

Copyright 1996 IEEE. Published in the Proceedings of the IEEE Fourth Workshop on Program Comprehension (WPA-96), March 29-31, 1996, Berlin, Germany. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.

Fast, Flexible Syntactic Pattern Matching and Processing*

William G. Griswold and Darren C. Atkinson
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{wgg,atkinson}@cs.ucsd.edu

Collin McCurdy
Department of Computer Science
Rice University
Houston, TX 77251-1892
cmccurdy@cs.rice.edu

Abstract

Program understanding can be assisted by tools that match patterns in the program source. Lexical pattern matchers provide excellent performance and ease of use, but have a limited vocabulary. Syntactic matchers provide more precision, but may sacrifice performance, retargetability, ease of use, or generality.

To achieve more of the benefits of both models, we extend the pattern syntax of AWK to support matching of abstract syntax trees, as demonstrated in a tool called TAWK. Its pattern syntax is language-independent, based on abstract tree patterns. As in AWK, patterns can have associated actions, which in TAWK are written in C for generality, familiarity, and performance. The use of C is simplified by high-level libraries and dynamic linking. To allow processing of program files containing non-syntactic constructs, mechanisms have been designed that allow transparent matching in a syntactic fashion. So far, TAWK has been retargeted to the MUMPS and C programming languages.

We survey and apply prototypical approaches to concretely demonstrate the tradeoffs. Our results indicate that TAWK can be used to quickly and easily perform a variety of common software engineering tasks, and the extensions to accommodate non-syntactic features significantly extend the generality of syntactic matchers.

1 Introduction

Understanding software well enough to modify, reuse, or extend it is notoriously expensive [1]. A large part of the difficulty is understanding the semantic relationships between program components that are syntactically far apart from each other. Typical questions a programmer might ask include: Where is this function called? Where is this variable modified? Is this flag always reset before entry to the initializer? Where is this record allocated and initialized?

Programmers are typically content to get answers to these questions using tools that do not use expensive data flow analysis. Lexical tools support queries using regular expression matching. Syntactic tools support queries using context-free matching. In the more flexible tools, a programmer-defined (i.e., user-defined) action can be performed on each match. The action language, since it is more powerful (albeit less concise) than the matching language, can be used to perform additional matching, process the matched component, maintain data structures, or provide specially formatted output.

Programmers typically want precision, speed, ease of use, and generality in a code matching tool:

- Precision is important because, for one, if many false matches are returned, the tool is not doing its job of filtering out irrelevant code.
- Speed is essential, for one, because the software being processed may be large—in the millions of lines—and the software may not be structured so that a complete search can be performed on just a subset of the code. In addition, a programmer may need to perform several related queries to answer a particular question.
- Ease of use is important because a query is often used only a few times. If it is difficult to write a pattern, then the time required detracts from other programming tasks.
- Generality is valuable because over time a programmer may need to perform a wide variety of searching tasks on a software system.

In this paper, we describe techniques for improving the balance between these issues for syntactic tools, which typically demonstrate superior precision, but tend to be inferior to lexical tools in the other dimensions because of the added complexities and constraints of the syntactic matching paradigm. In particular, we describe: memory management techniques for improving speed; the use of a familiar compiled programming language as the action language

* This work supported in part by NSF Grant CCR-9211002, UC MICRO Grant 94-053, SAIC, and a Hellman Fellowship. Collin McCurdy completed this work while at UCSD.

for speed, generality, and ease of use; a powerful and concise matching language for ease of use; and support for syntactic matching of text-based macros to increase generality. We close the paper with an evaluation of speed and usability. Throughout the paper we use the extraction of call-graph information as a prototypical example. Although not especially complicated, call-graph extraction is an appropriate problem for all of the approaches discussed, and succinctly reveals the strengths and weaknesses of each.

```

[1]  #!/usr/local/bin/mawk -f
[2]
[3]  # initialize regular expressions
[4]  BEGIN {
[5]    WS = "[ \t\n]*"
[6]    ID = "[a-zA-Z0-9_]+"
[7]    IDCC = "[^a-zA-Z0-9_]"
[8]    CALL = ID WS "\("
[9]    DEFN = ID WS "\{[^}]+\}"
[10]   KYWD = "for|while|do|switch|if|typedef"
[11]   KYWDIC = IDCC "(KYWD)" IDCC # token
[12]   OUTSIDE = 1
[13]   RS="\n\n" # rec. sep. is blank line
[14] }
[15]
[16] # When inside a procedure definition
[17] !OUTSIDE {
[18]   s = $0
[19]   while (s != "") {
[20]     if ((start = match(s, CALL)) {
[21]       match (substr(s, start), ID)
[22]       len = RLENGTH
[23]       if (!match(substr(s, start, len),
[24]                 KYWD))
[25]         print fdecl, ": ",
[26]             substr(s, start, len)
[27]       s = substr(s, start + len)
[28]     } else break
[29]   }
[30] }
[31]
[32] # When outside a procedure definition
[33] OUTSIDE {
[34]   if ((start = match($0, DEFN)) &&
[35]       !match($0, KYWDIC)) {
[36]     match (substr($0, start), ID)
[37]     fdecl = substr($0, start, RLENGTH)
[38]     OUTSIDE = 0
[39]   }
[40] }
[41]
[42] # outside true when line starts with ")"
[43] /\n}/ { OUTSIDE = 1 }

```

Figure 1: A simplified MAWK call graph generator for C. An action is enclosed in top-level braces. A pattern is any legal MAWK expression. A regular expression is a string or a literal offset by slashes. Variable \$0 contains the current input record. RLENGTH is a global set by match to record the length of the matched substring.

