Call Graph Reduction by Static Estimated Function Execution Probability

LO, Kwun Kit

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

©The Chinese University of Hong Kong
February, 2009

Thesis / Assessment Committee

Abstract

Abstract of thesis entitled:
Call Graph Reduction by Static Estimated Function Execution Probability
Submitted by LO, Kwun Kit
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in August 2008

When looking at an unfamiliar code base, developers have difficulty using intuition to guess which functions are central to the functionality of code. Tools such as reverse engineering tools, and feature, concern or aspect mining tools are helpful for providing structural insight into the system, and may provide support for developer's intuition, but they do not reveal which functions in the code are of central importance to the functioning of the system.

Instead, developers need to be provided a view which isolates important functionality in the code. Call graph reduction techniques attempt this, by removing functions that are likely to be peripheral to the system's functionality. The current state of the art approach attempts to do this by removing small functions, and helper functions. Though this does result in some isolation of the important functions, it leaves many non-core functions present in the call graph.

The thesis of this research is that we can provide a better isolation of important functions in the call graph by trimming it down to only functions that have a high probability of execution. We believe that this will provide better accuracy of included functions, and will result in a smaller call graph.

We evaluate these claims by comparing the execution probability approach against current the state of the art reduction technique, and against techniques that mimic developer intuition.

摘要

在處理不熟悉的源碼的時候，開發人員的直覺往往不能準確判斷哪些函數在系統功能上是比較重要。儘然逆向工程工具 ，以及一些關於功能、關注點或側面挖掘的工具可以幫助開發人員理解系統結構，這些工具並不能提供開發人員哪些函數是比較重要的資訊。

開發人員需要一些將重要函數從源碼中抽離的方法，而函數調用關係圖精簡化技術便是試圖以移除與系統功能不太相關的函數來達成此目的。現行的方法是將較小的函數以及輔助函數從函數調用關係圖中移除，雖然這可以將核心函數從中抽離，但不少不太相關的函數依然留存在函數調用關係圖之上。

本論文的研究就是將函數調用關係圖精簡至僅僅保留高調用機率的函數，從以更有效地將重要函數從源碼中抽離。我們相信這種方法可以更準確地精簡函數調用關係圖，並更有效地縮減其大小。

我們通過對現行的技術，以及仿效開發人員直覺判斷的技術，與我們的調用概率方法進行比較，以評估本論文的主張。

Acknowledgments

I would like to express my gratitude to my supervisor, Professor Elisa Baniassad, for her patient support, guidance and encouragement along the way. This dissertation would not have been possible without her assistance and help. I must thank Elisa for her teaching and advice during my master as well as undergraduate studies.

I also thank the members of my thesis examination committee: Professors Lap-chi Lau, Ho-fung Leung and Robert Walker, for their helpful comments in improving this thesis.

Thanks are given to Alan Chu and Jacky Chan for their insightful comments and advices during my study. They are the kind of persons that are knowledgeable in almost any topics that you can imagine. Also thanks to Stephen Leung, who inspired me a lot in the technical part of my research.

Finally, I would like to thank my family for their continuous encouragement.

KWUN KIT, LO

The Chinese University of Hong Kong
DECEMBER 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Understanding the calling pattern of subroutines is a fundamental activity for understanding an unfamiliar code base. Call graphs are a visualization of call relationships between subroutines. They have been long recognized as a useful structure for code based understanding[Ryd79]. However, as our preliminary study in Chapter 2 showed, in the face of large call graphs, developers' intuition-based examination does not always result in identification of functions important for understanding a system's core functionality.

We believe that this inability can be attributed to the large number of functions in the call graph that are not core to the functionality of the program. As would be expected, different functions have different levels of significance to the major functionality of the system. Some are important to the core functionality, and are important in learning the basic functioning of the system, while others, such as helper functions, are less central.

There has been some investigation into reducing the call graph to help focus the developer's attention on important functionality. Ying and Tarr [YT07] provide a technique that filters out functions from the call graph if they are small, or if they are near a leaf node of the call graph.

While their heuristics have shown benefits program investigation for developer in navigation and performing change tasks, their simple, though elegant, technique has some inherent drawbacks: the call graphs resulting from their approach still contain a high percentage of helper and non-core functions. Probability models, however, have been used successfully in ranking [IYF+03, ZJ07] and searching related components [Rob05, SFDB07]. We believe that a similar approach can be used in call graph reduction for the sake of filtering out non-core functions.

The thesis of this work is that **by filtering the call graph to retain only those functions with a high estimated probability of execution, we can arrive at a reduced call graph that is smaller, has a higher percentage of important functions, and which is more robust in the face of changing thresholds of function inclusion, than current approaches.**

To validate the thesis claims, we have conducted a series of case studies and comparisons, showing the improvements over current techniques in terms of the scale of reduction of the call graph (*reduction efficiency*), the importance of functions included in the call graph (*inclusion accuracy*), and the robustness of the approach in the face of changing thresholds of inclusion (*stability*).

In the next section, we give a high-level outline of the field of assisting programmers understand new code bases, and indicate where our approach falls within the field, and what benefits it brings. Then, we provide a high-level illustration of the application of our technique.

1.1 Existing Approaches in Program Understanding

In this section we provide a high-level view of the current work on helping developers understand code bases.

1.1.1 Localized Program Understanding

Approaches such as Program Slicing [Wei79] and Recommender Systems [IYF+03, IYYK05, SFDB07, Rob05] afford the developer the ability to investigate a program from a particular starting point. Program slicing extracts program elements that contribute to, or that are affected by the control or data flow of a selected statement or variable. Recommender systems provide the code of methods as a response to a user query: these methods should act as a guide to further development of a software system. While these systems are helpful for developers who already have a location of interest for investigation in a code base, or who are concerned with a particular software evolution task, they are not helpful for developers who are new to a whole code base, and who wish to familiarize themselves with it in general.

1.1.2 Whole System Analysis

Several fields of research are centered around providing a developer an overview of a system without needing to be provided a starting point. Some are aimed at facilitating system reorganization, and others provide a developer with a structural overview of the system.

Structure Reorganization

Tools related to structural reorganization provide the developer with a way to posit a new modularity for existing code. New modules might be components, as in component mining ([SGMB03, GK97, LL03, TH99, TH00, IYF$^+$03]), aspects, as in aspect mining ([MvDM04, BDET04, BDET05, BZ06, ZJ07], or formed by the developer, as in conceptual modules ([BM98, AL01]. While these tools afford developers an overview of an entire system, and facilitate general system understanding, the main goal of these techniques is not to provide developers with an initial starting point for examination, but to facilitate restructuring.

Structural Overview

Some techniques, such as pattern miners [SKL+02, PIKK98, Rad00]. feature location approaches [MM03, DDL+90, ACC+02, ZZL+06, Egy03, EKS01, EKS03, WS95, WC96] and reverse engineering tools [MK88, Mur96] give structural overviews of a code base. They do not, however, provide a topology of the *importance* of each module or component in the system. Developers would have to use intuition to form a guess as to what portions of the views to investigate to gain an initial understanding of a system. This may result in erroneous investigations, if developer intuition followed that observed in our preliminary study (Chapter 2). In that study, we found that developers rely on naming, to estimate the importance of functions in a system, and also found that this reliance often results in poor choices of target functions. It is likely that the same approach would need to be applied to the output of structural overview tools.

Call Graph Reduction

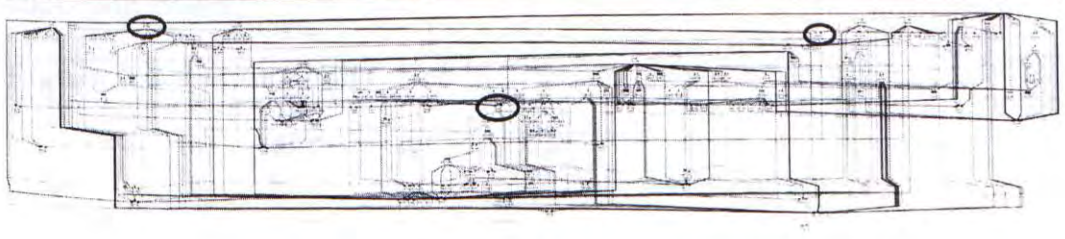
Call graph reduction is an approach that involves trimming down the size of call graph so that developers can focus initial investigation efforts on smaller call graph that contains a higher concentration of functions important to understanding the system. Ying and Tarr have proposed a heuristic technique for call graph reduction [YT07] in which they filter methods out of the call graph by removal of getter and setter methods (the **Don't-hit-bottom heuristic**), and by removing small methods (the **Skip-small-methods heuristic**). Their reasoning is that neither get/set routines nor small methods are likely to contribute to an understanding of the core of the program.

By applying these two filtering heuristics, they found that the reduced call graph is able to help developers identify relevant methods from the source code. However, as we will describe in Chapter 4 , Section 4.3, a high percentage of helper and non-core functions are still left in the reduced call graph.

1.2 Example of Function Execution Probability Reduction of the Call Graph

In this section, we illustrate our call graph reduction technique using `ispell` code base as an example.

Figure 1.1 Call Graph of `ispell`



`ispell` is an interactive spell check program. The code base consists of 168 functions and its overall size is approximately 9 K lines of code. Figure 1.1 shows the original call graph of `ispell`. Although it is a small program, the call structure is complicated, and a starting point for investigation is difficult to identify.

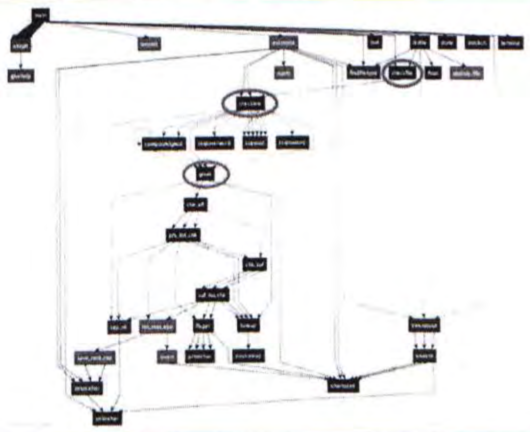
The `ispell` walkthrough guide [ACLS02] provides the developer with a starting point for investigation. These nodes are circled in red. But in the absence of that guide, the developer is left to guess which functions would be of interest. By trimming the call graph down to functions that have a high probability of execution, we can reduce the developer's search space for important functions.

First, we estimate the execution probability of each function based on the structural heuristic of the source code. Figure 1.2 is a visualization of the results: The function nodes are colored according to their estimated probability of execution. The black nodes are 100% likely to be executed and the

Figure 1.2 Estimated Execution Probability of `ispell`

unreachable nodes are colored in white. At this point, the developer can look at dark portions of the call graph to familiarize themselves with the code base. However, the graph is still quite large, so we apply the next step to elide less important nodes.

Figure 1.3 Reduced Call Graph of `ispell`



Next, our call graph reduction approach removes leaf nodes and the function nodes with an execution probability lower than a certain threshold. Figure 1.3 shows the reduced call graph of `ispell` with a threshold of 0.5. The core functions mentioned in the walkthrough guide are retained in the reduced call graph and are, again, circled in red. The developer now has a much reduced search space when looking for functions of central importance to the system.

With the execution probability rating approach we are able to reduce the call graph 20% more than the current state of the art approach[YT07].

1.3 Organization of the Dissertation

The organization of this dissertation is structured as follows: Chapter 2 describes a preliminary studies which motivate our work. Chapter 3 described the details of our approaches. Chapter 4 described the validation of the thesis claims. Chapter 5 discusses the advantages and limitations of using our approaches. Chapter 6 cover other related work. Finally, Chapter 7 concludes.

Chapter 2

Preliminary Study

We conducted a small preliminary study to establish, empirically, that developers have difficulty identifying, in a large call graph, the functions important to examine when forming an initial general understanding of the system. Secondarily, we wished to gain insight into the processes developers might use when exploring such call graphs.

This Chapter is organized as follows: Section 2.1 introduces the participants in this preliminary study. Section 2.2 describes the study design. Section 2.3 and 2.4 report the result of the `ispell` and `freebsd` respectively. Section 2.5 analyzes the threats to validity. Finally, Section 2.6 summarizes.

2.1 Participants

Four subjects participated in the preliminary study. Two of the subjects were postgraduate students in the Chinese University of Hong Kong (PG1 and PG2). The other two subjects were programmers working in industry (I1 and I2).

2.2 Study Design

We chose two systems for use in our study: `ispell` and `FreeBSD` Kernel Malloc. Each of these systems has large call graphs: `ispell`'s call graph has 168 nodes, and `FreeBSD`'s call graph also has 168 nodes. Although the two systems have the same number of nodes, `FreeBSD` has a larger number of lines of code and a larger call structure. For each of these systems, there exists expert documentation to guide a new developer in understanding the system. These documents mention specific functions considered important for understanding. In the case of `ispell`, 3 functions are considered important, and in the case of `FreeBSD`, 10 functions are named as important.

We asked the subjects to examine each call graph, asking them to choose which functions they would like to examine were they to be given the task of understanding the code. We then compared their picks to the functions suggested for investigation in the expert documentation about these systems.

First, we established that the participants had not seen the case study systems before, or examined their call graphs prior to the study.

We then provided the participants with two call graphs, and asked them to identify functions they would investigate were they to be presented with the task of understanding the code base. Subjects were provided with the call graph for each of the systems, but were not allowed to read the source code associated with them.

Finally, for each case, we analyzed the subject's function picks in the following ways:

1. **Number of Exact Matches:** the number of functions that exactly matched with the functions mentioned in the expert documentation

2. **Number of Correct Matches:** the number of functions that differ from the functions mentioned but contribute to the same functionality + the number of functions that are not mentioned in the documentation but actually contribute the major functionality of the program.

   For instance, `ispell` contains prominent secondary functionality: alternative word suggestion for words not found in the dictionary. Though the expert documentation did not include functions related to the core of this secondary functionality, we have identified functions central to this functionality manually, and count those identified by the participant as correct matches

3. **Number of Barely Correct Picks:** the number of initialization functions and the number of functions for core data structures operations. These functions may help developer to understand the system, but are too low level of details

4. **Number of Wrong Picks:** the number of help functions, clean-up functions and other functions that do not directly related to the major functionality of the program

5. **Number of Missed Functions:** the number of functions that are mentioned in the expert documentation, but not picked by the subject

2.3 ispell

`ispell`[1] is an interactive spell checker program for Unix System. In our case studies, version 3.1 is used. The `ispell` package contains 10 executable programs. Only the main program (ispell) is used in our case studies. The code base is consist of 12 modules (.c files) and 5 header files. The overall size is about 9 K lines of code. Figure 1.1 shows the call graph of `ispell`.

2.3.1 Subject I1 (`ispell`)

Subject I1 chose eight functions: `givehelp`, `correct`, `good`, `makepossibilities`, `dumpmode`, `combinecaps`, `expandmode` and `usage`.

Our analysis of the correctness of these picks is summarized in Table 2.1

[1] `ispell` can be downloaded from `http://fmg-www.cs.ucla.edu/geoff/ispell.html`

Table 2.1 Summary of Accuracy: Subject I1, System `ispell`

Number of Functions Picked:	8	
Number of Exact Matches:	1	`good`
Number of Correct Matches:	2	`correct`, `makepossibilities`
Number of Barely Correct Picks:	0	
Number of Wrong Picks:	5	`givehelp`, `dumpmode`, `combinecaps`, `expandmode`, `usage`
Number of Missed Functions:	2	`checkfile`, `checkline`

Exact Matches: The function `good` was picked by the subject. It is a function which check the correctness of a word.

Correct Matches: `correct` and `makepossibilities` are not mentioned in expert documentation, however, based on analysis of their contribution to the functionality of the system, we consider them to be correct picks. These two functions related to a subfeature of `ispell`: trying to make the best guess when a word is not in the dictionary.

Wrong Picks: `combinecaps` and `expandmode` are wrong picks. The former is a helper function, and thus, by our analysis, does not contribute to the major functionality of the system. The latter is related to command line argument parsing, which is not a core function of ispell. The subject also picked the functions which print out the usage: `usage`, `givehelp` and `dumpmode`). These functions are not related to the core functionality of the software and are considered to be wrong picks.

Missed Functions: The functions `checkline` and `checkfile` are not picked by subject I1.

Table 2.2 Summary of Accuracy: Subject PG1, System `ispell`

Number of Functions Picked:	9	
Number of Exact Matches:	2	`checkfile, checkline`
Number of Correct Matches:	4	`main, dofile, askmode, makepossibilities`
Number of Barely Correct Picks:	1	`treeinsert`
Number of Wrong Picks:	2	`mktemp, updatefile`
Number of Missing Functions:	1	`good`

2.3.2 Subject PG1 (`ispell`)

Subject PG1 chose nine functions: `main`, `dofile`, `update_file`, `mktemp`, `checkfile`, `askmode`, `checkline`, `treeinsert` and `makepossibilities`.

Our analysis of the correctness of these picks is summarized in Table 2.2.

Exact Matches: Two of the functions matched with the documentation: `checkline` and `checkfile`.

Correct Matches: Based on our analysis, four functions were classified as correct matches. `dofile` belongs to the same functionality group as `checkfile`. `askmode` is related to the "programming mode", which is a subfeature of `ispell`. `main` is the main function of the program. `makepossibilities` is central to the alternative suggestion function.

Barely Correct: `treeinsert` is responsible for inserting words into an internal tree representation. According to our analysis, it was determined as a barely correct pick because it is an initialization function.

Wrong Picks: `update_file` and `mktemp` are wrong picks. These functions are responsible for the creation of a backup file and temporary file respectively. Therefore, they are not directly related to the main functionality of the system.

Missed Functions: The function `good` was not picked by the subject.

Additional Observation: In this case, it seemed that the naming of functions was misleading to the subject. The subject picked `askmode` assuming that it is related to getting user input, but actually this function is related to the "programming mode". Under this mode, `ispell` interprets the input as command operation. This allows other program integrates `ispell` as part of their system. The subject also said that although he picked `makepossibilities`, he could not guess its usage.

2.3.3 Subject PG2 (`ispell`)

Table 2.3 Summary of Accuracy: Subject PG2, System `ispell`

Number of Functions Picked:	10	
Number of Exact Matches:	2	`checkfile, checkline`
Number of Correct Matches:	5	`dofile, askmode, correct, skiptoword, makepossibilities`
Number of Barely Correct Picks:	2	`treeinsert, treeoutput`
Number of Wrong Picks:	1	`update_file`
Number of Missing Functions:	1	`good`

Subject PG2 chose 10 functions: `dofile, askmode, checkline, correct, skiptoword, checkfile, update_file, treeinsert, treeoutput` and `makepossibilities`

Our analysis of the correctness of these picks is summarized in Table 2.3.

Exact Matches: Two picked functions matched with the documentations: `checkline` and `checkfile`.

Correct Picks: The functions picked by subject PG2 is almost the same as I1. `skiptoword` is a correct pick since it is a function that determine which word can be ignored when parsing a piece of text.

Barely Correct: Functions `treeinsert` and `treeoutput` are barely correct picks as they are operations on data structures.

Wrong Picks: `update_file` is a wrong pick since it is for the creation of backup file and does not contribute to the major functionality of the system.

Missed Functions: The function `good` was not picked by subject PG2.

2.3.4 Subject I2 (`ispell`)

Table 2.4 Summary of Accuracy: Subject I2, System `ispell`

Number of Functions Picked:	6	
Number of Exact Matches:	2	`checkfile, checkline`
Number of Correct Matches:	3	`main, askmode, dofile`
Number of Barely Correct Picks:	0	
Number of Wrong Picks:	1	`expandmode`
Number of Missed Functions:	1	`good`

Subject I2 chose 10 functions: `main`, `askmode`, `expandmode`, `dofile`, `checkfile`, `checkline`.

Our analysis of the correctness of these picks is summarized in Table 2.4.

Exact Matches: `checkfile` and `checkline` were picked by subject I2.

Correct Matched: three functions are correct matches: `main`, `askmode` and `dofile`. *main* is the main function of the program. `askmode` is the interpreter of "programming mode". `dofile` is for handling file input.

Wrong Picks: `expandmode` is a wrong pick since it is a handler for command line parameters.

Missed Functions: The function **good** was not picked by subject I2.

2.3.5 `ispell` Analysis

Table 2.5 Performance of Subjects in Preliminary Study (`ispell`)

Functions in Documentation	Chosen By
`checkfile`	PG1, PG2, I2
`checkline`	PG1, PG2, I2
`good`	I1

Of the three functions noted by the expert, three of the subjects correctly identified the functions **checkfile** and **checkline** (Table: 2.5). Only one subject identified the function **good**. This omission suggested that the name of function "good" does not afford its recognition as a core function.

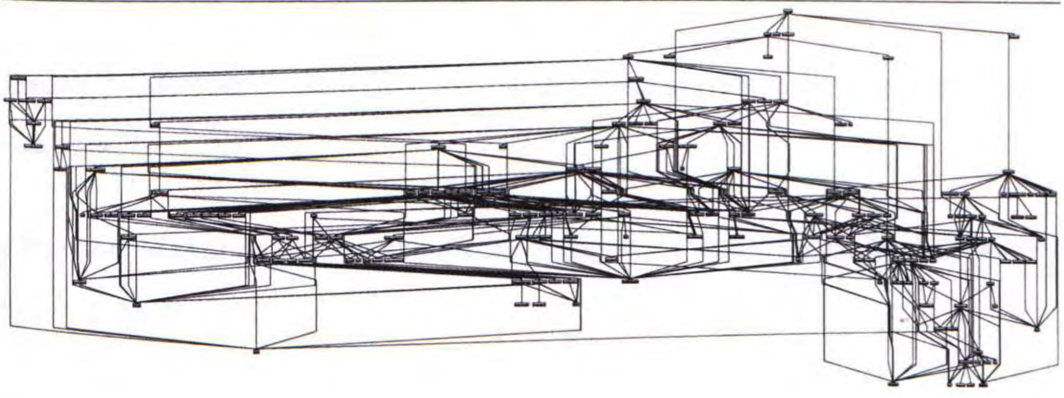
The functions **makepossibilities** and **askmode** were chosen by three of our subjects. Although the two functions are correct picks, our subjects reported that they have difficulty in guessing the meaning of the functions according to the function names.

2.4 FreeBSD Kernel Malloc

FreeBSD[2] is one of the free operating systems in common use. In these case studies, version 6.2 is used. The virtual machine subsystem is responsible for providing a virtual address space for each process and for managing the memory usage of the operating system. The source code of the subsystem is spread over 23 files and contains about 100K lines of code. The virtual memory subsystem is initialized by the `vm_mem_init` function defined in `vm_init.c`. Therefore, the initiation function `vm_mem_init` (instead of the `main` function)

[2]FreeBSD can be obtained and downloaded at `http://www.freebsd.org`

Figure 2.1 Call Graph of `FreeBSD Kernel Malloc`: Bird's eye view



is used in the function rating algorithm. Figure 2.1 shows the call graph of `FreeBSD Kernel Malloc`.

2.4.1 Subject I1 (`FreeBSD`)

Table 2.6 Summary of Accuracy: Subject I1, System `FreeBSD`

Number of Functions Picked:	9	
Number of Exact Matches:	1	`slab_zalloc`
Number of Correct Matches:	2	`vm_object_backing_scan`, `vm_map_lookup_entry`
Number of Barely Correct Picks:	2	`uma_startup`, `vm_page_startup`
Number of Wrong Picks:	4	`mtx_init`, `VFS_LOCK_GIANT`, `msleep`, `vm_paging_needed`
Number of Missing Functions:	6	`uma_zalloc`, `uma_zalloc_arg`, `uma_zalloc_bucket`, `uma_zone_slab`, `uma_zalloc_internal`, `uma_slab_alloc`

Nine functions were chosen by subject I1: `vm_page_startup`, `vm_map_lookup_entry`, `msleep`, `vm_paging_needed`, `VFS_LOCK_GIANT`, `vm_object_backing_scan`, `slab_zalloc`, `uma_startup` and `mtx_init`

Our analysis of the correctness of these picks is summarized in Table 2.6.

Exact Matches: One function matches exactly with the documentation: `slab_malloc`.

Correct Matches: `vm_object_backing_scan` and `vm_map_lookup_entry` are related to the paging mechanism.

Barely Correct: `uma_startup` and `vm_page_startup` are barely correct since they are mainly related to initialization of the kernel zone memory allocator and the paging queue.

Wrong Picks: `mtx_init`, `VFS_LOCK_GIANT`, `msleep` and `vm_paging_needed` are wrong picks since they are helper functions.

Missed Functions: Six functions were not picked by subject I1: `uma_zalloc`, `uma_zalloc_arg`, `uma_zalloc_bucket`, `uma_zone_slab`, `uma_zalloc_internal`, `uma_slab_alloc`.

2.4.2 Subject PG1 (FreeBSD)

Nine functions were chosen by subject PG1: `vm_mem_init`, `vm_page_startup`, `vm_pageq_add_new_page`, `pmap_page_init`, `uma_startup`, `zone_ctor`, `keg_ctor`, `vm_page_io_start`

Our analysis of the correctness of these picks is summarized in Table 2.7.

Exact Matches: None of the chosen functions matched with the documentation.

Correct Matches: `vm_mem_init` is the "main function" of the memory subsystem. It is a correct match according to our analysis.

Table 2.7 Summary of Accuracy: Subject PG1, System FreeBSD

Number of Functions Picked:	8	
Number of Exact Matches:	0	
Number of Correct Matches:	1	`vm_mem_init`
Number of Barely Correct Picks:	6	`vm_page_startup`, `vm_pageq_add_new_page`, `pmap_page_init`, `uma_startup`, `zone_ctor`, `keg_ctor`
Number of Wrong Picks:	1	`vm_page_io_start`
Number of Missed Functions:	7	`slab_zalloc`, `uma_zalloc`, `uma_zalloc_arg`, `uma_zalloc_bucket`, `uma_zone_slab`, `uma_zalloc_internal`, `uma_slab_alloc`

Barely Correct: `vm_page_startup`, `vm_pageq_add_new_page`, `pmap_page_init`, `uma_startup`, `zone_ctor`, `keg_ctor` are related to initialization of the system.

Wrong Picks: `vm_page_io_start` is a wrong pick since it is a helper function.

Missed Functions: Subject PG1 identified none of the documented functions.

2.4.3 Subject PG2 (FreeBSD)

Twelve functions were chosen by subject PG2: `vm_mem_init`, `vm_pager_init`, `vm_page_startup`, `vm_map_startup`, `vm_object_init`, `vm_keymen_init`, `vm_object_deallocate`, `vm_object_init`, `vm_object_reference`, `vm_page_insert`, `vm_page_lookup`, `vm_page_remove` and `vm_page_free`

Our analysis of the correctness of these picks is summarized in Table 2.8.

Table 2.8 Summary of Accuracy: Subject PG2, System FreeBSD

Number of Functions Picked:	12	
Number of Exact Matches:	0	
Number of Correct Matches:	5	vm_mem_init, vm_page_insert, vm_page_lookup, vm_page_remove, vm_page_free
Number of Barely Correct Picks:	6	vm_pager_init, vm_page_startup, vm_map_startup, vm_object_init, vm_keymen_init
Number of Wrong Picks:	2	vm_object_reference, vm_object_deallocate
Number of Missed Functions:	7	slab_zalloc, uma_zalloc, uma_zalloc_arg, uma_zalloc_bucket, uma_zone_slab, uma_zalloc_internal, uma_slab_alloc

Exact Matches: None of the chosen functions matched with the documentations.

Correct Matches: Five functions are correct matches: `vm_mem_init` is the main function, the other four functions (`vm_page_insert`, `vm_page_lookup`, `vm_page_remove`, `vm_page_free`) are related to the paging mechanism.

Barely Correct: `vm_pager_init`, `vm_page_startup`, `vm_map_startup`, `vm_object_init`, `vm_keymen_init`. All are responsible for initialization.

Wrong Picks: `vm_object_reference` and `vm_object_deallocate` are wrong picks since the former is a helper function which increases the reference count of `vm_object` while the latter is another helper function which decreases the reference count.

Missed Functions: All functions mentioned in the documentation were missed.

2.4.4 Subject I2 (FreeBSD)

Subject I2 dropped out of the study before providing any picks for the FreeBSD system's call graph. I2 reported that the call graph for the system is too complicated and hard to follow. He also said that he could not find any hints for getting started.

2.4.5 FreeBSD Analysis

Of the seven functions mentioned in documentation, only one function `slab_zalloc` is picked by subject I1 (Table 2.9). Subject I2 dropped out of the study and reported that the call graph of FreeBSD is too complicated. This suggest that the developer performance is hindered by the complexity of call graph.

Table 2.9 Performance of Subjects in Preliminary Study (FreeBSD)

Functions in Documentation	Chosen By
slab_zalloc	I1
uma_zalloc	
uma_zalloc_arg	
uma_zalloc_bucket	
uma_zone_slab	
uma_zalloc_internal	
uma_slab_alloc	

The subjects tend to choose initialization function in this study. Of the 29 functions chosen by our subjects, 13 of them with name ended with `_init` and `_startup`. This may be related to naming since the naming for initialization functions is more obvious than for the other functions of the system.

2.5 Threats to Validity

Internal Validity The internal validity of our preliminary study is threatened by the bias about the correctness of the functions picked by the participants. To minimize the potential bias, our judgement of exact match is based on the expert documentations. Although functions related to the secondary functionality of `ispell` is not mentioned in the documentation, our identification of secondary functionality is based on the man page of *ispell*. We also subcategories the fuzzy concept of correctness into *Exact Match*, *Correct Match* and *Barely Correct Match*.

External Validity Although only two software systems are used in this preliminary study, they vary in their size and application domain: `ispell` is a small spell checker and `FreeBSD` is a large piece of system source code. We observed that our participants have difficulties in identifying important functions in the call graph of both system. This suggests that the same problem may arise when the call graph of other system is used.

