% of errors, it had a a precision rate of 90.0%. These rates are excellent, especially given the grader found 83.6% of errors and had a precision of 100%.

Acknowledgements

This thesis has been simultaneously one of the hardest and most rewarding experiences of my academic career. Transforming a simple idea into a functional and, more importantly, useful program was extremely satisfying and is a boost to my confidence as I enter the professional world. I am thoroughly indebted to many individuals who supported me through this process. Unequivocally, I owe my greatest and most profound thanks to my advisor, Dr. Robert Dondero. Dr. Dondero patiently met with me every week this year and helped me in everything from research and general software development to programming design tactics and writing skills. Without his support and appreciation of good style, this thesis would never have made it to its current state.

I would also like to express my gratitude to Professor Brian Kernighan for helping me enter the Computer Science department my sophomore year and for putting up with my two Junior Projects. Without his help and support, I would not have become a successful CS major.

There are various individuals who have supported me who also deserve mention. I would like to thank Alice Zheng for her wonderful help in creating the CritTer logo. Ashton Brown and Slater Stich have been two of my closest friends and biggest supporters this year and throughout my entire college career. Without their help, I would not have been as productive or successful in my efforts on this thesis; without their friendship, my college career would have been significantly less enjoyable. I would also like to thank my teammates for being a wonderful, if often slightly annoying, set of brothers — I will actively miss all of our 'family' dinners next year. Additionally, I'd like to thank Marty Crotty, my coach, who has made me a better competitor and tougher person. Finally I would like to thank my parents and my sisters who constantly provide me with support and laughter. I owe all my success to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Writing typically contains three types of errors: syntactic, stylistic and semantic. In the case of writing prose, these errors take the form of spelling and grammar mistakes, poorly phrased passages and logic errors. When writing code, they form syntax errors, poorly styled code and malfunctioning code. Both spelling mistakes and syntax errors represent text that is not within the language (be it English, C, etc.). A poorly phrased passage in prose or code denote text that is technically valid but hard to understand. Finally, illogical arguments in prose and malfunctioning code both imply errors in the ideas behind the text. There are ways to find with these errors in both prose and code (see Table 1.1). While all these methods are useful, syntactic error checking is largely automated and therefore much more available and helpful. Good automated semantic error checking requires a currently unavailable level of artificial intelligence. Stylistic error checking, on the other hand, is feasible but has been addressed by very few tools.

Types of Errors	Tools for Prose	Tools for Code
Syntactic	Spell & Grammar Check	Compiler
Stylistic	Editor	Code Reviewers/Graders
Semantic	Reader	User/Tests

Table 1.1: Error Checking

From a different point of view, this problem can be formulated in terms of software quality. There are two perspectives on software quality: that of the user and that of the programmer. Users evaluate software on whether or not it behaves as it ought. Programmers, in addition, evaluate software on whether or not it is easily maintainable. Minimally, maintainability implies that code is easy to read and update. Evaluating a program from the user's perspective is common practice and most easily accomplished through automated testing. Though it is possible to evaluate a program from the programmer's perspective, existing tools that do so only check for specific qualities. Unfortunately, code quality is subjective, so any tool that only performs pre-defined inspections will never be satisfactory to every programmer.

The biggest reason to perform stylistic error checking is to improve readability (the ease with which another programmer can understand a piece of code). In the same way that poor phrasing in a paper often confounds its underlying arguments, poorly written code can easily obscure its underlying function. Furthermore, readable code is easier to revise and update.

In the academic world, professors and teaching assistants (TAs) often read students' code, especially in introductory level courses. In these courses, much of the focus is on enforcing "good style" (though the definition varies from professor to professor). The successful implementation of an automated stylistic error checker can immediately save work for professors by replacing the process of individually writing the same set of stylistic comments to multiple students with a set of automated warnings. In addition to reducing this repetitive and time consuming task, it also allows for a consistent evaluation. Students also directly benefit by applying this tool

to their code prior to submitting assignments — giving them the chance to improve their grades as well as their coding habits.

In an industrial setting, where it is necessary to read or edit another's code, maintaining readability is essential. Projects are often handed over to new employees or teams who are then expected to be able to contribute immediately. Poorly organized or written code makes this daunting task onerous. Many successful software companies make use of a codified internal style but the enforcement of this policy falls to the employees. Many transgressions are simply due to inattention and could easily be solved by an automated reminder system. Such a tool would improve readability and reduce the need to bother one's peers with another round of code reviews, thereby allowing the entire team to be more productive.

To address these needs, I have created CritTer (Critique from the Terminal), a customizable style-checker for C code. CritTer is run from the command line and executes a set of stylistic checks on the source files. Additionally, administrators can create checks to satisfy their personal needs.

Chapter 2

Related Products

Many tools exist to help improve code. Minimally, compilers often produce warnings about unused code or assignments within `if` statements. Tools like Clang[5] and Uno[15] go even further and look for bugs such as uninitialized variables, out-of-bounds array indexing and memory errors. These tools do not focus on style or readability explicitly and still largely operate on the same level as a compiler. Other tools try very hard to fill the stylistic error checking void. Each approaches the problem differently, but all succeed in finding some stylistic errors. Three such tools are Splint[10], PMD[4], and Checkstyle[2].

2.1 Splint

Splint is a tool for "statically checking C programs for security vulnerabilities and programming mistakes"[10, p. 9]. It works exactly as CritTer does from the user's perspective, i.e. as a command-line program which prints warnings to `stdout`. Splint displays warnings about basic stylistic errors such as assignments with mismatched types and ignored return values. With more effort, programmers can add annotations (fancy comments) to their code that gives Splint a specification against which to check. These annotations allow for stronger checks like memory management, null pointers and "violations of information hiding"[10, p. 9]. Examples of annotations in

```
typedef /*@abstract@*/ /*@mutable@*/ char *mstring;
typedef /*@abstract@*/ /*@immutable@*/ int weekDay;
```

Figure 2.1: Splint Annotations which define `mstring` and `weekDay` as abstract data types and further specify that they are mutable/immutable respectively.

action are shown in Figure 2.1. These checks supersede the set found in the original Lint, Splint's namesake: "Specification Lint" and "Secure Programming Lint".

While these annotations provide an extensive feature set, they are a huge inconvenience. They require programmers to specifically write their code to meet both the specification of the client and also of the tool. For new programmers — often the ones who need the most error checking — these annotations are almost impossible to implement on top of learning to program. David Evans, one of the authors of Splint, says as much in a private email. He states:

> One of the goals of the original design of Splint was for programmers who add no annotations to start getting some useful warnings right away, including warnings that encourage them to start adding annotations. For some aspects, such as `/*@null@*/` annotations I think this has worked okay, but for others like abstract types, memory management, etc., I don't think it has worked very well, and the warnings on these issues tend to either make developers want to stop using Splint, or at least just turn off all the warnings of that type, rather than start adding the annotations needed to enable better checking.[11]

Splint and CritTer differ in two significant ways. Splint performs a lot of inter-file checks regarding headers, interfaces, etc., whereas CritTer primarily focuses on intra-file checks. They also differ in how they specify what to check. Splint uses a configuration file and command line arguments to determine which of the several-hundred pre-defined messages and warnings to display. In contrast, CritTer allows

administrators to easily write their own checks and always runs every check that is defined. Because of this disparity, Splint is limited to checking for commonly accepted errors but CritTer has the freedom to operate idiosyncratically and check many different — possibly quite arbitrary — coding standards.

2.2 PMD and Checkstyle

PMD is a tool for checking Java code. It is integrated into a dozen or so popular IDEs. PMD comes with over 250 checks, which are mostly organized by purpose such as Braces Rules, Basic Rules, Coupling Rules, etc. Some checks also deal explicitly with a certain library or platform like Android, Jakarta and JUnit. PMD works by passing source code into a JavaCC-generated parser and receiving an Abstract Syntax Tree (a.k.a. AST, a tree-based model of the source code). PMD then traverses the AST and calls each rule to check for any violations. This pattern of examining a tree of nodes is called the Visitor Pattern[13]. Rules are written in their own classes and extend a base implementation. The rule itself can override three functions (start, visit and end) to perform various checks against the source code based on the nodes in the AST. The "dummy" example from the PMD website which counts how many expressions are in the source code is shown in Figure 2.2. PMD keeps track of these custom rules by reading additional XML files, called rulesets, which specify the various attributes of the rule (such as name, message, corresponding class, examples, etc.).

