Syntactic Approximation Using Iterative Lexical Analysis

Anthony Cox
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia, Canada
amcox@cs.dal.ca

Charles Clarke
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
claclark@plg.uwaterloo.ca

Abstract

Syntactic irregularities, which often occur in source-code undergoing maintenance, prevent the application of analysis and comprehension tools that employ traditional parsing techniques. As an alternative to parsing, we have developed an iterative lexical technique that is based on the repetitive application of regular expressions using a shortest-match strategy. The approach recognizes syntactic elements using iterative refinement, where unambiguous constructs are identified to provide contextual cues for the identification of more ambiguous constructs. The use of a shortest-match strategy supports the bottom up construction of a syntax tree by identifying smaller subtrees first. To examine the technique's effectiveness, we present the results of an experiment comparing iterative lexical analysis against parsing. The measures of precision and recall are used to evaluate and compare the two approaches.

1. Introduction

The majority of source-code analysis tools use traditional parsing to construct an abstract syntax tree (AST) and provide a basis for additional, advanced analyses. However, the nature of maintenance often requires the analysis of *irregular code* [19] – code that, for some reason, cannot be parsed. There are many reasons why a file, or program, can be classified as irregular. Tools needed to prepare the code for parsing, such as a preprocessor or precompiler, may not be available. Evolutionary changes and dialectic differences in programming language constructs may prevent currently available tools from being applied. Syntax errors and missing elements, such as preprocessor included files, may cause analysis tools to fail. Additional causes of irregularity are summarized by Moonen [19].

As an alternative to parsing, Murphy and Notkin have proposed the use of *hierarchical lexical analysis* (HLA) as a technique for building source models from lexically obtained information [21]. However, their research only explored the extraction of high level constructs and did not examine the use of HLA for obtaining a complete AST. The successful application of finite-state cascades, a variant of HLA that is used for natural language parsing [1], also suggests that a lexical approach is suitable for extracting abstract syntax.

While it is possible to use some form of 'fuzzy' parsing [16, 19], we believe that lexical approaches should also be examined to determine their ability to syntacticly analyze irregular code. Lexical tools are often faster to develop than parser based tools [19] and, when developed using hierarchical pattern sets, can be easily extended or adapted for novel situations. Extension is performed through the addition of new lexical levels or additional patterns in an existing level. The use of several discrete levels suggests a form of modular implementation that van den Brand *et al.* [26] believe is needed for program comprehension tools.

Our earlier work on MultiLex [8] examined the use of HLA for extracting a source file's syntactic structure. While early results were encouraging, it was found that the technique is difficult to apply. As well, the fixed number of levels provided by a specific analyzer makes it necessary to use supplementary data-structures when analyzing code with deeply nested constructs. Construction of a hierarchical analyzer is further complicated for languages, such as C, that are not context-free. The need for a symbol table to differentiate type-names and identifiers adds to the complexity of constructing an analyzer.

However, the shortcomings of HLA provided the insight necessary to develop the more effective technique described here – *iterative lexical analysis* (ILA). The use of iteration avoids the need to use supplementary data-structures when analyzing arbitrarily nested constructs. As well, when combined with a shortest-match strategy, iteration permits subelements to be identified and used in the construction of larger syntactic elements. ILA treats syntactic analysis as a form of string rewriting where a nonterminal, from the language's grammar, is substituted for each match of its definition. In this manner, a syntax tree can be constructed using a bottom up approach that iteratively reduces a source file. Given a pattern of the form `{.*}`, shortest-match finds innermost occurrences of nested brace pairs. Replacing each pair with the nonterminal symbol '`block`' permits iteration to work outward and find all such pairs.

As a first step in evaluating the use of ILA for the syntactic analysis of irregular code, this paper examines the use of ILA for the syntactic analysis of regular (parsable) code. In the next section, the theory and implementation of ILA are presented. To demonstrate the capacity of the approach, a simple pattern set to recognize strings with the form $a^n b^n c^n$ is then provided. This example is followed by the presentation of an experiment that measures, with respect to a traditional parser, the ability of ILA to extract the syntax of a test collection of C files. The experiment is discussed and followed by a review of related work and a brief conclusion.

2. Iterative Lexical Analysis

In a traditional source-code analyzer, lexical analysis is used to tokenize the source file prior to parsing. Rather than attempting to extract the abstract syntax of a 'raw' source file, ILA applies lexical analysis to the token stream emitted by a conventional tokenizer. As tokenizers can be quickly implemented using a lexical analyzer generator, such as `lex` [18], tokenization or *level 0 analysis* will not be further discussed. To ensure the identification of maximal length tokens, the Level 0 analyzer uses a longest-match strategy.