2 Lexical tools

Lexical tools are fast, flexible, and robust. The regular expression matching algorithms used by these tools tend to require little space and are computationally simple. Regular expressions are also usually not difficult to write or understand, and are often short enough to fit on a single line. Because of these properties a programmer can quickly formulate a query and try it out.

Lexical tools are also flexible in that they can seamlessly accommodate the non-syntactic aspects of software such as textual macros, conditional compilation, comments, and external files. These tools are also robust because they are insensitive to syntactic errors or variants in the code.

On the downside, the simplicity of regular expressions often does not allow a programmer to express the desired query, even when programming conventions are exploited. The exploitation of programming conventions, unless they are widely adhered to, limits the reusability of queries.

2.1 The AWK family of tools

The most well-known examples of lexical matching tools are GREP and AWK. GREP provides basic line-by-line regular expression pattern matching [2], and AWK extends that model by supporting programmer-defined actions on the matches written in a C-like language [3]. An input line—called a *record*—is divided into fields (i.e., tokens) according to a *field separator pattern*; each field is addressable by an integer. The unit of input can be customized by specifying a *record separator pattern*. MAWK is a slightly more expressive AWK, most notably supporting patterns as data, not just constants [4].

As an example, the MAWK script shown in Figure 1 is one possible approach to extracting call-graph information from a C program. The BEGIN clause (lines 4–14), which is executed on start up of the tool, defines the basic patterns for matching portions of function calls and declarations. The record separator is defined as two successive returns (line 13), which assumes that the programmer always separates procedure declarations by a blank line. The same assumption is made about the separation between the procedure header and its body. Since function calls and function prototypes look similar in C, this script maintains a state variable OUTSIDE (lines 12, 17, 33, 38, 43) that keeps track of whether the matcher should be looking for function calls or not. OUTSIDE starts out as true (1), and is set to false (0) when a function declaration is matched. OUTSIDE is then reset to true when a lone right brace is found to appear at the beginning of a line—a programming convention. Since the matching predicates only return the first match, the actions must match all function calls by repeatedly matching the first call and deleting the preceding text along with the matched function identifier (lines 17–30). Consequently, the actual matching is performed in the actions, not in the

patterns. This is not too problematic since patterns can be used anywhere in an AWK program. However, much of the declarative power of the patterns has been lost by using a destructive loop. Even though the matcher cannot correctly match parentheses, nested calls can be correctly identified, since the pattern only requires that an opening parenthesis is found after an identifier (lines 8, 20). To avoid matching conditional statements as calls or function declarations, matches whose identifiers are C keywords are rejected by an added test (lines 10, 11, 24, 35). Finally, since fields cannot be used to extract portions of the match, the function identifier is extracted by performing an identifier match after the call is located (line 21).

Separators delineate fields (and records) by defining what *is not* in them (e.g., whitespace). In contrast, programming language tokens are typically defined by first and follow sets [5], which is a characterization of what *is* in them. Consequently, fields are usually not a practical way to access portions of a matched record during processing (e.g., if whitespace does not delineate every token). Also, the program text matching a record separator pattern is discarded and cannot be used in a match. For instance, the record separator pattern "[; }]" matches the end of all basic and compound statements in C, but since the separators are discarded, it is difficult to determine which type of statement is being delineated.

A related problem is that the only effect of a successful match is to return the starting character index and length of the match. There is no declarative way to save a portion of a matched expression in a variable (e.g., the name of an identifier in a function call). Instead, after a match is successfully performed, another match must be performed to extract any needed data. The MAWK call-graph extractor is designed so that the needed identifier is located at the beginning of the match (line 20), so only one additional match must be performed (line 21).

With some effort, the script could be generalized. For example, the script now assumes that no calls appear in the text matched with a function declaration pattern. This assumption could be removed by putting the call matching functionality in a function and calling that function in both the header and body matching actions. Also, matching of white space could be generalized by extending the pattern in the WS variable with a regular expression for comments.