2.6 Summary

In this study we made several observations: Developers base their choices of functions on naming, and on perceived responsibility of the function within the system, and graph size and complexity may have an effect on the developer's ability to identify important functions.

Function Naming We observed that poor function naming can result in incorrect choices. For instance, in the `ispell` study, only one subject chose the core function `good` as a function of interest, probably because of its uncompelling name. In the `FreeBSD` system, many of the core functions were obscurely named, such as the functions which actually implemented the memory allocation policy: `uma_zalloc` and `vm_object_coalesce`. Of the 29 functions chosen by the subjects, only 31% are exact matches or correct functions (33% for I1, 12.5% for PG1 and 41.6% for PG2).

Functionality Triggers The majority of all subjects' choices were initialization functions since they seem to be triggered for the behavior of the entire system. Initialization functions covered an average of 57% of the chosen functions (33.3% for I1, 87.5% for PG1 and 50% for PG2). This may also be related to naming, since the initialization functions in these cases have names that clearly indicate their function.

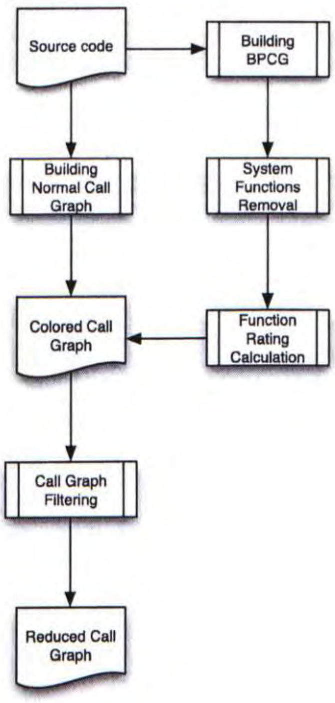
Graph Size/Complexity Comparing the results from the two call graphs, we can see that size and graph complexity may have an effect on the developer's ability to reason about important functions: The number of correct matches in the `FreeBSD` study is lower than the `ispell` study, and one participant dropped out of the study because of the complexity of the larger call graph.

Conclusions Although small, this study provided empirical evidence to suggest that the size and the complexity of a call graph limits the developers' ability to correctly identify important functions.

CHAPTER 3

Approach

Figure 3.1 Function Rating Process



Our approach entails trimming the call graph of functions that fall below a certain threshold of estimated probability of reachability, and in so doing, providing the developer with a call graph that contains a higher percentage of nodes that are important to understanding the system.

To estimate reachability, we use a novel representation called the Branch-*Pre*serving Call Graph (BPCG). This structure is a simplification of the *Branch-Reserving Call Graph* (BRCG) [QZZ+03] introduced by Qin et. al, which captures call-ordering, control flow, and branching information. The BPCG is the same as the BRCG excepts that it ignores call-ordering.

Figure 3.1 depicts the outline of the process. First, the source code is used to build a classical call graph and a BPCG. Next, optionally, system functions are removed.

Next, the execution probability function rating algorithm is applied to the BPCG. We color the classical call graph according to the calculated execution probabilities of the functions for visualization purposes. Finally, we apply our call graph reduction approach to produce a reduced call graph.

We have two approaches for reducing the call graph. One involves removal of functions with high fan-in counts, and the other involves removal of leaf nodes. For the remainder of this thesis, we will refer to the general technique of call graph reduction through trimming based on a threshold of function execution probabilities as FEPR (Function Execution Probability Reduction), to the removal of functions with high fan-in counts as: FEPR-*fanin*, and to removal of leaf nodes technique as FEPR-*leaf*.

In this chapter we will describe each step of the process in detail:

1. The Branch Preserving Call Graphs is formed from the code (Section 3.1)

2. System functions are removed (this is an optional step) (Section 3.2)

3. A function rating calculation algorithm is applied to the BCPG to obtain an execution probability spectrum (Section 3.3) (The complexity of the algorithm will be discussed in Section 3.3.1)

4. The original call graph is colored to reflect the spectrum from the previous step (Section 3.4)

5. The call graph is trimmed based on the function rating algorithm's results, and the chosen threshold for inclusion (this is done using one of two FEPR approaches) (Section 3.5)

3.1 Building Branch-Preserving Call Graphs

A Branch-Preserving Call Graph (BPCG) is a novel simplification of an existing structure, the Branch-Reserving Call Graph (BRCG). In this section we describe the BRCG, and then describe the simplification to form the BPCG.

3.1.1 Branch Reserving Call Graphs

A Branch Reserving Call Graph (BRCG) is an abstract source code representation introduced by Qin et al. They used BRCG's to automatically mine use cases from code. A BRCG is a call graph which represents both function-call statements and branch statements, and which maintains the ordering of the branch statements. In their work, a BRCG is defined as a tuple $< N, S, B >$, where:

1. N is the set of functions, branch statements, and branches in branch statements;
2. S is the set of sequential relationships, where $\forall < n_1, n_2 > \in S, n_1 \in N$ and $n_2 \in N$;
3. B is the set of branching relationships, where $\forall < n_1, n_2 > \in B, n_1 \in N$ and $n_2 \in N$; and
4. $\forall < n_1, n_2 > \in S$ and $\forall n_3 \in N, < n_1, n_3 > \notin B$, and $\forall < n_1, n_2 > \in B$ and $\forall n_3 \in N, < n_1, n_3 > \notin S$.

Figure 3.2 Original Source Code

```
void foo(){
    f1();
    if (condition){
        f2();
        f3();
    }
    else{
        f4();
        f5();
    }
    f6();
}
```

Figure 3.3 Resultant BRCG from the code in figure 3.2

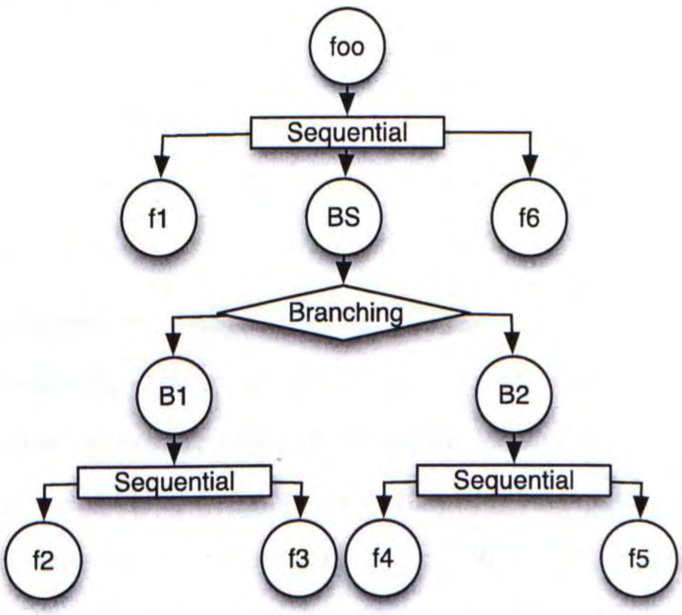


Figure 3.3 shows the BPCG for the C code in Figure 3.2. Each circle node represents a function. The code blocks within the functions are denoted by *sequential nodes.* Conditional statements are represented by the node *BS* and the nodes *B1* and *B2* denote the two branches.

As the control-flow of the program can be captured by BRCG, runtime execution traces can be simulated by traversing the graph. By pruning the unimportant nodes in the BRCG, the collected execution traces become the use cases of the program. This provides evidence that call graphs which contain branching information are useful for software engineering tasks, such as code base understanding.

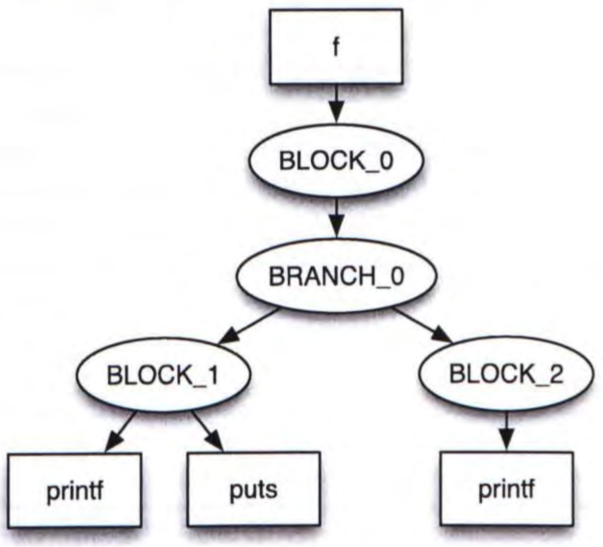
3.1.2 Branch-Preserving Call Graphs

Figure 3.4 Original Source Code

```
void f(int a, int b){
    if(a > b){
        printf("Branch 1\n")
        puts("A > B");
    }
    else{
        printf("B <= A");
    }
}
```

A Branch-Preserving Call Graph (BPCG) is an extension of the classical call graph: It is a call graph with branching information preserved. In a BPCG, a functions or procedures are depicted by a node labeled with their name. Code blocks, such as the blocks contained within procedures, functions, loops and conditional statements are represented by *BLOCK*-nodes. Branching (if and switch) statements themselves are represented as *BRANCH*-nodes. Figure 3.5 shows the BPCG for the C code in Figure 3.4. The function `f` has its own node at the top of the figure. The code block within function `f` is denoted by

Figure 3.5 Resultant BPCG from the code in figure 3.4



the *BLOCK*-Node, `BLOCK_0`. The *if*-statement is depicted by the *BRANCH*-Node, `BRANCH_0`. The body of code in the two branches of the *if*-statement (the if-block, and the else-block) are represented by `BLOCK_1` and `BLOCK_2` respectively. BPCG differs from BRCG as it ignores call-ordering.

Formally, BPCG can be defined as follows:

Definition 3.1 A Branch-Preserving Call Graph is a directed graph $G =< V_\wedge, V_\vee, V_f, E >$ where:

- $V_\wedge$ is a set of *BLOCK*-nodes, which represents composition relationships. In other words, each *BLOCK*-Node represents a code block;
- $V_\vee$ is a set of *BRANCH*-nodes, which represents branching relationships;
- V_f is a set of function node, represents the functions in the source code;
- E is as set of directed edge in which $u, v \in V_\wedge \cup V_\vee \cup V_f$, u invokes $v \implies (u, v) \in E$.

such that G is the union of all the subgraphs created by the *buildBPCG* algorithm depicted in Algorithm 1.

Algorithm 1 buildBPCG(u, s)

```
1: bpcg $g \Leftarrow \emptyset$
2: for all function $f$ in the source code do
3:     subgraph $h \Leftarrow \emptyset$
4:     $fn \Leftarrow createFunctionNode(f)$ and insert into $h$
5:     $blockNode \Leftarrow v_\wedge$ and insert into $h$, $v_\wedge \in V_\wedge$
6:     create a directed edge $(fn, blockNode) \in E$ and insert into $h$
7:     $createSubgraph(blockNode, codeBlockOf(fn)$ and insert into $h$
8:     $g$ union with $h$
9: end for
10: return $g$
```

Algorithm 2 createSubgraph(u, s)

```
1: subgraph $g \Leftarrow u$
2: for all $s \in$ { Function call statements, branch statements, and looping
   constructs in the code block s} do
3:     if $isCallStatement(s)$ then
4:         create a directed edge $(u, v) \in E$ to the node of the function called
           $v, v \in V_f$ and insert into $g$
5:     else if $isLoopingConstruct(s)$ then
6:         $blockNode \Leftarrow v_\wedge$ and insert into $h$, $v_\wedge \in V_\wedge$
7:         create a directed edge $(u, blockNode)) \in E$ and insert into $g$
8:         $s_{loop} \Leftarrow$ code block of the loop
9:         $g_{loop} \Leftarrow createSubGraph(u, s_{loop})$ and insert into $g$
10:        create a directed edge $(blockNode, rootNodeOf(g_{loop})) \in E$ and insert
           into $g$
11:    else
12:        $branchNode \Leftarrow v_\vee$ and inset into $g$, $v_\vee \in V_\vee$
13:        create a directed edge $(u, branchNode) \in E$ and insert into $g$
14:        for all branches $b$ in the branch statement do
15:            $blockNode \Leftarrow v_\wedge$ and insert into $h$, $v_\wedge \in V_\wedge$
16:            $s_b \Leftarrow codeBlockOf(b)$
17:            $g_b \Leftarrow createSubGraph(branchNode, s_b)$ and insert into $g$
18:            create a directed edge $(branchNode, rootNodeOf(g_b)) \in E$ and in-
               sert into $g$
19:        end for
20:    end if
21: end for
22: return $g$
```

A Branch-Preserving Call Graph builder is implemented with the help of ANTLR parser generator[1]. In this work, *if*-statements and *case*-statements are treated as branch statements. Recursive function calls are not included in the BPCG as the rating calculation step requires the BPCG to be acyclic. The source code of our Branch-Preserving Call Graph builder is included in Appendix B.

Unlike [QZZ$^+$03], looping constructs are not considered as branching statements because looping constructs rarely perform conditional branching. Instead, *BLOCK*-nodes are used to represent looping constructs and treat them as code blocks.

3.1.3 Example of BPCG Building Process

Figure 3.6 Original Source Code

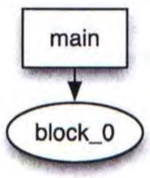
```
void odd(){
    puts("Odd Number");
}

void main(){
    int i;
    for(i=0; i<100; i++){
        if(i%2 == 0){
            printf("Hello World\n")
            puts("Even Number");
        }
        else{
            odd();
        }
    }
}
```

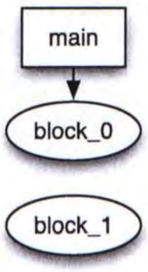
This section shows an example of how the BPCG of source code (Figure 3.6) is built using the BPCG building algorithm (Algorithm 1).

[1]ANTLR can be obtained at http://www.antlr.org/

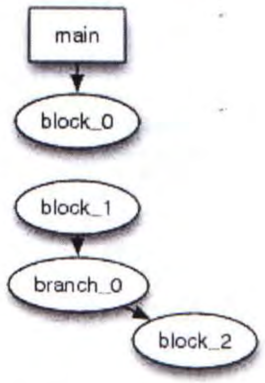
Figure 3.7 Building BPCG from the code in Figure 3.6



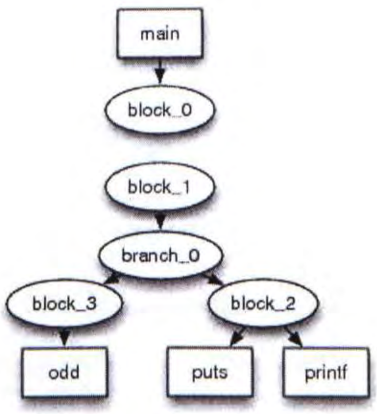
(a) Creating the Function node for function `main`



(b) Loop is denoted by a *block*-node

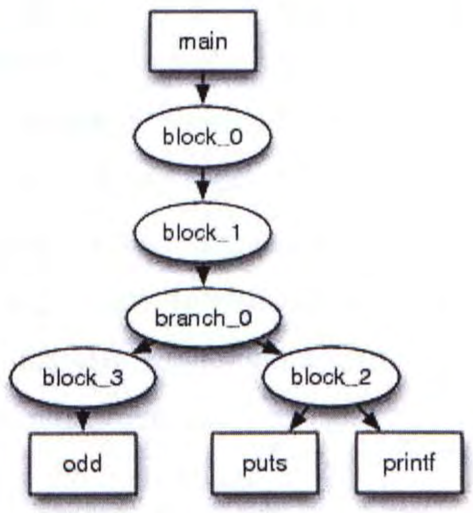


(c) Creating a *branch*-node for the *if*-statement

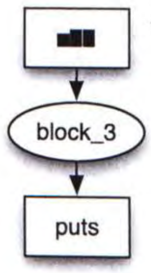


(d) The BPCG Subgraph for the *for*-loop block is created

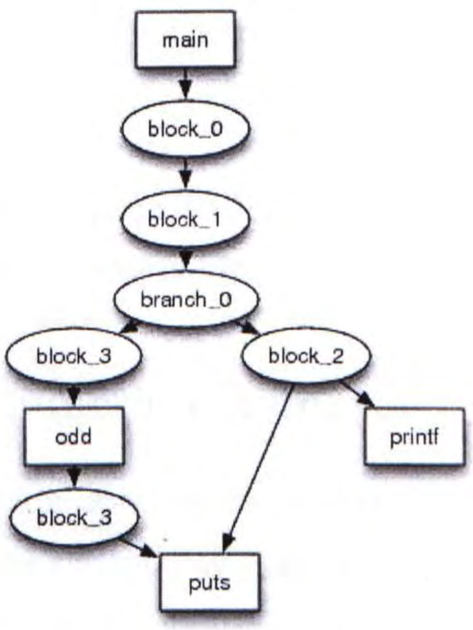
Figure 3.8 Building BPCG from the code in Figure 3.6 (conc.)



(a) BPCG Subgraph for function `main`



(b) BPCG Subgraph for function `odd`



(c) BPCG for the code in Figure 3.6

There are two functions defined in the source code shown in Figure 3.6 , `main` and `odd`. A BPCG subgraph is created for each function and the resulting BPCG is the union of these subgraphs.

Figures 3.7 and 3.8 show the process of building a BPCG subgraph for the `main` function. Initially, a function node is created and attached with a *block*-node (`block_0`) which represents the code block of the `main` function (Figure 3.7(a)). Then, another *block*-node (`block_1`) is created representing the *for*-loop inside the code block of function `main` (Figure 3.7(b)).

Inside the *for*-loop, there is an *if*-statement. The *createSubgraph* algorithm is called recursively. As shown in Figure 3.7(c), the *if*-statement is represented by a *branch*-node (`branch_0`) and the first branch of the *if*-statement is represented by a *block*-node (`block_0`).

The function nodes `printf` and `puts` are created and attached to the *block*-node of the first branch. Similarly, `odd` node is created and attached to the *block*-node of the second branch. Figure 3.7(d) shows the resulting subgraph of the whole *for*-loop block. The subgraph is then connected by an directed edge linking `block_0` and `brock_1`. Figure 3.8(a) shows the BPCG subgraph for function `main`.

Similarly, the BPCG subgraph for function `odd` is shown in Figure 3.8(b). The resulting BPCG for the code in Figure 3.6 is the union of the 2 subgraphs.

3.2 System Function Removal

The second step of the process is to remove function nodes representing the system functions before applying the function rating algorithm. Functions such as `printf()` do not contain interesting information in most applications. However, this kind of function is usually called frequently and thus has a misleadingly high probability in our function rating process. Although these functions are likely to be called in our call graph reduction phase, their inclusion at this phase causes a performance degradation, so we remove them here preemptively. This step is optional since system function nodes can be crucial in some application domains: Operating systems and utilities may rely heavily on these system functions, and they may be core to their functionality.

In our implementation, functions provided in the standard C Library and the POSIX API can be filtered as needed.

3.3 Function Rating Calculation

The function rating algorithm is based on the assumption that all branches have the same probability of being called.[2] With this assumption, it is possible to estimate the probability of execution of each function using BPCG.

It is comparatively easier to compute the probability of not reaching a particular function f from the main function than computing the reachability from main to f directly. Algorithm 3 is a recursive approach to estimating the probability of not reaching a particular function, where u is the starting node, v is the desired function and g is the Branch-Preserving Call Graph of the program.

branch-Nodes which have an out-degree of one are probably a single *if*-statement without an *ELSE* clause. Therefore, their probability is reduced by half.

Then, $1 - unreachableProbability(main, f, g)$ is the probability of execution of function f.

This approach deals with function reachability but not frequency of execution. Unlike the techniques in [WMGH94] and [WL94], estimation of loop-trip

[2]While it is true that this assumption is not a perfect one, it is reasonable for a static approach, since the actual probability profile of execution is impossible to predict without running the program. Further discussion is provided in Chapter 6.2

Algorithm 3 unreachableProbability(u, v, g)

```
Require: $g$ is acyclic
if $u == v$ then
  return 0
end if
if unReachable(u, v, g) then
  return 1
end if
if $u$ is a leaf node then
  return 1
else if $u$ is a block-Node then
  $\bar{p} \Leftarrow 1$
  for all child in $childrenOf(u, g)$ do
    $\bar{p} \Leftarrow \bar{p} \times unreachableProbability(child, v, g)$
  end for
  return $\bar{p}$
else if $u$ is an branch-Node then
  if $u$ has only one child then
    return $unreachableProbability(child, v, g)/2$
  else
    $\bar{p} \Leftarrow 0$
    $deg \Leftarrow degreeOf(u, g)$
    for all child in $childrenOf(u, g)$ do
      $\bar{p} \Leftarrow \bar{p} + unreachableProbability(child, v, g)/deg$
    end for
    return $\bar{p}$
  end if
else
  $child \Leftarrow childrenOf(u, g)$
  return $unreachableProbability(child, v, g)$
end if
```

Figure 3.9 Sample C Code with Looping Construct

```
int main(){
    int a, b;
    scanf("%d %d", &a, &b);
    if ( a > b ){
        int i;
        for ( i = 0 ; i < 3 ; ++i){
            f();
        }
    }
    else{
        g();
    }
    return EXIT_SUCCESS;
}
```

Figure 3.10 BPCG for the code in figure 3.9

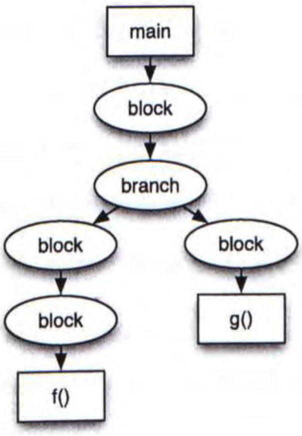
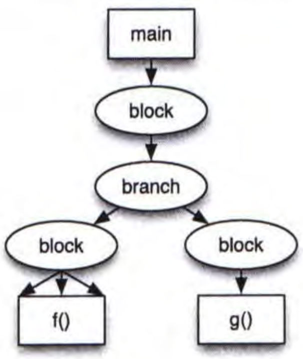


Figure 3.11 An Equivalent BPCG for figure 3.9



count is not necessary using this approach. This feature is illustrated in Figures 3.9, 3.10 and 3.11.

Figure 3.10 shows the Branch-Preserving Call Graph for the code in Figure 3.9. The *if*-statement is represented by an *branch*-node and the *for*-loop is represented by an *block*-node. If we rewrite the *for*-loop by calling the function `f` 3 times, we can obtain an equivalent Branch-Preserving Call Graph to that shown in Figure 3.11. It is obvious that the probability of reaching function `f` remains unchanged no matter how many iterations have been made.

3.3.1 Rating Algorithm Complexity

The complexity of Algorithm 3 depends on the implementation of the subroutine $unReachable(u, v, g)$. This subroutine determines whether function v is unreachable from function u in the BPCG g. A trivial implementation for $unReachable(u, v, g)$ is to use Depth First Search, for which the complexity is $O(V + E)$. The overall complexity becomes $O((V + E)^2)$ for each function to be analyzed.

Pre-calculating the reachability between all vertices inside a BPCG is a reasonable way to reduce the complexity. The complexity of the subroutine $unReachable(u, v, g)$ then can be reduced to $O(1)$ while the extra constant cost of pre-calculation is only $O(V(V + E))$. The overall complexity becomes

$O(V(V+E))$ for each function to be analyzed.

3.4 Building the Colored Call Graph

The output of the function rating algorithm is the execution probability for each function. This output forms a spectrum of probability of function execution likelihoods. For example, for the code in Figure 3.9, the execution likelihoods for this code would be: { *f:0.5, g:0.5* }.

With the execution probabilities calculated, we can create a colored call graph in which each function node is colored to reflect the probability of execution. Unreachable nodes are colored white, while black nodes are 100% likely to be executed. The colored call graph is a visualization of the probability spectrum so that the result can be easily visually interpreted.

3.5 Call Graph Reduction

Once the execution probabilities are calculated, we can make use of this result to remove the nodes in the call graph which have an execution probability beneath a certain threshold. In this way, we trim down the call graph such that only the core functions are left. In this section, we introduce two approaches for call graph reduction: One involves the automatic removal of function nodes with high fan-in counts, and the other involves automatic removal of leaf nodes.

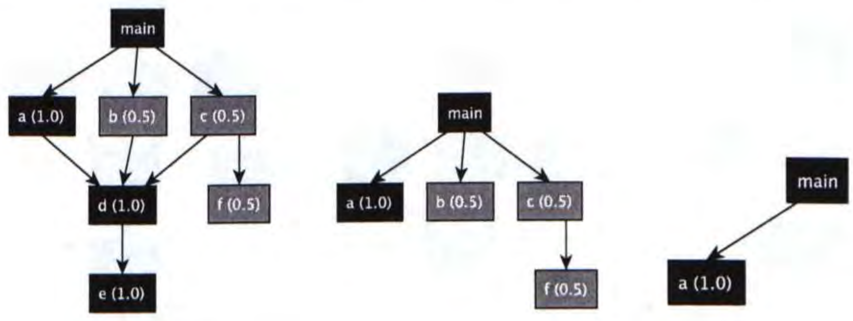
3.5.1 Remove-high-fan-in-functions Approach (FEPR-*fanin*)

Marin et al. have proposed an aspect-mining method [MvDM04] based on identifying methods which have a high fan-in on the call graph. They have observed that aspects, which can be seen to contain functionality superimposed on the core functionality of the system, [RB03], are usually implemented by methods with high fan-in values [MvDM04]. Moreover, according to their findings, a large number of methods with high fan-in values are usually get-setters or utility methods. Since we are only interested in the core functionality, it is reasonable to use this approach to filter out aspects and utility functions. Thus, we remove the functions with high fan-in values from the call graph. Our *Remove-high-fan-in-functions* process is described as follow:

1. Remove functions which are called by more than 2 functions and have fan-in value greater than threshold ***Fan-in Max***

2. Remove functions which have execution probability less than threshold ***Threshold***

Figure 3.12 Remove-high-fan-in-functions Approach (FEPR-*fanin*)



(a) Original Colored Call Graph (b) Remove High Fan-in Functions (c) Remove Functions with Low Execution Probability

Figure 3.12 shows our *Remove-high-fan-in-functions*(FEPR-*fanin*) process with **Fan-in Max** = 2 and **Threshold** = 0.6. Function d is removed since it has fan-in greater than **Fan-in Max**. Function e is also removed since it is only connected to function d which is already removed. Function b, c and f are removed since their execution probabilities are lower then **Threshold**.

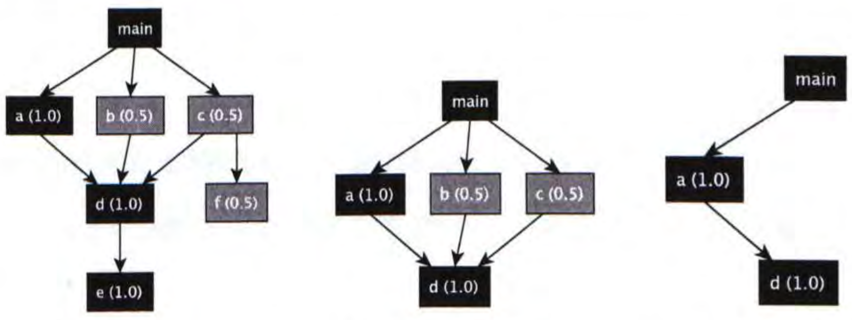
3.5.2 Remove-leaf-nodes Approach (FEPR-*leaf*)

Through our own examination of many call graphs, we have determined that leaf nodes of a call graph are usually utility functions that are not relevant to the task of program understanding. Therefore, removing the leaf nodes is a reasonable heuristic to trim down a call graph. One advantage over the previous approach is that the call tree structure is still maintained under this approach.

Our *Remove-leaf-nodes Approach* process is described as follow:

1. Remove the leaf nodes from the call graph
2. Remove functions which have execution probability less than threshold ***Threshold***

Figure 3.13 Remove-leaf-nodes Approach (FEPR-*leaf*)



(a) Original Colored Call Graph (b) Remove Leaf Nodes (c) Remove Functions with Low Execution Probability

Figure 3.13 shows our *Remove-leaf-nodes*(FEPR-*leaf*) process with **Threshold** = 0.6. Leaf nodes `e` and `f` are removed as shown in Figure 3.13(b). Function `b`, `c` are removed since their execution probabilities are lower then **Threshold**.

CHAPTER 4

Validation

The thesis of this work is: By trimming the call graph to remove functions below a certain function execution probability threshold, we can reduce the size of the call graph, while retaining a higher percentage of the functions important to an understanding of the code base than current approaches.

This thesis can be decomposed into the following claims:

1. **Inclusion Accuracy:** Trimming based on function execution probability chooses more important functions (*precision*), and misses fewer important functions (*miss rate*) than current approaches for call graph reduction.

2. **Reduction Efficiency:** Trimming based on function execution probability will allow a higher degree of reduction, resulting in smaller call graphs.

3. **Stability:** The Inclusion Accuracy and Reduction Efficiency for a graph's reduction will remain stable under different trimming-thresholds of function execution probability.

For the remainder of this chapter, we will refer to the technique of call graph reduction through trimming based on a threshold of function execution probabilities as FEPR (Function Execution Probability Reduction). As described in Chapter 3, we have developed two versions of FEPR : removal of functions with high fan-in counts, which we will refer to as: FEPR-*fanin*, and removal of leaf nodes, which we will refer to as: FEPR-*leaf*.

