11. Reusable Incremental Scanning

W.M. McKeeman and Shota Aki
Digital Equipment Corporation
110 Spitbrook Road
Nashua, NH 03062

Abstract
The design of a scanner for standard C that is independently reusable and incremental is presented. It is claimed that similar designs are possible for other compiler components and other programming languages. This scanner uses reusable components for source line management and string management. The lexing algorithm itself is not the main issue—it can be conventional. Reusability is achieved by separation of function, simple and consistent interfaces, and provision of component-level testing. Incremental capability is achieved by saving lists of tokens corresponding to blocks of source lines and, when the source is unchanged, reusing the tokens instead of rescanning.

Introduction
This paper presents the design of a scanner that is reusable, and can be line-at-a-time incremental.

There are two reasons to do incremental scanning. First, assuming that changes are much smaller than the whole size of the file, incremental scanning is faster—the time to update the token representation of the source is proportional to the size of the change made to the source. Second, incremental scanning enables the consumer of tokens downstream to perform its processing incrementally and therefore faster as well.

The advent of very fast central processors provides processing power that could make the performance gains of incremental computation irrelevant, although experience indicates that there is no limit on how many processor cycles we can use. Another perhaps more durable efficiency issue is minimization of file I/O. In a large memory environment, this scanner needs only infrequent access to files.

The reasons to promote reuse in general are obvious. It is not so obvious how to achieve it. Reuse depends on a delicate tradeoff between generality and specificity. Progress in reuse comes when one discovers a generalization (widely applicable component) which does not significantly degrade any one
use (efficiently applicable component). Components can be made reusable by paying attention to separation of function and to interface design. Reuse also depends on reducing the engineering effort to understand and incorporate an already implemented component into a larger whole. For this the designer can provide consistency between similar interfaces in both form and interpretation as well as amenities such as documentation and a demonstration.

This paper provides the signature and semantics for the interface of a reusable scanner. The demonstration of implementation adequacy assumes the existence of a driver program (or standalone test jig, in our parlance) which exercises all of the scanner functions. It turns out that there is synergy between making a component reusable and making the same component incremental. Both attributes require clear and narrow interfaces. Our approach is consistent in that it not only provides a reusable scanner interface but also assumes the availability of reusable components for handling strings and input text.

This scanner is usable in a compiler front end and also available for other uses such as pretty printers. The treatment here is for scanning standard C but the solution is generally applicable to popular programming languages with a few language-specific adjustments [2]. Ironically, K&R C is one exception to the general rule—with K&R, scanning is not easily separated from preprocessing.

The definition of a programming language usually includes a lexical grammar which specifies the transformation from character-stream source text into a token-stream needed for the next stage of processing. Even though there are tools for automating scanner construction based on a lexical grammar, most scanners are written by hand. It is easy to do and compiler writers perceive that generated scanners are less efficient [7, 8]. It is also true that most uses require some ad hoc tweaks to the scanner. Scanner generators may supply workarounds for the known needs to tweak, but cannot guarantee workarounds for future needs. The developer is therefore uncertain about whether a scanner generator will be up to the next task. For the work reported here, the next tweak was incremental processing. No generator provided the capability. This line of reasoning led us to use a lexical grammar for documentation while implementing the corresponding scanner by hand [2, §2.1.1.2, §A.1].

The paper is organized in the following sections:

- The scanner interfaces.
- The internals of the scanner.
- Example: a non-incremental consumer.
- Example: an incremental consumer.
- Performance.

The literature on incremental and parallel language processing overlaps because both kinds of processing use similar kinds of incremental structures. The
reader is referred to two comprehensive surveys of the literature for more information [3, 6]. One case study combines scanning and preprocessing, enabling either K&R-style or Standard C-style scanning to be chosen by the user [4]. Another prototype defines an editor interface that gives single character granularity for a simple language [1]. This paper presumes the reader has access to a description of Standard C [2, 5].

Much of the detail in the paper is taken from our implementation of an incremental scanner for C. Where we felt the presentation would be improved, or in retrospect we could have done a better job on the scanner, we have presented the view of how it should be, rather than how it is. This paper does not say much about parts of the scanner that are conventional since that information is available in any standard text.

**Scanner Interfaces**

The scanner interfaces are presented as a set of reusable abstractions. The form of, relation between, and naming conventions for these abstractions are part of the content of this paper. Each abstraction is organized about a principal data structure. The name of that data structure also names the abstraction and prefixes functions defined in the abstraction. These conventions are used throughout without further mention. (See Token, below, for example.)

