Alternation

Small finite sets are conveniently represented by listing their elements. Parentheses delimit expressions, and |, the *alternation operator*, separates alternatives.

For example, D, the set of the ten single digits, is defined as

D = (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9).

The characters (,), ' , ∗, + , and | are *meta- characters* (punctuation and regular expression operators).

Meta- characters must be quoted when used as ordinary characters to avoid ambiguity.

For example the expression

`( '(' | ')' | ; | , )`

defines four single character tokens (left parenthesis, right parenthesis, semicolon and comma). The parentheses are quoted when they represent individual tokens and are not used as delimiters in a larger regular expression.

Alternation is extended to sets of strings:

Let P and Q be sets of strings.

Then string $s \in (P \mid Q)$ if and only if $s \in P$ or $s \in Q$.

For example, if LC is the set of lower- case letters and UC is the set of upper- case letters, then (LC | UC) is the set of all letters (in either case).

Kleene Closure

A useful operation is *Kleene closure* represented by a postfix ∗ operator.

Let P be a set of strings. Then P^* represents all strings formed by the catenation of zero or more selections (possibly repeated) from P.

Zero selections are denoted by λ.

For example, LC^* is the set of all words composed of lower- case letters, of any length (including the zero length word, λ).

Precisely stated, a string $s \in P^*$ if and only if s can be broken into zero or more pieces: $s = s_1 s_2 \ldots s_n$ so that each $s_i \in P$ ($n \geq 0$, $1 \leq i \leq n$).

We allow $n = 0$, so λ is always in P.

Definition of Regular Expressions

Using catenations, alternation and Kleene closure, we can define *regular expressions* as follows:

- ∅ is a regular expression denoting the empty set (the set containing no strings). ∅ is rarely used, but is included for completeness.
- λ is a regular expression denoting the set that contains only the empty string. This set is not the same as the empty set, because it contains one element.
- A string s is a regular expression denoting a set containing the single string s.

- If A and B are regular expressions, then A | B, A B, and A^* are also regular expressions, denoting the alternation, catenation, and Kleene closure of the corresponding regular sets.

Each regular expression denotes a set of strings (a *regular set*). Any finite set of strings can be represented by a regular expression of the form (s_1 | s_2 | … | s_k). Thus the reserved words of ANSI C can be defined as
(auto | break | case | …).

The following additional operations useful. They are not strictly necessary, because their effect can be obtained using alternation, catenation, Kleene closure:

- P^+ denotes all strings consisting of *one* or more strings in P catenated together:
  $P^* = (P^+ | \lambda)$ and $P^+ = P\,P^*$.
  For example, $(0 | 1)^+$ is the set of all strings containing one or more bits.
- If A is a set of characters, Not(A) denotes (Σ – A); that is, all *characters* in Σ *not* included in A. Since Not(A) can never be larger than Σ and Σ is finite, Not(A) must also be finite, and is therefore regular. Not(A) does not contain λ since λ is not a character (it is a zero- length string).

For example, Not(Eol) is the set of all characters excluding Eol (the end of line character, `'\n'` in Java or C).

- It is possible to extend Not to strings, rather than just Σ. That is, if S is a set of strings, we define $\overline{S}$ to be
  (Σ^* – S); the set of all strings except those in S. Though $\overline{S}$ is usually infinite, it is also regular if S is.
- If k is a constant, the set A^k represents all strings formed by catenating k (possibly different) strings from A.
  That is, A^k = (A A A …) (k copies).
  Thus $(0 | 1)^{32}$ is the set of all bit strings exactly 32 bits long.

Examples

Let D be the ten single digits and let L be the set of all 52 letters. Then

- A Java or C++ single- line comment that begins with // and ends with Eol can be defined as:
  Comment = // Not(Eol)^* Eol
- A fixed decimal literal (e.g., `12.345`) can be defined as:
  Lit = D^+. D^+
- An optionally signed integer literal can be defined as:
  IntLiteral = ('+' | – | λ) D^+

  (Why the quotes on the plus?)

- A comment delimited by ## markers, which allows single #'s within the comment body:
  Comment2 =
  ## ((# | λ) Not(#))* ##

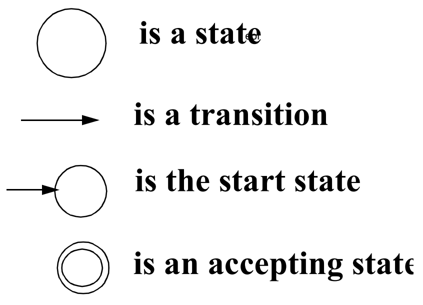
  All finite sets and many infinite sets are regular. But not all infinite sets are regular. Consider the set of balanced brackets of the form
  [[[...]]].
  This set is defined formally as
  $\{ [^m]^m \mid m \geq 1 \}$.
  This set is known *not* to be regular. Any regular expression that tries to define it either does not get *all* balanced nestings or it includes extra, unwanted strings.