In ILA, tokens are encoded using text strings. This encoding permits matching against the text that forms a token, increasing the flexibility of the tool. For example, identifying constants using the tokens '`constINT`' and '`constFLOAT`' maintains the difference between their types, but permits both to be matched with the regular expression `const.*T`. The use of a textual representation avoids the need to provide a mechanism for assigning attributes or types to tokens. As demonstrated in the example, it is possible to encode a subtype relationship using substrings within the token's identifying text.

The token stream emitted by the Level 0 analyzer is used as input for a specific iterative analyzer, such as ILA_c – our iterative lexical analyzer for C. In addition to the token stream, each analyzer requires a pattern set and iteration profile as input. The pattern set is composed of 'pattern-action' pairs, similar to the input for lex. As well, each pair is given a name and a level indicator. The name provides the value of the nonterminal that is substituted when a rewrite occurs. The level indicator partitions the pattern set into levels (subsets) by assigning the rule to a specific, numerically identified level from 1 to n.

Figure 1 provides a pictorial overview of ILA and illustrates the relationships between components. Round-edged boxes containing italicized type indicate functional modules and squared boxes indicated data elements.

The shortest substring matching algorithm used in ILA is based on earlier work by Clarke and Cormack [5]. The algorithm is characterized using the following rules:

1. A character is only matched against one pattern.
2. A character must be used to satisfy the shortest-match.
3. If two overlapping matches have the same length, the match that begins first in the character stream is selected.

The primary difference between ILA and the `cgrep` tool of Clarke is the use of disjoint match in ILA. In `cgrep` matches can overlap (but not nest) and therefore, are not considered as disjoint.

For patterns containing only characters, concatenation, and alternation, matching in ILA is equivalent to regular expression matching. However, the repetition operator, '$*$', performs differently. The pattern $c^*, c \in \Sigma$, always recognizes the empty string since this is the shortest solution to the operator. To make '$*$' function as expected, patterns of the form ac^*b are needed. This pattern finds the shortest sequence beginning with a, ending with b, and containing as many c's as needed. It is this key property of shortest-match that enables syntactic elements to be identified. For example, the pattern `(.*)` can be used to recognize a single pair of matched parentheses. Shortest-match also limits the applicability of 'fall-back' patterns used for error recovery. Patterns such as `return.*;` can be used to find 'return' statements containing erroneous expressions since shortest-match ensures that matching stops at the first semi-colon.

The iteration profile controls the matching of each pattern level against the token stream. Execution usually begins by applying the Level 1 patterns to the token stream. When a match occurs, the appropriate action is performed, the automaton that performs matching is reset, and matching continues, beginning with the token immediately after the match. When the end of the token stream is reached, the stream is 'rewound' and the iteration profile is consulted to determine the next pattern level to apply. In general, the iteration profile uses the number of matches that occurred at the current level to identify the next level. A common strategy is to have the level remain constant until a pass over the stream finds no matches, at which time the level is increased. It is also possible to have actions modify variables for use by the iteration profile in the determination of the next level.

In the pattern set, all rules have the form:

name:*level* *pattern* { `\* action *\` }.

Once a match is identified, the action assigned to the pattern is performed. While users can code any desired action using the C programming language, functions are provided to

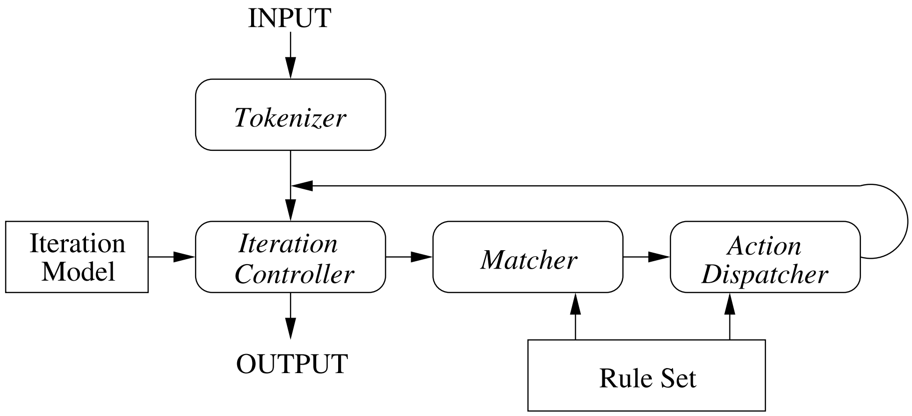

Figure 1. Overview: Iterative Lexical Analysis

Action	Description
`rewrite ()`	Matching text is replaced with the rule's name.
`replace (old, new)`	Replace substring `old` in match with substring `new`.
`copy (s, e)`	Emit substring between offsets `s` and `e` of matching text.
`append (string)`	Append `string` to the output.
`discard ()`	Discard the text that forms the match.
`halt ()`	Iteration and matching ceases.

