LexAGen: An Interactive Incremental Scanner Generator

Duane Szafron
Randy Ng
Department of Computing Science, University of Alberta, Edmonton,
AB, T6G 2H1 CANADA

SUMMARY

This paper describes LexAGen, an interactive scanner generator which is the first component of an interactive compiler generation environment. LexAGen can generate fast scanners for languages whose tokens can be specified by regular grammars. However, LexAGen also supports several context sensitive programming language constructs like nested comments and the interaction between floating point numbers and the range operator in Modula-2. In addition, LexAGen includes a fast new algorithm for keyword identification. However, the most important and novel aspects of LexAGen are that it constructs scanners incrementally and that specifications can be executed anytime for validation testing.

LexAGen specifications are expressed and entered interactively in a restricted BNF format (no left recursion). All syntactic errors and token conflicts are detected and reported immediately as LexAGen incrementally constructs a deterministic finite automaton to represent the scanner. At any time, the user can test the scanner fragment which has been entered by supplying text to be scanned. Alternatively, the user can generate a C-code scanner from the automaton. The generated automaton uses a direct execution approach and is quite fast.

LexAGen is implemented in Smalltalk-80. Its extensive use of interactive graphics makes it very easy to use. In addition, the object-oriented paradigm of Smalltalk-80 is the basis for the incremental analysis, error detection scheme and an intermediate representation which can be easily modified to generate scanners in other target languages like Pascal, Modula-2 and Ada.

KEY WORDS: Scanner Generation, Compiler Generation, Smalltalk-80, Incremental, Interactive

INTRODUCTION

The scanning process is so well understood that scanners can be generated automatically from token specifications. Many scanner generators exist, especially as parts of compiler writing systems, like the scanner generator Lex[1] which can be used with the parser generator Yacc[2] in most UNIX systems. Other representative scanner generators include GLA[3] and Mkscan[4].

There are three important facets to scanner generation: the ease of use of the scanner generator, the generality of the scanner which can be generated and the speed of the generated scanner. Most current scanner generators are deficient in one of these facets. For example, some generators are capable of generating fast specialized lexical analyzers while others are capable of generating slow but general ones. Most have poor user interfaces and error reporting facilities. This paper describes a new lexical-analyzer generator called LexAGen[5] which alleviates these three deficiencies. LexAGen was developed as a stand-alone tool for general use, but it is especially well suited for the recognition of common programming languages. LexAGen is not a batch-oriented generator in which a specification is translated into a lexical analyzer after the specification is complete. Instead, LexAGen incorporates the philosophy of integrated programming environments which incrementally create software while reporting errors as they occur.

LexAGen uses its graphical user interface to guide this incremental process. The user specifies a lexical analyzer using restricted BNF productions (no left recursion), and LexAGen incrementally implements this specification as a deterministic finite automaton. At any time, the user can modify and test the scanner or transform the internal representation into a stand alone C-Code program. The most important and original aspects of LexAGen are that it constructs scanners incrementally and that specifications can be executed anytime for validation testing.

Finite automata (FA) serve as a good model for the scanning process where each input character represents a transition and each state corresponds to a partial token. Each FA is either deterministic (DFA), in that there is at most one transition from a state for each input symbol or a non-deterministic (NFA), in that more than one transition from a state can be labelled by the same input symbol. Although the same set of languages can be recognized by DFA and NFA, DFA are typically faster than equivalent NFA, but they are much bigger. LexAGen uses DFA since in most applications speed of generated scanners is more important than size.

There are two general approaches that are taken when FA are used for lexical analysis. They are called *interpretation*, and *direct execution*. In the interpretation approach, all transitions are grouped together and represented by a single matrix, *table*[*state*, *symbol*], which is interpreted during lexical analysis by a driving program. In the direct execution approach, an FA is directly implemented as a high-level language program where states are represented as different locations in the code and transitions are represented by case statements. There are two major advantages to this approach. Only the non-error entries in the transition table need to be programmed and enhanced performance in terms of speed[6, 7]. LexAGen uses direct execution.

TOKEN SPECIFICATION

In LexAGen, the user interactively edits BNF productions to specify the context free constructs of a scanner. However, special mechanisms are provided for specifying context-sensitive constructs that are common in programming languages.

In BNF notation there are two kinds of symbols, terminal symbols which are strings of characters, and non-terminal symbols which are symbols that represent a named string of non-terminal symbols. Non-terminal symbols are enclosed in angle brackets, <>, so they can be differentiated from the terminal symbols. Besides the angle brackets, two other meta-symbols are used. The vertical bar, |, is used to represent alternation and ::= is used to define non-terminals using productions. For example, five productions can be used to define Modula-2 identifiers:

```
<identifier>              ::= <letter> | <letter> <identifier body>
<identifier body>         ::= <alphanumeric> | <alphanumeric> <identifier body>
<alphanumeric>            ::= <letter> | <digit>
<digit>                   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>                  ::= a | b . . .
```

The use of BNF notation does not overly restrict the expressive power of the lexical-analyzer generator. BNF offers a good notational framework for context-free grammars and since context-free grammars are a superset of regular grammars, BNF is actually more general than regular expressions. In LexAGen however, the BNF productions are restricted in that recursive non-terminals can only appear as the rightmost symbol in a production. This means that the context-free grammars are restricted to be right-linear and are therefore equivalent to regular grammars[8]. The restriction of right linearity serves to eliminate cycles in the DFA representation of the specification. There are two reasons we have used BNF instead of regular expressions: BNF clauses map more directly into the internal format we have used and we plan to design a parser generator companion for LexAGen which uses BNF and we would like to maintain a common notation between them.

When BNF is used to specify tokens, one additional pair of metacharacters is needed. While some non-terminals represent tokens, others are used to make the definitions more readable or to give names to commonly occurring sub-expressions. For example, the name *digit* may be used to represent the set of characters from zero to nine, even though digits are not usually tokens. The two types of non-terminals must be differentiated since the generated scanner must only return tokens. The metacharacters, << and >>, are used to identify those non-terminals that are tokens.

LexAGen has been designed to cope with the lexical structure of programming languages. Unfortunately, neither restricted BNF notation nor regular expressions are capable of representing all of the tokens which are commonly found in programming languages. For this reason, LexAGen supports three kinds of special tokens.

The first kind of special token is used to specify identifiers and keywords. Although an identifier is a general token which may be specified using BNF, keywords often have to satisfy the specification for identifiers and yet must be differentiated as different tokens. LexAGen provides a special mechanism for specifying those identifiers which are keywords.

The second kind of special token is used to specify strings and comments. Sometimes, it is necessary to exclude some characters from the semantic value of a token. For example, string delimiters are not part of the string token. Sometimes it is necessary to ignore some text altogether, like a comment. Although simple comments can be specified using restricted BNF notation, nested comments cannot. Since some programming languages allow nested comments, a special mechanism must be used to specify them. LexAGen allows both strings and nested comments to be specified. The key to recognizing nested comments is to keep track of the nesting level.

The third kind of special token is one in which simple look-ahead is required to resolve ambiguity. For example, the sequence ">=" may be interpreted either as a single token or as two tokens, ">" and "=". LexAGen, like most lexical-analyzer generators, resolves this ambiguity by choosing the longest match to recognize the token ">=" in this example. However in some other cases this method is not appropriate or adequate. In Modula-2 for instance, the sequence "123.." would be recognized as the real constant, "123.", and the decimal constructor ".". The correct interpretation is the cardinal constant "123" and the range operator "..". This problem has been addressed and solved in the Alex scanner generator[9].

In LexAGen, the user can force this the correct interpretation by attaching two character look-ahead to the end of an alternative. When the second look-ahead character is found, both look-ahead characters are left in the input stream (as untouched) and the production's token value is returned. For example, the Modula-2 range operator problem can be solved using the production:

```
<cardinal>                  ::= <unsigned integer><..
```

where the metacharacter, <, has been used to indicate that the next two characters are the look-ahead characters. When the first dot is found, the next character is examined in the usual manner. If it is a dot, then the look-ahead is successful and the first dot becomes the current character. The contents of the token buffer consist of the string "123", and the token value, CARDINAL, are returned to the client application. The next token found will then be the range operator "..". This mechanism is really a two-character look-ahead since both dots are examined in capturing the token, "123", while the mechanism of longest match essentially involves one-character look-ahead.