2.2 LSME

Murphy and Notkin's approach to source model extraction is specifically designed for quickly building high-level models of source code to aid high-level decision making [6]. Consequently, they chose to sacrifice some precision and accuracy to facilitate quickly developing an abstraction of the code. Their approach also makes several adjustments relative to other lexical approaches to make the matching of pro-

```

[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]
[13] procedure writeCall(fn, cf)
[14]   static idch
[15]   initial idch := (&ucase ++
[16]                   &lcase ++ &digits ++ '_' )
[17]
[18]   realfn := (fn ? (tab(upto(idch)), tab(0)))
[19]   realcf := (cf ? (tab(upto(idch)), tab(0)))
[20]   return write(realfn, " ", realcf)
[21] end
[22]
[23] # true if a keyword
[24] procedure kywdq(nm)
[25]   return nm == ("if" | "while" | "switch" |
[26]                "for" | "typedef")
[27] end
[28]
[29] # true if an operator (approximate)
[30] procedure opq(nm)
[31]   return any('?:;+-*/%!=|<>', nm) &
[32]           (*nm == 1 |
[33]            (*nm == 2 &
[34]             any('+-*/%!=|<>', nm[2])))
[35] end

```

Figure 2: A basic C call-graph extractor written in LSME.

gram text easier. Their approach is demonstrated in LSME (Lightweight Source Model Extractor).

First, LSME augments AWK's notion of a separator by also treating any single character literal appearing in a pattern as a token, regardless of the separator rules. This extension yields better tokenization since single character tokens like parentheses are frequently used without whitespace. But because whitespace is discarded as in a parser, lexical conventions based on indentation cannot be exploited.

Unlike AWK, LSME matches a pattern against an entire file, which allows it to scan over any syntactically complete unit. Consequently, to avoid both spurious matches and performance problems due to too many active matches, LSME employs heuristics to prune partial matches. For example, the length heuristic limits the length of a match. The assumption is that a meaningful match is unlikely to span an arbitrarily long piece of text.

The LSME script shown in Figure 2 is a call-graph extractor for C. It is shorter and more declarative than the MAWK script. As for syntax, a name appearing in angle brackets (such as `cn` on line 7) matches a token in the input, which is

bound to the name for use in actions. A literal token consisting of special characters is escaped by the backslash character. A pattern delimited by square brackets is optional (such as [`<type>`] on line 3), and a pattern delimited by curly braces followed by a plus (lines 5, 9) matches the pattern to one or more consecutive elements. Actions, written in Icon [7], are delimited by @ and can be placed anywhere in the pattern (lines 4, 8, 10). The first two actions apply additional criteria to the matched tokens and reject them with `fail`, if necessary. The code following the pattern is also Icon (lines 13–35).

This script contains two patterns, a function declaration pattern (lines 3–5), and a call pattern nested within the declaration pattern (lines 7–10). The nesting means that the call pattern is active only if the declaration pattern has matched. Like the MAWK script, it makes assumptions to keep the script simple and provide precise matching. For instance, when the call pattern matches a piece of text, the arguments matched in the text are precluded from matching calls as well. Thus in the expression `f(g(x))`, the call to `g` would be overlooked. With some effort, the argument pattern can be extended to match calls in the argument list, although arbitrarily deep nesting cannot be handled without some sort of procedural abstraction facility that supports recursion.

Interestingly, LSME has minimal regular expression capabilities: a token pattern is either a literal or a contiguous sequence of non-whitespace. Consequently, actions must be used to reject parenthesized expressions and special statements in the call script (lines 4, 8).

3 Syntactic tools

Syntactic matching tools have the advantage of precisely recognizing and encoding the lexical and syntactic structure of the program with no burden on the tool user. Tools such as REFINE [8], SCRUPLE [9], A* [10, 11], and GENOA [12] fall into this category.

SCRUPLE is a prototypical syntactic matching tool.¹ To perform a query, the programmer types in a program fragment, substituting syntax-specific wildcards for expressions, statements, or declarations where appropriate. The wildcards are specified using characters that do not conflict with the the source language’s tokenization rules. Most wildcard matches may be bound to variables and matched again later in the pattern. Using the actual language syntax in patterns makes learning SCRUPLE relatively simple, and patterns are easy to understand. A SCRUPLE pattern is parsed by using an extended language parser, and matching

¹In fact, A* is more appropriate for this study, but this came to the authors’ attention close to the time of publication of this paper, and could not be included. A* is an adaptation of AWK that builds concrete parse trees of programs for processing by an interpreted language. It provides YACC-like patterns, but without wildcards. It also provides novel control regimes for flexible traversal of the parse tree. A* does not compute attributes such as symbol table information, although it is possible in principle.

is performed on the abstract syntax tree (AST) of the program. SCRUPLE uses a state machine and stack to support top-down matching and backtracking on failed matches.

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

Figure 3: A general pattern for call-graph extraction using SCRUPLE. All successful matches (pairs of `decl` and `call`) are printed to an output window in the tool.

The pattern shown in Figure 3 is a call-graph extractor for C written in SCRUPLE. It is significantly shorter than the script written using Murphy and Notkin’s approach. However, it is also somewhat more cryptic. Unlike the lexical tools’ call-graph patterns, SCRUPLE’s declaration and call patterns must be part of a single syntactically complete pattern. Consequently, extra patterns must be used to “glue” the two together, specifying everything that can appear “between” the declaration and the subsequent calls. For instance, the wildcard patterns `@*`, `{* *}` and `#{* *}` are used to state that a function call can appear inside any (perhaps nested) statement, inside a (perhaps nested) expression of the body of the matched function declaration. If only calls or only declarations were being matched, these glue patterns could be omitted. However, since the relationships between program components are of central concern to a programmer, these “compound” patterns will occur frequently in SCRUPLE. Named wildcards help perform other parts of the match: `$t` (line 1) matches a type, and `$f` (lines 1, 4) matches a function identifier. The variables `decl` and `call` are bound to the results of the matched function declaration and call, respectively.