We also use a convention which we call struct-wrapped handles to enhance the reusability of our abstractions. The use of struct-wrapped handles allows an implementation of an abstraction some freedom in laying out the handles without affecting clients. The actual encoding of any handle (for tokens or other things mentioned below) is private to the implementing component. In fact the representation of the handle is often a struct-wrapped index into an array managed inside the abstraction. This scheme allows position-independent pointers to various kinds of objects, hides the details of the implementation of our memory manager from its clients, enables C compilers to strongly type-check handles, and avoids inadvertent unsafe modifications of these handle values. The use of wrappers should not degrade performance.

**Token Abstraction**

The token abstraction is implemented (in C) in module token.c with public interface token.h.1 The abstraction supplies primitive functions for stepping forward and backward in the source input stream one token at a time, and for examining various attributes of a given token. The user of this abstraction will want to jacket these primitives with a use-specific function, as is demonstrated in the examples later.

---

1In actual use, component source file names are prefixed with the processor name, as in cc.token.c.
The principal data type produced by the scanner is Token. All uses of type
Token except for declarations, assignments, and parameter passing are private
to the scanner. The values of type Token are referred to as token handles.
Used in a C translator, the token output of this abstraction is input to the
preprocessor.

typedef struct {private layout} Token

  t = TokenFirst()  -- iterator functions
  t = TokenNext(t)  -- start at head of current file
  t = TokenPrev(t)  -- step to next token

  c = TokenClass(t)  -- attribute functions
  s = TokenString(t)  -- small integer classifying token t
  s = TokenStringRaw(t) -- textual form of token t
  s = TokenWhite(t)  -- as above, preserving backslash and trigraphs
  s = TokenWhiteRaw(t)  -- white space preceding token t
  s = TokenWhiteRaw(t)  -- as above, preserving backslash and trigraphs
  h = TokenLoc(t)  -- source file location of token t

There is a significant implementation cost implied by the full generality of
this interface. For us the cost has been acceptable. The costs can be reduced
somewhat for simple uses. The iterator functions (above) imply the existence
of forward and backward pointers. If one limits the arguments to the iterator
functions to a window of the previous n unique argument values for some small
n, the pointers do not have to be permanently recorded, therefore permitting
the implementation to save the corresponding storage. For Standard C, it
is sufficient to enable TokenPrev(t) only during the scanning of #include
directives. The attribute functions, on the other hand, must always be callable
with any valid token handle.

Function TokenFirst() returns the first token in the input stream. Given
any token, its successor and predecessor are returned by TokenNext() and
TokenPrev() respectively. There is a special token class represented by end-of-
file (eofTOKEN). This token is returned by TokenFirst() if the input stream
is empty, and by TokenNext(t) if it is called with the last token in the
input stream. There is another token class for not-a-token (ntTOKEN). This
token is returned when TokenNext() is called with the end-of-file token and by
TokenPrev(TokenFirst()).

The small integer value representing the token class is implemented with an
eum in the interface. The enum-valued function TokenClass(t) is used a lot
and must therefore be especially efficient. There are about 40 distinct values
corresponding to the preprocessor input symbols in the C grammar [2, §A.1].
For example there is a class for operator ++, one class for all identifiers, one
class for all preprocessing numbers, and so on. In addition, there are exactly
a dozen identifiers that are keywords immediately following #, but not otherwise reserved in the preprocessor. Finally, there is defined which is reserved following #if and #elif, and new-line which is treated as a token in a preprocessing directive and otherwise as white space. There are about 80 distinct post-preprocessing values including reserved words [2, §A.2].

One solution to all this detail is for the scanner to give a unique class to every different symbol including key and reserved identifiers, and then provide mappings to the codes appropriate to the situation. For example, if has a unique class. Its various mappings and the mapping functions are shown below.

```plaintext
Preprocessing keyword context
(immediately following a # that starts a line)
if (TokenClass(t) == ifTOKEN)...

Preprocessor expression context (following #if, #elif)
if (InPpExpr(TokenClass(t)) == identifierTOKEN)...

All other preprocessing contexts
if (InPpBody(TokenClass(t)) == identifierTOKEN)...

During parsing
if (InParse(TokenClass(t)) == ifTOKEN)...
```

Only the mapping InPpExpr() exposes the special preprocessor arithmetic operator defined to be the token class definedTOKEN during preprocessor expression context.

Another solution is to use different token-class values after preprocessing, constructing the tokens anew at the end of phase 4 [2, §2.1].

