General Incremental Lexical Analysis

TIM A. WAGNER and SUSAN L. GRAHAM
University of California, Berkeley

We present the first fully general approach to the problem of incremental lexical analysis. Our approach utilizes existing generators of (batch) lexical analyzers to derive the information needed by an incremental run-time system. No changes to the generator's algorithms or run-time mechanism are required. The entire pattern language of the original tool is supported, including such features as multiple user-defined states, backtracking, ambiguity tolerance, and non-regular pattern recognition. No a priori bound is placed on the amount of lookahead; dependencies are tracked dynamically as required. This combined flexibility makes it possible to specify the lexical rules for real programming languages in a natural and expressive manner. The incremental lexers produced by our approach require little additional storage, run in optimal time, accommodate arbitrary (mixed) structural and textual modifications, and can retain conceptually unchanged tokens *within* the affected set through aggressive reuse. We present a correctness proof and a complete performance analysis and discuss the use of this algorithm as part of a system for fine-grained incremental recompilation.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments—*interactive*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*version control*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *parsing*; *translator writing systems and compiler generators*; E.1 [**Data**]: Data Structures—*trees*

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: Incremental lexical analysis, scanner, token, reuse, error recovery

1. INTRODUCTION

Batch lexers derive a stream of tokens by processing a stream of characters from left to right. Several tools that facilitate the construction of such lexers have been devised, including the well-known Unix tools `lex` [Lesk and Schmidt 1979] and `flex` [Paxson 1995]. These tools support an extension of regular expression notation as their *pattern set*. Each pattern is associated with a rule; in many problem domains the goal is to partition the character stream into tokens, and each rule typically constructs (part of) a token from the text matched by its corresponding pattern.

In some situations, such as in a software development environment (SDE), a series of character streams are repeatedly analyzed with few differences (relative to the total number of tokens in the program) from one application of

This research has been sponsored in part by the Advanced Research Projects Agency (ARPA) under Grant MDA972-92-J-1028, and in part by NSF grant CDA-8722788. The content of this paper does not necessarily reflect the position or policy of the U. S. Government.
Authors' addresses: Tim A. Wagner, 573 Soda Hall and Susan L. Graham, 771 Soda Hall; Department of Electrical Engineering and Computer Science, Computer Science Division, University of California, Berkeley, CA 94720-1776. email: `twagner@cs.berkeley.edu`, `graham@cs.berkeley.edu`; URL: `http://http.cs.berkeley.edu/~twagner`, `http://http.cs.berkeley.edu/~graham`.

lexical analysis to the next. Since tokens typically have very limited contextual dependencies, the resulting differences in the token stream are typically limited to the area immediately surrounding modification sites. In this setting it makes sense to retain the token stream as a persistent data structure and use it to decrease the time required for subsequent analyses. We will refer to this as *incremental lexing* and the tool that performs it as an *incremental lexer*. Our approach exploits existing technology for generating batch lexers by using the output of these tools produce in conjunction with an *incremental run-time system*.

This approach has several advantages. No changes to the generator's implementation are necessary. No assumptions about the implementation of the batch lexer are made; our prototype uses dynamic linking to load C and/or C++ code, so the generator can use any combination of tables and code without affecting the incremental run-time service. No new descriptive formalisms are introduced, and very few changes to a given lexical description are required to 'port' it to an incremental setting. (Alternatively, an imperative pattern language and embedded programming language may be used as a powerful compilation target for higher-level declarative descriptions.) The technique produces incremental lexers whose performance is competitive with hand-coded implementations.

Our approach is novel in supporting the full expressive power of the underlying tool's pattern language, including: user-defined states, multiple tokens/pattern match, multiple pattern matches/token, ambiguity tolerance, and unbounded lookahead. In fact, arbitrary code can be used in constructing a contiguous sequence of tokens; this can be used, for example, to match non-regular syntax.

By capturing the state of the batch lexing machine at the conclusion of each token's creation and saving it within the token, we are able to restart lexical analysis at point in the token stream. This provides extremely fine-grained incrementality. (Restarting lexing *within* tokens is not only unnecessary in practice but would require novel generation techniques and a significant increase in the complexity of the run-time system and the size of the persistent lexical stream.)

A common, though restrictive, method for handling the contextual dependencies arising in lexical analysis is to place an a priori bound on the maximum amount of lookahead, either by having the language designer provide it or by having the generator compute it through analysis of the lexical description. Either way the expressivity of the language suffers, since any fixed bound on the length of a dependency may be insufficient for a particular language. Languages with unbounded lookahead and natural descriptive techniques that similarly lead to unbounded lookahead are entirely prevented by a fixed bound approach. We remove this restriction by tracking dependencies *dynamically*, resulting in a larger class of languages which can be expressed and potentially faster running times (because the actual dependencies are used instead of a worst-case assumption).

Dynamic dependencies can be explicitly recorded in tokens at a modest space cost and no performance cost. But even for languages or descriptive techniques requiring potentially unbounded lookahead, the vast majority of cases involve a token relying only on the following character. For this reason, we can typically avoid explicit storage by representing only the exceptional cases using an associative data structure. This approach results in the most expressive possible notation with no performance penalty and negligible additional space.[1] (For the sake of clarity, we will always represent inter-token dependencies explicitly in our discussion and figures.)

Our incremental lexing algorithm operates in optimal time and is linear in the number of *affected characters*—the characters in modified tokens and in tokens whose lookahead reached (or which are reached by lookahead from) modified tokens. The batch lexer is invoked the minimal number of times: once for each token in the updated token stream whose contents, lookahead, or starting state may have been modified. No additional asymptotic overhead accrues due to the incremental run-time service; it operates in time linear in the number of old and new tokens containing affected characters.[2]

The overhead associated with locating modification sites depends on the representation in which incremental lexing is embedded. A simple list of modified tokens would add no asymptotic overhead and could be used to implement the algorithms presented here. However, our primary application is the incremental analysis of programs within an incremental software development environment (ISDE), in which case the tokens will be leaves of a syntax tree. This tree can be maintained in a balanced fashion if unbounded sequences (such as declarations and statements) are represented correctly [Gafter 1990; Wagner and Graham 1996a]. Under this assumption, the time to locate a particular token is logarithmic in the total number of tokens.

In addition to providing the full expressive power of the underlying batch pattern language and unbounded lookahead, this algorithm is also novel in addressing the problems of mixed structural and textual edits with arbitrary timing of analyses.. We also discuss issues pertinent to an ISDE, including optimal token reuse, efficient use of distributed history information, efficient reversibility of the lexer's transformation, and parallel operation of the incremental lexer with the incremental parser (needed for such languages as C and C++).

The remainder of this paper is organized as follows. Section 2 discusses related work. In Section 3 we provide the background for incremental lexing: the editing model, interface to the batch lexer, and persistent representation of state and contextual dependency information in the token stream. We also discuss the pass structure of the incremental lexing algorithm and introduce a running example. The next three sections elaborate on the implementation and analysis of each pass. In Section 4 we discuss the algorithm for combining a set of textual and/or structural modifications and dynamic dependency information

[1]A trivial amount of space is required during the actual analysis. In the absence of an explicit history service, at least two additional bits in each token and interior tree node would be required to summarize modifications during editing.

[2]Dependency analysis can be super-linear in the number of extant lookaheads; we make the reasonable assumption that this parameter is bounded by a small constant in any practical description.

to discover the *prefix set* of tokens. Section 5 describes incremental lexing per se, by giving an algorithm that traverses each affected region and restores the consistency of its text and token boundaries. Finally, Section 6 discusses the post-pass that updates inter-token dependencies when they are tracked dynamically. Each section discusses the correctness and performance of the algorithms it contains. The use of incremental lexing as part of an integrated approach to incremental software development is taken up in Section 7, which considers the topics of token reuse, reversibility of the lexical transformation, and error isolation and recovery.

2. RELATED WORK

Unlike incremental parsing, little has been published on incremental lexing despite a number of research and commercial environments that incorporate some variant of it [Bahlke and Snelting 1986; Reps and Teitelbaum 1989; Jacobs and Rideau-Gallot 1992; **?**; Fischer et al. 1986; **?**]. Although the theory of regular expressions is simpler than the parsing theory typically required for incremental analysis [Larchevêque 1995; Wagner and Graham 1996a], the need for dynamic dependency tracking results in a more complex run-time mechanism for incremental lexing.

Several mono-lingual environments have also included incremental lexical analysis [Delisle et al. 1986; Ross 1986]. These systems of course have the advantage of only requiring sufficient expressiveness for a single language. However, our goal was to devise a multi-lingual approach that could employ formal language definitions, reuse existing language definitions in a familiar formalism, and provide maximum expressiveness and flexibility without compromising performance.