LexAGen is currently restricted to a two-character look-ahead scheme since two-character look-ahead is sufficient for most modern common programming constructs. However, it is easy to modify LexAGen to support an alternative fixed number of look-ahead characters.

THE ENVIRONMENT

The most important innovation in LexAGen is the assistance it provides to guide the user through the process of design and testing. This assistance is based on a graphical user interface, and this section describes the special features of LexAGen which are based on that interface. A more complete description of the LexAGen environment appears in the LexAGen User's manual[10].

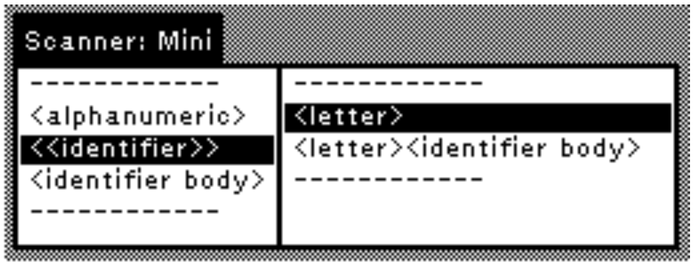
LexAGen is the first component of an envisioned integrated compiler generation environment. Integrated programming environments are usually described as those that support software creation, modification, execution, and debugging. One goal of integrated programming environments is to build tools that share a common internal representation of the underlying software structure. A second goal is to present a consistent user interface across tool boundaries. However, there is a third goal which is just as important. An integrated programming environment should encapsulate the mechanisms used to implement the environment's functionality (not just the internal structure) so that they can be re-used throughout different parts of the environment. For example, the same syntax checking mechanism can be used throughout the environment. LexAGen accomplishes all three of these goals.

In general, graphical user interfaces can have many positive effects on programming environments: error reduction, simplified incremental analysis and efficient debugging[11]. LexAGen is unique in applying these benefits to scanner generators. LexAGen is implemented in Smalltalk-80[12], and the user interface is based on the Smalltalk-80 interface. The user interacts with LexAGen using special windows, pop-up menus[13], and dialog boxes.

The User Interface

The user interface uses windows to capture and display information and menus to capture commands. There are five main windows: the scanner window, the alternatives collector (window), the state diagram window, the execution window and dialog boxes (entry windows). In addition, there are several less frequently used windows which will not be described in this paper. Each window has one or more panes with context sensitive hierarchical pop-up menus. The menus are context sensitive in two ways. They are sensitive to which pane contains the cursor when the menu pops up, but they are also sensitive to the current internal state represented by the pane. This allows LexAGen to display those menu operations that are applicable at a given time for a given context and reduces the number of errors which the user can make.

The scanner window is divided into the left and right panes. The names of non-terminals appear in the left pane. A non-terminal from the left pane can be highlighted by selecting it so that its alternatives appear in the right pane. Figure 1 shows the scanner window at some point during the specification of a scanner for a language called Mini .


Scanner: Mini

```
------------          ------------
<alphanumeric>        <letter>
<<identifier>>        <letter><identifier body>
<identifier body>     ------------
------------
```

Figure 1 The LexAGen scanner window

Some non-terminals are tokens and others are not. When the user declares that a non-terminal to be a token, the single angle brackets in the left pane of the scanner window are replaced by a pair of double angle brackets. For example, <<identifier>> in figure 2 is a token while <alphanumeric> is a non-terminal which is not a token. Note that the user can also "un-declare" a token at any time by choosing the appropriate menu command. For each token, the user can also specify a token number to be returned by the final lexical analyzer. If no token number is assigned by the user, then a unique default value is generated.

Each LexAGen command takes a fixed number of arguments. If a command has several fixed options then LexAGen presents a hierarchical menu with the appropriate options. However, if a command requires input from the user, LexAGen presents a dialog box requesting the necessary input. An alternatives window is used to enter production alternatives. For example, figure 2 shows the alternatives window which was used to enter the alternative, <letter> for the non-terminal, <<identifier>> which is highlighted in figure 1. The syntax error shown in figure 2 will be discussed later.

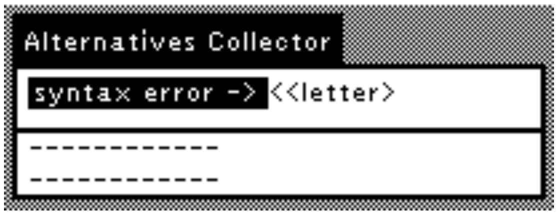


Figure 2 An alternatives window, showing a syntax error

LexAGen currently allows the user to designate one (and only one) production as the rule for keywords. Usually this production is the single production for identifiers. After this production has been chosen, the user can enter keywords which are lexically equivalent to identifiers. Each keyword entered is checked to ensure that it satisfies the rule but a unique token value is returned for each keyword. The new keyword identification algorithm described in the fifth section of this paper is used for this. On the other hand, keywords which are not lexically the same as identifiers (like Algol's) can be specified as separate tokens.

To specify strings, the user choses a string specification command. LexAGen then queries the user for a string delimiter. In addition, the user is queried as to which approach to use for embedded string delimiters. The user has a choice of doubling the delimiter or specifying an escape character. Note that the user can chose the string specification command more than once to specify multiple string delimiters like the apostrophe and double quote of Modula-2.

To specify comments, the user choses a comment command and is then asked for opening and closing delimiters. The command can be chosen more than once to specify multiple comment delimiter pairs like the pairs, {} and (* *), which are supported in many Pascal compilers.

Two-character look-ahead for an alternative is specified by choosing a command from the right pane menu, and the look-ahead characters are entered in a dialog window. The look-ahead characters are then attached to the end of the alternative with the metacharacter, <, prefixing them.

Incrementality

The major goal of incremental analysis is to avoid re-analyzing an entire structure whenever small changes are made to it. The simplest approach to this problem is to determine the smallest separate unit of incrementality. In LexAGen, the smallest unit of incrementality is the production alternative which is represented by a DFA. In fact, the DFA is the single underlying internal structure in LexAGen. Each production and the scanner itself are also represented by DFA. When the user adds an alternative to a production, syntactic analysis is performed to check for the alternative's correctness and semantic analysis is performed to update the production and other productions dependent on the production being re-defined. The latter analysis is necessary to ensure that all affected DFAs remain deterministic through updating. Furthermore, if the affected DFAs include the DFA representing the entire scanner, then the updating process must also ensure the overall correctness of the scanner.

For example, a token becomes ambiguous if it will accept a string which is accepted by another token. Consider the non-terminals, A and C defined by:

```
<<A>>    ::= ab | <B>b
<<C>>    ::= cb
```

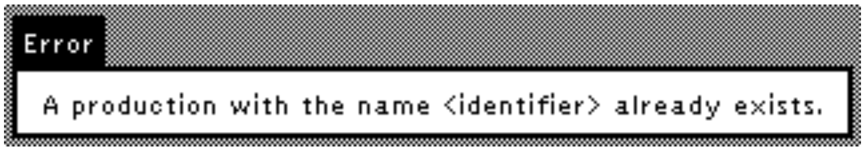
Note that both A and C are tokens. Suppose that the user wants to define <B> ::= c. Since A is dependent on B, it is necessary to update the DFA representing A and check for the correctness of the DFA representing the scanner with the new definition of B. There is a conflict resulting from the fact that when B is expanded in the definition of A, there are two tokens, A and C, which match the ambiguous string, "cb". This prohibits the token definition of B.

Graphical user interfaces have an immediate impact on the issue of incrementality in programming environments. They can simplify incremental analysis by requiring the user to enter information in special dialog boxes. As correct information is accepted, it can be transferred to other windows which maintain a representation of correct structures. Incorrect information can simply be left in the entry windows until other changes result in its final correctness[14].

For example, re-consider the previous token definitions of <<A>> and <<B>>, where the addition of the alternative <B> ::= c would result in a token conflict between the two tokens, A and C. If this alternative was entered into the alternatives window, the incremental analysis would result in an error report and the offending alternative would be left in the alternatives window instead of being moved to the scanner window. If the user discovered that the alternative for A should have been: <B>a instead of <B>b and corrected this alternative, then the contents of the alternatives window representing <B> ::= c could be accepted without having to be re-typed.