To validate these claims, we revisit the `ispell` and `FreeBSD` case studies from our preliminary analysis, and compare the performance of FEPR against other techniques:

- We begin by presenting the results for Inclusion Accuracy, Reduction Efficiency and Stability for the two FEPR techniques.
- We then apply Ying and Tarr's approach to the same case studies, and analyze their graphs in terms of Inclusion Accuracy, Reduction Efficiency and Stability.
- Next, we use a centrality measure to identify important functions in the call graph, and compare this measure against FEPR in terms of Inclusion Accuracy.
- We then compare functions chosen using breadth first search of the original call graph with the FEPR techniques in terms of Reduction Efficiency.
- Finally, we discuss the results for each of the measures from all studies.

4.1 Measures

In this section we describe how we assess each measure in the two case study systems.

4.1.1 Inclusion Accuracy (IA)

We evaluate the accuracy of the call graph reduction techniques by comparing the functions appearing in the FEPR generated call graph to the functions picked out by domain experts. A function found in the reduced call graph that is also in the expert documentation is counted as a "match", and a function listed in the expert documentation but not found in the reduced call graph is counted as a "miss".

The expert documentation used in the `ispell` case is the walkthrough guide: [ACLS02]. They have reported their process of software understanding and implementation of morphological analyzer for Italian language based on `ispell`. In that guide, three functions are mentioned as important for gaining an initial understanding of the codebase. The expert documentation used in the `FreeBSD` case is the walkthrough guide article by Lee [李08] on the kernel malloc mechanism. 10 functions are listed in that guide. The functions for each of these guides are shown in Table 4.1.

Table 4.1 Functions in Documentations

Code Base	**Functions**
`ispell`	*checkfile*, *checkline*, *good*
`FreeBSD Kernel Malloc`	*uma_zalloc*, *uma_zalloc_arg*, *uma_zalloc_bucket*, *uma_zone_slab*, *slab_zalloc*, *uma_zalloc_internal*, *uma_slab_alloc*, *uma_large_malloc*, *page_alloc*, *kmem_malloc*

Inclusion accuracy is measured by *Precision* and *Miss Rate*. The size of the call graph is the number of function nodes on the call graph.

1. **Precision:** $\frac{noOfMatch}{sizeOfReducedCallGraph}$

2. **Miss Rate**[1]**:** $\frac{noOfMiss}{noOfFunctionsMentionedInDocumentation}$

4.1.2 Reduction Efficiency (RE)

We evaluate the reduction efficiency by comparing the size of reduced call graph with the original call graph. It is measured by the percentage reduction of the call graph.

1. **Percentage Reduction:** The percentage reduction in the size of call graph. That is: $\frac{sizeOfNormalCallGraph - sizeOfReducedCallGraph}{sizeOfNormalCallGraph}$

[1]Miss Rate is also equals to $1 - recall$.

4.1.3 Stability (S)

Our rating approaches for call graph reduction required developer to specify the threshold for inclusion as a parameter and the maximum fan-in value of a function. It is important to see how the parameter values impact the result of our approaches. For a stable approach, the result should not change significantly when the parameter values only changed a little bit. This property stability is especially important when the developer wants to fine-tune their analysis.

To assess the stability of our approaches, we analyze the size of the reduced call graph under different parameter settings: The *Threshold* and *Fan-in Max* of FEPR-*fanin* approach, and the *Threshold* of FEPR-*leaf* approach.

4.2 Analysis of FEPR Techniques

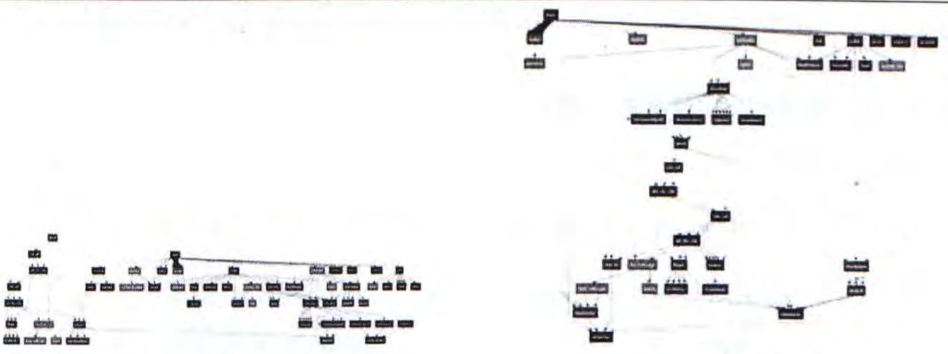
As mentioned in Section 3.5, we have two call graph reduction approaches based on function rating. The *remove-high-fan-in-functions* approach (FEPR-*fanin*) removes the high fan-in functions first and then trims down the call graph by removing functions under a specified threshold of execution probability. The *remove-leaf-nodes* approach (FEPR-*leaf*) removes the leave nodes first and then trims down the call graph by removing functions under a specified threshold of execution probability. In this section we first provide the settings used for the application of FEPR on the `ispell` and `FreeBSD` systems. We then report on the FEPR results for Inclusion Accuracy, Reduction Efficiency and Stability.

4.2.1 Settings

The following settings are used in our validation. For each approach, the parameter values that lead to the best performance are chosen.

ispell

Figure 4.1 Call Graphs Obtained by Our Call Graph Reduction Approaches (`ispell`) (Bird's eye views)



(a) FEPR-*fanin* (b) FEPR-*leaf*

1. **FEPR-*fanin*:** Reduced Call Graph by *remove-high-fan-in-functions* approach (Threshold=0.5, Fan-in Max=4). As shown in Figure 4.1(a).
2. **FEPR-*leaf*:** Reduced Call Graph by *remove-leaf-nodes* approach (Threshold=0.5). As shown in Figure 4.1(b).

Larger pictures of the call graphs are provided in Appendix A.

FreeBSD Kernel Malloc

Figure 4.2 Call Graphs Obtained by Our Call Graph Reduction Approaches (`FreeBSD`) (Bird's eye views)

(a) FEPR-*fanin* (b) FEPR-*leaf*

1. **FEPR-*fanin*:** Reduced Call Graph by *remove-high-fan-in-functions* approach (Threshold=0.4, Fan-in Max=4). As shown in Figure 4.2(a).

2. **FEPR-*leaf*:** Reduced Call Graph by *remove-leaf-nodes* approach (Threshold=0.4). As shown in Figure 4.2(b).

4.2.2 Inclusion Accuracy (IA):

The results of `ispell` study and `Freebsd` study are shown in Tables 4.2 and 4.3. The precision of FEPR-*leaf* approach is higher than FEPR-*fanin* approach. The miss rate of FEPR-*fanin* is slightly lower than FEPR-*leaf*.

Table 4.2 Inclusion Accuracy of FEPR approaches (`ispell`)

	FEPR-*fanin*	**FEPR-*leaf***
Size:	55	38
Precision:	3/55 = 0.05	3/35 = 0.09
Miss Rate:	0	0

Table 4.3 Inclusion Accuracy of FEPR approaches (`FreeBSD`)

	FEPR-*fanin*	**FEPR-*leaf***
Size:	55	35
Precision:	7/55 = 0.08	6/35 = 0.17
Miss Rate:	3/10 = 0.3	4/10 = 0.4

4.2.3 Reduction Efficiency (RE):

Table 4.4 shows the result of the `ispell` study using the rating approaches. In both approaches, more than 50% of the functions are removed from the call graph. Our *remove-leaf-nodes* (FEPR-*fanin*) approach reduced the call graph by 77.4%. The reduction by *remove-high-fan-in-functions* (FEPR-*leaf*) is 67.3%.

The result of `FreeBSD` case study is shown in Table 4.5. The percentage reduction of *remove-leaf-nodes* (FEPR-*fanin*) approach and *remove-high-fan-in-function* (FEPR-*leaf*) approach are 67.3% and 79.2% respectively.

Table 4.4 Reduction Efficiency of FEPR approaches (`ispell`)

	FEPR-*fanin*	**FEPR-*leaf***
Size of Call Graph:	55	38
Percentage Reduced:	(168-55)/168 = 67.3%	(168-38)/168 = 77.4 %

Table 4.5 Reduction Efficiency of FEPR approaches (`FreeBSD`)

	FEPR-*fanin*	**FEPR-*leaf***
Size of Call Graph:	55	35
Percentage Reduced:	(168-55)/168 = 67.3%	(168-35)/168 = 79.2

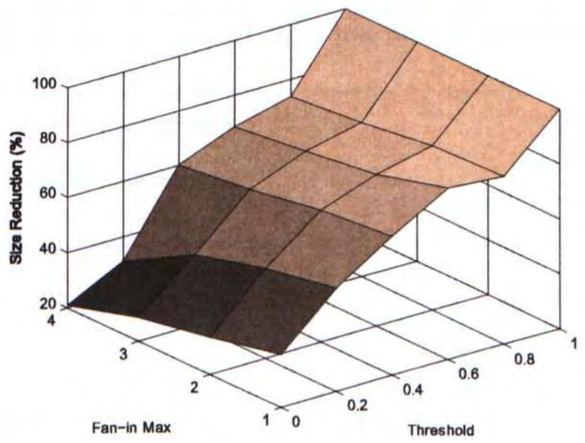
4.2.4 Stability (S)

Figure 4.3 shows the percentage reduction of the `ispell` call graph under different parameter settings using different approaches. The result of the FEPR-*fanin* approach is shown in Figure 4.3(a). As shown in the figure, there is no sharp change in percentage reduction when we change the parameter value of *Fan-in Max* and *Threshold* of FEPR-*fanin* approach. Figure 4.3(b) shows the change in percentage reduction when tuning the value of *Threshold* in FEPR-*leaf* approach. In both FEPR approaches, the percentage reduction on the call graph does not change much when there is some minor modifications to the parameters.

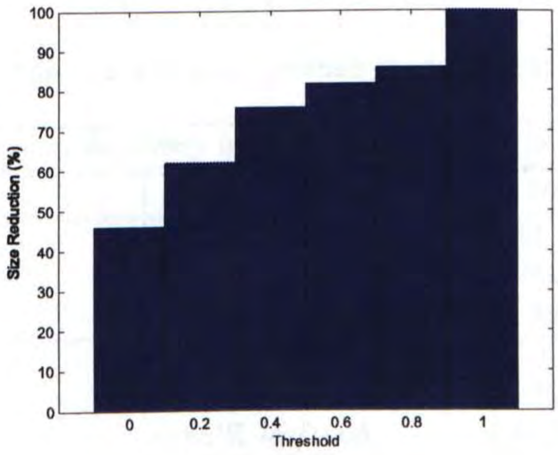
4.3 Ying and Tarr's Approach

As described in Chapter 1, Ying and Tarr produce a reduced call graph to isolate important nodes by removing small methods, and by removing leaf nodes. Here we compare the success of their technique to our own in terms of Inclusion Accuracy, Reduction Efficiency and Stability.

Figure 4.3 Percentage Reduction of FEPR approaches under Different Parameter Settings



(a) FEPR-*fanin*



(b) FEPR-*leaf*

4.3.1 Settings

1. **ispell:** Reduced Call Graph by Ying and Tarr's approach ($p_{bottom} = 1$, $p_{small} = 2$). As shown in Figure 4.4(a).
2. **FreeBSD:** Reduced Call Graph by Ying and Tarr's approach ($p_{bottom} = 1$, $p_{small} = 2$). As shown in Figure 4.4(b).

Figure 4.4 Call Graph Obtained by Ying and Tarr's Function Filtering Approaches

(a) `ispell`

(b) `FreeBSD`

Larger pictures of the call graphs are provided in Appendix A.

4.3.2 Inclusion Accuracy (IA)

The results of `ispell` study and `Freebsd` study are shown in Table 4.6.

Table 4.6 Inclusion Accuracy of Ying and Tarr's Approach

	`ispell`	`FreeBSD`
Size:	76 FEPR-*fanin*:+21 FEPR-*leaf*:+38	60 FEPR-*fanin*:+5 FEPR-*leaf*:+25
Precision:	3/67=0.04 FEPR-*fanin*:-.01 FEPR-*leaf*:-.05	5/60=0.083 FEPR-*fanin*:+.003 FEPR-*leaf*:-.083
Miss Rate:	0 FEPR-*fanin*:0 FEPR-*leaf*:0	5/10=0.5 FEPR-*fanin*:+.2 FEPR-*leaf*:+.1

The FEPR approaches outperform Ying and Tarr's function filter approach in both cases. The precision of our FEPR-*fanin* approach yields a similar result in precision to Ying and Tarr's approach but with a lower miss rate. The FEPR-*leaf* approach outperforms the other approaches in both precision and miss rate.

4.3.3 Reduction Efficiency (RE)

The results of `ispell` study and `FreeBSD` study are shown in Table 4.7.

Table 4.7 Reduction Efficiency of Ying and Tarr's Approach

	`ispell`	`FreeBSD`
Size of Call Graph:	76 FEPR-*fanin*:+21 FEPR-*leaf*:+38	60 FEPR-*fanin*:+5 FEPR-*leaf*:+25
Percentage Reduced:	(168-76)/168 = 54.8% FEPR-*fanin*:-12.5% FEPR-*leaf*:-22.6%	(168-60)/168 = 64.3% FEPR-*fanin*:-3% FEPR-*leaf*:-14.9%

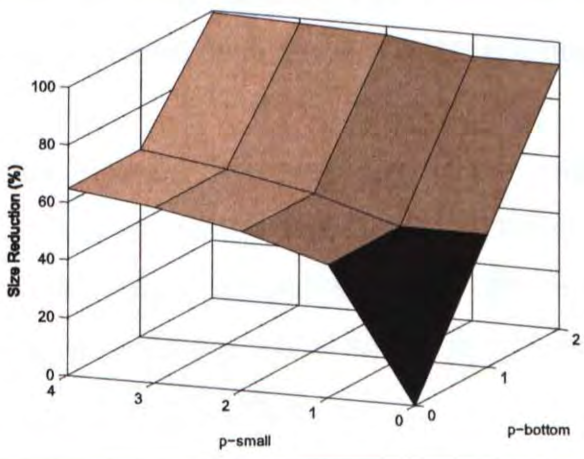
Table 4.7 shows that the performance in terms of reduction efficiency for the FEPR approaches is better than the performance for the Ying and Tarr approach. The FEPR-*leaf* approach performed best, bettering Ying and Tarr's approach in terms of overall graph size by 38 and 25 for `ispell` and `FreeBSD` respectively, and by 22.6% and 14.9% for `ispell` and `FreeBSD` respectively for percentage reduction.

4.3.4 Stability (S)

Figure 4.5 shows the percentage reduction on the call graph of `ispell` of Ying and Tarr's function filtering approach. Only the `ispell` system was used for Stability comparison, because the reduction of the call graph for the `FreeBSD` system resulted in such a proliferation of reduced call graphs that comparison was intractable.

As shown in Figure 4.3, our FEPR approaches are more stable than Ying and Tarr's approach (Figure 4.5). When the p_{bottom} is changed from 1 to 2,

Figure 4.5 Percentage Reduction under Different Parameter Settings



the percentage reduction reduces 40% more.

The performance of Ying and Tarr's approach was likely hurt by the shallow call depth of `ispell` call graph. When the parameter p_{bottom} is set to 2, almost all functions in the call graph are removed. Therefore, the p_{bottom} parameter can only be used as a coarse-tuning parameter. The p_{small} parameter is intended to filter out functions with small number of callees such as delegation function. Altering the p_{small} parameter does not have much effect on the degree to which the call graph is reduced. As shown in Figure 4.5, the percentage reduction only increased a little bit when the p_{small} changed from 1 to 4.

As a result of the limitations of these two parameters, it is hard to fine tune the result in this case. In contrast, the results of our approaches can be fine-tuned by adjusting the execution probability threshold. Figure 4.3 shows that the percentage reduction can be changed by more than 50% when the execution probability threshold is adjusted from 0.0 to 1.0.

4.4 Centrality Measure Approach

When faced with the task of using a call graph to gain an understanding of a system, one approach a developer might employ is to spot functions that seem central to the functionality of the system – particularly, functions that are called by, or which call, a high number of other functions. Here, we compare a call graph composed of such functions to a call graph produced by the FEPR technique.

The Centrality Measure determines the relative importance of a vertex within a graph. There are various measures of centrality in network analysis [Fre79]. In our study, *degree centrality* is used. Mathematically, the *degree centrality* of a vertex v in a graph G is defined as follows:

$$centrality(v) = \frac{degree(v)}{maxDegreeOfVerticesIn(G)}$$

We compare the quality of the functions with the highest centrality value, including the role of functions picked and the accuracy of using this approach.

4.4.1 Inclusion Accuracy (IA)

Table 4.8 Functions with the Highest Centrality

	Function(*Centrality Value*)
ispell	`main`(1.00),`printf`(0.73), `correct`(0.58), `fprintf`(0.55),`strcpy`(0.30), `checkline`(0.27), `TeX_strncmp(0.23)`,`usage`(0.23), `treeinit`(0.22), `linit`(0.22),`strlen`(0.21)
FreeBSD	`VM_OBJECT_UNLOCK`(1.00), `VM_OBJECT_LOCK_ASSERT`(0.97), `vm_object_deallocate`(0.94), `vm_object_backing_scan`(0.85), `vm_object_page_clean`(0.65), `vm_object_collapse`(0.62), `vm_object_page_collect_flush`(0.56), `ZONE_UNLOCK`(0.53), `VM_OBJECT_LOCK`(0.53), `vm_object_terminate`(0.50), `uma_zalloc_arg`(0.50)

Table 4.8 shows the functions with highest centrality value in the case

studies. We classify the functions by their role as shown in Table 4.9. The distribution of centrality values is shown in Figure 4.6.

Table 4.9 Classification of High Degree Centrality Functions

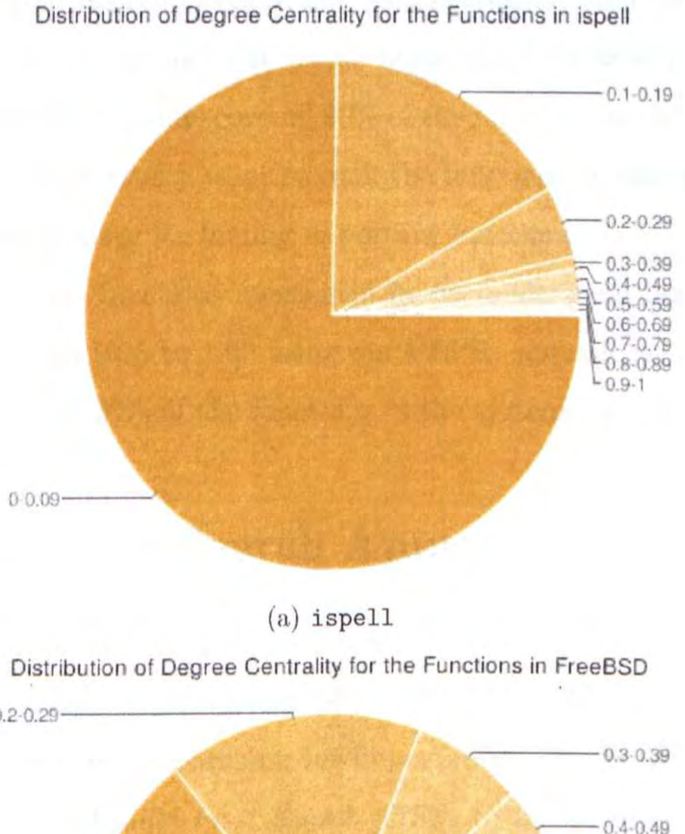
Role	**ispell**	**FreeBSD**
Helper	*printf*, *fprintf*, *strcpy*, *TeX_strncmp*, *strlen*	*VM_OBJECT_UNLOCK*, *VM_OBJECT_LOCK_ASSERT*, *ZONE_UNLOCK*, *VM_OBJECT_LOCK*
Initialization	*treeinit*, *linit*	
Clean up		*vm_object_deallocate*, *vm_object_page_clean*, *vm_object_page_collect_flush*, *vm_object_terminate*
Functional	*main*, *usage*, *correct*, *checkline*	*vm_object_backing_scan*, *vm_object_collapse*, *uma_zalloc_arg*

Table 4.10 Degree Centrality for the Functions Mentioned in Documentations

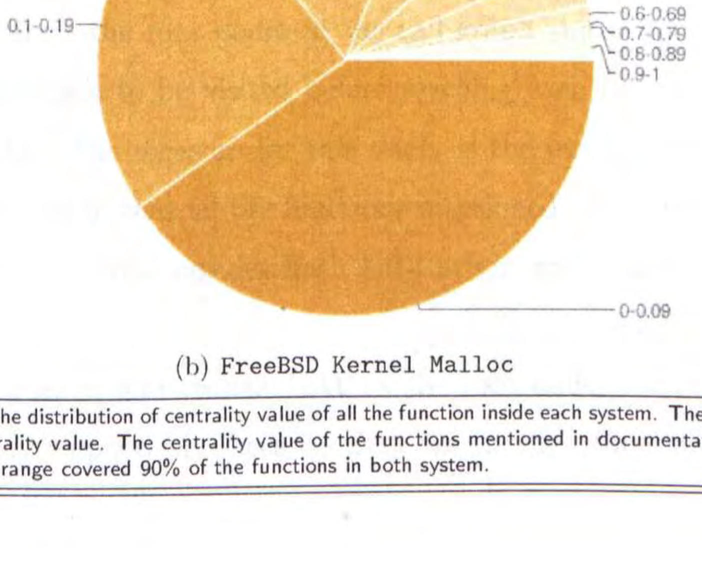
Code Base	**Functions**	**Degree Centrality**	**Execution Probability**
`ispell`	*checkline*	0.27	1
	good	0.12	0.95
	checkfile	0.05	1
`FreeBSD Kernel Malloc`	*uma_zalloc*	0.15	1
	uma_zalloc_arg	0.5	1
	uma_zalloc_bucket	0.32	1
	uma_zone_slab	0.12	1
	slab_zalloc	0.5	1
	uma_zalloc_internal	0.47	1
	uma_slab_alloc	0.06	1

Tables 4.9 shows the role of the functions with highest centrality value. Only 30% of functions are related to the functionality of the system. The table also shows that most of the functions chosen using this strategy are utility functions: Helper and clean up functions usually have higher fan-in value, and

Figure 4.6 Distribution of Centrality Values: Ying and Tarr's Approach



(a) `ispell`



(b) `FreeBSD Kernel Malloc`

The charts show the distribution of centrality value of all the function inside each system. The number indicates the range of centrality value. The centrality value of the functions mentioned in documentation is range from 0.05 to 0.5. This range covered 90% of the functions in both system.

initialization function usually have a higher fan-out value, and hence they have a misleadingly prominent ranking.

Table 4.10 also shows that the functions mentioned in the documentation do not have high degree of centrality (the centrality value of the functions mentioned in the documentation are all below 0.5). According to Figure 4.6, this range of centrality value covered 90% of the functions in the systems. This means that using centrality value to rank the functions is unable to effectively reduce the search space for finding important functions.

In contrast, the functions mentioned in documentation have a very high rating ranged from 0.95 to 1.00 using our FEPR approaches. This range of rating only covered 20% of the functions in the systems (Figure 4.3).

4.5 Top-down Search Approach

It has been shown that programmers tend to use a top-down approach when trying to gain a high-level overview of a system [vMV95]. In this case study, we mimic the process of searching for important functions using a top-down strategy by using *Breadth-First Search (BFS)* to try to reach the functions mentioned in the `ispell` and `FreeBSD` walkthrough guides.

We start from the root nodes in the call graph and count the number of functions that have to be visited before reaching each function in the walk-through guides. The measure for this study is the number of functions that have to be visited so that all the functions mentioned by the expert documentation are visited. We consider both left-to-right and right-to-left searching orders.

We picked `main` and `vm_mem_init` as the root nodes for the `ispell` and `FreeBSD` study respectively, because they are the first functions called when the system is executed.

4.5.1 Reduction Efficiency (RE)

Table 4.11 shows the number of nodes that have to be visited before reaching all the functions mentioned in the expert documentation. The developer has to visit an average of 74 ($\frac{91+57}{2}$) functions in `ispell` and 80 ($\frac{103+57}{2}$) functions in `FreeBSD Kernel Malloc` using the top-down approach.

Table 4.11 BFS Visiting Order on Original Call Graph

Code Base	**Functions**	**Order (Left to Right)**	**Order (Right to Left)**
`ispell`	*checkline*	83	52
	good	91	57
	checkfile	8	23
`FreeBSD Kernel Malloc`	*uma_zalloc*	101	55
	uma_zalloc_arg	102	56
	uma_zalloc_bucket	103	57
	uma_zone_slab	25	31
	slab_zalloc	13	37
	uma_zalloc_internal	15	29
	uma_slab_alloc	23	33

Since our FEPR-*fanin* approach and Ying and Tarr's approach do not preserve the call tree structure, we cannot evaluate their performance in direct comparison to the BFS approach. Therefore, we only compare the BFS results with the FEPR-*leaf* approach. These results are shown in Table 4.12.

When comparing the BFS results with the FEPR-*leaf* approach, only an average of 20.5 ($\frac{23+18}{2}$) and 22.5 ($\frac{28+17}{2}$) functions are visited in the reduced call graph of `ispell` and `FreeBSD Kernel Malloc` respectively. In contrast, an average of 74 ($\frac{91+57}{2}$) and 80 ($\frac{103+57}{2}$) functions are visited in the original call graph of `ispell` and `FreeBSD Kernel Malloc` (Results shown in Table 4.11). The FEPR-*leaf* approach reduced the number of nodes visited by about 70%.

Table 4.12 BFS Visiting Order Using FEPR-*leaf* Approach

Code Base	**Functions**	**Order (Left to Right)**	**Order (Right to Left)**
`ispell`	*checkline*	18	14
	good	23	18
	checkfile	7	7
`FreeBSD Kernel Malloc`	*uma_zalloc*	26	15
	uma_zalloc_arg	27	16
	uma_zalloc_bucket	28	17
	uma_zone_slab	17	12
	slab_zalloc	18	9
	uma_zalloc_internal	15	10
	uma_slab_alloc	missed	missed

4.6 Synthesized Analysis

4.6.1 Inclusion Accuracy (IA)

To evaluate accuracy, we compared our FEPR approaches with Ying and Tarr's approach and the centrality approach. The precision of the FEPR-*fanin* approach (`ispell`:0.05, `FreeBSD`:0.08) is similar to Ying and Tarr's approach (`ispell`:0.04, `FreeBSD`:0.083) but with a lower miss rate (`ispell`:no different, `FreeBSD`:0.3). The precision of the FEPR-*leaf* approach the best among the approaches (`ispell`:0.09, `FreeBSD`:0.17). The miss rate of the FEPR-*leaf* (`ispell`:0, `FreeBSD`:0.4) is slightly higher than that for FEPR-*fanin* (`ispell`:0, `FreeBSD`:0.3) approach, but still better than Ying and Tarr's approach (`ispell`:0, `FreeBSD`:0.5). FEPR approaches are more accurate in rating the importance of functions than the centrality approach. Functions mentioned in expert documentation do not have high centrality values, but obtain high rankings using our probability ranking approach.

4.6.2 Reduction Efficiency (RE)

To evaluate reduction efficiency, we compared our FEPR approaches with Ying and Tarr's approach and the top-down approach. The percentage reduction of our FEPR approaches is better than Ying and Tarr's approach in `ispell` and `FreeBSD` study, and reduced the node visited when finding important functions in the original call graph by 70%.

4.6.3 Stability (S)

To evaluate parameter stability, we compared the percentage reduction of the call graph using the FEPR approaches with Ying and Tarr's approach under different parameter settings. We found that our FEPR approaches are more stable than Ying and Tarr's approach: There is no sudden change in the size of the reduced call graph when the parameters change gradually using the FEPR approaches. In contrast, the percentage reduction by Ying and Tarr's approach changed sharply when we changed the parameter p_{bottom} from 1 to 2. Therefore, in these cases, our approaches perform better in terms of fine-tuning than Ying and Tarr's approach.

4.6.4 Threats to Validity

Internal Validity

The internal validity of our studies is threatened by the bias about the importance of a function. Determining whether a function is important or not is a subjective judgement. To minimize the potential bias, we identified exact matches from expert documentation. We also identified correct functions in the codebases before applying any call graph reduction or other analysis. In this way we were unable to choose functions that did well in our approach, but were ignored in others.

Construct Validity

The construct validity of our studies is threatened by the fact that Ying and Tarr's function filter approach is designed for Java language while the code bases of our studies are written in C. Although their heuristics do not make use any Object-Oriented language properties, it is possible that there are some potential difference in the reduction performance when applying their technique on C code.

Generalizability

We have evaluated our technique in two different kinds of systems: a spell checker (`ispell`) which is a small application-based system and an Operating System Kernel (`FreeBSD`) which is a large piece of system software. The case studies show that our technique upholds our claims in the two systems. This suggests our call graph reduction would also be effective in other systems.

4.7 Summary

In this chapter, we compared our FEPR approaches against other techniques to evaluate our thesis claims: FEPR approaches are better in inclusion accuracy than existing approaches; FEPR approaches can more effectively reduce the size of call graph; FEPR approaches results are stable enough for fine-tuning and are more stable than current approaches.

We evaluated the claim of improved inclusion accuracy by comparing the FEPR approaches to Ying and Tarr's approach and to the centrality approach. We showed that our FEPR approach has a higher precision, lower miss rate and better in ranking function importance.

We evaluate the claim of improved reduction efficiency by comparing the FEPR approaches against Ying and Tarr's approach and against a top-down search of the original call graph using BFS. We showed that the FEPR approaches are more efficient in reducing the size of call graph and in decreasing the number of functions to be visited.

We evaluated the claim of improved stability by comparing the FEPR approaches against Ying and Tarr's approach. We found that the reduction efficiency remains more stable under different thresholds of function execution probability than Ying and Tarr's approach.

CHAPTER 5

Discussion

5.1 Flexibility of Analysis

The static nature of the proposed approach allows for a high-degree of flexibility of analysis. Dynamic analysis suffers from the inherent limitation that it cannot be used to analyze unexecutable code, whereas static analysis does not.