Checkstyle provides similar functionality to PMD in that it checks Java code for

```
package net.sourceforge.pmd.rules;

import java.util.concurrent.atomic.AtomicLong;
import net.sourceforge.pmd.AbstractJavaRule;
import net.sourceforge.pmd.RuleContext;
import net.sourceforge.pmd.ast.ASTExpression;

public class CountRule extends AbstractJavaRule {

  private static final String COUNT = "count";

  @Override
  public void start(RuleContext ctx) {
    ctx.setAttribute(COUNT, new AtomicLong());
    super.start(ctx);
  }

  @Override
  public Object visit(ASTExpression node, Object data) {
    // How many Expression nodes are there in all files parsed!
    RuleContext ctx = (RuleContext)data;
    AtomicLong total = (AtomicLong)ctx.getAttribute(COUNT);
    total.incrementAndGet();
    return super.visit(node, data);
  }

  @Override
  public void end(RuleContext ctx) {
    AtomicLong total = (AtomicLong)ctx.getAttribute(COUNT);
    addViolation(ctx, null, new Object[] { total });
    ctx.removeAttribute(COUNT);
    super.start(ctx);
  }
}
```

Figure 2.2: Example of a PMD rule which counts the number of expressions in the source code.

stylistic errors. It was designed to help programmers adhere to coding standards. Later, its designers added checks for bug prevention, class design problems, and other common errors. Accordingly, Checkstyle comes standard with many checks include those regarding duplicate code, class design, whitespace, etc. Like PMD, it uses an AST and the Visitor Pattern to check code. Custom rules are registered through an XML file and passed to Checkstyle at runtime. An example check which determines how many methods are in a class is shown in Figure 2.3.

PMD and Checkstyle are great tools; nevertheless, because they only work for Java, they do not solve my problem: stylistic error checking in C. In essence PMD, Checkstyle and CritTer perform very similarly; however, PMD and Checkstyle are built upon entirely different frameworks from CritTer. The use of the Visitor Pattern and an AST requires PMD and Checkstyle to read though the entirety of the code before they can produce any warnings. In contrast, CritTer performs error checking as it reads the code. PMD and Checkstyle also contain graphical user interfaces, both to aid writing checks and to find errors (the latter due to their integration with IDEs). CritTer, on the other hand, is a command line program. Another difference is CritTer must be recompiled in order to take advantage of any added checks as opposed to responding at runtime to a configuration file.

```
package com.mycompany.checks;
import com.puppycrawl.tools.checkstyle.api.*;

public class MethodLimitCheck extends Check
{
    private static final int DEFAULT_MAX = 30;
    private int max = DEFAULT_MAX;

    @Override
    public int[] getDefaultTokens()
    {
        return new int[]{TokenTypes.CLASS_DEF, TokenTypes.INTERFACE_DEF};
    }

    @Override
    public void visitToken(DetailAST ast)
    {
        // find the OBJBLOCK node below the CLASS_DEF/INTERFACE_DEF
        DetailAST objBlock = ast.findFirstToken(TokenTypes.OBJBLOCK);

        // count the number of direct children of the OBJBLOCK that
        // are METHOD_DEFS
        int methodDefs = objBlock.getChildCount(TokenTypes.METHOD_DEF);

        // report error if limit is reached
        if (methodDefs > this.max) {
            log(ast.getLineNo(),
                "too many methods, only " + this.max + " are allowed");
        }
   }
}
```

Figure 2.3: Example of a Checkstyle check which counts the number of methods in a class.

Chapter 3

What CritTer Does

CritTer reads in a set of C source code files and determines if they contain any of the defined stylistic errors. It is run from the command line inside one's working directory. CritTer is given a list of .c files to check and reads through each in the given order, pausing to read through included header files. Upon encountering an error, CritTer prints a warning to `stderr` containing the full location of the error, an error level and a message (an example is shown in Figure 3.1).

CritTer places the responsibility on the administrator to define the set of stylistic errors to check. CritTer comes with a set of predefined checks that the administrator may use or discard at his/her discretion. Checks are event driven and are called when the appropriate element in the code is reached. For example, it is often desirable to make sure that variables have long enough names to be adequately descriptive. In order to check this property, whenever CritTer recognizes a variable inside a declaration it checks that the variable's name exceeds a minimum length.

(a) test.c

```
92  for (int q = 0; q<5; q++) {
93    printf("hi");
94  }
```

(b) stderr

```
$ critTer test.c
test.c:92.22-92.23: big problem:
    Do not use magic numbers (5)
```

Figure 3.1: Example of an Error and Corresponding Warning. Here, CritTer is complaining that the for loop's exit condition contains a 'magic number'. The warning contains the location of the error, the error level as well as the message.

The administrator can write his/her own checks as functions to be invoked at each of the relevant callback points.

The predefined checks are listed in Table 3.1. These checks reflect two main ideas: to demonstrate the strength and abilities of CritTer as well as my own stylistic choices. In order to show off some of the power of CritTer, I wrote a variety of checks that show the various distinct elements that can be examined. Some checks, such as isFunctionTooLongByLines and isFunctionTooLongByStatements, are the same check defined on a different unit of length (line vs. statement). The style choices I made represent ideas from a variety of sources including Fowler's Bad Smells of Code[12], Google's style guide[20], Code Complete[18], C-Style: Standards and Guidelines[19], The Practice of Programming[8], PMD[4] and Checkstyle[2], as well as my own experience and preferences.

Check Name	Purpose
isFileTooLong	Check if the file exceeds a maximum length.
hasBraces	Check if the statement within an `if`, `else`, `for`, `while`, and `do while` statement is a compound statement.
isFunctionTooLongByLines	Check if a function exceeds a maximum line count.
isFunctionTooLongByStatements	Checks if a function exceeds a maximum statement count.
tooManyParameters	Check if there are too many parameters in the function declaration.
neverUseCPlusPlusComments	Warn against using C++ style single line comments.
hasComment	Check for comments before some construct.
switchHasDefault	Check that each `switch` statement has a `default` case.
switchCasesHaveBreaks	Check that each `switch` case has a `break`statement.
isTooDeeplyNested	Check whether a region of code (i.e. a compound statement) nests too deeply.
useEnumNotDefine	Warn against using `#define` instead of `enum` for declarations.
neverUseGotos	Warn against using `GOTO` statements.
isVariableNameTooShort	Check if a variable's name exceeds a minimum length.
isMagicNumber	Warn against using magic numbers outside of a declaration.
globalHasComment	Check if each global variable has a comment.
isLoopTooLong	Check if the loop length exceeds a maximum length.
isCompoundStatementEmpty	Check if the compound statement is empty.
tooManyFunctionsInFile	Check if there are too many functions in a file.
isIfElsePlacementValid	Warn against poor `if`/`else` placement as defined by the Google style guide.
isFunctionCommentValid	Check if function comments have the appropriate contents. Specifically check that the comment mentions each parameter (by name) and what the function returns.
arePointerParametersValidated	Check if each pointer type parameter into a function is mentioned within an `assert()` before being used.
doFunctionsHaveCommonPrefix	Check that function names contain a common prefix.
functionHasEnoughLocal-Comments	Check that there are enough local comments in the function relative to the number of control/selection statements.
structFieldsHaveComments	Check that all fields in a struct have a comment.

Table 3.1: Predefined Checks

Chapter 4

How CritTer Works

Figure 4.1 shows how CritTer is divided up into multiple, loosely coupled modules. Each of these has a unique purpose and is designed to keep the code as clean as possible. The "Knowledge Barrier" distinguishes the easily customizable and understandable modules from those which should not be modified without extreme caution. The modules can also be conceptually grouped into three categories: Parsing the Code (Lexer and Parser), Calling the Checks (Hooks and Sax), and Writing the Checks (Checks, Comments, and Locations).

4.1 Parsing the Code

CritTer is built on top of Flex and Bison, a lexical analyzer and parser generator, respectively. These two programs each take in a specification file (which defines a set of tokens and a corresponding context free grammar) and output a set of C files to parse code. Control goes back and forth between the lexical analyzer — which divides the code into distinct tokens — and the parser — which determines how the tokens fit together. CritTer is able to parse valid ANSI C code but it does not compile or in anyway track the contents. This means, for example, that CritTer sees any variable or function name as just as an `IDENTIFIER` (any set of letters[1] that does not

[1]Strictly, `IDENTIFIER`s fit the regular expression: `[a-zA-Z_]([a-zA-Z_]|[0-9])*`.