This approach helps the underlying environment cope with incorrect information easily, by ensuring that a correct representation of the internal structure is always maintained. An additional advantage is that the user receives immediate feedback as to the correctness of a specification. For example, if a syntax error occurs when an alternative is being added, it is reported immediately as shown in figure 2, and the alternative is not added to the scanner window.

Alert boxes are used to report static semantic errors whenever they occur as the result of information entered in dialog boxes. For example, figure 3 shows an alert box which reports a semantic error when the user tries to add a non-terminal whose name is already used as a non-terminal (appears in the left pane of the scanner window).


Error

A production with the name <identifier> already exists.

Figure 3 A static semantic error caused by information entered in a dialog box.

The only static semantic error which LexAGen allows is the use of undeclared non-terminals in alternatives. For example, the non-terminal, <binary digit>, may be used in an alternative, even though the name, "binary digit", does not appear as a non-terminal in the left pane of the scanner window. This behavior has been allowed for user convenience. In fact, declared and undeclared references share the same internal representation, but they are kept separately. Their existence does not harm the process of incremental analysis, nor the final recognition of expressions. At any time the user may list the names of all non-terminals which have been used but not declared.

Debugging

Graphical interfaces can also have a significant impact on the run-time features of programming environments, especially debugging (see reference 14). Debugging consists of three major activities: selection, viewing, and modification of the software's internal state. Even though LexAGen is a special-purpose programming environment, debugging is a necessary part of the process of generating a scanner. The graphical user interface is used to enhance the productivity of the user and, therefore, speeds up the process of scanner generation. This is especially true in the situation where the scanner specification is not fixed or known as with the generation of scanners for languages being designed. However, even in the case of fixed specifications, LexAGen eliminates the need for context switches between editing, compiling and executing and eliminates the need to write driver programs.

In general, a mouse and a bit-mapped display can be used to select and edit arbitrary programming structures. In LexAGen, the user can select a DFA for viewing or executing by chosing an alternative in the right pane of the browser or the definition of a non-terminal in the left pane. The DFA can either be viewed in a state diagram window or executed in an execution window. At present, LexAGen displays a textual representation as shown in the left box of figure

4, instead of a graphical one as shown in the right box. Nevertheless, this display has proven to be useful for software testing, especially during the development stages of LexAGen itself.

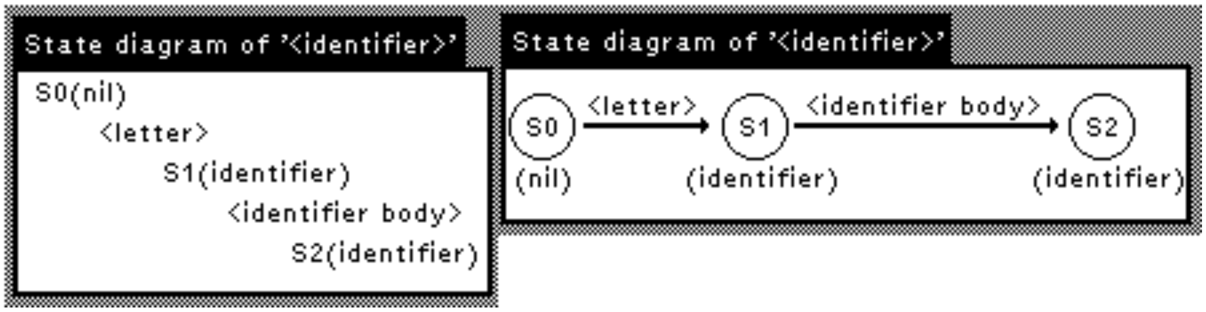

```
State diagram of '<identifier>'
S0(nil)
     <letter>
          S1(identifier)
               <identifier body>
                    S2(identifier)
```

Figure 4 The state diagram for <identifier>, text format and graphical format

LexAGen currently prohibits the direct modification of the states and transitions of a displayed DFA since the smallest unit of incrementality is a DFA alternative. However, it may be possible to relax this restriction by employing the graphics-mode display and extending the smallest incremental unit from an alternative to arbitrary finite states internal to the DFA's structure.

If a DFA is selected and the DFA execution command is chosen, then an execution window is presented. After the user enters an input string into the top pane, the DFA is applied to the input to produce output in the bottom pane as shown in figure 5. Alternately, the DFA representing the entire scanner can be executed. That is, the scanner and its component parts (productions) can be tested and edited incrementally without generating the scanner or leaving the environment!

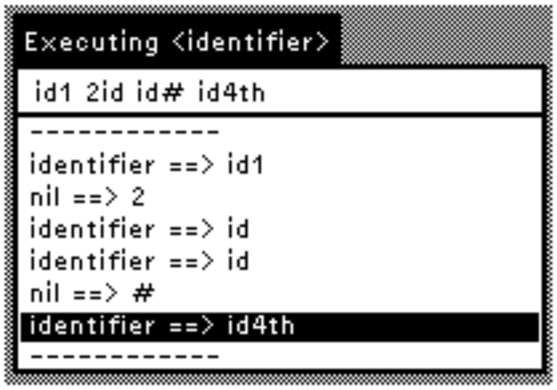

```
Executing <identifier>
id1 2id id# id4th
------------
identifier ==> id1
nil ==> 2
identifier ==> id
identifier ==> id
nil ==> #
identifier ==> id4th
------------
```

Figure 5 Executing the DFA for <identifier>

IMPLEMENTATION

There were two specific design goals that influenced both the functionality of LexAGen and its implementation. The first goal was to design an integrated software environment. The second goal was to represent a specification by a general, uniform, and incrementally editable data structure which could be translated in a straightforward manner to a compact, efficient scanner.

The first design goal was realized by basing LexAGen on the Smalltalk-80 environment and user interface paradigm. The second design goal was realized by using DFAs to represent the specification, implementing the DFAs in Smalltalk-80 and generating a C-code scanner.

The Influence of Smalltalk

Since LexAGen is implemented in Smalltalk-80, it is necessary to understand something about the Smalltalk-80 environment and language in order to understand the implementation of LexAGen. The most profound influence that Smalltalk-80 had on the LexAGen environment came from the structure of the Smalltalk-80 user interface. This structure is referred to as the model-view-controller (MVC) paradigm[15]. Although a Smalltalk-80 user is not strictly forced to use this scheme, the support provided makes user interface construction relatively straightforward.

As the terminology suggests, each component of the user interface is divided into three parts. The first part is the model which represents the application in the traditional scheme of UIMS (User Interface Management Systems). The model contains all of the application dependent information and code. The view and controller compose what is traditionally referred to as the user interface. The view component is responsible for displaying information on the screen. The controller is responsible for accepting user input including: the display of pop-up menus, cursor tracking, and the mapping of user inputs into messages for the application.

The two-pane scanner window shown in figure 1 is an example of the application of the MVC paradigm in LexAGen. The model, AutomataBrowser, is a data structure which contains production information. There is one view for each pane where the left view displays production names and the right view displays production alternatives. The controller displays the context-sensitive menus by querying the browser for information about the selected structures. It also translates menu selections into messages to the model, some of which result in the manipulation of individual productions.

Smalltalk-80 is an object-oriented language[16]. An object is an abstract entity that consists of two parts: its state and its behavior (the set of messages to which it can respond). The state of an object is represented by instance variables whose values are other objects. Behavior is specified using the concept of classes. That is, every object is an instance of a class and the behavior of the object is determined by that class. Finally, Smalltalk-80 organizes its classes using a tree-structured inheritance mechanism, where classes inherit behavior from their parent classes.

Classed objects which inherit behavior have been used throughout LexAGen. Although objects are used for the user interface and code generation, the most important classes are those representing DFAs. Instances of automata are used to represent components of scanner specifications. Incremental editing of DFAs is achieved by defining a set of operations which are implemented as messages. Although the DFAs represent components of the specification, they also represent the scanner as well since in the end, they are used to generate C-code.

In LexAGen, there are three levels of data representation. The first level represents the user interface. The second level represents the scanner specification and ultimately the scanner itself. Finally, the third level is responsible for code generation.