In addition, analysis can be performed on part of the source code by replacing the original starter function, `main` as shown in the case study of `FreeBSD`. API library functions can also be analyzed in a similar way. After constructing the Branch-Preserving Call Graph of the API library, the function probability spectrum can be estimated by using the API interface function as the starter function.

5.2 Existence of Function Pointers, GOTOs and Early Exits

Ideally, a Branch-Preserving Call Graph will capture all of the possible control flow of a program. However, the dynamic behavior of the programming language, including the use of function pointers, increases the difficulty in constructive the BPCG.

Antoniol *et al.* show that the existence of function pointers can hurt the accuracy of the constructed call graph [ACT99]. Additionally, the existence of GOTO statements and early exits increase the difficulty of building an accurate BPCG.

While building a completely precise call graph is equivalent to solving the halting problem in general, there have been attempts to construct a high precision call graph in the presence of function pointers ([MRR04], [Atk04]). However, the implementation of a call graph builder is non-trivial. Data flow information must also be considered during call graph construction. This will make the construction and further extension of the BPCG much more complicated.

It would be possible to use partial analysis to get a localized set of priorities for a subsystem that is reachable only by function pointers. Just like we can use `vm_fault` as the starter function when analyzing the page fault handling mechanism.

5.3 Precision of Branch-Preserving Call Graphs

There is always a tradeoff between computational complexity and precision. A Branch-Preserving Call Graph is a simple abstract representation of source code which provides just enough information to estimate the execution probability of the software. By ignoring data flow information, the construction of a BPCG is more efficient than other static abstract representation such as the dependence graph which is computationally expensive [Wei84]. It is also computationally efficient to use BPCG for further processing.

However, the weakness of BPCG is also due to the absence of data information. Consider the code fragment shown in Figure 5.1. in this work, function f cannot be detected as unreachable code.

Figure 5.1 Example Showing the Imprecision of BPCG

```
if (x > 1 && x < 1) {
    f(x);
}
```

Constraint Propagation is one way to solve this problem[KR00, Bin94]. Using the techniques in constraint programming, branch prediction can be more accurate. Although precise, there is a computational complexity tradeoff.

5.4 Function Ranking and Recommender System

Other than using the estimated function execution probability for call graph reduction, it also provides a mean to quantify and rank the importance for the functions in a certain code base.

Table 5.1 shows an example of estimated execution probability for the functions with highest rating in `sort.c`, which is a UNIX sort utility included in the GNU Coreutils package[1] comprised of about 2500 lines of code and 39 functions. The functions with a high rating are considered as important and suggested to be investigated first when understanding an unfamiliar code base. Our technique in execution probability estimation can be integrated with Recommender System such that the importance of functions is also be considered.

[1]GNU Coreutils can be downloaded at http://www.gnu.org/software/coreutils. In here, Coreutils 5.0 is used.

Table 5.1 Estimated Function Execution Probability for functions in `sort.c`

Function	**Rating**
inittables	1
buffer_linelim	1
fillbuf	1
compare	1
sigemptyset	1
sigaction	1
main	1
posix2_version	1
bindtextdomain	1
textdomain	1
hard_locale	1
localeconv	1
sigaddset	1
gettext	0.996019
error	0.995581
die	0.99473
__builtin_alloca	0.953705
keycompare	0.953705
limfield	0.951715
xalloc_die	0.93896
initbuf	0.9375
begfield	0.908898
trim_trailing_blanks	0.908898
create_temp_file	0.875
sigprocmask	0.875
mkstemp	0.875
fstat	0.75
stat	0.75
mergefps	0.75
merge	0.75

5.5 Extending the Approach Beyond C

Currently, this approach is implemented to work on code written in the C programming language. In theory, our approach can be applied to other languages as well, such as C++ or Java. However, as mentioned before, the precision of the approach relies on the accuracy of the construction of a correct Branch Preserving Call Graph. With the existence of function pointers, call graph construction is not trivial. Object-oriented programming languages present further problems, due to their reliance on dynamic properties such as polymorphism and dynamic binding. Static call graph construction algorithms are available [GDDC97], and the approaches could be straightforwardly extended to extract Branch-Preserving Call Graphs, but the trade-off between accuracy and efficiency is unavoidable.

CHAPTER 6

Related Work

We categorize the related work in two groups: current program understanding approaches (section 6.1), branch prediction and static profiling approaches (section 6.2).

6.1 Existing Approaches in Program Understanding

In this section, we describe work aimed at helping developers understand source code.

6.1.1 Localized Program Understanding

Localized program understanding approaches help developers to investigate a program from a particular starting point. Our FEPR approaches differ from these localized approaches in that they are intended to provide a general overview of the source code, but not to facilitate a focused investigation of specialized areas of interest.

Program Slicing : Program slicing identifies the parts of a program that may affect the values computed at some point of interest [Wei79]: it extracts the program elements that are potentially affected by, or that affect the control flow or data flow of the selected statements or variables. As such, program slicing is a localized analysis technique. It can facilitate investigation of one, or of a collection of points of interest. It is not, however, intended to provide a general overview of a code base, or to identify potential points of interest.

Although program slicing is a useful technique for reverse engineering, the construction of program slices is time consuming [Wei84]. The size of program slice is usually large [Wei84, BGH07]. Our approach, on the other hand, has a complexity of $O((V + E)^2)$, and even for a large program works in less than an hour.

Even though a program slice can be view as a reduced call graph, FEPR approaches can achieve a better reduction efficiency than program slicing in terms of narrowing the whole program down to points of interest for a newcomer to a code base. Program slicing is not suitable for initial understanding as points of interest must be provided, while it is not possible for a developer to come up with the potential points of interest without inspecting the code manually. The FEPR approaches do not require developers to provide any starting points of investigation. Developers who do not have knowledge about the system are still able to apply the FEPR approaches. Furthermore, the estimated execution probability of the functions can provide hints to developers in locating the starting points for investigation on the reduced call graph.

Recommender System : A recommender system can suggest related methods in response to user queries. When provided a set of methods of interest, the system can automatically suggest a list of related methods [IYF$^+$03, IYYK05, SFDB07, Rob05]. Xie et al. make use of external open source repositories in mining API usage patterns. In their work, a developer can obtain suggestions for API invocation sequences and usages samples by providing a query describing a method or class of an API [XP06, TX07].

Recommender systems can help developers search for functions related to a function of interest. Like program slicing, user has to provide starting points of investigation when using recommender systems. It is not possible for a developer to come up with such stating points when he is completely new to the system. In contrast, FEPR technique does not require any user input. It operates on the whole system and estimate the execution probability of each function. For this reason, FEPR is a more suitable technique for initial understanding.

6.1.2 Whole Program Analysis

Whole program analysis techniques allow developers to obtain an overview of a system without providing a starting point.

Structural Recognition

Structure recognition tools are aimed at discovering candidate modules from source code. They facilitate refactoring of the source code into a more desired modularization. This differs from the FEPR approaches because the FEPR approaches are intended to facilitate initial understanding, but not provide support for restructuring or structure discovery. Such approaches are valuable for positing a new structure for a system, but are less effective in helping a developer navigate and find salient points of an existing system structure.

Component and Aspect Mining : The goal of component mining is to identify software components from the source code. Graph-based partitioning [SGMB03, GK97, LL03] and metric-based partitioning [TH99, TH00] are the two major approaches in component mining. Inoue et al. proposed a component ranking technique called Component Rank [IYF$^+$03] based on the page rank web-searching technique [PBMW98].

Aspect mining involves the identification of crosscutting concerns in the source code. Crosscutting concerns are identified by looking for functionality that is repeated, or that does not fit cleanly into the existing modularization of a system. Different approaches based on fan-in analysis [MvDM04], clone detection [BDET04, BDET05], version history [BZ06] and random walk [ZJ07] has been proposed.

Component and aspect mining focus on mining candidate components that can be refactored into modules. They do not tell the developer about how these modules interact with each other. FEPR approaches aimed at trim down the size of a call graph such that it is more useful in providing an high level overview to the developers. In addition, call relations between core functions are retained in the reduced call graph. This is particularly useful for developers attempting to understand the call interactions between the functions.

Conceptual Modules : Conceptual modules is a technique to allow the developer to posit new desired modules for a code base [BM98]. A conceptual module is a logical module which consists of a set of lines of code. Once it is defined, a developer can perform queries of a desired structure on conceptual modules to find out the correspondent code segment in the source code.

The conceptual module approach is intended to aid the developer in finding relevant code segments for a desired structure. It is not suitable for obtaining an initial understanding of a code base as the lines contributing to a conceptual module have to be specified by a developer who has already established an initial understanding of the system. Even if the conceptual modules were already specified, a developer who is new to the system would not know where to start in terms of navigating and understanding the structure.

Structure Overview

In this section, we describe the techniques that aimed at providing an overview on the code base.

Pattern Miners : Pattern miners [SKL$^+$02, PIKK98, Rad00] allow a developer to discover design patterns in the source code. Given an architectural description, pattern miners return a collection of segments of code that together conform to that description.

While pattern mining approaches can aid developer understand a system by locating the code segment related to some particular patterns, they cannot tell the developer which part of the source code is more important to the core functionality. This is important for initial understanding as developer does not know where to start even if a high level overview were already obtained through pattern mining. Using the FEPR approaches, functions that are not functionaly significant are filtered out in the reduced call graph. The relative importance of functions can be deduced by the estimated execution probability.

Feature Location : Feature location affords a developer an understanding of a new system by providing a mapping between the source code and the features of a program. This effort can be split into static approaches and dynamic approaches. Some static approaches involve the use of natural language techniques [MM03, DDL$^+$90] or information retrieval approaches [ACC$^+$02, ZZL$^+$06]. However, it is less accurate for these static approaches. Dynamic approaches include the use of scenario-driven analysis and software reconnaisance [Egy03, EKS01, EKS03, WS95, WC96]. These approaches must be applied to a runnable system. Scenarios and test cases have to be carefully designed before performing the analysis. FEPR approaches are static techniques and can be applied to a non-running system.

Feature location techniques intend to reestablish linkage between source code and features. Therefore developer must have a list of features or scenarios as input when applying these tools. Because of this, feature location approaches are not suitable for initial understanding as the features or scenarios of a system can only be provided by the developers who are already familiar with the system. In contrast, the FEPR approaches do not require human intervention during the analysis and no initial understanding of the system is required.

Semiautomatic Overview Tools : Semiautomatic overview tools such as Rigi [MK88] and RMTool [Mur96] can aid developer in gleaning a high level structure of the source code. Rigi is a reverse engineering tool through which a developer can obtain a high level visualization of system structure by applying functionality clustering operations.

RMTool is intended to assist the establishment of architectural conformance of a system. It requires a developer to specify an architecture for conformance checking. However, this cannot be done by a developer who knows nothing about the system. Furthermore, developers who would like to form an initial understanding are likely not initially interested in the architecture's conformance. Even if an architecture diagram of the code base were to be given to the developer for the sake of applying RMTool, the tool would not tell them which portions of the code are central to understanding the system.

Rigi is intended to help developers build up an high level overview of a system architecture. Developers have to perform visual clustering of the visualization of system structure until an high level overview is obtained. This process is infeasible for a developer who is new to the system as the clustering process requires knowledge of the system. Additionally, like RMTool, Rigi does not provide any hints about which part of the source code is core to the system.

Unlike these tools, the FEPR approaches are intended to help developers who are inexperienced with the code base to know where to start their task of forming an initial understanding. Users are not required to have any knowledge about the system when applying the FEPR approaches. The estimated execution probability of functions also provide hints to developer where to start in the reduced call graph.

6.2 Branch Prediction and Static Profiling

The goal of static profiling is to estimate the program profile of a program prior to runtime. Program profiles may be split into different kinds of profiles, such as block count, procedure time, reference count or procedure entry count. Program profiles are particularly useful in compiler optimization. Since static profiling relies on the structure of the source code to predict the runtime behavior of the program, accurate branch prediction is necessary.

Static Branch Prediction [BL93] predicts the direction of branches, (conditional branches and loop branches) based on the original source code, or based on the binary code structure. By knowing the branching behavior, a compiler can obtain a better scheduling of machine instructions thus yield a better optimization.

There are many techniques for branch prediction that use a simple heuristic [WMGH94, WL94] or constraint propagation [KR00, Bin94]. Wagner et al. use a simple estimate of branch probabilities in their work in program optimization [WMGH94]. They assign a predefined weighting for each kind of branching statement during calculation. Ball and Larus obtain a list of static heuristics for branch prediction [BL93] by analyzing the statistics of branching behavior. Their result was used by Boogerd and Moonen for their work in software inspection [BM06].

Some approaches for branch prediction were integrated into generalized techniques for static profiling. Wagner et al. also apply Ball and Larus's branching heuristics [BL93] to estimate the block execution frequencies profile [WMGH94]. However, their estimation is limited in that it is only capable of inter-procedural level analysis and cannot extend to perform whole-program analysis directly. Wu and Larus applied the same branching heuristics in estimating execution frequency of function calls [WL94]. They improve Wanger's work by combining different heuristics and extending the scope of analysis. Wu and Larus apply the Dampster-Shafer theory, a method of calculating event probability by combining different pieces of evidence, to combine the predictions obtained from different heuristics. Their approach has been shown to work at an intra-procedure level.

The estimation of loop trip count is necessary in both Wu's and Wagner's approaches in order to estimate the frequency profile. However, loop-trip count is not straightforward to establish. While both approaches use Ball and Larus's Loop branch heuristic when estimating the loop trip count, there is considerable doubt as to the accuracy of the Loop branch heuristic. When considering the mean miss rate and the standard deviation (Mean: 25%, Standard Deviation: 28%) as reported in the Ball and Larus study [BL93], it is clear that enormous errors can be made, especially when nested loops are encountered. Execution frequency can also be overestimated by assuming that the execution paths are independent. Moreover, as point out by Wong [Won99], their approaches are on machine code level and are susceptible to differences in compilers and architectures.

Wong has proposed a set of heuristics for branch prediction at the source code level [Won99]. He observed that some kind of branches, such as branches contain print out statement to `STDERR`, are unlikely to be executed at run time. By considering the general programmer coding habit, it is likely to obtain to more precise branch prediction.

The FEPR approaches avoid the estimation of loop trip count by estimating the function execution probability by graph reachability instead of function execution frequency: It is impossible for execution frequency to be accurately estimated using static analysis.

In this work, all branches are "assumed equal", because as there is no way to predict the actual probability profile without running the program. Although more accurate results maybe achievable by adopting branch heuristics, heuristics like [WMGH94, BL93] are at the machine code level which are not appropriate to apply them on our work. Wong's source code level heuristics [Won99] maybe suitable for our work, but these heuristics have not deeply statistically evaluated as in Ball's work [BL93]. Furthermore, we believed that acceptable accuracy can be achieved without the use branch heuristics. High accuracy has been reported in probability-based recommender systems [Rob05, SFDB07] even though their systems assumed that each method call has the same probability of execution.

CHAPTER 7

Conclusions

It is not easy to navigate the source code to understand the program. Our preliminary study found that developer intuition does not help much in identifying core functions in a call graph, especially when the call structure is complicated and large. We believed that this inability is due to the large number of functions in the call graph that are not core to the functionality of the program.

Our aim, is to reduce the size of call graph so that a developer can focus on the functions that are core to the functionality of the program. To achieve this goal, we proposed a technique to filter out the functions in the call graph with a low estimated probability of execution.

The thesis of this work is that by filtering the call graph to retain only those functions with a high estimated probability of execution, we can arrive at a reduced call graph that is smaller, has a higher percentage of important functions, and which is more stable in the face of changing thresholds of function inclusion, than current approaches.

To validate the thesis claims, we have conducted a series of case studies. To evaluate accuracy, we compared our FEPR approaches with Ying and Tarr's approach and the centrality approach. The results showed that our FEPR approaches are better than Ying and Tarr's approach in inclusion accuracy and better than the centrality approach in rating the importance of functions. To evaluation reduction efficiency, we compared our approaches with Ying and Tarr's approach and with a top-down approach. The results showed that

FEPR approach is better than Ying and Tarr's approach in reducing the size of call graph and in reducing the number of nodes to be visited when searching for core functions. To evaluate parameter stability, we compared the percentage reduction of the call graph using the FEPR approaches with Ying and Tarr's approach under different parameter settings. We found that our FEPR approaches are more stable than Ying and Tarr's approach.

In addition to the demonstrating the validity of the thesis statement, the research makes three contributions.

First, we have conducted a preliminary study showing the developer's inability in identifying core functions when faced with a large call graph.

Second, we provided a variation of the call graph, called the Branch-Preserving Call Graph, to represent call relations and branching relations of the source code.

Third, we provided two FEPR approaches in call graph reduction and validated that our approaches are more effective than current approaches.

There are many possible future directions for this work as discussed in Chapter 5. One is to incorporate Wong's code level branching heuristics [Won99] into our heuristic when estimating the function execution probability. This may help improving the accuracy of our call graph reduction approaches. Another possible direction is to extend our work to Recommender System. This would provide a better accuracy as the importance of function is also taken into account.

Appendix A

Call Graphs in Case Studies

This chapter includes the call graphs used in validation:

1. **Figure A.1:** Reduced Call Graph by *remove-high-fan-in-functions* approach (Threshold=0.5, Fan-in Max=4)
2. **Figure A.2:** Reduced Call Graph by *remove-leaf-nodes* approach (Threshold=0.5).
3. **Figure A.3:** Reduced Call Graph by Ying and Tarr's approach ($p_{bottom} = 1$, $p_{small} = 2$).
4. **Figure A.4:** Reduced Call Graph by *remove-high-fan-in-functions* approach (Threshold=0.4, Fan-in Max=4).
5. **Figure A.5:** Reduced Call Graph by *remove-leaf-nodes* approach (Threshold=0.4).
6. **Figure A.6:** Reduced Call Graph by Ying and Tarr's approach ($p_{bottom} = 1$, $p_{small} = 2$).

Figure A.1 Reduced Call Graph by *remove-high-fan-in-functions* approach (Threshold=0.5, Fan-in Max=4)

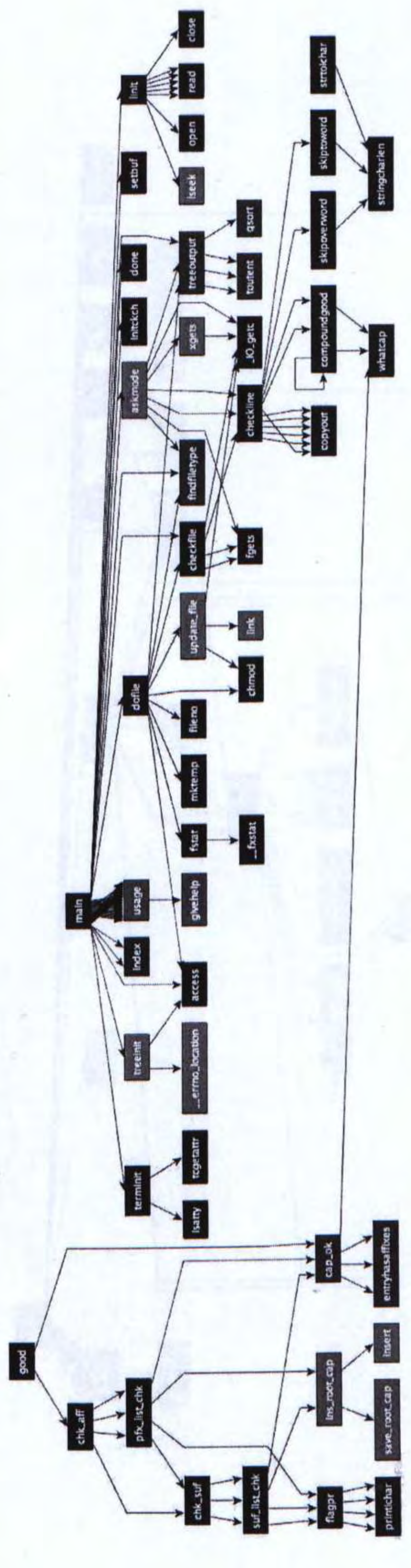


Figure A.2 Reduced Call Graph by *remove-leaf-nodes* approach (Threshold=0.5)

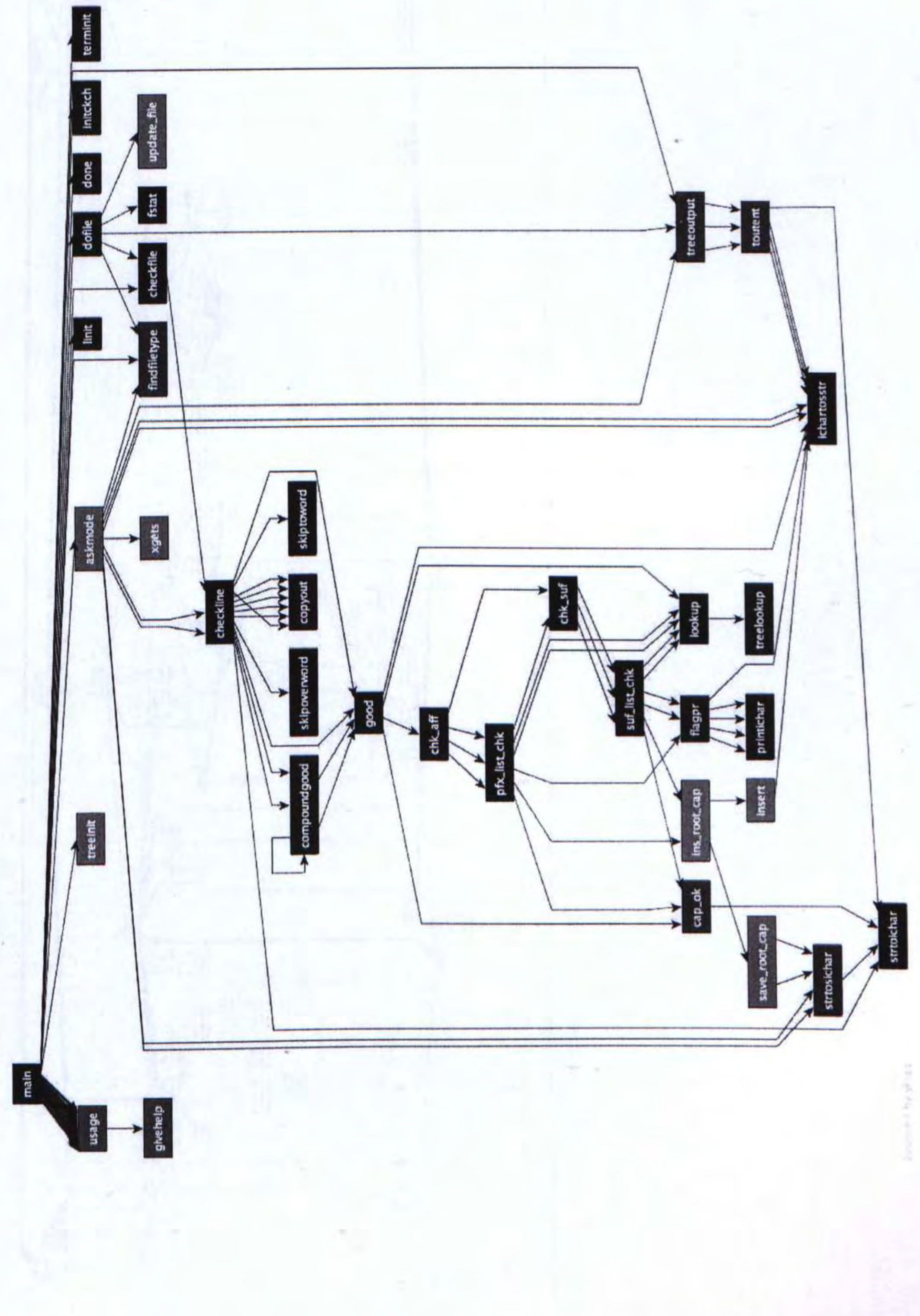


Figure A.3 Reduced Call Graph by Ying and Tarr's approach ($p_{bottom} = 1$, $p_{small} = 2$)

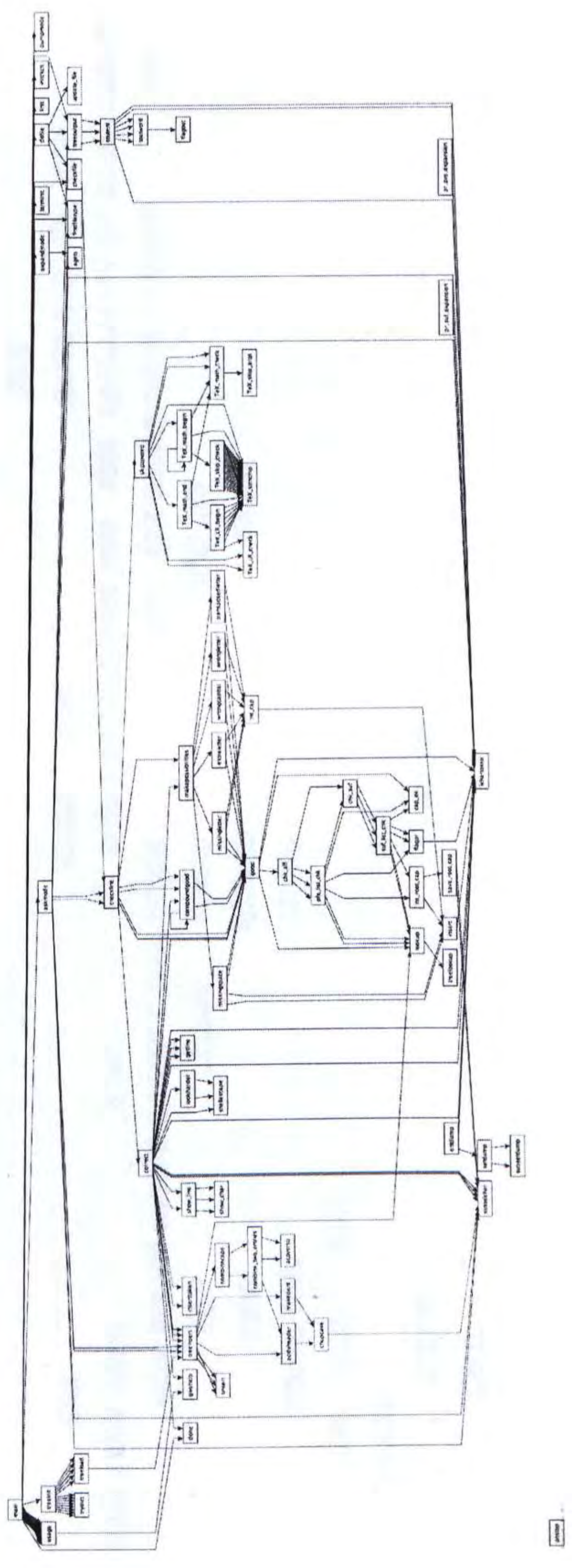


Figure A.4 Reduced Call Graph by *remove-high-fan-in-functions* approach (Threshold=0.4, Fan-in Max=4)

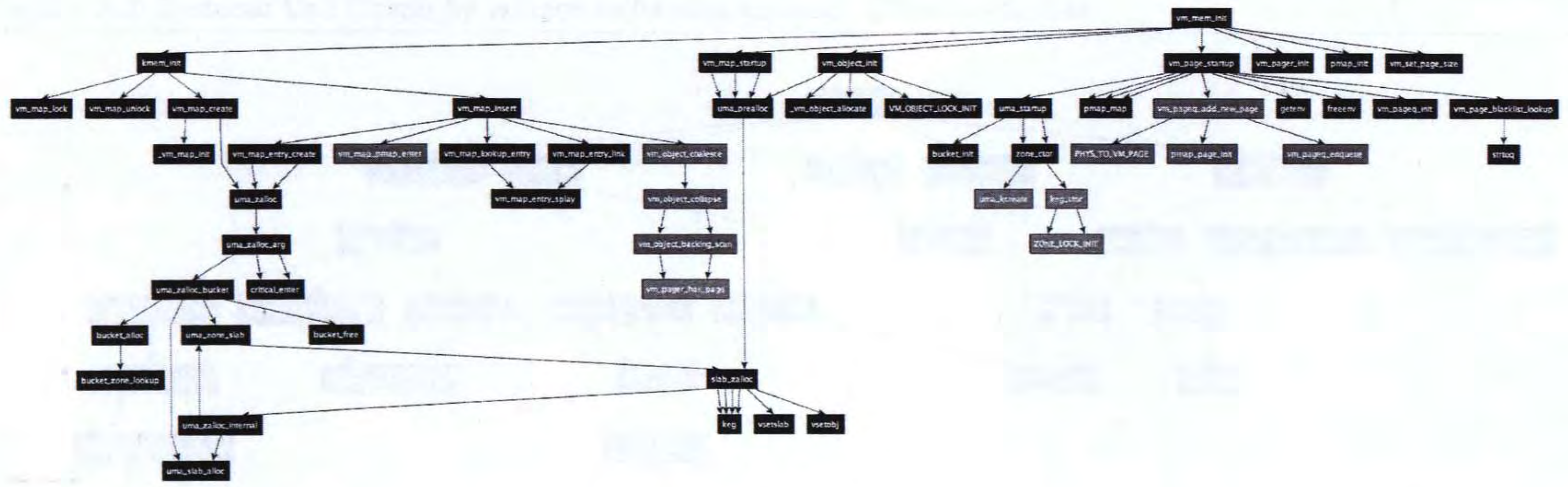


Figure A.5 Reduced Call Graph by *remove-leaf-nodes* approach (Threshold=0.4)