A functional shortcoming of the call-graph pattern is that the return type of the function—matched by `$t`—is required. Another pattern would be required to match a function declaration lacking a specified return type. Although omitting the type wildcard would introduce a reduce–reduce parsing ambiguity between function declarations and function calls when parsing the pattern, the function identifier could be split into two classes, declaration and call, to resolve the ambiguity.²

As shown in the call-graph extractor, using concrete syntax for describing patterns has some limitations. For one, a pattern that spans a lot of different syntax requires a literal or syntax-specific wildcard for each part of the syntax, even if most parts are not of interest. In addition, the matching language lacks wildcards for optional constructs (e.g.,

²Santanu Paul, Personal Communication.

an optional initialization), although list or set wildcards may match nothing. Also, general properties tend to be hard to express in the concrete syntax. As an example, writing a concrete syntax pattern to match any kind of loop in a C program is hard, since the syntax of each is significantly different: the conditional forms of the `for` and `while` have different syntax, and the `do-while` conditional comes at the end of the construct. Disjunction itself is hard to specify syntactically. For instance, SCRUPLE contains a disjunction operator for specifying *kinds* of statements to match. However, this is a departure from using concrete syntax, as one of the kinds is `do-while`, yet there is no such lexical token in C. This irregularity, although necessary, can make SCRUPLE harder to use and does not address the problem that the three loop bodies have different syntaxes so they cannot be subsequently matched in any general way.

After a pattern is applied to a program, the bound variables and their values are printed to an output window. SCRUPLE does not have an action language to process matched components, although it is possible. This could be powerful because the AST allows for more sophisticated processing than that allowed by lexical tools.

Although potentially fast, SCRUPLE makes a few key design decisions that impact performance:

- The pattern subject for matching is the entire program, allowing a pattern match to backtrack all the way to the beginning of the program being matched.
- The entire program's syntax tree is stored in memory, both to support arbitrary backtracking and to amortize the cost of multiple queries. This design decision is inappropriate for large programs. Their AST's can be hundreds of megabytes and exceed virtual memory, requiring adaptive methods for handling the AST [13, 14].
- The SCRUPLE designers noted that matching on groups of declarations should not be constrained by declaration order, since the declarations are independent.³ Consequently, SCRUPLE provides wildcards on *sets* of some classes of syntax. Although elegant, the designers note this can cause a combinatorial explosion in the number of matching possibilities. The added flexibility of an action language could avoid this dilemma.

4 TAWK: An alternative syntactic approach

To improve on the current state of the art we would like to have the precision of syntactic matching with the speed, ease of use, and generality of lexical matching. Our approach to achieving these goals has several components:

³In fact, in many languages, including C, the order of declarations is significant due to visibility rules and initialization.

- To improve performance we adopt the technique used by lexical matchers of *a priori* limiting the size of the program fragments being matched. We call this *granularity control*, and the programmer can specify the type of syntactic unit on which to match. For example, in C the granularity can be *program*, *file*, *top-level declaration*, or *statement*. This approach also positively impacts TAWK programming style.
- To improve the expressiveness of syntactic patterns, our pattern matching language is based on the programming language's AST, rather than its concrete syntax. The matched pattern as well as subcomponents can be bound to variables. Although potentially more cryptic than concrete syntax, the patterns can be more concise and more general. Some irregularities in the concrete syntax can be avoided in the abstract syntax.
- By using abstract syntax, the pattern matcher itself is language independent. No special effort is required to retarget it. The front end that builds and defines the AST must be retargeted, but this cost can be spread across tools other than the pattern matcher. TAWK has been retargeted to C and MUMPS.
- Rather than define a specialized action language for writing actions, an existing compiled language is used—in our case, C. This choice reduces the learning curve of the language, ensures good performance, and provides a full-featured language with a minimum of effort. Libraries and macros are provided that define the AST datatype and several aggregate types (e.g., tables). The actions are automatically compiled and linked, and dynamic linking is used to minimize runtime compiling costs.
- We extend our language parser and AST to accommodate classes of textual macros that have a syntactic appearance. In such cases the macro bodies and their invocations can be included in the AST and can be matched. Other cases are processed normally by the preprocessor.
- To increase the expressiveness of our pattern language, we provide pattern abstractions. Although not as powerful as MAWK's abstraction facilities, pattern abstractions provide most of the abstraction needed to support separation of concerns and reuse.

4.1 The TAWK matching language

Concepts. Like an AWK program, a TAWK program consists of a sequence of pattern–action pairs. Backtracking is performed within compound patterns to explore all possible alternatives until the entire pattern is matched or exhausted. Alternatives are explored left-to-right. Every pattern–action

pair is tried in turn from top to bottom in the sequence, so several patterns can match.