Functions TokenString(t) and TokenWhite(t) each return values of type String, which is presented next. The value of TokenString(t) is text of the token itself. The value of TokenWhite(t) is the text of the white space preceding the token. The preceding white space is needed during macro definition and expansion in a preprocessor for Standard C. Any backslash-newline pairs and trigraphs that occurred in the source file do not appear in the values of these functions. The two other versions (suffixed with Raw) provide the exact textual form. Printing the Raw form of the strings for all the tokens in a file, in order, reproduces the source text.

Function TokenLoc(t) returns a value of type Locator which is a further abstraction providing access to token starting and ending column positions, file name and line, and other kinds of information associating token to source-text position. Function TokenLoc(t) is used in compilers principally to place diagnostics relative to the source text. What TokenLoc(t) locates is precisely TokenStringRaw(t) since that is what is in the source file. Further details about locators are not essential to this presentation.

The term white space has a technical meaning [2, §3.1]. It is not affected by the interpretation of the token stream as preprocessing directives.
Token Increment Abstraction

The token increment abstraction is implemented (in C) in module tokinc.c with public interface tokinc.h. It supplies an interface similar to the token abstraction except that it operates on token increments.

A token increment is represented by type TokInc. Values of type TokInc are called tokinc handles or just tokincs. Objects of type TokInc can be declared, assigned, and passed as parameters.

A token increment is an abstraction for managing tokens. The scanner maintains an image of the source file as a list of logical source lines (also known as the tokinc sequence). Usually a real source line corresponds one-to-one with a logical source line in C. However, multiple real source lines turn into a single logical source line when real lines are glued together by a terminating backslash (\) character or by multi-line comments. A tokinc also records the creation-history of the tokens that are associated with it, using a time stamp.

Token increments enable incremental scanning. The granularity of the incremental update is the tokinc. When source changes, the scanner only needs to rebuild the tokincs that correspond to the changed lines in the source.

typedef struct {private layout} TokInc

w = TokIncFirst() — iterator functions
w = TokIncNext(w) — start at head of current file
w = TokIncPrev(w) — step to next increment

TokIncLineDelete(j) — line j has been deleted
TokIncLineInsert(j, a) — line j inserted ahead of line a

attribute functions

T = TokInc2Token(w) — first token in tokinc w
w = TokInc0fToken(t) — token increment associated with token t
k = TokIncMade(w) — time of construction of tokinc w

The tokinc iterators behave in the same manner as the token iterators. When the start or end of the tokinc sequence is exceeded, a special handle value of nTOKINC is returned by the iterators.

Functions TokIncLineDelete() and TokIncLineInsert() are the list manipulation functions to modify the tokinc sequence as the line structure of the source changes (e.g. old lines deleted, new lines inserted). Tokincs are hashed by line handles (see the Line abstraction below) to facilitate fast deletion and insertion operations.

The tokinc abstraction provides two attribute functions. TokInc2Token() returns the leading token of its associated logical source line. Thus the following invariant is always maintained:

TokenFirst() == TokInc2Token(TokIncFirst())
The function `TokIncr0ffToken()` is used in the incremental mechanisms. Given a token, it will return the containing token increment. More details on token increments are given in the next section.

Time stamps, represented by a type `TimeStamp`, are discussed in general in a later section. Function `TokIncMade()` returns the time stamp of the specified token increment. In this case the time stamp indicates the age of the tokens in the tokinc as well. Individual time stamps for each token could have been used but since a new list of tokens is created each time a change to a logical source line occurs, a single time stamp assigned to a tokinc is sufficient.

**Line Abstraction**

The line abstraction is implemented (in C) in module `line.c` with public interface `line.h`. It supplies an interface similar to the previously described abstractions except that it operates on source lines.

A line is represented by a type `Line`. Values of type `Line` are called line handles. Items of type `Line` can be declared, assigned, and passed as parameters.

- The line abstraction provides access to the text of a source file. It is the main input abstraction for the scanner. The actual origin of the text is hidden behind this interface, i.e. it could be from disk or from an editor. There must be a mechanism for selecting the file; the specifics are not important for this paper.

- The following public interface to source text is sufficient for scanning.

```c
typedef struct {private layout} Line

j = LineFirst()  // iterator functions
j = LineNext(j)  // start at head of current file
j = LinePrev(j)  // step to next line

j = Line2Cstring(j)  // attribute functions
h = LineLoc(j)        // text of j, null-terminated
k = LineMade(j)       // source file location of line j
LineCleanSet(j)       // time of construction of line j
b = LineCleanGet(j)   // set line to status clean

```

- The line iterators behave in the same manner as all the other iterators. When the start or end of the line sequence is exceeded, a special handle value `ntLINE` is returned by the iterators.