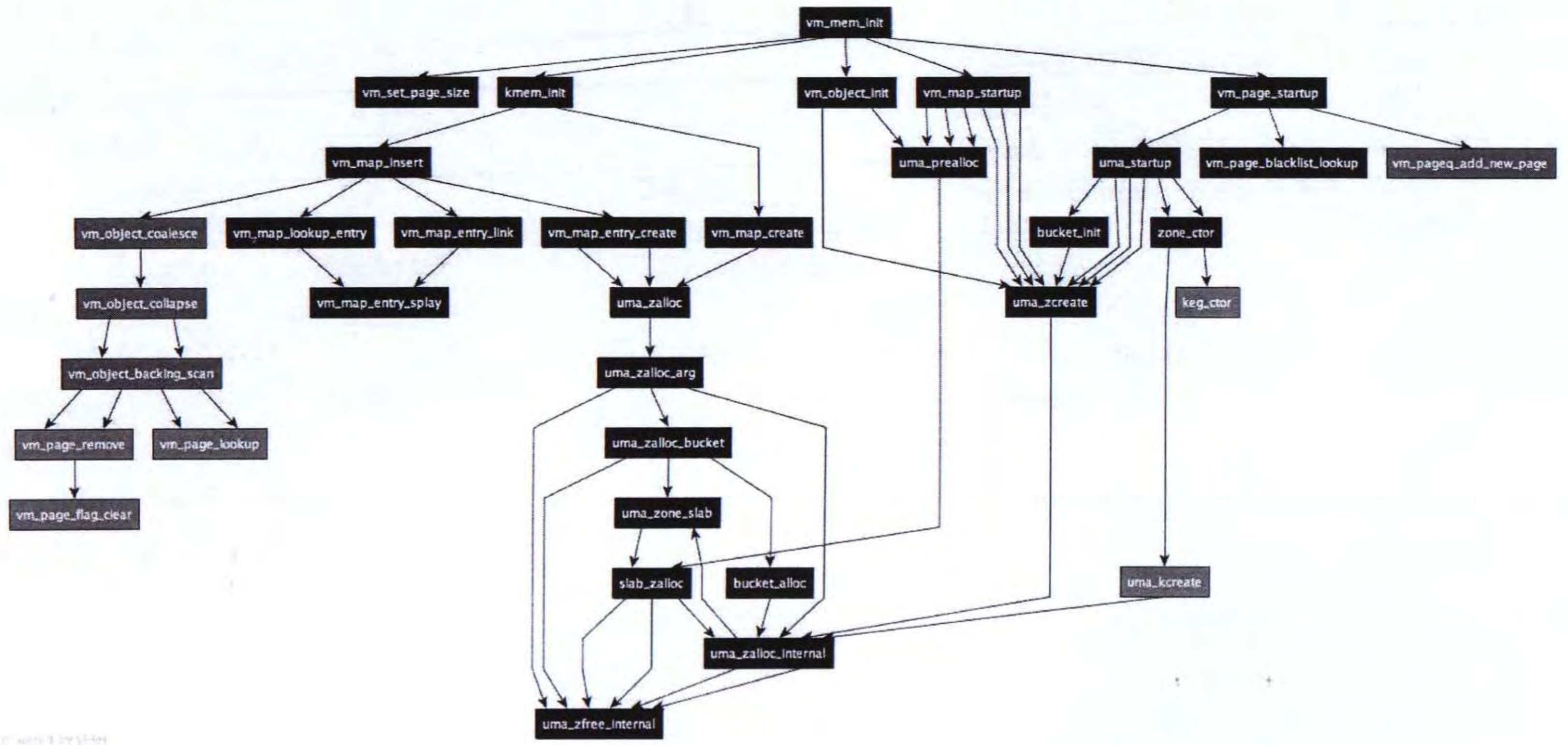
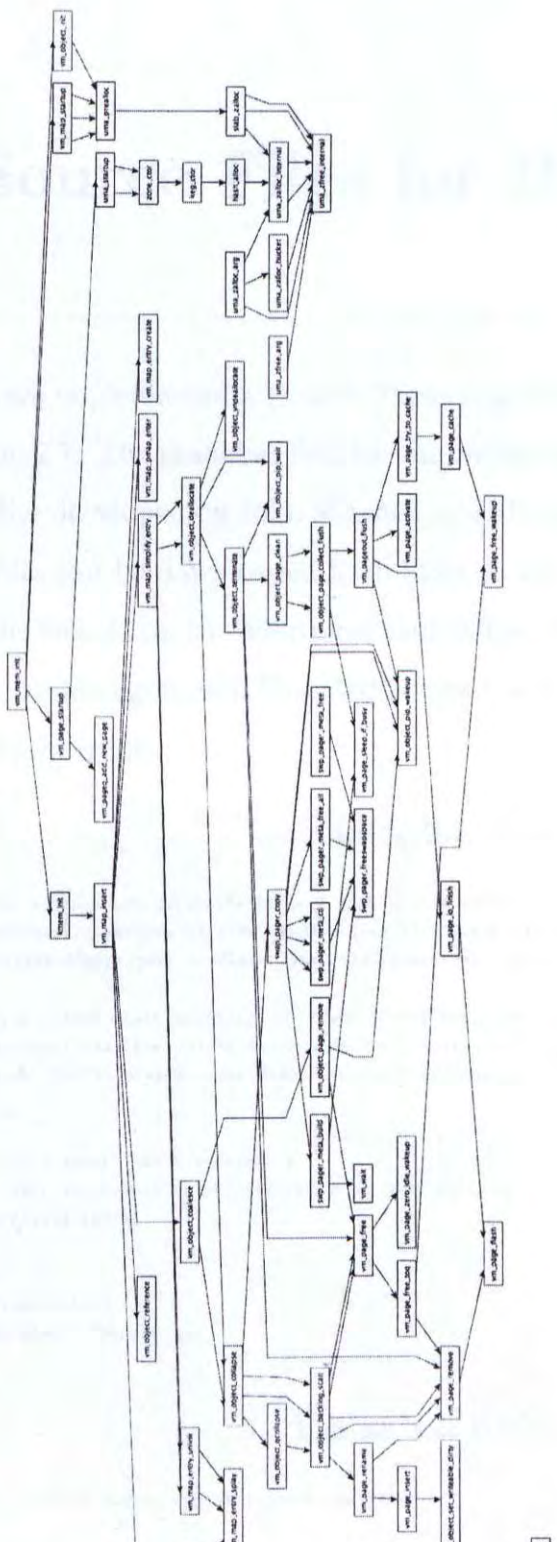


Figure A.6 Reduced Call Graph by Ying and Tarr's approach ($p_{bottom} = 1$, $p_{small} = 2$)



APPENDIX B

Source Files for BPCG Builder

We have implemented a Branch-Preserving Call Graph Builder using ANTLR version 2.7. The grammar files for our builder is modified by the GNU C grammer files developed by John Mitchell and Monty Zukowski. The original grammar files can be downloaded from `http://www.antlr.org/grammar/cgram`.

The `Makefile`, the additional source files (`BPCG.java` and `Node.java`), and the 3 modified grammar files (`StdCParser.g`, `GnuCParser.g` and `expandedCParser.g`) are shown below.

Listing B.1: Makefile

```
javafiles = BPCG.java CSymbolTable.java CToken.java GNUCTokenTypes.java GnuCLexer.java
     GnuCLexerTokenTypes.java GnuCParser.java LineObject.java Node.java PreprocessorInfoChannel.java
     STDCTokenTypes.java StdCLexer.java StdCParser.java TNode.java

classfiles = BPCG.class CSymbolTable.class CToken.class GNUCTokenTypes.class GnuCLexer.class
     GnuCLexerTokenTypes.class GnuCParser.class LineObject.class Node.class PreprocessorInfoChannel.
     class STDCTokenTypes.class StdCLexer.class StdCParser.class TNode.class

all :
        java antlr.Tool StdCParser.g
        java antlr.Tool -glib "StdCParser.g" GnuCParser.g
  javac $(javafiles)

clean :
  rm $(classfiles)
  rm expandedGnuCParser.g
```

Listing B.2: BPCG.java

```
/*
 * BPCG - Branch-Preserving Call Graph generator
 *
 * AUTHOR: KK Lo, 7/2007
 *
 * DESCRIPTION:
 *      Output the branch-preserving call graph for c source code in
 *      Dot format to STDOUT.
 *
 *      Usage:
 *              Reading from file(s):
 *                      java BPCG <file 1> <file 2> ... <file n>
 *              Reading from STDIN:
 *                      java BPCG -
 *
 *      This call graph generator required ANTLR 2.7 to compile and
 *      run.
 *
 * REVISION HISTORY:
 *      Date            Name            Description
 *      11.07.2007      KK Lo           Initial Release
 *
 */

import java.io.*;
import antlr.*;

public class BPCG{

    public static void main(String[] args){
        //Print out the Dot header
        System.out.println("digraph G {");

        for (int i=0; i<args.length; i++){
            try{

                String fileName = args[i];
                DataInputStream dis = null;
                if (fileName.equals("-")) {
                    //Reading from STDIN
                    dis = new DataInputStream( System.in );
                }
                else {
                    //Reading from files
                    dis = new DataInputStream(
                                new FileInputStream(fileName));
                }

                GnuCLexer lexer = new GnuCLexer ( dis );
                lexer.setTokenObjectClass("CToken");
                lexer.initialize();

                // Parse the input expression.
                GnuCParser parser = new GnuCParser (lexer);

                // set AST node type to TNode or get nasty
                // cast class errors
                parser.setASTNodeType(TNode.class.getName());
                TNode.setTokenVocabulary("GNUCTokenTypes");

                // invoke parser
                try {
                    parser.translationUnit(new Node("Root"));
                }
                catch (RecognitionException e) {
                    System.err.println("Fatal IO error:\n"+e);
                    System.exit(1);
                }
                catch (TokenStreamException e) {
                    System.err.println("Fatal IO error:\n"+e);
                    System.exit(1);
                }

                // Garbage collection hint
                System.gc();
            }
            catch ( IOException e ){
                System.err.println("Error in input:");
                System.err.println (e);
                e.printStackTrace();
            }
        }
        //Footer of Dot file format
        System.out.println("}");
    }
}
```

Listing B.3: Node.java

```
/*
 * Node - A node object represent a Or-node, And-node, function-node
 *        in Branch-Preserving Call Graph
 *
 * AUTHOR: KK Lo, 7/2007
 *
 * REVISION HISTORY:
 *     Date             Name             Description
 *     11.07.2007       KK Lo            Initial Release
 *
 */

public class Node{
        private String s;
        // static ID for and-node & or-node such that each node have
        // different ID
        private static long andId=0;
        private static long orId=0;

        /**
         * Constructor:
         * @param type  type of the node. E.g. function
         */
        public  Node(String type){
            long myId;
            if(type.equals("AND")){
                myId = andId++;
                s = "AND_" + myId;
            }
            else if(type.equals("OR")){
                myId = orId++;
                s = "OR_" + myId;
            }
            else{
                s = type;
            }
        }

        /**
         * overrided the toString method
         */
        public String toString(){
            return s;
        }

}
```

Listing B.4: StdCParser.g

```
/*
 * StdCParser.g - Grammar file for Std-C Grammar
 *              - Modified for building Branch-Preserving Call Graph
 *
 * AUTHOR: KK Lo, 7/2007
 *
 * REVISION HISTORY:
 *      Date            Name            Description
 *      11.07.2007      KK Lo           Initial Release
 */

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  Copyright (c) Non, Inc. 1997 -- All Rights Reserved

PROJECT:        C Compiler
MODULE:         Parser
FILE:           stdc.g

AUTHOR:         John D. Mitchell (john@non.net), Jul 12, 1997

REVISION HISTORY:

Name    Date            Description
----    ----            -----------
JDM     97.07.12        Initial version.
JTC     97.11.18        Declaration vs declarator & misc. hacking.
JDM     97.11.20        Fixed:  declaration vs funcDef,
parenthesized expressions,
declarator iteration,
varargs recognition,
empty source file recognition,
and some typos.


DESCRIPTION:

This grammar supports the Standard C language.

Note clearly that this grammar does *NOT* deal with
preprocessor functionality (including things like trigraphs)
Nor does this grammar deal with multi-byte characters nor strings
containing multi-byte characters [these constructs are "exercises
for the reader" as it were :-)].

Please refer to the ISO/ANSI C Language Standard if you believe
this grammar to be in error.  Please cite chapter and verse in any
```

```
correspondence to the author to back up your claim.

TODO:

- typedefName is commented out, needs a symbol table to resolve
ambiguity.

- trees

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/


{
    import java.io.*;
    import java.util.*;
    import antlr.CommonAST;
    import antlr.DumpASTVisitor;

}


class StdCParser extends Parser;

options
{
    k = 2;
    exportVocab = STDC;
    buildAST = true;
    ASTLabelType = "TNode";

    // Copied following options from java grammar.
    codeGenMakeSwitchThreshold = 2;
    codeGenBitsetTestThreshold = 3;
}
{

    // Suppport C++-style single-line comments?
    public static boolean CPPComments = true;

    // access to symbol table
    public CSymbolTable symbolTable = new CSymbolTable();

    // source for names to unnamed scopes
    protected int unnamedScopeCounter = 0;

    public boolean isTypedefName(String name) {
        boolean returnValue = false;
        TNode node = symbolTable.lookupNameInCurrentScope(name);
        for (; node != null; node = (TNode) node.getNextSibling() ) {
            if(node.getType() == LITERAL_typedef) {
                returnValue = true;
                break;
            }
        }
        return returnValue;
    }


    public String getAScopeName() {
        return "" + (unnamedScopeCounter++);
    }

    public void pushScope(String scopeName) {
        symbolTable.pushScope(scopeName);
    }

    public void popScope() {
        symbolTable.popScope();
    }

    int traceDepth = 0;
    public void reportError(RecognitionException ex) {
        try {
            System.err.println("ANTLR Parsing Error: "+ex + " token name:" + tokenNames[LA(1)]);
            ex.printStackTrace(System.err);
        }
        catch (TokenStreamException e) {
            System.err.println("ANTLR Parsing Error: "+ex);
            ex.printStackTrace(System.err);
        }
    }
    public void reportError(String s) {
        System.err.println("ANTLR Parsing Error from String: " + s);
    }
    public void reportWarning(String s) {
        System.err.println("ANTLR Parsing Warning from String: " + s);
    }
    public void match(int t) throws MismatchedTokenException {
        boolean debugging = false;

        if ( debugging ) {
            for (int x=0; x<traceDepth; x++) System.out.print(" ");
            try {
                System.out.println("Match("+tokenNames[t]+") with LA(1)="+
                        tokenNames[LA(1)] + ((inputState.guessing>0)?" [inputState.guessing "+
                            inputState.guessing + "]":""));
            }
            catch (TokenStreamException e) {
                System.out.println("Match("+tokenNames[t]+") " + ((inputState.guessing>0)?" [inputState.
                    guessing "+ inputState.guessing + "]":""));

            }

        }
        try {
            if ( LA(1)!=t ) {
                if ( debugging ){
                    for (int x=0; x<traceDepth; x++) System.out.print(" ");
                    System.out.println("token mismatch: "+tokenNames[LA(1)]
                            + "!="+tokenNames[t]);
                }
                throw new MismatchedTokenException(tokenNames, LT(1), t, false, getFilename());

            } else {
                // mark token as consumed -- fetch next token deferred until LA/LT
                consume();
            }
        }
        catch (TokenStreamException e) {
        }

    }
    public void traceIn(String rname) {
        traceDepth += 1;
        for (int x=0; x<traceDepth; x++) System.out.print(" ");
        try {
            System.out.println("> "+rname+"; LA(1)==("+ tokenNames[LT(1).getType()]
                    + ") " + LT(1).getText() + " [inputState.guessing "+ inputState.guessing + "]");
        }
        catch (TokenStreamException e) {
        }
    }
    public void traceOut(String rname) {
        for (int x=0; x<traceDepth; x++) System.out.print(" ");
        try {
            System.out.println("< "+rname+"; LA(1)==("+ tokenNames[LT(1).getType()]
                    + ") "+LT(1).getText() + " [inputState.guessing "+ inputState.guessing + "]");
        }
        catch (TokenStreamException e) {
        }
        traceDepth -= 1;
    }

}


translationUnit[Node parent]
:       externalList[parent]

|       /* Empty source files are *not* allowed.  */
{
    System.err.println ( "Empty source file!" );
}
;


externalList[Node parent]
:       ( externalDef[parent] )+
;


externalDef[Node parent]
:       ( "typedef" | declaration[parent] )=> declaration[parent]
|       functionDef[parent]
|       asm_expr[parent]
;


asm_expr[Node parent]
:       "asm"^
("volatile")? LCURLY! expr[parent] RCURLY! SEMI!
;


declaration[Node parent]
{ AST ds1 = null; }
:       ds:declSpecifiers[parent]       { ds1 = astFactory.dupList(#ds); }
```

```
(
 initDeclList[ds1, parent]
)?
SEMI!
{ ## = #( #[NDeclaration], ##); }

;


declSpecifiers[Node parent]
{ int specCount=0; }
:       (               options { // this loop properly aborts when
        //  it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :
s:storageClassSpecifier
| typeQualifier
| ( "struct" | "union" | "enum" | typeSpecifier[specCount, parent] )=>
specCount = typeSpecifier[specCount, parent]
        )+
;

storageClassSpecifier
:       "auto"
|       "register"
|       "typedef"
|       functionStorageClassSpecifier
;


functionStorageClassSpecifier
:       "extern"
|       "static"
;


typeQualifier
:       "const"
|       "volatile"
;

typeSpecifier [int specCount, Node parent] returns [int retSpecCount]
{ retSpecCount = specCount + 1; }
:
(       "void"
        |       "char"
        |       "short"
        |       "int"
        |       "long"
        |       "float"
        |       "double"
        |       "signed"
        |       "unsigned"
        |       structOrUnionSpecifier[parent]
        |       enumSpecifier[parent]
        |       { specCount == 0 }? typedefName
)
;
```

```
typedefName
:       { isTypedefName ( LT(1).getText() ) }?
i:ID                    { ## = #(#[NTypedefName], #i); }
;

structOrUnionSpecifier [Node parent]
{ String scopeName; }
:       sou:structOrUnion!
( ( ID LCURLY )=> i:ID l:LCURLY
  {
  scopeName = #sou.getText() + " " + #i.getText();
#l.setText(scopeName);
  pushScope(scopeName);
  }
  structDeclarationList [parent]
  { popScope();}
  RCURLY!
  |   l1:LCURLY
  {
  scopeName = getAScopeName();
#l1.setText(scopeName);
  pushScope(scopeName);
  }
  structDeclarationList [parent]
  { popScope(); }
  RCURLY!
  | ID
)
{
## = #( #sou, ## );
}
;


structOrUnion
:       "struct"
|       "union"
;


structDeclarationList [Node parent]
:       ( structDeclaration [parent] )+
;


structDeclaration [Node parent]
:       specifierQualifierList [parent] structDeclaratorList [parent] ( SEMI! )+
;


specifierQualifierList [Node parent]
{ int specCount = 0; }
:       (               options {   // this loop properly aborts when
        // it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :
        ( "struct" | "union" | "enum" | typeSpecifier [specCount , parent] )=>
        specCount = typeSpecifier [specCount , parent]
        | typeQualifier
        )+
;


structDeclaratorList [Node parent]
:       structDeclarator [parent] ( COMMA! structDeclarator [parent] )*
;


structDeclarator [Node parent]
:
(       COLON constExpr [parent]
        |       declarator [false, parent] ( COLON constExpr [parent] )?
)
{ ## = #( #[NStructDeclarator], ##); }
;


enumSpecifier [Node parent]
:       "enum"^
( ( ID LCURLY )=> i:ID LCURLY enumList [i.getText(), parent] RCURLY!
  | LCURLY enumList ["anonymous", parent] RCURLY!
  | ID
)
;


enumList [String enumName, Node parent]
:       enumerator [enumName, parent] ( COMMA! enumerator [enumName, parent] )*
;

enumerator [String enumName, Node parent]
:       i:ID                    { symbolTable.add( i.getText(),
#(   null,
#[LITERAL_enum, "enum"],
#[ ID, enumName]
 )
        );
}
(ASSIGN constExpr [parent])?
;


initDeclList [AST declarationSpecifiers, Node parent]
:       initDecl [declarationSpecifiers, parent]
( COMMA! initDecl [declarationSpecifiers, parent] )*
;


initDecl [AST declarationSpecifiers, Node parent]
{ String declName = ""; }
:       declName = d:declarator [false, parent]
{   AST ds1, d1;
    ds1 = astFactory.dupList (declarationSpecifiers);
    d1 = astFactory.dupList (#d);
    symbolTable.add (declName, #(null, ds1, d1) );
}
( ASSIGN initializer [parent]
  | COLON expr[parent]
)?
{ ## = #( #[NInitDecl], ## ); }

;

pointerGroup
:       ( STAR ( typeQualifier )* )+    { ## = #( #[NPointerGroup], ##); }
;



idList
:       ID ( COMMA! ID )*
;


initializer[Node parent]
:       ( assignExpr[parent]
        |       LCURLY initializerList[parent] ( COMMA! )? RCURLY!
        )
{ ## = #( #[NInitializer], ## ); }
;


initializerList[Node parent]
:       initializer[parent] ( COMMA! initializer[parent] )*
;


declarator[boolean isFunctionDefinition, Node parent] returns [String declName]
{ declName = ""; }
:
( pointerGroup )?

( id:ID                         { declName = id.getText(); }
  | LPAREN declName = declarator[false, parent] RPAREN
)

( !  LPAREN
  {
  if (isFunctionDefinition) {
  pushScope(declName);
  }
  else {
  pushScope("!"+declName);
  }
  }
  (
   (declSpecifiers[parent])=> p:parameterTypeList[parent]
   {
## = #( null, ##, #( #[NParameterTypeList], #p ) );
   }

   | (i:idList)?
   {
## = #( null, ##, #( #[NParameterTypeList], #i ) );
   }
  )
```

```
   {
   popScope();
   }
 RPAREN
 | LBRACKET ( constExpr[parent] )? RBRACKET
 )*
 { ## = #( #[NDeclarator], ## ); }
 ;

parameterTypeList[Node parent]
 :      parameterDeclaration[parent]
 (   options {
     warnWhenFollowAmbig = false;
     } :
     COMMA!
     parameterDeclaration[parent]
 )*
 ( COMMA!
   VARARGS
 )?
 ;


parameterDeclaration[Node parent]
{ String declName; }
 :      ds:declSpecifiers[parent]
 ( ( declarator[false, parent] )=> declName = d:declarator[false, parent]
   {
   AST d2, ds2;
   d2 = astFactory.dupList(#d);
   ds2 = astFactory.dupList(#ds);
   symbolTable.add(declName, #(null, ds2, d2));
   }
   | nonemptyAbstractDeclarator[parent]
 )?
 {
 ## = #( #[NParameterDeclaration], ## );
 }
 ;

/* JTC:
 * This handles both new and old style functions.
 * see declarator rule to see differences in parameters
 * and here (declaration SEMI)* is the param type decls for the
 * old style.  may want to do some checking to check for illegal
 * combinations (but I assume all parsed code will be legal?)
 */

functionDef[Node parent]
{ String declName;
    Node andNode = null;
}
:       ( (functionDeclSpecifiers[parent])=> ds:functionDeclSpecifiers[parent]
        |  //epsilon
        )
declName = d:declarator[true, parent]
{
    AST d2, ds2;
    d2 = astFactory.dupList(#d);
    ds2 = astFactory.dupList(#ds);
    symbolTable.add(declName, #(null, ds2, d2));
    pushScope(declName);
    //I add the code here
    andNode = new Node("AND");
    System.out.println(declName + " [shape=rectangle] ;");
    System.out.println(declName + " -> " + andNode + ";");
    parent = andNode;

}
( declaration[parent] )* (VARARGS)? ( SEMI! )*
{ popScope();
}
compoundStatement[declName, "function", parent]
{ ## = #( #[NFunctionDef], ## );}
;

functionDeclSpecifiers[Node parent]
{ int specCount = 0; }
:       (               options {   // this loop properly aborts when
        // it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :
        functionStorageClassSpecifier
        | typeQualifier
        | ( "struct" | "union" | "enum" | typeSpecifier[specCount, parent] )=>
        specCount = typeSpecifier[specCount, parent]
        )+
;

declarationList[Node parent]
:       (               options {   // this loop properly aborts when
        // it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :
        ( declarationPredictor[parent] )=> declaration[parent]
        )+
;

declarationPredictor[Node parent]
:       (options {      //only want to look at declaration if I don't see typedef
        warnWhenFollowAmbig = false;
        }:
        "typedef"
        | declaration[parent]
        )
;


compoundStatement[String scopeName, String type, Node parent]
:       LCURLY!
{
    pushScope(scopeName);
}
( ( declarationPredictor[parent])=> declarationList[parent] )?
( statementList[parent] )?
{ popScope(); }
RCURLY!
{ ## = #( #[NCompoundStatement, scopeName], ##);
```

```
}
;


statementList[Node parent]
:       ( statement[null, parent] )+
;
statement[String type, Node parent]
:       SEMI                    // Empty statements

|       compoundStatement[getAScopeName(), type, parent]        // Group of statements

|       expr[parent] SEMI!              { ## = #( #[NStatementExpr], ## ); } // Expressions

// Iteration statements:

|       "while"^ LPAREN! expr[parent] RPAREN! statement["while", parent]
|       "do"^ statement["do", parent] "while"! LPAREN! expr[parent] RPAREN! SEMI!
|!       "for"
LPAREN ( e1:expr[parent] )? SEMI ( e2:expr[parent] )? SEMI ( e3:expr[parent] )? RPAREN
s:statement["for", parent]
{
    if ( #e1 == null) { #e1 = #[ NEmptyExpression ]; }
    if ( #e2 == null) { #e2 = #[ NEmptyExpression ]; }
    if ( #e3 == null) { #e3 = #[ NEmptyExpression ]; }
## = #( #[LITERAL_for, "for"], #e1, #e2, #e3, #s );
}


// Jump statements:

|       "goto"^ ID SEMI!
|       "continue" SEMI!
|       "break" SEMI!
|       "return"^ ( expr[parent] )? SEMI!


// Labeled statements:
|       ID COLON! (options {warnWhenFollowAmbig=false;}:statement["label", parent])? { ## = #( #[NLabel
    ], ## ); }
|       "case"^ constExpr[parent] COLON! statement["case", parent]
|       "default"^ COLON! statement["default", parent]



// Selection statements:

|       "if"^
LPAREN! expr[parent] RPAREN! statement["if", parent]
( //standard if-else ambiguity
  options {
  warnWhenFollowAmbig = false;
  } :
  "else" statement["else", parent] )?
|       "switch"^ LPAREN! expr[parent] RPAREN! statement["switch", parent]
;
```

```
expr[Node parent]
:       assignExpr[parent] (options {
        /* MBZ:
           COMMA is ambiguous between comma expressions and
           argument lists.  argExprList should get priority,
           and it does by being deeper in the expr rule tree
           and using (COMMA assignExpr)*
         */
        warnWhenFollowAmbig = false;
        } :
c:COMMA^ { #c.setType(NCommaExpr); } assignExpr[parent]
        )*
;


assignExpr[Node parent]
:       conditionalExpr[parent] ( a:assignOperator! assignExpr[parent] { ## = #( #a, ## );} )?
;

assignOperator
:       ASSIGN
|       DIV_ASSIGN
|       PLUS_ASSIGN
|       MINUS_ASSIGN
|       STAR_ASSIGN
|       MOD_ASSIGN
|       RSHIFT_ASSIGN
|       LSHIFT_ASSIGN
|       BAND_ASSIGN
|       BOR_ASSIGN
|       BXOR_ASSIGN
;


conditionalExpr[Node parent]
:       logicalOrExpr[parent]
( QUESTION^ expr[parent] COLON! conditionalExpr[parent] )?
;


constExpr[Node parent]
:       conditionalExpr[parent]
;

logicalOrExpr[Node parent]
:       logicalAndExpr[parent] ( LOR^ logicalAndExpr[parent] )*
;


logicalAndExpr[Node parent]
:       inclusiveOrExpr[parent] ( LAND^ inclusiveOrExpr[parent] )*
;

inclusiveOrExpr[Node parent]
:       exclusiveOrExpr[parent] ( BOR^ exclusiveOrExpr[parent] )*
;
```

```
exclusiveOrExpr[Node parent]
:       bitAndExpr[parent] ( BXOR^ bitAndExpr[parent] )*
;


bitAndExpr[Node parent]
:       equalityExpr[parent] ( BAND^ equalityExpr[parent] )*
;



equalityExpr[Node parent]
:       relationalExpr[parent]
( ( EQUAL^ | NOT_EQUAL^ ) relationalExpr[parent] )*
;


relationalExpr[Node parent]
:       shiftExpr[parent]
( ( LT^ | LTE^ | GT^ | GTE^ ) shiftExpr[parent] )*
;



shiftExpr[Node parent]
:       additiveExpr[parent]
( ( LSHIFT^ | RSHIFT^ ) additiveExpr[parent] )*
;


additiveExpr[Node parent]
:       multExpr[parent]
( ( PLUS^ | MINUS^ ) multExpr[parent] )*
;


multExpr[Node parent]
:       castExpr[parent]
( ( STAR^ | DIV^ | MOD^ ) castExpr[parent] )*
;


castExpr[Node parent]
    :       ( LPAREN typeName[parent] RPAREN )=>
LPAREN! typeName[parent] RPAREN! ( castExpr[parent] )
{ ## = #( #[NCast, "("], ## ); }

|       unaryExpr[parent]
;


typeName[Node parent]
:       specifierQualifierList[parent] (nonemptyAbstractDeclarator[parent])?
;


nonemptyAbstractDeclarator[Node parent]
:   (
        pointerGroup
        (   (LPAREN
              (   nonemptyAbstractDeclarator[parent]
                | parameterTypeList[parent]
              )?
              RPAREN)
            | (LBRACKET (expr[parent])? RBRACKET)
        )*

        |   (   (LPAREN
                  (   nonemptyAbstractDeclarator[parent]
                    | parameterTypeList[parent]
                  )?
                  RPAREN)
            | (LBRACKET (expr[parent])? RBRACKET)
            )+
     )
{   ## = #( #[NNonemptyAbstractDeclarator], ## ); }

;

/* JTC:

   LR rules:

   abstractDeclarator
   :        nonemptyAbstractDeclarator
   |        // null
   ;

   nonemptyAbstractDeclarator
   :        LPAREN  nonemptyAbstractDeclarator RPAREN
   |        abstractDeclarator LPAREN RPAREN
   |        abstractDeclarator (LBRACKET (expr)? RBRACKET)
   |        STAR abstractDeclarator
   ;
 */

unaryExpr[Node parent]
:       postfixExpr[parent]
|       INC^ unaryExpr[parent]
|       DEC^ unaryExpr[parent]
|       u:unaryOperator castExpr[parent] { ## = #( #[NUnaryExpr], ## ); }

|       "sizeof"^
( ( LPAREN typeName[parent] )=> LPAREN typeName[parent] RPAREN
  | unaryExpr[parent]
)
;


unaryOperator
:       BAND
|       STAR
|       PLUS
|       MINUS
|       BNOT
|       LNOT
;
```

```
postfixExpr[Node parent]{String idName="";}
:       idName = f:primaryExpr[parent]
(
 postfixSuffix[parent, idName]                    {## = #( #[NPostfixExpr], ## );}
)?
;
postfixSuffix[Node parent, String idName]
:
//{System.out.println("STD: POSTFIXSUFFIX");}
( PTR ID
  | DOT ID
  | functionCall[parent, idName]
  | LBRACKET expr[parent] RBRACKET
  | INC
  | DEC
)+
;

functionCall[Node parent, String idName]
:
//{System.out.println("STD: FUNCTION CALL");}
LPAREN^ (a:argExprList[parent])? RPAREN
{
##.setType( NFunctionCallArgs );
    System.out.println(parent + " -> " + idName + ";");
}
;


primaryExpr[Node parent] returns [String idName]{idName = "";}
:       id:ID {idName = id.getText();}
|       charConst
|       intConst
|       floatConst
|       stringConst

// JTC:
// ID should catch the enumerator
// leaving it in gives ambiguous err
//      | enumerator
|       LPAREN! expr[parent] RPAREN!        { ## = #( #[NExpressionGroup, "("], ## ); }
;

argExprList[Node parent]
:       assignExpr[parent] ( COMMA! assignExpr[parent] )*
;



protected
charConst
:       CharLiteral
;


protected
stringConst
:       (StringLiteral)+                { ## = #(#[NStringSeq], ##); }
```

```
;


protected
intConst
:       IntOctalConst
|       LongOctalConst
|       UnsignedOctalConst
|       IntIntConst
|       LongIntConst
|       UnsignedIntConst
|       IntHexConst
|       LongHexConst
|       UnsignedHexConst
;


protected
floatConst
:       FloatDoubleConst
|       DoubleDoubleConst
|       LongDoubleConst
;




dummy
:       NTypedefName
|       NInitDecl
|       NDeclarator
|       NStructDeclarator
|       NDeclaration
|       NCast
|       NPointerGroup
|       NExpressionGroup
|       NFunctionCallArgs
|       NNonemptyAbstractDeclarator
|       NInitializer
|       NStatementExpr
|       NEmptyExpression
|       NParameterTypeList
|       NFunctionDef
|       NCompoundStatement
|       NParameterDeclaration
|       NCommaExpr
|       NUnaryExpr
|       NLabel
|       NPostfixExpr
|       NRangeExpr
|       NStringSeq
|       NInitializerElementLabel
|       NLcurlyInitializer
|       NAsmAttribute
|       NGnuAsmExpr
|       NTypeMissing
;
```

```
{
    //import CToken;
    import java.io.*;
    //import LineObject;
    import antlr.*;
}

class StdCLexer extends Lexer;

options
{
    k = 3;
    exportVocab = STDC;
    testLiterals = false;
}

{
    LineObject lineObject = new LineObject();
    String originalSource = "";
    PreprocessorInfoChannel preprocessorInfoChannel = new PreprocessorInfoChannel();
    int tokenNumber = 0;
    boolean countingTokens = true;
    int deferredLineCount = 0;

    public void setCountingTokens(boolean ct)
    {
        countingTokens = ct;
        if ( countingTokens ) {
            tokenNumber = 0;
        }
        else {
            tokenNumber = 1;
        }
    }

    public void setOriginalSource(String src)
    {
        originalSource = src;
        lineObject.setSource(src);
    }
    public void setSource(String src)
    {
        lineObject.setSource(src);
    }

    public PreprocessorInfoChannel getPreprocessorInfoChannel()
    {
        return preprocessorInfoChannel;
    }

    public void setPreprocessingDirective(String pre)
    {
        preprocessorInfoChannel.addLineForTokenNumber( pre, new Integer(tokenNumber) );
    }

    protected Token makeToken(int t)
    {
        if ( t != Token.SKIP && countingTokens) {
            tokenNumber++;
        }
        CToken tok = (CToken) super.makeToken(t);
        tok.setLine(lineObject.line);
        tok.setSource(lineObject.source);
        tok.setTokenNumber(tokenNumber);

        lineObject.line += deferredLineCount;
        deferredLineCount = 0;
        return tok;
    }

    public void deferredNewline() {
        deferredLineCount++;
    }

    public void newline() {
        lineObject.newline();
    }




}

protected
Vocabulary
:       '\3'..'\377'
;


/* Operators: */

ASSIGN          : '=' ;
COLON           : ':' ;
COMMA           : ',' ;
QUESTION        : '?' ;
SEMI            : ';' ;
PTR             : "->" ;


// DOT & VARARGS are commented out since they are generated as part of
// the Number rule below due to some bizarre lexical ambiguity shme.

// DOT  :       '.' ;
protected
DOT:;

// VARARGS      : "..." ;
protected
VARARGS:;
```

```
LPAREN          : '(' ;
RPAREN          : ')' ;
LBRACKET        : '[' ;
RBRACKET        : ']' ;
LCURLY          : '{' ;
RCURLY          : '}' ;

EQUAL           : "==" ;
NOT_EQUAL       : "!=" ;
LTE             : "<=" ;
LT              : "<" ;
GTE             : ">=" ;
GT              : ">" ;

DIV             : '/' ;
DIV_ASSIGN      : "/=" ;
PLUS            : '+' ;
PLUS_ASSIGN     : "+=" ;
INC             : "++" ;
MINUS           : '-' ;
MINUS_ASSIGN    : "-=" ;
DEC             : "--" ;
STAR            : '*' ;
STAR_ASSIGN     : "*=" ;
MOD             : '%' ;
MOD_ASSIGN      : "%=" ;
RSHIFT          : ">>" ;
RSHIFT_ASSIGN   : ">>=" ;
LSHIFT          : "<<" ;
LSHIFT_ASSIGN   : "<<=" ;

LAND            : "&&" ;
LNOT            : '!' ;
LOR             : "||" ;

BAND            : '&' ;
BAND_ASSIGN     : "&=" ;
BNOT            : '~' ;
BOR             : '|' ;
BOR_ASSIGN      : "|=" ;
BXOR            : '^' ;
BXOR_ASSIGN     : "^=" ;


Whitespace
:       ( ( '\003'..'\010' | '\t' | '\013' | '\f' | '\016'.. '\037' | '\177'..'\377' | ' ' )
        | "\r\n"                { newline(); }
        | ( '\n' | '\r' )       { newline(); }
        )                       { _ttype = Token.SKIP;  }
;


Comment
:       "/*"
( { LA(2) != '/' }? '*'
  | "\r\n"                { deferredNewline(); }
  | ( '\r' | '\n' )       { deferredNewline();    }
  | ~( '*'| '\r' | '\n' )
)*
"*/"                    { _ttype = Token.SKIP;
}
;



CPPComment
:
"//" ( ~('\n') )*
{
    _ttype = Token.SKIP;
}
;

PREPROC_DIRECTIVE
options {
    paraphrase = "a line directive";
}

:
'#'
( ( "line" || (( ' ' | '\t' | '\014')+ '0'..'9')) => LineDirective
  | (~'\n')*                              { setPreprocessingDirective(getText()); }
)
{
    _ttype = Token.SKIP;
}
;

protected  Space:
( ' ' | '\t' | '\014')
;

protected LineDirective
{
    boolean oldCountingTokens = countingTokens;
    countingTokens = false;
}
:
{
    lineObject = new LineObject();
    deferredLineCount = 0;
}
("line")?  //this would be for if the directive started "#line", but not there for GNU directives
(Space)+
n:Number { lineObject.setLine(Integer.parseInt(n.getText())); }
(Space)+
(       fn:StringLiteral {  try {
        lineObject.setSource(fn.getText().substring(1,fn.getText().length()-1));
        }
        catch (StringIndexOutOfBoundsException e) { /*not possible*/ }
        }
        | fi:ID { lineObject.setSource(fi.getText()); }
)?
(Space)*
("1"            { lineObject.setEnteringFile(true); } )?
(Space)*
("2"            { lineObject.setReturningToFile(true); } )?
(Space)*
```

```
("3"            { lineObject.setSystemHeader(true); } )?
(Space)*
("4"            { lineObject.setTreatAsC(true); } )?
(~('\r' | '\n'))*
("\r\n" | "\r" | "\n")
{
    preprocessorInfoChannel.addLineForTokenNumber(new LineObject(lineObject), new Integer(tokenNumber));
    countingTokens = oldCountingTokens;
}
;



/* Literals: */

/*
 * Note that we do NOT handle tri-graphs nor multi-byte sequences.
 */


/*
 * Note that we can't have empty character constants (even though we
 * can have empty strings :-).
 */
CharLiteral
:       '\'' ( Escape | ~( '\'' ) ) '\''
;



/*
 * Can't have raw imbedded newlines in string constants.  Strict reading of
 * the standard gives odd dichotomy between newlines & carriage returns.
 * Go figure.
 */
StringLiteral
:       '"'
( Escape
  | (
      '\r'        { deferredNewline(); }
      | '\n'        {
      deferredNewline();
      _ttype = BadStringLiteral;
      }
      | '\\' '\n'   {
      deferredNewline();
      }
    )
  | ~( '"' | '\r' | '\n' | '\\' )
)*
'"'
;


protected BadStringLiteral
:       // Imaginary token.
;



/*
```

```
 * Handle the various escape sequences.
 *
 * Note carefully that these numeric escape *sequences* are *not* of the
 * same form as the C language numeric *constants*.
 *
 * There is no such thing as a binary numeric escape sequence.
 *
 * Octal escape sequences are either 1, 2, or 3 octal digits exactly.
 *
 * There is no such thing as a decimal escape sequence.
 *
 * Hexadecimal escape sequences are begun with a leading \x and continue
 * until a non-hexadecimal character is found.
 *
 * No real handling of tri-graph sequences, yet.
 */

protected
Escape
:       '\\'
( options{warnWhenFollowAmbig=false;}:
  'a'
  | 'b'
  | 'f'
  | 'n'
  | 'r'
  | 't'
  | 'v'
  | '"'
  | '\''
  | '\\'
  | '?'
  | ('0'..'3') ( options{warnWhenFollowAmbig=false;}: Digit ( options{warnWhenFollowAmbig=false;}: Digit
        )? )?
  | ('4'..'7') ( options{warnWhenFollowAmbig=false;}: Digit )?
  | 'x' ( options{warnWhenFollowAmbig=false;}: Digit | 'a'..'f' | 'A'..'F' )+
)
;


/* Numeric Constants: */

protected
Digit
:       '0'..'9'
;

protected
LongSuffix
:       'l'
|       'L'
;

protected
UnsignedSuffix
:       'u'
|       'U'
;
```

```
protected
FloatSuffix
:       'f'
|       'F'
;

protected
Exponent
:       ( 'e' | 'E' ) ( '+' | '-' )? ( Digit )+
;


protected
DoubleDoubleConst :;

protected
FloatDoubleConst :;

protected
LongDoubleConst :;

protected
IntOctalConst :;

protected
LongOctalConst :;

protected
UnsignedOctalConst :;

protected
IntIntConst :;

protected
LongIntConst :;

protected
UnsignedIntConst :;

protected
IntHexConst :;

protected
LongHexConst :;

protected
UnsignedHexConst :;



Number
:       ( ( Digit )+ ( '.' | 'e' | 'E' ) )=> ( Digit )+
( '.' ( Digit )* ( Exponent )?
  | Exponent
)                           { _ttype = DoubleDoubleConst;   }
( FloatSuffix               { _ttype = FloatDoubleConst;    }
  | LongSuffix                { _ttype = LongDoubleConst;     }
)?
```

```
|       ( "..." )=> "..."       { _ttype = VARARGS;     }

|       '.'                     { _ttype = DOT; }
( ( Digit )+ ( Exponent )?
  { _ttype = DoubleDoubleConst;   }
  ( FloatSuffix          { _ttype = FloatDoubleConst;    }
    | LongSuffix           { _ttype = LongDoubleConst;     }
  )?
)?

|       '0' ( '0'..'7' )*       { _ttype = IntOctalConst;       }
( LongSuffix             { _ttype = LongOctalConst;      }
  | UnsignedSuffix         { _ttype = UnsignedOctalConst;  }
)?

|       '1'..'9' ( Digit )*     { _ttype = IntIntConst;         }
( LongSuffix             { _ttype = LongIntConst;        }
  | UnsignedSuffix         { _ttype = UnsignedIntConst;    }
)?

|       '0' ( 'x' | 'X' ) ( 'a'..'f' | 'A'..'F' | Digit )+
{ _ttype = IntHexConst;         }
( LongSuffix             { _ttype = LongHexConst;        }
  | UnsignedSuffix         { _ttype = UnsignedHexConst;    }
)?
;


ID
options
{
    testLiterals = true;
}
:       ( 'a'..'z' | 'A'..'Z' | '_' )
( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' )*
;
```

