Lexical Analysis

Formal Language
A language which can be defined by a precise specification and is amenable for use with computers.

Natural Language
A language used by people, which cannot be defined perfectly with a precise specification system.

Lexical Analysis

Language Elements
SET
- A set is a collection of unique objects.
- A set may contain an infinite number of objects.
- Empty set is a set with no elements and designate it either by {} or by φ .

Example:
{boy, girl, animal} is a set of words
{girl, boy, animal, girl} is the same set as {boy, girl, animal} .

Lexical Analysis

STRING

- A *string is a list of characters from a given alphabet.*
- *The elements of a string* need not be unique, and the order in which they are listed is important.
- For example,"abc" and "cba" are different strings, as are "abb" and "ab".
- A null string is a string which consists of no characters *and designate it by ε.*

Lexical Analysis

Language
A set of strings from a given alphabet.

Examples of languages from the alphabet {0,1}:
1. {0,10,1011}
2. {}
3. {ε,0,00,000,0000,00000,...}
4. The set of all strings of zeroes and ones having an even number of ones.

Examples of languages from the alphabet of characters available on a computer keyboard:
1. {0,10,1011}
2. {ε}
3. Java syntax
4. Italian syntax

Lexical Analysis

Finite State Machine

A theoretical machine consisting of a finite set of states, a finite input alphabet, and a state transition function which specifies the machine's state, given its present state and the current input.

Lexical Analysis

A finite state machine consists of:

1. A finite set of states, one of which is designated the starting state, and zero or more of which are designated accepting states. The starting state may also be an accepting state.
2. A state transition function which has two arguments – a state and an input symbol (from a given input alphabet) – and returns as result a state.

Lexical Analysis

How the FS machine works?

1. The input is a string of symbols from the input alphabet.
2. The machine is initially in the starting state.
3. As each symbol is read from the input string, the machine proceeds to a new state as indicated by the transition function, which is a function of the input symbol and the current state of the machine.

Lexical Analysis

How the FS machine works?

4. When the entire input string has been read, the machine is either in an accepting state or in a non-accepting state.
5. If it is in an accepting state, then we say the input string has been accepted. Otherwise the input string has not been accepted, i.e. it has been rejected.

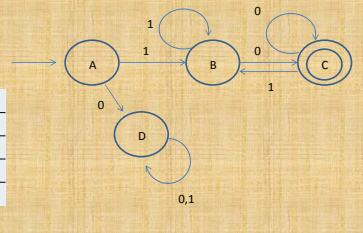
Lexical Analysis

Finite State Machine

The set of all input strings which would be accepted by the machine form a language, and in this way the finite state machine provides a precise specification of a language.

Lexical Analysis

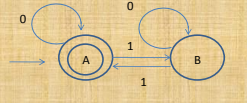
	0	1
A	D	B
B	C	B
* C	C	B
D	D	D



This machine accepts any string of zeroes and ones which begins with a one and ends with a zero

Lexical Analysis

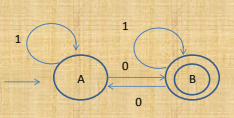
	0	1
* A	A	B
B	B	A



This machine accepts any string of zeroes and ones which contains an even number of ones (which includes the null string)

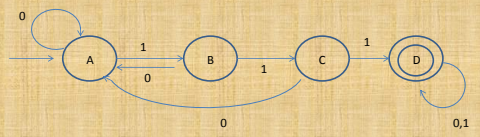
Lexical Analysis

	0	1
A	B	A
* B	A	A



Strings containing an odd number of zeros (the input alphabet is {0,1})

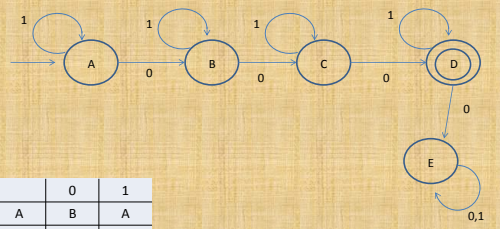
Lexical Analysis



	0	1
A	A	A
B	A	C
C	A	D
* D	D	D

Strings containing three consecutive ones (the input alphabet is {0,1})

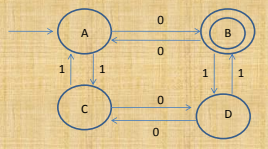
Lexical Analysis



	0	1
A	B	A
B	C	B
C	D	C
* D	E	D
E	E	E

Strings containing exactly three zeros (the input alphabet is {0,1})

Lexical Analysis



	0	1
A	B	C
* B	A	D
C	D	A
D	C	B

Strings containing an odd number of zeros and an even number of ones (the input alphabet is {0,1})

Lexical Analysis

Regular Expression

An expression involving three operations on sets of strings – union, concatenation, and Kleene * (also known as closure).