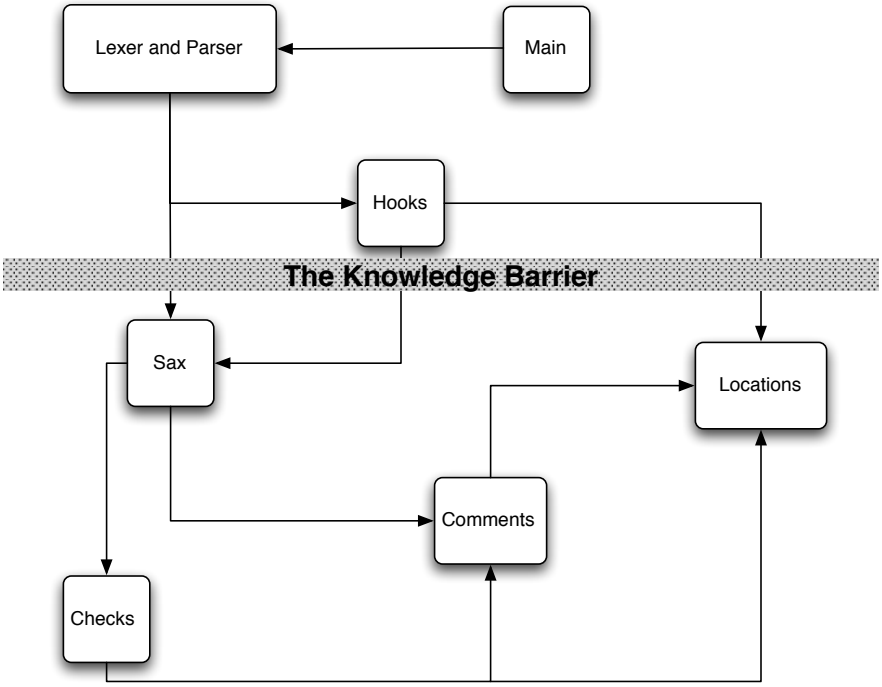


Figure 4.1: Module Interaction of CritTer. Arrows represent the flow of control between the modules.

already designate a data type) without any context as to where it was defined or used before. Because CritTer does not compile code, it cannot evaluate expressions within preprocessor directives. Therefore, CritTer cannot perform conditional compilation (specifically, it cannot follow `#if` and `#define`). In order to combat the issue of multiple inclusion of header files, CritTer stores the name of each file it opens and does not open that file again, even if it is included from another file. CritTer does not read standard header files (such as `stdlib.h` and `strings.h`) because they define data types within `#define`s (which CritTer cannot evaluate and therefore cannot recognize as types). For example, the file sys/cdefs.h, which stdio.h eventually includes, has

the line `#define __signed signed` and then uses `__signed` throughout the file. To adjust for this issue, the lexer instead contains a hack for determining if character strings in the code are `IDENTIFIER`s or data type names. The lexer does a string comparison against common types defined in standard headers such as `size_t`, `FILE`, `pid_t`, etc. If any of these hardcoded checks pass, then the lexer tells the parser it has found a type name instead of an `IDENTIFIER`.[2]

Bison and Flex track the location of any token or grammar construct. They store this information in a `YYLTYPE` structure. Normally a `YYLTYPE` contains 4 fields, `first_line`, `first_column`, `last_line` and `last_column`; however, I have also added a `filename` field in order to produce more accurate checks and warnings across a set of files. Each grammar rule can contain multiple actions that consist of C code. These actions can reference the location of the entire construct, or any of single component of it, through the prebuilt location mechanism. Event handlers are functions in the Sax and Hooks modules that respond to finding different code constructs. CritTer calls the event handlers from actions, passing in the location of the relevant text. Figure 4.2 shows an excerpt of the grammar where actions and the event handlers within the actions are underlined. Some actions are 'hidden' in dummy rules (called 'subroutines' by Bison) in order to avoid ambiguities within the grammar. Examples of this practice are shown in Figure 4.2 with the `beginCompound`, `beginFOR` and `beginIF` rules. Locations passed from the middle of a rule (all those passed to `begin` handlers) represent only the location of that segment and not the entire construct.

[2]The lack of preprocessing also means that escaped new lines are left in the code. If these characters occur inside a string, CritTer's lexical analysis fails, which in turn causes Bison to falsely report syntax errors. Since this style is rare (and often discouraged), I decided to ignore this issue.

For example, in Figure 4.2, `beginWhile` will only be passed the location of the word "`while`" whereas `endWhile` will be passed the location of the entire `while` statement.

Instead of writing them anew, I found Flex and Bison input files specifying the C language online[16] and modified them to add additional functionality. The only major modification of the actual grammar was to add the ability to recognize and dynamically add `typedef` definitions as types. CritTer stores these type names in an internal symbol table and the lexer checks to make sure that potential `IDENTIFIER`s are not already listed in the table. Additionally, I modified some of the grammar rules to include dummy rules with actions. In order to accommodate the inclusion of header files, I had to expand the given lexer functionality to transfer control between files. The specific method of using a stack of buffers and file pointers is heavily inspired by the examples in the O'Reilly Flex & Bison book[17]. When transferring to a different file, the lexer adds the current file to a stack with its file pointer, internal state, and current line number. When it reaches the end of the file, the lexer pops the current file off the stack and goes back to its previous state. The end of program occurs when there are no more files on the stack.[3]

[3]CritTer starts by adding all of the given files to the stack and dynamically adding additional header files. I chose to pre-load .c files in this manner because the mechanism was already in place and it simplified the interface between the main module and the lexer/parser. In order to hide this implementation detail, the .c files are pushed onto the stack in reverse order.

Additionally, the lexer reads in header filenames without any additional context about the path of the current file. Because of this, CritTer is unable to find and read header files that are included from within subdirectories. It is relatively rare within an academic context to break a single program into subdirectories and accordingly I decided not to focus on this issue.

```
beginCompound : /* empty */ {beginCompoundStatement(@$);}

compound_statement
  : '{' beginCompound '}'                    {endCompoundStatement(@$);}
  | '{' beginCompound statement_list '}'     {endCompoundStatement(@$);}
  | '{' beginCompound declaration_list '} ' {endCompoundStatement(@$);}
  | '{' beginCompound declaration_list statement_list '}'
      {endCompoundStatement(@$);}
  ;

beginIF : /*empty*/ {beginIf(@$);}

selection_statement
  : IF beginIF '(' expression ')' statement {endIf(@$);}
  | IF beginIF '(' expression ')' statement ELSE
      {endIf(@6); beginElse(@7);} statement {endElse(@9);}
  | SWITCH {beginSwitch(@1);} '(' expression ')' statement
      {endSwitch(@$);}
  ;

beginFOR : /*empty*/ {beginFor(@$);}

iteration_statement
  : WHILE {beginWhile(@1);} '(' expression ')' statement {endWhile(@$);}
  | DO {beginDoWhile(@1);} statement WHILE '(' expression ')' ';'
      {endDoWhile(@$);}
  | FOR beginFOR '(' expression_statement expression_statement ')'
      statement  {endFor(@$);}
  | FOR beginFOR '(' expression_statement expression_statement expression
      ')' statement {endFor(@$);}
  | FOR beginFOR '(' declaration expression_statement ')' statement
      {endFor(@$);}
  | FOR beginFOR '(' declaration expression_statement expression ')'
      statement  {endFor(@$);}
```

Figure 4.2: Excerpt of the Grammar. Here, the different 'paragraphs' are the different grammar rules. Actions are underlined and my additions to the grammar are in bold.

4.2 Some Theory

Bison uses an LALR(1) parsing algorithm, meaning it uses Left to Right, Rightmost Derivation to create a parse tree using one token of lookahead. This algorithm maintains a parser table which allows it to avoid backtracking as it parses the source file. Because of this property, calls are never mistakenly made from the lexer or parser into other modules of CritTer.

When compilers translate code into machine language, they first perform lexical analysis and parsing, just like CritTer. Where the two start to differ is in the actions for each grammar rule. Compilers store information regarding the semantic value, or meaning of the source code, inside each construct. This practice allows compilers to build an Abstract Syntax Tree, which "conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation"[7]. Other modules can then look over the entire tree and determine the meaning of the code as well as stylistic attributes. Both PMD and Checkstyle use this method.

Originally, CritTer operated in a similar manner; it stored an enumerated value corresponding to the type of construction as the semantic value of each node (see Appendix C). The checks then examined the value of each construction. Instead of expanding along this line of development, I decided to transition to an event-based system, largely inspired by SAX[1]. This kind of framework "reports parsing events (such as the start and end of elements) directly to the application through callbacks, and does not usually build an internal tree. The application implements handlers to deal with the different events, much like handling events in a graphical user interface"[1]. Specifically, SAX uses event handlers to parse XML files. There

are three main handlers which are used at the beginning and end of each XML element as well as to capture the text in between.