Listing B.5: GnuCParser.g

```
/*
 * GnuCParser.g - Grammar file for Gnu-C Grammar
 *              - Modified for building Branch-Preserving Call Graph
 *
 * AUTHOR: KK Lo, 7/2007
 *
 * REVISION HISTORY:
 *      Date            Name            Description
 *      11.07.2007      KK Lo           Initial Release
 */

/*
 * %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 *
 * Copyright (c) Non, Inc. 1998 -- All Rights Reserved
 *
 * PROJECT:         C Compiler MODULE:          GnuCParser FILE:
 * GnuCParser.g
 *
 * AUTHOR:          Monty Zukowski (jamz@cdsnet.net) April 28, 1998
 *
 * DESCRIPTION: This is a grammar for the GNU C compiler.  It is a grammar
 * subclass of StdCParser, overriding only those rules which are different
 * from Standard C.
 *
 * %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 */


{
    import          java.io.*;
    import          java.util.*;

    import          antlr.CommonAST;
    import          antlr.DumpASTVisitor;
}


class GnuCParser extends StdCParser;

options
{
    k = 2;
    exportVocab = GNUC;
    buildAST = true;
    ASTLabelType = "TNode";

    //Copied following options from java grammar.
    codeGenMakeSwitchThreshold = 2;
    codeGenBitsetTestThreshold = 3;
}


{
    //Suppport C++ - style single - line comments ?
    public static boolean CPPComments = true;

    //access to symbol table
    public CSymbolTable symbolTable = new CSymbolTable();

    //source for names
    to unnamed scopes
        protected int   unnamedScopeCounter = 0;

    public boolean  isTypedefName(String name) {
        boolean         returnValue = false;
        TNode           node = symbolTable.lookupNameInCurrentScope(name);
        for             (; node != null; node = (TNode) node.getNextSibling()) {
            if (node.getType() == LITERAL_typedef) {
                returnValue = true;
                break;
            }
        }
        return returnValue;
    }


    public String   getAScopeName() {
        return "" + (unnamedScopeCounter++);
    }

    public void     pushScope(String scopeName) {
        symbolTable.pushScope(scopeName);
    }

    public void     popScope() {
        symbolTable.popScope();
    }

    int             traceDepth = 0;
    public void     reportError(RecognitionException ex) {
        try {
            System.err.println("ANTLR Parsing Error: " + ex + " token name:" + tokenNames[LA(1)]);
            ex.printStackTrace(System.err);
        }
        catch(TokenStreamException e) {
            System.err.println("ANTLR Parsing Error: " + ex);
            ex.printStackTrace(System.err);
        }
    }
    public void     reportError(String s) {
        System.err.println("ANTLR Parsing Error from String: " + s);
    }
    public void     reportWarning(String s) {
        System.err.println("ANTLR Parsing Warning from String: " + s);
    }
    public void     match(int t) throws MismatchedTokenException {
        boolean         debugging = false;

        if              (debugging) {
            for (int x = 0; x < traceDepth; x++)
                System.out.print(" ");
            try {
                System.out.println("Match(" + tokenNames[t] + ") with LA(1)=" +
                        tokenNames[LA(1)] + ((inputState.guessing > 0) ? " [inputState.guessing " +
                            inputState.guessing + "]" : ""));
            }
            catch(TokenStreamException e) {
                System.out.println("Match(" + tokenNames[t] + ") " + ((inputState.guessing > 0) ? " [
                        inputState.guessing " + inputState.guessing + "]" : ""));

            }

        }
        try {
            if (LA(1) != t) {
                if (debugging) {
                    for (int x = 0; x < traceDepth; x++)
                        System.out.print(" ");
                    System.out.println("token mismatch: " + tokenNames[LA(1)]
                            + "!=" + tokenNames[t]);
                }
                throw new       MismatchedTokenException(tokenNames, LT(1), t, false, getFilename());

            } else {
                //mark token as consumed -- fetch next token deferred until LA / LT
                consume();
                }
            }
            catch(TokenStreamException e) {
            }

        }
        public void     traceIn(String rname) {
            traceDepth += 1;
            for (int x = 0; x < traceDepth; x++)
                System.out.print(" ");
            try {
                System.out.println("> " + rname + "; LA(1)==(" + tokenNames[LT(1).getType()]
                        + ") " + LT(1).getText() + " [inputState.guessing " + inputState.guessing + "]");
            }
            catch(TokenStreamException e) {
            }
        }
        public void     traceOut(String rname) {
            for (int x = 0; x < traceDepth; x++)
                System.out.print(" ");
            try {
                System.out.println("< " + rname + "; LA(1)==(" + tokenNames[LT(1).getType()]
                        + ") " + LT(1).getText() + " [inputState.guessing " + inputState.guessing + "]");
            }
            catch(TokenStreamException e) {
            }
            traceDepth -= 1;
        }

}


translationUnit[Node parent]
:(externalList[parent]) ? /* Empty source files are allowed.  */
;
asm_expr[Node parent]
:"asm" ^
("volatile") ? LCURLY expr[parent] RCURLY(SEMI) +
;

idList
:ID(options
        {
        warnWhenFollowAmbig = false;
        }:COMMA ID) *
;

externalDef[Node parent]
:("typedef" | declaration[parent]) = >declaration[parent]
| (functionPrefix[parent]) = >functionDef[parent]
| typelessDeclaration[parent]
| asm_expr[parent]
| SEMI
;

/* these two are here because GCC allows "cat = 13;" as a valid program! */
functionPrefix[Node parent] {
    String          declName;
}
```

```
 : ((functionDeclSpecifiers[parent]) = >ds:functionDeclSpecifiers[parent]
         | //epsilon
   )
 declName = d:declarator[true, parent]
 (declaration[parent]) * (VARARGS) ? (SEMI) *
 LCURLY
 ;

 typelessDeclaration[Node parent] {
     AST              typeMissing =
 #[NTypeMissing]; }
     :initDeclList[typeMissing, parent] SEMI {
 ## = #( #[NTypeMissing], ##); }
     ;

     initializer[Node parent]
     :((((initializerElementLabel[parent]) = >initializerElementLabel[parent]) ?
               (assignExpr[parent] | lcurlyInitializer[parent]) {
 ## = #( #[NInitializer], ## ); }
       )
           | lcurlyInitializer[parent]
      );

     //GCC allows more specific initializers
     initializerElementLabel[Node parent]
 :((LBRACKET((constExpr[parent] VARARGS) = >rangeExpr[parent] | constExpr[parent]) RBRACKET(ASSIGN) ?)
         | ID COLON
         | DOT ID ASSIGN
  ) {
 ## = #( #[NInitializerElementLabel], ##) ; }
 ;

 //GCC allows empty initializer lists
 lcurlyInitializer[Node parent]
 :
 LCURLY ^ (initializerList[parent] (COMMA !) ?) ? RCURLY
 {
 ##.setType( NLcurlyInitializer ); }
 ;

 initializerList[Node parent]
 :initializer[parent] (options {
         warnWhenFollowAmbig = false;
         }:             COMMA ! initializer[parent]) *
 ;


 declarator[boolean isFunctionDefinition, Node parent] returns[String declName] {
     declName = "";
 }
 :
 (pointerGroup) ?

 (id : ID {
  declName = id.getText();
  }
  |LPAREN declName = declarator[false, parent] RPAREN
 )
 (declaratorParamaterList[isFunctionDefinition, declName, parent]
  | LBRACKET(expr[parent]) ? RBRACKET
 ) *
 {
 ## = #( #[NDeclarator], ## ); }
 ;

 declaratorParamaterList[boolean isFunctionDefinition, String declName, Node parent]
 :
 LPAREN ^
 {
     if (isFunctionDefinition) {
         pushScope(declName);
     } else {
         pushScope("!" + declName);
     }
 }
 (
  (declSpecifiers[parent]) = >parameterTypeList[parent]
  | (idList) ?
 ) {
     popScope();
 }
 (COMMA !) ?
 RPAREN
 {
 ##.setType(NParameterTypeList); }
 ;

parameterTypeList[Node parent]
:     parameterDeclaration[parent]
(options {
 warnWhenFollowAmbig = false;
 }:
 (COMMA | SEMI)
 parameterDeclaration[parent]
) *
((COMMA | SEMI)
 VARARGS
) ?
;


declarationList[Node parent]
:     (options {
        //this loop properly aborts when
        // it finds a non - typedefName ID MBZ
        warnWhenFollowAmbig = false;
        }:

        localLabelDeclaration
        | (declarationPredictor[parent]) = >declaration[parent]
        ) +
;
localLabelDeclaration
: (//GNU note:    any __label__ declarations must come before regular declarations.
        "__label__" ^ ID(options {
            warnWhenFollowAmbig = false;
            }:    COMMA ! ID) * (COMMA !) ? (SEMI !) +
  );
```

```
declaration[Node parent] {
    AST              ds1 = null;
}
: ds:   declSpecifiers[parent] {
    ds1 = astFactory.dupList(
#ds); }
    (
     initDeclList[ds1, parent]
    ) ?
    (SEMI) +
{
## = #( #[NDeclaration], ##); }

;

functionStorageClassSpecifier
:     "extern"
| "static"
| "inline"
;

typeSpecifier[int specCount, Node parent] returns[int retSpecCount] {
    retSpecCount = specCount + 1;
}
:
("void"
 | "char"
 | "short"
 | "int"
 | "long"
 | "float"
 | "double"
 | "signed"
 | "unsigned"
 | structOrUnionSpecifier[parent] (options {
     warnWhenFollowAmbig = false;
     }:   attributeDecl) *
 |enumSpecifier[parent]
 | {
 specCount == 0
 } ? typedefName
 | "typeof" ^ LPAREN
 ((typeName[parent]) = >typeName[parent]
  | expr[parent]
 )
 RPAREN
 | "__complex"
);


structOrUnionSpecifier[Node parent] {
    String           scopeName;
}
: sou:    structOrUnion !
((ID LCURLY) = >i: ID l:LCURLY
 {
 scopeName =
#sou.getText() + " " + #i.getText();
#l.setText(scopeName);
 pushScope(scopeName);
 }
 (structDeclarationList[parent]) ?
 {
 popScope();
 }
 RCURLY
 | l1:      LCURLY
 {
 scopeName = getAScopeName();
#l1.setText(scopeName);
 pushScope(scopeName);
 }
 (structDeclarationList[parent]) ?
 {
 popScope();
 }
RCURLY
| ID
) {
## = #( #sou, ## );
}
;


structDeclaration[Node parent]
:      specifierQualifierList[parent] structDeclaratorList[parent] (COMMA !) ? (SEMI !) +
;

structDeclaratorList[Node parent]
:      structDeclarator[parent] (options {
         warnWhenFollowAmbig = false;
         }:     COMMA ! structDeclarator[parent]) *
;

structDeclarator[Node parent]
:      (declarator[false, parent]) ?
(COLON constExpr[parent]) ?
(attributeDecl) *
{
## = #( #[NStructDeclarator], ##); }
;



enumSpecifier[Node parent]
:      "enum" ^
((ID LCURLY) = >i:  ID LCURLY enumList[i.getText(), parent] RCURLY
 | LCURLY enumList["anonymous", parent] RCURLY
 | ID
);
enumList[String enumName, Node parent]
:      enumerator[enumName, parent] (options {
         warnWhenFollowAmbig = false;
         }:     COMMA ! enumerator[enumName, parent]) * (COMMA !) ?
;
```

```
initDeclList[AST declarationSpecifiers, Node parent]
:       initDecl[declarationSpecifiers, parent]
(options {
 warnWhenFollowAmbig = false;
 }:    COMMA ! initDecl[declarationSpecifiers, parent]) *
(COMMA !) ?
;

initDecl[AST declarationSpecifiers, Node parent] {
    String          declName = "";
}
: declName = d:declarator[false, parent] {
    AST             ds1,
    d1;
    ds1 = astFactory.dupList(declarationSpecifiers);
    d1 = astFactory.dupList(
#d);
        symbolTable.add(declName,
#(null, ds1, d1) );
}
(attributeDecl) *
(ASSIGN initializer[parent]
 | COLON expr[parent]
) ?
{
## = #( #[NInitDecl], ## ); }
;

attributeDecl
:       "__attribute" ^ LPAREN LPAREN attributeList RPAREN RPAREN
| "asm" ^ LPAREN stringConst RPAREN {
##.setType( NAsmAttribute ); }
;

attributeList
:       attribute(options {
          warnWhenFollowAmbig = false;
          }:    COMMA attribute) * (COMMA) ?
;

      attribute
:       (~(LPAREN | RPAREN | COMMA)
          | LPAREN attributeList RPAREN
          ) *
;
compoundStatement[String scopeName, String type, Node parent]
:       LCURLY ^

{
    pushScope(scopeName);
}
(//this ambiguity is ok, declarationList and nestedFunctionDef end properly
 options {
 warnWhenFollowAmbig = false;
 }:
 ("typedef" | "__label__" | declaration[parent]) = >declarationList[parent]
 | (nestedFunctionDef[parent]) = >nestedFunctionDef[parent]
) *
```

```
 (statementList[parent]) ?
 {
     popScope();
 }
 RCURLY
 {
 ##.setType( NCompoundStatement ); ##.setAttribute( "scopeName", scopeName );
 }
 ;

 nestedFunctionDef[Node parent] {
     String          declName;
 }
 :   ("auto") ? //only for nested
 functions
 ((functionDeclSpecifiers[parent]) = >ds : functionDeclSpecifiers[parent]
 ) ?
 declName = d : declarator[false, parent] {
     AST             d2,
     ds2;
     d2 = astFactory.dupList(
 #d);
         ds2 = astFactory.dupList(
 #ds);
         symbolTable.add(declName,
 #(null, ds2, d2));
     pushScope(declName);
 }
 (declaration[parent]) *
 {
     popScope();
 }
 compoundStatement[declName, "function", new Node(declName)] {
 ## = #( #[NFunctionDef], ## );}
 ;

 statement[String type, Node parent]
 //////////////////////////////////
 {
     Node            andNode1 = null;
     Node            andNode2 = null;
     Node            orNode = null;
 }
 /////////////////////////////////

 :     SEMI // Empty statements

 | compoundStatement[getAScopeName(), type, parent] // Group of statements

 | expr[parent] SEMI ! {
 ## = #( #[NStatementExpr], ## );} // Expressions

 //Iteration statements:

 |"while" ^ LPAREN ! expr[parent] RPAREN !
 {
     andNode1 = new Node("AND");
     System.out.println(parent + " -> " + andNode1 + ";");
 }
```

```
statement["while", andNode1]
| "do" ^
{
    andNode1 = new Node("AND");
    System.out.println(parent + " -> " + andNode1 + ";");
}
statement["do", andNode1] "while" ! LPAREN ! expr[parent] RPAREN ! SEMI !
|!"for"
LPAREN(e1:     expr[parent]) ? SEMI(e2 : expr[parent]) ? SEMI(e3 : expr[parent]) ? RPAREN
{
    andNode1 = new Node("AND");
    System.out.println(parent + " -> " + andNode1 + ";");
}
s:    statement["for", andNode1] {
                    if (
#e1 == null) { #e1 = #[ NEmptyExpression ]; }
                        if (
#e2 == null) { #e2 = #[ NEmptyExpression ]; }
                          if (
#e3 == null) { #e3 = #[ NEmptyExpression ]; }
## = #( #[LITERAL_for , "for"], #e1, #e2, #e3, #s );
                }


//Jump statements:

|"goto" ^ expr[parent] SEMI !
|"continue" SEMI !
|"break" SEMI !
|"return" ^ (expr[parent]) ? SEMI !


|ID COLON ! (options {
        warnWhenFollowAmbig = false;
        }:    statement["label", parent]) ? {
## = #( #[NLabel], ## ); }
//GNU allows range expressions in case statements
| "case" ^
{
    andNode1 = new Node("AND");
    System.out.println(parent + " -> " + andNode1 + ";");
}
((constExpr[andNode1] VARARGS) = >rangeExpr[andNode1] | constExpr[andNode1]) COLON ! (options {
                                                                                      warnWhenFollowAmbig
                                                                                          = false;
                                                                                      }:    statement["
                                                                                          case",
                                                                                          andNode1]) ?
|"default" ^
{
    andNode1 = new Node("AND");
    System.out.println(parent + " -> " + andNode1 + ";");
}
COLON ! (options {
        warnWhenFollowAmbig = false;
        }:    statement["default", andNode1]) ?

//Selection statements :
|
"if" ^
LPAREN ! expr[parent] RPAREN ! {
    orNode = new Node("OR");
    System.out.println(parent + " -> " + orNode + ";");
    //There are 2 branches for each
    if -
    else
        statement
            andNode1 = new Node("AND");
    andNode2 = new Node("AND");
}
statement["if", andNode1] {
    //If block
    System.out.println(orNode + " -> " + andNode1 + ";");
}
(//standard if -
 else
 ambiguity
 options {
 warnWhenFollowAmbig = false;
 }:
 "else" statement["else", andNode2] {
 //
 else
 block
 System.out.println(orNode + " -> " + andNode2 + ";");
 }
) ?
|"switch" ^ LPAREN ! expr[parent] {
    orNode = new Node("OR");
    System.out.println(parent + " -> " + orNode + ";");
}
RPAREN ! statement["switch", orNode];



conditionalExpr[Node parent]
:     logicalOrExpr[parent]
(QUESTION ^ (expr[parent]) ? COLON conditionalExpr[parent]) ?
;

rangeExpr[Node parent] // used in initializers only
:     constExpr[parent] VARARGS constExpr[parent] {
## = #(#[NRangeExpr], ##); }
;

castExpr[Node parent]
:(LPAREN typeName[parent] RPAREN) = >
LPAREN ^ typeName[parent] RPAREN(castExpr[parent] | lcurlyInitializer[parent]) {
##.setType(NCast); }

|unaryExpr[parent];
nonemptyAbstractDeclarator[Node parent]
:(
        pointerGroup
        ((LPAREN
          (nonemptyAbstractDeclarator[parent]
           | parameterTypeList[parent]
          ) ?
```

```
            (COMMA !) ?
            RPAREN)
           | (LBRACKET(expr[parent]) ? RBRACKET)
          ) *

          |((LPAREN
                  (nonemptyAbstractDeclarator[parent]
                   | parameterTypeList[parent]
                  ) ?
                  (COMMA !) ?
                  RPAREN)
              | (LBRACKET(expr[parent]) ? RBRACKET)
           ) +
 ) {
## = #( #[NNonemptyAbstractDeclarator], ## ); }

;



unaryExpr[Node parent]
:postfixExpr[parent]
| INC ^ castExpr[parent]
| DEC ^ castExpr[parent]
| u:unaryOperator castExpr[parent] {
## = #( #[NUnaryExpr], ## ); }

|"sizeof" ^
((LPAREN typeName[parent]) = >LPAREN typeName[parent] RPAREN
 | unaryExpr[parent]
)
| "__alignof" ^
((LPAREN typeName[parent]) = >LPAREN typeName[parent] RPAREN
 | unaryExpr[parent]
)
| gnuAsmExpr[parent];

unaryOperator
:BAND
| STAR
| PLUS
| MINUS
| BNOT // also stands for complex conjugation
| LNOT
| LAND // for label dereference(&&label)
|"__real"
| "__imag"
;

gnuAsmExpr[Node parent]
:"asm" ^ ("volatile") ?
LPAREN stringConst
(options {
 warnWhenFollowAmbig = false;
 }:
 COLON(strOptExprPair[parent] (COMMA strOptExprPair[parent]) *) ?
 (options {
  warnWhenFollowAmbig = false;
  }:
   COLON(strOptExprPair[parent] (COMMA strOptExprPair[parent]) *) ?
  ) ?
 ) ?
 (COLON stringConst(COMMA stringConst) *) ?
 RPAREN
 {
 ##.setType(NGnuAsmExpr); }
 ;

 //GCC requires the PARENs
 strOptExprPair[Node parent]
 :      stringConst(LPAREN expr[parent] RPAREN) ?
 ;


 primaryExpr[Node parent] returns[String idName] {
      idName = "";
 }
 : id:   ID {
      idName = id.getText();
 }
 |Number
 | charConst
 | stringConst
 // JTC:
 //ID should catch the enumerator
 // leaving it in gives ambiguous err
 // |enumerator
 | (LPAREN LCURLY) = >LPAREN ^ compoundStatement[getAScopeName(), "scope", parent] RPAREN
 | LPAREN ^ expr[parent] RPAREN {
 ##.setType(NExpressionGroup); }
 ;



 {
      //import CToken;
      import          java.io.*;
      //import LineObject;
      import          antlr.*;
 }

 class GnuCLexer extends StdCLexer;
 options
 {
      k = 3;
      importVocab = GNUC;
      testLiterals = false;
 }
 tokens {
      LITERAL___extension__ = "__extension__";
 }

 {
      public void     initialize(String src) {
          setOriginalSource(src);
          initialize();
      }

      public void     initialize() {
```

```
    literals.put(new ANTLRHashString("__alignof__", this), new Integer(LITERAL___alignof));
    literals.put(new ANTLRHashString("__asm", this), new Integer(LITERAL_asm));
    literals.put(new ANTLRHashString("__asm__", this), new Integer(LITERAL_asm));
    literals.put(new ANTLRHashString("__attribute__", this), new Integer(LITERAL___attribute));
    literals.put(new ANTLRHashString("__complex__", this), new Integer(LITERAL___complex));
    literals.put(new ANTLRHashString("__const", this), new Integer(LITERAL_const));
    literals.put(new ANTLRHashString("__const__", this), new Integer(LITERAL_const));
    literals.put(new ANTLRHashString("__imag__", this), new Integer(LITERAL___imag));
    literals.put(new ANTLRHashString("__inline", this), new Integer(LITERAL_inline));
    literals.put(new ANTLRHashString("__inline__", this), new Integer(LITERAL_inline));
    literals.put(new ANTLRHashString("__real__", this), new Integer(LITERAL___real));
    literals.put(new ANTLRHashString("__signed", this), new Integer(LITERAL_signed));
    literals.put(new ANTLRHashString("__signed__", this), new Integer(LITERAL_signed));
    literals.put(new ANTLRHashString("__typeof", this), new Integer(LITERAL_typeof));
    literals.put(new ANTLRHashString("__typeof__", this), new Integer(LITERAL_typeof));
    literals.put(new ANTLRHashString("__volatile", this), new Integer(LITERAL_volatile));
    literals.put(new ANTLRHashString("__volatile__", this), new Integer(LITERAL_volatile));
}


LineObject      lineObject = new LineObject();
String          originalSource = "";
PreprocessorInfoChannel preprocessorInfoChannel = new PreprocessorInfoChannel();
int             tokenNumber = 0;
boolean         countingTokens = true;
int             deferredLineCount = 0;

public void     setCountingTokens(boolean ct) {
    countingTokens = ct;
    if (countingTokens) {
        tokenNumber = 0;
    } else {
        tokenNumber = 1;
    }
}

public void     setOriginalSource(String src) {
    originalSource = src;
    lineObject.setSource(src);
}
public void     setSource(String src) {
    lineObject.setSource(src);
}

public PreprocessorInfoChannel getPreprocessorInfoChannel() {
    return preprocessorInfoChannel;
}

public void     setPreprocessingDirective(String pre) {
    preprocessorInfoChannel.addLineForTokenNumber(pre, new Integer(tokenNumber));
}

protected Token makeToken(int t) {
    if (t != Token.SKIP && countingTokens) {
        tokenNumber++;
    }
    CToken          tok = (CToken) super.makeToken(t);
    tok.setLine(lineObject.line);
    tok.setSource(lineObject.source);
```

```
        tok.setTokenNumber(tokenNumber);

        lineObject.line += deferredLineCount;
        deferredLineCount = 0;
        return tok;
    }

    public void     deferredNewline() {
        deferredLineCount++;
    }

    public void     newline() {
        lineObject.newline();
    }




}
Whitespace
:               ((' ' | '\t' | '\014')
        |               "\r\n" {
        newline();
        }
        |('\n' | '\r') {
        newline();
        }
        ) {
    _ttype = Token.SKIP;
}
;


protected
Escape
:     '\\'
(options {
 warnWhenFollowAmbig = false;
 }:
 ~('0'..'7' | 'x')
 | ('0'..'3') (options {
     warnWhenFollowAmbig = false;
     }:   Digit) *
 |('4'..'7') (options {
     warnWhenFollowAmbig = false;
     }:   Digit) *
 |'x'(options {
     warnWhenFollowAmbig = false;
 }:   Digit | 'a'..'f' | 'A'..'F') +
);

protected         IntSuffix
:                 'L'
|                 'l'
|                 'U'
|                 'u'
|                 'I'
|               'i'
|               'J'
|               'j'
;
protected       NumberSuffix
:
IntSuffix
|               'F'
|               'f'
;

Number
:      ((Digit) + ('.' | 'e' | 'E')) = >(Digit) +
('.'(Digit) * (Exponent) ?
 |Exponent
)
(NumberSuffix
) *

|("...") = >"..." {
     _ttype = VARARGS;
}

|'.' {
     _ttype = DOT;
}
((Digit) + (Exponent) ?
 {
 _ttype = Number;
 }
 (NumberSuffix
 ) *
) ?

|'0'('0'..'7') *
(NumberSuffix
) *

|'1'..'9'(Digit) *
(NumberSuffix
) *

|'0'('x' | 'X') ('a'..'f' | 'A'..'F' | Digit) +
(IntSuffix
) *
;

IDMEAT
:
i:     ID {

    if (i.getType() == LITERAL___extension__) {
        $setType(Token.SKIP);
    } else {
        $setType(i.getType());
    }

}
;
```

```
protected        ID
options
{
    testLiterals = true;
}
:               ('a'..'z' | 'A'..'Z' | '_' | '$')
('a'..'z' | 'A'..'Z' | '_' | '$' | '0'..'9') *
;

WideCharLiteral
:
'L' CharLiteral
{
    $setType(CharLiteral);
}
;



WideStringLiteral
:
'L' StringLiteral
{
    $setType(StringLiteral);
}
;

StringLiteral
:
'"'
(('\\' ~ ('\n')) = >Escape
 | ('\r' {
     newline();
     }
     |'\n' {
     newline();
     }
     |'\\' '\n' {
     newline();
     }
   )
 | ~('"' | '\r' | '\n' | '\\')
) *
'"'
;
```