Lexical Analysis

Union

The union of two sets is that set which contains all the elements in each of the two sets and nothing else.

The union operation on languages is designated with a '+'. For example,
{abc,ab,ba} + {ba,bb} = {abc,ab,ba,bb}

Note that the union of any language with the empty set is that language:
L + {} = L

Lexical Analysis

Concatenation

This operation will be designated by a raised dot (whether operating on strings or languages), which may be omitted.

This is simply the juxtaposition of two strings forming a new string. For example,

abc . ba = abcba

Note that any string concatenated with the null string is that string itself:

s . ε = s

Lexical Analysis

Concatenation

The concatenation of two languages is that language formed by concatenating each string in one language with each string in the other language.

For example,
{ab,a, c} . {b,ε} = {ab.b,ab.ε,a.b,a.ε,c.b, c.ε}
= {abb,ab,a,cb,c}

Note that if L1 and L2 are two languages, then L1 . L2 is not necessarily equal to L2 . L1.
Also, L . {ε} = L, but L . φ = φ

Lexical Analysis

Kleene *

This operation is a unary operation (designated by a postfix asterisk) and is often called closure.

If L is a language, we define:
L0 = {ε}
L1 = L
L2 = L . L
Ln = L . Ln-1
L* = L0 + L1 + L2 + L3 + L4 + L5 + ...
Note that φ* = {ε}.

Lexical Analysis

Kleene *

Precedence may be specified with parentheses, but if parentheses are omitted, concatenation takes precedence over union, and Kleene * takes precedence over concatenation.

If L1 , L2 and L3 are languages, then:
L1+ L2 . L3 = L1 + (L2.L3)
L1.L2* = L1.(L2*)

Lexical Analysis

Regular Expression

An example of a regular expression is: (0+1)*
To understand what strings are in this language, let L = {0,1}.

We need to find L*:
L0 = {ε}
L1 = {0,1}
L2 = L.L = {00,01,10,11}
L3 = L.L2 = {000,001,010,011,100,101,110,111}
L* = {ε, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101,110, 111, 0000, ...}
= the set of all strings of zeros and ones.

Lexical Analysis

Regular Expression

Another example:

1.(0+1)*.0
= 1(0+1)*0
= {10, 100, 110, 1000, 1010, 1100, 1110, ...}

= the set of all strings of zeros and ones which begin with a 1 and end with a 0.

Lexical Analysis

Regular Expression

Note that we do not need to be concerned with the order of evaluation of several concatenations in one regular expression, since it is an associative operation.

The same is true of union:
L.(L.L) = (L.L).L
L+(L+L) = (L+L)+L

Lexical Analysis

Regular Expression

The regular expression (0*1)* could generate the string 0001101. The outer * repeats three times; the first time the inner * repeats three times, the second time the inner * repeats zero times, and the third time the inner * repeats once.

Lexical Analysis

Regular Expression

Sample Problem
For each of the following regular expressions, list six strings which are in its language.
1. (a(b+c)*)*d
2. (a+b)*.(c+d)
3. (a*b*)* ε

Solution:
1. d ad abd acd aad abbcbd
2. c d ac abd babc bad
3. a b ab ba aa

Note that (a*b*)* = (a+b)*

Lexical Analysis

Regular Expression

Sample Problem
Give a regular expression for each of the languages described in Sample Problem 2.0 (a)

Solutions:
1. 1*01*(01*01*)*
2. (0+1)*111(0+1)*
3. 1*01*01*01*
4. (00+11)*(01+10)(1(0(11)*0)*1+0(1(00)*1)*0)*1(0(11)*0)* + (00+11)*0

An algorithm for converting a finite state machine to an equivalent regular expression is beyond the scope of this text, but may be found in Hopcroft & Ullman [1979].

Lexical Analysis

Lexical Tokens

Lexical analysis attempts to isolate the "words" in an input string.

A *word, also known as a lexeme, a lexical item, or a lexical token, is a string of* input characters which is taken as a unit and passed on to the next phase of compilation.