Other approaches rely on extremely restrictive editing models or fail to be truly incremental. The Galaxy environment [Beetem and Beetem 1991] touches *every* token on any textual modification to the program. The Synthesizer Generator [Reps and Teitelbaum 1989] limits the user to a single outstanding edit, for which batch lexical analysis is employed to incorporate a textual modification. PSG [Bahlke and Snelting 1986] permits *either* textual or structural modifications, but not both, and halts its incorporation of textual modifications at the first error it encounters, rather than continuing its analysis. Its lexical generator, Aladin [Fischer et al. 1992], produces incremental lexers that cannot use more than a single character of lookahead.

The Pan system [Ballance et al. 1992] possesses truly incremental lexical analysis, in that an unlimited number of disjoint textual edits can be applied to the program between analyses, and lexing is then applied only to the out-of-date portions of the program. Pan is also a multi-lingual system, compiling language descriptions off-line and loading the compiled results into a single, multi-language environment. However, this system limits contextual dependencies to a single token.[3]

[3]Regular expression notation is used for the pattern language, but many of the language features of `flex` are not provided. A special-purpose syntax is available for describing nested comment conventions (which would otherwise be inexpressible). Our approach supports non-regular patterns

The POE environment [Fischer et al. 1986] provides per-character error reporting by using a lexical analyzer capable of stopping and resuming at any point. This complicates the design, is not compatible with existing generators, and does not provide a useful increase in practical performance, since one can simply restart analysis from the beginning of the previous token if desired. While it is not our belief that character-by-character analysis is of practical use, the algorithms described here can implement this policy without modification, since it is merely a special case of our more general editing model.

Other researchers [Szafron and Ng 1990; Heering et al. 1992] have focused on the problem of incremental *generation* of batch lexical analyzers, as opposed to incremental lexing. Since language specifications are long-lived and infrequently changed relative to the number and frequency of changes made to programs written in those languages, we have been more concerned with the speed of the compile cycle than the speed of the compiler-compile cycle. Nevertheless, an incremental lexical specification for a typical programming language (C) can be generated in under 3 seconds on a typical Unix workstation and dynamically loaded into a running environment in under 1/10 second, making lexical specification compilation itself an interactive process without resorting to incremental or lazy generation techniques.

Gandalf? PECAN? Others? Leif? Fred? (do Sue or Bill have this?)

3. FRAMEWORK AND OVERVIEW

In this section we discuss the framework of incremental lexing, including the editing model, the form of both individual tokens and (versions of) the token stream, and the model of the batch lexer. This section also introduces our running example and concludes with an overview of the pass structure of the incremental lexical analysis algorithm.

3.1 A Running Example

Figure 1 contains the original and modified program text that serves as our sample editing sequence. The idea behind this user-supplied transformation is that the timing of the decision to execute debugging statements in the program is being changed from compilation (hence the use of the preprocessor directive to perform conditional compilation) to run-time. The lexical description in Figure 2 uses `flex`'s notation [Paxson 1995] to specify a portion of the tokenization required by the preprocessor language. No previously published work or existing environments supporting incremental lexing can correctly implement this combination of language features and editing sequence. Our approach not only supports this transformation, it does so optimally.

3.2 Token Representation

Incremental lexing is the incremental maintenance of the mapping between a textual stream and a token stream. Each character belongs to exactly one token, and the lexer must partition the textual stream by locating the inter-

in a general fashion, by permitting arbitrary code to be used in the construction of tokens. The translation from Pan's notation into `flex`-style patterns and rule actions is straightforward.

```
                    Before:
...;
/* check for debugging */ #if(DEBUG==1) ...

                    After:
...;
/* check for debugging */ if(DEBUG==1) ...
```

Fig. 1. A sample editing scenario used as our running example. The deletion of the # character changes the meaning of the line by altering it from a preprocessor directive to a sequence of ordinary tokens in C or C++. The essential problem for the incremental lexer is discovering that the textual modification requires replacing the preprocessor keyword PP_IF with the normal keyword KW_IF, even though the lexeme remains unchanged.

```
whitespace      [ \t]*
comment         "/*"([^*]|"*"[^/])*"*/"
ident           [_a-zA-Z][_a-zA-Z0-9]*
intconst        [1-9][0-9]*
%start pp_directive
%%
comment                         return CMNT();
whitespace                      return WS();
\n                              BEGIN(INITIAL); return WS();
^\(whitespace|comment)*#        BEGIN(pp_directive);
<pp_directive>if                return PP_IF();
<INITIAL>if                     return KW_IF();
#                               return PND();
(                               return LP();
ident                           return IDENT();
==                              return EQEQ();
intconst                        return INTCONST();
)                               return RP();
.                               error();
%%
```

Fig. 2. A partial lexical specification for our running example, using the notation of flex. Each line before the first %% denotes a macro definition which may appear in the patterns. Each line above the final %% contains a pattern on the left and a corresponding 'action' on the right. For the purposes of this paper, the rule can be thought of as a procedure written in C++ and combined with the automatically-generated batch lexing machine. The typical action is simply a constructor for the token class. By default the text matching the pattern is passed to the constructor to form the token's lexeme (this and other details of token construction are omitted from the figure). The notation <state> denotes restricts a rules applicability to the named state; state transitions are indicated by BEGIN statements. The backslash indicates trailing (right) context that is required in order to match the pattern, but does not become part of the pattern text handed to the corresponding action. A caret at the beginning of a pattern indicates that beginning-of-file or newline must precede the pattern. INITIAL refers to the machine's normal state.

```
struct TOKEN {
  INT type;            Token type (class).
  STRING lexeme;       Token's text.
  TREE_NODE *parent;   Parent in tree.
  Fields used and maintained by the incremental run-time mechanism:
  INT state;           State of lexing machine when the token is constructed.
  INT lookahead;       Number of characters read beyond lexeme.
  INT lookback;        Earliest preceding token whose lookahead reached lexeme.
};
```

Fig. 3. Representation of tokens. The `type` field is set once, when the token is constructed. The `parent` field is used by the editor to maintain a balanced tree representation. The `lexeme` and `state` fields are set when the token is constructed and maintained thereafter by the incremental run-time service, which also maintains the dependency-related fields. In practice, the incremental fields are represented implicitly, but for clarity we include them here.

token boundaries and assigning a type to each resulting lexeme. We will assume that the mapping from a token to its lexeme is explicit and computable in constant time. Tokens persist until deleted explicitly through editing or implicitly when an invocation of incremental analysis fails to retain them in the resulting token stream. Tokens are created by the batch lexer; if the editor explicitly construct tokens, it must ensure the correctness of their fields or consider them as outdated portions of the stream. The beginning of the token stream is marked with a sentinel called `bos`, the end of the stream by `eos`. In a typical ISDE (and in our presentation of the algorithm), tokens will be represented as the leaves of a balanced binary tree. Figure 3 summarizes the internal representation of a token; the incremental fields are described below.

3.3 Edit Model and Historical Views

Explicit editing of both text *and* tree structure is permitted, with an unlimited number of edit sites and arbitrary timing in the application of lexical analysis. Both textual and structural editing will, in general, temporarily violate the consistency between tokens and their lexemes; invoking the incremental lexer will restore consistency.[4]

Since the editing model is fully general, the exact form of various operations is immaterial. A textual insertion between two tokens, for example, can be recorded as an insertion to the end of the earlier lexeme,[5] as an insertion to the beginning of the latter lexeme, or even as a structural operation that introduces a new 'placeholder' token between them. (Deleting the entire lexeme of a token offers a similar variety of representation choices.)

Our presentation will assume the existence of a powerful set of history services. These services record changes made to the text and structure and assist the lexer to traverse the modified portions of the tree efficiently. The history

[4]Note that consistency is not the same as correctness. 'Errors', in the sense of textual sequences not permitted by the language definition, can be represented in several ways and can persist indefinitely in the token stream until corrected by the user. Section 7.3 discusses error representation in more detail.

[5]This is the representation used in our prototype. Insertions at the beginning of the program are added to the 'lexeme' of `bos`.

```
Functions exported by the batch lexer:
INT get_state ()
void set_state (STATE state)
list of TOKEN* more_tokens ()

Functions called by the batch lexer:
INT next_char ()
```

Fig. 4. Interface to the batch lexing machine. The primary operation is `more_tokens`, which constructs an atomic token sequence (Section 3.5) and returns it. Two new operations are required of the batch machine when it is used in an incremental setting: `get_state` and `set_state`. These have no counterpart in batch analysis; they are used to record the current state when a token is constructed and to place the batch machine into a preserved inter-token state in order to re-start it at a location other than the beginning of the token stream. The `next_char` function is provided to the batch lexer by the incremental run-time service: it uses the lexemes of the persistent token stream to provide the input, as opposed to the buffering and file I/O used in a conventional batch setting.

service makes this focused traversal possible by providing two attributes on each internal and leaf node of the tree. The *local* attribute indicates that local changes occurred; this represents direct textual or structural editing at that site. The *nested* attribute indicates nodes on a path to one or more locally changed sites.