I chose to make this transition because the event-based system required far less overhead to implement. Instead of spending time building the tree framework and the corresponding methods to traverse it, I was able to focus on implementing stylistic checks. Additionally, the SAX-style framework has the added benefit of being able to examine code as it parses, as opposed to after the file has been parsed completely. This feature also increases scalability because the SAX framework discards the parts of the file(s) it has already read.

4.3 Calling the Checks

Much like SAX, CritTer calls event handlers at the beginning and end of constructs (functions, declarations, statements, parameter lists, etc.) as well as when it finds singular elements (variable names, `break` statements, parameters, etc.). In the code, handlers are prefaced by the words "`begin`", "`end`" and "`register`" to signal at what point each is called (as shown in Figure 4.2). At minimum, each handler is passed the location of the relevant text — in the case of `IDENTIFIER`s and numeric constants, the handler is also passed the relevant text itself. Each of these event handlers exists in the file sax.c which in turn calls the administrator-defined checks (as shown in Figure 4.3). While these checks could be written into the event handlers themselves, it is advantageous to separate them into their own functions in order to preserve the readability of the sax.c file and the code in general. Appendix A lists the predefined checks, their function signatures, and the handlers which call them.

```
void registerConstant(YYLTYPE location, char* constant) {
  isMagicNumber(location, MIDDLE, constant);
}

void beginCompoundStatement(YYLTYPE location) {
  isCompoundStatementEmpty(location, BEGINNING);
  lastCalled_set(beginCompoundStatement);
  isTooDeeplyNested(location, BEGINNING);
  isFunctionCommentValid(location, BEGIN_FUNCTION_BODY, NULL);
}

void endCompoundStatement(YYLTYPE location) {
  isCompoundStatementEmpty(location, END);
  lastCalled_set(endCompoundStatement);
  isTooDeeplyNested(location, END);
}

void beginDeclaration(YYLTYPE location) {
  isMagicNumber(location, BEGINNING, NULL);
  isVariableNameTooShort(location, BEGINNING, NULL);
}

void endDeclaration(YYLTYPE location) {
  isMagicNumber(location, END, NULL);
  globalHasComment(location, MIDDLE);
  isVariableNameTooShort(location, END, NULL);
}
```

Figure 4.3: A Subset of the Event Handlers in the Sax Module

Unfortunately, some handlers cannot actually be called at the time the construct is recognized. This is because Bison executes actions as they are encountered inside each grammar rule. If actions were placed at the beginning of a rule, Bison would not know which to act upon. In all the rules listed in Figure 4.2, the action is preceded by some distinguishing token (e.g. `WHILE`) or by the entire rule. However, Figure 4.4 shows some rules that both need actions at the beginning of the statement and lack distinguishing tokens. Specifically we would like to know when we start a function definition, but we cannot be sure that we are in a function definition until Bison finishes parsing the function's signature. To fix this issue I added the Hooks module. This module intercepts what would be normal calls within the SAX framework and then reorders them at the appropriate time. Each call into the Hooks module does one of two things: it enqueues a Sax level function call and its location or it dequeues any item after a specified location (Figure 4.5). With the beginning of a function, all the elements of the signature are placed on the queue and then dequeued when `h_beginFunctionDefinition` is called.

The Hooks module also makes the appropriate calls into the Sax layer regarding `IDENTIFIER`s and numeric constants. The lowest order structure Bison can manipulate is the token, meaning it cannot know the textual representation of a given `IDENTIFIER` or constant. Flex, on the other hand, operates on the actual text. Each module makes one call into Hooks for each `IDENTIFIER` or constant regarding the text or location. The Hooks module then takes the information from these separate calls and combines them into one call in the SAX layer. This can best be seen in Figure 4.6.

```
declarator
  : pointer direct_declarator
  | direct_declarator
  ;

direct_declarator
  : IDENTIFIER       {h_registerIdentifier(@$);}
  | '(' declarator ')'
  | direct_declarator '['  {h_beginDirectDeclarator(@1);}
      constant_expression ']'  {h_endDirectDeclarator(@$);}
  | direct_declarator '['  {h_beginDirectDeclarator(@1);} ']'
      {h_endDirectDeclarator(@$);}
  | direct_declarator '('  {h_beginDirectDeclarator(@1);}
      parameter_type_list ')' {h_endDirectDeclarator(@$);}
  | direct_declarator '('  {h_beginDirectDeclarator(@1);} identifier_list
      ')'   {h_endDirectDeclarator(@$);}
  | direct_declarator '('  {h_beginDirectDeclarator(@1);} ')'
      {h_endDirectDeclarator(@$);}
  ;

function_definition
  : declaration_specifiers declarator {h_beginFunctionDefinition(@2);}
      declaration_list compound_statement {endFunctionDefinition(@$);}
  | declaration_specifiers declarator {h_beginFunctionDefinition(@2);}}
      compound_statement {endFunctionDefinition(@$);}
  | declarator {h_beginFunctionDefinition(@1);} declaration_list
      compound_statement {endFunctionDefinition(@$);}
  | declarator {h_beginFunctionDefinition(@1);} compound_statement
      {endFunctionDefinition(@$);};
```

Figure 4.4: Additional Excerpt of the Grammar

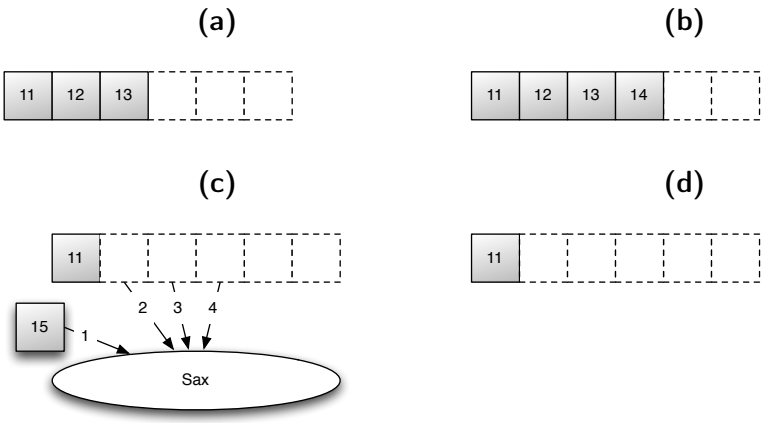


Figure 4.5: Representation of the Hooks Module. (a) The initial queue with functions associated to locations 11, 12 and 13. (b) The queue after another function/location pair has been enqueued. (c) The call at location 15 causes a call into the Sax layer followed by every stored call after the given location (12) in the queue. (d) The resulting queue.

Event handlers for major and common constructs, the beginning and end of each file, and the program have been implemented. Smaller items, including the handlers for registration of operators and data types, have yet to be implemented. It is easy to add more handlers; however, administrators should only attempt to do so after fully comprehending how the system works. Specifically, it is crucial to route handlers through Hooks only when they occur inside constructs which are already rerouted inside Hooks (such as statements and declarations). Otherwise events could be called in the wrong order (if they miss going through Hooks) or not at all (if they go through Hooks without anything to release them from the queue).

In addition to the basic SAX style system, I have implemented one shortcut to help identify code context without an excess of global variables. Every time a handler is called, it sets the `lastCalledFunction` through a setter. Checks can then use this

Hooks	Sax	Relevant Code
h_registerIdentifierText		`example`
h_registerIdentifier		`example`
h_beginParameterList		`(`
h_registerIdentifierText		`a`
h_registerIdentifier		`a`
h_registerParameter		`int a`
h_registerIdentifierText		`b`
h_registerIdentifier		`b`
h_registerParameter		`double b`
h_endParameterList		`)`
h_beginFunctionDefinition	beginFunctionDefinition	
	registerIdentifier	`example`
	beginParameterList	`(`
	registerIdentifier	`a`
	registerParameter	`int a`
	registerIdentifier	`b`
	registerParameter	`double b`
	endParameterList	`)`
N/A	beginCompoundStatement	`{`
...	...	`...`
N/A	endCompoundStatement	`}`
N/A	endFunctionDefinition	