- Function `Line2Cstring(j)` provides a read-only null-terminated C string for line j. Function `LineLoc()` returns a value of type `Locator` (discussed earlier under tokens).
The function `LineMade(j)` is useful for incremental scanning. It returns a value of type `TimeStamp`. Alternatively, one may use the pair `LineCleanSet(j)` and `LineCleanGet(j)` if the line abstraction supplies a clean/dirty bit instead of a time stamp.

The typical use of source information is to start at the beginning and step through the whole input, one line at a time. On occasion it is necessary to look ahead in the input before processing the current line. For compilers, it is acceptable to restrict all the functions to some small window containing just the last few accessed lines. This is a reasonable compromise with the full generality of the functions.

In Fortran, for example, the lookahead line must be examined for a continuation mark before the current line can be scanned. In C, the scanner cannot process `#include` directives in one pass—there are three possibilities, the third of which applies only when the other two fail. This backup situation is confined to a logical source line, but that may cross many physical source lines because of comments and backslash suppressing line breaks. Because our incremental scanner works on a granularity of physical source lines, resetting to try the third possibility can therefore require stepping back many lines.\(^3\)

**String Abstraction**

The string abstraction is implemented (in C) in module `string.c` with public interface `string.h`.

The type `String` is an abstraction of C `char*` data. It is closely related to the standard library functions [2, §A.6.11]. Variables of type `String` can be declared, assigned, and passed as parameters. Values of type `String` are called string handles.

The string abstraction is used by the scanner to manage the storage of all text. For example, the text-attribute functions for the token abstraction return values of type `String`.

The string abstraction has a collection of functions and holds a collection of strings. Text `c` is registered in the abstraction via one of the function calls `CatString2String(c)` or `Chars2String(c,n)`. The first form expects a null-terminated C string. The second form expects at least a non-null character. The first function is just an abbreviated form of the second. That is `CatString2String(c)` is equivalent to `Chars2String(c, strlen(c))` so long as `c` is null-terminated.

The value `String2CString(TokenString(t))` is a null-terminated char * text of the token `t`. Similarly, `String2CString(TokenWhite(t))` provides a (perhaps empty) null-terminated sequence of comments, blanks, tabs, and newlines immediately preceding the token.

The principal data type is `String`.

---

\(^3\)Because of the 509 character limit for logical source line, there is a limit to the number of lines of backup, but it is too large to be interesting.
typedef struct {private layout} String

s = Cstring2String(c)  -- tabulate null-terminated string
s = Chars2String(c, n) -- tabulate n characters
b = String2Cstring(s)  -- null-terminated char* representation
n = StringEq1(s1, s2)  -- like strcmp() == 0
n = StringLen(s)       -- like strlen()

The characteristics of the package are that StringEq1(s1, s2) is fast and that all variable-length storage is automatically managed, thus relieving the compiler writer (or other users) of that task. Outside the string package, variables can be declared and values can be assigned and passed around with low overhead. Values of type String are only created inside the string package. Because source program input is repetitive (use an identifier once, use it a dozen times...) and only one unique copy of each string need be tabulated, the size of the internal tables grows more slowly than the total size of source input.

The actual package which motivates this writeup has other functions, for context switching, catenation, hashing, testing, and so on. Further details are omitted here.

Scanner Internals

This section presents the technology behind the scanner interfaces. The presentation includes a discussion of timestamps, how lexical analysis is implemented, and a description of how coherency is maintained between the tokens sequence and the line structure of the source.

Figure 1 illustrates the relationship between the main abstractions described in the previous section. The incremental nature of the scanner is revealed as well.

The input to the scanner is the sequence of lines. The scanner builds an image of the input as a sequence of logical lines (type Tokinc). Parallel to the tokinc sequence is the sequence of tokens. Each tokinc provides direct access to the token that starts its logical line.

The consumer of the tokens can traverse the length of the token stream to obtain the token-representation of the source. The scanner can incrementally update portions of the token sequence by rebuilding only the tokincs and tokens associated with the modified lines.

