

## Chapter 6: Supporting Queries on Source Code

### 6.1 Overview

A conclusion from the previous chapter is that software maintainers seek information to complete a particular task. To this end, they search source code for elements from which to build conceptual models of how the software operates. One tool that is frequently used for this purpose is the UNIX regular expression matching utility, `grep`. Adding semantic awareness of a programming language to `grep` would help support bottom-up code comprehension models. In this chapter, search tools and source code analysis tools are examined as possible role models for a “semantic `grep`”.

We make a distinction between tools to search source code and tools to analyze source code. Analysis tools are employed to extract facts from a software system. These tools will be examined for the approach they take to building a factbase. Search tools are employed to make specific queries, and are often applied to factbases. Although analysis tools can be tuned to answer specific questions, software maintainers’ knowledge acquisition strategies are more oriented towards searching. As goal-directed information seekers, they ask questions and look for answers that are necessary to complete the task at hand; they are not trying to build an encyclopedia about the software system.

This chapter begins with an examination of `grep` based on work done by Singer and Lethbridge [Singer97b]. The remainder of the chapter examines different approaches to searching or analysing source code. Some of the software tools that we consider are specifically for working with source code, while others are tools for general purpose searching or information retrieval. Only relatively lightweight tools were considered because we are building a tool that fits cleanly with the PBS paradigm of lightweight tools. As a result, tools such as Rigi [Müller93], SNIFF+ [SNIFF+96], and CIA [Chen90] were not included in this survey.

Among the general purpose search tools that will be examined are additional members of the `grep` family, `cgrep` [Clarke96], `sgrep` [Jakko95], and `agrep` [Wu92]. Two environments for searching source code, `tksee` [Singer97a] and `SCRUPLE` [Paul94], are included in the survey. Finally, language-independent approaches to extracting structural information from source, `LSME` [Murphy96] and `TAWK` [Griswo96] are discussed. Similar to the analysis performed on `grep`, the strengths and limitations of these tools will be examined. These attributes will provide suggestions for the design of a semantic `grep`.

## 6.2 `grep`

The `grep` tool is a UNIX utility that performs regular expression matching over files on a line-by-line basis. The success of this tool is evident in the family of tools that it has spawned. The `egrep` variant does extended regular expression matching, `fgrep` does fast string matching, `zgrep` searches in zipped files, and there are others, `cgrep`, `sgrep`, and `agrep` that will be described in Sections 6.3-6.5. In our survey, 47 out of 69 respondents used `grep` to search source code. The prevalence of this tool was also noticed by Singer and Lethbridge [Singer97b] and they make a number of observations about `grep` as a model for a program comprehension tool. A selection of the strengths and limitations they observed along with our own observations are presented in Sections 6.2.1 and 6.2.2.

### 6.2.1 *Strengths of `grep`*

In this subsection, the positive aspects of `grep` are described. Attributes PG1-PG5 are taken from Singer and Lethbridge [Singer97b] and PG6-PG8 come from our own experiences and analyses.

#### **PG1. Success, little cost of failure, and understanding of limitations = trust.**

The `grep` tool excels at performing a specific task. Consequently, it is easy to specify a search and the results are returned quickly, frequently with a relevant match. When the search fails, little time or cognitive effort was wasted. Since `grep` is a small utility, users can understand its many limitations.

**PG2. Command-line interface.**

With its command-line interface, `grep` is able to fit with the other UNIX utilities, and by extension, users' interface style. Furthermore, `grep` can be included easily in scripts and macros to automate repeated tasks.

**PG3. Straightforward specification**

The only required arguments for a `grep` search are a target pattern and a search domain, all other information is optional. The target pattern can be a simple string or a regular expression. The search domain is one or more files or directories that can be specified through file name expansion by the operating environment.

**PG4. Results displayed in “parallel”**

Since `grep` acts as a filter, all matches to the search patterns are displayed, as they are found. These results can be scanned quickly for the relevant match. In contrast, many editors display the results in sequence, stepping the user from match to match inside a file.