Figure 4.6: Timeline of event handler calls into the hooks and sax module for: `void example(int a, double b){...}`

variable to easily figure out what the previous context was without additional calls or variables (as shown in Figure 4.7).

```
/**
 * Check if the compound statement is empty.
 */
void isCompoundStatementEmpty(YYLTYPE location, int progress) {
  static void (*context)(YYLTYPE);

  switch (progress) {
    case BEGINNING:
      context = lastCalled_get();
      break;
    case END:
      if (lastCalled_get() == beginCompoundStatement) {
        /* create a good error message */
        char *parent = NULL;

        if (context == beginIf) { parent = "if statements"; }
        else if (context == beginElse) { parent = "else statements"; }
        else if (context == beginFor) { parent = "for loops"; }
        else if (context == beginWhile) { parent = "while loops"; }
        else if (context == beginDoWhile) { parent = "doWhile loops"; }

        if (parent) {
          lyyerrorf(ERROR_HIGH, location, "Do not use empty %s", parent);
        } else {
          lyyerror(ERROR_HIGH, location,
                   "Do not use empty block statements");
        }
      }
      break;
    default:
      break;
  }
}
```

Figure 4.7: Example of a Check which Utilizes lastCalledFunction

4.4 Writing the Checks

Minimally, each check needs access to the location of the code construct in order to be able to produce a warning. Additionally, checks often need further information regarding the surrounding context of the possible error. A simple example is the check against using "magic numbers"[8, p. 19] (see Figure 4.8). Many programmers consider using numeric constants directly inside the code very poor style and recommend defining a symbolic constant to hold that value. Therefore CritTer should only throw a warning when it finds a magic number inside a normal statement as opposed to inside a declaration where it is necessarily defined. This check then needs to know every time a declaration begins and ends as well as each time a number is found. This contextual information can be stored in global variables in the Checks module or passed into the individual check through its parameters (as in Figure 4.8).

In order to throw a warning, the administrator can call one of three functions: `yyerror`, `lyyerror`, and `lyyerrorf`. Each of these functions prints a warning message to `stderr` preceded by the error's location in the code and an error level (as seen in Figure 3.1). `yyerror` and `lyyerrorf` are each wrappers to `lyyerror`, which takes in an error level (`enum errorLevel`), a location (`YYLTYPE`) and a warning message (`char *`). In essence, `lyyerror` is really a wrapper to `fprintf` and defines the formatting for the warnings messages and locations. Instead of receiving a warning message, `lyyerrorf` accepts a format string and a variable argument list, which it uses with `vsprintf` to create a warning message. It then passes the newly created message to `lyyerror` with the rest of its arguments. `yyerror` only accepts a warning message and calls `lyyerror` with Bison's internal location in the code and a default high

```
void isMagicNumber(YYLTYPE location, int progress, char* constant) {
  int acceptableNumbers[3] = {0, 1, 2};
  int numAcceptable = sizeof(acceptableNumbers)/sizeof(int);

  static int inDeclaration = 0;

  switch (progress) {
    case BEGINNING:
      inDeclaration++;
      break;
    case MIDDLE:
      if (lastCalled_get() == registerCase) {
        lyyerror(ERROR_HIGH, location, "Do not use magic numbers");
      } else if (inDeclaration == 0) {
        int number = (int)strtol(constant, (char**)NULL, 0);
        int i;

        /* see if number is within the acceptableNumbers array */
        for (i = 0; i < numAcceptable; i++) {
          if (number == acceptableNumbers[i]) {
            return;
          }
        }
        lyyerror(ERROR_HIGH, location, "Do not use magic numbers");
      }
      break;
    case END:
      inDeclaration--;
      break;
    default:
      break;
  }
}
```

Figure 4.8: CritTer check with additional context that throws a warning on encountering a magic number outside of a declaration.

```
/* Warn against using magic numbers outside of a declaration.        */
/* (Presumably, inside a declaration, a variable will be initialized */
/* to a magic number and then used throughout the rest of the code). */
```

Figure 4.9: Example of Adjacent Comments

error level. This is because `yyerror` is called internally through Bison to represent syntax errors. `yyerror` is the only predefined error reporting function; `lyyerror` is an extension suggested by O'Reilly[17] when using location tracking with Bison/Flex. The 'l' represents the variable location. `lyyerrorf` is modeled after `printf` and deals with formatted warning messages in one consolidated function.

Additionally, there are two helper modules designed to facilitate writing checks: Locations and Comments. The Locations module contains several functions to manipulate `YYLTYPE`s. Specifically it contains methods to compare locations as well as allocate, copy and free `YYLTYPE`s. The Comments module tracks all the comments found while running CritTer. Comments' contents and locations are registered through calls from the Sax layer and are stored in a dynamic array. Adjacent comments (like those in Figure 4.9) are recognized and combined into one larger comment. Additionally the module provides the ability to search through this array to find comments in or near a given location (using the methods from Locations). Finally, the Comments module provides two useful functions for analyzing the comment itself. The first determines if a comment has words — signifying that it is not a delimiting comment in the form of `/*----*/`. The second checks whether a comment contains a given string with or without case sensitivity.

CritTer has difficulty checking some coding styles. For example, when checking

that each global variable has a comment, it is trivial to check if there is a comment before that declaration. If, however, the comment appears after the declaration, as is common in some header files, CritTer is unable to find the comment. This is because, at the end of a declaration, CritTer has not yet read or stored the forthcoming comment. There are three solutions to this issue. The first is a creative hack in which one stores every location of a global variable and then searches for comments after each location at the end of the file. The second method would be to perform lookahead within the lexer to determine if a comment was about to follow. The third and preferred solution is to change the coding standard to have comments precede declarations.[4]

[4]Princeton University's Introduction to Programming Systems course[6] will be changing its coding standards this summer to facilitate using CritTer this fall and thereafter. One of the largest changes will be to put comments before function declarations in header files.

Chapter 5

How to Use CritTer

5.1 Users

Find out from your administrator where you should find their version of CritTer. After following their installation instructions, go to your working directory from the command line. Type "`critTer *.c`" (or if you only want to check one or two files, type their names instead of the `*.c`). CritTer will output any warnings about your code to `stderr`.

5.2 Administrators

5.2.1 Use

In academics, the best use of CritTer is as an automated grading system. It is simple to assign a point reduction system based on the number of warnings CritTer returns over a submission. For example, one might deduct a two point penalty per high error level message, a point per normal error level message and a half a point per low error level message. Not only does it reduce the work needed to grade a submission, but by allowing students to pre-check their work, the submissions become easier to read through good, consistent style.

In industry, CritTer should be used by programmers before submitting code for peer review. This creates an automated system to alert against any code that does not adhere to the accepted coding standard. In this way, CritTer helps make the code base more consistent and readable without direct peer enforcement. Furthermore, the team can be more productive when they spend less time correcting their peers' stylistic errors.

5.2.2 Customization

Before customizing the code, it is important to both understand how CritTer works (see Chapter 4) and have looked through the code conventions in Appendix B.

Add a Check

The first step of adding a new stylistic check is to determine precisely what you wish to check and which handlers will give you the necessary information. Then determine how much context is needed to implement this check. For example, to throw a warning on C++ style comments (comments in form of "`// Comment text which ends on a newline`") requires no context — it is only dependent on the existence of that code. In contrast, checking for braces around the content of `for` loops requires very minimal context: whether or not `endCompoundStatement` was the last function called (i.e. right before `endFor` was called, did CritTer encounter a '`}`' or something else). This minimal context can be established through the use of the `lastCalled_get()` function which returns a pointer to the last Sax handler called. More complex checks may need additional context. For example, to check that each

`switch` statement has a `default` case, the check needs to keep track of whether it has seen a `default` within the current `switch` block. The easiest way to do this is to have the check called from `beginSwitch`, `registerDefault` and `endSwitch` and pass in a different 'progress' value at each different call. Throughout the code, the enumerated values `BEGINNING`, `MIDDLE` and `END` provide such values (as shown in Figure 5.1).

After determining the relevant handlers and additional necessary parameters,[1] one must actually write the check. The easiest way to deal with contextual processing is to use a `switch` statement and conditionally set static local variables (Figure 5.1). In order to throw a warning, one must pass `location` to either `lyyerror` or `lyyerrorf` with an error level and either a message or format string and arguments respectively (for additional information see Section 4.4).

Add an Event Handler

In order to add an event handler, it is necessary to edit the grammar file. This is not trivial and should be undertaken with great care. Having said that, creating a handler itself is actually quite simple. The first step is to figure out which grammar rule(s) are relevant to the event you would like to capture. In some cases this is incredibly trivial, in others it takes some effort to understand what the grammar is describing. In my experience, the best method to figure out the various rules is to perform a manual depth-first-search through the different components of the rule until it becomes clear.