Table 1. Summary of Provided Action Functions

perform six standard actions. Table 1 provides a summary of these functions.

The `rewrite` action is equivalent to a reduction in a shift-reduce parser and is the most frequently used action. `Replace` rewrites only a part of the match, permitting rules to contain unmodified contextual cues. The `copy` action decomposes a match so that it can be restructured. `Append` inserts new text into the output, allowing 'hints' to be inserted for use in later passes. `Discard` deletes text and permits erroneous or extraneous tokens to be discarded so that analysis can continue. The `halt` action is used to stop iteration when a translation unit has been fully identified, or when analysis can no longer continue.

In addition to performing transformations, all patterns may have side-effects. Side-effects are blocks of code that are executed to perform input and output that does not affect the token stream. Side-effects are used to construct ASTs or report pattern matches and syntactic errors. To facilitate AST construction, a stream of `void` pointers is maintained and manipulated in parallel to the character stream.

In Figure 2 a pattern set for recognizing strings of the form $a^n b^n c^n, n \in integers$, is presented. This example demonstrates the ability of ILA to recognize languages that are not context-free. The patterns assume that the Level 0 analyzer identifies `a`, `b` and `c` as tokens and puts a single space between tokens as a token separator. As well, the Level 0 analyzer appends the 'end-of-stream' marker, `EOS`, to the end of the token stream. The iteration profile is trivial; iteration occurs at level 1, the only level, until no match occurs. All rules names, such as `PAIR` and `ACCEPT` are capitalized for clarity. The `:1` that follows the rule's name within its definition assigns the rule to pattern level 1.

The rule `PAIR` (line 5) identifies '`a b`' pairs and replaces them with the text `PAIR`. On line 6, the rule `swap` rearranges a substring of the form '`PAIR b`' into the substring '`b PAIR`'. The rule `drop` (line 9) locates '`PAIR c`' pairs and discards the match. If the number of `a`'s, `b`'s and `c`'s is equal, a string containing only `EOS` is produced and the text '`Accept`' is produced as a side-effect. If the number is unequal, an unspecified string is generated. This example proves that the recognition capabilities of ILA exceed those of a context-free grammar.

3 Syntactic Extraction with ILA

ILA treats syntactic analysis as a form of incremental string rewriting. Elements are identified using a pattern and an associated action is used to perform substitutions on substrings in the matched text. In Figure 3 a typical substitution sequence is shown. Line -1 is the initial text used as input

```
%{
  #include <stdio.h>
%}
%%
PAIR:1 a.b { rewrite (); }
SWAP:1 PAIR.b {
  copy (5, 5); copy (4, 4); copy (0, 3);
}
DROP:1 PAIR[ ]*c { discard (); }
ACCEPT:1 [ ]*EOS {
  rewrite (); fprintf (stderr, "Accept\n");
}
%%
/* User code section (empty) */
```

Figure 2. Pattern Set to Recognize $a^n b^n c^n$

```
-1        int zero () { return (0); }
 0     int ident ( ) { return ( cnst ); }
 1    dspec name ( ) { return ( expr ); }
 2       dspec declr { return expr ; }
 3            dspec declr { stmt }
 4              dspec declr block
 5                    fundef
```

Figure 3. Example Substitution Sequence

to the analyzer and Line 0 is the output of the Level 0 tokenizer. Lines 1 to 5 represent the output from the first 5 transformation passes.

Ensuring that the algorithm terminates is a user's responsibility. In general, termination is easy to achieve if rewriting either decreases or keeps constant the number of tokens. In the latter case, it is necessary to avoid mutually recursive substitutions. For example, substituting 'name' for 'ident' and then at some other point, substituting 'ident' for 'name'. In our experience, this situation is easily avoided when performing syntax extraction.

To test the applicability of ILA for use in syntactic analysis, we developed ILA_c, a syntax extractor for the C programming language. After some initial experimentation, we found the development of ILA patterns for C to be easier than when using HLA. Furthermore, the patterns bear a strong resemblance to the C grammar [15] that was used for reference.

The iteration profile for ILA_c is provided in Figure 4. Each circle represents a pattern level and each arrow represents a transition, between levels, that may be taken after the current pass. The arrows are labeled to indicate the condition associated with their use. Unlabeled transitions have no condition associated with them, and are therefore always followed after the current pass is completed. The variable m stores the number of matches at the current level. It should be remembered that this is an example of a particular iteration profile and that other profiles can be used to obtain alternative behaviour.

The purpose of each level is identified in Table 2. Levels 1 and 2 serve to condense the token stream as much as possible and group patterns that can easily be found in the initial token stream. Levels 3, 4 and 5 are basic 'parsing' levels that perform the majority of syntactic analysis. They are performed sequentially, but may be combined into a single level if desired. Level 6 contains patterns for which a longer match at levels 3 to 5 is desired. An example of such a pattern is: '`if expr stmt`' since an 'if' statement with an 'else' clause is matched at Level 5.