Lexical Analysis

Lexical Tokens

Examples of words are:

1) ***keywords - while, if, else, for, …*** These are words which may have a particular predefined meaning to the compiler, as opposed to identifiers which have no particular meaning. Reserved words are keywords which are not available to the programmer for use as identifiers.

2) ***identifiers*** - words that the programmer constructs to attach a name to a construct, usually having some indication as to the purpose or intent of the construct. Identifiers may be used to identify variables, classes, constants, functions, etc.

Lexical Analysis

Lexical Tokens

3) ***operators*** **-** symbols used for arithmetic, character, or logical operations, such as +,-,=,!=, etc. Notice that operators may consist of more than one character.

4) ***numeric constants*** **-** numbers such as 124, 12.35, 0.09E-23, etc. These must be converted to a numeric format so that they can be used in arithmetic operations, because the compiler initially sees all input as a string of characters. Numeric constants may be stored in a table.

Lexical Analysis

Lexical Tokens

5)character constants - single characters or strings of characters enclosed in quotes.

6)special characters - characters used as delimiters such as .,(,),{,},;,:. These are generally single-character words.

Lexical Analysis

Lexical Tokens

7)comments - Though comments must be detected in the lexical analysis phase, they are not put out as tokens to the next phase of compilation.

8)white space - Spaces and tabs are generally ignored by the compiler, except to serve as delimiters in most languages, and are not put out as tokens.

9)newline - In languages with free format, newline characters should also be ignored, otherwise a newline token should be put out by the lexical scanner.

Lexical Analysis

Lexical Tokens

The output of this phase is a stream of tokens, one token for each word encountered in the input program. Each token consists of two parts:

(1) a class indicating which kind of token and
(2) a value indicating which member of the class.

Lexical Analysis

Lexical Tokens

```
while ( x33 <= 2.5e+33 - total ) calc ( x33 ) ; //!
  1   6  2  3     4     3   2   6  2   6  2  6 6
Token       Token
Class       Value
   1        [code for while]
   6        [code for (]
   2        [ptr to symbol table entry for x33]
   3        [code for <=]
   4        [ptr to constant table entry for 2.5e+33]
   3        [code for -]
   2        [ptr to symbol table entry for total]
   6        [code for )]
   2        [ptr to symbol table entry for calc]
   6        [code for (]
   2        [ptr to symbol table entry for x33]
   6        [code for )]
   6        [code for ;]
```

Lexical Analysis

Symbol Table

A data structure used to store identifiers and possibly other lexical entities during compilation.

Lexical Analysis

Symbol Table

❖During lexical analysis, a **symbol table** is constructed as identifiers are encountered***.***

❖This is a data structure which stores each identifier once, regardless of the number of times it occurs in the source program.

Lexical Analysis

Symbol Table

- It also stores information about the identifier, such as the kind of identifier and where associated run-time information (such as the value assigned to a variable) is stored.
- This data structure is often organized as a binary search tree, or hash table, for efficiency in searching.

Lexical Analysis

Symbol Table

- When compiling block structured languages such as Java, C, or Algol, the symbol table processing is more involved.
- Since the same identifier can have different declarations in different blocks or procedures, both instances of the identifier must be recorded.

Lexical Analysis

Symbol Table

- This can be done by setting up a separate symbol table for each block, or by specifying block scopes in a single symbol table.
- This would be done during the parse or syntax analysis phase of the compiler; the scanner could simply store the identifier in a string space array and return a pointer to its first character.

Lexical Analysis

Lexical Tokens

Numeric constants must be converted to an appropriate internal form.

For example, the constant "3.4e+6" should be thought of as a string of six characters which needs to be translated to floating point (or fixed point integer) format so that the computer can perform appropriate arithmetic operations with it.

Lexical Analysis

Lexical Tokens

The lexical analysis phase does not check for proper syntax.

The input could be `} while if ( {` and the lexical phase would put out five tokens corresponding to the five words in the input.

Lexical Analysis

Lexical Tokens

If the source language is not case sensitive, the scanner must accommodate this feature.

A preprocessor could be used to translate all alphabetic characters to upper (or lower) case.