**PG5. Scaffolded**

Regardless of how much experience a user has had with `grep` or the software system being searched, the user will be able to obtain results from `grep`. The user does not need to learn regular expression syntax and `grep` command options in order to use the tool. This knowledge can be acquired as the user feels the need, and as a result there is a smooth transition from novice to expert.

**PG6. Portable and flexible**

Programmers can use `grep` with whatever software systems they are working on. Software maintainers can bring their skill with this tool to any project, in any programming language. While `grep` is primarily a UNIX tool, implementations are available on other operating systems.

**PG7. Little overhead**

No indexing of the search domain is needed. Users are not required to generate an index or factbase of the source before using `grep`. Sometimes this requirement alone is enough to dissuade a software maintainer from experimenting with a tool.

**PG8. Responsiveness**

Users don't have to open a new window and wait for it to initialize, just to perform a "simple" search. The time spent waiting is too disruptive if a programmer is trying to sustain a complex train of thought.

**6.2.2 Limitations of `grep`**

Singer and Lethbridge also listed a number of limitations of `grep`, which indicate possible areas of improvement. A subset of their observations is presented here as NG1-NG5 and we add observations NG6 and NG7.

**NG1. Interpretation of output requires effort**

When `grep` returns a large number of matches, it is sometimes difficult to find the most relevant one. The string matching the search target may be difficult to find and the search results themselves may need to be searched. Each match consists of a single line, which often does not provide enough context to interpret the match.

**NG2. No near searches**

Matches can not be approximate and must be exact according to regular expression rules. If there is a spelling mistake in a search target, there is no facility in `grep` to deal with this.

**NG3. No semantic searches**

The `grep` utility treats all input as straight text. When searching source code, there is no way to limit the search domain, for instance to identifiers or comments.

**NG4. No memory**

Search targets or contexts are not stored and cannot be revisited for refinement or modification. Saved sequences with annotations could be useful for teaching software immigrants about a software systems. The command history of a shell can help, but is not the complete solution.

**NG5. No browsing**

The search results can't be browsed like hypertext, for example clicking on a matched line to display the entire file.

**NG6. Fixed “hit” size**

Searches using `grep` return whole lines that match. Sometimes the desired unit of return or match may be larger than a line, such as a module or function definition. At other times it may be smaller, such as a function parameter, an identifier, a specific column of a line, or a literal string.

**NG7. Sensitive to whitespace**

Many programming languages are insensitive to whitespace, in that the number, or type, of spaces between tokens is not significant. Searches in `grep`, however, are sensitive to whitespace. For example, the expression “`add();`” will not match “`add ();`”. Although, it is possible to write an expression that is insensitive to whitespace, but to do so accurately would require more effort than most users are willing to expend. Expressions such as `add[[:space:]]*( ) ;` or `add[ \t ]*( ) ;` require both a non-trivial amount of knowledge of regular expressions and time to type out the specification.

**6.2.3 Analysis of *grep*'s Attributes**

Clearly, `grep` acquires many of its strengths and limitations from being a UNIX utility. It would be difficult to address some of its shortcomings without modifying this interface and compromising some of its strengths. For example, browsing through pointing and clicking is difficult to achieve within a command-line interface. Adding semantic awareness would

require `grep` to parse its input, either when the search is invoked or beforehand to build an index. Regardless of the option chosen, at least one of speed, flexibility, and portability would be affected. Another factor to consider when adding semantic awareness to `grep`, is the syntax needed to query the different elements. This problem is further compounded if the user wants to search in units other than files, such as modules or subsystems. A syntax that supports queries on these search elements needs to be added.

### 6.3 `cgrep`

One of `grep`'s limitations is that matches must appear on a single line. The `cgrep` (context `grep`) tool addresses this shortcoming by treating the input as a character stream and interpreting the newline character as ordinary text, so that it can return matches with arbitrary sizes [Clarke96]. It uses a shortest-match algorithm and allows matches to overlap, but not nest, which means the program reports “every substring of the input text that matches the regular expression and that does not itself contain a matching substring.” This change results in a faster algorithm and is motivated by experience with structured text databases.