TAWK's pattern matching language was inspired by a somewhat different language called TGREP [15], designed for matching natural language represented in a tree form. We have adapted and extended TGREP's concepts to meet the needs of matching programming language text represented as an AST. The first concept is the notion of *node type*. A node in an AST has a type based on the kind of syntactic construct it represents. For instance, all nodes representing identifiers are of the `identifier` type in the C TAWK tool (called CAWK). A node can also have a *value*. For instance, a node for an identifier `f00` might have a symbol table entry for the identifier `f00`. Such an entry would be useful in an action so that tests regarding the scope of `f00` could be performed. Of course, referring to a symbol table entry in a pattern would be unwieldy, so a pattern trying to match on the value of an identifier node matches on the string name of the symbol table entry. A node representing a compound language construct may also have one or more *children*. For instance, the language expression `f00 + bar` might have a root node `+`, and two children `f00` and `bar`. In this case, `f00`'s *right sibling* is `bar`. Consequently, a node's type, value, children, or siblings are properties that may be matched in a pattern.

Granularity is also an essential, albeit implicit, part of a pattern. Unlike SCRUPLE, in which a pattern is applied to an entire program AST at once, a TAWK pattern is applied in turn to all subtrees of a certain type or property. For instance, a pattern may be applied to all top-level declarations. The subtree under current consideration is called the *current subject*. Pattern matching is *anchored*, meaning that a pattern must match the entire subject. When a pattern matches the current subject, it is passed to the action for further processing. Granularity control reduces backtracking by limiting the size of the matching subject, and provides some expressive leverage in writing patterns and actions.

The last important part of a pattern is how a successfully matched tree is communicated from a pattern to its action. First, the current subject is passed to the action in the variable `$$`. Also, as in SCRUPLE the pattern may contain named variables that are bound during matching.

Syntax. The basic syntax for a pattern–action pair is *expression* { *action* }. Either *expression* or *action* can be empty. An action with an empty pattern expression always matches. An empty action prints out the matched subtree.

An atomic pattern has syntax *type* : *value-expression* : *variable*. Most parts are optional, although a value expression must have a type so it can be interpreted unambiguously. A value expression is a literal to be exactly matched, or a regular expression of the form */regexp/* to match a substring of a node's value. A type can be complemented with

a leading tilde (`~`), but such a pattern cannot have a value expression, since it could be ambiguous. As an example, the pattern:

```
identifier: /~f/:$var
```

matches an identifier node for any identifier starting with the letter “f”. If such a match occurs for this expression, the node is bound to the variable `$var`.

There are three special atomic patterns. `BEGIN` and `END` match the first and last subtree inputs, respectively, for use in tasks such as initialization and printing of final output. They cannot be combined with any other pattern. The underscore character (`_`) matches any subtree.

A compound pattern either modifies the meaning of a pattern or combines one or more patterns. The syntax borrows from typical ASCII syntaxes for regular expressions, but with extensions to handle tree structure. In particular:

$e_1 e_2$	Matches either expression e_1 or e_2 .
$e_1 e_2$	Matches e_1 immediately followed by right sibling e_2 .
$(e\ c_1 \dots c_n)$	Matches a tree rooted at e with ordered children c_i . All children of e must be matched.
e^*	Matches zero or more consecutive siblings matching e ; e may be omitted.
e^+	Matches one or more siblings matching e .
$e?$	Matches zero or one occurrences of e .
$[e]$	Match a tree containing e as a descendant.
$< e >$	Matches an occurrence of e ; used to override the default precedence rules.

A compound pattern result may be bound to a variable using the same syntax as with atomic patterns. A pattern that matches a sequence binds the variable to the last-matched item, or to the null value if the list is empty.

As an example, we can use C call-graph recognition:

```
(declaration:FUNCTION:$fdef
 * [expression:FUNCTION:$fcall] *)
```

which matches every function call appearing in each function declaration, binding the results to the variables `$fcall` and `$fdef`, respectively. The stars allow the nested call pattern to match against the function body without the tool user having to know or specify all the parts of the function definition syntax. This style will result in the nested call pattern being tested against every component of the function definition, such as the result type. However, since only the body can contain function calls, this pattern is precise.

4.2 Abstraction

In order to facilitate writing complex queries, we provide a simple pattern abstraction mechanism. For reasons of expediency, our current implementation is based on the C preprocessor, CPP, although a syntactic mechanism would be preferable. Using CPP, the previous query can be rewritten using a simple macro to become:

```
#define FDECL declaration:FUNCTION
#define CALL expression:FUNCTION
(FDECL:$fdef * [CALL:$fcall] *)
```

If a macro such as the CALL macro might prove useful in another script, it can be saved in a file and included using CPP's `#include` mechanism. This mechanism can also be used to rename node types and values chosen by the designer that the tool user finds unintuitive.

4.3 Handling extralinguistic constructs

As a result of the widespread use of textual (non-syntactic) macros in languages like C, matching on code as it appears to the programmer is problematic for syntactic tools. Matching on the code after the expansion of macros is problematic as well, since this code may have little resemblance to the code the programmer understands as being “the code”.