The Automata Browser

Recall that the AutomataBrowser is the model component from the MVC paradigm. This browser is the link between the user and the internal representation of the scanner specification. It stores information in a number of internal tables as well as a single "grand" automaton which represents the complete scanner. Two of the tables are used for the user interface (one for the each pane) and the other auxiliary tables are used to represent properties of the scanner.

The left pane table is essentially a symbol table implemented as a Smalltalk-80 Dictionary (see reference 12 for the definition of a Dictionary). It maintains all information associated with user-defined production names. Each key is a production name and each value is a named automaton representing all the alternatives for that name. The view for the left pane of the browser simply displays the keys of the table.

The right pane table is basically a memory cache implemented as a dictionary, where each key is a literal expression representing the right hand side of a single alternative and each value is a simple automaton representing that alternative. Whenever a name is selected in the left pane, the browser consults the corresponding production for all of its alternatives and records all information about the alternatives (i.e. literal and internal forms) in this table. Whenever a literal expression is selected in the right pane, this memory cache table is used to access the corresponding automaton. For example, if the name <A> had an alternative b<B>, then the literal "b<B>" would be a key in the memory cache table and its value would be the automaton representing b<B>. Alternately, the browser could consult the selected production for the alternatives each time the user makes a selection in the right pane of the browser. However, due to the high frequency of activities in the right pane, the increased look-up speed is worth the extra space taken by the memory cache.

One of the auxiliary tables is used to record the names of undeclared productions. That is, it records non-terminals which have been referenced in productions but do not appear in the left pane. The second auxiliary table is a dictionary used to store information about the values to be returned upon successful recognition of tokens. The keys are token names and the values are the numeric token values to be returned by the generated scanner.

The final two auxiliary tables serve as specification libraries. The first library contains a collection of productions that are commonly used in programming languages. Some examples are:

<digit>, <lowercase letter>, <uppercase letter>, <letter>, and <white space>. This library is read-only, and it provides a quick access to common productions that require extensive enumerations. The second library is a save area for user-defined productions which are general enough that they may be used in more than one scanner. Both of these libraries are implemented as dictionaries with the same structure as the production-name symbol table described previously.

Finally, the browser stores information about the generated scanner in the form of a single grand automaton and maintains a number of special data structures which are used in this scanner. As mentioned previously, LexAGen has incorporated certain special programming language constructs. Specifically, the following information is maintained: the production which defines identifiers, the set of keywords, comment delimiters, and string delimiters.

The Scanner

The second level of representation uses a collection of specialized classes to implement DFAs to represent scanner productions. There are two classes: Automata and AutomataState, along with subclasses of these classes (NamedAutomata and LookaheadState respectively). Each instance of the class Automata denotes a DFA that can represent a single alternative of a production or the entire scanner. Each automaton has two instance variables: lexics and startState. The variable, lexics, is a string referring to the literal form of the underlying automaton. The instance variable, startState, contains an instance of AutomataState.

Instances of the class NamedAutomata are used to represent complete productions that are named. That is, an instance of NamedAutomata is a structured object which contains many individual automata as alternatives. Instances of NamedAutomata contain four additional instance variables: subAutomata, dependents, priority, and selfReferenced. The most important instance variable is subAutomata, which is a collection of all alternative automata making up the named automaton. Note that the inherited instance variable, startState, has as its value the start state of a single automaton, which represents the alternation of all of the sub-automata.

An instance of the class AutomataState represents a single state in a deterministic finite automaton and has two instance variables: transitionsTable and tokenValue. The instance variable, transitionsTable, is a dictionary of transitions leading out of the state. The keys of the dictionary are the labels of the directed arcs in the transition diagram and the values are the states to which these transitions lead.

The second instance variable, tokenValue, contains the value of the token which is returned by the generated scanner when the next input character does not correspond to a legal transition or when there is no more input. If the state is not an accepting state, then the tokenValue is the Smalltalk constant, nil.

A subclass of AutomataState called, LookaheadState, is used to implement the two-character look-ahead. The behavior of this state differs from the class AutomataState in that an input character is examined without being read, and the previously read character is re-inserted into the input stream.

In LexAGen, DFA construction consists of two operations: concatenation and alternation. Concatenation is a straightforward operation which is performed on the components of each single alternative. As an example of concatenation, consider the addition of the alternative, "abc" to a production. A start state and three other states with transitions representing the symbols, a, b, and c, are concatenated to form the topmost automaton shown in figure 6.

Concatenation of a single alternative is invoked by the browser iteratively after checking for input correctness. The browser parses the input expression into a sequence of syntactically correct labels, either terminals (single-character symbols like a and b) or non-terminals (like <letter> and <digit>). If a syntax error is found, then it is reported immediately. For example, the string "ab<C" contains a syntax error. If a string is syntactically correct, then a start state is created as the current state. For each label, a new state is created and the current state is connected to the new state using the label as the transition.

Alternation is the process of combining alternatives, and it is the core of the construction process. The alternation operation merges two DFAs and optimizes the resulting DFA to collapse common states. Usually, one DFA is a production and the other is a new alternative. First, the start states of the two DFAs are merged. Then, the transitions from the start state of the second DFA are added to the start state of the first DFA. If there were transitions whose labels were common to the start states of the two DFAs, then these states must be merged as well and this process continues until the two DFAs have been merged. After merging, the result is optimized.

For example, consider the production A which has been defined as <A> ::= abc. Suppose the user wants to define an alternative for A which is the expression "aad". After the alternative has been converted into a DFA, it is merged with the DFA of the production. A successful merging results in a new start state of the production's DFA, while leaving the start state of the alternative's DFA unchanged. Figure 6 shows the two DFAs, the merged result and the result of optimizing the merged DFA. Since the start states of both DFAs contained a transition on the common terminal "a", the two states with input "a" were merged. Since the labels of the transitions leaving this newly merged state were "a" and "b", the next two states were not merged. Finally, since the merged DFA contained two identical states (two accepting states with token value A and no transitions), the optimizer merged these two states.

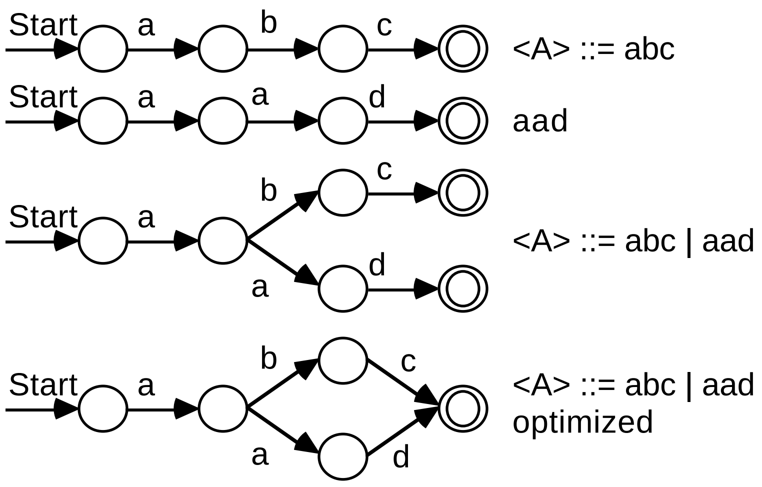


Figure 6 Alternation through merging

When the two DFAs being merged consist only of terminals, the merging operation is quite simple. However if some of the transitions are non-terminals, it is possible that some of the non-terminals may need to be expanded to prevent the merging process from resulting in an NFA. For example, consider the productions <A> ::= ca and <B> ::= cb. If the alternative <A> ::= <B>d is added to production <A> without expanding it, then the NFA shown in figure 7 results.

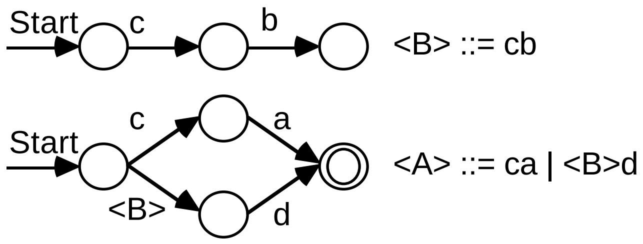


Figure 7 An NFA due to non-expansion of a non-terminal in a merge

This automaton is non-deterministic since the start state of the merged automaton has transitions on c and <B>, but <B> has c as its first transition. However, if the non-terminal, <B>, is expanded during the merge, then the resulting automaton is the DFA shown in figure 8.