Level 7 contains some approximate, heuristic patterns that are applied when no other pattern is applicable. The expression '`list * size;`' can be either the declaration of the variable `size`, or a multiplication of two variables, depending upon whether `size` is a type name or a variable reference. As programmers rarely perform multiplication without storing the result using an assignment, it is most probable that the expression declares `size` to be a pointer to an object of type `list`. The patterns of Level 7

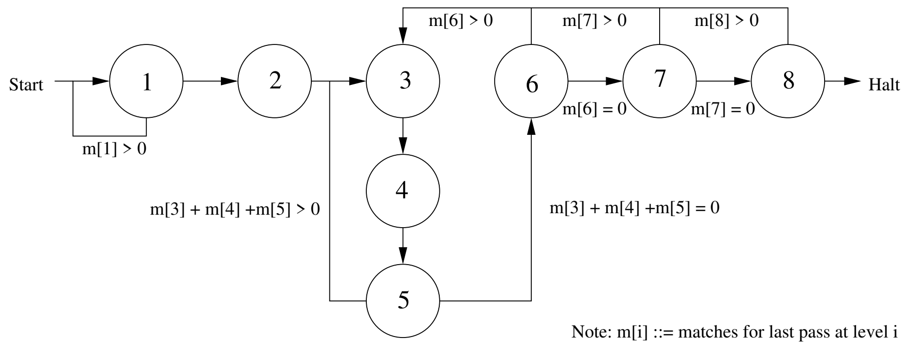

Figure 4. Iteration Profile for ILA_c

Level	Purpose	Patterns	Description
1	simple matches	24	Trivial syntactic elements, e.g. break statements
2	short simple	6	Elements needing a longer match at level 1
3	declarations	82	Declarations and their components
4	expressions	60	Expression components
5	statements	27	Statements and blocks
6	short matches	5	Elements needing a longer match at levels 3, 4, 5
7	good guesses	16	Relatively low risk matches for ambiguous text
8	desperation	4	High risk matches for ambiguous text

Table 2. Summary of Pattern Levels for ILA_c

use heuristics such as these to resolve ambiguous situations. Level 8 contains less probable patterns that are used as a last resort when less risky heuristics have failed.

The small number of patterns in the later levels is a result of the pattern set's specialization for the extraction of an accurate syntactic model. The set was developed to achieve results that compared well against a traditional parser, rather than to create a robust analyzer. Our goal is to first create an accurate pattern set before modifying it to improve robustness. This does not mean that ILA_c cannot handle irregular input, but that it is not yet specifically tailored to do so.

We have chosen to place accuracy before robustness to facilitate comparison, and consequently generate confidence in the tool. To verify the accuracy of ILA_c, it is necessary to compare its output against the output of a tool that is known to be correct. As parsers generate a correct AST, the accuracy of ILA_c can be determined by simulating a parser and then comparing the output against that of a parser.

The sequential nature of Levels 3 to 5 demonstrates the potential of ILA to support a modular approach to analyzer construction. Van den Brand *et al.* [26] suggest that re-engineering tasks require parsing support with modularization and compositional facilities. As demonstrated, ILA can provide this form of support by permitting additional levels to be integrated into an existing analyzer. While interaction will occur between each level, the use of an appropriate protocol or interface will minimize the occurrence of undesirable interactions. It is straightforward to modify an analyzer for a specific language dialect through the addition of an initial 'normalization' level.

Although a shortest-match algorithm is used, longer matches can be iteratively constructed by combining a sequence of shorter matches. For example, matching a list of items can be can be accomplished in $O(\log_2 |list|)$ matches by iteratively matching each pair of items in the list.

4. Comparative Experiments

In order to ensure the accuracy of the information obtained when using ILA to perform syntactic analysis of irregular code, we performed an experiment using ILA as a parsing substitute on syntactically valid code. By demonstrating the capacity of the technique and comparing it against a traditional parser, we believe that confidence in the technique is increased.

The experiment applies both techniques to a set of arbitrarily selected C files. The files are first preprocessed, so that they can be parsed, and then used as input to ILA_c. The files are also analyzed using a traditional parser-based analyzer constructed using `lex` and `yacc` [13]. The parser-based analyzer (PAn), developed as part of the Jupiter source-code repository system [9], provides a baseline for evaluating the output of ILA_c.

To simplify the experiment, we do not actually construct an AST for the analyzed code and instead identify the substrings associated with each AST node. By identifying the starting and ending offsets of each substring, it is straightforward to record the AST in an ASCII file of records, using one record for each AST node. This technique permits two trees to be compared using the Unix `diff` utility.