[1] Each check should have at least one parameter: `YYLTYPE location`. This value is necessary in order to produce a proper warning message. All other parameters are optional and should follow `location`. Many checks can be completed using only a progress value or informative string.

```
/**
 * Check that each switch statement has a default case.
 */
void switchHasDefault(YYLTYPE location, int progress) {
  static int started = 0;
  static int found = 0;

  switch (progress) {
    case BEGINNING:
      started = 1;
      found = 0;
      break;
    case MIDDLE:
      found = 1;
      break;
    case END:
      if (!found && started) {
        lyyerror(ERROR_HIGH, location,
                 "Always include a default in switch statements");
      }
      started = 0;
      break;
    default:
      break;
  }
}
```

Figure 5.1: CritTer Check with Contextual Processing. `switchHasDefault` is called from `beginSwitch` with `BEGINNING`, `endSwitch` with `END` and `registerDefault` with `MIDDLE`.

After finding the grammar rule, adding a `register` or `end` handler is very simple: define the handler in sax.c/h and add the action "`{newHandler(@X);}`" after the component you want to recognize. The `@X` references the location of either the component (where `X` = the number of the component) or the entire rule (where `X` = '`$`'). Figure 4.2 and Figure 4.4 show examples of these calls. Adding `begin` handlers can be much more difficult than the previous cases although, in principle, the process is identical. This is due to the possibility of adding ambiguities to the grammar.[2] When this happens, Bison will throw several errors during compilation and the parser will most likely break. The first tactic to avoid this issue is to never place actions as the first element in a rule; they should always appear after (at least) one component. If this approach is insufficient, you should try burying the action inside a dummy rule (such as `beginCompound`, `beginIF`, and `beginFOR` in Figure 4.2).

There are only two reasons to route your new handler through the Hooks module instead of going directly to Sax; the most likely reason is the event occurs inside a construct that already goes through Hooks. Declarations, function signatures, and statements currently go through Hooks. This means that events like `registerConst` must also go through Hooks in order to be released to Sax at the right time (see Section 4.3). The second motive is the reason why those constructs already go through Hooks: it is the only way of getting an accurate `begin` handler. By this, I mean it is either impossible or exceptionally complicated to create a `begin` handler in the correct place in the grammar such that it is executed before all of its components. These constructs dequeue all the appropriate previous calls once the `h_endXX` handler

[2]The shift/reduce conflict for if-else statements was removed by giving explicit precedence for if-else statements over if statements as suggested by O'Reilly[17, p. 188].

is called.

If you do not need to go through Hooks, after defining the action in the grammar file and the handler in sax.c/h, all you need to do is call `lastCalled_set()` from the handler. If the new handler needs to go through Hooks, you need to create two handlers: `newHandler` in Sax and `h_newHandler` in Hooks (where `h_newHandler` is called from the action in the grammar file). If the event just needs to be released at the correct time (i.e. it appears within a hooked construct), the Hooks handler should call `enqueueFunctionAndLocation` to enqueue the Sax handler. If the handler needs to dequeue some elements, it should call `dequeueUntil`, followed by the sax handler and `lastCalled_set()`. Examples of both direct and indirect routes from the Lexer/Parser to Sax are shown in Figure 5.2.

5.2.3 Compilation, Testing and Installation

CritTer contains a Makefile which contains targets for compilation, testing and installation. To compile the given version of CritTer (or a customized version without additional files), simply type "`make`". To compile CritTer with additional files, edit the `all` target to include the new files and then type "`make`". To install CritTer (i.e. copy into `/usr/local/bin/`) type "`make install`".

Testing can be accomplished by typing "`make test`" which uses two shell scripts to run the local version of CritTer over a set of test files and then compares the new output to the previous output. The first script, `runOnTests.sh`, has a set of paths over which to run CritTer. The error messages are piped to `output.txt` after the old output has been copied to `output_old.txt`. The second script, `checkTestOutput.sh`,

(a) Direct Grammar

```
iteration_statement
  : WHILE {beginWhile(@1);} '(' expression ')' statement   {endWhile(@$);}
```

(b) Direct Sax Level Event Handlers

```
void beginWhile(YYLTYPE location) {
  lastCalled_set(beginWhile);
  functionHasEnoughLocalComments(location, MIDDLE, 0);
}

void endWhile(YYLTYPE location) {
  hasBraces(location, "while");
  lastCalled_set(endWhile);
  isLoopTooLong(location);
}
```

(c) Indirect Grammar

```
parameter_list
  : parameter_declaration {h_registerParameter(@$);}
```

(d) Indirect Hooks Level Event Handlers

```
void h_registerParameter(YYLTYPE location) {
  enqueueFunctionAndLocation(registerParameter, location);
}
```

(e) Indirect Sax Level Event Handlers

```
void registerParameter(YYLTYPE location) {
  tooManyParameters(location, MIDDLE);
  arePointerParametersValidated(location, REGISTER_PARAM, NULL);
}
```

Figure 5.2: Direct vs. Indirect Event Handlers. (a, b) A while statement has a direct call into the Sax module. (c, d, e)The registration of function parameters needs to indirectly route through Hooks in order to have the Sax level event handlers called after beginFunctionDefinition.

uses a list of all the warning messages and `grep` to break apart the output files check by check. The script then `diff`s each section of the files to determine if a check has been broken. In order to add new checks to the testing mechanism, one simply needs to add the check's function name and a significant (non-variable) part of the warning message into the appropriate arrays inside `runOnTests.sh`.

Chapter 6

Evaluation

In order to evaluate of CritTer's performance, Dr. Robert Dondero graded 10 randomly chosen final project submissions for Princeton University's Introduction to Programming Systems (COS 217) [6]. There were over 650 assignments available, each of which was anonymized in order to protect the students' privacy. Dr. Dondero has taught the course for many years, making him the perfect person to judge each submission's style. We judged CritTer's performance compared to "true" errors, which were determined after both Dr. Dondero and CritTer looked at each submission. This post-analysis judgement of errors was necessary in order to properly take into account errors that Dr. Dondero originally missed and CritTer found. We did allow CritTer to perform an iterative analysis of the submissions as a whole, mirroring the development process we hope other administrators will go through as they develop their own checks. The iterative process mostly affected checks like `functionIsTooLong` where there was a threshold value that needed to be tuned.

We measured two properties: precision and recall. Precision represents what fraction of the output was in response to a true error. In our experiment, Dr. Dondero will always have a precision of 100%.[1] Recall is an idea from information retrieval to represent the fraction of relevant documents that were retrieved. Here we use it to

[1] It is possible for Dr. Dondero to lower this value by changing his mind regarding an error but this never happened in practice.

		Was an error present? Yes	No
Did CritTer report an error?	Yes	46	11
	No	9	
Did Dr. Dondero report an error?	Yes	33	0
	No	22	

Table 6.1: Test Results for Magic Numbers. For this check, CritTer had a recall of 83.6% ($\frac{46}{46+9}$) and a precision of 80.7% ($\frac{46}{46+11}$). Dr. Dondero had a recall of 60.0% ($\frac{33}{33+22}$).

mean the number of true errors that were found. Throughout CritTer's development, we considered recall to be more important than precision. We prefer CritTer to find all the true errors and produce extraneous output rather than missing some of the true errors and minimizing extraneous output. We think it is better for students and instructors to have to defend their code rather than letting possible errors slide.[2] The checks I wrote reflect this preference. However, other administrators can write more 'conservative' checks which alter this relationship between recall and precision.

Four checks were very useful and represented about half of all of CritTer's output: magic numbers, validating pointer parameters, comments above global variables, and validating function comments. The check for magic numbers was one of the qualitatively hardest for Dr. Dondero to perform because numerals do not stand out from code. This is shown in the data in Table 6.1 where Dr. Dondero missed almost as many magic numbers as he caught. This check is also interesting in that, by its very nature, it will miss some errors; specifically when 0, 1 or 2 is used 'magically'.

[2]The notable exception is the check for magic numbers in which producing warnings against the use of 0, 1 and 2 in the code would far outweigh the few times those numbers would be used 'magically'.

		Was an error present? Yes	No
Did CritTer report an error?	Yes	102	21
	No	9	
Did Dr. Dondero report an error?	Yes	74	0
	No	37	

Table 6.2: Test Results for Validating Pointer Parameters. For this check, CritTer had a recall of 91.9% and a precision of 82.9%. Dr. Dondero had a recall of 66.7%.