A complete solution to handling textual macros is unlikely, since a macro preprocessor such as C's preprocessor, CPP, in fact defines a language that has a different, incompatible grammar than the language text in which it is embedded. The only general solution, perhaps, is to substitute a syntactic macro language [16, 17].

SCRUPLE does not cope with macros, but it may be able to in some situations. Since patterns are written in concrete syntax, they can be run through the preprocessor, turning the pattern into a form that can be applied to the postprocessed program text. Then the matches need to be mapped back to the original program text, which should be possible if the preprocessor, lexer, and parser keep track of positional information. One limitation is that if a parameter in a macro pattern does not accept legal C code as arguments, then wildcards cannot be used as an argument for that parameter in a pattern.

Any syntactic matching tool can cope marginally with conditional compilation by supplying a set of compilation flags to instantiate one version of the program. Matching multiple versions can be accomplished by providing alternative flag settings and rerunning the tool. Redundant matches between versions can be filtered in a separate phase. Although not efficient, it can yield precise and accurate results.

Since macros are often used to create abstractions that are essential to the form of the code, we felt that it was important to be able to perform matches on them. Our approach is to leave those macro invocations (with or without parameters) that are “language-like” in the code, unprocessed. More precisely, for a C program that contains CPP

macros, if a macro declaration's body is parsable as C code, then the declaration is included in the AST as a kind of “extended C”. Likewise, if an invocation of one of these “C-like” macros appears where a C expression may appear, then it is left as-is, rather than expanded. For instance, the first two macro declarations below are C-like expressions, but the third is not:

```
#define inc(a,v) ((a) += (v))
#define pi (3.14159)
#define begin {
```

Both C-like macro declarations and their uses may be matched like true C code. This approach is similar in power to the approach suggested for handling macros in SCRUPLE.

We have found that the great majority of macros are written in a C-like fashion. Also, the modified preprocessor is capable of reporting all macros that fail to meet the criteria, so the tool user can adapt a script as needed to match on the implementation of a macro, rather than its abstraction. Because all expansions of a macro are guaranteed to have the same essential structure, such matching can be accurate.

Our design of an AST for C puts both macro and function declarations under the `declaration` node type, so they can be matched uniformly. Macro invocations that look like function calls are represented as function calls in the AST, although they can be distinguished by checking attributes of the call identifier.⁴

Our approach to handling macros is implemented in a straightforward manner. We modified CPP to apply a C expression parser to each macro definition. If the parse succeeds, the definition is left in the code. Subsequent invocations of the macro in code files are ignored by CPP. We then extended the C parser to recognize C-like macro declarations and generate AST subtrees for macro declarations and invocations.

Our pattern for a call-graph recognizer can now be updated to include macro declarations as well as function declarations. One choice is to remove the `FUNCTION` constraint on the declaration node:

```
(declaration:$fdef * [CALL:$fcall] *)
```

The first part of the pattern will now match all declaration nodes, but since calls can only appear within function and macro declarations, the overall pattern has the desired behavior. This pattern has the (somewhat surprising) effect of matching macro invocations in static initializations of global variables, which have no apparent caller. A variant with a “more precise” declaration pattern appears in the next subsection.

⁴Many choices are possible in the design of the AST. Although we chose not to distinguish macro invocations in the AST, the subject was a matter of debate amongst the authors.

4.4 Actions

TAWK uses C for its action language. Using a compiled language like C provides performance, familiarity, and allows writing sophisticated actions, but introduces other problems. For one, the time to compile a script and link it to the TAWK application could slow development and execution. However, if a script is small, compilation time is insignificant, and the dynamic linking provided by many modern operating systems effectively eliminates linking costs. More problematically, C is low-level, requiring significant programming for even simple tasks. To address the first concern, TAWK provides a set of libraries and include files to effectively extend C. For instance, the AST library provides a high-level AST abstraction that includes AST access functions, iterators, traversals, and printers. Also provided are libraries for aggregate data types spanning several varieties of lists, sets, and tables. Consequently, simpler scripts can be written by making calls to these routines. For instance, a call-graph extractor can be written as follows:

```
[CALL:$fcall] {
    printf("%s: %s\n", FunctionName($fcall),
           CallName($fcall));
}
```

Note that the pattern only matches on calls. This simplification is possible, for one, because all calls must appear in macro or function declarations, and two, because the AST abstraction supports inherited attributes such as the current function definition. This example assumes—as do the other scripts in this paper—that the called function is denoted by a constant-valued function identifier. However, node types and attributes can be used in TAWK to recognize when the called function is computed by an expression or retrieved from a variable so the case can be appropriately highlighted in the output from the script.