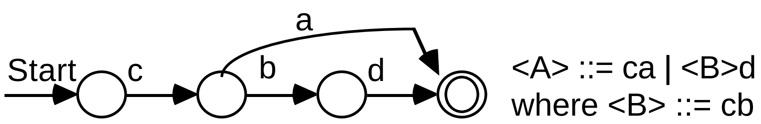


Figure 8 A merge where a non-terminal must be expanded

On the other hand, not all of the non-terminals should be expanded when merging automata, since the expansions reduce the efficiency of editing. For example, consider the production <A> ::= <B>a, where the production <B> is a complex automaton with many alternatives. If <A> was stored in expanded form, then any change to <B> would have to be re-created in <A>. Therefore, in LexAGen, non-terminals are only expanded when necessary.

There is one other consideration when merging two DFAs. If two DFA states represent accepting states but have different token values, they cannot be merged. The interface ensures that different alternatives for the same production share a common token value. That is, if the user wants to add an alternative to a production whose name is A, then the token value of A is automatically used for the accepting state generated by the alternative. However, the scanner itself is represented by a DFA, and can contain accepting states with different token values, like integers and identifiers. When merging two states in the grand automaton, conflicts must be prevented.

For example, consider a production <A> ::= ca with a token value of A and a production <B> ::= cb with a token value of B. If the user tries to add the alternative, <B> ::= ca, then there is no problem in merging the two alternatives for <B>. However, the grand automaton would contain two conflicting states since the string "ca" would be accepted with two token values, A and B. LexAGen detects conflicts during the merging process when the grand automaton is updated as the result of an editing process. If a potential conflict exists, then an error is reported to the user and the editing operation is disallowed.

Based on these considerations, algorithm Merge is used for merging the states of two DFAs. The set FIRST(X) is defined[17] as the set of terminals that begin strings which are derived from X.

Algorithm Merge

```
Given two states, $S_1$ and $S_2$:
    (1) If $S_1$ and $S_2$ have different token values, then a conflict would exist and so an error is
    reported.  Otherwise, apply step (2).
    (2) Add all the transitions of $S_2$ to $S_1$, one at a time, according to:
        Let T be a transition of $S_2$ consisting of a label, L (terminal or non-terminal), and a state, S.
        (a) If $S_1$ contains L, then merge S with the state in $S_1$ connected to L.
        (b) If $S_1$ contains a label, M (terminal or non-terminal), such that
            $\mathrm{FIRST}(M) \cap \mathrm{FIRST}(L) \neq \phi$
        then:
            (i) If M is a dependent of L (i.e.  M is defined in terms of L), then expand M in $S_1$ and
                try to add T to $S_1$ again.
            (ii) Otherwise, create a new state, R (so that $S_2$ is not affected), and add T to R.
            Then expand L in R and merge R with $S_1$.
        (c) Otherwise add T to $S_1$ since $S_1$ does not contain L either explicitly or implicitly in a
        non-terminal.
```

The Strategy for Incremental Analysis

In LexAGen, incremental analysis is the process by which affected DFAs are updated after changes have been made to the specification. Specifically, incremental analysis involves re-analysis of all dependent DFAs when some DFA is changed. For example, suppose that the non-terminal, A, is defined in terms of the non-terminals, B, C, and D, where B and C have been defined in terms of the non-terminal, D. If the user tries to add an alternative to D, all the productions that depend on it directly or indirectly (A, B and C) must be updated. Figure 9 shows the dependency graph for A, B, C, and D. In addition, if at any point an update would result in a conflict in the grand automaton for the scanner, then LexAGen would disallow the addition of the alternative to D and report an error. Cycles will not exist in the dependency graph used for updating DFAs since left-recursion is disallowed in the BNF.

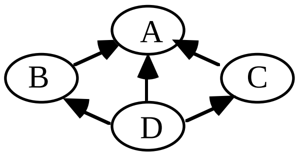


Figure 9 A dependency graph for automata

The implementation of incremental analysis is based on dependencies in Smalltalk-80, where objects can be made dependent of other objects. When an object changes, it sends itself a "changed" message which causes "update" messages to be broadcast to all of its dependents. Since dependents may have dependents, a graph structure of "update" messages exists.

It is possible for an object to receive many "update" messages. For example, consider the situation of figure 9. If node D changes, then it will send "update" messages to A, B, and C. Nodes B and C will subsequently send two more "update" messages to their dependent, A. As a result, A receives three such messages. Such redundant dependencies are common in scanner specifications and can result in slow updating. To solve this problem, LexAGen replaces the standard dependency graph by an ordered dependency graph.

LexAGen uses a two-pass approach to implement the ordering of the dependency graph. Each named DFA maintains an internal priority flag which is initialized to negative one. A changed DFA sets its internal priority flag to zero and then broadcasts a "pre-update" message to all of its dependents. This "pre-update" message contains a priority parameter which is one greater than the internal priority flag of the sending DFA. When a DFA receives a "pre-update" message, it compares its internal priority flag to the priority parameter. If the priority parameter is higher, then it resets its internal priority flag to the value of the priority parameter and broadcasts a "pre-update" message to its dependents with a priority parameter equal to its internal priority flag plus one.

This algorithm orders the dependency graph by assigning to each node, a priority value equal to its level in the graph. For example, in the case of the DFAs shown in Figure 9, the priorities would be: zero for D, one for C, one for B, and two for A. That is, every DFA has a priority flag whose value is greater than the values of the priority flags of all of the DFAs on which it depends.

Once the first pass has been completed, the DFAs are updated in a breadth-first order. That is, the root DFA broadcasts an "update" message which asks its dependents to update themselves from lowest to highest priority count. The root DFA broadcasts an "update" message with priority one. All dependents with internal priority flag values of one, update themselves and broadcast an "update" message with priority two and so on. Notice that each dependent is only updated once and that the update occurs after all of the productions on which it depends.

The Code Generator

The third level of representation is the code generator. When the user requests code for a specification, LexAGen expands the scanner DFA and translates it into intermediate code. The intermediate code is then translated into the target language. For example, figure 10 shows the un-expanded transition diagrams for DFAs A and B, where A is defined as <A> ::= a | a<A> | <B>d and B is defined as <B> ::= b | bc, as well as the expanded form of the DFA for A.

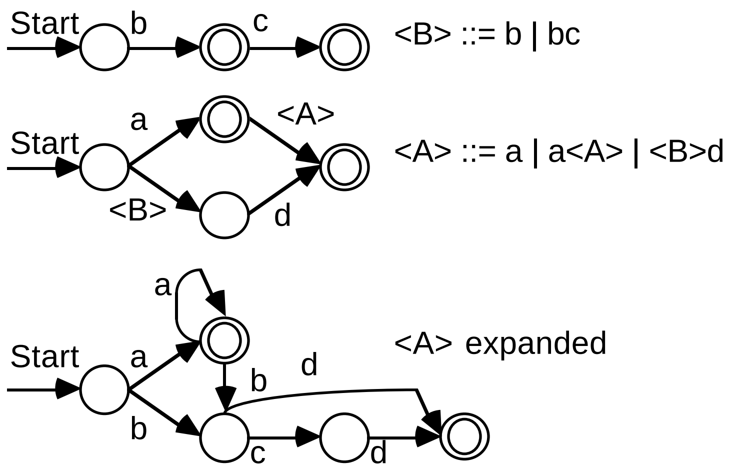


Figure 10 The expansion of a DFA

The code generator is implemented by the class AutomataCode and its nine subclasses: AutomataAdvance, AutomataBreak, AutomataBackup, AutomataCase, AutomataDefault, AutomataGoTo, AutomataLabel, AutomataSwitch and AutomataToken. Instances of the class AutomataCode have one instance variable, operations, which represents a stack of operations to be performed. Each subclass represents a single kind of operation to be translated into the target language. Instances of these subclasses represent the intermediate code.

Currently the target language is C, but other languages can be supported easily. This can be done by changing the behavior of the subclasses which represent the intermediate code, so that they are translated into a different target language. The generated code is composed of nested case statements which resemble the exact structure of the automaton being generated. Figure 11 shows part of the executable C-code program produced by the coder for the automaton of Figure 10.

```
switch (*p) {  /* examine current character */
        case 'a' :
            LABEL0:
            switch (*++p) { /* examine next character */
                case 'a' :
                    goto LABEL0;
                case 'b' :
                    switch (*++p) { /* examine next character*/
                        case 'c' :
                            switch (*++p) { /* examine next character */
                                case 'd' :
                                    ++p; /* advance to next character */
                                    return(A); /* accept token */
                            } --p; break; /* backup by one character */
        {* code removed for brevity *}
        default : ++p; /* advance to next character */
    }
    return(NIL); /* accept no token */
```

Figure 11 Excerpt of generated C-code

In fact, the coder generates a complete module for this program with its own program interface, where the name is given by the user. In addition, the coder produces a file of token values as an include file which may be used by other application programs (for instance, a parser). A library that incorporates a specialized buffering technique for reading input characters and maintains a token buffer to store the token characters is also provided.

KEYWORD IDENTIFICATION

Keyword identification is the process of searching a list of pre-defined keywords to determine if a general identifier is really a language keyword. LexAGen generates a scanner which distinguishes keywords from general identifiers. Given the set of keywords for a language, a data structure called a pruned O-trie forest is constructed. The forest is used to generate code which identifies the keywords for that specific language. LexAGen uses this approach to produce keyword identification code for the last phase of its generated scanners.

The Trie-Based Method

A trie-based method, generally speaking, is an indexing scheme that views an alphabetic key as composed of a sequence of characters. In essence, a trie-based method closely resembles a key-comparison based method or a B-tree search scheme[18]. However, trie-based methods do not rely on the notion of comparing whole keys in constant time. As a consequence, comparison of keys is no longer the elementary operation for standard measures of complexity.

The k-ary tree structure created by successively dividing keys into smaller sets using different attributes is called a trie (pronounced as try)[19, 20]. Given a set of data items, where k is the maximum of the lengths of the data items, a *full trie* is a tree of depth no more than k such that all paths from the root to the leaves are distinct and there is a unique path from the root to each leaf node corresponding to an item in the set.

A full trie for the Ada reserved words, delay, delta, end, and entry is shown in figure 12. A *pruned trie* is a full trie with no redundant non-leaf nodes along any leaf chain. Redundant non-leaf nodes are those consecutive single-successor nodes that lead to a leaf. The left sub-figure in figure 12 is not a pruned trie while the middle sub-figure is pruned.

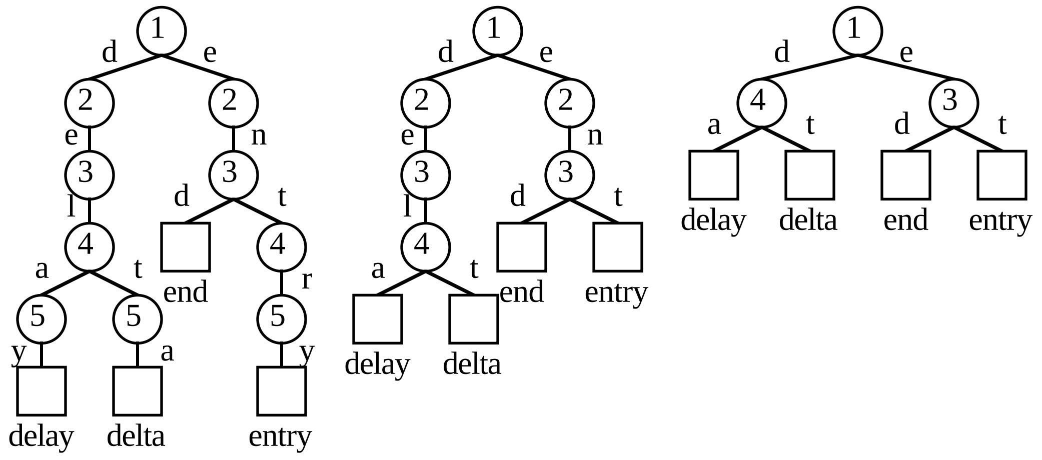


Figure 12 A full trie, a pruned trie and a pruned O-trie

A *pruned O-trie* is a generalization of a pruned trie where different paths from the root may use different attributes. The right sub-figure in figure 12 shows a pruned O-trie in which the path from the root to the leaf labeled "delay" uses character positions 1 and 4, while the path from the root to the leaf labeled "end" uses character positions 1 and 3.

In the LexAGen environment, the length-of-key attribute is used to divide the keys into sets of equal lengths and then character positions are used for the rest of the attributes. To increase efficiency, different paths use different positions. To support this algorithm, a *pruned O-trie forest* is defined as a collection of pruned O-tries with the property that each pruned O-trie represents a subset of equal-length data items.

For example, consider the following reserved words from Ada: delay, delta, entry, if, in, of, raise, range. These keys can be divided into two subsets, consisting of keys of length two and keys of length five. As shown in figure 13, length-two keys using the attributes "character position one" and then "character position two" produce a pruned O-trie. The pruned O-trie composed of length-five keys uses the attributes "character position one" and then either "character position three" or "character position four", depending on the path chosen.

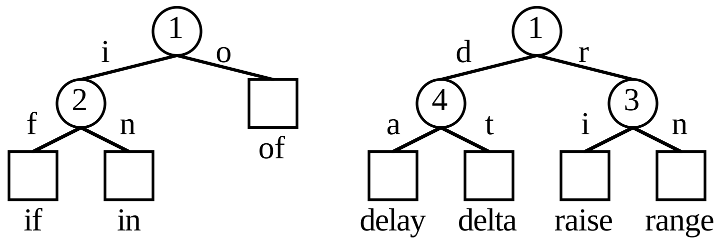


Figure 13 A pruned O-trie forest

Instead of testing all characters or attributes in a string, the search follows along a pruned chain that leads to a leaf node[21]. An additional string comparison is then done to verify that the string matches the contents of the leaf node. Notice that in the example of figure 13 a tree of smaller depth (depth one) could have been constructed for the keywords of length five if "character position four" had been used instead of "character position one". The heuristic algorithm presented in this paper finds a tree of *minimal depth* and this algorithm is incorporated into LexAGen.

Looking for an identifier in a pruned O-trie forest consists of identifying the pruned O-trie corresponding to the length of the identifier and searching the trie on a path from the root to a leaf. If it is assumed that the correct pruned O-trie can be located in the forest in constant time (say, using a jump table implementation of a case), then it is straightforward to show that the worst-case searching time for a pruned O-trie forest is of the order of the maximum length of the keywords(see reference 5).

Trie Index Construction

A branch-and-bound algorithm has been devised to construct a pruned O-trie forest for an arbitrary collection of keywords. The algorithm is used as the last phase of lexical analysis. Since the trie-index construction problem is NP-complete (see reference 23), the algorithm may take exponential time to return a minimal-depth pruned O-trie forest. If it is assumed that the probabilities of all keywords are equal, then the probabilities of all leaf nodes in the forest are equal so that this algorithm yields an optimal forest which generates the most efficient search. The cost of a pruned O-trie is defined as the sum of the depths of each of its leaves.

Algorithm O-Trie Construction

(1) Split the set of keywords into subsets of equal-length data items.

(2) For each of these subsets, apply step (3) with depth equal to 1.

(3) Given a set of n, m-length data items at depth d, create a decision table with m entries corresponding to the characters in positions 1 to m. The decision-table entry at location i is a dictionary whose keys are all of the ith characters from all of the n data items. The value of each key at location i is the subset of data items which contain the key character at location i. If the

number of entries in any one dictionary is equal to n, then return to the calling step with a three-component record containing: the value of i, a cost which is the product of n and d, and the ith dictionary. Otherwise, go on to steps (4) and (5), and return the results to the calling step.

(4) From the decision table, create two control stacks, A and E, which store records of the form defined in (3), where the cost is either an actual minimum cost or an expected minimum cost. Both stacks are sorted by cost in ascending order (i.e. the top-of-stack record corresponds to the pruned O-trie with the least cost value among all the tries in the stack). The cost of the ith dictionary is defined to be the sum of the individual costs associated with the data items in its value sets. If the size of a value set is equal to one, then its cost is equal to the depth, d; otherwise, its cost is equal to the product of its size and (d + 1). If the size of any value sets is greater than two, then the cost is an expected minimum cost; otherwise, the cost is an actual minimum cost. Records which contain actual costs are pushed onto the A stack, and records which contain expected costs are pushed onto the E stack. As the algorithm proceeds and expected costs are refined into actual costs, the E stack will shrink and the A stack will grow. Go to step (5).

(5) (a) If (Stack E is empty) or (Stack A is non-empty and the top-of-stack record in Stack A has a cost which is less than the cost in the top-of-stack record in Stack E), then return the top-of-stack record from A to step (3).

(b) Otherwise, pop Stack E, and replace its cost and dictionary components using (6). Push the resulting record onto the A stack since the cost will now be actual. Repeat step (5).

(6) Reset the cost component of the current record to zero. Apply step (3) to each value set of the current record whose cardinality is greater than one using a depth of d + 1. Remove the cost component of the record returned by step (3) and add it to the cost component of the current record. Replace all value sets of size greater than one in the current record by the record returned by step (3) with the cost component removed. However, add d to the cost component of the current record for each value set of size one. Return the modified record to step (5).

Example O-trie Construction

Consider a language, H which contains the keywords: {and, end, mod, eot, any}.

Steps (1) and (2) identify the set { and, end, mod, eot, any } and assign (d = 1).

Step (3) produces a decision table with (n = 5, m = 3, d = 1):
<i=1, [(a, {and, any}), (e, {end, eot}), (m, {mod})]>
<i=2, [(n, {and, any, end}), (o, {eot, mod})]>
<i=3, [(d, {and, end, mod}), (t, {eot}), (y, {any})]>
Since none of the dictionaries contains a five-element value set, go on to step (4).

Step (4) creates two control stacks. The A stack contains one record with contents:
<i=1, C=2*2+2*2+1=9, [(a, {and, any}), (e, {end, eot}), (m, {mod})]>
The E stack contains two records with contents:
<i=3, C=3*2+1+1=8, [(d, {and, end, mod}), (t, {eot}), (y, {any})]>
<i=2, C=3*2+2*2=10, [(n, {and, any, end}), (o, {eot, mod})]>
Go on to step (5).

In step (5), condition (a) does not apply since the top-of-stack record for Stack A has cost 9 and the top-of-stack record for Stack E has cost 8; so use condition (b) and call step (6).

In step (6), reset the cost component of the current record to zero:
<i=3, C=0, [(d, {and, end, mod}), (t, {eot}), (y, {any})]>
The current record contains one value set of size greater than one: {and, end, mod}. So, apply step (3) to this set with a depth of 1 + 1 = 2. The record for this set returned by step (3) is:
<i=1, C=3*2=6, [(a, {and}), (e, {end}), (m, {mod})]>
Remove the cost component from the above record and add to the cost component of the current record; also, replace the value set by the record returned by step (3) with the cost component removed:
<i=3, C=6, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y, {any})]>.
The current record contains two value sets of size one: {eot} and {any}. So, add 2*1 = 2 to the cost component of the current record and return the modified record to step (5)

Returning to step (5) we have:
<i=3, C=8, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y, {any})]>.
Therefore, Stack A contains the records:
<i=3, C=8, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y, {any})]>.
<i=1, C=9, [(a, {and, any}), (e, {end, eot}), (m, {mod})]>
The E stack contains one record with contents:
<i=2, C=10, [(n, {and, any, end}), (o, {eot, mod})]>
Now, condition (a) applies since the top-of-stack record for Stack A has cost 8 and the top-of-stack record for Stack E has cost 10; so return the top-of-stack record from A to step (3).

Since the record:
<i=3, C=8, [(d, <i=1, [(a, {and}), (e, {end}), (m, {mod})]>), (e, {eot}), (y, {any})]>,
was returned to step (3), this record is also returned as the result of step (3).

That is, the O-trie for length three keywords is created by first considering the third character to divide the keywords into three sets: {and, end, mod}, {eot} and {any} and the first set is further subdivided by considering the first character as shown in figure 14.

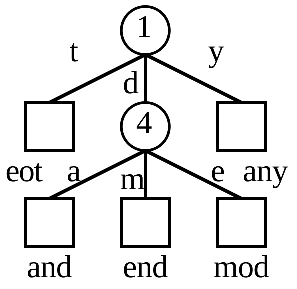


Figure 14 An O-trie produced by Algorithm O-trie Construct

The LexAGen environment produces C code which implements the optimal pruned O-trie forest produced by the algorithm. Each non-leaf node is represented by a case statement of character labels, where the labels are those characters which identify the branches in the trie. Leaf nodes are represented by case statements which correspond to those character labels not previously appearing in the non-leaf nodes along any leaf chain.

In theory, the access time of the minimal-depth pruned O-trie forest is independent of the number of keywords and depends linearly on the maximum length of the keywords. In practice, the average depth of a minimal-depth pruned O-trie forest is considerably smaller than the maximum length of the keywords, especially given a large length, and the depth is usually one or two as shown in figure 15.

	Pascal		Modula-2		Ada	
length	size	depth	size	depth	size	depth
2	6	2	7	2	7	2
3	9	2	8	1	12	2
4	7	2	8	1	12	2
5	6	1	6	1	10	2
6	4	1	6	2	8	2
7	2	1	1	0	8	2
8	1	0	-	-	3	1
9	1	0	2	1	3	1
10, 11, 12, 14	-	-	1	0	-	-

Figure 15 Keyword O-trie characteristics for common languages

AN ASSESMENT OF LEXAGEN

For the sake of efficiency, lexical analyzers for production compilers are often hand-coded. However, lexical-analyzer generators can produce efficient scanners (see references 9 and 17). In addition, generated scanners have certain advantages. Generated scanners can be produced quickly and easily. For example, although scanner generators utilize the best pattern-matching algorithms, the individual who uses one, needs to know nothing about pattern matching. Automatically generated scanners also have fewer programming bugs. Furthermore, the lexical description used to produce a generated scanner is not only a specification, but also serves as valuable documentation for the generated lexical analyzer.

The most important criteria in evaluating scanners is usually speed of execution and to some extent code size. However, two other criteria should also be considered: the generality of scanners which can be produced and the user interface of the generator. Of course, the relative weights which are placed on these criteria must depend on the user and the application. The generation of a scanner for a prototype language which is under development is a different process than the generation of a production scanner for a stable language. Generality and ease of use are far more important in the first case than in the second. Unfortunately, these three criteria are usually in competition. That is, speed is gained by restricting generality and ease of use or generality is maintained at the expense of speed.

In this section LexAGen is assessed by comparing it to three other scanner generators: Lex (see reference 1), GLA (see reference 3) and Mkscan (see reference 4). Although many other scanner generators are in use today, these three generators are a representative cross-section. They are compared on the basis of: execution speed (and code size), generality and ease of use. Figure

16 contains a summary of the results of this section, ordered by speed which is usually the most important criteria.

Name	Speed	Generality	Ease of Use
GLA	++	-	-
LexAGen	+	+	++
Mkscan	+	-	+
Lex	-	++	-

Figure 16 A summary of four scanner generators

Scanner Execution Speed

As far as relative sizes and speeds of generated scanners are concerned, LexAGen has achieved its goal to produce general and efficient scanners. In an experiment, four scanners for Pascal were created using LexAGen, GLA, Mkscan, and Lex. The four scanners were compiled on a SUN 2/50 under the 4.2BSD UNIX system. When compiled, the object code sizes for the four scanners were 9.07K, 24.98K, 4.68K, and 10.74K bytes, respectively.

The large size of the GLA-generated scanner is attributed to the existence of the supporting modules. It should also be noted that both LexAGen and Lex include the keyword identification algorithm mentioned in section 6 as part of the generated scanners, and this accounts for the extra size when compared to the Mkscan scanner which uses a hashing technique to distinguish keywords from identifiers. The Lex scanner was constructed in such a way that all keywords are recognized as identifiers and handled separately by the module for keyword identification.

Scanner timings were obtained by tokenizing one large Standard Pascal program (see reference 5 for the characteristics of the input data). All tests were made under the 4.2BSD UNIX system on the same machine where the scanners were compiled. Each test was run 10 times, and the mean of these samples was used. The time was the sum of the total amount of time spent executing in user mode and system mode (executing system calls for the scanners).