The baseline syntax identified by PAn reports only the roots of some subtrees. For example, the expressions '`unsigned long int`' and '`main (argc, argv)`' are identified as a 'declaration specifier' and a 'declarator', respectively. Therefore, the number of constructs identified by each tool will be lower than the total number of actual syntactic nodes in each file.

Without access to a symbol table, ILA_c cannot provide the same accuracy as a parser. Therefore, the goal of this experiment is to extract the best possible approximation for the abstract syntax of some arbitrary file containing a complete and error-free translation unit. As well, we are trying to extract the complete, but condensed, set of syntactic constructs and not some artificial subset deemed as interesting. The pattern set only uses patterns such as `{.*}` when more exact patterns fail. In most instances, compound statements are identified with the more accurate pattern:

```
\{ (dec)* (stmt)* \}.
```

The backslashes, '\', before the curly braces cause the braces to be considered as characters to be matched and not as pattern meta-symbols.

As part of the development of ILA_c, the pattern set was applied to a variety of training files. The results were compared against the output of PAn on the same files to permit refinement and improvement of the patterns. The training files were written by a variety of authors to ensure that the pattern set was not specialized for a particular coding style. None of the training data comes from the applications that provided the test files used in the experiment.

When all development was complete, the two analyzers were tested on the source files identified in Table 3. The test files were selected for their perceived diversity with regard to their author and purpose. It is important that ILA_c was not previously used on any of the test files, or files from the same applications, since this 'blind' approach prevented unintentional pattern specialization and the generation of unrealistic results. We have used this experimental design previously to evaluate HLA [8] and found that it enabled us to replicate the findings of Murphy and Notkin [22]. Specifically, we achieved equivalent accuracy rates when identifying function definitions (99%) and function calls (95%).

In information retrieval, the accuracy and retrieval qualities of a tool can be measured using *precision*, p, and *recall*, r. For an identifiable set of entities, E, where a tool finds a subset of these entities, e, the following definitions apply:

$$\begin{aligned} precision &= \frac{|E \cap e|}{|e|} \\ recall &= \frac{|E \cap e|}{|E|}. \end{aligned}$$

Precision measures the correctness of the tool's findings, while recall measures the tool's ability to locate elements of the solution set. When syntactic analysis is viewed as the retrieval of syntax nodes, these measures are suitable for evaluating the accuracy of parsing techniques. Precision and recall are also used to measure the effectiveness of natural language parsers [12], thus indicating their applicability for programming language parsers.

Our comparison assumes that PAn is completely correct, $p = 1.0$, indicating that all identified syntactic entities are valid entities. Similarly PAn is assumed to have complete recall, $r = 1.0$, by finding all existing syntactic entities. We use PAn to generate a baseline record of each test file's abstract syntax. Then, after applying ILA_c to the test files, the results are compared to generate p and r values for each file. The precision and recall of ILA_c are described by the fractions $\frac{|PAn \cap ILA_c|}{|ILA_c|}$ and $\frac{|PAn \cap ILA_c|}{|PAn|}$.

Rather than provide precision and recall values for each construct in each file, only the values for each file are reported, as seen in Figure 4. For the entire test collection, we found that ILA_c had a mean precision of $.971$ and a mean recall of $.947$. Of course, without the use of a symbol table, we did not expect to obtain perfect precision and recall values of 1.0 when using ILA_c. For comparison, `clex`, a hierarchical lexical analyzer developed as a forerunner to ILA_c, obtained a mean precision of $.948$ and a mean recall of $.918$ for 6 selected constructs in 7 test files [7].

The current iteration profile, though accurate, is highly inefficient. For the longest file, `xvmbp.i` (8464 lines and 37468 tokens), analysis took 6 minutes 41 seconds on a 350 Mhz AMD-K6 processor. ILA_c required 1163 total passes over the token stream to perform 55880 total matches against a pattern set containing 225 total patterns. While this time is unacceptable for a production tool, it should be noted that no concern was given to creating an efficient analyzer. Examination of the analysis reveals that passes 161 to 569 performed 1 match per pass at level 3 and 0 matches per pass at levels 4 and 5, suggesting the need for a more efficient iteration profile. However, ILA_c is not intended to replace parsing and instead should be considered as a supplementary tool that, regardless of its current inefficiency, can be used to extract information when parsing is not possible.