The current version of the tree (and hence the token stream that represents the left-to-right ordering of its leaves) is fully maintained at all times. In addition to the current version, however, the history services also support *historical views*–the ability to access other versions of the tree as if they were logically separate copies. (When no version argument is present in a query function, the current version is implied.)

The incremental lexing algorithms as we present them make extensive use of historical views.[6] Three versions of the token stream are of interest. The *current* version represents the output of the incremental lexer (and parser, when operated in tandem). The *previous* version represents the token stream as it existed immediately prior to the start of re-analysis. The order of the characters between the previous and current versions is unchanged, but the token types and boundaries will, in general, have changed. The lexemes of the previous version supply the input to the batch lexer. The final version of interest is the previously-analyzed or *last-lexed* version. The set of modifications applied by the user between the last lexed version and the previous version constitute the regions requiring re-analysis (along with additional areas dependent upon them, as discussed below). At the conclusion of the analysis, the current version becomes the last-lexed version for the next round of editing.

3.4 Batch Lexer Model

The model of the batch lexer is represented by the interface in Figure 4. The incremental run-time system uses the batch lexer as a subroutine, calling

[6]Simultaneous access to multiple token streams dramatically simplifies the details of the implementation. However, it is not an intrinsic requirement, and the mechanism could be simulated within the editing operations and incremental run-time service if necessary.

more_tokens to produce the next token sequence. The incremental lexer provides the batch lexer with the next_char function to read characters from the lexemes of the previous token stream.

Reusing a batch lexer in an incremental environment requires addressing two additional issues: the ability to restart the batch machine at a point other than the beginning of the character stream (when it is in the INITIAL state) and the contextual dependencies that result when the batch lexer calls next_char to read characters beyond a token's own lexeme.

3.5 Preserving Lexing States

The state field in each token is used to preserve a snapshot of the internal configuration of the batch lexing automaton. At the conclusion of a rule that constructs a token, the constructor will call get_state to read the state of the lexer and preserve it in the token. At some later time, this preserved state information will permit the incremental run-time service to restart analysis immediately to the right of this token by passing the saved state to the set_state function.

Our approach does not specify the form of the state information, but does assume that it is small enough to be conveniently recorded in each token and that it can be passed to/from the batch lexer in constant time using the functions of Figure 4. flex and lex, for example, both require only a small integer to record an inter-token state. (This is referred to as the 'start state' in those generators.) Since any user-defined states have already been incorporated into the start state by the generator, no additional run-time mechanism is required to support this feature of description languages.[7] The beginning state (called INITIAL in flex) is stored as bos's state.

In our example, there are two states, one for handling preprocessor directive lines and another (the normal state) for processing tokens in the base language. The presence of a '#' character after a newline with only whitespace or comments between them signals the start of a preprocessor directive. Once in the preprocessor directive state, another newline signals the shift back to the normal state.

The relationship between patterns in the lexical description and lexemes is not necessarily straightforward. Multiple pattern matches (and their corresponding rule invocations) may be required to construct a single token, in which case only the state at the conclusion of the final rule needs to be preserved. Multiple tokens may also be constructed from the text matching a single pattern, in which case all the tokens involved are returned simultaneously from more_tokens. The alternative, returning a single token for each invocation of the batch lexer, requires an additional interface function to inform the incremental service when the batch lexer has reached a consistent

[7] One feature provided by some lexical description languages is the ability to define patterns exclusive to a particular user-defined state. If a particular pattern can occur in multiple, exclusive states and the state no longer matches, a token must be re-created in order to re-label its state. While this behavior is optimal with respect to the description (and the techniques of Section 7.1 can restore the original token), improved incrementality would be obtained by introducing a third state for the common element(s).

stopping point.

Arbitrary code may be used during the construction of a single token. This is useful in constructing tokens that have non-regular syntax, such as nesting comments, or for complex patterns where standard library code exists, such as floating point constants. However, rules that construct (pieces of) *different* tokens may not communicate except by contextual dependencies that are reflected in the lexical description.[8] Doing so would introduce additional dependencies not visible to the incremental run-time service, and would thus result in incorrect incremental behavior.

This restriction can be relaxed by permitting a contiguous *sequence* of tokens to be treated as a single entity for the purposes of incrementality. The incremental lexer will be unable to restart analysis within such a sequence and it will require the batch lexer to reconstruct it (when necessary) in its entirety. However, as long as such regions are kept reasonably short, this represents a useful technique for constructing contiguous token sequences using unrestricted techniques without a significant loss of incremental performance. Each token in such a sequence except the final one will possess a special state value that indicates it is not a valid starting position.

3.6 Computing and Using Lookback Counts

To construct a token, the lexer must scan at least the characters of that token's own lexeme. In some cases, this is sufficient—parentheses are a simple example where the text of the token is sufficient to determine its right boundary. In many cases, however, at least one additional character beyond the end of the lexeme must be examined to detect that the lexeme is complete. For example, identifiers in most programming languages can be of arbitrary length, and the lexer can only be certain that an identifier is complete when it encounters a character not in the legal set of identifier spellings. In general it is not even possible to place a compile-time limit on the number of characters (or even the resulting number of tokens) of lookahead involved; doing so will artificially restrict the set of languages which can be described or the instances of those languages that will exhibit correct incremental behavior.

In a batch environment, the need to support lookahead impacts buffering and I/O operations but not token construction. In an incremental setting with a persistent token stream, lookaheads must be preserved within tokens, because this information helps to provide the link between modifications made by the user and the set of tokens that require re-analysis when the incremental lexer is next invoked. Previous approaches have all used a simple, restricted scheme: a token's lookahead set is required to lie within its own lexeme and the characters of the following token. This results in a trivial relationship between modifications and re-analysis: each modified token *and each token that precedes a modified token* requires re-analysis.

[8]Either *left context*, in the form of a specific state, or *right context*, in the form of lookahead as discussed in the next section. Long-range dependencies such as name binding [Horwitz 1985] are outside the purview of incremental lexing, and we will never mean this type of relationship when we use the term 'contextual dependency'.

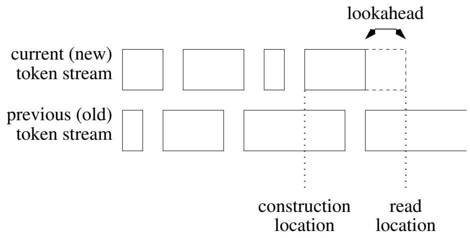


Fig. 5. The determine of lookahead from the activities of the batch lexer. By monitoring the difference between the *construction location* (the start of a re-lexed token) and the *read location* (the rightmost character examined by the batch lexer through a call to `next_char`), the incremental run-time system can determine the length of the character read set. The lookahead is the difference between this value and the length of the token. (Because the token boundaries in the new stream are not necessarily related to those in the previous stream, locations are <*token*, *offset*> pairs.)

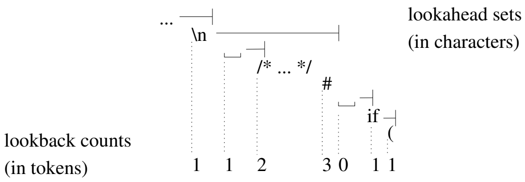


Fig. 6. The relation between the characters read to produce a token and the resulting token lookback counts. Lookahead sets are shown in the figure as horizontal lines. During the dependency update phase, lookahead sets are converted into token lookback counts, shown at the bottom of the figure.

However, natural descriptions of real programming languages, including C and C++, sometimes require lookaheads that span multiple tokens. Our approach supports descriptions requiring arbitrary lookahead by computing and maintaining dynamic dependency information. These contextual dependencies are stored within the tokens themselves, using the `lookahead` and `lookback` fields shown in Figure 3.

The lookahead is computed by monitoring the batch lexer's calls to `next_char` and the length of the resulting tokens it produces.[9] The batch machine itself does not need to be modified in any way. The *character read set* is the number of characters read by the batch lexing machine during the construction of a token; by subtracting the length of its own lexeme, we derive the *character lookahead*—whenever a character in this range is disturbed through textual editing (insertions, deletions, or overwrites) or structural editing, the token just constructed must be re-analyzed, since it might not be constructed in the same fashion given the possibly-changed text to the right of its lexeme. Figure 5 illustrates how the incremental run-time system computes a token's lookahead set from the construction and read locations in the previous version of the token stream.