Finite Automata and Scanners

A *finite automaton* (FA) can be used to recognize the tokens specified by a regular expression. FAs are simple, idealized computers that recognize strings belonging to regular sets. An FA consists of:

- A finite set of *states*
- A set of *transitions* (or *moves*) from one state to another, labeled with characters in Σ
- A special state called the *start* state
- A subset of the states called the *accepting*, or *final,* states

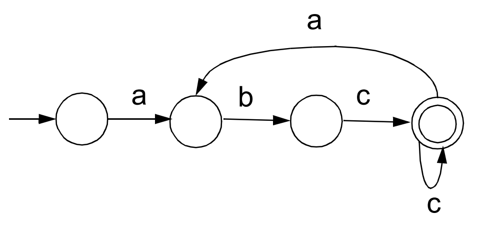
These four components of a finite automaton are often represented graphically:



Finite automata (the plural of automaton is automata) are represented graphically using *transition diagrams.* We start at the start state. If the next input character matches the label on

a transition from the current state, we go to the state it points to. If no move is possible, we stop. If we finish in an accepting state, the sequence of characters read forms a *valid* token; otherwise, we have not seen a valid token.

In this diagram, the valid tokens are the strings described by the regular expression $(a\ b\ (c)^+\ )^+$.



Deterministic Finite Automata

As an abbreviation, a transition may be labeled with more than one character (for example, Not(c)). The transition may be taken if the current input character matches any of the characters labeling the transition.

If an FA always has a *unique* transition (for a given state and character), the FA is *deterministic* (that is, a deterministic FA, or DFA). Deterministic finite automata are easy to program and often drive a scanner.

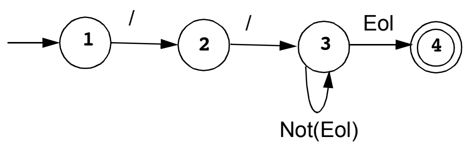
If there are transitions to more than one state for some character, then the FA is *nondeterministic* (that is, an NFA).

A DFA is conveniently represented in a computer by a *transition table.* A transition table, T, is a two dimensional array indexed by a DFA state and a vocabulary symbol.

Table entries are either a DFA state or an error flag (often represented as a blank table entry). If we are in state s, and read character c, then T[s,c] will be the next state we visit, or T[s,c] will contain an error marker indicating that c cannot extend the current token. For example, the regular expression

// Not(Eol)^* Eol

which defines a Java or C++ single- line comment, might be translated into


$$\begin{tikzcd}
\arrow[r] & 1 \arrow[r, "/"] & 2 \arrow[r, "/"] & 3 \arrow[r, "\text{Eol}"] \arrow[loop below, "\text{Not(Eol)}"] & \boxed{4}
\end{tikzcd}$$

The corresponding transition table is:

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

A complete transition table contains one column for each character. To save space, *table compression* may be used. Only non- error entries are explicitly represented in the table, using hashing, indirection or linked structures.

All regular expressions can be translated into DFAs that accept (as valid tokens) the strings defined by the regular expressions. This translation can be done manually by a programmer or automatically using a scanner generator.

A DFA can be coded in:

- Table- driven form
- Explicit control form

In the table- driven form, the transition table that defines a DFA's actions is explicitly represented in a run- time table that is "interpreted" by a driver program.

In the direct control form, the transition table that defines a DFA's actions appears implicitly as the control logic of the program.

For example, suppose **`CurrentChar`** is the current input character. End of file is represented by a special character value, **`eof`**. Using the DFA for the Java comments shown earlier, a table- driven scanner is:

```
State = StartState
while (true){
  if (CurrentChar == eof)
      break
  NextState =
      T[State][CurrentChar]
  if(NextState == error)
      break
  State = NextState
  read(CurrentChar)
}
if (State in AcceptingStates)
     // Process valid token
else // Signal a lexical error
```

This form of scanner is produced by a scanner generator; it is definition- independent. The scanner is a driver that can scan *any* token if T contains the appropriate transition table.

Here is an explicit- control scanner for the same comment definition:

```
 if (CurrentChar == '/'){
   read(CurrentChar)
   if (CurrentChar == '/')
     repeat
       read(CurrentChar)
     until (CurrentChar in
            {eol, eof})
   else //Signal lexical error
else // Signal lexical error
if (CurrentChar == eol)
  // Process valid token
else //Signal lexical error
```

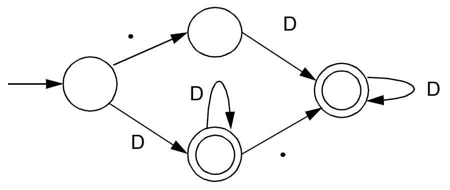
The token being scanned is “hardwired” into the logic of the code. The scanner is usually easy to read and often is more efficient, but is specific to a single token definition.

More Examples

- A FORTRAN- like real literal (which requires digits on either or both sides of a decimal point, or just a string of digits) can be defined as

  RealLit = $(D^+ (\lambda \mid \textbf{.})) \mid (D^* \textbf{.} D^+)$

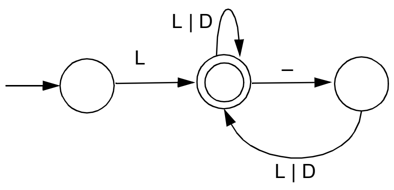
  This corresponds to the DFA



- An identifier consisting of letters, digits, and underscores, which begins with a letter and allows no adjacent or trailing underscores, may be defined as

  ID = L (L | D)* (_ (L | D)$^+$)*

  This definition includes identifiers like `sum` or `unit_cost`, but excludes `_one` and `two_` and `grand___total`. The DFA is:



Lex/Flex/JLex

Lex is a well- known Unix scanner generator. It builds a scanner, in C, from a set of regular expressions that define the tokens to be scanned.

Flex is a newer and faster version of Lex.

JLex is a Java version of Lex. It generates a scanner coded in Java, though its regular expression definitions are very close to those used by Lex and Flex.

Lex, Flex and JLex are largely *non- procedural*. You don't need to tell the tools *how* to scan. All you need to tell it *what* you want scanned (by giving it definitions of valid tokens).

This approach greatly simplifies building a scanner, since most of the details of scanning (I/O, buffering, character matching, etc.) are automatically handled.

JLex

JLex is coded in Java. To use it, you enter

```
java JLex.Main f.jlex
```

Your **`CLASSPATH`** should be set to search the directories where JLex's classes are stored.
(In build files we provide the **`CLASSPATH`** used will includ JLex's classes).

After JLex runs (assuming there are no errors in your token specifications), the Java source file **`f.jlex.java`** is created. (**`f`** stands for any file name you choose. Thus **`csx.jlex`** might hold token definitions for CSX, and **`csx.jlex.java`** would hold the generated scanner).

You compile **`f.jlex.java`** just like any Java program, using your favorite Java compiler.

After compilation, the class file **`Yylex.class`** is created.

It contains the methods:

- **`Token yylex()`** which is the actual scanner. The constructor for **`Yylex`** takes the file you want scanned, so **`new Yylex(System.in)`** will build a scanner that reads from **`System.in`**. **`Token`** is the token class you want returned by the scanner; you can tell JLex what class you want returned.
- **`String yytext()`** returns the character text matched by the last call to **`yylex`**.
-

Input to JLex

There are three sections, delimited by **`%%`**. The general structure is:

```
User Code
%%
Jlex Directives
%%
Regular Expression rules
```

The User Code section is Java source code to be copied into the generated Java source file. It contains utility classes or return type classes you need. Thus if you want to return a class **`IntlitToken`** (for integer literals that are scanned), you include its definition in the User Code section.

JLex directives are various instructions you can give JLex to customize the scanner you generate.

These are detailed in the JLex manual. The most important are:

- **`%{`**
  `Code copied into the Yylex class (extra fields or methods you may want)`
  `%}`
- **`%eof{`**
  `Java code to be executed when the end of file is reached`
  `%eof}`
- **`%type classname`**
  `classname` is the return type you want for the scanner method, **`yylex()`**

Macro Definitions

In section two you may also define *macros*, that are used in section three. A macro allows you to give a name to a regular expression or character class. This allows you to reuse definitions and make regular expression rule more readable.

Macro definitions are of the form

`name = def`

Macros are defined one per line.

Here are some simple examples:

```
Digit=[0-9]
AnyLet=[A-Za-z]
```

In section 3, you use a macro by placing its name within **`{`** and **`}`**. Thus **`{Digit}`** expands to the character class defining the digits 0 to 9.

Regular Expression Rules

The third section of the JLex input file is a series of token definition rules of the form

`RegExpr     {Java code}`

When a token matching the given **`RegExpr`** is matched, the corresponding Java code (enclosed in "{" and "}") is executed. JLex figures out what **`RegExpr`** applies; you need only say what the token looks like (using **`RegExpr`**) and what you want done when the token is matched (this is usually to return some token object, perhaps with some processing of the token text).

Here are some examples:

```
"+"    {return new Token(sym.Plus);}

(" ")+  {/* skip white space */}

{Digit}+ {return
 new IntToken(sym.Intlit,
 new Integer(yytext()).intValue());}
```

Regular Expressions in JLex

To define a token in JLex, the user to associates a regular expression with commands coded in Java.

When input characters that match a regular expression are read, the corresponding Java code is executed. As a user of JLex you don't need to tell it *how* to match tokens; you need only say *what* you want done when a particular token is matched.

Tokens like white space are deleted simply by having their associated command not return anything. Scanning continues until a command with a return in it is executed.

The simplest form of regular expression is a single string that matches exactly itself.

For example,

```
if    {return new Token(sym.If);}
```

If you wish, you can quote the string representing the reserved word (**`"if"`**), but since the string contains no delimiters or operators, quoting it is unnecessary.

For a regular expression operator, like +, quoting is necessary:

```
"+"    {return
          new Token(sym.Plus);}
```