The token increments and associated tokens can be saved, even written to a file. The token-producing interface could be driven directly from the file information without reference to the source. This allows scanning to be separated in time from the use of the tokens, effectively turning the token increments into an intermediate language. An important consequence shows up later in #include directive processing, where rescanning can be avoided by reusing the token increments left over from a previous inclusion of the same file.
Clean, Dirty, Time Stamps

In incremental computation, the dependencies between the components form the basis for reprocessing. If something older depends on data that is newer, the older must be replaced with a recomputed newer value. It must therefore be possible to compare times of creation for all of the atomic increments (e.g., types TokInc, Token, and Line). Dirty bits or time stamps provide the essential information—they are associated with increments of various kinds.

A dirty bit for an increment is set to TRUE by the producer whenever the value is changed. The dirty bit is set to FALSE by the consumer when the value of the increment has been read and assimilated. Write-access to the dirty bit must be given to both the producer and consumer.\(^4\)

A time stamp may be associated with increments. The order of time stamps is the same as the order of time-of-creation for the increments. Typically a time stamp is an int value and the time stamp generator just keeps adding one for each new value. With a 32-bit int, one need not fear of running out of values. A stale increment will have an older (smaller) time stamp than something it depends upon. The clean/dirty information is derived by comparing time stamps. With this solution only the producer needs write-access to the increment. Furthermore, several consumers can share an increment from a single producer. In C, for example, lines in header files are shared by all the modules that include them.

It is necessary that time stamps be correlated across all of the various incremental constructs. For example, the time stamps of lines must be comparable.

\(^4\)This simple device breaks down when there is more than one consumer—each consumer must have a way of separately setting FALSE.
with the time stamps of tokens. Whenever a token is stale with respect to the line it comes from, the old token must be discarded and a new token created in its place with time stamp value at least as young as the new line from which it is taken.

If incremental constructs are cascaded (for example, across editing, scanning, preprocessing, parsing, etc.), it may be convenient for an increment to have two kinds of time stamps, one comparable to its input and one comparable to its output. The two kinds of time stamps need not be comparable with each other. This permits each abstraction to own its own time-stamp generator without any need to coordinate with those in other abstractions.

**Lexical Analysis**

Scanners have three principal responsibilities: lexically separating the input, associating the pieces with the token classes needed by the next stage of processing, and diagnosing a few kinds of input errors. Scanners may also be responsible for detecting and responding to the context in which the tokens are found. For example, in C the text of a string for a header name follows different rules than the text of string literals. Other languages have more complex context dependencies.

Having provided interfaces for the input and output of the scanner, we now show how to isolate lexemes and make tokens. There are many ways to do it. Most readers will have already done it more than one way. We divide the tokens into groups detectable by their first character and switch inside the scanner to cases, one for each group. For C, we use a dozen unique switch values. They are represented by an enumeration local to the scanner:

```c
typedef enum {
    STARTNOTHING,    /* most control characters */
    STARTSHARP,      /* # and ## */
    STARTSINGLE,     /* , ( @ ... */
    STARTMULTIPLE,   /* + = * ... */
    STARTID,         /* ids, wide char & string */
    STARTPPNUM,      /* 0 1 ... dot ellipsis */
    STARTCHAR,       /* ' */
    STARTSTRING,     /* " (string or header name)* */
    STARTCOMMENT,    /* /* */ */
    STARTHEADER,     /* <stdio.h> < <= << <<= */
    STARTWHITE,      /* blank tab ... */
    STARTENDOFLINE,  /* null (from Line2Cstring) */
} LexemeGroup;
```