Token *lookback* counts are used to summarize and invert the information

[9]Some generators require a special option to indicate that minimal lookahead is to be used. Such flags should be used to generate the best possible performance in an incremental setting.

contained in the character lookaheads of previous tokens. Token lookback counts represent the previous tokens that are dependent on one more more characters of a given token: if a token t has a lookahead that reaches y, then y's lookback is sufficient to reach t. Lookback counts are necessary as well as sufficient: no lookback count can be reduced without violating correctness. Figure 6 demonstrates the relationship between lexemes, lookahead sets, and lookback counts in our running example.

The advantage of permitting unbounded token lookahead is the ability to reuse natural lexical descriptions in an incremental setting. However, most token lookahead and lookback counts in a program written in a conventional programming language will be zero or one. We can choose conservative implicit values for the character lookahead and token lookback values that cover the vast majority of tokens. This reduces the space requirement to the constant lookback case without loss of performance—at worst we will re-lex a fixed number of additional tokens for each modified token sequence. (The techniques of Section 7.1 can prevent loss of conceptually retained tokens in this case, so there is truly no penalty for the implicit representation.) The exceptional cases can be represented via an associative data structure. In the case that inter-token states are predominantly a single value, a similar technique may used to effectively eliminate the entire space cost of incrementality.

The computation of lookback values requires special handling for eos. Even though it possesses no explicit lexeme, the detection that no further text is present represents a type of lookahead information. Thus, we treat eos as if it contained a single character; any preceding tokens which read and detect the end-of-stream condition include this pseudo-character in their lookahead sets. The translation to a lookback count is then treated uniformly by the algorithm in Section 6.

3.7 Overview of Algorithm

Incremental lexing begins with a tree where all the nested and local changes have been identified; together these form a set of paths defining an embedded tree structure within the larger tree. There are three main stages to the analysis. In the first (marking) stage, the dynamic dependencies of the previous token stream are combined with the embedded change tree and used to discover the *prefix set*, the set of tokens that begin each contiguous region requiring re-lexing. Having expanded the embedded change tree to incorporate this lookback set, the second (lexing) stage then traverses each out-of-date region until the new and old token sequences once again coincide. In the third stage, dependency updating, the embedded change tree (updated once again from the results of the second stage) is traversed a final time to update dynamic dependencies for each token created by phase two and any unchanged tokens that were examined. (If the range of inter-token dependencies is fixed to an a priori value, the final stage is simply omitted.)

The three passes are conceptually distinct, although they could be applied simultaneously during a single traversal of the tree. Our prototype uses explicit passes both for simplicity of implementation and because a single pass implementation would greatly complicate the interaction with incremental parsing.

Separating the passes does not degrade asymptotic or practical performance and follows the same control structure of other analysis/transformation tools.

Occasionally languages are designed such that the conceptual phases of analysis (lexical, syntactic, semantic) are not separable, for example C and C++[Roskind 1992]. To partially solve this problem, our incremental lexing algorithm is designed to operated as a subroutine of incremental parsing, producing a single token (or atomic token sequence) on each call.[10] Thus the interface to the incremental lexer supports starting and stopping on a region-by-region basis. The only assumption the incremental lexer makes is that, from its perspective, regions are processed left-to-right. (If the driver routine has a priori knowledge that changes to the regions cannot conflict, then this restriction may be relaxed.) The client interface to the incremental lexer consists of the subroutines in Figure 11.

4. MARKING PHASE

In order to efficiently re-lex the token stream, we need to know the starting point of each outdated region. These regions comprise the tokens that have received direct modifications (textual edits) and tokens that have been affected by modification of the tree structure. The affected regions also include tokens that are dependent upon characters in the modified tokens (tokens with lookahead sets that reach one or more modified tokens).

4.1 Effect of Textual and Structural Editing on Dependencies

It is easy to see how a textual edit affects token dependencies: if a character is inserted to, deleted from, or overwritten within a token, then the previous `lookback`th tokens must be considered suspect. Figure 7 illustrates the particular case of our running example.[11] No interaction with the incremental lexer is required at the time of the edit; when lexing is next requested, the current structure of the tree is used in conjunction with the lookback counts computed during the last-lexed version to determine the extent of the effect.

Structural edits are more complex to handle (and also complicate the situation for text editing by potentially re-arranging the order of tokens). While it is possible to treat a structural edit as a textual edit that modifies every token in the yield of the subtree, this would require $O(N)$ time to analyze a subtree containing N tokens. Instead, we perform a precise computation of the possible effect of the change through history-based dependency analysis.

Each structural operation is treated as a replacement; insertions replace a sentinel (completing production) with new material and deletions replace a subtree with a sentinel. The dependency analysis for a structural edit is similar to a modification of the first character in the subtree being removed and the

[10] As is the case with the internal operation of the incremental lexer, the parser's update operations occur to a logically separate tree from its read operations; this prevents any disruption to the lexer as the parser re-writes tree structure and vice-versa.

[11] Slightly greater theoretical precision can be achieved by also using the `lookahead` fields: not all the tokens in the lookback set necessarily reach the modified token, and those that do reach it typically depend only on its left edge. However, this level of precision does not improve the asymptotic or practical performance, and requires knowledge of intra-lexeme modification sites.

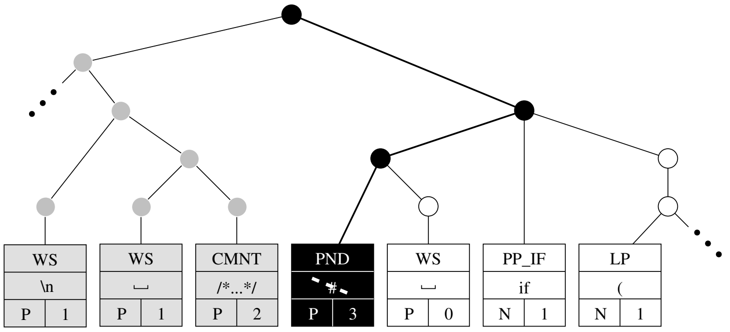
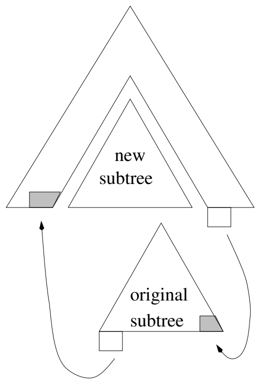


Fig. 7. Effects of textual editing. This figure illustrates the effect of deleting the `#` character in our running example. The token containing the modification and the path from it to the root of the tree are shown in black. The lookback value in the modified `PND` token will be used by the marking routine to discover the set of tokens affected by this change; these tokens and the additional interior tree nodes required to locate them are shown in grey. The bottom row in each token contains the preserved state ('P' denotes the preprocessor directive state, and 'N' the normal state), lookahead count, and lookback count.

Fig. 8. The effect of subtree replacement on incremental lexical analysis. For each replaced subtree, the lookback count in its leading token is used to determine the set of tokens affected by the edit. Both the structural traversal and the dependency analysis are with respect to the last-lexed version. The same analysis is done for the token immediately following the replaced subtree, in order to include tokens within the subtree that are now out-of-date. (For insertions, the first step can be skipped; for deletions the latter step.)



first character following the subtree. More specifically, for each replacement point, the first token in the subtree (if one exists and if it existed in the last-lexed version of the tree) is used to invalidate tokens with lookahead sets that reached it. The tokens affected are determined by traversing the *last-lexed* token stream, *not* the current one. The token following the subtree replacement point is treated similarly. No other tokens can be affected by this subtree replacement since their lookahead sets were not disturbed by it. The effect of structural editing is shown schematically in Figure 8.

4.2 Marking Algorithm

Marking is the process of discovering the *prefix set*, the set of tokens that prefix each region requiring action by the incremental lexer. The input to this phase is a tree where all tokens and internal nodes modified since the last analysis are marked. (We refer to the modified nodes as 'implicitly' marked and the addition tokens discovered by this phase as 'explicitly' marked.) In order to efficiently locate the modified areas of the tree, each interior node on a

```
Locate all the edit sites within node.
Call mark_from() on each edited terminal and the boundaries of each structural edit.
void apply_marking (NODE *node) {
  if (is_token(node) && node->text_changes(last_lexed))
    mark_from(node); Handle textual changes.
  else {
    Handle structural changes.
    if (node->child_changes(last_lexed))
      for (int i = 0; i < node->arity(); ++i) {
        NODE *old_child = node->child(i, last_lexed);
        if (old_child != node->child(i)) {
          Mark first token not earlier than the leading edge of the original subtree.
          mark_from(first_token(old_child, last_lexed));
          Mark first token after the original subtree.
          mark_from(first_token_after(old_child, last_lexed));
        }
      }
    Recursively process any edits within this subtree.
    if (node->nested_changes(last_lexed))
      for (int i = 0; i < node->arity(); ++i)
        apply_marking(node->child(i));
  }
}
```

Fig. 9. Driving routine for marking algorithm. This routine locates all modifications (both textual and structural) applied since the previous invocation of lexical analysis. `first_token` returns the first token in the yield of its argument concatenated with the remainder of the token stream. `first_token_after` is similar, but returns instead the first token after the yield of its argument node. All functions that access structure have an optional argument to specify the version of the tree used for the query.

path to one or more modified nodes is identified as possessing *nested changes*; additional nested changes are added as needed for later passes to locate the explicitly marked tokens.