CritTer checks whether functions validate pointer parameters inside an `assert()` before the parameter is actually used. The check fails systematically in two cases. The first is that CritTer will output a warning when the parameter is being properly validated through code like '`if (param != NULL)`'. This is unavoidable as CritTer does not detect the meaning behind the code. The second systematic failure involves opaque pointer types; CritTer has no way of knowing whether a newly defined type is a wrapper around a pointer. Therefore, it does not make sure parameters of those types are validated. Data are shown in Table 6.2.

One of the more interesting checks looked for comments above each global variable. Yet again, CritTer came up against systematic failure. While CritTer found each missing comment, it had a large tendency to output extraneous warnings. There were two main causes: self commenting code and an uncheckable coding standard. Many times, CritTer encountered global variables similar to "`enum BOOLEAN {FALSE, TRUE}`". Declarations of this kind are self-commenting and further comments would decrease readability. Warnings to add comments are therefore extraneous. The other systematic failure was due to a previously acceptable standard of putting comments

		Was an error present? Yes	No
Did CritTer report an error?	Yes	76	53 (224)
	No	0	
Did Dr. Dondero report an error?	Yes	62	0
	No	14	

Table 6.3: Test Results for Comments Above Global Variables. The number in the parenthesis in upper right corner represents the raw number of errors reported; the first number is the number of errors reported which were not solved after super imposing the upcoming change in coding standards. For this check, with the raw data, CritTer had a recall of 100% and a precision of 25.3%. After filtering the data, CritTer had a precision of 58.9%. Dr. Dondero had a recall of 81.6%.

beneath declarations. As mentioned in Section 4.4, CritTer is unable to associate comments beneath a declaration to that code. Because of this, and the upcoming change to the COS 217 coding standard, Dr. Dondero and I felt that it was inappropriate to include these results against CritTer. To accomplish this, we have filtered the data to remove the 224 extraneous warnings due to comments placed after global declarations. Table 6.3 shows the data both before and after filtering.

The check to validate function comments is an ideal check. One of Dr. Dondero's biggest time drains in grading assignments is checking that each function has an appropriate comment. CritTer is able to easily check whether the function has a comment and whether that comment refers to each of its parameters and its return value. More importantly, CritTer had a perfect record in finding these errors within our dataset as seen in Table 6.4.

In addition to those mentioned above, CritTer ran 16 checks over the dataset. In some cases, like useEnumNotDefine and functionIsTooLong, CritTer and Dr. Don-

			Was an error present? Yes	No
Comment Present?	Did CritTer report an error?	Yes	138	0
		No	0	
	Did Dr. Dondero report an error?	Yes	131	0
		No	7	
Refers to Parameters?	Did CritTer report an error?	Yes	36	0
		No	0	
	Did Dr. Dondero report an error?	Yes	18	0
		No	18	
Refers to Return Value?	Did CritTer report an error?	Yes	9	0
		No	0	
	Did Dr. Dondero report an error?	Yes	4	0
		No	5	
Totals:	Did CritTer report an error?	Yes	183	0
		No	0	
	Did Dr. Dondero report an error?	Yes	153	0
		No	30	

Table 6.4: Test Results for Validating Function Comments. For this check, CritTer had a recall of 100% and a precision of 100%. Dr. Dondero had a recall of 83.6%.

dero found the same set of errors. In other cases, as with neverUseCPlusPlusStyleComments and switchHasDefault, Dr. Dondero did not find any errors yet CritTer found a set with 100% precision. Several checks did not have enough data points to perform any meaningful analysis. For example, tooManyFunctionsInFile, tooManyParameters, fileIsTooLong, isCompoundStatementEmpty and switchCasesHaveBreaks each had less than 4 warnings. In some cases, specifically for isTooDeeplyNested, Dr. Dondero agreed that CritTer's output was correct even if there wasn't a specific remedy for the code. During the term, Dr. Dondero normally solves this issue by telling the student that "it would be better to refactor the code, but it's not clear how".

In going over the graded assignments, it became clear that CritTer had not yet been tuned to find all the desired types of style errors. One of the most prevalent ignored issues were lines that exceeded a maximum length. CritTer can be configured to check for this error by editing the code in the lexer which tracks the column position of the source file. Another useful check would be to recognize duplicate definitions of the same `struct` in different files. While this would be possible to implement in CritTer by storing all references to `struct`s, it would not be a simple process. Splint, however, does check for this type of error and can be used in conjunction with CritTer to provide a more complete set of checks. Other issues that appeared were: missing `#include` guards, needing to create an opaque pointer type from a `struct`, and poor indentation.

Overall, CritTer did very well. CritTer produced a total of 1226 warnings over the entire dataset. 951 of these represented true stylistic errors in the code. CritTer's recall was 14.5% higher than Dr. Dondero's and maintained a relatively high precision

		Was an error present? Yes	No
Did CritTer report an error?	Yes	933	104 (275)
	No	18	
Did Dr. Dondero report an error?	Yes	796	0
	No	155	

Table 6.5: Test Results Across All Checks. Across all checks, CritTer had a recall of 98.1% and a precision of 77.2%. After filtering the data, CritTer had a precision of 90.0%. Dr. Dondero had a recall of 83.6%.

of 77.2%. After filtering, CritTer's precision jumped to 90.0%. Dr. Dondero and I both judged that precision as excellent. Data, before and after filtering, can be seen in Table 6.5. These tests conclusively prove the benefit that CritTer can provide in terms of automatic style checking. Even without filtering the data, over three quarters of CritTer's output was pertinent. Furthermore, it increased error detection by 16.3%.

Given these impressive results, it is clear CritTer can help fill the automated stylistic error checking gap. CritTer can definitely help Professors (and their TAs) grade assignments both by saving time and increasing the number of stylistic errors found. CritTer can also help students improve their coding habits and their grades. Additionally, CritTer's impressive performance and high degree of customizability give us reason to believe that it will be useful to the world outside of academia. Companies with a defined coding standard can customize CritTer to help clean up their code base and ease the code review process.

Appendices

Appendix A

Predefined Check Functions and their Relevant Event Handlers

Check Function & Purpose	Relevant Sax Handlers
`isFileTooLong(YYLTYPE location)` Check if the file exceeds a maximum length.	endFile
`hasBraces(YYLTYPE location, char* construct)` Check if the statement within an `if` statement, `else` clause, `for` statement, `while` statement, and `do while` statement is a compound statement.	endWhile, endDoWhile, endFor, endIf, endElse
`isFunctionTooLongByLines(YYLTYPE location)` Check if a function exceeds a maximum line count.	endFunctionDefinition
`isFunctionTooLongByStatements(YYLTYPE location, int progress)` Checks if a function exceeds a maximum statement count.	beginFunctionDefinition, endFunctionDefinition, endStatement
`tooManyParameters(YYLTYPE location, int progress)` Check if there are too many parameters in the function declaration.	beginParameterList, registerParameter, endParameterList
`neverUseCPlusPlusComments(YYLTYPE location)` Warn against using C++ style single line comments.	N/A (Called from the Lexer)
`hasComment(YYLTYPE location, char* construct)` Check for comments before some construct.	endFile (also from `globalHasComment`)

Continued

Check Function & Purpose	Relevant Sax Handlers
`switchHasDefault(YYLTYPE location, int progress)` Check that each `switch` statement has a `default` case.	beginSwitch, registerDefault, endSwitch
`switchCasesHaveBreaks(YYLTYPE location, int progress, int isCase)` Check that each `switch` case has a `break` or `return` statement.	beginSwitch, registerDefault, registerCase, registerBreak, registerReturn, registerReturnSomething, endSwitch
`isTooDeeplyNested(YYLTYPE location, int progress)` Check whether a region of code (i.e. a compound statement) nests too deeply.	beginCompoundStatement, endCompoundStatement
`useEnumNotDefine(YYLTYPE location, int progress)` Warn against using `#define` instead of `enum` for declarations.	registerDefineIntegralType
`neverUseGotos(YYLTYPE location)` Warn against using `GOTO` statements.	registerGoto
`isVariableNameTooShort(YYLTYPE location, int progress, char* identifier)` Check if a variable's name exceeds a minimum length.	registerIdentifier, beginDeclaration, endDeclaration
`isMagicNumber(YYLTYPE location, int progress, char* constant)` Warn against using magic numbers outside of a declaration.	registerConstant, beginDeclaration, endDeclaration