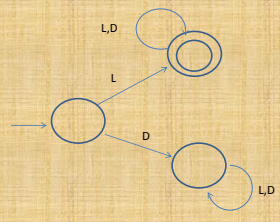
Implementation with Finite State Machines

```
boolean [] accept = new boolean [STATES];
int [][] fsm = new int[STATES][INPUTS]; // state table
                                // initialize table here...
int inp = 0;                    // input symbol (0..INPUTS)
int state = 0;                  // starting state;
try
{
   inp = System.in.read() - '0'; // character input,
                                 // convert to int.
   while (inp>=0 && inp<INPUTS)
   {
      state = fsm[state][inp];      // next state
      inp = System.in.read() - '0'; // get next input
    }
}
catch (IOException ioe)
{
   System.out.println ("IO error " + ioe);
}
   if (accept[state]) System.out.println ("Accepted");
       System.out.println ("Rejected"); } }
```

Lexical Analysis

Example of Finite State Machines for Lexical Analysis

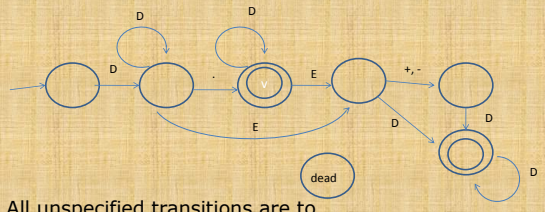
A finite state machine which accepts any identifier beginning with a letter and followed by any number of letters and digits. The letter "L" represents any letter (a-z), and the letter "D" represents any numeric digit (0-9).



Lexical Analysis

Example of Finite State Machines for Lexical Analysis

A finite state machine which accepts numeric constants and the letter "D" represents any numeric digit (0-9).

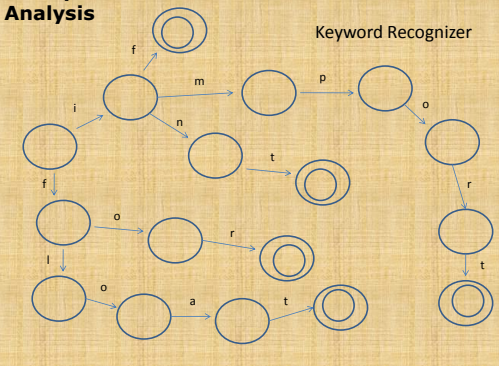


All unspecified transitions are to the "dead" state.

Lexical Analysis

Example of Finite State Machines for Lexical Analysis

Keyword Recognizer



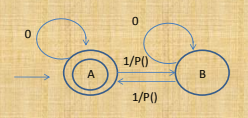
Lexical Analysis

Actions for Finite State Machines

- Lexical analysis may involve building a symbol table, converting numeric constants to the appropriate data type, and putting out tokens.
- For this reason, we can associate an action, or function to be invoked, with each state transition in the finite state machine.
- For example, to put out keyword tokens corresponding to each of the keywords recognized by the machine, we could associate an action with each state transition in the finite state machine.

Lexical Analysis

Actions for Finite State Machines




```
void P()
{ if (parity==0) parity
= 1;
else parity = 0;
}
```

A machine is to generate a parity bit so that the input string and parity bit will always have an even number of ones. The parity bit, parity, is initialized to 0 and is complemented by the function P().

Lexical Analysis

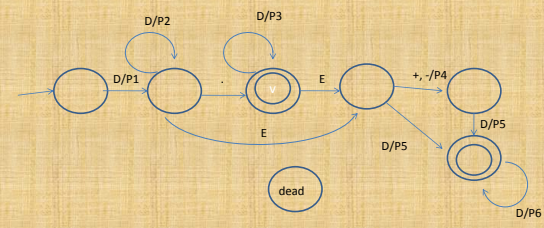
Sample Problem
Design a finite state machine, with actions, to read numeric strings and convert them to an appropriate internal numeric format, such as floating point.

Solution:
In the state diagram, we have included function calls designated P1(), P2(), P3(), ... which are to be invoked as the corresponding transition occurs.

A transition marked i/P() means that if the input is i, invoke function P() before changing state and reading the next input symbol.

Lexical Analysis

Solution:
The functions referred to in the state diagram are shown below:



All unspecified transitions are to the "dead" state.