File	Author	Application
`pager.c`	M. Haardt	`cs` cross-reference analyzer
`svs.c`	D. Lemke	Spy vs spy Xwindows game
`bit.c`	D. Dube	Bit Scheme interpreter
`isosurface.c`	ETH Zeurich	Molecular imager and analyzer
`lzrw2.c`	R. Williams	Lempel-Ziv compression
`untex.c`	M. Staats	Latex markup processor
`xvbmp.c`	J. Bradley	Xview image display utility
`sprite.c`	J. Griffiths	Jlib video graphics library
`symbolic.c`	C. Fraser, D. Hanson	LCC portable C compiler
`dbtest.c`	K. Bostic	Berkeley database toolkit

Table 3. Test Collection used in Comparative Experiment

File	$\|PAn\|$	$\|ILA_c\|$	$\|PAn \cap ILA_c\|$	Precision	Recall
`pager.c`	9431	9215	9050	.982	.960
`svs.c`	23383	22981	22739	.990	.973
`bit.c`	15783	14467	13958	.965	.884
`isosurface.c`	19806	18982	17810	.934	.899
`lzrw.c`	6218	6185	6090	.985	.979
`untex.c`	5093	4912	4820	.981	.946
`xvbmp.c`	33705	33262	33001	.992	.979
`sprite.c`	30079	29904	28474	.952	.946
`symbolic.c`	16660	16360	15614	.954	.937
`dbtest.c`	11396	11319	10991	.971	.964

Table 4. Precision and Recall Values

5 Discussion

When analyzing incomplete or irregular code, it is not always possible to construct a symbol table. For example, needed definitions may contain syntax errors, or be located in a missing preprocessor included file. For this reason, ILA_c forgos the use of a symbol table. However, the performance of ILA_c exceeds that of `clex` which has a rudimentary symbol table to identify type names. While counterintuitive, this improvement stems from the ease of use that ILA has over HLA. It is easier to write accurate patterns in ILA than it is in HLA.

As well, iteration permits contextual cues to be located and used in later passes, providing additional information for resolving ambiguity. Given the declaration:

```
int x (int y, val z);,
```

the context of use makes it trivial to determine that `val` is a type. In ILA_c, a pattern of the form:

$$\texttt{dec , ident}_1\ \texttt{ident}_2\ \texttt{,}|\backslash\texttt{)}$$

identifies $\texttt{ident}_1$ as a type name. Subscripts are used to identify each occurrence of the substring `ident` within the pattern. In the pattern, the preceding declaration, `dec`, found on an earlier pass, provides the context needed to determine that '$\texttt{ident}_1\ \texttt{ident}_2$' is also a declaration with $\texttt{ident}_1$ providing the declaration specifier and $\texttt{ident}_2$ the declarator.

In ILA, a pattern takes precedence over another pattern when its matching text is shorter. As a consequence, it is possible to take precedence into account when matching programming language operators. For an expression `expr + expr * expr`, the pattern `expr * expr` will be given precedence over the pattern `expr + expr` when `*` is considered as shorter than `+`. While it is possible to insert patterns at Level 1 to match and lengthen instances of `+`, a simple extension to ILA avoids the use of this unusual technique. Pre-identified characters can be stored in a table along with a specified length value, l. Any match using an identified character is treated as having $l - 1$ extra characters, lowering its precedence.

Although the precision and recall of ILA are satisfactory, there is potential for them to be increased. ILA_c is a first attempt at creating a full scale iterative analyzer for a production language and as a result suffers from errors made due to a lack of experience. With additional practice at creating iterative analyzers, it is likely that better tools will be generated. In an attempt to make ILA_c as general as possible, patterns for both old (K&R) style and new (ANSI) style declarations were included. As well, patterns were provided to identify declarations with no declaration specifier and the subsequently implied type of `int`. The interaction between the patterns for these different styles made it difficult to support each style as well as is possible. The use of specialized pattern sets for specific coding styles should lead to better precision and recall values.

The precision and recall values also indicate a design decision that is made in any retrieval tool that works with ambiguous information. Ambiguity creates an inverse relationship between precision and recall. It is always possible to improve recall by lowering precision. Calling every identifier a variable reference ensures that all references are found, $r = 1.0$, but decreases precision by misclassifying other uses of identifiers. Our pattern set is conservative, favouring precision over recall, and generating an accurate syntax tree at the cost of completeness. It should be remembered that precision and recall are measures of a specific tool with the values obtained in this experiment describing ILA_c and not ILA in general.

6 Related Work

The use of lexical analysis to generate source code models has been explored previously by Murphy and Notkin [21]. Their Lightweight Source Model Extraction (LSME) tool uses hierarchical regular expressions to extract unit level models. Although there is no limitation on LSME to prevent it from building low level models, LSME was specifically designed to build high level models. While ILA and LSME both use a hierarchical approach to regular expression pattern matching, the two tools differ in several ways. ILA requires the user to provide definitions for the base level tokens that matching employs whereas LSME breaks the input stream into two token types: identifiers and single characters. As a result, ILA specifications are more complex to write, but permit the reuse of existing scanners for source languages. As well, the iterative nature of ILA avoids LSME's inability to identify constructs that are nested arbitrarily deep.

Finite-state cascades (FSC) [1] have been used to parse natural languages. Described as "little more than a pipeline of `lex`-style lexical analyzers," FSC are identical to hierarchical lexical analyzers. FSC are considered fast and robust, achieving acceptable error rates through the containment of ambiguity within recognized lower-level constructs.

`Lex` [18] also uses regular expressions to define the patterns that are to be matched. However, on its own, `lex` is unsuitable for extracting low level models since regular expressions lack sufficient expressibility to fully describe the low level syntax of most programming languages. In general, a substring of the text can only match a single regular expression, making it impossible to simultaneously match both a statement and an expression occurring within it.

TLex [14] is an alternative `lex`-like tool that permits a wider range of patterns to be matched. Additional support is given for the context surrounding a match, and simple macro facilities enable recursive patterns to be expanded a finite number of times. Once a match has been made in TLex, a parse tree describing the match is available for use in any associated actions. TLex suffers from the same restrictions as `lex`, making it unsuitable for generating low level lexical analyzers.

AWK [2] is similar in many ways to `lex`, being based on regular expressions, but it is less general in applicability since patterns must be matched against predetermined records. While the user can redefine the structure of a record, it is not possible to do so such that a record matches some set of source language constructs. Thus, in AWK, low level information extraction is complicated by the fact that incomplete fragments of a language construct can occur within a record.

As an alternative to lexical matching of a source file, syntactic matching can be used. Tools such as TAWK [11], Aria [10], TXL [6], Refine [24], and A* [17] fall into this category. With a syntactic matcher, the source file must first be parsed to create an abstract syntax tree, and any patterns then matched against the tree.

Fuzzy parsing [16] attempts to create parsers that are more flexible by permitting them to discard tokens and recognize only fragments of a language. A fuzzy parser has a set of 'anchor' or start symbols and a set of grammar fragments, one for each start symbol. Upon encountering an anchor, the parser applies the appropriate grammar fragment to recognize a substring of the language. In Sniff [4], fuzzy parsing is used to recognize high level constructs, such as function and object declarations, in the provided analyzers for C++, Java and Fortran.

Moonen [19] presents a formalism called *island grammars* for the expression of languages that contain irregularities and are thus not representable using traditional grammars. Island grammars are two level grammars with a lower *island* level and an upper *water* level. Productions at the water level are only applied when no production at the island level is applicable. Fuzzy parsers are essentially parsers for island grammars where the water is left unspecified. Source-code analysis using island grammars has been proposed as a viable technique for lightweight *impact analysis* [20] – the determination of the cost for implementing a specific change to some arbitrary source-code. Cost can be described with respect to the number of modules, places, lines or statements that require modification.

Breadth-first parsing [23] uses a two pass algorithm to first identify high level constructs, before applying a subgrammar to parse each construct. It is claimed that, similar to lexical approaches, breadth-first parsing provides better error handling capability than traditional parsing.

ILA can be viewed as a variant of traditional LR parsing where a grammar production's handle (right side) is reduced through its replacement with the non-terminal on its left side [3]. ILA is different since its handles may contain iteration operators that permit a non-terminal to represent an infinite number of handles. While YACC [13] provides an error recovery feature that uses synchronization tokens, ILA is more flexible and can use multiple iteration operators to identify multiple, non-contiguous, synchronization substrings. As well, handle replacement is less restrictive since only parts of the handle may be replaced, or the replacement may use an alternative to the left side non-terminal.

CRTAGS[1] extends the Unix `ctags` tool with a fuzzy parser. Similar to Sniff, CRTAGS only recognizes high level constructs; however, it can do so for the additional languages Pascal, Perl, Verilog, VHDL and Yacc. Source Explorer[2] is another commercially available tool which provides a fuzzy parser. By avoiding the use of a preprocessor, Source Explorer can analyze the contents of any `#if` blocks, independent on the evaluation of the guard condition, and is not dependent on included files being available.

The effects of a preprocessor on software analysis have been examined in several contexts. Somé and Lethbridge [25] explored the problems of extracting facts in the presence of conditional compilation. Vo and Chen [27] created a tool called Incl to analyze include hierarchies for C and C++, including an analysis of macro expansion.

7. Conclusion

When syntactically irregular code is encountered during program comprehension and maintenance tasks, maintainers are usually forced to resort to ad hoc techniques for analyzing and examining the files. While hierarchical lexical approaches can be used to extract a large subset of the code's abstract syntax, the methods are difficult to apply. As an alternative, we suggest that an iterative lexical approach based on a shortest substring matching algorithm is practical.

To assess the accuracy of ILA on error-free code, and consequently improve confidence in its results when applied to irregular code, we preformed an experiment comparing ILA to parsing. With a mean precision of $.971$ and a mean recall of $.947$, the effectiveness of ILA_c should be satisfactory for constructing approximate syntactic models to support program comprehension and maintenance. However, there remains additional research and development to be performed.