Continued

Check Function & Purpose	Relevant Sax Handlers
`globalHasComment(YYLTYPE location, int progress)` Check if each global variable has a comment.	beginFunctionDefinition, endFunctionDefinition, endDeclaration
`isLoopTooLong(YYLTYPE location)` Check if the loop length exceeds a maximum length.	endWhile, endDoWhile, endFor
`isCompoundStatementEmpty(YYLTYPE location, int progress)` Check if the compound statement is empty.	beginCompoundStatement, endCompoundStatement
`tooManyFunctionsInFile(YYLTYPE location, int progress)` Check if there are too many functions in a file.	endFile, beginFunctionDefinition
`isIfElsePlacementValid(YYLTYPE location, int progress)` Warn against poor `if`/`else` placement as defined by the Google style guide.	endIf, beginElse
`isFunctionCommentValid(YYLTYPE location, enum commandType command, char* text)` Check if function comments have the appropriate contents. Specifically check that the comment mentions each parameter (by name) and what the function returns.	beginFunctionDefinition, endFunctionDefinition, beginParameterList, endParameterList, registerIdentifier, beginCompoundStatement, registerReturnSomething
`arePointerParametersValidated(YYLTYPE location, enum commandType command, char* identifier)` Check if each pointer type parameter into a function is mentioned within an `assert()` before being used.	beginFunctionDefinition, endFunctionDefinition, beginParameterList, registerParameter, endParameterList, registerIdentifier, registerPointer, endStatement

Continued

Check Function & Purpose	Relevant Sax Handlers
`void doFunctionsHaveCommonPrefix(YYLTYPE location, int progress, char* identifier)` Check that function names contain a common prefix.	beginProgram, endProgram, endFile, beginFunctionDefinition, endFunctionDefinition, registerIdentifier
`functionHasEnoughLocalComments(YYLTYPE location, int progress, int isComment)` Check that there are enough local comments in the function relative to the number of control/selection statements.	beginComment, beginFunctionDefinition, endFunctionDefinition, beginWhile, beginDoWhile, beginFor, beginIf, beginSwitch
`structFieldsHaveComments(YYLTYPE location, int progress)` Check that all fields in a struct have a comment.	beginStructDefinition, registerStructField, endStructDefinition

Appendix B

Conventions and Necessities

B.1 General

- Use the DynArray class to handle persistent arrays, stacks and queues. It is a dynamically growing array which holds void pointers. This is the base implementation of how comments are stored as well as the Hook module queues.
- To change the tab size, edit the `count` function inside c.l.
- To change the string representation for each error level, edit the `lyyerror` function in c.y. Currently, `ERROR_HIGH` = "big problem", `ERROR_NORMAL` = "error", and `ERROR_LOW` = "low priority".
- Make sure that each `IDENTIFIER` in the grammar is either registered through `h_registerIdentifer` or is explicitly ignored through `h_ignoreIdentifierText`. Otherwise the wrong identifier text will be dequeued and passed into the Sax module. This is because the lexer will always enqueue the `IDENTIFIER`'s text before the grammar recognizes it.
- Throughout the code, 0 and 1 are used synonymously for false and true respectively whereas `NULL` is used for pointers.

B.2 Sax and Hooks Modules

- When adding new calls, use the "`begin`", "`register`" and "`end`" prefixes where `begin` and `end` are used with larger constructs and `register` is used with constructs that are conceptually a single item (like a parameter).
- Always make sure to add the `lastCalled_set` function for any handlers that do not go through the Hooks module.
- Be careful of doing memory management at a file level. Most of it is done at a program wide level because files are added onto a stack. This means that for three files, CritTer sees three calls to `beginFile` before any of the `endFile` calls. This makes it far easier to have allocation and releasing of memory at the single `beginProgram` and `endProgram` calls.
- Keep all actual checks outside of the Sax handlers to improve code readability.
- Prefix all calls that go through the Hooks module with "`h_`".

B.3 Checks Module

- All warning messages should be passed to their respective function without an ending newline; one will be added so that each warning appears on one line.
- Use local static variables as opposed to global variables to determine context within checks.

- Phrase check function names as questions if there is something to check or commands if there is an automatic warning. For example, if it is never acceptable to use `GOTO` statements, phrase the check as a command like "neverUseGotos".

- To compare entire locations use the methods in the Locations module. Comparing single elements can be done inline.

- Be careful not to free things which were either never allocated (e.g. the current location passed to the Sax/Hook handler), might be shared between objects (e.g. filename strings) or will be freed on its own later (e.g. comment texts and locations).

- When adding a new check, add the function name and a significant (non-variable) part of the warning message to the arrays in `runOnTests.sh`. If changing the warning message of an existing check, make sure to update the arrays.

Appendix C

Progression of Development

CritTer's development has been reasonably linear and has consisted of slowly adding modules as new situations arose. Getting Bison and Flex to parse the sample C code took a significant portion of time at the beginning, especially the added elements such as tracking `typedef`s and dynamically reading header files. Once the code could be read without issue, the next step was to start adding some minimal checks.

The first version of CritTer consisted of the Main, Lexer and Parser, and Checks modules (although they were not conceived as such at that point). At this stage, Checks contained four checks which were called directly from the actions in the grammar (Figure C.1). Minimal comment tracking had been implemented inside the Checks module within a single function. Context was determined by setting an enumerated value for the statement as a whole (underlined in Figure C.1). While this initial version was functional, it involved tedious manipulation of the grammar as well as very little ability to perform contextual processing.

This dependency on the grammar file was the motivation to change the framework in version 2. The first step was to adopt the SAX style of processing. The alternative was to try to use an AST and Visitor Pattern (like PMD and Checkstyle) but this would have involved a lot of overhead to write the framework code. The SAX style is relatively lightweight and was the far easier (and simpler) alternative. The transition from version 1 to version 2 was fairly straightforward and immediately allowed for

```
selection_statement
  : IF '(' expression ')' statement {$$ = IF_SELECTION; ifHasBraces($5,
      @$);}
  | IF '(' expression ')' statement ELSE statement
      {$$ = IF_ELSE_SELECTION; ifHasBraces($5, @$); ifHasBraces($7, @$);}
  | SWITCH '(' expression ')' statement
  ;
```

Figure C.1: Version 1.0 Grammar Excerpt where `ifHasBraces` is a check that determines if an `if` statement had braces.

additional checks to be added. The comment tracking system was moved into its own module and all of the previous enumerated value contexts were removed. However, in adding more checks, I realized that some calls into the Sax module were simply not going to be in the right order.

This problem prompted final version (3) which added the Hooks module. The inspiration behind the Hooks module was the queue of function pointers. The insight of storing the pointers to the event handlers made this entire implementation (and version) possible. After adding the Hooks module, I was able to add even more checks and focus on cleaning up the code. In the process of adding more checks, I decided to store the last called event handler in order to minimize the number of handlers any given check needed to be called from. This eventually led to the `lastCalledFunction` methods. At this point I moved all the code regarding `YYLTYPE`s into the Locations module and added finer grain methods to find relevant comments by location.

Bibliography

[1] SAX. http://sax.sourceforge.net/. 18

[2] Checkstyle. http://checkstyle.sourceforge.net/index.html, 2010. 4, 11

[3] Frama-C. http://frama-c.com/index.html, 2010.

[4] PMD. http://pmd.sourceforge.net/, 2010. 4, 11

[5] Clang Static Analyzer. http://clang-analyzer.llvm.org/, 2011. 4

[6] Computer Science 217: Introduction to Programming Systems. http://www.cs.princeton.edu/courses/archive/spring11/cos217/, 2011. 30, 39

[7] A.W. Appel. *Modern compiler implementation in ML*. Cambridge University Press, 2004. 18

[8] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 2007. 11, 27

[9] D. Brown, J. Levine, and T. Mason. *Lex & yacc*. O'Reilly Media, Inc., 1992.

[10] D. Evans. *Splint Manual*. Splint, 3.1.1-1 edition, June 2003. 4

[11] D. Evans. Splint and Senior Thesis. 09 2010. 5

[12] M. Fowler, K. Beck, J. Brandt, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 2008. 11

[13] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Boston, 2008. 6

[14] GNU Bison. *Bison*, 2.4.3 edition, 2010.

[15] G. J. Holzmann. UNO: Static Source Code Checking for UserDefined Properties. In *6th World Conf. on Integrated Design and Process Technology, IDPT '02*, 2002. 4

[16] J. Lee and J. Degener. ANSI C Yacc grammar. `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`, 1995. 16

[17] J. R. Levine. *Flex & bison: [Unix text processing tools]*. O'Reilly, 2009. 16, 29, 35

[18] S. McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2 edition, 2004. 11

[19] D. Straker. *C-Style: Standards & Guidelines*. Prentice Hall, 1992. 11

[20] B. Weinberger, C. Silverstein, G. Eitzmann, M. Mentovai, and T. Landray. Google C++ Style Guide. `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`. 11