Lexical Analysis

```
int Places, N, D, Exp, Sign; // global variables
void P1()
{
       Places = 0;   //Places after decimal point
       N = D;        // Input symbol is a numeric digit
       Exp = 0;      // Default exponent of 10 is 0
       Sign = +1;    // Default sign of exponent is
             // positive
}
void P2()
{
       N = N*10 + D; // Input symbol is a numeric digit
}
void P3()
{
       N = N*10 + D; // Input symbol is a numeric digit
                     // after a decimal point
       Places = Places + 1; // Count decimal places
}
```

Lexical Analysis

```
void P4()
{
       if (input=='-') then sign = -1; // sign of exponent
}
void P5()
{
       Exp = D; // Input symbol is a numeric digit in the
                // exponent
}
void P6()
{
       Exp = Exp*10 + D; // Input symbol is a numeric
                         // digit in the Exponent
}
```

The value of the numeric constant may then be computed as follows:

`Result = N * Math.pow (10, Sign*Exp - Places);`
where `Math.pow(x,y) = x^y`

Lexical Analysis

Lexical Tables

Lexical tables could include a symbol table for identifiers, a table of numeric constants, string constants, statement labels, and line numbers for languages such as Basic.

Lexical Analysis

Lexical Tables

Sequential Search

- ✓ The table could be organized as an array or linked list. Each time a word is encountered, the list is scanned and if the word is not already in the list, it is added at the end.
- ✓ The time required to build a table of n words is $O(n^2)$.
- ✓ This method is generally not used for symbol tables, or tables of line numbers, but could be used for tables of statement labels, or constants.

Lexical Analysis

Lexical Tables
Binary Tree

- The table could be organized as a binary tree.
- Having the property that all of the words in the left subtree of any word precede that word (according to a sort sequence), and all of the words in the right subtree follow that word called a binary search tree.

Lexical Analysis

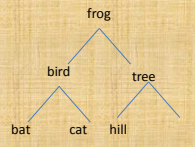
Lexical Tables
Binary Search Tree

- Since the tree is initially empty, the first word encountered is placed at the root.
- Each time a word, w, is encountered the search begins at the root; w is compared with the word at the root.
- If w is smaller, it must be in the left subtree; if it is greater, it must be in the right subtree; and if it is equal, it is already in the tree.

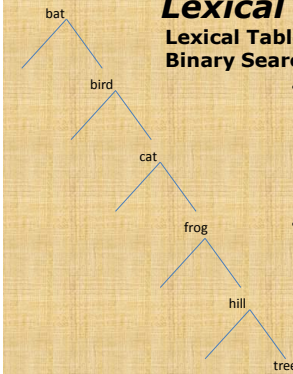
Lexical Analysis

Lexical Tables
Binary Search Tree

- This is repeated until w has been found in the tree, or we arrive at a leaf node not equal to w, in which case w must be inserted at that point.



Lexical Analysis



Lexical Tables
Binary Search Tree

- It is possible for the tree to take the form of a linked list (in which case the tree is said not to be balanced).
- The time required to build such a table of n words is $O(n \log^2 n)$ in the best case (the tree is balanced), but could be $O(n^2)$ in the worst case (the tree is not balanced).

Lexical Analysis

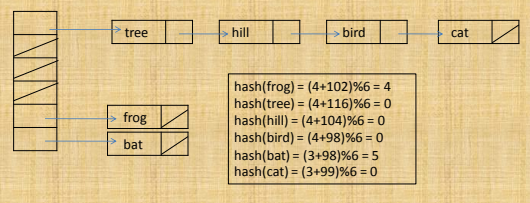
Lexical Tables
Hash Table

- ✓ We start with an array of null pointers, each of which is to become the head of a linked list.
- ✓ A word to be stored in the table is added to one of the lists. A hash function is used to determine which list the word is to be stored in.
- ✓ This is a function which takes as argument the word itself and returns an integer value which is a valid subscript to the array of pointers.

Lexical Analysis

Lexical Tables
Hash Table

- ✓ The corresponding list is then searched sequentially, until the word is found already in the table, or the end of the list is encountered, in which case the word is appended to that list.



Lexical Analysis

Lexical Analysis with SableCC

SableCC and JavaCC are two porpular utility programs for lexical analysis using java program. SableCC has several advantages over JavaCC:

- SableCC is designed to make good use of the advantages of Java; it is object oriented and makes extensive use of class inheritance.
- With SableCC compilation errors are easier to fix.

Lexical Analysis

Lexical Analysis with SableCC

- SableCC generates modular software, with each class in a separate file.
- SableCC generates syntax trees, from which atoms or code can be generated.
- SableCC can accommodate a wider class of languages than JavaCC (which permits only LL(1) grammars).