Listing B.6: expandedGnuCParser.g

```
/*
 * expandedGnuCParser.g - Grammar file for expanded Gnu-C Grammar
 *                      - Modified for building
 *                        Branch-Preserving Call Graph
 *
 * AUTHOR: KK Lo, 7/2007
 *
 * REVISION HISTORY:
 *      Date            Name            Description
 *      11.07.2007      KK Lo           Initial Release
 */

{
    import java.io.*;
    import java.util.*;

    import antlr.CommonAST;
    import antlr.DumpASTVisitor;
}class GnuCParser extends Parser;

options {
    k= 2;
    exportVocab= GNUC;
    buildAST= true;
    ASTLabelType= "TNode";
    codeGenMakeSwitchThreshold= 2;
    codeGenBitsetTestThreshold= 3;
    importVocab=STDC;
}

{
    // Suppport C++-style single-line comments?
    public static boolean CPPComments = true;

    // access to symbol table
    public CSymbolTable symbolTable = new CSymbolTable();

    // source for names to unnamed scopes
    protected int unnamedScopeCounter = 0;

    public boolean isTypedefName(String name) {
        boolean returnValue = false;
        TNode node = symbolTable.lookupNameInCurrentScope(name);
        for (; node != null; node = (TNode) node.getNextSibling() ) {
            if(node.getType() == LITERAL_typedef) {
                returnValue = true;
                break;
            }
        }
        return returnValue;
    }


    public String getAScopeName() {
        return "" + (unnamedScopeCounter++);
    }

    public void pushScope(String scopeName) {
        symbolTable.pushScope(scopeName);
    }

    public void popScope() {
        symbolTable.popScope();
    }

    int traceDepth = 0;
    public void reportError(RecognitionException ex) {
        try {
            System.err.println("ANTLR Parsing Error: "+ex + " token name:" + tokenNames[LA(1)]);
            ex.printStackTrace(System.err);
        }
        catch (TokenStreamException e) {
            System.err.println("ANTLR Parsing Error: "+ex);
            ex.printStackTrace(System.err);
        }
    }
    public void reportError(String s) {
        System.err.println("ANTLR Parsing Error from String: " + s);
    }
    public void reportWarning(String s) {
        System.err.println("ANTLR Parsing Warning from String: " + s);
    }
    public void match(int t) throws MismatchedTokenException {
        boolean debugging = false;

        if ( debugging ) {
            for (int x=0; x<traceDepth; x++) System.out.print(" ");
            try {
                System.out.println("Match("+tokenNames[t]+") with LA(1)="+
                        tokenNames[LA(1)] + ((inputState.guessing>0)?" [inputState.guessing "+
                            inputState.guessing + "]":""));
            }
            catch (TokenStreamException e) {
                System.out.println("Match("+tokenNames[t]+") " + ((inputState.guessing>0)?" [inputState.
                    guessing "+ inputState.guessing + "]":""));

            }

        }
        try {
            if ( LA(1)!=t ) {
                if ( debugging ){
                    for (int x=0; x<traceDepth; x++) System.out.print(" ");
                    System.out.println("token mismatch: "+tokenNames[LA(1)]
                            + "!="+tokenNames[t]);
                }
                throw new MismatchedTokenException(tokenNames, LT(1), t, false, getFilename());

            } else {
                // mark token as consumed -- fetch next token deferred until LA/LT
                consume();
            }
        }
        catch (TokenStreamException e) {
        }

    }
    public void traceIn(String rname) {
        traceDepth += 1;
        for (int x=0; x<traceDepth; x++) System.out.print(" ");
        try {
            System.out.println("> "+rname+"; LA(1)==("+ tokenNames[LT(1).getType()]
                    + ") " + LT(1).getText() + " [inputState.guessing "+ inputState.guessing + "]");
        }
        catch (TokenStreamException e) {
        }
    }
    public void traceOut(String rname) {
        for (int x=0; x<traceDepth; x++) System.out.print(" ");
        try {
            System.out.println("< "+rname+"; LA(1)==("+ tokenNames[LT(1).getType()]
                    + ") "+LT(1).getText() + " [inputState.guessing "+ inputState.guessing + "]");
        }
        catch (TokenStreamException e) {
        }
        traceDepth -= 1;
    }

}
translationUnit[Node parent] :( externalList[parent] )?        /* Empty source files are allowed.  */
;

asm_expr[Node parent] :"asm"^
("volatile")? LCURLY expr[parent] RCURLY ( SEMI )+
;

idList :ID ( options{warnWhenFollowAmbig=false;}: COMMA ID )*
;

externalDef[Node parent] :( "typedef" | declaration[parent] )=> declaration[parent]
|       ( functionPrefix[parent] )=> functionDef[parent]
|       typelessDeclaration[parent]
|       asm_expr[parent]
|       SEMI
;

functionPrefix[Node parent] { String declName; }
:( (functionDeclSpecifiers[parent])=> ds:functionDeclSpecifiers[parent]
        |  //epsilon
 )
declName = d:declarator[true, parent]
( declaration[parent] )* (VARARGS)? ( SEMI )*
LCURLY
;

typelessDeclaration[Node parent] { AST typeMissing = #[NTypeMissing]; }
:initDeclList[typeMissing, parent] SEMI          { ## = #( #[NTypeMissing], ##); }
;

initializer[Node parent] :( ( ( (initializerElementLabel[parent])=> initializerElementLabel[parent] )?
            ( assignExpr[parent] | lcurlyInitializer[parent] )  { ## = #( #[NInitializer], ## ); }
            )
        | lcurlyInitializer[parent]
        )
;

initializerElementLabel[Node parent] :(   ( LBRACKET ((constExpr[parent] VARARGS)=> rangeExpr[parent] |
     constExpr[parent]) RBRACKET (ASSIGN)? )
        | ID COLON
        | DOT ID ASSIGN
        )
{ ## = #( #[NInitializerElementLabel], ##) ; }
;

lcurlyInitializer[Node parent] :LCURLY^ (initializerList[parent] ( COMMA! )? )? RCURLY
{ ##.setType( NLcurlyInitializer ); }
;
```

```
initializerList[Node parent] :initializer[parent] ( options{warnWhenFollowAmbig=false;}:COMMA!
     initializer[parent] )*
;

declarator[boolean isFunctionDefinition, Node parent] returns [String declName]{ declName = ""; }
:( pointerGroup )?

( id:ID                          { declName = id.getText();
  }
  | LPAREN declName = declarator[false, parent] RPAREN
)

( declaratorParamaterList[isFunctionDefinition, declName, parent]
  | LBRACKET ( expr[parent] )? RBRACKET
)*
{ ## = #( #[NDeclarator], ## ); }
;

declaratorParamaterList[boolean isFunctionDefinition, String declName, Node parent] :LPAREN^
{
    if (isFunctionDefinition) {
        pushScope(declName);
    }
    else {
        pushScope("!"+declName);
    }
}
(
 (declSpecifiers[parent])=> parameterTypeList[parent]
 | (idList)?
)
{
    popScope();
}
( COMMA! )?
RPAREN
{ ##.setType(NParameterTypeList); }
;

parameterTypeList[Node parent] :parameterDeclaration[parent]
(   options {
    warnWhenFollowAmbig = false;
    } :
    ( COMMA | SEMI )
    parameterDeclaration[parent]
)*
( ( COMMA | SEMI )
  VARARGS
)?
;

declarationList[Node parent] :(                options {   // this loop properly aborts when
        // it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :

        localLabelDeclaration
        |  ( declarationPredictor[parent] )=> declaration[parent]
        )+
 ;

localLabelDeclaration :( //GNU note: any __label__ declarations must come before regular declarations.
                        "__label__"^ ID (options{warnWhenFollowAmbig=false;}: COMMA! ID)* ( COMMA! )? (
                                SEMI! )+
                        )
 ;

declaration[Node parent] { AST ds1 = null; }
:ds:declSpecifiers[parent]       { ds1 = astFactory.dupList(#ds); }
(
 initDeclList[ds1, parent]
)?
( SEMI )+
{ ## = #( #[NDeclaration], ##); }

;

functionStorageClassSpecifier :"extern"
|       "static"
|       "inline"
;

typeSpecifier[int specCount, Node parent] returns [int retSpecCount]{ retSpecCount = specCount + 1; }
:( "void"
        |       "char"
        |       "short"
        |       "int"
        |       "long"
        |       "float"
        |       "double"
        |       "signed"
        |       "unsigned"
        |       structOrUnionSpecifier[parent]  ( options{warnWhenFollowAmbig=false;}: attributeDecl )*
        |       enumSpecifier[parent]
        |       { specCount==0 }? typedefName
        |       "typeof"^ LPAREN
        ( ( typeName[parent] )=> typeName[parent]
          | expr[parent]
        )
        RPAREN
        |       "__complex"
)
;

structOrUnionSpecifier[Node parent] { String scopeName; }
:sou:structOrUnion!
( ( ID LCURLY )=> i:ID l:LCURLY
  {
  scopeName = #sou.getText() + " " + #i.getText();
#l.setText(scopeName);
  pushScope(scopeName);
  }
  ( structDeclarationList[parent] )?
  { popScope();}
  RCURLY
  |   l1:LCURLY
  {
   scopeName = getAScopeName();
 #l1.setText(scopeName);
   pushScope(scopeName);
   }
   ( structDeclarationList[parent] )?
   { popScope(); }
   RCURLY
   | ID
 )
 {
 ## = #( #sou, ## );
 }
 ;

 structDeclaration[Node parent] :specifierQualifierList[parent] structDeclaratorList[parent] ( COMMA! )?
      ( SEMI! )+
 ;

 structDeclaratorList[Node parent] :structDeclarator[parent] ( options{warnWhenFollowAmbig=false;}: COMMA
      ! structDeclarator[parent] )*
 ;

 structDeclarator[Node parent] :( declarator[false, parent] )?
 ( COLON constExpr[parent] )?
 ( attributeDecl )*
 { ## = #( #[NStructDeclarator], ##); }
 ;

 enumSpecifier[Node parent] :"enum"^
 ( ( ID LCURLY )=> i:ID LCURLY enumList[i.getText(), parent] RCURLY
   | LCURLY enumList["anonymous", parent] RCURLY
   | ID
 )
 ;

 enumList[String enumName, Node parent] :enumerator[enumName, parent] ( options{warnWhenFollowAmbig=false
      ;}: COMMA! enumerator[enumName, parent] )* ( COMMA! )?
 ;

 initDeclList[AST declarationSpecifiers, Node parent] :initDecl[declarationSpecifiers, parent]
 ( options{warnWhenFollowAmbig=false;}: COMMA! initDecl[declarationSpecifiers, parent] )*
 ( COMMA! )?
 ;

 initDecl[AST declarationSpecifiers, Node parent] { String declName = ""; }
 :declName = d:declarator[false, parent]
 {   AST ds1, d1;
     ds1 = astFactory.dupList(declarationSpecifiers);
     d1 = astFactory.dupList(#d);
     symbolTable.add(declName, #(null, ds1, d1) );
 }
 ( attributeDecl )*
 ( ASSIGN initializer[parent]
   | COLON expr[parent]
 )?
 { ## = #( #[NInitDecl], ## ); }
 ;

 attributeDecl :"__attribute"^ LPAREN LPAREN attributeList RPAREN RPAREN
```

```
| "asm"^ LPAREN stringConst RPAREN { ##.setType( NAsmAttribute ); }
;

attributeList :attribute ( options{warnWhenFollowAmbig=false;}: COMMA attribute)* ( COMMA )?
;

attribute :( ~(LPAREN | RPAREN | COMMA)
            | LPAREN attributeList RPAREN
          )*
;

compoundStatement [String scopeName, String type, Node parent] :LCURLY^

{
    pushScope(scopeName);
}
(       //this ambiguity is ok, declarationList and nestedFunctionDef end properly
        options {
        warnWhenFollowAmbig = false;
        } :
        ( "typedef" | "__label__" | declaration[parent] )=> declarationList[parent]
        | (nestedFunctionDef[parent])=> nestedFunctionDef[parent]
)*
( statementList[parent] )?
{ popScope(); }
RCURLY
{ ##.setType( NCompoundStatement ); ##.setAttribute( "scopeName", scopeName );
}
;

nestedFunctionDef [Node parent] { String declName; }
:( "auto" )? //only for nested functions
( (functionDeclSpecifiers[parent])=> ds:functionDeclSpecifiers[parent]
)?
declName = d:declarator[false, parent]
{
    AST d2, ds2;
    d2 = astFactory.dupList(#d);
    ds2 = astFactory.dupList(#ds);
    symbolTable.add(declName, #(null, ds2, d2));
    pushScope(declName);
}
( declaration[parent] )*
{ popScope(); }
compoundStatement[declName, "function", new Node(declName)]
{ ## = #( #[NFunctionDef], ## );}
;

statement[String type, Node parent] {
    Node andNode1 = null;
    Node andNode2 = null;
    Node orNode = null;
}
:SEMI                   // Empty statements

|       compoundStatement[getAScopeName(), type, parent]       // Group of statements

|       expr[parent] SEMI!             { ## = #( #[NStatementExpr], ## );} // Expressions
```

```
// Iteration statements:

|       "while"^ LPAREN! expr[parent] RPAREN!
{
     andNode1 = new Node("AND");
     System.out.println(parent + " -> " + andNode1 + ";");
}
statement["while", andNode1]
|       "do"^
{
     andNode1 = new Node("AND");
     System.out.println(parent + " -> " + andNode1 + ";");
}
statement["do", andNode1] "while"! LPAREN! expr[parent] RPAREN! SEMI!
|!       "for"
LPAREN ( e1:expr[parent] )? SEMI ( e2:expr[parent] )? SEMI ( e3:expr[parent] )? RPAREN
{
     andNode1 = new Node("AND");
     System.out.println(parent + " -> " + andNode1 + ";");
}
s:statement["for", andNode1]
{
     if ( #e1 == null) { #e1 = #[ NEmptyExpression ]; }
     if ( #e2 == null) { #e2 = #[ NEmptyExpression ]; }
     if ( #e3 == null) { #e3 = #[ NEmptyExpression ]; }
## = #( #[LITERAL_for, "for"], #e1, #e2, #e3, #s );
}


// Jump statements:

|       "goto"^ expr[parent] SEMI!
|       "continue" SEMI!
|       "break" SEMI!
|       "return"^ ( expr[parent] )? SEMI!


|       ID COLON! (options {warnWhenFollowAmbig=false;}: statement["label", parent])?  { ## = #( #[
     NLabel], ## ); }
// GNU allows range expressions in case statements
|       "case"^
{
     andNode1 = new Node("AND");
     System.out.println(parent + " -> " + andNode1 + ";");
}
((constExpr[andNode1] VARARGS)=> rangeExpr[andNode1] | constExpr[andNode1]) COLON! ( options{
     warnWhenFollowAmbig=false;}:statement["case", andNode1] )?
|       "default"^
{
     andNode1 = new Node("AND");
     System.out.println(parent + " -> " + andNode1 + ";");
}
COLON! ( options{warnWhenFollowAmbig=false;}: statement["default", andNode1] )?

// Selection statements:
|
"if"^
LPAREN! expr[parent] RPAREN! {
     orNode = new Node("OR");
```

```
      System.out.println(parent + " -> " + orNode + ";");
      //There are 2 branches for each if-else statement
      andNode1 = new Node("AND");
      andNode2 = new Node("AND");
 }
 statement["if", andNode1]
 {
      // If block
      System.out.println(orNode + " -> " + andNode1 + ";");
 }
 ( //standard if-else ambiguity
   options {
   warnWhenFollowAmbig = false;
   } :
   "else" statement["else", andNode2]
   {
   // else block
   System.out.println(orNode + " -> " + andNode2 + ";");
   }
)?
|       "switch"^ LPAREN! expr[parent]
{
     orNode = new Node("OR");
     System.out.println(parent + " -> " + orNode  + ";");
}
RPAREN! statement["switch", orNode]
;

conditionalExpr[Node parent] :logicalOrExpr[parent]
( QUESTION^ (expr[parent])? COLON conditionalExpr[parent] )?
;

rangeExpr[Node parent] :constExpr[parent] VARARGS constExpr[parent]
{ ## = #(#[NRangeExpr], ##); }
;

    castExpr[Node parent] :( LPAREN typeName[parent] RPAREN )=>
LPAREN^ typeName[parent] RPAREN ( castExpr[parent] | lcurlyInitializer[parent] )
{ ##.setType(NCast); }

|       unaryExpr[parent]
;

nonemptyAbstractDeclarator[Node parent] :(
        pointerGroup
        (   (LPAREN
             (   nonemptyAbstractDeclarator[parent]
                 | parameterTypeList[parent]
             )?
             ( COMMA! )?
             RPAREN)
            | (LBRACKET (expr[parent])? RBRACKET)
        )*

        |   (   (LPAREN
                 (   nonemptyAbstractDeclarator[parent]
                     | parameterTypeList[parent]
                 )?
                 ( COMMA! )?
```

```
                RPAREN)
              | (LBRACKET (expr[parent])? RBRACKET)
              )+
          )
{   ## = #( #[NNonemptyAbstractDeclarator], ## ); }

;

unaryExpr[Node parent] :postfixExpr[parent]
|        INC^ castExpr[parent]
|        DEC^ castExpr[parent]
|        u:unaryOperator castExpr[parent] { ## = #( #[NUnaryExpr], ## ); }

|        "sizeof"^
( ( LPAREN typeName[parent] )=> LPAREN typeName[parent] RPAREN
  | unaryExpr[parent]
)
|        "__alignof"^
( ( LPAREN typeName[parent] )=> LPAREN typeName[parent] RPAREN
  | unaryExpr[parent]
)
|        gnuAsmExpr[parent]
;

unaryOperator :BAND
|        STAR
|        PLUS
|        MINUS
|        BNOT    //also stands for complex conjugation
|        LNOT
|        LAND    //for label dereference (&&label)
|        "__real"
|        "__imag"
;

gnuAsmExpr[Node parent] :"asm"^ ("volatile")?
LPAREN stringConst
( options { warnWhenFollowAmbig = false; }:
  COLON (strOptExprPair[parent] ( COMMA strOptExprPair[parent])* )?
  ( options { warnWhenFollowAmbig = false; }:
    COLON (strOptExprPair[parent] ( COMMA strOptExprPair[parent])* )?
  )?
)?
( COLON stringConst ( COMMA stringConst)* )?
RPAREN
{ ##.setType(NGnuAsmExpr); }
;

strOptExprPair[Node parent] :stringConst ( LPAREN expr[parent] RPAREN )?
;

primaryExpr[Node parent] returns [String idName]{idName = "";}
:id:ID {idName = id.getText();}
|        Number
|        charConst
|        stringConst
// JTC:
// ID should catch the enumerator
// leaving it in gives ambiguous err
//        | enumerator
|        (LPAREN LCURLY) => LPAREN^ compoundStatement[getAScopeName(), "scope", parent] RPAREN
|        LPAREN^ expr[parent] RPAREN          { ##.setType(NExpressionGroup); }
;

// inherited from grammar StdCParser
externalList[Node parent] :( externalDef[parent] )+
;

// inherited from grammar StdCParser
declSpecifiers[Node parent] { int specCount=0; }
:(               options { // this loop properly aborts when
        //  it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :
s:storageClassSpecifier
| typeQualifier
| ( "struct" | "union" | "enum" | typeSpecifier[specCount , parent] )=>
specCount = typeSpecifier[specCount , parent]
 )+
;

// inherited from grammar StdCParser
storageClassSpecifier :"auto"
|        "register"
|        "typedef"
|        functionStorageClassSpecifier
;

// inherited from grammar StdCParser
typeQualifier :"const"
|        "volatile"
;

// inherited from grammar StdCParser
typedefName :{ isTypedefName ( LT(1).getText() ) }?
i:ID                        { ## = #(#[NTypedefName], #i); }
;

// inherited from grammar StdCParser
structOrUnion :"struct"
|        "union"
;

// inherited from grammar StdCParser
structDeclarationList[Node parent] :( structDeclaration[parent] )+
;

// inherited from grammar StdCParser
specifierQualifierList[Node parent] { int specCount = 0; }
:(               options {   // this loop properly aborts when
        // it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :
        ( "struct" | "union" | "enum" | typeSpecifier[specCount , parent] )=>
        specCount = typeSpecifier[specCount , parent]
        | typeQualifier
 )+
;
```

```
// inherited from grammar StdCParser
enumerator[String enumName, Node parent] :i:ID              { symbolTable.add( i.getText(),
#(   null,
#[LITERAL_enum, "enum"],
#[ ID, enumName]
 )
        );
}
(ASSIGN constExpr[parent])?
;

// inherited from grammar StdCParser
pointerGroup :( STAR ( typeQualifier )* )+    { ## = #( #[NPointerGroup], ##); }
;

// inherited from grammar StdCParser
parameterDeclaration[Node parent] { String declName; }
:ds:declSpecifiers[parent]
( ( declarator[false, parent] )=> declName = d:declarator[false, parent]
  {
  AST d2, ds2;
  d2 = astFactory.dupList(#d);
  ds2 = astFactory.dupList(#ds);
  symbolTable.add(declName, #(null, ds2, d2));
  }
  | nonemptyAbstractDeclarator[parent]
)?
{
## = #( #[NParameterDeclaration], ## );
}
;

// inherited from grammar StdCParser
functionDef[Node parent] { String declName;
    Node andNode = null;
}
:( (functionDeclSpecifiers[parent])=> ds:functionDeclSpecifiers[parent]
       | //epsilon
 )
declName = d:declarator[true, parent]
{
    AST d2, ds2;
    d2 = astFactory.dupList(#d);
    ds2 = astFactory.dupList(#ds);
    symbolTable.add(declName, #(null, ds2, d2));
    pushScope(declName);
    //I add the code here
    andNode = new Node("AND");
    System.out.println(declName + " [shape=rectangle] ;");
    System.out.println(declName + " -> " + andNode + ";");
    parent = andNode;

}
( declaration[parent] )* (VARARGS)? ( SEMI! )*
{ popScope();
}
compoundStatement[declName, "function", parent]
{ ## = #( #[NFunctionDef], ## );}
```

```
;

// inherited from grammar StdCParser
functionDeclSpecifiers[Node parent] { int specCount = 0; }
:(              options {   // this loop properly aborts when
        // it finds a non-typedefName ID MBZ
        warnWhenFollowAmbig = false;
        } :
        functionStorageClassSpecifier
        | typeQualifier
        | ( "struct" | "union" | "enum" | typeSpecifier[specCount, parent] )=>
        specCount = typeSpecifier[specCount, parent]
 )+
;

// inherited from grammar StdCParser
declarationPredictor[Node parent] :(options {      //only want to look at declaration if I don't see
     typedef
        warnWhenFollowAmbig = false;
        }:
        "typedef"
        | declaration[parent]
        )
;

// inherited from grammar StdCParser
statementList[Node parent] :( statement[null, parent] )+
;

// inherited from grammar StdCParser
expr[Node parent] :assignExpr[parent] (options {
        /* MBZ:
           COMMA is ambiguous between comma expressions and
           argument lists.  argExprList should get priority,
           and it does by being deeper in the expr rule tree
           and using (COMMA assignExpr)*
         */
        warnWhenFollowAmbig = false;
        } :
c:COMMA^ { #c.setType(NCommaExpr); } assignExpr[parent]
        )*
;

// inherited from grammar StdCParser
assignExpr[Node parent] :conditionalExpr[parent] ( a:assignOperator! assignExpr[parent] { ## = #( #a, ##
     );} )?
;

// inherited from grammar StdCParser
assignOperator  :ASSIGN
|       DIV_ASSIGN
|       PLUS_ASSIGN
|       MINUS_ASSIGN
|       STAR_ASSIGN
|       MOD_ASSIGN
|       RSHIFT_ASSIGN
|       LSHIFT_ASSIGN
|       BAND_ASSIGN
|       BOR_ASSIGN
|       BXOR_ASSIGN
;

// inherited from grammar StdCParser
constExpr[Node parent] :conditionalExpr[parent]
;

// inherited from grammar StdCParser
logicalOrExpr[Node parent] :logicalAndExpr[parent] ( LOR^ logicalAndExpr[parent] )*
;

// inherited from grammar StdCParser
logicalAndExpr[Node parent] :inclusiveOrExpr[parent] ( LAND^ inclusiveOrExpr[parent] )*
;

// inherited from grammar StdCParser
inclusiveOrExpr[Node parent] :exclusiveOrExpr[parent] ( BOR^ exclusiveOrExpr[parent] )*
;

// inherited from grammar StdCParser
exclusiveOrExpr[Node parent] :bitAndExpr[parent] ( BXOR^ bitAndExpr[parent] )*
;

// inherited from grammar StdCParser
bitAndExpr[Node parent] :equalityExpr[parent] ( BAND^ equalityExpr[parent] )*
;

// inherited from grammar StdCParser
equalityExpr[Node parent] :relationalExpr[parent]
( ( EQUAL^ | NOT_EQUAL^ ) relationalExpr[parent] )*
;

// inherited from grammar StdCParser
relationalExpr[Node parent] :shiftExpr[parent]
( ( LT^ | LTE^ | GT^ | GTE^ ) shiftExpr[parent] )*
;

// inherited from grammar StdCParser
shiftExpr[Node parent] :additiveExpr[parent]
( ( LSHIFT^ | RSHIFT^ ) additiveExpr[parent] )*
;

// inherited from grammar StdCParser
additiveExpr[Node parent] :multExpr[parent]
( ( PLUS^ | MINUS^ ) multExpr[parent] )*
;

// inherited from grammar StdCParser
multExpr[Node parent] :castExpr[parent]
( ( STAR^ | DIV^ | MOD^ ) castExpr[parent] )*
;

// inherited from grammar StdCParser
typeName[Node parent] :specifierQualifierList[parent] (nonemptyAbstractDeclarator[parent])?
;

// inherited from grammar StdCParser
postfixExpr[Node parent] {String idName="";}
:idName = f:primaryExpr[parent]
```

```
(
 postfixSuffix[parent, idName]                {## = #( #[NPostfixExpr], ## );}
)?
;

// inherited from grammar StdCParser
postfixSuffix[Node parent, String idName] ://{System.out.println("STD: POSTFIXSUFFIX");}
( PTR ID
  | DOT ID
  | functionCall[parent, idName]
  | LBRACKET expr[parent] RBRACKET
  | INC
  | DEC
)+
;

// inherited from grammar StdCParser
functionCall[Node parent, String idName] ://{System.out.println("STD: FUNCTION CALL");}
LPAREN^ (a:argExprList[parent])? RPAREN
{
##.setType( NFunctionCallArgs );
    System.out.println(parent + " -> " + idName + ";");
}
;

// inherited from grammar StdCParser
argExprList[Node parent] :assignExpr[parent] ( COMMA! assignExpr[parent] )*
;

// inherited from grammar StdCParser
protected charConst :CharLiteral
;

// inherited from grammar StdCParser
protected stringConst :(StringLiteral)+             { ## = #(#[NStringSeq], ##); }
;

// inherited from grammar StdCParser
protected intConst :IntOctalConst
|       LongOctalConst
|       UnsignedOctalConst
|       IntIntConst
|       LongIntConst
|       UnsignedIntConst
|       IntHexConst
|       LongHexConst
|       UnsignedHexConst
;

// inherited from grammar StdCParser
protected floatConst :FloatDoubleConst
|       DoubleDoubleConst
|       LongDoubleConst
;

// inherited from grammar StdCParser
dummy :NTypedefName
|       NInitDecl
|       NDeclarator
|       NStructDeclarator
|       NDeclaration
|       NCast
|       NPointerGroup
|       NExpressionGroup
|       NFunctionCallArgs
|       NNonemptyAbstractDeclarator
|       NInitializer
|       NStatementExpr
|       NEmptyExpression
|       NParameterTypeList
|       NFunctionDef
|       NCompoundStatement
|       NParameterDeclaration
|       NCommaExpr
|       NUnaryExpr
|       NLabel
|       NPostfixExpr
|       NRangeExpr
|       NStringSeq
|       NInitializerElementLabel
|       NLcurlyInitializer
|       NAsmAttribute
|       NGnuAsmExpr
|       NTypeMissing
;

{
    //import CToken;
    import java.io.*;
    //import LineObject;
    import antlr.*;
}class GnuCLexer extends Lexer;

options {
    k= 3;
    importVocab= GNUC;
    testLiterals= false;
}

tokens {
    LITERAL___extension__ = "__extension__";
}
{
    public void initialize(String src)
    {
        setOriginalSource(src);
        initialize();
    }

    public void initialize()
    {
        literals.put(new ANTLRHashString("__alignof__", this), new Integer(LITERAL___alignof));
        literals.put(new ANTLRHashString("__asm", this), new Integer(LITERAL_asm));
        literals.put(new ANTLRHashString("__asm__", this), new Integer(LITERAL_asm));
        literals.put(new ANTLRHashString("__attribute__", this), new Integer(LITERAL___attribute));
        literals.put(new ANTLRHashString("__complex__", this), new Integer(LITERAL___complex));
        literals.put(new ANTLRHashString("__const", this), new Integer(LITERAL_const));
        literals.put(new ANTLRHashString("__const__", this), new Integer(LITERAL_const));
        literals.put(new ANTLRHashString("__imag__", this), new Integer(LITERAL___imag));
        literals.put(new ANTLRHashString("__inline", this), new Integer(LITERAL_inline));
        literals.put(new ANTLRHashString("__inline__", this), new Integer(LITERAL_inline));
        literals.put(new ANTLRHashString("__real__", this), new Integer(LITERAL___real));
        literals.put(new ANTLRHashString("__signed", this), new Integer(LITERAL_signed));
        literals.put(new ANTLRHashString("__signed__", this), new Integer(LITERAL_signed));
        literals.put(new ANTLRHashString("__typeof", this), new Integer(LITERAL_typeof));
        literals.put(new ANTLRHashString("__typeof__", this), new Integer(LITERAL_typeof));
        literals.put(new ANTLRHashString("__volatile", this), new Integer(LITERAL_volatile));
        literals.put(new ANTLRHashString("__volatile__", this), new Integer(LITERAL_volatile));
    }


    LineObject lineObject = new LineObject();
    String originalSource = "";
    PreprocessorInfoChannel preprocessorInfoChannel = new PreprocessorInfoChannel();
    int tokenNumber = 0;
    boolean countingTokens = true;
    int deferredLineCount = 0;

    public void setCountingTokens(boolean ct)
    {
        countingTokens = ct;
        if ( countingTokens ) {
            tokenNumber = 0;
        }
        else {
            tokenNumber = 1;
        }
    }

    public void setOriginalSource(String src)
    {
        originalSource = src;
        lineObject.setSource(src);
    }
    public void setSource(String src)
    {
        lineObject.setSource(src);
    }

    public PreprocessorInfoChannel getPreprocessorInfoChannel()
    {
        return preprocessorInfoChannel;
    }

    public void setPreprocessingDirective(String pre)
    {
        preprocessorInfoChannel.addLineForTokenNumber( pre, new Integer(tokenNumber) );
    }

    protected Token makeToken(int t)
    {
        if ( t != Token.SKIP && countingTokens) {
            tokenNumber++;
        }
        CToken tok = (CToken) super.makeToken(t);
        tok.setLine(lineObject.line);
        tok.setSource(lineObject.source);
        tok.setTokenNumber(tokenNumber);

        lineObject.line += deferredLineCount;
        deferredLineCount = 0;
        return tok;
    }

    public void deferredNewline() {
        deferredLineCount++;
    }

    public void newline() {
        lineObject.newline();
    }




}
Whitespace :( ( ' ' | '\t' | '\014')
             | "\r\n"                  { newline(); }
             | ( '\n' | '\r' )         { newline();    }
            )                        { _ttype = Token.SKIP;  }
;

protected Escape :'\\'
( options{warnWhenFollowAmbig=false;}:
  ~('0'..'7' | 'x')
  | ('0'..'3') ( options{warnWhenFollowAmbig=false;}: Digit )*
  | ('4'..'7') ( options{warnWhenFollowAmbig=false;}: Digit )*
  | 'x' ( options{warnWhenFollowAmbig=false;}: Digit | 'a'..'f' | 'A'..'F' )+
)
;

protected IntSuffix :'L'
| 'l'
| 'U'
| 'u'
| 'I'
| 'i'
| 'J'
| 'j'
;

protected NumberSuffix :IntSuffix
| 'F'
| 'f'
;

Number :( ( Digit )+ ( '.' | 'e' | 'E' ) )=> ( Digit )+
( '.' ( Digit )* ( Exponent )?
  | Exponent
)
( NumberSuffix
)*

|       ( "..." )=> "..."       { _ttype = VARARGS;     }
|       '.'                     { _ttype = DOT; }
( ( Digit )+ ( Exponent )?
  { _ttype = Number;   }
  ( NumberSuffix
  )*
)?

|       '0' ( '0'..'7' )*
( NumberSuffix
)*

|       '1'..'9' ( Digit )*
( NumberSuffix
)*

|       '0' ( 'x' | 'X' ) ( 'a'..'f' | 'A'..'F' | Digit )+
( IntSuffix
)*
;

IDMEAT :i:ID                  {

            if ( i.getType() == LITERAL___extension__ ) {
                $setType(Token.SKIP);
            }
            else {
                $setType(i.getType());
            }

        }
;

protected ID
options {
    testLiterals= true;
}
:( 'a'..'z' | 'A'..'Z' | '_' | '$')
( 'a'..'z' | 'A'..'Z' | '_' | '$' | '0'..'9' )*
;

WideCharLiteral :'L' CharLiteral
{ $setType(CharLiteral); }
;

WideStringLiteral :'L' StringLiteral
{ $setType(StringLiteral); }
;

StringLiteral :'"'
( ('\\' ~('\n'))=> Escape
  | ( '\r'          { newline(); }
      | '\n'        {
      newline();
      }
      | '\\' '\n'   {
      newline();
      }
    )
    | ~( '"' | '\r' | '\n' | '\\' )
  )*
  '"'
  ;

// inherited from grammar StdCLexer
protected Vocabulary :'\3'..'\377'
;

// inherited from grammar StdCLexer
ASSIGN :'=' ;

// inherited from grammar StdCLexer
COLON :':' ;

// inherited from grammar StdCLexer
COMMA :',' ;

// inherited from grammar StdCLexer
QUESTION :'?' ;

// inherited from grammar StdCLexer
SEMI :';' ;

// inherited from grammar StdCLexer
PTR :"->" ;

// inherited from grammar StdCLexer
protected DOT :;

// inherited from grammar StdCLexer
protected VARARGS :;

// inherited from grammar StdCLexer
LPAREN :'(' ;

// inherited from grammar StdCLexer
RPAREN :')' ;

// inherited from grammar StdCLexer
LBRACKET :'[' ;

// inherited from grammar StdCLexer
RBRACKET :']' ;

// inherited from grammar StdCLexer
LCURLY :'{' ;

// inherited from grammar StdCLexer
RCURLY :'}' ;

// inherited from grammar StdCLexer
EQUAL :"==" ;

// inherited from grammar StdCLexer
NOT_EQUAL :"!=" ;

// inherited from grammar StdCLexer
LTE :"<=" ;
```

```
// inherited from grammar StdCLexer
LT :"<" ;

// inherited from grammar StdCLexer
GTE :">=" ;

// inherited from grammar StdCLexer
GT :">" ;

// inherited from grammar StdCLexer
DIV :'/' ;

// inherited from grammar StdCLexer
DIV_ASSIGN :"/=" ;

// inherited from grammar StdCLexer
PLUS :'+' ;

// inherited from grammar StdCLexer
PLUS_ASSIGN :"+=" ;

// inherited from grammar StdCLexer
INC :"++" ;

// inherited from grammar StdCLexer
MINUS :'-' ;

// inherited from grammar StdCLexer
MINUS_ASSIGN :"-=" ;

// inherited from grammar StdCLexer
DEC :"--" ;

// inherited from grammar StdCLexer
STAR :'*' ;

// inherited from grammar StdCLexer
STAR_ASSIGN :"*=" ;

// inherited from grammar StdCLexer
MOD :'%' ;

// inherited from grammar StdCLexer
MOD_ASSIGN :"%=" ;

// inherited from grammar StdCLexer
RSHIFT :">>" ;

// inherited from grammar StdCLexer
RSHIFT_ASSIGN :">>=" ;

// inherited from grammar StdCLexer
LSHIFT :"<<" ;

// inherited from grammar StdCLexer
LSHIFT_ASSIGN :"<<=" ;

// inherited from grammar StdCLexer
```

```
LAND :"&&" ;

// inherited from grammar StdCLexer
LNOT :'!' ;

// inherited from grammar StdCLexer
LOR :"||" ;

// inherited from grammar StdCLexer
BAND :'&' ;

// inherited from grammar StdCLexer
BAND_ASSIGN :"&=" ;

// inherited from grammar StdCLexer
BNOT :'~' ;

// inherited from grammar StdCLexer
BOR :'|' ;

// inherited from grammar StdCLexer
BOR_ASSIGN :"|=" ;

// inherited from grammar StdCLexer
BXOR :'^' ;

// inherited from grammar StdCLexer
BXOR_ASSIGN :"^=" ;

// inherited from grammar StdCLexer
Comment :"/*"
( { LA(2) != '/' }? '*'
  | "\r\n"                  { deferredNewline(); }
  | ( '\r' | '\n' )         { deferredNewline();    }
  | ~( '*'| '\r' | '\n' )
)*
"*/"                      { _ttype = Token.SKIP;
}
;

// inherited from grammar StdCLexer
CPPComment :"//" ( ~('\n') )*
{
    _ttype = Token.SKIP;
}
;

// inherited from grammar StdCLexer
PREPROC_DIRECTIVE
options {
    paraphrase= "a line directive";
}
:'#'
( ( "line" || (( ' ' | '\t' | '\014')+ '0'..'9')) => LineDirective
  | (~'\n')*                                  { setPreprocessingDirective(getText()); }
)
{
    _ttype = Token.SKIP;
}
```

```
;

// inherited from grammar StdCLexer
protected Space :( ' ' | '\t' | '\014')
;

// inherited from grammar StdCLexer
protected LineDirective {
    boolean oldCountingTokens = countingTokens;
    countingTokens = false;
}
:{
    lineObject = new LineObject();
    deferredLineCount = 0;
}
("line")? //this would be for if the directive started "#line", but not there for GNU directives
(Space)+
n:Number { lineObject.setLine(Integer.parseInt(n.getText())); }
(Space)+
(       fn:StringLiteral {  try {
        lineObject.setSource(fn.getText().substring(1,fn.getText().length()-1));
        }
        catch (StringIndexOutOfBoundsException e) { /*not possible*/ }
        }
        | fi:ID { lineObject.setSource(fi.getText()); }
)?
(Space)*
("1"            { lineObject.setEnteringFile(true); } )?
(Space)*
("2"            { lineObject.setReturningToFile(true); } )?
(Space)*
("3"            { lineObject.setSystemHeader(true); } )?
(Space)*
("4"            { lineObject.setTreatAsC(true); } )?
(~('\r' | '\n'))*
("\r\n" | "\r" | "\n")
{
    preprocessorInfoChannel.addLineForTokenNumber(new LineObject(lineObject), new Integer(tokenNumber));
    countingTokens = oldCountingTokens;
}
;

// inherited from grammar StdCLexer
CharLiteral :'\'' ( Escape | ~( '\'' ) ) '\''
;

// inherited from grammar StdCLexer
protected BadStringLiteral :// Imaginary token.
;

// inherited from grammar StdCLexer
protected Digit :'0'..'9'
;

// inherited from grammar StdCLexer
protected LongSuffix :'l'
|       'L'
;
```

```
// inherited from grammar StdCLexer
protected UnsignedSuffix :'u'
|       'U'
;

// inherited from grammar StdCLexer
protected FloatSuffix :'f'
|       'F'
;

// inherited from grammar StdCLexer
protected Exponent :( 'e' | 'E' ) ( '+' | '-' )? ( Digit )+
;

// inherited from grammar StdCLexer
protected DoubleDoubleConst :;

// inherited from grammar StdCLexer
protected FloatDoubleConst :;

// inherited from grammar StdCLexer
protected LongDoubleConst :;

// inherited from grammar StdCLexer
protected IntOctalConst :;

// inherited from grammar StdCLexer
protected LongOctalConst :;

// inherited from grammar StdCLexer
protected UnsignedOctalConst :;

// inherited from grammar StdCLexer
protected IntIntConst :;

// inherited from grammar StdCLexer
protected LongIntConst :;

// inherited from grammar StdCLexer
protected UnsignedIntConst :;

// inherited from grammar StdCLexer
protected IntHexConst :;

// inherited from grammar StdCLexer
protected LongHexConst :;

// inherited from grammar StdCLexer
protected UnsignedHexConst :;
```

