A Language Independent Scanner Generator

Teodor Rus and *Tom Halverson*
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242
(319) 335-0742, rus@herky.cs.uiowa.edu

Abstract:

This paper discusses a methodology for scanner generation that supports automatic generation of *off the shelf* scanners from specifications. The lexicon specification rules are layered on two levels: rules specifying "universal" lexical constructs which are used as building blocks by most programming languages and rules specifying customized lexical constructs of a specific programming language, using universal lexical constructs as building blocks. The universal lexicon is specified by regular expressions over a global alphabet used by most programming languages, such as the character set of a keyboard, and is efficiently implemented by deterministic finite automata. The customized lexicon is conveniently specified by regular expressions of *properties of universal lexical constructs* and is implemented by nondeterministic automata whose transition function is determined by the truth value of properties of universal lexemes. Tools that transform a lexicon specification into a stand alone scanner are developed and reported. This methodology is convenient because the user operates with meaningful constructs and no programming is required. The lexical analyzers thus specified are efficient and their correctness is mathematically guaranteed. Examples and experiments that demonstrate the implementation of stand alone scanners can be found at the URL http://www.cs.uiowa/~rus/ in the package "component-based software development tools".

Keywords: finite automata, lexicon specification, scanner generator.

Contents

1 Introduction

For a given regular expression e, an algorithm that inputs a string x and determines whether or not x is an element of the language specified by e is called *scanner(e)*. That is, *scanner(e)* answers the question *is* $x \in L(e)$? A lexical definition is an ordered set of equalities of the form

$$\begin{aligned} N_1 &= e_1 \\ N_2 &= e_2 \\ \ldots & \\ N_m &= e_m \end{aligned}$$

where N_i, $1 \leq i \leq m$, are unique names called *tokens*, and e_i are regular expressions constructed over the alphabet $\Sigma(e_i)$, $1 \leq i \leq m$. For a given lexical definition, d, there is an algorithm called a *lexical analyzer*, that inputs a string x and tokenizes it. The tokenizing process consists of recognizing each substring of x that is a lexical element specified by the right hand side, rhs, of some equation from d and replacing it by the left hand side, lhs, of that equation. The character sequence forming that token is referred to as the token's lexeme. Further, the terms scanner and lexical analyzer will be used interchangeably.

The conventional approach to describing the lexical constructs of a programming language relies heavily on the use of regular expressions and lexical definition [1, 5, 13]. Therefore, it seems natural to design generators of lexical analyzers that operate on regular expressions. But the lexicon of concrete programming languages sometimes contains constructs that are not regular. Some lexical analyzer generators, such as Lex [10] and Rex [4], treated these non-regular constructs as "exceptions" to be handled outside of the syntax of the lexicon specification language, usually by requiring the user to write code in a standard programming language. Other lexical analyzer generators, such as Alex [11], GLA [15], and Mkscan [6], have made a move toward incorporating the non-regular features into the syntax of the lexicon specification. Still others, have moved the entire lexical analysis process into the parser, thus specifying them with the same BNF rules as used for the rest of the language syntax [14]. Such scannerless parsing techniques do provide some benefits by eliminating the scanner/parser interface, however the grammars soon become unwieldy. Thus, a separate scanning tool is still important. The various approaches used in the above lexical analyzer generators did not lead to a solution to the problem of developing a universal lexical analyzer that could be easily and conveniently integrated in a compiler while at the same time being used as a stand alone language processing component performing lexical analysis [8].

In contrast to previous generators of lexical analyzers, the lexicon specification discussed in this paper is written in terms of primitive lexical constructs rather than using a given character set taken as the language alphabet. These primitive lexical constructs, such as *identifier* and *number*, are found in all programming languages and are generated by regular expressions defined over the language alphabet. In addition, these lexical constructs have structural properties such as the regular expression defining them, the lexemes representing

them, the length of their lexemes, etc., that can be used to develop a two level lexicon specification that can be recognized by a two level scanning algorithm. Similar work was reported about a scanning algorithms called the TwoLev [8]. A two-level scanner operates as a scanner-within-a-scanner. The inner scanner, called the First Level Scanner (FLS), is efficiently generated from a given set of regular expressions. The unique feature of these regular expressions is that they do not depend on the programming language that is being implemented, yet the tokens that are returned by the FLS have a clear meaning in virtually every programming language. The Second Level Scanner (SLS) works exclusively with the lexemes recognized by the FLS, thus allowing us to extend the notion of a regular language as a lexicon specifier to that of a regular language of properties of the primitive lexical entities. An expression of such properties will be referred to as a condition.

In this framework, a user developing a lexical analyzer can think in terms of higher level constructs which have well-defined semantics rather than thinking about character sets. The specification of a lexical analyzer then consists of a set of easy-to-construct equations in the extended regular language of conditions. Formally, a lexical definition is a two level construct of the form

$$\begin{aligned} N_1 &= e_1 \\ N_2 &= e_2 \\ &\dots \\ N_m &= e_m \\ T_1 &= P_1(L(e_1), \dots, L(e_m)) \\ T_2 &= P_2(L(e_1), \dots, L(e_m)) \\ &\dots \\ T_n &= P_n(L(e_1), \dots, L(e_m)) \end{aligned}$$

where e_i, $1 \leq i \leq m$, are regular expressions over the language alphabet, $\Sigma(e_i)$, specifying lexemes universally valid in all programming languages and N_i are token names of the languages $L(e_i)$ specified by e_i, $1 \leq i \leq m$. $P_k(L(e_1), \dots, L(e_m))$, $1 \leq k \leq n$, are regular expressions over a finite alphabet of conditions using properties of elements of the languages $L(e_1)$, $L(e_2)$, ..., $L(e_m)$. Our assumption is that e_1, e_2, ..., e_m can be efficiently implemented by a first level scanner, FLS. Hence, replacing the languages $L(e_i)$ by their token names N_i, $1 \leq i \leq m$, and having in view that $L(e_i)$, $1 \leq i \leq m$, are fixed, the lexicon specification becomes:

$$\begin{aligned} T_1 &= P_1(N_1, \dots, N_m) \\ T_2 &= P_2(N_1, \dots, N_m) \\ &\dots \\ T_n &= P_n(N_1, \dots, N_m) \end{aligned}$$

That is, rather than layering the lexicon on syntactic levels as done by conventional approach we use semantic properties of universal constructs to produce a layering of the lexicon.

Regular expressions of properties provide an increased expressive power when compared with conventional regular expressions. This leads to the development of language independent generators of lexical analyzers that produce stand alone lexicon processing tools to be used as scanners in conventional compilers, as the first step of the transformations performed by an algebraic compiler, as lexical analyzers of natural language processing tools, or in any other application that could use a scanner.

The paper is structured as follows: Section 2 discusses the lexicon specification by regular expressions of conditions; Section 3 presents our technology of mapping lexicon specifications into stand alone lexical analyzers; Section 4 discusses the methodology for integrating the scanning algorithm with other tools thus generating stand alone applications; Section 5 presents preliminary conclusions on using this methodology in various applications such as parser generation in algebraic compilers and in teaching compiler construction.

2 Lexicon specification by lexical equations

A *lexical entity* in a programming language is the lowest level class of constructs in the alphabet of the language that has meaning within the language. For example, individual letters or digits generally do not have a specified meaning, but the class of identifiers do. Language processing tools consider lexical entities to be the smallest constructs of interest. A lexical entity can represent either an individual construct, as does the operator :=, or a collection of constructs, as does the class of identifiers. Also, most lexical entities can be described by regular expressions, but concrete programming languages may contain lexical entities that are not regular in nature, such as recursive comments. Therefore, the lexicon specification language must be more powerful than a conventional regular language to allow it to specify all of the lexical constructs. On the other hand, there is a well-established methodology for generating scanners from regular expressions [13]. Hence, our approach is to use regular expressions over finite sets of properties called conditions rather than over conventional alphabets. This allows us to increase the expressive power of regular expressions while preserving much of the well-known methodology for scanner generation from regular expressions.

We assume that each lexical entity is specified by a *lexicon specification rule* of the form $LHS = RHS$ where LHS is the token name and RHS is a pattern that specifies the lexemes that will be tokenized to LHS. For a given programming language, PL, the lexicon specification rules are collected in the *Lexicon Specification File* (*PL.LSF*). See PartF77.LSF and PartC.LSF in Appendix B and C respectively.

The language specified by these lexical equations is often treated as the first level of valid language constructs of a high-level language. This is achieved by allowing the left-hand sides of such lexical equations to be used as terminals in the BNF rules that specify the syntactic constructs of the high-level language.

2.1 First level lexicon specification

The primitive lexical entities considered in this paper are constructed from a "universal" character set. Lexical items constructed from characters in this set may have a universal meaning such as number or word. The common property of all these classes of universal lexical entities is that elements of a class can be distinguished from the elements of other classes both syntactically and semantically by their syntactic form. In addition, the elements within a class can be distinguished by examining properties of their components. The lexical entities chosen as building blocks for the specification of a language lexicon are called *universal lexemes* and form the *first level lexicon*.

The lexemes that seem to be universally used by all programming languages are: letter sequences (identifiers), digit sequences (numbers), white spaces, unprintable characters, and other characters (punctuation, operators, separators, etc.). These universal lexical entities can be specified by the following regular expressions over the character set:

I: identifiers, defined by the regular expression $L(I) = Letter\ Letter^*$,
N: numbers, defined by the expression $L(N) = Digit\ Digit^*$,
W: white spaces and tabs,
U: unprintable characters (such as newline),
O: other characters (punctuation, separators, etc.)

These classes of constructs are universal across nearly all programming languages in the sense that they are used as building blocks of the actual lexical constructs found in the programming language.

In order to use these lexical items as fundamental entities in the construction of the lexicon of a programming language, we characterize them by the attributes *Token* which designates the class, i.e., $Token \in \{I, N, W, U, O\}$, *Lex*, which is the actual string of characters identifying the entity of a class, and *Len*, which is the number of characters making up a lexeme. Other attributes can be easily added and are obviously necessary, but will not be described here. The symbols *Token, Lex, Len* are further used as metavariables whose values are the respective properties of the universal lexemes.

The FLS is a deterministic automaton which uses as input a stream of characters in the alphabet of the programming language, and groups them into the above five classes that may be found in all programming languages. At each call, the FLS returns one tuple of attributes $\langle Token, Lex, Len\rangle$ called a *universal lexeme*.

Example 1: If a source text contained the string *var_3*, the sequence of universal lexemes returned by repeated calls to the FLS would be: $\langle I,$ "*var*", $3\rangle$, $\langle O,$ "_", $1\rangle$, and $\langle N,$ "3", $1\rangle$.

2.2 Second level lexicon specification

The second level lexicon is specified by regular expressions of conditions over first level lexical entities, which show how to combine language independent first level lexemes to obtain valid

lexical constructs of a particular language.

2.2.1 Conditions

Conditions are properties of the universal lexemes expressible in terms of the fundamental attributes *Token, Lex, Len* that characterize them. To formally define the concept of a condition we observe that these attributes refer to three fundamental types of data: set, string, and integer. Hence, we need to define set operations, string operations, and integer operations on the universal lexeme. These operations, in turn, will allow us to construct the lexicon of actual programming languages as regular expressions over expressions representing properties of universal lexemes. To express such properties, we will develop a language of fundamental properties of universal lexemes where data are tuples $\langle Token, Lex, Len\rangle$ and operations include relations on sets, strings, and integers, and logical operators, such as $\wedge$, $\vee$, and $\neg$.

The $Token$ attribute of an universal lexeme is expressed by set membership, $Token \in \{I, N, W, U, O\}$. Therefore, the set membership predicate must be supported as a fundamental operation. However, since the number of token types is finite the membership predicate can be expressed by the equality and logical-or operators, that is $(Token = I) \vee (Token = N) \vee (Token = W) \vee (Token = U) \vee (Token = O)$. The Lex attribute of a universal lexeme is a string. Hence, operations on strings must be supported. We allow fundamental operations of the form $Lex\ rel\ string$ where $rel \in \{<, \leq, =, >, \geq\}$ and $string$ is a constant of type string. However the interpretation of the relations $<$, $\leq$, $=$, $>$, and $\geq$, depends upon the $Token$ attribute. That is, if $Token = N$ then these are relations with numbers and if $Token \neq N$ then these are lexicographic relations. The Len attribute of a lexeme is an integer. Therefore, usual integer relations $<$, $\leq$, $=$, $>$, and $\geq$, also need to be supported.

In addition to the operations specified above, one also needs operations on the component characters of a universal lexeme. This is because the definition of regular expressions accept characters of the language alphabet as lexical entities and the character sets that are used to compose lexemes of concrete programming languages are not necessarily disjoint. For example, A, B, C, D, E, F are both letters that can be used to construct identifiers and also digits of a hexadecimal number. Hence, to allow an easy specification of lexical entities whose components expand over different character sets, we provide character level operations on the Lex component of a universal lexeme. These operations are:

1. $LexChar(i)$ is the i^{th} character of the lexeme Lex.
2. $LexChar(i)\ rel\ C$ where $rel \in \{<, =, >, \leq, \geq\}$ checks if the alphabetic relation rel holds between the i^{th} character of Lex and the constant character C.
3. $LexChar(i)\ in\ \{C_1, C_2, \ldots, C_n\}$ checks if the i^{th} character of the lexeme Lex is in the set of characters $\{C_1, C_2, \ldots, C_n\}$. If this set is ordered and continuous over the character range then this operations can be expressed by $LexChar(i)\ in\ [C_1..C_n]$.

4. $LexChar(i, j)$ *in* $\{C_1, C_2, \ldots, C_n\}$ checks if the characters in positions i through j of the lexeme Lex are in the set $\{C_1, C_2, \ldots, C_n\}$. Again, if the set $\{C_1, C_2, \ldots, C_n\}$ is ordered then this can also be expressed by $Lex(i, j)$ *in* $[C_1..C_n]$.

Now, conditions are recursively defined by the following rules:

1. A condition is a property of the attributes of a universal lexeme expressible in the language of fundamental properties defined above. For example, $Token = I$, $Len <= 8$, $Lex =$ "*do*", *LexChar(1) = 'd'*, and $LexChar(1, Len)$ *in* $['A'..'F']$ are conditions.

2. A condition is a logical expression on conditions constructed with the operators *or*, *and* and *not*; for example, $Token =$ *I and* ($Len > 3$ *or* $Lex =$ "*aa*"), is a condition.

Notice: Since the meaning of fields and operations varies by token, we require that the first relation occurring in a condition specifies the $Token$ attribute. We call this relation the *class specifier*. All following relations in that condition (each of which may then specify either the Lex, Len, or other attributes), will refer to the same universal lexeme as the class specifier. With this in mind, the term condition will be used to refer to an expression that includes the class specifier and the term conditional property or property will refer the subexpressions of a condition that does not include the class specifier. Thus, the condition $Token =$ *I and* ($Len > 3$ *or* $Lex =$ "*aa*") specifies an identifier which has the property that its length is greater than three or its lexeme is the string "aa".

2.2.2 Regular expressions of conditions

Consider the alphabet $\mathbf{A} = \{I, N, W, U, O, \epsilon\}$ and the family of languages

$$L(A) = \{L(I), L(N), L(W), L(U), L(O), \epsilon\}$$

where ϵ is the symbol that denotes the empty string. The regular expressions of conditions and the language specified by them are constructed from conditions by the usual rules:

1. ϵ is a regular expression of conditions and the language specified by it is $\emptyset$.

2. Any valid condition c whose token $T_c \in \{I, N, W, U, O\}$ is a regular expression of conditions and the language specified by it is $L(c) = \{x \in L(T_c) | c(x) = true\}$.

3. If e_1 and e_2 are regular expressions of conditions then $e_1|e_2$ (where $|$ denotes the choice operation) is a regular expression of conditions and the language specified by $e_1|e_2$ is $L(e_1) \cup L(e_2)$.

4. If e_1 and e_2 are regular expressions of conditions then $e_1 \circ e_2$ (where $\circ$ denotes the operation of concatenation) is a regular expression of conditions and the language specified by $e_1 \circ e_2$ is $L(e_1)L(e_2)$. From now on we will denote the operation of concatenation by juxtaposition.

5. If e is a regular expression of conditions then $(e)^*$ (where $*$ denotes the Kleene star) is a regular expression of conditions and the language specified by $(e)^*$ is $\cup_{i=0}^{\infty} L(e)^i$.

For a given finite set C of conditions, let $Reg(C)$ denote the collection of regular expressions of conditions defined by rules (1) through (5) above. The language specified by a regular expression of conditions $e \in Reg(C)$ can be recognized by a nondeterministic finite automaton $NFA_C = \langle Q, C, \delta, q_0, F\rangle$ whose alphabet C is the set of conditions used to construct e extended with the ϵ symbol. NFA_C performs as follows:

1. For a condition $c \in C$ and a state $q \in Q$ the state transition function $\delta(q, c) = R$ reads: *"in state q, evaluate condition c and if true, then goto state $R \in Q$"*.

2. A string $c \in C^*$, is accepted by NFA_C if $\delta(q_0, c) \in F$ where for every $c_1 \in C$ and $c_2 \in C^*$, $\delta(q, c_1c_2) = \delta(\delta(q, c_1), c_2)$.

If $c_1c_1 \ldots c_n \in C^*$, $1 \leq i \leq n$, is accepted by NFA_C then for all $x_i \in L(Token(c_i))$, $x_i \in \Sigma^*$, $x_1x_2 \ldots x_n$ is also recognized by NFA_C. To simplify notation, without loss of generality, we may use $L(c_i)$ instead of $L(Token(c_i))$. Then the language specified by NFA_C over the initial alphabet Σ is

$$L(NFA_C) = \{x_1 \ldots x_n \in \Sigma^* | \exists c_1 \ldots c_n \in C^* \wedge \delta(q_0, c_1 \ldots c_n) \in F \wedge x_i \in L(c_i), 1 \leq i \leq n\}$$

2.2.3 Lexicon specification equations

The lexicon of a programming language is specified by equations of the form $LHS = RHS$ where LHS is a string used as a token name and RHS is a regular expression of conditions on the universal lexemes. A collection of these equations is used to build an automaton that will recognize the constructs specified by them. This automaton consumes universal lexemes from the FLS and constructs *tokenized lexemes* of the source language. Such a lexeme is a triple: $\langle Name, Lexeme, Length\rangle$, where *Name* is the LHS string of an equation, *Lexeme* is the string resulting from the concatenation of the Lex attributes of the universal lexemes consumed by the automaton before entering a final state, and *Length* is the length of *Lexeme*. As before, there will certainly be more information required in a second level lexeme, but we present here a simplified view. We have developed formal rules for lexicon specification by regular expressions of conditions that are equations of the form:

$$Name = \langle Descriptor\rangle\ [\ [\ |\ \langle Descriptor\rangle\ ] \ldots]\ [\ \langle Context\rangle]\ \langle Semantics\rangle\ |\ \textbf{self};$$

where $[\ |\ \langle Descriptor\rangle\ ] \ldots$ denotes any number of optional choices of $\langle Descriptor\rangle$. The informal specifcation of the $Descriptor, Context$, and $Semantics$ is given by the following BNF rules:

$\langle Descriptor\rangle$ ::= **body**: $\langle E\rangle$ | **begin**: $\langle E\rangle$ **body**: $\langle E\rangle$ **end**: $\langle E\rangle$ |
begin: $\langle E\rangle$ **body**: **any_until_end** **end**: $\langle E\rangle$ |

begin: $\langle E \rangle$ **body**: $\langle E \rangle$ **end**: $\langle E \rangle$ **recursive**

$\langle E \rangle$ ::= $\langle Condition \rangle$ | $\langle E \rangle$ $\langle E \rangle$ | $\langle E \rangle$ "|" $\langle E \rangle$ | "(" $\langle E \rangle$ ")" " *" | "[" $\langle E \rangle$ "]"

$\langle Context \rangle$ ::= **context**: $\{\langle \langle E \rangle, \langle E \rangle \rangle\}$ | **noncontext**: $\{\langle \langle E \rangle, \langle E \rangle \rangle\}$

$\langle Semantics \rangle$ ::= [**action** : $\langle ActList \rangle$];

$\langle Condition \rangle$ is a conditional expression and $\langle E \rangle$ is a regular expression of conditions as described earlier. The operator **any_until_end** used in this specification is a notational convenience which denotes a condition which is satisfied by all tokens which do not satisfy the conditional expression specified by the $\langle E \rangle$ in the clause **end** : $\langle E \rangle$. Similarly, a condition represented by **any_tok** can be used to refer to any single, arbitrary universal lexeme. The clause **context**: $\{\langle \langle E \rangle, \langle E \rangle \rangle\}$ denotes a list of pairs of regular expressions of conditions specifying the context in which lexical item $Name$ can be found and the clause **noncontext:** $\{\langle \langle E \rangle, \langle E \rangle \rangle\}$ denotes a list of pairs of regular expressions of conditions specifying the context in which $Name$ cannot be found. $\langle ActList \rangle$ represents the list of actions to be performed when a lexical construct specified by this equation is recognized. The components using the keywords **begin, body, end, context**, and **noncontext** define the syntactical portion of a lexical specification rule, while the keyword **action** defines the semantic portion. The keyword **recursive** will allow nested begin and end sequences. The brackets [ ] are essentially used as a keyword and indicates that the first level tokens recognized by the enclosed expression should not be included in the constructed second level lexeme.

Many terminals, such as reserved words and punctuation symbols, would be specified by equations defining precisely that terminal. Thus, we allow them to be specified by lexical equations of the form $TokName$ = **self** where $self$ is a regular expression of conditions that recognizes as valid only the string $TokName$. For example, the Pascal assignment operator := is specified by the lexical equation

" := " = **self**

in which case **self** represents the regular expression of conditions

$Token = O$ *and* $Lex =$ " : " $Token = O$ *and* $Lex =$ " = "

Example 2: The lexical equation defining the usual form of an identifier is:

"*id*" = **body**: Token = I (Token = I | Token = N | Token = O and Lex = "_")*

Then, given the sequence of first level tokens from Example 1, the SLS algorithm could use this equation to allow the construction of the second level token: $\langle id,$ "*var*_3", $5 \rangle$.

Some lexical constructs are more easily defined by specifying a beginning and an ending sequence. Comments and strings can be defined in this manner. In order to specify such constructs, we can use the two keywords **begin** and **end**. These keywords *must* be paired.

Example 3: Comments in many languages such as C or JavaScript could be defined by the following equation using the **begin-end** pairs.

c_style_com= **begin**: Token = O and Lex = “/” Token = O and Lex = ”*”
body: **any_until_end**
end: Token = O and Lex = “*” Token = O and Lex = ”/”

Another feature that is necessary in a scanner is the ability to recognize “syntactic sugar” in a construct which does not appear in the final tokenized lexeme. An example of this is the way special characters in strings are treated in many languages. Since a string is delimited by quotes, in order to place a quote in the string itself, the usual convention is to place two quotes together. However, the resulting string should only contain one quote. This feature is implemented by placing square brackets, [], around an expression. This indicates that the enclosed expression must be matched by some universal lexeme(s) from the *FLS*, but these lexemes will not be included in the lexeme being constructed. In Example 4 below, the second quote in the body of the string will not appear in the result.

Example 4: A partial specification for Modula-2 strings would be:

string= **begin**: Token = O and Lex = “””
body: (**any_tok** | Token = O and Lex = “”” [Token = O and Lex = “”””])*
end: Token = O and Lex = “””

The keyword **recursive** will allow nested begin and end sequences, as in the definition of a Modula-2 comment given in the Example 5 below.

Example 5: A specification for Modula-2 comment would be:

mod_com= **begin**: Token = O and Lex = “(” Token = O and Lex = ”*”
body: **any_until_end**
end: Token = O and Lex = “*” Token = O and Lex = ”)” **recursive**

Example 6: the specification of lexical entities whose character sets are not disjoint is illustrated by the following regular expression of conditions that specifies C hex numbers:

c_hex_num = **body**: Token=N and Lex=0
(Token = I and Lex=”x” |
Token=I and LexChar(0)=’x’ and LexChar(1,len) in [’A’..’F’])
(Token = N | Token = I and LexChar(0,len) in [’A’..’F’])*

The information associated with a lexical equation by the keyword **context** signifies that in order to be recognized by that lexical equation a lexical construct must be matched by the regular expression of conditions from the right-hand side of the lexical equation and must be discovered in the given context. The information associated with a lexical equation by the keyword **noncontext** signifies that if a lexical construct is matched by the regular expression

of conditions in the right-hand side of the lexical equation and is discovered in that context then this lexical construct should **not** be recognized by this equation. Only one or the other can be specified, since context and noncontext must be mutually exclusive. Following the keywords **context** and **noncontext** is a list of pairs of conditional expressions defining the left and right context or noncontext. The special keyword **none** can appear as the only left or right context, indicating no context checking, while the keywords **bof** (beginning of file) can be included as the first constant of the left context, and **eof** (end of file) can be the last constant of the right context.

Example 7: Suppose two constructs called *label* and *number* must be recognized, where *label* must be an integer with the context consisting of a colon following it, whereas *number* is an integer not followed by colon. Then the specifications would be:

label = **body**: Token=N
context: ⟨ **none** , Token=O and Lex=":" ⟩;
number = **body**: Token=N
noncontext: ⟨ **none**, Token=O and Lex=":" ⟩;

This provides discrimination of identically defined constructs based on the context only.

3 Implementation of the lexical analyzers

The lexical analyzer reads the input text, tokenizes it according to the lexicon specification, and performs specific actions whenever a lexical item is discovered. The input text is read by the first level scanner, FLS, which is a conventional DFA generated from the regular expressions specifying the universal lexemes. The tokenizing function is implemented by the second level scanner, SLS, which is a non-conventional NFA. This NFA reads and evaluates conditions on universal lexemes returned by the FLS rather than reading symbols of a character alphabet. This section describes the implementation of FLS and SLS. The integration of FLS and SLS into a user customized lexical analyzer is discussed more thoroughly in Section 4

3.1 First level lexicon implementation

The FLS is automatically generated from the specification of the first level lexical entities by the conventional technology of mapping regular expressions into finite automata which recognize the language specified by these expressions [7]. This technology consists of the following rules:

1. Let the implementation of the algorithm in [7] be `ScanGen`.

2. Let the specification of the first level lexical entities

$$\begin{aligned} I &= \textit{Letter Letter}^* \\ N &= \textit{Digit Digit}^* \\ W &= \textit{white spaces and tabs} \\ O &= \textit{punctuation, operators, and other characters} \\ U &= \textit{unprintable characters} \end{aligned}$$

   be given in a file with suffix `.FLS`, such as `f.FLS`.

3. `ScanGen` takes as the input `f.FLS` and generates a scan table, `ST.h`, that controls the finite automaton $DFA(f.FLS) = < Q, \Sigma, \delta : Q \times \Sigma \rightarrow Q, 0, F >$ where $Q = \{0, 1, 2, 3, 4, 5, \}$, $F = \{1, 2, 3, 4, 5\}$, shown in Figure 1. $DFA(f.FLS)$ recognizes the

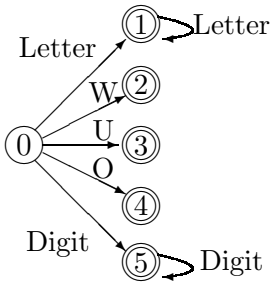

Figure 1: DFA for the First Level Scanner

   language generated by the regular expressions provided in `f.FLS`. The entries of the scan table `ST.h` are tuples $ST[i][j] = < state, char >$ where $state \in Q$ and $char$ is a character of the language alphabet.

4. The scanner `FLS` reads characters from an input text file and behaves according to an implementation of the DFA in Figure 1.

This methodology for first level scanner generation is shown in Figure 2. Note that `FLToken` is a data type designed by the scanner implementor. We assume here that `FLToken` is defined by the following data structure:

char TokenClass	char Lexeme[MaxLength]	int Len

In reality, we also keep syntactic details such as start-line, start-column, end-line, and end-column. Furthermore, it is also necessary to collect any information that may be needed for later processing and to pass it on so that it is not lost.

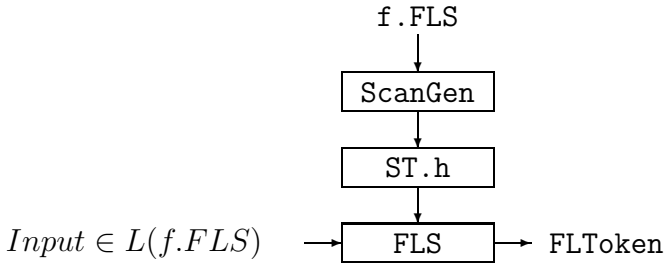

Figure 2: First Level Scanner generation

3.2 Second level lexicon implementation

In a conventional NFA, the transition table is indexed by the current state and an alphabet symbol [3, 5, 13]. The modified NFA implementing SLS uses conditions rather than symbols as the alphabet. The set of conditions C that make up the alphabet is given in the second level scanner specification, called the lexicon specification file, in the form of a file with suffix .LSF, such as `g.LSF`. It is impractical to use the conditions in C as indices. However, the token class attribute of any condition in a regular expression of conditions is constant over that condition. Thus, the class specifier of the conditions can be used as the column index in the transition table controlling the NFA. Hence, the state transition of the $NFA(g.LSF) = < Q, C, \delta : Q \times C \rightarrow Q, s_0, F >$ implementing SLS is controlled by a two-level table structured as follows:

1. The Transition Table, TT, is a two dimensional table whose rows are labeled by the states of the $NFA(g.LSF)$ and the columns are labeled by the elements of the set $\{I, N, W, U, O, \epsilon\}$ that are the tokens (i.e., class specifiers) of the first level lexicon.

2. Each entry $TT[q][t]$, $q \in Q$, $t \in \{I, N, W, U, O, \epsilon\}$ is a list of transitions $\{T_1, T_2, \ldots, T_k\}$. Each T_i, $1 \leq i \leq k$, is a tuple of the form $\langle cond_i, state_i \rangle$ where $cond_i$ is an index to a condition in the Condition Table, CT. This condition is to be evaluated when the automaton is in state q and the current FLS lexeme, $ulex$, is in class t. $State_i$ shows the destination state of the transition performed by the NFA, i.e.,

$$\delta(q, cond_i, ulex) = \begin{cases} state_i, & \text{if } \mathcal{V}(CT[cond_i](ulex)) = true; \\ undefined, & \text{otherwise.} \end{cases}$$

   where $\mathcal{V} : C \rightarrow \{true, false\}$ is a mapping that evaluate conditions. If $\delta(q, cond_i, ulex)$ is undefined the automaton remains in state q and performs $\delta(q, cond_{i+1}, ulex)$, for $i = 1, 2, \ldots, k$.

We have developed an automatic implementation of the second level lexical analyzer, SLS, Figure 3. This implementation contains three phases and is performed by three tools as follows:

Phase 1: develop the BNF rules that specify the language of the lexical equations used as lexical specification mechanism. These rules define a grammar, `LexGram`, which is the same for every lexicon that one needs to specify with our system and is given in the Appendix A. However, it certainly can be modified if necessary.

Phase 2: develop the lexicon specification file, $PL.LSF$, which contains an element of the language specified by the BNF rules in `LexGram` at Phase 1. An example of such a file is the Fortran 77 lexicon specification given in the Appendix B.

Phase 3: a constructor algorithm called $LSFParser$ reads the lexical equations from $PL.LSF$ file and constructs the NFA that recognizes the lexical constructs specified by them (see Section3.2.1).

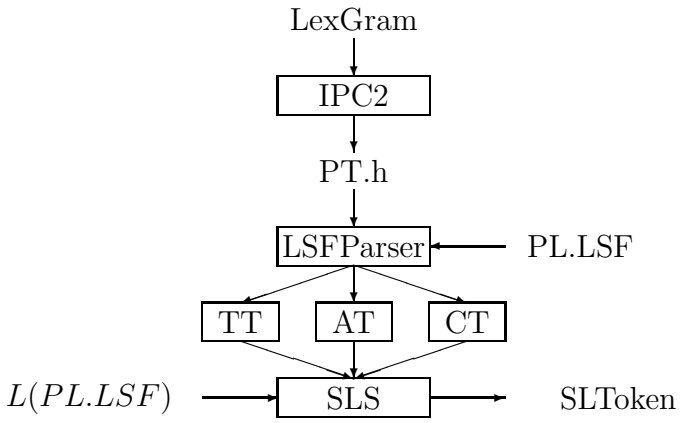

Figure 3: Methodology for SLS construction

The default grammar, `LexGram`, is usually sufficient so the user activity is commonly limited to the steps that begin with the modification or creation of PL.LSF.

3.2.1 LSFParser implementation

The specification language used to write the lexical equations in $PL.LSF$ is defined by the LALR-grammar [2] `LexGram` given in Appendix A. To provide for customization of the lexical analyzer to a particular language we allow each specification rule in `LexGram` to be augmented with one or more functions that are to be performed by the algorithm that constructs the lexical analyzer. That is, each specification rule in `LexGram` may have the form $LHS = RHS\ ;\ Function$. These rules are mapped by `IPC2` [9, 12] into an extended parse table whose entries are tuples $\langle Action, Function \rangle$ where $Action$ is the usual LALR-action and $Function$ is a reference to the corresponding function provided in the specification. Only the function name is provided in `LexGram` and stored in the parse table. The function definitions are provided separately. This function will be called when the parser makes a

reduction. That is, the $LSFParser$ is an LALR parser controlled by the extended parse table constructed by IPC2 which reads the equations in $PL.LSF$, checks their syntactic validity, and maps them into the TT and CT tables required by SLS, and the AT table required by further applications.

3.2.2 Mapping regular expressions of conditions into finite automata

Every regular expression of conditions recognized by $LSFParser$ is composed of four kind of elements: **conditions** expressing properties of the universal lexemes, **keywords** (such as `recursive, context, begin, end,[]`, etc.) which may indicate non-regular features of the lexeme specification, **regular operators** ($|$, $\circ$, $*$), and **semantic actions**. The regular operators are used by $LSFParser$ to generate the automaton recognizing lexemes; conditions are mapped by $LSFParser$ into postfix form and then are stored in the condition table, CT; keywords are mapped into predefined programs; semantic actions are mapped into records in the action table allowing future processors to perform the desired functions on the lexemes discovered by SLS.

Each equation from the lexicon specification file will correspond to an NFA. The automaton for the equation, $LHS_i = RHS_i$, will recognize the language described by the regular expression of conditions in RHS_i in order to construct a token from the language identified by LHS_i. Thus NFA_i will be constructed for $LHS_i = RHS_i$ for each conditional equation, as seen in Figure 4.

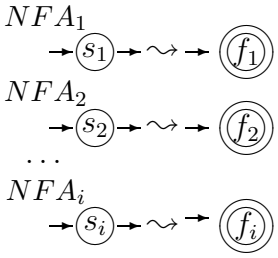

Figure 4: Collection of NFAs

Each NFA_i, $i = 1, 2, \ldots$, is characterized by the following information: the start state, which is an index in TT where the state s_i is stored, the final state, which is the index in TT where the state f_i is stored, token name, which is LHS_i, lexeme rule, which is RHS_i, the action to be performed when LHS_i is recognized, which is the function associated with the rule $LHS_i = RHS_i$ used to construct NFA_i, and the context in which the token specified by RHS_i is found. As was described, each transition of such an NFA is labeled by a property. If this property is a condition, then the transition to the next state is made only if the condition is satisfied by the current universal lexeme. Each condition is converted into a CT (condition table) entry when the lexicon specification file is processed. Following the

explanation of this process, the creation of the transition table will be presented to complete the NFA construction.

An entry in the CT represents a condition used in the second level lexicon specification equations recognized by $LSFParser$. For implementation reasons, each condition is mapped first into a postfix form and then is stored into CT as a record of the form:

$$\langle n : Entry_1, Entry_2, \ldots, Entry_n \rangle$$

where n is the number of elements in the expression and $Entry_k$, $k = 1, 2, \ldots, n$, is a meta-variable, a constant value from the domain of some meta-variable, or an operator. Meta-variables, such as LEX, or LEN, denote attributes of lexemes returned by the FLS. Operators are relations and logical connectors. Relational operators REL are comparison operators used to construct conditions, i.e., $REL \in \{EQ, NE, GE, LE, GT, LT\}$; logical connectors AND, OR, or NOT allow conditions to be combined.

Example 8: The specification rule

dumb = **body**: Token = I and Len <= 8 (Token=O and Lex = "@" |
Token = O and Lex = "=" or Lex = ">")

describes a language with elements such as `ab@`, `cdef=`, and `ghi>` and is a regular expression over three conditions:

c_1 : $Token =$ *I and Len* $<=$ 8
c_2 : $Token =$ *O and Lex* = "@"
c_3 : $Token =$ *O and Lex* = " = " *or Lex* = " > "

The names c_i, $1 \leq i \leq 3$, will be used further to refer to these conditions. Figure 5 shows $NFA(dumb.LSF) = \langle \{q, r, s, t\}, \{c_1, c_2, c_3\}, \delta, q, \{s, t\} \rangle$ that recognizes the language specified by this regular expression of conditions. Recognizing an element of class `dumb` would consume exactly two FLTokens.

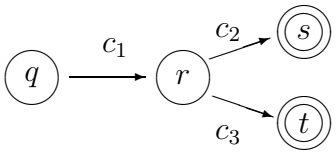

Figure 5: Conditional NFA for Example 8

For the NFA in Figure 5, the transition from state q to state r can be made only if condition c_1 is satisfied by the current universal lexeme. As noted earlier, in the implementation of $NFA(dumb.LSF)$ the class specifier of each condition is factored out leaving only the properties over the other attributes of the FLS lexemes. For the above example this leads to the three properties of the universal lexemes:

p_1 : $Len <= 8$
p_2 : $Lex =$ "@"
p_3 : $Lex =$ " = " *or* $Lex =$ " > "

The names p_i, $1 \leq i \leq 3$, will be used further to refer to these properties. This means that the transition from q to r is allowed if the FLS lexeme is in class I and satisfies the property p_1; transition from r to s is allowed if FLS lexeme is in class O and satisfies the property p_2; transition from r to t is allowed if FLS lexeme is in class O and satisfies the property p_3. To test if a property p is satisfied by the current universal lexeme, `ulex`, the algorithm `EvalCond(`I_p`,ulex)` defined below, that operates on a stack of strings, is applied to the condition p identified by the index I_p in CT.

```
1  Boolean EvalCond( int index, FLToken ulex )
2    Cond = CT[index]
3    Len = Cond[0]
4    for (i = 1; i <= Len; ++i)
5      if (Cond[i] in {LEX, LEN})
6        val = ulex.Cond[i]; Push( val, Stack )
7      else if (Cond[i] in ConstantValues)
8        Push(Cond[i],Stack)
9      else if (Cond[i] in {AND, OR, LT, LE, EQ, GE, GT, NE})
10       val2 = Pop(Stack); val1 = Pop(Stack); res  = Cond[i](val1,val2);
11       Push(res,Stack)
12     else if (Cond[i] in {NOT})
13           val1 = Pop(Stack); res  = Cond[i](val1); Push(res,Stack)
14     else return SystemError;
15   return Top(Stack)
```

The Boolean value on top of the stack when the algorithm terminates is the result of evaluating the condition found in the condition table at $CT[index]$ on the current input token, `ulex`. Demonstrations of `EvalCond()` evaluating the properties p_1, p_2, p_3 are presented in Example 9.

Example 9: the LSFParser stores the properties p_1, p_2, p_3 into the CT as shown in Table 1. The algorithm `EvalCond()` used by the NFA evaluates these properties as follows:

I_{p_1}	3	LEN	8	LE				
I_{p_2}	3	LEX	"@"	EQ				
I_{p_3}	7	LEX	" = "	EQ	LEX	" > "	EQ	OR

Table 1: Condition table for example 8

1. if $ulex = \langle I, \text{“}sam\text{”}, 3\rangle$ and $index = I_{p_1}$ then the algorithm performs the following stack operations:

i	Cond[i]	Stack (Top on right)
1	LEN	3
2	8	3 8
3	LE	LE(3,8)=`true`
		`true`

2. if $ulex = \langle O, \text{“} < \text{”}, 1\rangle$ and $index = I_{p_2}$ then the algorithm performs the following stack operations:

i	Cond[i]	Stack (Top on right)
1	LEX	“<”
2	“@”	“<” “@”
3	EQ	EQ(“@”,“<”)=`false`
		`false`

3. if $ulex = \langle O, \text{“} > \text{”}, 1\rangle$ and $index = I_{p_3}$ then this algorithm performs the following stack operations:

i	Cond[i]	Stack (Top on right)
1	LEX	“>”
2	“=”	“>” “”=”
3	EQ	EQ(“>”,“=”)=`false`
4	LEX	`false` “>”
5	“>”	`false` “>” “>”
6	EQ	`false` EQ(“>”,“>”)=`true`
7	OR	OR`(false,true)=true`
		`true`

The NFAs constructed by LSFParser to recognize properties p_i, $1 \leq i \leq 3$ are shown in Figure 6. The meaning of the first NFA, for example, is: *in state 1, if the current FLS*

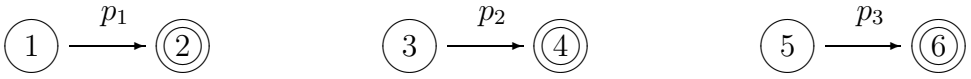

Figure 6: Building blocks for Example 8

lexeme, ulex, is of class I and satisfies the property $p1 : Len <= 8$ *then the automaton may*

perform a transition to state 2. In other words, given a universal lexeme from class I which is a string of no more than 8 characters, the NFA may make the transition from state 1 to state 2. Note that the difference between the algorithm used in [7] to construct a common NFA that recognizes regular expressions and our algorithm used to construct a conditional NFA that recognizes regular expression of conditions: NFAs constructed by [7] make transitions when they read the characters from the alphabet while conditional NFAs make transitions when they evaluate conditions on the lexemes returned by the FLS.

Now, in order to construct an NFA that recognizes regular expression of conditions we make the following assumptions:

a. The fundamental automata are of the structure shown in Figure 7 where ϵ is the

Figure 7: Fundamental automata for making up conditional NFA

empty conditional expression that evaluates a $True$ condition and does not consume a token and p is an conditional expression that contains no regular operators $(|, \circ.*)$ and evaluates to true or false.

b. Every conditional NFA has exactly one start node and exactly one final node.

With these assumptions the algorithm that constructs a conditional NFA consists of the following rules:

1. If a regular expression of condition consists of only one condition, p, (that is, it contains no regular operators $|, \circ, *$) then construct the conditional NFA that recognizes it as shown in Figure 7.

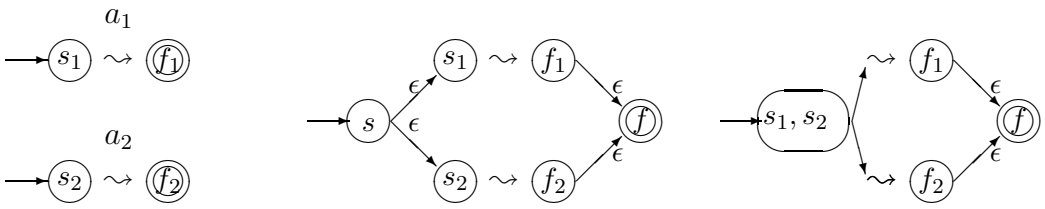

Figure 8: Mapping two automata a_1, a_2 into $a_1|a_2$

2. If the regular expression of conditions is of the form $e_1|e_2$ where e_1 is recognized by the conditional automaton a_1 and e_2 is recognized by the conditional automaton a_2 then

construct the automaton $a_1|a_2$ as shown if Figure 8. The node f is a new final state and the node labeled s_1, s_2 is the result of merging s_1 and s_2. Since merged nodes will be referred to by the first name, this one will be s_1.

3. If the regular expression of conditions is of the form $e_1 \circ e_2$ where e_1 is recognized by the conditional automaton a_1 and e_2 is recognized by the conditional automaton a_2 then construct the automaton $a_1 \circ a_2$ as shown if Figure 9

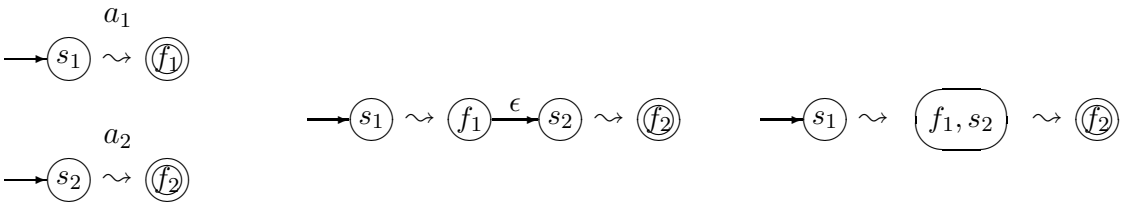

Figure 9: Mapping two automata a_1, a_2 into $a_1 \circ a_2$

4. If the regular expression of conditions is of the form e^* where e is recognized by the conditional automaton a then construct the automaton a^* as shown if Figure 10.

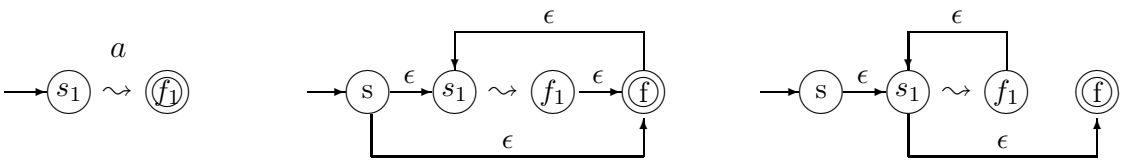

Figure 10: Mapping the automata a into the automata a^*

Observations:

- The conventional, post-construction optimization methodology does not easily apply to a conditional NFA because its transitions are determined by the truth values of some predicates.
- Optimization with the goal of eliminating nondeterminism, or otherwise simplifying an automaton, usually works by grouping states into equivalence classes, i.e., states reachable by an equivalent sequence of input. However, the question of deciding if two elements of a character alphabet are equivalent is much different from deciding if two conditions are equivalent. Thus we found that the usual optimization methodology can not be applied. However, to reduce the number of states and ϵ-transitions in the constructed NFA, we modified the construction rules by grouping states and eliminating ϵ-transitions at rule application rather then on a post-construction basis. If the NFA thus constructed is optimum or not is an open question.

- All construction rules maintain the property of a single start state with no in-coming transitions and a single final state with no out-going transitions. This restriction prevents the need to work inside the NFA.

Example 10: For the conditional expression from Example 8, and the conditions and properties previously labeled c_i and p_i, $1 \leq i \leq 3$, the construction algorithm will produce the intermediate stages shown in Figures 6 and 11. Compare this to the hand built NFA in Figure 5 or the one that would be build by the conventional rules (left as an exercise).

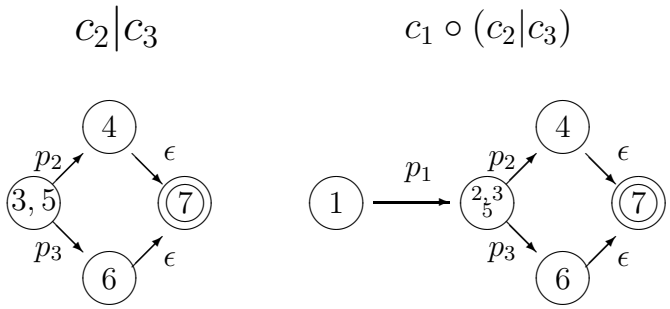

Figure 11: Compositional construction for Example 8

Recall that the table that controls the behavior of a conditional NFA is a transition table, $TT[s][c]$, where s is a state and c an first level token class. $TT[s][c]$ indicates the possible transitions that the NFA may make when in state s and given a universal lexeme, $ulex$, of class c is discovered in the input. Since this is an NFA, $TT[s][c]$ contains a list of transitions. Each transition indicates, by referring to a CT entry, the additional properties that $ulex$ must satisfy in order to allow the state transition to be made. It is important to note that entries representing ϵ-transitions will not consume the universal lexeme. The transition will be made on a constant `true` condition and the current token will be available to be used on the next transition as needed.

Example 11: Figure 12 and Table 2 displays the TT and CT tables that controls the NFA in Figure 11.

Keywords in the regular expression of conditions are mapped by $LSFParser$ into transitions in TT labeled by predefined programs. Actually, the constant `true` condition is a simple example of such a program. The records in TT supplied by the keywords implement the non-regular features of the lexicon specification. When $LSFParser$ discovers a keyword in a regular expression of conditions it may change the automaton constructed so far, introducing new states and generating predefined programs to be evaluated by the NFA when it reaches these states, see [8] for the example of such a change when **recursive** keyword is discovered. When a program representing a keyword is evaluated, it may generate a side-effect (such as incrementing or decrementing a counter, testing the value of a variable and

	I	N	O	U	W	ϵ
1	$\{T_1\}$					
2			$\{T_2, T_3\}$			
4						$\{T_4\}$
6						$\{T_5\}$
7						

Figure 12: Transition table of the automaton in Figure 11

Name	CT[index]	NextState
T_1	I_{p_1}	2
T_2	I_{p_2}	4
T_3	I_{p_3}	6
T_4	$True$	7
T_5	$True$	7

Table 2: Condition table of automaton in Figure 11

making a decision depending upon the result of this test, not including a portion of the text in the lexeme, etc) and returns a predefined Boolean value specific to the interpreted program. These commands provide the power to recognize virtually any construct in any programming language but are invisible to the user. The user only has to be concerned with knowing the keywords that can be used in the specification of the lexicon and what they mean. Thus, the goals of flexibility and ease of specification are achieved.

3.2.3 The Action Table

The purpose of the Action Table, AT, is to direct an algorithm as to how to construct entries in the database using a tokenized lexeme once SLS has recognized it. This processing, however, is beyond the scope of this paper. It takes place on tokens returned by SLS to later applications for further processing. An example of such a processing application would be a parser that uses such actions to construct a symbol table.

3.2.4 Algorithm implementing a conditional NFA

The algorithm `ConditionalNFA()` that implements a conditional NFA takes as input universal lexemes from the FLS, each of which is defined by the triple $\langle Token, Lex, Len\rangle$, and produces as output a tokenized lexeme of the language specified by a specification rule of the form $N = REC; Action$ where N is the token name, REC is a regular expression of conditions, and $Action$ is to be performed by the application program that uses

`ConditionalNFA()` to recognize lexemes specified by REC. This output is described here by the triple $\langle N, Lexeme, Length\rangle$ where $Lexeme$ is specified by REC and has the length $Length$. This algorithm is controlled by the tuple $nfa = \langle TT, CT, AT\rangle$ constructed by the LSFParser when it processes the specification rule $N = REC; Action$ as seen in Section 3.2.2. However, each specification rule $LHS_i = RHS_i; Action$ discovered in a lexicon specification file, .LSF, is mapped by the LSFParser into a tuple $nfa_i = \langle TT_i, CT_i, AT_i\rangle$. Thus, the algorithm `ConditionalNFA()` is customized to perform the actions of NFA_i specified by the regular expression of conditions RHS_i by parameterizing it in terms of $nfa_i = \langle TT_i, CT_i, AT_i\rangle$ and the universal lexemes returned by the FLS. Hence, the implementation of the `ConditionalNFA()` algorithm employs the following:

- A buffer of universal lexemes, `UniBuf`, used to store the lexemes read from the FLS. The location of the current lexeme in `UniBuf` is marked by a buffer pointer, *bp*.
- A transition stack, `TranStack`, upon which to store tuples of the form $\langle state, alt, bp\rangle$ where *state* is the current NFA state, *alt* is the next alternative transition from *state*, and *bp* is the buffer pointer locating the current FLS in the `UniBuf`.

Backtracking is implemented on this transition stack and is used to allow the trial of alternate paths in the NFA. The `TranStack` and `UniBuf` allow the NFA to back up in the input stream and to try other transitions from a given state when the current path fails. The algorithm `ConditionalNFA()` uses the `TranStack` to record potential backtracking locations by pushing and popping tuples $\langle state, alt, bp\rangle$ at each node which has alternative transitions. For that we introduce the transition type `transtype = (CT_index cond, TT_index state, transtype next)` as the type of *alt*. If backtracking returns to this node, the information found on the top of `TranStack` allows the next alternative transition to be attempted. This way the implementation technique of using sets of states at each transition is avoided.

Note that since an ϵ-transition consumes no universal lexeme, the presence of an ϵ-transition in a state of an automaton is an alternative to any non-ϵ transition from that state. This alternative is taken by the algorithm when all others fail. Hence, the ϵ-transition from a state must be pushed first on the transition stack `TranStack`. To simplify presentation, the two push operations performed when the automaton moves to a new state are called here `GoToState` which is described by:

```
GoToState:
 if ((tran = TT[state][epsilon].first) != NULL)
   push_TS(state, tran, ulex);
 if ((tran = TT[state][ulex.class].first) != NULL)
   push_TS(state, tran, ulex);
```

We use `push_TS()`, `pop_TS()`, and `empty_TS()` functions to operate on `TranStack` in the following description of the `ConditionalNFA()`:

```
1  ConditionalNFA ( NFA_type nfa, UniBuf_pointer ulex )
```

```
2    TT_index state; CT_index cond; transtype tran;
3    state  = nfa.start;
4    marker = ulex ;
5    GoToState;
6    while ( state != nfa.final && !is_empty_TS() )
7      (state, tran, ulex) = pop_TS();
8      if (tran.next  != NULL)
9        push_TS(state, tran.next, ulex);
10     cond = tran.cond;
11     if (cond is a CT_index)
12       if (EvalCond(CT[cond], ulex) == true)
13         state = tran.state; ulex  = ulex.next; GoToState;
14     else cond is a program
15       if (predefined program action)
16         state = tran.state; GoToState
17   end while
18   if (state == nfa.final) return true
19   else ulex = marker; return false
```

Observation: to decrease nondeterminism we remove the useless attempts by looking at the class specifier of the `ulex` before moving to the next state thus making sure that the current state has a viable transition.

Example 12: Consider the input text ... `sam > 34` ... and the conditional automaton specified in Example 8, whose condition table is Table 2, whose transition table is in Figure 12, and whose graphical representation can be seen in Figure 11. Assume that this text has been consumed by the FLS and the portion `sam > 34` is still available in `UniBuf` as shown in Figure 13. Further, assume that the algorithm `ConditionalNFA()` operates in

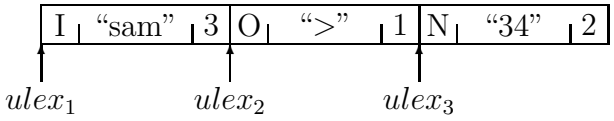

Figure 13: Text representation in `UniBuf`

.

this configuration where nfa.start = 1, nfa.final = 7. Here we examine the behavior of this algorithm by looking at the content of its `TranStack` during the algorithm operation:

- The algorithm pushes the tuple $(1, T_1, ulex_1)$ on the `TranStack` at line 5 and and enters the while loop at line 6.
- In the first iteration of the while loop, at line 7, the only item on the stack is popped out, giving `state = 1, tran =` T_1, and `ulex =` $ulex_1$. Since TT[1][I] = $\{T_1\}$, the

only transition from state 1 on class I is T_1, therefor `tran.next = NULL` and the stack is left empty. At line 11 the algorithm find T_1.cond to be CT the index I_{p_1}. It was shown in Example 9 that `EvalCond(CT[`I_{p_1}`], `$ulex_1$`) = true`. The transition T_1 to state 2 is performed and $ulex_1$ is consumed. Since TT[2][O] = $\{T_2, T_3\}$ and there is no ϵ transitions from state 2, (2, T_2, $ulex_2$) is pushed onto `TranStack` at line 13 and the algorithm returns to the while loop.

- In the second iteration of the while loop, at line 7, the top stack item is popped out, giving `state = 2, tran =` T_2, and `ulex=`$ulex_2$. T_2.next = T_3 is another alternative at this state, so (2, T_3, $ulex_2$) is pushed on the stack at line 9 for later examination. At line 11 the algorithm finds T_2.cond to be the CT index I_{p_2}. As seen at (2) in Example 9, `EvalCond(CT[`I_{p_2}`], `$ulex_2$`) = false`, and the loop is re-executed.

- In the third iteration of the while loop, at the line 7, the stack is popped again, `state = 2, tran=`T_3`, and ulex=`$ulex_2$. So a different transition from state 2 is attempted with $ulex_2$. At line 8 there are no other transitions from state 2 so the stack is left empty. At line 11 the algorithm finds T_3.cond to be the CT index I_{p_3}. `EvalCond(CT[`I_{p_3}`, `$ulex_2$`) = true`, as seen in Example 9. Thus, a transition to to state 6 on $ulex_3$ in the $UniBuf$ can be made. At line 13 the algorithm finds TT[6][ϵ] = $\{T_5\}$ and TT[6][N] = NULL. Thus, it pushes (6, T_5, $ulex_3$) on the stack. and returns to the loop.

- In the fourth iteration of the while loop, at line 7 the top item stack is popped out, giving `state = 6, tran=`T_5`, and ulex=`$ulex_3$. There are no other transitions from state 6, so the stack is left empty. Now, at line 11 the algorithm finds that T_5`.cond` is not a `CT_index`; at line 14 it finds that T_5`.cond` is the keyword `true`, which represents a predefined program. This program returns `true` for any universal lexeme. However, it does not consume input, so $ulex_3$ is left in the buffer. The destination for T_5 is state 7 which now becomes the current state for the algorithm.

- In the fifth iteration of the while loop, the algorithm finds `state = 7` to be final and the stack to be empty. At line 18, the successful `true` result is returned and the algorithm terminates.

The algorithm which calls `ConditionalNFA(nfa, ulex_1)` determines that no context was associated with the specification rule generating this `nfa`. Thus the information from $ulex_1$ and $ulex_2$ can be collected to construct $\langle$"$dumb$", "$sam >$", 4$\rangle$. as the outgoing second level token. The next section will examine how the input tokens produced by the FLS are made available to the calling algorithm and also what will be done with these output tokens.

3.2.5 The second level scanning algorithm

The second level scanning algorithm, SLS, recognizes a second level lexeme, SLL, of a lexical language specified by the specification rules $LHS_i = RHS_i; Action_1,\ i = 1, 2, \ldots, n$, where $RHS_i,\ i = 1, 2, \ldots, n$, are regular expression of conditions. The input of SLS is a

sequence of first level lexemes, $FLL = \langle Token, String, Length \rangle$. To simplify the description we will assume that these FLLs are provided in the buffer `UniBuf` and are accessible by the pointers `UniBuf.current` and `UniBuf.next`. Thus, SLS takes two parameters: the buffer `UniBuf` where a sequence of FLLs are found and the list of tuples `nfa` = $(TT_1, CT_1, AT_1), (TT_2, CT_2, AT_2), \ldots, (TT_n, CT_n, AT_n)$ that represent the nondeterministic conditional automata specified by equations $LHS_i = RHS_i; Action_i$, $i = 1, 2, \ldots, n$, that specify the second level lexicon. The algorithm SLS performs two major actions: first it uses `ConditionalNFA(nfa, UniBuf.current)` to determine if there is a sequence of FLLs beginning at `UniBuf.current` that may belong to SLL associated with nfa, and then, if context or noncontext is provided in the RHS it determines if the context of SLL in the `UniBuf` matches the context specified with RHS. If both tasks are successful, the recognized section of `UniBuf` is collected into a SLL structure and the `UniBuf` is advanced.

The algorithm `ConditionalNFA()` discussed in Section 3.2.4 is used by SLS as the computation engine to recognize the SLL as well as to recognize its context, if necessary. The right context is recognized by `ConditionalNFA(nfa_rc, UniBuf.current)` where `nfa_rc` is the NFA specified by the regular expression of conditions used in the right context of the specification rule; the left context is recognized by `ConditionalNFA(nfa_lc, UniBuf.mark)` where `nfa_lc` is the NFA specified by the regular expression of conditions used in the left context of the specification rule. When testing `nfa_lc`, the NFA consumes the FLLs moving to the left in the input. Hence, SLS can be informally specified as follows:

```
1  SecondLevelLexeme SLS (List_of_NFA_type nfa, UniBuf_pointer mark)
2     UniBuf_pointer ulex; Boolean result; SecondLevelLexeme SLL;
4     for each (nfa_i in nfa)
4        ulex = UniBuf.mark;
5        result = ConditionalNFA(nfa_i, ulex);
7        if ( ( (result == true) and (not(context)) and (not(noncontext)) )
8             or
9              ( (result == true) and (context)          and
10               (ConditionalNFA(nfa_rc, UniBuf.current) and
11               (ConditionalNFA(nfa_lc, UniBuf.mark) )
12            or
13             ( (result == true) and (noncontext)            and
14               (not(ConditionalNFA(nfa_rc, UniBuf.current)) or
15               (not(ConditionalNFA(nfa_lc, UniBuf.mark) )
16          )
18         SLL = (LHS_i, UniBuf.mark..UniBuf.current, Length); return SLL;
19    if (UniBuf.mark == UniBuf.current) return error; all nfa_i failed
```

4 Integrating FLS and SLS

The first level scanner, FLS, and the second level scanner, SLS, have been designed and implemented in Section 3 as stand alone tools. That is, these tools should be usable directly

in a language processing application where the user is required only to customize them to a particular language, say `mylang`. The customization of FLS and SLS to the language `mylang` is achieved by developing the lexicon specification of `mylang`. This can be done by first customizing FLS by developing the specification of the first level lexicon, i.e., by developing the file `mylang.FLS` and then by developing the specification of the second level lexicon, i.e., by developing the file `mylang.LSF`. One can also accept the universal lexemes as defined in this paper and use the implementation package we provide, developing only the file `mylang.LSF`. This is the usual course of action. However, in both cases one needs to join FLS and SLS together into a new tool, say MyScan. We have developed a methodology that allows the user to achieve this goal without special knowledge of what FLS and SLS are actually doing other than their input/output types. That is, given two stand alone tools, $Input_1 \rightarrow TOOL_1 \rightarrow Output_1$ and $Input_2 \rightarrow TOOL_2 \rightarrow Output_2$ that operate as input/output devices we provide a methodology to integrate these tools into a new tool $Input_3 \rightarrow TOOL_3 \rightarrow Output_3$. $TOOL_3$ is composed from $TOOL_1$ and $TOOL_2$ without knowledge of their internal functioning other than types of $Input_1$, $Output_1$, $Input_2$, and $Output_2$. This methodology provides the fundamentals for compositional development of language processing tools.

4.1 Functional integration of stand alone tools

A stand alone tool is a software package that takes a well defined input and produces a well defined output and can be used in an application without knowledge of its internal functioning. An example of a stand alone tool is FLS. The FLS takes as input a text file and produces as output an FLL object of type described by the data structure `FirstLevelLexeme`. The FLS can be offered to its users as a stand alone tool i.e., as a package of programs and documentation where the user is asked to know only that the input is a text file, the output is `FLL` of type `FirstLevelLexeme`, and the calling mechanism is `FLL = FLS(Text)`. An application in which this tool can be used as stand-alone component would have the form:

```
FLSApplication(Input Text, Output DataBase);
   FirstLevelLexeme FLL;
   Open (Text);
   FLL = FLS(Text);
   while (FLL.Token != EOF)
        {
         switch (mytoken.Token)
               {
                case I: { Result = process identifier }
                case N: { Result = process number }
                case W: { Result = process white space }
                case O: { Result = process other characters }
                case U: { Result = process unprintable character }
               }
         Update(DataBase, Result);
```

```
 FLL = FLS (Text);
}
```

An obvious example of an application that would like to use FLS as a component would be SLS. Integrating FLS into the SLS defines a user-customized scanner. Theoretically this means that we need to compose two functions $f_1 : I_1 \rightarrow O_1$ and $f_2 : I_2 \rightarrow O_2$ where $O_1 \neq I_2$. This can be achieved by a third function, which will be called a *filter*, that performs the conversion of O_1 to I_2, i.e., $F_{O_1,I_2} : O_1 \rightarrow I_2$, as seen in Figure 14.

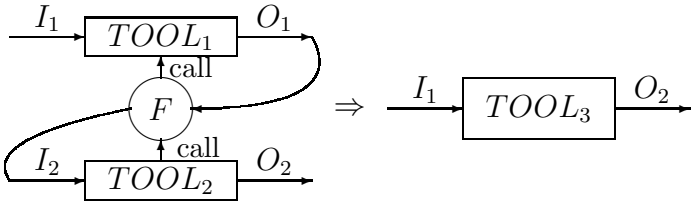

Figure 14: Functional integration of stand alone software tools

4.2 Integrating FLS and SLS into a user designed scanner

In order to apply the idea presented in Figure 14 for the integration of the `FLS()` and `SLS()` thus generating a user customized scanner, `MyScan()`, we observe that `FLS()` inputs a text file and returns `FirstLevelLexeme`s, i.e., $FLS : Text \rightarrow FirstLevelLexeme$, and `SLS()` inputs `FirstLevelLexeme`s and outputs `SecondLevelLexeme`s. In general, however, the output of the first tool may be different from the input of the second tool that make up the integrated tool. Therefore, to preserve the generality we assume that `SLS()` inputs tokens of type `SLInLexeme` and outputs `SecondLevelLexeme`, i.e., SLS : `SLInLexeme` $\rightarrow$ `SecondLevelLexeme`. A filter that would achieve the integration gets `FirstLevelLexeme`s from `FLS()` and maps them into the `SLInLexeme`s. The pseudo-code implementing this filter is:

```
SLInLexeme Filter (Text)
  {
   FirstLevelLexeme FLL;
   SLInLexeme       SLLin;
   FLL = FLS(Text)
   SLLin = map(FLL)
   return(SLLin)
  }
```

where *map*() performs the mapping of a `FirstLevelLexeme` into a `SLInLexeme`.

The SLS() algorithm discussed in Section 3.2.5 uses an input buffer to feed its computation engine, `ConditionalNFA()`. This buffer receives its content by calling such a filter function. In this way, the actual source of the input is irrelevant to the `SLS()`, as long as it is returned in the correct form. Hence, `MyScan()` obtained by the integration of `FLS()` and `SLS()` as discussed above has the following pseudo-code:

```
SecondLevelLexeme MyScan (Text)
  {
   Filter(Input Text);
   SLInLexeme        SLLin;
   SecondLevelLexeme SLLout;
   SLLin  = Filter (Text);
   SLLout = SLS (SLLin);
  }
```

One can use `MyScan()` in an application that inputs a text file `Text`, breaks it into second level lexemes, and processes each second level lexeme appropriately. `MyScan()` function will pass `Text` along on its call to a filter and wait for its input. The pseudo code of such an application would be:

```
ApplicationSLS (Input Text, Output DataBase)
   {
    SecondLevelLexeme SLL;
    Open(Text);
    SLL = MyScan(Text);
    while (SLL.Token != EOF)
         {
          switch ( SLL.Token )
                {
                 case LHS1 : { Result = process(LHS1) }
                 case LHS2 : { Result = process(LHS2) }
                 . . .
                 case LHSk : { Result = process(LHSk) }
                 default: process error
                }
           Update(DataBase, Result);
           SLL = MyScan (Text);
          }
   }
```

Note, `LHS1, LHS2, ..., LHSk` are the left-hand sides of the conditional equations specifying the second level lexicon using properties of the lexical elements of the first level lexicon. That is, the left hand sides of the equations in PL.LSF.

5 Conclusions

This paper reports on the effort to design a stand-alone lexical analyzer that is based on a specification written in terms of regular expressions of conditions. This is contrasted to the conventional approach of using regular expressions over a finite alphabet of a terminal characters. The major goals of this work are convenience of specification methodology for tool customization, flexibility provided by the language processing tools derived from this specification, and correctness of the tools developed in this way.

Implementing the scanner over two processing layers provides both advantages and disadvantages. The measurable advantages result from breaking the effort into two parts, each of which can be more easily understood. Regular expressions of conditions allow the definition of the lexicon over more abstract objects. The language designer can reason about higher-level constructs. Another important advantage is support for non-regular language features provided by keyword programs which may influence automaton construction or behavior. More importantly, the keyword facility is designed in a way which will allow extension by adding other features as they become necessary. On the other hand, this two level approach requires an additional level of interaction between the tools. The output of the first level scanner is usually passed on as the input to the second level scanner. The disadvantages we observed result from the additional processing step in the *scanner* which doesn't help the efficiency.

A major point requiring further study is the optimization of the conditional NFAs implementing the lexicon specification equations. The ultimate goal would be to construct a deterministic conditional automaton which would eliminate any worry about the effect that backtracking may have on the efficiency of the algorithm. However, at this time, we are unsure how to proceed with this optimization. To help alleviate concern over efficiency, it has been observed that in most cases very little backtracking actually occurs. Also, due to the structure of the specification expressions there are not that may places that would allow backtracking. An important optimization is obtained by the elimination of epsilon transitions and removing some states during the automaton construction rather than as a post construction operation. It is an open problem whether or not this kind of optimization may lead to a deterministic conditional NFA. But the goal to this point has been only the development of the specification rules, the extensibility of the tools, and the correctness of the resulting algorithms.

It is our belief that the first applications of our tools will be for prototype projects rather then large scale languages. Our methodologies provide for easy description of language processing tools. This will allow the language designer to specify and implement the language using convenient tools. The focus has been on designing the specification language and the techniques to generate tools from these specifications. Future work must certainly aim to make these tools more efficient. In the scope of this paper, we believe that most applications of a scanner will be over fairly small source constructs. Thus, the level of efficiency provided may not be terribly important, since nearly instantaneous results are not really discernible over such a small time frame. Another niche we envision for these tools is for the modular compiler designer.

There are many provisions to allow for the modification of the scanner generator. These include the formal specification of the language of conditions, the mechanical construction of the recognizing NFAs based on this specification, and the functional form of this specification. Adding new features to the conditions in the specification is an incremental process. Thus, a new feature can be added in a convenient manner. As alluded to earlier, this may include adding new non-regular construction rules.

Measurements of program performance are also intentionally left out of this paper. Test results can be construed in many ways and a fair trial is difficult to come by. This is especially true when comparing results of recent research projects to tools which have been evolving over several years. Also, as the implementor of the programs described in this project, it has been more important to produce understandable code that can be modified by future participants in the project rather than trying to squeeze out all of the possible efficiency. It is also hoped that compiler designers using these tools will be able to add capabilities as needed. A very important facility, which is almost impossible to measure, is the ease of specification development. Though [8] does provide some numbers for interested readers. However this too is easy to slant to particular views.

Overall, this work demonstrates the potential usefulness of this scanner technology. The specification over regular expressions of conditions has been demonstrated by many students to be quite understandable. The tools generated from these specifications have been used in several other projects and have performed well. Finally, the generation tools have evolved several times to add new features to the specifications language, and each time it has been a straight forward task. Though further work is undoubtedly needed, these methodologies demonstrate the basis of the work and hopefully provide a bit of insight into the potential we see for them.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *COMPILERS Principles, Techniques, and Tools*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] F. DeRemer and T. Penello. Efficient computation of lalr(1) look–ahead sets. *ACM Transactions on Programming Languages and Systems*, 4:615–649, 1981.

[3] F.L. DeRemer. Lexical analysis. In F Bauer, , and Eickel, editors, *Compiler Construction*, pages 109–145, Berlin Heidelberg New Yrok, 1976. Springer–Verlag.

[4] J. Grosch. Generators for high-speed front-ends. *Lecture Notes in Computer Science*, 321:80–92, 1989.

[5] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computations*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1979.

[6] R.N. Horspool and M.R. Levy. Mkscan – an interactive scanner generator. *Software – Practice and Experience*, 17(6):369–378, 1987.

[7] W.L. Johnson, J.H. Porter, S.I. Ackley, and D.T. Ross. Automatic generation of efficient lexical analyzers using finite state techniques. *Communications of the ACM*, 11(12):805–813, 1968.

[8] J. Knaack and T. Rus. Twolev: A two level scanning algorithms. In *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology, AMAST'91*, pages 175–179, Iowa City, IA, 52242, 22–25 May 1991.

[9] J.P. LePeau and T. Rus. Interactive parser construction. Technical Report 88–02, The University of Iowa, Department of Computer Science, Iowa City, IA 52242, 1988.

[10] M.E. Lesk. Lex – a lexical analyzer generator. Technical Report 39, Bell Telephone Laboratories, Computing Science, Murray Hill, NJ, 1975.

[11] H. Mössenböck. Alex – a simple and efficient scanner generator. *SIGPLAN Notices*, 21(5):69–78, 1986.

[12] T. Rus. Interactive parser generato. Avalilable at http://www.cs.uiowa.edu/~rus/Courses/Compiler, 1996.

[13] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[14] E. Visser. Multi-level specifications. In *Language Prototyping – an algebraic specification approach*, AMAST Series in Computing, Vol. 5, pages 105–197. World Scientific, 1996.

[15] W.M. Waite, V.P. Heuring, and R.W. Gray. Gla – a generator for lexical analyzers. Technical Report 86–1–1, University of Colorado at Boulder, Department of Electrical and Computer Engineering, Boulder, CO, 1986.

A BNF specification of lexical equation language

Rules that have the same left hand side and are associated with the same semantics processing function are written in the form $LHS = RHS_1|RHS_2|\ldots|RHS_k; Function$

LHS	=	RHS	Function
LSF	=	Specs ;	
Specs	=	Properties Eqs ;	
Properties	=	Property | Properties Property ;	
Property	=	"keyword_case_sensitive" "=" YesNo ";";	keyword_sensitive
Property	=	"name_case_sensitive" "=" YesNo ";";	case_sensitive
Property	=	"space_separator" "=" YesNo ";";	space_separator
Property	=	"tab_separator" "=" YesNo ";";	tab_separator
Property	=	"newline_separator" "=" YesNo ";";	newline_separator
YesNo	=	"yes";	answer_yes
YesNo	=	"no";	answer_no
Eqs	=	Eq | Eqs Eq ;	
Eq	=	LHS "=" Desc Semantics	
		| LHS "=" Desc	
		| LHS "=" Desc Context Semantics	
		| LHS "=" Desc Context ;	
LHS	=	"string" ;	hold_lhs_name
Desc	=	Delims | Body ;	end_rec
Desc	=	"self" ;	do_self_split
Delims	=	Delim | Delims "|" Delim ;	
Delim	=	Begin Body End ;	begin_end
Delim	=	Begin AnyToEnd End ;	begin_any_end
Delim	=	Begin Body End "recursive";	not_yet
AnyToEnd	=	"body" ":" "any_until_end" ;	do_any_tok
Begin	=	"begin" ":" REC ;	
Body	=	"body" ":" REC ;	
End	=	"end" ":" REC ";" ;	
REC	=	REC "|" REC1 ;	do_choice
REC	=	REC1 ;	
REC1	=	REC1 REC2 ;	do_compose
REC1	=	REC2 ;	
REC2	=	REC3 "*";	do_star
REC2	=	REC3 ;	
REC3	=	Cond ;	do_cond
REC3	=	NCond ;	
REC3	=	"any_tok";	do_any_tok
REC3	=	"(" REC ")";	
REC3	=	"[" REC "]";	discard_lex

```
Cond       =  ClSpec | ClSpec "and" Expr ;
NCond      =  "not" "(" Cond ")";                                  not_cond
ClSpec     =  Class | "token" "=" Class ;
Class      =  "i" | "n" | "f" | "w" | "u" | "o" ;                  hold_class
Expr       =  Expr1 ;
Expr       =  Expr "or" Expr1 ;                                    do_cond_or
Expr1      =  Expr2 ;
Expr1      =  Expr1 "and" Expr2 ;                                  do_cond_and
Expr2      =  "not" Expr2 ;                                        do_cond_not
Expr2      =  "lex" Rel "int";                                     lex_int
Expr2      =  "len" Rel "int";                                     len_int
Expr2      =  "lin" Rel "int";                                     line_int
Expr2      =  "col" Rel "int";                                     column_int
Expr2      =  "(" Expr ")";
Expr2      =  "lexchar" "(" Index ")" Rel Elem ;                   lexch1e
Expr2      =  "lexchar" "(" Index ")" "in" Set ;                   lexch1s
Expr2      =  "lexchar" "(" Index ")" "in" Range ;                 lexch1r
Expr2      =  "lexchar" "(" Index "," Index ")" Rel Elem ;         lexch2e
Expr2      =  "lexchar" "(" Index "," Index ")" "in" Set ;         lexch2s
Expr2      =  "lexchar" "(" Index "," Index ")" "in" Range ;       lexch2r
Set        =  "{" ElList "}" ;                                     set
ElList     =  ElList "," Elem ;                                    set_add
ElList     =  Elem ;                                               set_init
Range      =  "[" Elem ".." Elem "]" ;                             range
Index      =  "int" ;                                              index_int
Index      =  "len" ;                                              index_len
Elem       =  "int" | "char" ;                                     element
Rel        =  "=" | "!=" | ">" | ">=" | "<" | "<=";                relation
Context    =  C_or_N ":" CList ;                                   new_machine
C_or_N     =  "context";                                           context
C_or_N     =  "noncontext";                                        noncontext
CList      =  LElem ;                                              add_con_pair
CList      =  LElem "," CList ;                                    add_con_list
LElem      =  "<" LE "," RE ">";                                   make_con_pair
LE         =  "none" | "bof" | REC ;                               collect_left
RE         =  "none" | "eof" | REC ;                               collect_right
Semantics  =  "action" ":" Actions ;
Actions    =  Action | Actions "," Action ;
Action     =  "addns" ;                                            addns
Action     =  "nofif" ;                                            nofif
```

B Partial Fortran 77 lexicon specification

```
case_sensitive=yes;
"comment" = begin: Token = I and Lex = "C" and Col = 1
            body:  any_until_end
            end:   Token=U and Lex="\n";
            action: nofif
"continuation" = body: ( Token=I and Col = 6 and Len = 1 )
                     | ( Token=N and Col = 6 and Len = 1 )
                     | ( Token=O and Col = 6 and Len = 1 )
                 action: nofif
"sequence_number" = body: Token=N and Col > 72
                 action: nofif
"label" = body: Token=N and Col<6
                 action: addns
"power_op" =  body: Token=O and Lex="*"  Token=O and Lex="*"
"concat_op" = body: Token=O and Lex="/"  Token=O and Lex="/"
"rel_op" = body:(Token=O and Lex="."Token=I and Lex="EQ"Token=O and Lex="." )
         | ( Token=O and Lex="."  Token=I and Lex="NE"  Token=O and Lex="." )
         | ( Token=O and Lex="."  Token=I and Lex="LT"  Token=O and Lex="." )
         | ( Token=O and Lex="."  Token=I and Lex="LE"  Token=O and Lex="." )
           action: addns
"and_op" =  body: Token=O and Lex="."  Token=I and Lex="AND"  Token=O and Lex="."
"equiv_op" = body:(Token=O and Lex="." Token=I and Lex="EQV" Token=O and Lex=".")
            | ( Token=O and Lex="." Token=I and Lex="NEQV" Token=O and Lex="." )
"string_constant" = begin: [Token = O and Lex = "'"]
           body:  (any | Token = O and Lex = "'" [Token = O and Lex = "'"])*
           end:   [Token = O and Lex = "'"];
           action: addns
"id_name" = body: Token=I ( Token=I | Token=N | Token=O and Lex="_" )*
            action: addns
"(/" = self
"/)" = self
"(" = self
")" = self
"*" = self
"+" = self
"=>" = self
" =" = self
"allocate" = self
"call" = self
"close" = self
"write" = self
```

C Partial C lexicon specification

```
keyword_case_sensitive=no;
name_case_sensitive=yes;
space_separator = yes;
tab_separator = no;
newline_separator = yes;
"%"  = self
"&"  = self
"("  = self
")"  = self
"*"  = self
"+"  = self
","  = self
"-"  = self
"/"  = self
":"  = self
";"  = self
"->"  = self
"=="  = self
"char"  = self
"do"  = self
"else"  = self
"for"  = self
"if"  = self
"int"  = self
"while"  = self
"LEFT_SHIFT" = body: Token=O and Lex="<" Token=O and Lex="<"
"RIGHT_SHIFT" = body: Token=O and Lex=">" Token=O and Lex=">"
"IDENTIFIER" = body: Token=I (Token=I | Token=N | Token=O and Lex="_")*
    action: addns
"FLOATLITERAL" = body: Token=N Token=O and Lex="." Token=N
    action: addns
"INTEGER" = body: Token=N
    noncontext: <none, Token=O and Lex = ".">, <none,Token=I and LexChar(0)='x'>
    action: addns
"STRING" = begin:Token=O and Lex="""  body:any_until_end end:Token=O and Lex="""  ;
    action: addns
"COMMENT" = begin: Token=O and Lex="/" Token=O and Lex="*"
    body: any_until_end end:Token=O and Lex="*" Token=O and Lex="/" ;
    action: nofif, addns
"HEXA_LITERAL" = body: Token=N and Lex=0 (Token=I and Lex="x" |
            Token=I and LexChar(0)='x' and LexChar(1,len) in ['A'..'F'])
            ((Token=N) | (Token=I and LexChar(0,len) in ['A'..'F']))*
    action: addns
```