Lexical Analysis

Lexical Analysis with SableCC

SableCC Input File

The input to SableCC consists of a text file, named with a .grammar suffix, with six sections.

1. Package declaration
2. Helper declarations
3. States declarations
4. Token declarations
5. Ignored tokens
6. Productions

The sections on Helper declarations, States declarations, and Token declarations are relevant to lexical analysis.

Lexical Analysis

Lexical Analysis with SableCC

SableCC Input File

The input file, named *language.grammar will be arranged as shown* below:

```
Package package-name;

Helpers
     [ Helper declarations, if any, go here ]
States
     [ State declarations, if any, go here ]
Tokens
     [ Token declarations go here ]
```

Lexical Analysis

Lexical Analysis with SableCC

Token Declarations

- A Token declaration takes the form:
  Token-name = Token-definition ;
  For example: `left_paren = '(' ;`
- A Token definition may be any of the following:
  A character in single quotes, such as `'w'`, `'9'`, or `'$'`.
- A number, written in decimal or hexadecimal, representing the ascii (actually unicode) code for a character. Thus, the number 13 represents a newline character (the character '\n' works as well)

Lexical Analysis

Lexical Analysis with SableCC

Token Declarations

A set of characters, specified in one of the following ways:

1. A single quoted character qualifies as a set consisting of one character.
2. A range of characters, with the first and last placed in brackets:

```
['a'..'z'] // all lower case letters
['0'..'9'] // all numeric characters
[9..99]    // all characters whose codes are in
           // the range 9 through 99,inclusive
```

Lexical Analysis
Lexical Analysis with SableCC
Token Declarations

3. A union of two sets, specified in brackets with a plus as in [set1 + set2].
   Example:
```
[['a'..'z'] + ['A'..'Z']] // represents all
                          // letters
```
4. A difference of two sets, specified in brackets with a minus as in [set1 - set2]. This represents all the characters in set1 which are not also in set2.
   Example:
```
[[0..127] - ['\t' + '\n']]
      // represents all ascii characters
      // except tab and newline.
```

5. A string of characters in single quotes, such as 'while'.

Lexical Analysis
Lexical Analysis with SableCC
Token Declarations

Regular expressions; If p and q are token definitions, then so are:

(p) parentheses may be used to determine the order of operations
pq the concatenation of two token definitions is a valid token definition.
p|q the union of two token definitions (note the plus symbol has a different meaning).

Lexical Analysis
Lexical Analysis with SableCC
Token Declarations

Regular expressions; If p and q are token definitions, then so are:

p* the closure (kleene *) is a valid token definition, representing 0 or more repetitions of p.
p+ similar to closure, represents 1 or more repetitions of the definition p.
p? represents an optional p, i.e. 0 or 1 repetitions of the definition p.

Lexical Analysis
Lexical Analysis with SableCC
Token Declarations

If s1 and s2 are sets, s1+s2 is their union.
If p is a regular expression, p+ is also a regular expression.
To specify union of regular expressions, use '|'.

```
number = ['0'..'9']+ ; // A number is 1 or more
                       // decimal digits
identifier = [['a'..'z']+['A'..'Z']]
(['a'..'z'] | ['A'..'Z'] | ['0'..'9'] | '_')* ;
          // An identifier must begin with an
                // alphabetic character.
rel_op = ['<' + '>'] '='? | '==' | '!=' ;
                // Six relational operators
```

Lexical Analysis
Token Declarations
Lexical Analysis with SableCC
Token Declarations

When two token definitions match input, the one matching the longer input string is selected.

When two token definitions match input strings of the same length, the token definition listed first is selected.

```
Tokens
identifier = ['a'..'z']+ ;
keyword = 'while' | 'for' | 'class' ;
```

Lexical Analysis
Lexical Analysis with SableCC
Token Declarations

An input of 'while' would be returned as an identifier, as would an input of 'whilex'. Instead the tokens should be defined as:

```
Tokens
keyword = 'while' | 'for' | 'class' ;
identifier = ['a'..'z']+ ;
```

With this definition, the input 'whilex' would be returned as an identifier.

Lexical Analysis

Lexical Analysis with SableCC

Helper Declarations

Any helper which is defined in the Helpers section may be used as part of a token defnition in the Tokens section.