The driver for the marking phase traverses an optimal path through the tree that reaches each edited site. For internal nodes (structural modifications), it first locates the tokens that may have been affected by the subtree replacement. Each implicitly marked token is then passed to a marking routine that discovers additional tokens dependent upon the changed material by using the dynamic dependency information in the `lookback` fields of the modified tokens.

Since the `state` field of each token records the batch lexer's internal state at the *completion* of a rule, the incremental lexer will use the token *before* the first affected token in each region to pass to the batch lexer's `set_state` function. One complication is that only startable tokens (the final token of each sequence returned by `more_tokens`) can serve as valid starting points. For each marked token, we must therefore step backwards in the *current* version of the tree until we find a node whose `state` field indicates a valid point for re-initializing the batch lexing machine. (In practice, this is typically the previous node.) Because the nodes marked by `mark_from` are not necessarily contiguous in the current tree, `ensure_startable` must be applied to *each* marked node. Figure 9 contains the entry point to the marking phase.

The following theorem demonstrates the correctness of the marking phase

```
Explicitly mark tokens dependent upon tok for re-lexing.
This backup occurs in the last-lexed tree.
void mark_from (TOKEN *tok) {
  if (!tok->exists() || !tok->exists(last_lexed)) return;
  ensure_startable(tok);
  Check everything in its lookback set.
  for (int ov = tok->lookback; ov > 0; --ov) {
    tok = previous_token(tok, last_lexed);
    if (tok == bos) return;
    if (!tok->exists() || marked(tok)) continue;
    mark(tok);
    ensure_startable(tok);
  }
}

Ensure that we have a valid state to re-start the lexer here.
This backup occurs in the current tree.
void ensure_startable (TOKEN *tok) {
  for (TOKEN *tok2 = previous_token(tok);
       !startable_state(tok2) && !marked(tok2);
       tok2 = previous_token(tok2));
    mark(tok2);
}
```

Fig. 10. Marking routine. All tokens from the last-lexed version that are still present in the token stream and that read one or more characters in `tok`'s lexeme are explicitly marked. In addition, the algorithm ensures that the batch lexer can be restarted at the beginning of each outdated region by calling `ensure_startable` for both `tok` and any explicitly marked tokens.

(the extension to multiple-token sequences is straightforward).

THEOREM 4.1. *The marking algorithm marks all and only those tokens requiring re-analysis that are not themselves modified.*

PROOF. For the sufficiency test, we proceed by contradiction. Suppose there exists a token t with at least one modified character in its lookahead set. If the character is within its own lexeme, then t is marked by the modification itself. If the character was in a different token modified through a textual edit, then the token that contained it must have had an lookback field at least large enough to encompass t, and `mark_from` would thus have marked t. The only remaining possibility is that the altered lookahead arose through a structural edit. In this case t was separated from a token containing one or more characters in its lookahead set through a subtree replacement. Without loss of generality, assume that t was to the left of the original replaced subtree in the last-lexed version. Since t's lookahead set extends into the left edge of the subtree, the marking algorithm must have included t in the set of tokens marked when this structural edit point was processed. Hence t is actually marked.

For the necessary test, we merely observe that any explicitly marked token had at least one character in its lookahead set modified through one or more editing operations applied since the previous lexical analysis. □

With regard to running time, marking per se examines only old tokens containing affected characters. It is thus linear in the number of affected characters and affected tokens and clearly optimal.

5. LEXING PHASE

Lexing is the process of repairing a contiguous region of affected characters by reading the (possibly-changed) lexemes from the previous token stream and invoking the batch lexing machine to re-create that portion of the new token stream. Lexing is applied to each outdated region in turn, beginning with the next token in the prefix set not yet visited. In order to stop lexing a region, we must ensure that the construction location (Figure 5) is at the beginning of an unmarked token and that the last newly-lexed token contains a startable state matching the state in the previous token of the previous stream.

The routines comprising the lexing pass are shown in Figure 11. These routines can be used directly by an incremental parser for languages that require lexing and parsing to be performed in tandem. Figure 12 illustrates the required use of these routines by implementing the lexing phase as a standalone operation.

THEOREM 5.1. *At the conclusion of the lexing phase, the token stream is identical to the token stream that would result from executing the same batch machine on the concatenation of the lexemes. Furthermore, each token records the state of the batch lexing machine at the point of the token's construction.*

PROOF. The correctness of the marking phase and batch lexer are assumed. We proceed by induction over the token stream as it exists immediately prior to the start of the lexing phase. The base case is simple: the beginning of stream markers and initial lexical states are clearly the same in both the batch and the incremental streams.

For the inductive case, assume that the lexemes of the preceding $N-1$ tokens have been correctly lexed and that the internal state of the batch lexer is the same in both cases. If the current token is unmarked, then the state of both machines is equivalent, the characters of the token's lexeme and lookahead set are unchanged from the previous invocation, and thus the old token may be safely reused (and the state with which it is labeled corresponds to the state of the batch machine at the conclusion of the rule creating the token).

If the current token is marked, then the batch lexers will read the same set of characters in the same state, and thus produce the same stream of tokens until the stopping condition obtains. At this point, the next character to be consumed and the state of the batch lexing machine correspond to the 'leading edge' of an unmarked token in the old stream, completing the inductive step. □

This phase touches only old tokens that are marked, for which the starting state or offset has been changed, or that are part of a previous atomic sequence. The number of invocations of the batch lexer (and number of tokens examined in the new stream) is therefore optimal and is clearly linear in the total number of affected characters.

```
Begin incrementally lexing a new region starting at tok.
TOKEN *first_new_token (TOKEN *tok) {
  read_token = tok; read_offset = 0;
  if (tok == bos) batch_lexer->set_state(INITIAL_STATE);
  else batch_lexer->set_state(previous_token(tok)->state);
  token_list = ∅;
  return next_new_token();
}

Return the next re-lexed token.
TOKEN *next_new_token () {
  if (token_list == ∅) token_list = batch_lexer->more_tokens();
  for each tok in token_list {
    if (tok is last element) tok->state = batch_lexer->get_state();
    else tok->state = unstartable_state;
    advance(construction_loc, tok->length);
    tok->lookahead = delta_in_chars(read_loc, construction_loc);
  }
  return last_token = remove first token in token_list;
}

Determine when previous and current token streams merge again.
bool can_stop_lexing () {
  return
    token_list == ∅ && construction_loc.offset == 0 &&
    !marked(construction_loc.tok) && is_startable(last_token->state) &&
    last_token->state == previous_token(construction_loc.tok, previous)->state;
}

Incremental run-time service provides this to batch lexer to read from
lexemes in the previous version of the token stream.
int next_char () {
  while (read_offset == read_token->length && read_token != eos) {
    read_token = next_token(read_token, previous);
    read_offset = 0;
  }
  if (read_token == eos) return -1;
  return read_token->lexeme[read_offset++];
}
```

Fig. 11. Lexing algorithm. The input to this algorithm is a marked token stream. The output is a (possibly changed) token stream that is identical to one produced by executing the batch lexer on the concatenation of the previous stream's lexemes. The `next_subtree` function returns the node following its argument's rightmost descendant in a DFS ordering.

Restore consistency to the entire token stream.
(Operations to incorporate tokens into the tree structure are not shown.)
```
void lex_phase () {
  for (TOKEN *tok = find_next_region(root);
       tok != eos;
       tok = find_next_region(tok)) {
    tok = first_new_token(tok);
    while (!can_stop_lexing()) tok = next_new_token(tok);
  }
}
```

Find the next marked token within or after `node`.
```
TOKEN *find_next_region (NODE *node) {
  if (node == eos || (is_token(node) && marked(node))) return (TOKEN*)node;
  if (node->nested_changes())
    return find_next_region(node->child(0));
  return find_next_region(next_subtree(node));
}
```

Fig. 12. Driver for the lexing phase, when used in a standalone fashion. For languages in which incremental lexing must be performed in tandem with incremental parsing, the incremental parser can also use the routines in Figure 11 directly.