Bibliography

[ACC+02] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng., 28(10):970–983, 2002.

[ACLS02] Lerina Aversano, Gerardo Canfora, Andrea De Lucia, and Silvio Stefanucci. Evolving ispell: A case study of program understanding for reuse. In IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension, page 197, Washington, DC, USA, 2002. IEEE Computer Society.

[ACT99] G. Antoniol, F. Calzolari, and P. Tonella. Impact of function pointers on the call graph. In CSMR '99: Proceedings of the Third European Conference on Software Maintenance and Reengineering, page 51, Washington, DC, USA, 1999. IEEE Computer Society.

[AL01] Gail C. Murphy Albert Lai. Capturing concerns with conceptual modules. In Position Paper for the Workshop on Advanced Separation of Concerns, held as part of ICSE 2001, 10 2001.

[Atk04] Darren C. Atkinson. Accurate call graph extraction of programs with function pointers using type signatures. In APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering

Conference (APSEC'04), pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[BDET04] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection techniques for identifying crosscutting concerns. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM). IEEE Computer Society Press, 9 2004.

[BDET05] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying crosscutting concern code. IEEE Transactions on Software Engineering, 31(10):804–818, 10 2005.

[BGH07] David Binkley, Nicolas Gold, and Mark Harman. An empirical study of static program slice size. ACM Trans. Softw. Eng. Methodol., 16(2):8, 2007.

[Bin94] David Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In CC '94: Proceedings of the 5th International Conference on Compiler Construction, pages 374–388, London, UK, 1994. Springer-Verlag.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. In PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, pages 300–313, New York, NY, USA, 1993. ACM Press.

[BM98] Elisa L. A. Baniassad and Gail C. Murphy. Conceptual module querying for software reengineering. In ICSE '98: Proceedings of the 20th international conference on Software engineering, pages 64–73, Washington, DC, USA, 1998. IEEE Computer Society.

[BM06] Cathal Boogerd and Leon Moonen. Ranking software inspection results using execution likelihood. In Jaap van der Heijden, editor, Proceedings of the Philips Software Conference (PSC), page 10. Philips, November 2006.

[BZ06] Silvia Breu and Thomas Zimmermann. Mining aspects from history. In Sebastian Uchitel and Steve Easterbrook, editors, 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006). ACM Press, September 2006. Accepted for publication.

[DDL+90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. Journal of the American Society of Information Science, 41(6):391–407, 1990.

[Egy03] Alexander Egyed. A scenario-driven approach to trace dependency analysis. IEEE Trans. Softw. Eng., 29(2):116–132, 2003.

[EKS01] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), pages 602–611, Washington, DC, USA, 2001. IEEE Computer Society.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. IEEE Trans. Softw. Eng., 29(3):210–224, 2003.

[Fre79] Linton Clarke Freeman. Centrality in social networks: Conceptual clarification I. Social Networks, 1:215–239, 1979.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 108–124, New York, NY, USA, 1997. ACM Press.

[GK97] Jean-Francois Girard and Rainer Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In ICSM '97: Proceedings of the International Conference on Software Maintenance, pages 58–65, Washington, DC, USA, 1997. IEEE Computer Society.

[IYF+03] Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank: relative significance rank for software component search. In ICSE '03: Proceedings of the 25th International Conference on Software Engineering, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.

[IYYK05] Member-Katsuro Inoue, Member-Reishi Yokomori, Member-Tetsuo Yamamoto, and Member-Shinji Kusumoto. Ranking significance of software components based on use relations. IEEE Trans. Softw. Eng., 31(3):213–225, 2005.

[KR00] Jens Knoop and Oliver Rüthing. Constant propagation on the value graph: Simple constants and beyond. In CC '00: Proceedings of the 9th International Conference on Compiler Construction, pages 94–109, London, UK, 2000. Springer-Verlag.

[LL03] Jonas Lundberg and Welf Löwe. Architecture recovery by semi-automatic component identification. Electronic Notes in Theoretical Computer Science, 82(5):98–114, April 2003.

[MK88] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In ICSE '88: Proceedings of the 10th international conference on Software engineering, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[MM03] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In ICSE '03: Proceedings of the 25th International Conference on Software Engineering, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[MRR04] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. Automated Software Engg., 11(1):7–26, 2004.

[Mur96] Gail C. Murphy. Lightweight structural summarization as an aid to software evolution. PhD thesis, 1996. Chairperson-Alan Borning.

[MvDM04] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

[PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[PIKK98] Lutz Prechelt, Fakultat Fur Informatik, Christian Kramer, and Kanthor Ag Karlsruhe. Functionality versus practicality: Employing existing tools for recovering structural design patterns. Journal of Universal Computer Science, 4:88–2, 1998.

[QZZ+03] Tao Qin, Lu Zhang, Zhiying Zhou, Dan Hao, and Jiasu Sun. Discovering use cases from source code using the branch-reserving call graph. In APSEC '03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference, page 60, Washington, DC, USA, 2003. IEEE Computer Society.

[Rad00] Ansgar Radermacher. Support for design patterns through graph transformation tools. In AGTIVE '99: Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance, pages 111–126, London, UK, 2000. Springer-Verlag.

[RB03] Awais Rashid and Lynne Blair. Editorial: Aspect-oriented programming and separation of crosscutting concerns. Comput. J., 46(5):527–528, 2003.

[Rob05] Martin P. Robillard. Automatic generation of suggestions for program investigation. SIGSOFT Softw. Eng. Notes, 30(5):11–20, 2005.

[Ryd79] B. G. Ryder. Constructing the call graph of a program. IEEE Trans. Softw. Eng., 5(3):216–226, 1979.

[SFDB07] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending random walks. In ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 15–24, New York, NY, USA, 2007. ACM.

[SGMB03] Simon C. Shaw, Michael Goldstein, Malcolm Munro, and Elizabeth Burd. Moral dominance relations for program comprehension. IEEE Trans. Softw. Eng., 29(9):851–863, 2003.

[SKL$^+$02] Reinhard Schauer, Rudolf K. Keller, B. Lagué, Gregory Robitaille, Séstieu Robitaille, and Guy Saing-Denis. The spool design repository: architecture, schema, and mechanisms. pages 269–294, 2002.

[TH99] Vassilios Tzerpos and R. C. Holt. Mojo: A distance metric for software clusterings. In WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering, page 187, Washington, DC, USA, 1999. IEEE Computer Society.

[TH00] Vassilios Tzerpos and R.C. Holt. Acdc: An algorithm for comprehension-driven clustering. wcre, 0:258, 2000.

[TX07] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pages 204–213, New York, NY, USA, 2007. ACM.

[vMV95] A. von Mayrhauser and A. M. Vans. Industrial experience with an integrated code comprehension model. Software Engineering Journal, 10(5):171–182, 1995.

[WC96] Norman Wilde and Christopher Casey. Early field experience with the software reconnaissance technique for program comprehension. In ICSM '96: Proceedings of the 1996 International

Conference on Software Maintenance, pages 312–318, Washington, DC, USA, 1996. IEEE Computer Society.

[Wei79] Mark David Weiser. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, Ann Arbor, MI, USA, 1979.

[Wei84] Mark Weiser. Program slicing. IEEE Trans. Software Eng., 10(4):352–357, 1984.

[WL94] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture, pages 1–11, New York, NY, USA, 1994. ACM Press.

[WMGH94] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, pages 85–96, New York, NY, USA, 1994. ACM Press.

[Won99] Weng-Fai Wong. Source level static branch prediction. The Computer Journal, (42), 1999.

[WS95] Norman Wilde and Michael C. Scully. Software reconnaissance: mapping program features to code. Journal of Software Maintenance, 7(1):49–62, 1995.

[XP06] Tao Xie and Jian Pei. Mapo: mining api usages from open source repositories. In MSR '06: Proceedings of the 2006 international workshop on Mining software repositories, pages 54–57, New York, NY, USA, 2006. ACM.

[YT07] Annie T. T. Ying and Peri L. Tarr. Filtering out methods you wish you hadn't navigated. In eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, pages 11–15, New York, NY, USA, 2007. ACM.

[ZJ07] Charles Zhang and Hans-Arno Jacobsen. Efficiently mining crosscutting concerns through random walks. In AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, pages 226–238, New York, NY, USA, 2007. ACM.

[ZZL+06] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. ACM Trans. Softw. Eng. Methodol., 15(2):195–226, 2006.

[李08] 李昂. Freebsd-7 内核 malloc 原代碼分析. Published in `lllaaa.cublog.cn`, January 2008.