In each switch case the details of constructing the token are filled in. Within the case for STARTMULTIPLE, for example, there is a nested switch on the leading character, and within these cases there are switches on the next character, and
so on. The following code fragment illustrates the structure for lexing the four C symbols !, =, !=, and ==. The function `CharGet()` is a local function extracting characters from the current line of source input.

```c

    case STARTMULTIPLE:
        switch (c) { /* c is leading character */
            case '!' :
                c = CharGet(); /* discard '!' */
                switch (c) {
                    case '=': /* '!=', */
                        c = CharGet(); /* discard the '=' */
                        class = NEQ; /* token class for '!=', */
                        break;
                    default: /* '!', */
                        class = NOT; /* token class for '!' */
                        break;
                }
            break;
            case '=' :
                c = CharGet(); /* discard '==', */
                switch (c) {
                    case '=': /* '==', */
                        c = CharGet(); /* discard the second '=' */
                        class = EQL; /* token class for '==', */
                        break;
                    default: /* '==', */
                        class = ASG; /* token class for '=' */
                        break;
                }
            break;
        }
    break;

    case ...
        /* other multiples... */
    }

The case on STARTID handles wide string and character literals as well as ordinary identifiers and reserved words. The case on STARTPNUM handles the dot operator and ellipsis as well as `preprocessing-number`

The collecting of the type `String` values for the tokens as well as the processing of backslash at the end of a line and trigraphs is hidden inside function `CharGet()`. The suppressed new-line characters never appear outside `CharGet()` Trigraphs are translated into their equivalents. In the infrequent case that the raw version of the text differs from the normal value, special construction is required. This is also implemented inside `CharGet()`. The line abstraction produces null-terminated C strings for lines, so the new-line character has to be rematerialized for some white space values.
Once the token class has been determined, and all the other information recorded (such as locating line and column), the token is assembled. There is no implication that the data is all packed into a token—the data can be held in internal tables. The Token abstraction provides a scanner-private routine for assembling a token given its class, text, raw-text, whitespace, raw-whitespace, and locator.

This design leads to a module that is big but easy to build and maintain. The design also gives convenient hooks for the language-specific context-dependent situations. Note also that the lexical analyzer is isolated from the rest of the scanner module since its only external interfaces are CharGet() (with calls to Line routines) for input and the Token construction routine for output.

**Token Increment Construction**

In a traditional scanner, the assembled token is returned and used immediately. As a consequence, the scanner holds a partially processed source line between calls.

An incremental scanner will save each token by appending it to a local queue. This queue may be of arbitrary length, which allows the scanner to process text and produce tokens until a natural stopping point is reached (usually the end of a line). This queue-centered design simplifies (at little cost to efficiency) the between-call state-saving of the scanner since there are no partially processed lines of text hanging around. More significantly, the queue-centered design also enables incremental scanning. Each queue-full corresponds to the tokens generated by scanning a logical line (tokinc). The token iterators manipulate these queues to traverse the token stream.

By the nature of the constructing mechanism, each token increment is aligned with the line structure of the source input text so that if the source text is unchanged then so also is the token increment—a clean line leaves a clean token increment. If the source text is changed, then the scanner will create a new queue-full of tokens and a new tokinc, which is inserted in the appropriate location in the tokinc sequence to mirror the new line structure of the source.

The first time the source is scanned, source lines can be processed sequentially from front-to-back. Token increments are produced in the order the lines are scanned, and the resulting tokens are associated with tokincs. After a source file has been scanned, a correspondence between source lines and the token increments is established (as in Figure 1). For C the correspondence is many-one—more than one source line may be needed to make a complete token increment (because of backslash and multi-line comments).

During edit, source lines may remain unmodified, be inserted, or be deleted. All other kinds of edit activity can be expressed as a combination of these. After some edit activity, the previous correspondence between saved increments and source lines is broken—there will be token increments that no longer have a source (deleted source), and source lines that have no token increment (inserted
source). The difference can be expressed as a sequence of deletes and inserts. The sequence is not unique. Two sufficient sequences are the real-time sequence as recorded by the editor and the front-to-back sequence produced by a file differencing algorithm. Either sequence can be used to drive the functions:

    TokInclineDelete(deleted);
    TokInclineInsert(inserted, before);

When a line is deleted, the scanner must free the corresponding token increment, perhaps construct another to replace it, and reestablish the sequence between the remaining token increments. All of this requires that the scanner find the token increment corresponding to the deleted source line. An efficient solution is to provide a hash table taking the source line handle deleted into the corresponding tokinc handle. The hash table is maintained by the token increment abstraction. The inverse operation, finding the source lines from the token increment, requires that the token increment contain the handle of at least one of the source lines it represents.