### 6.4 `sgrep`

The `sgrep` (structured `grep`) tool performs searches on text files or streams that have structural markup, such as e-mail, USENET news, source code, HTML, bibliographies, etc [Jaakko95]. Searches return *regions* that are delimited by strings or tags. Regions can be arbitrarily long, overlapping, or nested. Although `sgrep` is essentially a command-line tool, there is a `tcl/tk` graphical user interface, `sgtool`, available.

The syntax for specifying queries in `sgrep` is based on the GCL query language taken from Clarke [Clarke95a]. The command-line specifications for `sgrep` searches can be quite complex, but the tool can handle macros to simplify frequent targets. However, accompanying this specification complexity is a corresponding ability to handle complex search targets. For example, the query ‘show the if-statements containing the string “access” in their condition in the `setOptions` function of the source files \*.c’ is specified as:

```
sgrep `if" not in("/" quote "*" / or ("\\n#" .. "\\n")) \\
```

```
( "(" .. ")" ) containing "access" \\
  in ( "setOptions(" .. ( "{" .. "}" ) ) \\
  .. ( "{" .. "}" or ";" ) ' *.c
```

With the following macro definitions,

```
define (BLOCK, ( "{" .. "}" ))
define (COMMENT, ( "/*" quote "*/" ))
define (PPLINE, ( "#" in start or "\\n" _ . ( "\\n" or
  end) ))
define (IF_COND, ( "if" not in (COMMENT or PPLINE) ..
  ( "(" .. ")" ) ) )
```

the above command can be simplified to:

```
sgrep -p m4 -f c.macros -e `IF_COND containing "access" \\
  in ( "setOptions(" .. BLOCK ) .. (BLOCK or ";" ) ' *.c
```

Even with these simplifications, the search specification syntax is quite complex. Users would probably have to add aliases and scripts to make the tool more usable.

## 6.5 `agrep`

The `agrep` tool [Wu92] is another `grep` variant with three modifications. It allows approximate matches by permitting a user-specified number of substitutions, insertions, or deletions. Second, instead of restricting ‘hits’ to single lines, `agrep` can return matching records, such as entire email messages. Finally, it allows the logical combination of patterns using AND or OR.

Approximate matching would fulfill some of the suggestions from the source code searching survey that asked for fuzzy searches. It could be used in situations when the maintainer didn’t know the exact name of an identifier. As noted above by Singer and Lethbridge, while the wildcards in regular expressions allow some degree of flexibility in the matches, they do require the search to be specified accurately.

## 6.6 `tksee`

Singer et al. addressed limitations of `grep` with `tksee` (Software Exploration Environment with a `tk` interface) [Singer97a]. It is essentially a semantic `grep` inside a graphical user interface. Search targets can be regular expressions, strings, identifiers, function and variable definitions and uses, macros, etc., and can be entered in a text box, initiated by a pointer, or

selected from a menu. Matches and their attributes can be browsed and searched. Search history can be browsed by clicking and search sequences or sets can be saved.

The architecture of `tksee` is similar to that of `PBS`. Its back end is a fast, object-oriented database containing facts extracted from the source code. The factbase is language independent and contains some clustering information. Data is passed between tools in a Tuple Attribute Language variant, `TA++`. Clients must connect to the server, sometimes over a network, to access the factbase.