6. LOOKBACK UPDATE PHASE

When the lexing phase completes, all the fields in each token in the stream are correct with the exception of the lookback counts. The `lookback` field will be undefined for each token produced by either `first_token` or `next_token`. In addition, newly-constructed tokens may have read characters from lexemes in unchanged tokens. (Conversely, they may have failed to read as far as previous analyses did into the unchanged region.) The lookback phase handles both types of updates.[12]

6.1 Algorithm

During lexing, the character lookahead set for each token is preserved in its `lookahead` field at the time the token is constructed. The lookback update phase consists of transforming these character lookahead counts into token lookback counts. The algorithm's central data structure is a *character lookahead list* that keeps track of multiple outstanding lookaheads. As each token is processed, its character lookahead is added to this list and any lookaheads that terminate within this token's lexeme are removed from the list.

The lookahead sets for our running example are shown graphically in Figure 6. When the comment token is processed, one lookahead is removed (the preceding whitespace token) and one remains (the newline token). The comment token itself has no lookahead, so no entries are added. Figure 13 shows the lookahead list immediately after processing each token.

Each region is processed in three parts: a bootstrap section, a re-lexed sec-

[12]It *is* possible to perform this update in parallel with the lexing phase, but doing so greatly complicates the algorithm. When lexing is performed in parallel with incremental parsing, on-line dependency updating is even more difficult, since the new tree structure is fragmented until parsing completes.

Region	Token	Lexeme	Length	Lookahead list
re-lexed	$\texttt{WS}_1$	`\n`	1	$< \texttt{WS}_1, 28, 0 >$
	$\texttt{WS}_2$	␣	1	$< \texttt{WS}_1, 27, 1 > < \texttt{WS}_2, 1, 0 >$
	CMNT	`/*...*/`	25	$< \texttt{WS}_1, 2, 2 >$
	$\texttt{WS}_3$	␣	1	$< \texttt{WS}_1, 1, 3 > < \texttt{WS}_3, 1, 0 >$
	KW_IF	`if`	2	$< \texttt{KW\_IF}, 1, 0 >$
syncing	LPAREN	`(`	1	$< \texttt{LPAREN}, 0, 0 >$

Fig. 13. Updating lookback counts in a section of the token stream. The bootstrap section is empty in our running example, so processing begins with the first re-lexed token (the newline). We maintain the invariant that the list of lookahead sets contains all and only the lookaheads that reach the lexeme of the current token. When we finish processing lookaheads for re-lexed tokens and find a match between the computed and stored lookback counts, the region is complete. In the example, this occurs when the left parenthesis is encountered.

tion, and a sync section. For the middle section, composed of the re-lexed tokens, the algorithm computes the lookback count for the token based on the contents of the lookahead list. Then each lookahead set in the list is advanced by the length of the token's lexeme, and the token's own lookahead is added.

In order to maintain the invariant that the lookahead list contains all and only lookahead sets from the current version of the token stream that reach the token being processed, we may need to initialize the lookahead list from lookaheads in tokens that *precede* the first re-lexed token. To determine which tokens are included in the bootstrap section, we first note that the token preceding the first (re-lexed) token in the region must exist in the current, previous, and last-lexed version of the token stream. The token that follows it in the last-lexed version contains the relevant lookback count, and we continue adding preceding tokens to the bootstrap section until this count is exhausted or until we discover a point where the token streams differ (indicating that tokens to the left have already been processed).

The symmetric problem arises following the re-lexed tokens: because their lookahead sets may have penetrated (or now fail to penetrate) unchanged tokens that follow, we must continue updating lookbacks until we reach `eos` or the next re-lexed token or until we meet two conditions simultaneously: the lookback list contains no elements from re-lexed tokens (which would imply that we haven't finished updating all the relevant lookback counts) *and* the lookback computed from the lookahead list is the same as that stored in the token being processed. The latter condition is necessary to handle shrinking lookaheads from one version of the token stream to the next.

Figure 14 contains the driver that locates each region requiring lookback processing. The actual updating is performed by the `fix_lookbacks` routine, shown in Figure 15.

THEOREM 6.1. *At the conclusion of the lookback update phase, each token t in the new stream has a lookback value b such that the earliest token in the stream with a lookahead extending into t is the b^{th} previous token.*

PROOF. Consider the tokens processed by a call to `fix_lookbacks`. The search for the left edge of the bootstrap section is terminated either by the earliest token whose lookahead penetrates the current token, by a token that was re-

```
Find and update each modified region of tokens.
void update_lookbacks () {
  NODE *node = root;
  while (node)
    if (is_token(node)) {
      TOKEN *tok = (TOKEN*)node;
      if (tok was re-lexed) node = fix_lookbacks(tok);
      else node = next_subtree(node);
    }
    else if (node->nested_changes()) node = node->child(0);
    else node = next_subtree(node);
}
```

Fig. 14. Driver routine for lookback recomputation.

```
Process a re-lexed region starting at tok.
TOKEN *fix_lookbacks (TOKEN *tok) {
  la_set = ∅;
  if (tok == bos) goto L;
  This token exists in all 3 versions at the same offset.
  TOKEN *boot_tok = previous_token(tok);
  Extract lookback count (if changed in current version, use old value).
  int lb = next_token(boot_tok, last_lexed)->lookback;
  while (--lb > 0 &&
       previous_token(boot_tok, last_lexed) == previous_token(boot_tok, previous) &&
       previous_token(boot_tok) is not re-lexed)
    boot_tok = previous_token(boot_tok);
  Initialize the lookahead set from the bootstrap region.
  while (boot_tok != tok) {
    la_set.advance(tok->length);
    la_set.add_item(tok);
    tok = next_token(tok);
  }
  L: do {
    Set the lookback for re-lexed tokens.
    while (tok was re-lexed) {
      tok->lookback = la_set.compute_lookback();
      la_set.advance(tok->length);
      la_set.add_item(tok);
      tok = next_token(tok);
    }
    Symmetric to bootstrap: process unmodified tokens reached by lookahead from re-lexed area.
    do {
      tok->lookback = la_set.compute_lookback();
      la_set.advance(tok->length);
      la_set.add_item(tok);
      tok = next_token(tok);
    } while (tok != eos && tok is unchanged && !la_set.all_items_discardable() &&
             tok->lookback != la_set.lookback());
    Continue looping until we reach a 'clean' token.
  } while (tok was re-lexed);
  return tok; Return first clean token or eos to caller.
}
```

Fig. 15. Update algorithm for a contiguous range of modified tokens. The driver routine is shown in Figure 14. The operations on the list of lookaheads are defined in Figure 16.

```
advance (int offset)
  replace <tok,cal,cnt> in list with <tok,cla-offset,cnt+1>

int compute_lookback ()
  remove <tok,cla,cnt> s.t. cla <= 0 from list
  if (list == ∅) return 0;
  else return max cnt | <tok,cla,cnt> in list

add_item (TOKEN *tok)
  add <tok,tok->lookahead,0> to list

bool all_items_discardable ()
  ∀ <tok,cla,cnt> in list, tok is not re-lexed
```

Fig. 16. Functions to update the lookahead list during lookback processing.

lexed, or by bos. In the first case, the local token stream is the same in the current, previous, and last-lexed versions, so the lookback count itemized all and only the tokens with relevant lookaheads. In the latter case, the re-lexed token's lookahead has already been processed in full by induction, so again the lookahead list contains all and only the relevant lookahead sets.[13]

We assume the correctness of the lookahead list operations; the lookback count assigned to each re-lexed token is therefore necessary and sufficient by the invariant that the lookahead list contains all and only the lookahead sets reaching the current token. The processing of each re-lexed token clearly maintains that invariant.

For unchanged tokens to the right of the re-lexed section that include characters read by one or more re-lexed tokens, the invariant on the lookahead list's contents remains unchanged. We now examine the stopping condition. The cases of encountering a re-lexed token or eos are trivially correct. The remaining case requires two conditions to hold simultaneously: the lookahead list consists entirely of tokens that have *not* been re-lexed, and the lookback count to be assigned to the current token matches its stored value. This is clearly sufficient, since the lookback count computed for the next token would necessarily match the value of its lookback field. It is also necessary, since a violation of either condition can easily be seen to result in cases where the lookback counts are insufficient or over-estimated.

When the outer loop in fix_lookbacks terminates, the token returned possesses a lookback count that is necessary, sufficient, and unchanged from its last-lexed value (or is eos). This ensures that the *unchanged* token sequence terminated by the next call to fix_lookbacks possesses the same property. □

COROLLARY 6.2. *Lookback processing examines the minimal number of tokens.*

[13]It is true that not all of the tokens in the bootstrap section necessarily have lookaheads reaching the first character of the first re-lexed token in the region: as always, a lookback count represents the union of of lookahead sets, and one or more earlier tokens may have been changed. However, the bootstrap region's processing is sufficient and, given that the only tokens it enters are ones that *could* possess relevant lookaheads, necessary.

Lookback processing is clearly linear in the number of affected tokens and characters; the overhead of tree traversal is the same as in the previous phases.