If the deleted line is part of a multi-line increment, the neighboring lines must be included in the rescanning. For C, if the deleted line is the last line in a multi-line increment, rescanning may swallow the following token increment also. When all of this is done, the correspondence is reestablished.

Suppose, on the other hand, that a line has been inserted—it is new and therefore will be marked dirty. The hashing mechanism cannot look up a new line. The hash is used instead to find the increment containing the information from the source line now following the inserted source line. If the source line following the inserted source line has been newly inserted as well, then a temporarily disjoint tokinc sequence is created. This sequence grows until a source line that corresponds to a tokinc in the main sequence is provided, at which point the entire disjoint sequence is inserted before the tokinc in the main sequence. The previously stated considerations about rescanning and swallowing the following increment and reestablishing the correspondence apply. In addition, inserting a line may cause a multi-line increment to be broken into two separate increments.

The delete/insert notifications can be applied immediately, or queued for later application either all at once or in small batches. Likewise, an editor can delay making the notifications, sending them a batch at a time. This trick allows an editor to locally optimize the queue contents, canceling insert/delete pairs. What is required is that the actions are applied, in order, prior to using the tokens.

If the source lines have no independently assigned time stamps, time stamps can be "invented" by the scanner as the insert notifications arrive. The correspondence between line handles and time stamps is then maintained in a separate data structure in the scanner rather than by the editor.

If the editor provides time stamps or clean/dirty bits for lines, the incremental problem reduces to creating a valid insert/delete sequence from the old
token increments, the new source text, and the change information. The source lines in the editor may be examined in file order. The change information for each source line handle is examined, and perhaps compared with the corresponding record of it in the token increment. If the line is dirty, it must be rescanned—effectively treated as a delete/insert pair.

A disagreement between the new sequence of line handles and the old record of line handles in the token increments means either a line was deleted, or was inserted, or both. Since there is a fast way to go from the source line handle to the corresponding token increment, the correspondence can be checked in that direction. If the offending source line handle is found in a token increment, one may deduce that all lines corresponding to the token increment sequence from the point of disagreement to the found line have been deleted. This results in a sequence of deletes until the handle agreement is reestablished and the correspondence is lengthened one step—therefore insuring that progress is made.

If the source line handle is not found in any tokinc, this source line has been inserted. The line is scanned and its token increment inserted into the token increment sequence, which again insures that progress is made.

The use of change information gives a delete/insert sequence (nearly) identical to the sequence one would get from file differencing.

Example 1: Non-incremental Consumer

The use of tokens can be as varied as the number of users. So long as the reusable primitives supply enough information, any special interface can be built by the user in terms of the primitives.

A sample non-incremental consumer is described below. This consumer uses the scanner interface in a non-incremental fashion, i.e., tokens from the scanner are processed from start to finish without making any use of its history.

Function Init() must be called once to initialize access to this module. It sets a private variable called NextToken to the first token in the source input stream. Function Scan() may then be called to process tokens in input stream order. Function LookAhead() may also be called to examine one token ahead in the input stream without affecting the input stream order used by Scan().

The private variable CurzToken is used to track the most recent token returned by Scan(). Its use will be described in the next section.
#include "token.h"

static Token NextToken;
static Token CurrToken;

void
Init(void) {
    NextToken = TokenFirst();
}

Token
Scan(void) {
    CurrToken = NextToken;
    NextToken = TokenNext(NextToken);
    return CurrToken;
}

Token
LookAhead(void) {
    return NextToken;
}

A non-incremental consumer may use these routines to display all the tokens
from the source input. Function Scan() is used to get each token. The con-
sumer terminates processing when EOF_TOKEN is returned by Scan(). Function
LookAhead() may be used to control formatting of the current token based on
what follows it. This example is contrived but it does illustrate how the scanner
interface may be reused in a simple situation.

One final point, even though the consumer is non-incremental, the scanner
is managing its token-representation of the source incrementally. Thus, the
consumer gains some efficiency in performance since the scanner does not have
to work as hard.

Example 2: Incremental Consumer

The previous example is extended in this section to describe an incremental
consumer. This consumer uses the creation-history of the tokens from the
source input stream. That is, if a section of the input stream has not changed
since the last time it was processed, then the consumer may skip over it.

The interface and associated semantics used by the non-incremental con-
sumer may be reused by the incremental consumer. That is, Scan() and
LookAhead() are used as previously described to process each token in the
input stream the very first time. They are also used to process sections of the
input stream that have changed.

Two new routines are introduced. Boolean function CleanInput() returns
TRUE to indicate that the consumer may skip over the current section of the