The experiment described in this paper validates the use of ILA for syntactic analysis of valid (error-free) code. In subsequent research, the application of ILA on irregular code needs to be investigated. The existing pattern set needs additional refinement to address any failings revealed by comparison with the parser before it is extended to improve its tolerance for irregular code.

One of the primary goals of ILA is to permit the extraction of syntactic information from code fragments that are conditionally discarded by the preprocessor. After modifying CPP to save discarded code, ILA_c can be applied to the fragments to extract information associated with uncompiled variants. The 'bottom up' approach used by ILA is intended to facilitate analysis of these code fragments. The use of a map between the fragments and their originating source file permits syntactic information to be relocated to source file.

While compositional construction of analyzers is possible by sequencing independently developed pattern sets, we have not developed a formal framework for this composition. Additional research is needed on interfacing pattern sets to ensure that the token stream is maintained in a consistent form and to prevent non-termination from occurring when pattern sets rewrite the stream using mutually recursive substitutions.

Clarke and Cormack [5] describe the provision of a containment operator for limiting the search universe. Such an operator could be used in ILA to specify a context for the use of selected patterns. For example, statements could be matched only when contained in a function body, the only valid location where a statement can occur. As ILA is based on Clarke and Cormack's shortest-match algorithm, their operator should also be implementable in ILA.

Experience has shown that ILA patterns are easier to create than those for HLA, and provide good precision and recall. As well, ILA provides modularity that facilitates the modification of a general pattern set for use with a specified application, language dialect or coding style. Although we have yet to apply ILA to irregular code, the results obtained from applying ILA to error-free code suggest that the technique will achieve good results with irregular code.

[1] `www.vital.com/crtags.htm`

[2] `www.intland.com/html/technical_overview.html`

References

[1] S. Abney. Partial parsing via finite-state cascades. *Journal of Natural Language Engineering*, 2(4):337–344, 1996.

[2] A. Aho, B. Kernighan, and P. Weinberger. AWK – A pattern scanning and processing language. *Software Practice and Experience*, 9(4):267–280, 1979.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[4] W. Bischofberger. Sniff—a pragmatic approach to a C++ programming environment. In *USENIX C++ Conference*, pages 67–82, Portland, Oregon, August 1992.

[5] 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.

[6] J. Cordy, C. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[7] A. Cox. *A Source-Based Approach to Representing and Managing Information Extracted by Program Analysis*. PhD thesis, University of Waterloo, Waterloo, Canada, 2002.

[8] A. Cox and C. Clarke. A comparative evaluation of techniques for syntactic level source code analysis. In 7^{th} *Asia-Pacific Software Engineering Conference*, pages 282–289, Singapore, December 2000.

[9] A. Cox and C. Clarke. Representing and accessing extracted information. In *International Conference on Software Maintenance*, pages 12–21, Florence, Italy, November 2001.

[10] P. Devanbu, D. Rosenblum, and A. Wolf. Generating testing and analysis tools with Aria. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

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

[12] P. Jacobs and L. Rau. Innovations in text interpretation. *Artificial Intelligence*, 63(1):143–191, 1993.

[13] S. Johnson. Yacc–yet another compiler-compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, July 1975.

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

[15] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[16] R. Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27(6):637–649, 1996.

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

[18] M. Lesk. Lex–a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1975.

[19] L. Moonen. Generating robust parsers using island grammars. In *Eighth Working Conference on Reverse Engineering*, pages 13–22, Stuttgart, Germany, October 2001. IEEE.

[20] L. Moonen. Lightweight impact analysis using island grammars. In *Tenth International Workshop on Program Comprehension*, pages 219–228, Paris, France, June 2002. IEEE.

[21] G. Murphy and D. Notkin. Lightweight source model extraction. In *Symposium on the Foundations of Software Engineering*, pages 116–127. ACM SIGSOFT, October 1995.

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

[23] J. Ophel. Breadth-first parsing. Department of Computing and Electrical Engineering Research Memo 97/12, Heriot Watt University, Edinburgh, Scotland, 1997.

[24] Reasoning Systems, Inc., Palo Alto, California. *REFINE/C User's Guide, Version 1.1*, May 1994.

[25] S. Somé and T. Lethbridge. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Sixth International Workshop on Program Comprehension*, pages 118–125, Ischia, Italy, June 1998. IEEE.

[26] M. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Sixth International Workshop on Program Comprehension*, pages 108–117, Ischia, Italy, July 1998. IEEE.

[27] K.-P. Vo and Y.-F. Chen. Incl: A tool to analyze include files. In *USENIX Summer Conference*, pages 199–208, San Antonio, Texas, June 1992.