7. INCREMENTAL LEXICAL ANALYSIS IN AN ISDE

Batch lexical analysis is useful in a number of situations, and incrementality is applicable to several of these. Our primary interest, however, is the use of incremental lexical analysis as a component of an ISDE. In this case the token stream is part of the persistent (structural) program representation, and an incremental parsing algorithm is the 'client' of the incremental lexer.

The algorithms we have discussed so far can be applied without change in this setting. In this section we describe three additional topics primarily of interest within an ISDE: the preservation of information through *token reuse*, the *reversibility* of the transformation induced by incremental lexical analysis, and the issue of *error detection and recovery*.

7.1 Token Reuse

The analysis and transformational tools in an ISDE should be designed to reuse physical nodes whenever they are logically unchanged: the maintenance of associated information can be implemented most efficiently when the physical identity of an item matches its conceptual identity. Token reuse also improves performance because reuse calculation is cheap—far cheaper than additional incremental reevaluation by semantic analysis and other tools. Token reuse thus decreases the amount of *total* work performed in response to the original program modifications. It also has a strong impact on the user interface, because inexact visual indications of updates are confusing (they violate user intuition) and because unneeded token reconstruction loses annotations authored by the user (or by previously-applied tools that are not incremental).

Because our incremental lexical analysis algorithm is optimal, it intrinsically reuses that portion of the token stream provably unaffected by the user's modifications. However, *any* analysis is inherently conservative, and many common modifications result in re-lexing tokens that are conceptually unchanged, creating new tokens isomorphic to the old ones. Other times some component of a token actually *has* changed, but is not a part of the user model and the token should thus be reused. This latter type of reuse occurs in our running example, where the `state` fields of the re-lexed tokens are altered, but the user-visible information (location, type, and lexeme of the tokens) is unchanged.

We will consider two different approaches to reuse.[14] The first, *bottom-up reuse*, is computed directly by the incremental lexer as it operates. In our running example, it is easy to see that the re-lexed tokens can be reused: only the `state` fields will change from the previous version of the stream to the new version.

More generally, we compute bottom-up reuse by examining tokens in the previous version of the stream that would not otherwise be incorporated in the new version. When the type of a newly-constructed token matches the

[14]Node reuse by the incremental parser, including a definition of optimality, can be found in Wagner and Graham [1996a].

```
bool bottom_up_reuse_test (TOKEN *tok) {
  if (construction_loc.tok->type == tok->type &&
      construction_loc.tok != last_reused_token &&
      construction_loc.tok != eos) {
    construction_loc.tok->state     = tok->state;
    construction_loc.tok->lexeme    = tok->lexeme;
    construction_loc.tok->lookahead = tok->lookahead;
    Treat this token as re-lexed during lookback update phase.
    force_lookback_recomputation(construction_loc.tok);
    last_reused_token = construction_loc.tok;
    return true;
  }
  return false;
}
```

Fig. 17. Computing bottom-up reuse during incremental lexing. The `next_new_token` function is modified to apply this test to all tokens passed over when updating the construction location through calls to `advance`. If an old token can be reused, the new information is copied into its fields. Out-of-order reuse could also be attempted, but top-down reuse is much more effective at performing such global comparisons and is easier to implement.

type of a token being eliminated, the old token can be reused by copying the `lexeme`, `state`, and `lookahead` fields from the new token (the new token is then discarded). The algorithm used by our prototype is shown in Figure 17.

Bottom-up reuse captures the reuse cases that can be easily discovered using only local information, and it is both efficient and simple to implement. Bottom-up reuse is also necessary to allow the parser to reuse interior nodes—it is possible to exhibit situations where bottom-up reuse by the incremental lexer is necessary to 'seed' bottom-up reuse of interior tree structure. However, the inability to consider global comparisons between the old and new versions of the token stream can result in missed opportunities for reuse. (Figure 18 illustrates a case where bottom-up reuse alone is insufficient.) We therefore apply another type of reuse based on structural comparisons, referred to as *top-down reuse*.

Unlike bottom-up reuse, which is performed in parallel with incremental lexing, top-down reuse is performed as a fourth pass, after incremental lexing and parsing have completed. It begins by applying a reachability analysis, shown in Figure 19, to discover *deleted* nodes: nodes present in the previous version of the tree that have been eliminated in the new version. The actual reuse is performed by the algorithm in Figure 20: for each modified interior node, it attempts to restore isomorphic structure in its descendants. Eventually such a path can lead to a token, which is tested and restored to the tree in the same fashion as interior nodes. In combination, the two types of reuse restore virtually every token that the user would consider unchanged.

7.2 Reversibility

The transformation induced by incremental language analysis, including lexical analysis, can be considered as an update to the program, just as textual or structural editing represent 'transformations'. In a software development environment, the user operations will be undoable, and it is desirable for the

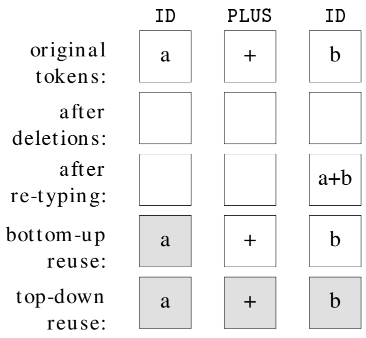


Fig. 18. Example of token reuse. In this scenario, the lexemes from three contiguous tokens are textually deleted and then re-typed (a common occurrence during editing). After re-lexing with only bottom-up reuse, the initial token in the sequence will be reused, but the remaining two tokens will be re-created; any annotations associated with them would then be lost. Applying top-down reuse results in the restoration of all three tokens. In more complicated examples involving incremental parsing, bottom-up and top-down reuse must be used in combination to achieve the best results.

language-based transformations to be undoable as well [Wagner and Graham 1995]. This makes the user interface more comprehensible: every update can be undone using the same interface and every transformation possesses the same semantics. Since we use low-level history services to enable efficient analysis in the first place, it is only natural that they should record the resulting transformation in a manner that is uniform with user-supplied modifications.

To enable efficient reversibility of the incremental lexing operation, all we need to do is ensure that versions of the token stream other than the current one can be selected by the history services without violating correctness. In a typical representation, the history services will already be versioning the lexeme and parent link of each token (the `type` field is read-only information, and therefore is the same in every version of the token). To this we must add the incremental information: the `state`, `lookahead`, and `lookback` fields.[15] (As mentioned in Section 3.6, explicit representation of some or all of this material can usually be avoided.) With this done, the same history services can be used to alter the current version of the program without requiring any involvement from the incremental lexer.

7.3 Error Recovery

As mentioned previously, there is an important distinction between *inconsistency*, which is a transient state where one or more modifications have rendered the token/lexeme relationship potentially invalid within some regions, and *errors*, which indicate character sequences that are not admitted by the language definition. Our model is one where both valid and invalid editing operations are permitted, with the various analysis/transformation tools discovering the maximum amount of information in the presence of any errors

[15]The lookback updating stage already requires that we retain access to the last-lexed value of lookback counts until that pass has completed.

```
Find root of each deleted substructure.
void process_deletions (NODE *node) {
  if (!node->created_in() &&
      node->child_changes(previous))
    for (int i = 0, arity = node->arity(); i < arity; i++) {
      NODE *old_kid = node->child(i, previous);
      if (old_kid->exists() && !node->has_child(old_kid) &&
          (old_kid->parent() == NULL || !old_kid->in_tree()))
        old_kid->handle_deletion();
    }
  if (node->nested_changes(previous))
    for (int i = 0; i < node->arity(); ++i)
      process_deletions(node->child(i));
}

Mark contiguous deleted nodes.
void handle_deletion (NODE *node) {
  int i;
  Copy children so we can discard this node.
  NODE *kids[node->arity()];
  for (i = 0; i < node->arity(); ++i) kids[i] = node->child(i);
  mark_deleted(node);
  Iterate over kids, checking each one for deletion.
  for (i = 0; i < node->arity(); ++i)
    if (kids[i] && kids[i]->exists())
      if (kids[i]->parent() == NULL || kids[i]->parent() == node)
        kids[i]->handle_deletion();
      else if (!kids[i]->in_tree()) kids[i]->handle_deletion();
}

A node is in the a version of a tree if a retraceable path to the root exists.
bool in_tree (NODE *node, VERSION v) {
  if (!node->exists(v)) return false;
  NODE *p = node->parent(v);
  for (; p && p->exists(v) && parent->has_child(node, v); p = p->parent(v));
    node = p;
  return node == root;
}
```

Fig. 19. Reachability analysis for deleted structure. This algorithm locates unreachable nodes—nodes present in the previous version of the tree but not in the current version. The search can be performed in $O(d + s \lg N)$ steps, for d deleted nodes, s modification sites, and N total nodes in the new tree.

