Lexical Source-Code Transformation

Anthony Cox, Tony Abou-Assaleh, Wei Ai, and Vlado Keselj
Faculty of Computer Science, Dalhousie University
Halifax, Nova Scotia, Canada
`{amcox,taa,weia,vlado}@cs.dal.ca`

Abstract

As an alternative to syntactic matching on a program's abstract syntax tree, we explore the use of lexical matching on a program's source-code. Lexical techniques have been shown to be effective for the approximation of an abstract syntax tree, thus permitting tools that use regular expressions to effectively specify rewrite targets. In this paper, the features needed to support lexical rewriting are examined. As well, we report on several tools created to explore these features and suggest directions for future research.

1 Introduction

It is not always possible, or desirable, to parse a source file during maintenance. Errors, missing header files, or embedded language constructs can prevent parsing. Even if the file can be parsed, it might not be beneficial to do so, such as when performing maintenance on the macros in a C file.

Parsing is needed when a maintenance tool manipulates the source-code's abstract syntax tree (AST). As shown by Cordy *et al.* [5] AST transformation is an effective and versatile maintenance technique. The creators of the Software Refinery Toolkit [2] also support this view. However, tools such as TAWK [8], TXL [4], A* [10] , and Refine [2] use syntactic matching and must first parse a file to enable matching against the code's AST.

When code can not be parsed, an approximate AST can be extracted using a lexical technique [11, 6, 7]. This observation indicates that transformation-based maintenance is possible using lexical tools. We now examine the features that such tools should provide and then present tools we developed to explore these features.

2 Tool Features

To facilitate adoption, key tool elements should be well-known and highly accepted by programmers. Patterns and actions should be expressed using familiar formalisms. As well, tools should strive for simplicity; for example, by avoiding complex disambiguating rules. The following features adhere to these principles as much as possible.

Regular Expressions: Within the Unix community, patterns are traditionally expressed using regular expressions (RE). While other, specialized pattern languages exist, none are as widely used as RE. RE have been well studied and provide a strong candidate for a pattern language.

Unrestrictive Action Language: Actions should be specified in a complete programming language to avoid any limitations. Whether the actions are compiled, as in lex, or interpreted, as in AWK, is dependent upon the desired runtime efficiency of the tool. We currently advocate the use of a C-like language for its versatility and popularity.

Stream-Based Match: In source-code, the line structure (format) and syntactic structure are unrelated. Consequently, tools that use line-based, or record-based, matching are ineffective for code maintenance. Stream-based matching considers a file as a stream of characters, with newlines possessing no special properties. Cgrep [3] and grep (when used with the -z option) perform stream-based matching and can effectively locate syntactic constructs.

Disjoint Match: It is possible for two matches to overlap. For example, the expression `(ab)|(bc)` can be matched twice against the text `abc`. Disjoint matching resets the matcher after each match, thus giving priority to the first match in the stream. Anecdotal evidence suggests that this behaviour is what programmers most expect.

Unambiguous Matching: A RE is ambiguous if there exists a string that it matches in more than one way. For example, the expression `(abbc)|(ab*c)` has two matches against the string `abbc`. As it is possible to restructure every ambiguous RE into an unambiguous one [1], ambiguity can be removed, avoiding the need for complex disambiguating rules. In our experience, unambiguous RE are easy to write, once one learns where ambiguity occurs.

Shortest Match: Most RE matching tools use a longest-match algorithm, except for Perl [12], TLex [9], and cgrep [3], which support shortest-match. While both algorithms can match the same sets of strings, for extracting nested

```
dgrep -P -o -z '^[^\n]*?\n\w*main.*?^{.*?^}' file.c
```

Figure 1. Extracting a Function Definition with Dgrep

constructs, shortest-match is simpler (e.g. using shortest-match {.*} finds blocks in C).

Iterative Operation: When shortest-match is used, matching always finds the innermost of a nested construct. To locate all instances of the construct, iteration permits matching to occur from innermost to outermost. The iterative application of shortest-match simulates a bottom up traversal of the AST. A similar facility exists in A*, which has notation to specify multiple passes over an AST.

Sub-String Match: It is often useful to limit matching to a set of sub-strings identified in the text, such as matching RE$_1$ *contained-in* RE$_2$, or RE$_1$ *containing* RE$_2$. Cgrep uses *universes* to provide partial sub-string matching. Universes permit RE$_2$ (e.g. a variable) to be located in RE$_1$ (e.g. a block), supporting the 'contained-in' pattern. *Start states* in lex provide a similar functionality. Additional research is needed to explore the 'containing' pattern.

Other Features: Lex has many extensions to improve expressibility; REJECT permits matches to be rejected and yyless permits parts of a match to be returned to the text stream. TAWK, cgrep and lex, all use macros for pattern definition and simplification. LSME [11] permits substrings of a match to be named and accessed in actions. These features are also of value in a code rewriting tool.

3 Our Tools

Egret is a cross between cgrep and lex. Users create a lex-like specification that is processed and compiled to produce a transformation tool. Egret uses an unambiguous variant of cgrep's stream-based shortest-match algorithm. Cgrep's matching algorithm has been shown to be effective for the extraction of an approximate AST [6, 7]. Many of the features we have presented result from our experiences in implementing and testing Egret.

Dgrep is a modified version of GNU grep where the restriction preventing the use of the -z option with the -P option has been removed to demonstrate that stream-based match can be used with Perl's *ungreedy-match*. Perl differs from cgrep in that the leftmost criteria overrides the shortest-match criteria. For example, the pattern a*b when matched against xaaabx, returns the match aaab using Perl's ungreedy match and b using cgrep's shortest-match. Figure 1 demonstrates the use of dgrep to locate the definition of the function 'main' from code written using GNU stylistic conventions.

4 Future Work and Conclusion

The design and implementation of Egret identified many features needed by lexical source-code transformation tools. Dgrep explored code maintenance using Perl's RE syntax combined with stream-based matching. These implementations suggest that it would be useful to extend Perl to support unambiguous, stream-based, disjoint matching. Perl provides a rich set of programming facilities and offers many of the features needed for code transformation. Consequently, we believe the development of a Perl transformation module will provide an effective tool for performing lexical level source-code maintenance.

References

[1] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, 1971.

[2] S. Burson, G. Kotik, and L. Markosian. A program transformation approach to automating software re-engineering. In *International Computer Software and Applications Conference*, pages 314–322, Chicago, Illinois, October 1990.

[3] C. Clarke and G. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3):413–426, May 1997.

[4] J. Cordy, I. Carmichael, and R. Halliday. *The TXL Programming Language – Version 10.2*. TXL Software Research Inc, Kingston, ON, 2002.

[5] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, 2002.

[6] A. Cox and C. Clarke. A comparative evaluation of techniques for syntactic level source code analysis. In *Asia-Pacific Software Engineering Conference*, pages 282–289, December 2000.

[7] A. Cox and C. Clarke. Syntactic approximation using iterative lexical analysis. In *International Workshop on Program Comprehension*, pages 282–289, June 2003.

[8] W. Griswold, D. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *International Workshop on Program Comprehension*, March 1996.

[9] S. Kearns. TLex. *Software Practice and Experience*, 21(8):805–821, August 1991.

[10] D. Ladd and J. C. Ramming. A*: A language for implementing language procesors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.

[11] G. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[12] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, Sebastopol, CA, 3rd edition, 2000.