Lexical Analysis

Lexical Analysis with SableCC

Helper Declarations

```
Helpers
  digit = ['0'..'9'] ;
  letter = [['a'..'z'] + ['A'..'Z']] ;
  sign = '+' | '-' ;
  newline = 10 | 13 ; // ascii codes
  tab = 9 ;           // ascii code for tab
Tokens
  number = sign? digit+ ; // A number is an optional
                          // sign, followed by 1 or more
                         // digits.
  identifier = letter (letter | digit | '_')* ;
                          // An identifier is a letter
                         // followed by 0 or more
                        // letters, digits,
                       // underscores.
  space = ' ' | newline | tab ;
```

Lexical Analysis

Lexical Analysis with SableCC

Sample Problem
Show the sequence of tokens which would be recognized by the preceding definitions of number, identifier, and space for the following input (also show the text which corresponds to each token):

```
334 abc abc334
```

Solution:

```
number 334
space
identifier abc
space
identifier abc334
```

Lexical Analysis

Lexical Analysis with SableCC

State Declarations, Left Context, and Right Context

```
States
     statename1, statename2, statename3,... ;
```

The first state listed is the start state; the scanner will start out in this state.

In the Tokens section, any definition may be preceded by a list of state names and optional state transitions in curly braces.

Lexical Analysis

Lexical Analysis with SableCC

State Declarations, Left Context, and Right Context

The definition will be applied only if the scanner is in the specified state:

```
{statename} token = def ;
  // apply this definition only if the scanner is
 // in state statename (and remain in that state)
```

A transition operator , ->.,may follow any state name inside the braces:

```
{statename->newstate} token = def;
// apply this definition only if the scanner is
// in statename,and change the state to new
// state.
```

Lexical Analysis

Lexical Analysis with SableCC

State Declarations, Left Context, and Right Context

A definition may be associated with more than one state:

```
{state1->state2, state3->state4, state5} token = def;
    // apply this definition only if the scanner is
    // in state1 (change to state2), or if the
    // scanner is in state3(change to state4), or if
    // the scanner is in state5(remain in state5).
```

Definitions which are not associated with any states may be applied regardless of the state of the scanner:

```
token = def; // apply this definition regardless of the
             // current state of the scanner.
```

Lexical Analysis

Lexical Analysis with SableCC

State Declarations, Left Context, and Right Context

The following example is taken from the SableCC web site. Its purpose is to make the scanner toggle back and forth between two states depending on whether it is at the beginning of a line in the input.

```
States
  bol, inline; // Declare the state names. bol is
               // the start state.
Tokens
  {bol->inline, inline} char = [[0..0xfff] - [10 + 13]];
              // Scanning a non-newline char. Apply
              // this in either state, New state is
              // inline.
  {bol, inline->bol} eol = 10 | 13 | 10 13;
             // Scanning a newline char. Apply this in
             // either state. New state is bol.
```

Lexical Analysis

Lexical Analysis with SableCC

State Declarations, Left Context, and Right Context

Sample Problem

Show the token and state definitions needed to process a text file containing numbers, currency values, and spaces. Currency values begin with a dollar sign, such as '$3045' and '$9'. Assume all numbers and currency values are whole numbers. Your definitions should be able to distinguish between currency values (money) and ordinary numbers (number). You may also use helpers.

Lexical Analysis

Lexical Analysis with SableCC

State Declarations, Left Context, and Right Context

Solution:

```
Helpers
num = ['0'..'9']+ ; // 1 or more digits
States
def, currency;      // def is start state.
Tokens
space = (' ' | 10 | 13 | '\t') ;
{def -> currency} dollar = '$' ;
                          // change to currency
{currency -> def} money = num;
                          // change to def
{def} number = num;       // remain in def
```

Lexical Analysis

An Example of a SableCC Input File

```
Package lexing ;
     // A Java package is produced for the
     // generated scanner
Helpers
  num = ['0'..'9']+;
     // A num is 1 or more decimal digits
  letter = ['a'..'z'] | ['A'..'Z'] ;
     // A letter is a single upper or
     // lowercase character.
```

Lexical Analysis

An Example of a SableCC Input File

```
Tokens
  number = num;
  //A number token is a whole number
  ident = letter (letter | num)* ;
  //An ident token is a letter followed by
  // 0 or more letters and numbers.
  arith_op = [ ['+' + '-' ] + ['*' + '/' ] ];
  // Arithmetic operators
  rel_op = ['<' + '>']| '=='|'<='|'>='|'!=';
  // Relational operators
  paren = ['(' + ')'];          // Parentheses
  blank = (' '|'\t'|10|'\n')+;  // White space
  unknown = [0..0xffff];
  // Any single character which is not part
  // of one of the above tokens.
```