## 6.7 SCRUPLE

In `SCRUPLE` [Paul94], there are elements of both `sgrep` and `tksee`. To find matches, `SCRUPLE` parses the source code when the tool is invoked, and there are versions of the tool that work with `C` and `PL/AS`. Users specify search targets using a pattern language, and the results are displayed to standard output. Paul and Prakesh initially designed `SCRUPLE` to be a command-line tool that extended `grep`, but later added an X-windows graphical user interface. In the GUI, when a search is completed, the user is walked through the code to each match.

```

1 $t $f_decl()
2   {*
3       @*
4       @{* #{* $f_call (#*) *} *}
5       @*
6   *}
```

**Figure 6.1: Call Graph Extractor for C in SCRUPLE from [Griswo96]**

In `SCRUPLE`'s pattern language, there are various wildcard symbols for different syntactic entities. There are generic wildcards, wildcards for sets, and named wildcards. For example, a declaration is represented by “`$d`”, a set of arbitrary declarations is “`$*d`”, while the declaration of entity “count” is “`$d_{count}`”. Other syntactic entities for which there are wildcards are types, variables, functions, expressions, and statements. The query, “find all declarations of the variable `x`” is specified as “`$t x;`”. An example of a more complex query is “`$t $f_x <xmax> ($v*) { @* }`”, which means “find all functions that have

references to the identifier xmax.” A call graph extractor can be written using SCRUPLE in just 6 lines, as shown in Figure 6.1. While this specification language is quite powerful, it is programming language specific. In other words, a new pattern language is designed for each programming language to be searched.

## 6.8 LSME

A “source model” is a view of a software system, for example call graphs, file dependencies, etc. and is usually extracted by parsing the source code. In contrast, LSME (Lexical Source Model Extraction) [Murphy96] extracts source models without a language-specific parser. Conceptually, LSME is much like awk [Aho79] in that it scans through the source code doing pattern matching of constructs and regular expressions, and executes commands as matches are found. Like awk, LSME specifications are conceptually closer to a script or program, than a command-line utility.

The LSME tool achieves language independence by requiring the user to specify the match syntax. Running the specification:

```
[ <type> ]<funcName>\([ {<arg>}+ ]\)[ {<type>, argcDecl> ; }+ ]\{
    <calledFcnName> \([ {, param>}+ ]\ )
    @write ("calls", fcnName, calledFcnName) @
```

on Kernighan and Ritchie style C source code [Kernig78], will find all function calls and writes a tuple for each one. A call graph extractor can be written for LSME in 30 lines, as shown in Figure 6.3. While LSME’s lexical approach is responsible for both its main advantage and disadvantage. The tool may not accurately find all matches, but it is portable across languages. The LSME system has been used to extract source models from C, C++, CLOS, Eiffel, and TCL source code.

```

1  comment /* */
2
3  [<type>]<fn>
4  @ if kywdq(fn) | opq(fn) then fail @
5  \ ( [ {<param>}+ ] \) [ { {<atype>}+ ; }+ ] \ {
6
7      <cn>
8      @ if kywdq(cn) | opq(cn) then fail @
9      \ ( [ { <arg> [ , ] }+ ] \)
10 @ writeCall (fn , cn) @
11
12 procedure writeCall(fn, cf)
13   static idch
14   initial idch := (&ucase ++ &lcase ++ &digits ++ '_' )
15
16   realfn := (fn ? (tab(upto(idch)), tab(0)))
17   realcf := (cf ? (tab(upto(idch)), tab(0)))
18   return write(realfn, " ", realcf)
19 end
20
21 # true if a keyword
22 procedure kywdq(nm)
23   return nm == ("if" | "while" | "Switch" | "for" | "typedef")
24 end
25
26 #true if an operator (approximate)
27 procedure opq(nm)
28   return any('\?;:+-*/%!=|<>', nm) &
29   (*nm == 1 | *nm == 2 & any('\+=*/%&=|<>', nm[2]))
30 end

```

**Figure 6.2: Call Graph Extractor for C in LSME [Griswo96]**

There is another awk variant, TAWK, that takes an approach similar to LSME. There are some technical differences, but the key distinction is that it is language-dependent [Griswo96].

