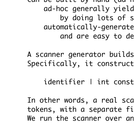
Notes for CSC 2/454, Sept. 2, 9, and 14, 2020

==========
SCANNING

Scanner is responsible for
    tokenizing source
    removing comments
    saving text of identifiers, numbers, strings
    saving source locations (file, line, column) for error messages

a DFA for identifiers:



Can be built by hand (ad hoc) or automatically from regular expressions (REs)
    ad-hoc generally yields the fastest, most compact code
        by doing lots of special-purpose things
    automatically-generated scanners can come close, though
        and are easy to develop and change

A scanner generator builds a DFA automatically from a set of REs
Specifically, it constructs a machine that accepts the "language"

    identifier | int const | real const | comment | symbol | ...

In other words, a real scanner accepts the alternation of a language's
tokens, with a separate final state for each.
We run the scanner over and over to get one token after another.

(Note that theoreticians use "language" to mean a set of strings --
not nec. the valid programs of a programming language.)

Nearly universal rule:
    always take the longest possible token from the input
    thus foobar is foobar and never f or foo or foob
    more to the point, 3.14159 is a real const and never 3, ., and 14159

An RE generates a regular language; a DFA _recognizes_ it.

The standard Unix lex (flex) outputs C code.
Some other tools produce numeric tables that are read by a separate driver.
The table is the transition function
    two-dimensional array indexed by current state and input character
    entries specify
        next state
        whether to keep scanning, return a token, or announce an error

Longest-possible token rule means we return only when the next
character can't be used to continue the current token.
That next character must generally be saved for the next token.

In some cases you may need to peek at more than one character of
lookahead in order to know whether to proceed.

    In Pascal, when you had a 3 and you a saw a dot, did you proceed (in
    hopes of getting 3.14) or did you stop (in fear of getting 3..5)?

    In messier cases, you may not be able to get by with any fixed amount of
    lookahead. In Fortran IV (c. 1962), for example, one had

        DO 5 I = 1,25   loop
        DO 5 I = 1.25   assignment
        DO 5,I = 1,25   alternate syntax for loop, f77

For most languages it suffices to remember we were in a potentially
final state, and save enough information that we can back up to it if we
get stuck later.
For some languages (famously, Fortran), that isn't enough.
    Sometimes need semantic information in order to scan (yuck).

Building a scanner from regular expressions

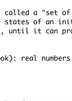
multi-step process
(1) write REs by hand, including for whitespace and comments, but with
        identifiers and reserve words (keywords) combined
(2) build NFA from REs
(3) build DFA from NFA
(4) minimize DFA
(5) add extra logic to
        implement the longest-possible-token rule, with backup
        discard white space and comments (i.e., start over when you
            realize that's what you found)
        distinguish reserve words from identifiers
        save text of "interesting" tokens
        tag returned tokens with location and text
        return an extra $$ token at end-of-file

Step (2) is inductive. It starts with a trivial DFA to accept a single
character:

Note that this machine has a single start state, a single final state,
no transitions into the start state, and no transitions out of the final
state. We'll maintain these invariants in three inductive steps:

Concatenation (A B):

Alternation (A | B):



Kleene closure (A*):

Step (3) uses what's called a "set of subsets" construction.
Step (4) divides the states of an initial DFA into progressively finer
equivalences classes, until it can prove that additional refinement
makes no difference.

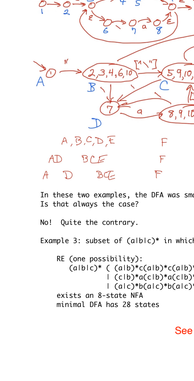
Example 1 (in the book): real numbers (no exponential notation)

    RN = d*(.d|d.)d*

    14-state NFA results from construction
    5-state subset DFA
    4-state minimal DFA

Example 2: character strings with optional backslash-escaped quotes

    S = " ([^\"] | \a)* "   "a" for anything



In these two examples, the DFA was smaller than the original NFA.
Is that always the case?

No! Quite the contrary.

Example 3: subset of (a|b|c)* in which some letter appears at least 3 times.

    RE (one possibility):
        (a|b|c)* ((a|b)*c(a|b)*c(a|b)*c(a|b)*
                 | (c|b)*a(c|b)*a(c|b)*a(c|b)*
                 | (a|c)*b(a|c)*b(a|c)*b(a|c)*) (a|b|c)*
    exists an 8-state NFA
    minimal DFA has 28 states

See the PLP Companion Site, Sec. 2.4.1

That NFA, of course, doesn't come from the standard construction.
Are there any that _do_, and for which the DFA is bigger yet?
Absolutely!

Example 4: subset of (0|1|2|3|4|5|6|7|8|9)* in which some digit appears
at least 10 times.

    minimal DFA has 10,000,000,001 states
    one RE is
    (0|1|2|3|4|5|6|7|8|9)*
    (
        (0|1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0
      | (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0
      | (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0
      | (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)*)
    |
        (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1
      | (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1
      | (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1
      | (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)*)
    | ...
    |
        (0|1|2|3|4|5|6|7|8|9)* 9 (0|1|2|3|4|5|6|7|8)* 9
      | (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9
      | (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9
      | (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9
      | (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)*)
)
    (0|1|2|3|4|5|6|7|8|9)*

    anybody believe the automatically constructed NFA for that has 10
    billion states?

Recall that real scanner accepts alternation of tokens, with a separate
final state for each. Scanner generator
  - starts with NFAs for all the separate tokens
  - creates a new start state with an ε-transition to the start state of
    each token NFA
  - turns that into a DFA
  - runs the minimization algorithm starting not with two classes (final
    and non-final) but with K+1: non-final, final for token T1, final for
    token T2, ..., final for token TK.

Try Exercise 2.5 in the book!

In most compilers, the parser drives the front end.
The scanner is a subroutine (function) called by the parser.

We can also build a RE from a DFA (as shown in Section 2.4.1 on the PLP
companion site). This completes a proof that the two notations are
equally powerful. Nobody does this in practice, however; it amounts to
converting the computer-friendly notation into a human-friendly
notation, and we usually want to go the other way.

In a real compiler, the input may have errors, including lexical errors.
    Consider, in Java or C, the input

        int myVaria$ble

    That dollar sign isn't supposed to be there.

What should a scanner do?
Generally suffices to
    return the longest token that starts at the beginning of the
        remaining part of the input program
        (after deleting white space and comments)
    or, if there isn't a valid token starting there, delete characters
        until there is, print an error message reporting the deletion,
        and return the token found

Pseudocode for a table-driver scanner can be found in the text.