Lexical Analysis

Running SableCC

```
package lexing;
import lexing.lexer.*;
import lexing.node.*;
import java.io.*; // Needed for pushbackreader and
                  // inputstream
class Lexing
{
  static Lexer lexer;
  static Object token;
  public static void main(String [] args)
  {
    lexer = new Lexer;
    (new PushbackReader
      (new InputStreamReader (System.in), 1024));
    token = null;
    try
    {
```

Lexical Analysis
Running SableCC

```
while ( ! (token instanceof EOF))
{
  token = lexer.next(); // read next token
  if (token instanceof TNumber)
    System.out.print ("Number: ");
  else if (token instanceof TIdent)
    System.out.print ("Identifier: ");
  else if (token instanceof TArithOp)
    System.out.print ("Arith Op: ");
  else if (token instanceof TRelOp)
    System.out.print ("Relational Op: ");
  else if (token instanceof TParen)
    System.out.print ("Parentheses ");
  else if (token instanceof TBlank) ;
       // Ignore white space
  else if (token instanceof TUnknown)
    System.out.print ("Unknown ");
  if (!(token instanceof TBlank))
    System.out.println (token); // print token as a string
}
}
```

Lexical Analysis
Running SableCC

sablecc languagename.grammar
sablecc lexing.grammar

javac languagename/.java*
javac lexing/*.java

We have now generated the scanner in lexing.Lexing.class.

To execute the scanner:
java languagename.Classname

Lexical Analysis
Running SableCC

In our case this would be:
java lexing.Lexing

This will read from the standard input file (keyboard) and should display tokens as they are recognized. Use the end-of-file character to terminate the input (ctrl-d for unix, ctrl-z for Windows/DOS).

```
java lexing.Lexing
sum = sum + salary ;
Identifier: sum
Unknown =
Identifier: sum
Arith Op: +
Identifier: salary
Unknown ;
```

Lexical Analysis
Case Study: Lexical Analysis for Decaf

```
Helpers // Examples
  letter = ['a'..'z'] | ['A'..'Z'] ;      // w
  digit = ['0'..'9'] ;                    // 3
  digits = digit+ ;                    // 2040099
  exp = ['e' + 'E'] ['+' + '-']? digits; // E-34
  newline = [10 + 13] ;                   // '\n'
  non_star = [[0..0xffff] - '*'] ;        // /
  non_slash = [[0..0xffff] - '/'];        // *
  non_star_slash = [[0..0xffff] - ['*' + '/']];
                                          // $
```

Lexical Analysis
Case Study: Lexical Analysis for Decaf

A language which is not case-sensitive, such as BASIC or Pascal, would require a different strategy for keywords.

The keyword while could be defined as

```
while = ['w' + 'W'] ['h' + 'H'] ['i' + 'I']
['l' + 'L'] ['e' + 'E'];
```

Alternatively, a preprocessor could convert all letters (not inside strings) to lower case.

Lexical Analysis
Case Study: Lexical Analysis for Decaf

```
Tokens
 comment1 = '//' [[0..0xffff]-newline]* newline ;
 comment2 = '/*' non_star* '*'
 (non_star_slash non_star* '*'+)* '/' ;
 space = ' ' | 9 | newline ; // 9 = tab
 clas = 'class' ; // key words (reserved)
 public = 'public' ;
 static = 'static' ;
 void = 'void' ;
 main = 'main' ;
```

Lexical Analysis

Case Study: Lexical Analysis for Decaf

```
string = 'String' ;
int = 'int' ;
float = 'float' ;
for = 'for' ;
while = 'while' ;
if = 'if' ;
else = 'else' ;
assign = '=' ;
compare = '==' | '<' | '>' | '<=' |'>='|'!=';
plus = '+' ;
minus = '-' ;
mult = '*' ;
div = '/' ;
```

Lexical Analysis

Case Study: Lexical Analysis for Decaf

```
l_par = '(';
r_par = ')';
l_brace = '{';
r_brace = '}';
l_bracket = '[';
r_bracket = ']';
comma = ',';
semi = ';';
identifier = letter(letter|digit|'_')*;
number = (digits '.'? digits?|'.'digits) exp?;
misc = [0..0xffff];
```