In more complicated processing, aggregate data types can play a key role in keeping a script simple. For example, with a little extra effort, sets can be used to eliminate duplicates. The script in Figure 4 declares two global variables (lines 6, 7) to record what function or macro is being processed and what calls have been seen in it. By exploiting the fact that patterns are evaluated top-to-bottom, setting up and destroying the “called” set can be triggered by declaration patterns appearing both before and after the call pattern (lines 10, 26). This approach eliminates some complicated logic, awkward initialization, and more clearly separates the script into logical components for initialization, main action, and finalization. Note that the pattern macro FDECL (lines 2–3) is constrained to matching function and macro declarations by the use of node values and disjunction. This was chosen to avoid the spurious and perhaps costly creation and destruction of sets for other types of declarations, although it also has the effect of not matching the uses of macros in global static initializers.

```
[1]  %{
[2]  #define FDECL \
[3]  <declaration:FUNCTION|declaration:MACRO>
[4]  #define CALL expression:FUNCTION
[5]
[6]  String curCaller;
[7]  Set cSet; /* funcs called by curCaller */
[8]  %}
[9]
[10] FDECL:$fdef {           /* function BEGIN */
[11]   curCaller = FunctionName($fdef);
[12]   cSet =
[13]     SetCreate(StrCompare, StrId, NULL);
[14] }
[15]
[16] [CALL:$fcall] {
[17]   String called = CallName($fcall);
[18]
[19]   if (called && SetMember(cSet, called)) {
[20]     SetInsert(cSet, called);
[21]     printf("%s: %s\n", curCaller, called);
[22]   }
[23] }
[24]
[25] /* function END */
[26] FDECL:$fdef { SetDestroy(cSet); }
```

Figure 4: TAWK call-graph script that rejects duplicates.

4.5 Design and implementation details

A TAWK pattern is parsed and converted directly into a state machine that can be executed by the TAWK pattern interpreter. The interpreter uses techniques similar to SCRUPLE’s [9]. In particular, the pattern is converted into an augmented state machine that walks over the AST using *up*, *down*, *left*, and *right* moves and performs matching tests on the AST nodes as specified by the pattern. Variables are bound as successful matches are performed.

TAWK’s actions are merely C code with references to pattern variables added. A preprocessor replaces each pattern variable reference with a lookup in a table. Each action is then compiled as a separate procedure that is invoked whenever its associated pattern successfully matches an input AST.

TAWK is built on top of the Ponder toolset [14], which provides facilities for producing AST’s from code, as well as the libraries for AST’s and the other data types used in the actions. Retargeting TAWK to a new language requires retargeting Ponder and then initializing a table that translates from AST node type names (i.e., strings) to their internal representation (i.e., integers).

5 Measurements

To assess the precision, accuracy, and speed of the approaches discussed, we ran the call-graph extractors on 50,000 lines of the FELt finite-element programming system [18]. SCRUPLE could not be made available for these

	cause of extra matches	cause of missed matches	num. calls	time (sec)
MAWK script	text in comments, strings	none found	8066	10
LSME script (compiled)	text in strings	call arguments, heuristics	8442	165
TAWK basic script	macro expansion	macro expansion	7480	93

Table 1: Some sources of mistakes and quantitative results in call-graph extraction. Computations were performed on a Sparc 10/61 with 160MB of RAM, running SunOS. Times are wall-clock time as computed by the `time` command.

measurements. Table 1 contains an abbreviated quantitative and qualitative summary of the results.

It is hard to quantitatively assess the relative adequacy of the outputs, since the failure modes of each are rather different, and the outputs were not put to any practical use. However, from our inspection of the data, it appears that the lexical tools err mostly on the safe side of generating extra outputs rather than missing them. TAWK errs “equally” on both sides when a macro must be expanded because it is not C-like: calls in the inlined macro are not attributed to the macro (resulting in missed matches); instead, the calls are attributed to the caller of the macro (resulting in extra matches). Macros were not a serious problem in this program, but some had to be expanded, for example, because a type is passed as an argument, which is not C-like:

```
#define New(a,n) ((a *) malloc(sizeof(a)*(n)))
```

The final versions of the lexical scripts benefited from side-by-side comparisons with the outputs of the other tools, but many extra matches were easy to pick out because the matched identifiers were not legal identifiers. Missed matches are harder, and depend upon either a comparison with another tool, examination of the source, or a programmer who knows the source well. In the case of LSME, fixing the script to reduce missed matches was tricky, because its high-level matching paradigm provides only indirect control over the low-level matching process.

Interestingly, the speeds of the tools vary by a factor of 17, but all are within an acceptable range. The “unique” version of the TAWK call-graph extractor (Figure 4) runs in the same time as the basic TAWK script, suggesting that AST construction dominates the execution time.

6 Discussion

Maintaining and evolving software is complicated because of the intricate relationships between different parts of a program. Changing one part of the program often entails making related changes to several other parts of the program that are not obviously related. To assist programmers in making such changes, a pattern matching and processing tool can help in finding related components or even in making the changes.

However, pattern matching and processing faces the complication of finding a good balance between expressive

power, ease of use, and performance. Lexical techniques can be faster than syntactic techniques, but cannot provide the information required to perform precise, sophisticated processing. Lexical techniques are also easier to use and more robust with respect to the peculiarities of language definition and use.

To partially offset the traditional disadvantages of syntactic techniques, we chose to provide pattern syntax and action support that is reminiscent of lexical techniques. We also provide the full power of inherited and synthesized attributes in actions, which can obviate matching in many instances. Conceptually simple actions are kept simple by providing rich abstract data types, and complex actions are supported by having full access to C. Consequently, TAWK can be used for GREP-like searches at one extreme or for data flow analysis at the other.