## 6.9 Comparison of Tools

Although the tools have conceptual differences in how they approach the problem of manipulating source code, it is possible to compare their features. Table 6.1 summarizes the features offered by the tools. The basic differences are that `grep` and its variants are simple UNIX utilities that behave as filters on an input stream, while `tksee`, `SCRUPLE`, and `LSME` were designed to analyze or search source code. This is particularly evident in the last two tools in the specialized query specification languages used.

Tool Name	grep	cgrep	sgrep	agrep	tksee	SCRUPLE	LSME
Tool Type							
Searching	X	X	X	X	X	X	
Analysis						X	X
User Interface							
Command line	X	X	X	X		X	X
GUI			X		X	X	
Approximate matching	*	*	*	X	*	X	†
Semantic searches					X	X	
Source language independent	X	X	X	X			X
Requires factbase or parsing					X	X	
Query language	regular exp.	regular exp.	GCL-based	regular exp.-based	regular exp. and navigation	pattern language tailored to source language	own pattern language with ICON

\* wildcard matching only

† only at the lexical level

**Table 6.1: Comparison of Tool Characteristics**

## 6.10 Lessons Learned

We made a series of design decisions for our own source code searching tool based on the user studies and tool analyses from this and earlier chapters. These decisions were influenced by our goals as described in Chapters 4 and 5, `grep`'s strengths and limitations, and the various query mechanisms utilized by the tools.

- **Use an existing language to specify searches**

Except for `cgrep` and `agrep`, all the tools examined used their own query or pattern language to specify searches. (In the case of `tksee`, complex searches are accessed through the GUI.) There are three lessons that can be learned. 1) In order to support

semantic or structural searches, the basic `grep` command syntax would not be sufficient; 2) The last thing the world needs is another query language. 3) Many of the languages used in the tools examined have well documented syntax and semantics; one has an algebraic basis. Reusing a query languages would take advantage of the work already done on its formal specification.

- **Start with a command-line search tool**

Many of `grep`'s strengths, such as ease of use, compatibility with operating environment, etc., arise from its command-line interface. A semantic `grep` that does not retain this aspect of the `grep` paradigm is unlikely to be adopted by software maintainers.

- **Add a graphical user interface later**

Both `SCRUPLE` and `sgrep` started out as command-line tools and later a graphical user interface was grafted on top of them. This seems to be a reasonable course of action to follow, since the GUI can be added when the tool is integrated with the Software Bookshelf.

- **Maintain language-independence**

A tool becomes much more powerful when it isn't tied to a particular programming language, as is evident in the `grep` family and `LSME`. Familiarity with a tool or pattern language becomes more valuable because of its portability. The PBS tools achieve language-independence by using a common factbase. A search tool that is part of PBS could do the same.

Further examination of these tools led to the selection of `GCL` (generalized concordance lists) as the query language for the search tool. The syntax and semantics of `sgrep` is based on `GCL`. There were three main reasons for this decision: it was designed to work with structured documents, of which program source is an example, and it has an algebraic basis for the grammar [Clarke95a]. Since `GCL` is a general-purpose query language, the search space need not be limited to source code. Written documentation, `HTML` pages, and any other structured material can be searched. Later implementation of our source code searching

tool could easily be adapted to include these documents as well. With this additional capability, PBS becomes more like a traditional information repository. The GCL language is described in the next chapter, along with a design of a the search tool `grug` (**grep using GCL**) and how it fits with PBS.

### **6.11 Summary**

In this chapter, a number of searching and source code analysis tools were examined for their approaches to solving the problem of extracting semantic information from source code. The general-purpose search tools were investigated for their interface and the syntax used to specify searches. The analysis tools were considered for the mechanisms used to extract facts from source code. Based on observations of these tools, a number of design decision were made. An existing query language, GCL was chosen to specify patterns in the search tool `grug` (**grep using GCL**). Initially, `grug` will be a command-line tool and a graphical user interface will be added later. Finally, the design of `grug` should not restrict it to a specific programming language. In the next chapter, these design decisions are be applied to the specification of `grug` and Searchable Bookshelf.