input stream (since it has not changed) and reuse the corresponding output rather than producing it over again. A return value of FALSE indicates a change has occurred and that new output will needed. Function SkipInput() is the incremental analogue of Scan(); it returns the first token that immediately follows the skippable section of the input stream. It should be called only if CleanInput() returns TRUE.

The unit of skipping is the token increment. Recall that a token increment describes a logical source line and therefore a contiguous sequence of lines within the input stream. More than one token increment may be needed to form an increment for the consumer. The routines CleanInput() and SkipInput() allow the consumer to process and skip-process over the input stream in multiples of token increments.

The routine TokInc0fToken(t) may be used to track the sequence of token increments associated with the tokens returned by Scan() and SkipInput().

#include "tokinc.h"

... reuse Non-incremental Consumer code ...

int
CleanInput(int skipSize, TimeStamp maxTS, TokInc ess) {
    TokInc ti;

    ti = TokInc0fToken(CurrToken); /* get container */
    for (;;) {
        if (ti == ntOKINC) break; /* off end of sequence*/
        if (TokIncMade(ti)>maxTS) break; /* out of date */
        if (--skipSize == 0) break; /* end of subsequence */
        ti = TokIncNext(ti); /* OK, move ahead */
    }
    return (skipSize==0) && (ti==ess); /* reached end? */
}
Token
SkipInput(TokInc ess) {
    TokInc ti;
    ti = TokIncNext(ess); /* skip past this */
    if (ti == nTOKINC) { /* off end, find eof */
        CurrToken = TokInc2Token(ess);
        for(;;) {
            if (TokenClass(CurrToken) == eofTOKEN) break;
            CurrToken = TokenNext(CurrToken);
        }
    } else { /* normal tokinc */
        CurrToken = TokInc2Token(ti); /* first token */
        NextToken = TokenNext(CurrToken); /* prep lookahead */
    }
    return CurrToken; /* start next tokinc */
}

When CleanInput() returns FALSE, the scanner is called to process the section of the input stream that has changed. When new output is produced, the consumer must record the following attributes of the section of the input stream that created it:

- **skipSize** specifies the length of the token increment subsequence processed to produce the output (1 or greater).

- **maxTS** specifies the maximum time stamp from the token increment subsequence.

- **sss** specifies the token increment that starts the subsequence. Function CleanInput() uses CurrToken to infer sss.

- **ess** specifies the token increment that terminates the subsequence.

When the same section of the input stream is revisited by the consumer the next time around, the attributes are passed to CleanInput() to determine if it can be skipped. The consumer must remember the token that starts the section so that the next time around it will know when to call CleanInput(). The code for CleanInput() detects the insertion, modification, and deletion of logical lines to the section of the input stream described by the attributes.

The consumer is responsible for skipping over and managing its output data structures.

**Performance**

The reusable interface without the incremental capability is competitive in speed with traditional scanners. Used incrementally, the scanner is much faster.
The cost for this efficiency is space to manage all of the data structures in memory. There are three main consumers of memory in the scanner: the String, TokInc, and Token abstractions. The amount of memory consumed is proportional to the size of the source being scanned.

We have developed a scanner for Standard C along the lines outlined in this paper. The scanner has not seen hard use and is therefore not a very good source of performance data. On the other hand, it is the only source of data we have. With that disclaimer in mind, we state that our incremental scanning is about 14 times faster than our first-time scanning. The cost is roughly the ratio of checking clean/dirty information (via a call to the editor) to the cost of checking plus rescanning dirty text. The memory use, taken from a single data point (scanning the implementation of the lexical analyzer itself) is given below.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>source size</td>
<td>50,688 bytes</td>
</tr>
<tr>
<td>string table</td>
<td>6,545 bytes</td>
</tr>
<tr>
<td>token increments</td>
<td>45,942 bytes</td>
</tr>
<tr>
<td>token table</td>
<td>200,606 bytes</td>
</tr>
</tbody>
</table>

One can make tradeoffs in space versus speed by adjusting the granularity (size) of the increments. The finer the increment, the more memory is consumed. The coarser the increment, the less incremental the scanner. However, coarser increments can provide larger skips for the consumer of the scanner.

Conclusions

It is feasible to separate the scanner out of a compiler so that it can be used for other purposes. A scanner can be made incremental at approximately the granularity of source lines. The combination of these two capabilities provides a useful and flexible component for an organization dealing with processing of programming languages.

From the viewpoint of the traditional consumer of scanner output, and aside from efficiency, it does not matter whether the scanner saves and reuses tokens or recreates them from the source text. The scanner is therefore at liberty to implement any incremental mechanism consistent with its interface.

The speedup provided by incremental scanning is real, but not likely to be significant relative to the other costs such as preprocessing, parsing, symbol analysis, and so on. On the other hand, having done line-granularity incremental scanning, we have also gathered the time-stamp information enabling incremental processing for following passes. We expect the speedup ratio for a whole system to be better than 14. Our preliminary C preprocessor and parser data support this expectation. Details will follow in a later paper.
References


William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. He can be reached at mckeeman@tle.dec.com.

Shota Aki is a Principal Software Engineer at Digital. He was a principal developer of the VAX APL interpreter. His current interests are in the areas of CASE, Analysis and Design Methods, Compilers, and Tools. He can be reached at aki@tle.dec.com.