We have enhanced the generality of the parsing approach by extending the parser and AST to incorporate common macro idioms. Robustness remains a problem, but a syntax error results in the rejection of only the current file, at most. Moreover, parser error recovery [19] can often adequately recover from an error to permit constructing a usable AST for an erroneous construct.

Although our approaches to macro matching and action support complement other existing syntactic techniques, other choices represent alternatives to existing techniques. For instance, granularity control and always starting processing from the source code is a departure from SCRUPLE’s approach for matching against the whole program AST, which is kept resident in memory. Granularity control naturally limits backtracking and space requirements. Although the cost of building an AST can be amortized over several searches, this approach is impractical for large programs because it forces the paging system to be used for storing and retrieving a data structure that is several times larger than the original program source. Performance scales better by rebuilding the AST for each use of the tool and performing matches at smaller granularities. Although syntactic techniques must pay a price for parsing and building an AST, our memory management techniques keep TAWK’s performance competitive with lexical techniques.

Likewise, our use of abstract syntax patterns rather than concrete patterns contrasts strongly with the approaches of SCRUPLE and REFINE. We find the conciseness and flexi-

bility of working directly with the tree structure to be advantageous. For instance, we can use “star” to match as many siblings in a subtree as needed, regardless of their syntactic class. Such a use of “star” in concrete syntax patterns would introduce syntactic ambiguities into the pattern language. Also, granularity control augments the matching paradigm by providing a natural unit of “match” to pass to an action and allowing the placement and repetition of patterns in the script to provide unique control of actions. On the other hand, the handling of textual macros is easier in concrete syntax patterns, since the preprocessor can be applied to them in many instances.

Our experience indicates that the performance, generality, and precision achieved by the techniques applied in TAWK meet the needs of programmers. However, ease of use can be compromised by the matching language’s use of abstract syntax. The name used to denote a kind of program construct can be alien to a programmer. Appropriate documentation can mitigate this problem, but the programmer user may still take issue with the tool designer’s choices in the abstract syntax and find some aspects difficult to remember. The concrete syntax approach of SCRUPLE must also deal with this problem in regard to its wildcards and handling of syntax for constructs like loops. Some of the problem for both of these tools is due to the irregularities and complexities of the language design itself, and appear to have no easy solution. Sometimes, it is possible to avoid the less intuitive aspects of abstract syntax matching by writing simplified patterns and then using actions to complete the matching. Other times, pattern abstraction can help.

Acknowledgments

We are grateful to Michael Pinkerton, who implemented a prototype of the action support and participated in design discussions of the pattern syntax. We also thank Gail Murphy, David Notkin, and Santanu Paul for sharing their knowledge of program pattern matching. We thank the reviewers for their comments on how to improve this paper.

References

- [1] B. W. Boehm. The high cost of software. In E. Horowitz, editor, *Practical Strategies for Developing Large Software Systems*. Addison-Wesley, Reading, MA, 1975.
- [2] A. V. Aho. Pattern matching in strings. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, New York, 1980.
- [3] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk – a pattern scanning and processing language. *Software—Practice and Experience*, 9(4):267–280, April 1979.
- [4] M. D. Brennan. mawk – pattern scanning and text processing language. Unix Manual Page, January 1992. Available by FTP from oxy.edu (134.69.1.2), public/mawk.
- [5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [6] G. C. Murphy and D. Notkin. Lightweight source model extraction. In *ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering*, pages 116–27, October 1995.
- [7] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1990.
- [8] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software re-engineering. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference*, pages 314–322, 1990.
- [9] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.
- [10] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 1–10, May 1994.
- [11] D. A. Ladd and J. C. Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.
- [12] P.T. Devanbu. GENOA – a customizable, language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, May 1992.
- [13] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, March 1996.
- [14] W. G. Griswold and D. C. Atkinson. Managing the design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1–2):99–116, July–August 1995.
- [15] M. Marcus. The Penn Treebank Project, preliminary release. Version 0.5; Compact Disc (CD-ROM), 1992.
- [16] Guy L. Steele. *COMMON LISP, the Language*. Digital Press, Burlington, MA, 2nd edition, 1991.
- [17] D. Weise and R. Crew. Programmable syntax macros. In *Proceedings of the SIGPLAN '93 Conference on Programming Languages Design and Implementation*, pages 156–65, July 1993. SIGPLAN Notices 28(7).
- [18] J. I. Gobat and D. C. Atkinson. The FELt system: User’s guide and manual. Technical Report CS94-376, University of California, San Diego, Department of Computer Science and Engineering, 1994 (revised 1995). Available from the Internet at <http://www-cse.ucsd.edu/users/atkinson/FELt/felt.html>.
- [19] S. L. Graham, C. B. Haley, and W. N. Joy. Practical LR error recovery. In *Proceedings of the SIGPLAN '79 Symposium on Compiler Construction*, pages 168–175, 1979. SIGPLAN Notices, 14(8).