Compute top-down reuse for the entire tree.
```
top_down_reuse () {
  process_deletions(root);
  top_down_reuse_traversal(root);
}
```

Apply a localized top-down reuse check at each modification site.
```
top_down_reuse_traversal (NODE *node) {
  if (node->child_changes() && node->exists(previous))
    reuse_isomorphic_structure(node);
  else if (node->nested_changes())
    for (int i = 0; i < node->arity(); ++i)
      top_down_reuse_traversal(child);
}
```

Restore reuse paths descending from `node`.
```
reuse_isomorphic_structure (NODE *node) {
  for (int i = 0; i < node->arity(); i++) {
    NODE *child_now = node->child(i);
    NODE *child_before = node->child(i, previous);
    if (child_now->is_new() && is_deleted(child_before) &&
        child_now->type == child_before->type) {
      child_before->undelete();
      replace_with(child_now, child_before);
      reuse_isomorphic_structure(child_before);
    } else if (child_now->nested_changes())
      top_down_reuse_traversal(child_now);
  }
}
```

Fig. 20. Computing top-down reuse. After locating unreachable nodes using the algorithm in Figure 19, this algorithm visits each modification site and attempts to replace newly-created nodes with discarded nodes along descending paths. The `replace_with` function copies the values in the fields of the new token to the corresponding fields in the reused token.

that arise.[16] Errors may thus also include lexically-valid but syntactically (or semantically) incorrect modifications; the lexer becomes involved because it is the beginning of the analysis phase, and has the primary responsibility for the initial incorporation of modifications into the analyzed program representation.

In a batch environment, a lexical description often includes a simple scheme for handling characters that cannot be incorporated by other ('normal') patterns. The rule accompanying this default pattern then emits an error message. A similar solution can easily be provided in an interactive domain by defining a distinguished token type to represent unmatched text. (Either the pattern in the lexical description or the incremental run-time service should ensure that contiguous unmatched characters are always combined into a single 'unmatched text' token.) The representation of unmatched tokens in the parse tree can be handled similarly to those for explicit whitespace (the details are beyond the scope of this presentation, but can be found in Wagner and Graham [1996c]). In the lexical description for our running example (Figure 2), the final pattern absorbs unmatched characters.

A similar approach can be taken to programmer-supplied error patterns, which typically operate by recognizing a superset of the actual language and then distinguishing correct lexical structures from 'near misses'. A simple example would be a language that limited the length of identifiers; the rule could determine the number of characters in the pattern text and construct either a normal or erroneous identifier depending on the result. Each error pattern has a typed token to represent it, and thus communicates all the available information to subsequent analysis tools. No special support is required for detecting or handling errors in this fashion in the incremental lexer.

A more general approach to error recovery uses the interactive and history-based nature of the ISDE to its full advantage. We observe that lexical and grammatical errors are distinct from semantic inconsistencies (scoping programs, type violations) in that they typically make it impossible to incorporate the latest modifications into the program representation in a way that results in both valid tokens and valid tree structure.

The solution is to avoid incorporating invalid modifications. If the incremental lexer or incremental parser do not recognize a valid (sub) string of the language, the modifications that caused the problem are left *unincorporated*. Erroroneous modifications can safely co-exist with legal modifications; the presence of an error does not prevent the incorporation of other, legal modifications in different areas. (Naturally, several closely-spaced modifications may remain unincorporated if any one of them is invalid.)

The lexer plays an important role in this approach, because it has the primary

[16]Another solution is to prevent erroneous modifications from being made. While we feel that restrictive, generative approaches to software development are unnecessary and undesirable, the algorithms described here can be easily applied to this editing model as well. By running the lexical analysis algorithm in a *read-only* mode, an alteration can be checked for compatibility, with the analysis algorithm returning an error indicator instead of permitting an invalid update. Alternatively, lexical analysis can be allowed to proceed normally, followed by a post hoc check to determine whether all transformations it induced were legal; if not, some or all of the transformations can be efficiently discarded [Wagner and Graham 1995].

task of locating modifications that affect the direct program representation. Normally it does so by interrogating the history services to locate modifications since the last-lexed version. When persistent errors are allowed, however, they must be treated differently. A simple solution treats unincorporated changes as modifications that are 're-applied' immediately subsequent to the conclusion of incremental lexing; this means that any erroneous modification will be re-tried the next time incremental lexing is invoked. More efficient solutions use historical information and incremental parsing theory to limit the potential range of each error; the details of history-based syntactic error isolation appear in Wagner and Graham [1996b].

8. CONCLUSION

The algorithm presented in this paper is the first published technique for language-independent incremental lexical analysis that supports the full pattern set of conventional batch generators and runs in optimal space and time. It thus provides the maximum amount of expressiveness, enabling the lexical characteristics of real programming languages to be described in a natural manner without requiring either the language description writer or the lexical generator to compute a limit on the length of inter-token dependencies. Existing lexical analyzer generators can be used without modification. The performance of our automatically-generated incremental lexers rivals hand-written approaches and the generation process itself is fast enough to enable rapid debugging and prototyping of new lexical descriptions.

In an interactive software development environment, our approach to incremental lexing retains useful information and minimizes changes through aggressive reuse computation, using both bottom-up and top-down approaches. The incremental lexer is designed to operate as a subroutine of the incremental parser to handle languages lacking the separate analysis property. Error recovery and the efficient reversibility of the lexing transformation are also supported.

ACKNOWLEDGMENTS

Special thanks to Vern Paxson, for assistance and comments on early drafts of this paper and to the ***Ensemble*** developers for helping to create an appropriate testbed for this project.

REFERENCES

BAHLKE, R. AND SNELTING, G. 1986. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst. 8,* 4 (Oct.), 547–576.

BALLANCE, R. A., GRAHAM, S. L., AND VAN DE VANTER, M. L. 1992. The Pan language-based editing system. *ACM Trans. Softw. Eng. and Meth. 1,* 1 (Jan.), 95–127.

BEETEM, J. F. AND BEETEM, A. F. 1991. Incremental scanning and parsing with Galaxy. *IEEE Trans. Softw. Eng. 17,* 7 (July), 641–651.

DELISLE, N. M., MENICOSY, D. E., AND SCHWARTZ, M. D. 1986. Viewing a programming environment as a single tool. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments.* ACM, ACM Press, New York, 49–56.

FISCHER, B., HAMMER, C., AND STRUCKMANN, W. 1992. ALADIN: A scanner generator for incremental programming environments. *Software—Practice & Experience 22,* 11, 1011–1025.

FISCHER, C. N., JOHNSON, G. F., MAUNEY, J., PAL, A., AND STOCK, D. L. 1986. The Poe languages-based editor project. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*. ACM, ACM Press, New York.

GAFTER, N. M. 1990. Parallel incremental compilation. Ph.D. thesis, University of Rochester, Rochester, N.Y.

HEERING, J., KLINT, P., AND REKERS, J. 1992. Incremental generation of lexical scanners. *ACM Trans. Program. Lang. Syst. 14,* 4 (Oct.), 490–520.

HORWITZ, S. B. 1985. Generating language-based editors: A relationally-attributed approach. Tech. Rep. TR 85-696, Department of Computer Science, Cornell University.

JACOBS, I. AND RIDEAU-GALLOT, L. 1992. A Centaur tutorial. Tech. Rep. 140, INRIA. July.

LARCHEVÊQUE, J. M. 1995. Optimal incremental parsing. *ACM Trans. Program. Lang. Syst. 17,* 1, 1–15.

LESK, M. E. AND SCHMIDT, E. 1979. LEX—a lexical analyzer generator. In *Unix Programmer's Manual*, 7th ed. Bell Telephone Laboratories.

PAXSON, V. 1995. `flex 2.5.2` man pages. Free Software Foundation.

REPS, T. W. AND TEITELBAUM, T. 1989. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. LNCS. Springer-Verlag, Berlin.

ROSKIND, J. 1992. Parsing C, the last word. article 92–01–056 of `comp.compilers`.

ROSS, G. 1986. Integral C – A practical environment for C programming. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*. ACM, ACM Press, New York, 42–48.

SZAFRON, D. AND NG, R. 1990. LexAGen: An interactive incremental scanner generator. *Software—Practice & Experience 20,* 5 (May), 459–483.

WAGNER, T. A. AND GRAHAM, S. L. 1995. Integrating incremental analysis with version management. In *5th European Softw. Eng. Conf.* (Sitges, Spain). LNCS. Springer-Verlag, Berlin, 205–218.

WAGNER, T. A. AND GRAHAM, S. L. 1996a. History-sensitive incremental parsing. Submitted to *ACM Trans. Program. Lang. Syst.*

WAGNER, T. A. AND GRAHAM, S. L. 1996b. Isolating errors—a history-based approach to incremental language analysis. In preparation.

WAGNER, T. A. AND GRAHAM, S. L. 1996c. Modelling explicit whitespace in an incremental sde. In preparation.