GLA generated the fastest scanner. The ratios of the speeds of the other generated scanners to the speed of the GLA scanner are: 1.04 (LexAGen), 1.10 (Mkscan), and 2.69 (Lex). Of course different results occur if different input characteristics are used, but these results are representative.

Generality

Superficially, the generality of LexAGen and Lex are identical in that they both support the full set of regular languages. However, their support for context sensitive language features differ slightly. Lex has multiple character look-ahead while LexAGen has only two character look-ahead. On the other hand, LexAGen directly supports nested comments while Lex does not.

The generality difference between LexAGen and Lex are insignificant compared to the generality differences between these two generators and the other two. Neither GLA nor Mkscan support the full set of regular languages.

GLA has obtained its increase speed by imposing constraints on the allowable symbol sets for its specifications. Unfortunately, GLA is too restrictive to specify several of the tokens in standard programming languages. For example, consider floating numbers. GLA allows the user some freedom in specifying a floating point number (the initial character set, the continuation character set, and so on). Surprisingly however, GLA does not provide enough freedom to correctly specify floating point numbers in Pascal or Modula-2. GLA requires that the decimal point for floating numbers be preceded by an initial or continuation character and that it be followed by a continuation character.

For example, GLA does not allow the user to specify that "12e5" is a legal Pascal floating point number. In addition, GLA requires that "12.12e" is a legal floating point number in all languages, if the character 'e' has been designated as the exponent character. But this is not a legal floating point number in Pascal or Modula-2. Finally, GLA is incapable of specifying that "12." is a legal floating point number, even though this is the case in Modula-2.

As a second example, GLA does not allow C-style hexadecimal constants like "0x123" to be specified. As a third example, GLA does not support the nested comments in Modula-2. Even though GLA could be modified to support all of the tokens in all existing programming languages, there is nothing to prevent future languages from using reasonable constructs which GLA could not support. The problem is that a scanner generator needs a mechanism for specifying general tokens.

Mkscan guides the user through a series of choices in an interactive manner. However, this interactive style was developed by sacrificing generality. In particular, Mkscan classifies tokens according to their common use in programming languages and groups them into four categories: identifiers, keywords, constants, and special symbols.

Mkscan was designed to produce scanners for programming languages and seems especially biased towards the family of Pascal-like languages. However, nested comments are not supported, so a correct Modula-2 scanner cannot be generated.

Ease of Use

There are two important factors regarding ease of use: the specification language and the interaction style. In all four scanners the specification language is quite complex since the specification of a scanner is a non-trivial exercise. Therefore, the differences are slight and there are some problems with each.

In Lex, users who want to specify only simple patterns have to learn the full, general expression notation used by Lex. Similarly the generality of LexAGen requires that the users be familiar with BNF grammar notation.

GLA has the extra requirement that along with the specification language, users must also learn the functions of a collection of supporting modules which are incorporated with the generated scanner. The symbol table module is an example of such a supporting module. This is satisfactory for users who are generating scanners which are to be used with parsers generated by traditional parser generators. However, it may be too complicated for users who are generating stand-alone scanners or scanners to be used with non-traditional parsers (say, incremental parsers) where the supporting modules may be redundant or simply not required.

Mkscan advocates that the notion of a regular expression is just one particular way of conceptualizing tokens and therefore is more than a part of the implementation than of the specification. However, Mkscan employs certain metacharacters and a pattern notation that is equivalent to regular expressions.

Although the complexity of the specification languages are comparable, the interaction styles of the four generators vary widely. Lex and GLA have virtually no user interaction, while LexAGen and Mkscan are very interactive. There are four major interaction points: error reporting, changes to the specification, specification testing and scanner generation.

For Lex and GLA, the user must supply the entire scanner specification before the generator is invoked to look for errors. On the other hand, Mkscan has limited interactive error reporting and LexAGen has completely interactive error reporting.

For Lex and GLA, any changes to the specification must be made using a text editor and the entire scanner must be regenerated. However, Mkscan makes it relatively easy to modify previously generated scanners. It does this by storing information about the scanner as comments at the beginning of the file which contains the scanner. If a user wishes to modify a scanner, the scanner's file is read by Mkscan and the comments are used as the specification. Notice that when a scanner is generated, the user gets the specification along with the scanner. In LexAGen, changes are recorded incrementally and are immediately visible to the user.

As far as specification testing, Lex, GLA and Mkscan require the user to generate a scanner and write a driver program to test the specification. LexAGen is unique in allowing the user to test the scanner before it is generated. Like any programming environment, this reduces the context switches necessary to complete the task. That is, editing, compiling and testing are done in a common environment.

CONCLUSION

LexAGen is the result of an attempt to provide a scanner generator which is easy to use and which generates fast scanners for general specifications. The generality of LexAGen comes from its ability to support the full set of regular languages plus some general extensions which are sufficient for most modern programming constructs. LexAGen is also unique among scanner generators in applying many benefits of graphical user interfaces to scanner generation. For instance, LexAGen is the first (and only) scanner generator that incorporates incremental development. Furthermore, LexAGen provides full-scale user interaction ranging from immediate error reporting to specification execution.

ACKNOWLEDGEMENT

This research was supported in part by research grant #OGP8191 from the National Sciences and Engineering Research Council of Canada.

REFERENCES

1. M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator," *Computing Science Technical Report 39*, Bell Telephone Laboratories, Murray Hill, NJ, 1975.
2. S. C. Johnson, "Yacc - Yet Another Compiler-Compiler," *Computing Science Technical Report 32,* Bell Telephone Laboratories, Murray Hill, NJ (1975).
3. W. M. Waite, V.P. Heuring, and R.W. Gray, "GLA - A Generator for Lexical Analyzers," *Software Engineering Group Report No. 86-1-1*, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO (1986).
4. R. N. Horspool and M.R. Levy, "Mkscan - An Interactive Scanner Generator," *Software - Practice & Experience,* **17**(6), 369-378 (1987).
5. R. W. G. Ng, "LexAGen - A Lexical-Analyzer Generator", Master of Science Thesis, University of Alberta, Canada (1988).
6. V. P. Heuring, "Automatic Generation of Fast Lexical Analyzers," *Software - Practice & Experience,* **16**(9), 801-808 (1986).
7. Waite, W.M., "The Cost of Lexical Analysis," *Software - Practice & Experience*, **16**(5), 473-488 (1986).
8. G. E. Révész,, *Introduction to Formal Languages*, McGraw-Hill, NY (1983).
9. H. Mössenböck, "Alex - A Simple and Efficient Scanner Generator," *ACM SIGPLAN Notices*, **21**(5), 69-78 (1986).
10. D. Szafron and R. Ng, "LexAGen Users Manual," *Technical Report TR 89-8,* Department of Computing Science, University of Alberta, Edmonton, AB (1989).

11. N.M. Delisle and D.E. Menicosy, and M.D. Schwartz, "Viewing a Programming Environment as a Single Tool," *ACM SIGPLAN Notices*, **19**(5), 49-56 (1984).
12. A. Golderg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA (1985).
13. A. Golderg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA (1984).
14. D. Szafron and B. Wilkerson, "Some Effects of Graphical User Interfaces on Programming Environments," *Proceedings of the CIPS/ACI Congress '86*, (1986).
15. D. Szafron and B. Wilkerson, "The Smalltalk-80 MVC Paradigm with Plugable Views," *Technical Report TR 88-8, D*epartment of Computing Science, University of Alberta, Edmonton, AB (1988).
16. P. Wegner, "Dimensions of Object-Based Language Design," *OOPSLA '87 Proceeding* 168-182 (October 1987).
17. A. V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).
18. R. Bayer and E. McCreight, "Organisation and Maintenance of Large Ordered Indexes,"*Acta Informatica*, **1**, 173-189 (1972).
19. R. de la Briandais, "File Searching Using Variable Length Keys," *Proc. WJCC*, 295-298 (1959).
20. Fredkin, E., "Trie Memory," *Communications of the ACM*, **3**(9), 490-499 (1960).
21. D. Comer and R. Sethi, "Complexity of Trie Index Construction (extended abstract)," *Proc. 17th Ann. Symp. on Foundations of Computer Science*, IEEE Computer Society, Long Beach, CA, 197-207 (1976).