Reading Assignment

Read Chapter 3 of
Crafting a Compiler.

Scanning

A scanner transforms a character stream into a token stream.

A scanner is sometimes called a *lexical analyzer* or *lexer.*

Scanners use a formal notation (*regular expressions*) to specify the precise structure of tokens.

But why bother? Aren't tokens very simple in structure?

Token structure can be more detailed and subtle than one might expect. Consider simple quoted strings in C, C++ or Java. The body of a string can be any sequence of characters *except* a quote character (which must be escaped). But is this simple definition really correct?

Can a newline character appear in a string? In C it cannot, unless it is escaped with a backslash.

C, C++ and Java allow escaped newlines in strings, Pascal forbids them entirely. Ada forbids *all* unprintable characters.

Are null strings (zero-length) allowed? In C, C++, Java and Ada they are, but Pascal forbids them.

(In Pascal a string is a packed array of characters, and zero length arrays are disallowed.)

A precise definition of tokens can ensure that lexical rules are clearly stated and properly enforced.

Regular Expressions

Regular expressions specify simple (possibly infinite) sets of strings. Regular expressions routinely specify the tokens used in programming languages.

Regular expressions can drive a *scanner generator*.

Regular expressions are widely used in computer utilities:

- The Unix utility *grep* uses regular expressions to define search patterns in files.
- Unix shells allow regular expressions in file lists for a command.

- Most editors provide a “context search” command that specifies desired matches using regular expressions.
- The Windows Find utility allows some regular expressions.

Regular Sets

The sets of strings defined by *regular expressions* are called *regular sets.*

When scanning, a token class will be a regular set, whose structure is defined by a regular expression.

Particular instances of a token class are sometimes called *lexemes,* though we will simply call a string in a token class an *instance* of that token. Thus we call the string `abc` an identifier if it matches the regular expression that defines valid identifier tokens.

Regular expressions use a finite character set, or *vocabulary* (denoted Σ).

This vocabulary is normally the character set used by a computer. Today, the *ASCII* character set, which contains a total of 128 characters, is very widely used.

Java uses the *Unicode* character set which includes all the ASCII characters as well as a wide variety of other characters.

An empty or *null* string is allowed (denoted λ, “lambda”). Lambda represents an empty buffer in which no characters have yet been matched. It also represents optional parts of tokens. An integer literal may begin with a plus or minus, or it may begin with λ if it is unsigned.

Catenation

Strings are built from characters in the character set Σ via *catenation*.

As characters are catenated to a string, it grows in length. The string `do` is built by first catenating `d` to λ, and then catenating `o` to the string `d`. The null string, when catenated with any string s, yields s. That is, $s\,\lambda \equiv \lambda\, s \equiv s$. Catenating λ to a string is like adding 0 to an integer—nothing changes.

Catenation is extended to sets of strings:

Let P and Q be sets of strings. (The symbol ∈ represents set membership.) If $s_1 \in P$ and $s_2 \in Q$ then string $s_1 s_2 \in (P\ Q)$.

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 $s_i \in P$ $(n \ge 0, 1 \le i \le n)$.

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 $\{ [^m]^m \mid m \ge 1 \}$.
  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*:*

○ is a state

→ is a transition

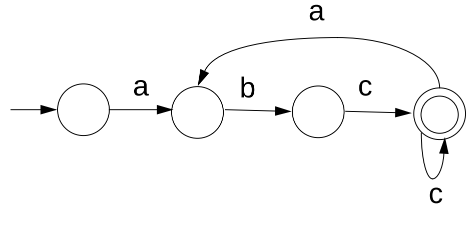
→○ is the start state

◎ is an accepting state

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.

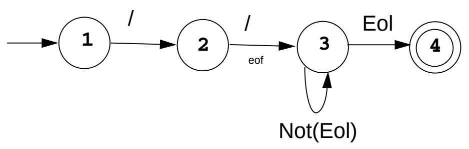
If there are transitions to more than one state for some character, then the FA is *nondeterministic* (that is, an NFA).

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



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
```

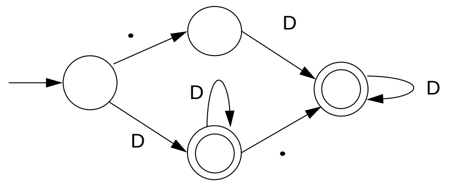
The token being scanned is “hardwired” into the logic of the code. The scanner is usually easy to read and often is more efficient, but is specific to a single token definition.

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 = $(D^+ (\lambda \mid \mathbf{.})) \mid (D^* \,\mathbf{.}\, D^+)$

  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: