CSCI-C311 Programming Languages

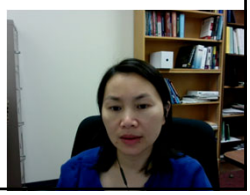
Scanning

Dr. Hang Dinh

1

Outline and Reading

- After this lecture, you will learn
  - How scanning (lexical analysis) works
  - How to convert NFA to DFA
- Reading
  - Scott 4e - Section 2.2



2

Scanning

- Recall: scanner and parser are responsible for discovering the syntactic structure of a program.
- Scanner is responsible for
  - tokenizing source (i.e., grouping input characters into tokens)
  - removing comments
  - saving text of identifiers, numbers, strings
  - saving source locations (file, line, column) for error messages
  - processing *pragmas* (*significant comments*) which provide directives/hints to the compiler.

3

Scanning Example: Calculator Language

- Tokens for a "*calculator language*" with input, output, variables and assignment are defined by the following grammar rules:

$assign \rightarrow :=$
$plus \rightarrow +$
$minus \rightarrow -$
$times \rightarrow *$
$div \rightarrow /$
$lparen \rightarrow ($
$rparen \rightarrow)$
$id \rightarrow letter\ (letter \mid digit)^*$
except for **`read`** and **`write`**
$number \rightarrow digit\ digit^* \mid digit^*\ (.\ digit \mid digit\ .)\ digit^*$

4

Scanning Example: Calculator Language

- Tokens for comment in *"calculator language"* :

$$comment \rightarrow /* \ (nonstar \mid * \ nonslash)* \ *^{*}/$$
$$\mid // \ (nonnewline)* \ newline$$

Kleene star

Character '$*$'

where

- $nonstar$ denotes any character other the $*$
- $nonslash$ denotes any character other the $/$
- $nonnewline$ denotes any character other the $newline$

5

Ad hoc Approach to Scanning

- An ad hoc scanner can work as follows:
  - Check the simpler and more common cases first
  - peek ahead when we need to
  - embed loops for comments and for long tokens such as identifiers and numbers.
- After finding a token, the scanner returns to the parser
  - When invoked again, it repeats the ad hoc algorithm from the beginning, using the next available characters of input
- Accept the **longest possible token** in each invocation of the scanner
  - E.g., **3.14** is a real number and never **3**, **.**, and **14**
  - White space is ignored, except to the extent that it separates tokens

6

Ad hoc Scanner for Calculator Language

- Recall rules defining tokens:
  - $assign \rightarrow :=$
  - $plus \rightarrow +$
  - $minus \rightarrow -$
  - $times \rightarrow *$
  - $lparen \rightarrow ($
  - r$paren \rightarrow)$

Pseudocode part 1 of 3

skip any initial white space (spaces, tabs, and newlines)
if cur_char ∈ {'(', ')', '+', '−', '*'}
    return the corresponding single-character token
if cur_char = ':'
    read the next character
    if it is '=' then return *assign* else announce an error

7

Ad hoc Scanner for Calculator Language

- Recall rules defining tokens:
  - $div \rightarrow /$
  - $comment \rightarrow /* \ (nonstar \mid * \ nonslash)* \ *^* */$
    $\mid // \ (nonnewline)* \ newline$

Pseudocode part 2 of 3

if cur_char = '/'
    peek at the next character
    if it is '*' or '/'
        read additional characters until "*/" or *newline* is seen, respectively
        jump back to top of code
    else return *div*

8

Ad hoc Scanner for Calculator Language

- Recall rules defining tokens:

$id \rightarrow letter\ (letter \mid digit)^*$

except for **`read`** and **`write`**

$number \rightarrow digit\ digit^*$

$\mid digit^*\ (.\ digit \mid digit\ .)\ digit^*$

Pseudocode part 3 of 3

```
if cur_char = .
    read the next character
    if it is a digit
        read any additional digits
        return number
    else announce an error
if cur_char is a digit
    read any additional digits and at most one decimal point
    return number
if cur_char is a letter
    read any additional letters and digits
    check to see whether the resulting string is read or write
    if so then return the corresponding token
    else return id
else announce an error
```

9

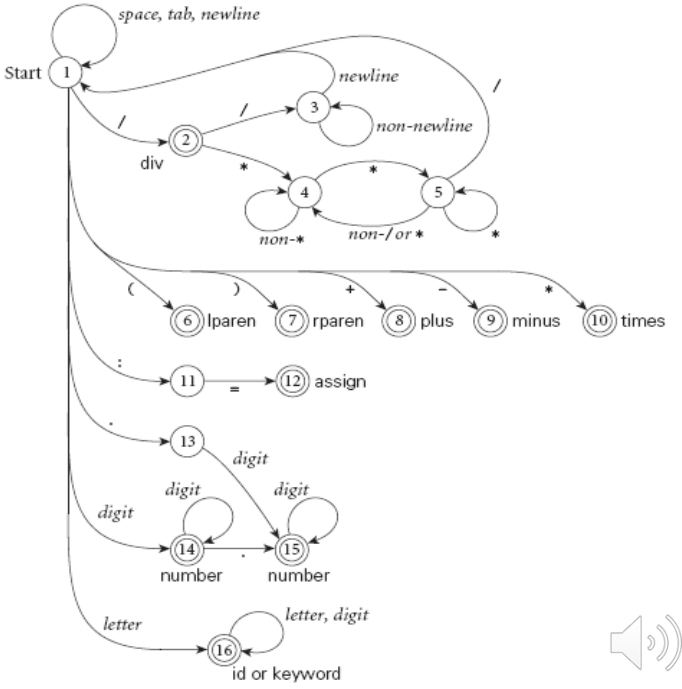
DFA Approach to Scanning

- Production compilers often use ad hoc scanners
  - The code is fast and compact
- During language development, it's preferable to build a scanner as an explicit representation of a deterministic finite automaton (DFA).
  - Regular expressions "generate" a regular language; DFAs "recognize" it
  - We run the machine over and over to get one token after another
  - Apply the **longest possible token** rule: the scanner returns to the parser only when the next character cannot be used to continue the current token

10

DFA for Calculator Language

- Pictorial representation of a scanner for calculator tokens, in the form of a DFA
- Hand-written DFA
  - Parallels the pseudocode of the ad hoc scanner



11

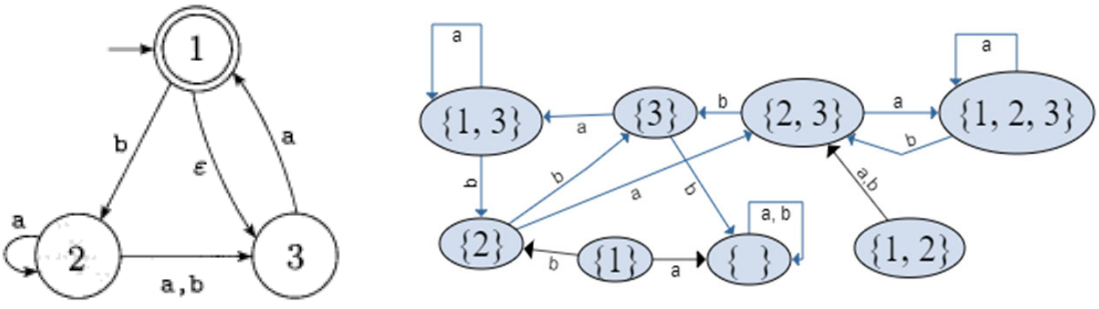
Systematic DFA Approach to Scanning

- Finite automata can be generated from a set of regular expressions
  - making it easy to regenerate a scanner when token definitions change.
  - It's done by ***scanner generator*** tools, such as Unix's **`lex/flex`**
- Automatically generate a DFA from a regular expression
  1. Convert the regular expression into a nondeterministic finite automaton (NFA)
  2. Convert the NFA into an equivalent DFA
  3. Space optimization: generate a final DFA with the minimum possible number of states.

12

Convert NFA to DFA

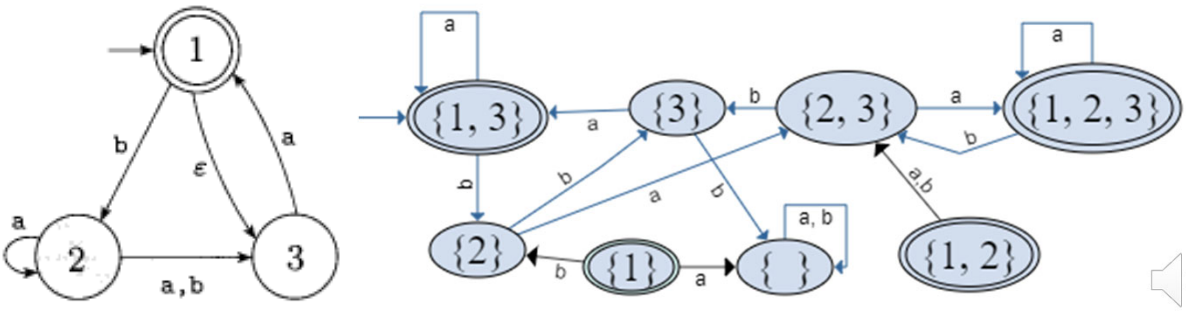
- Use the "set of subsets" construction to transform an NFA into DFA
  - Each state of the DFA is a subset of the NFA's set of states
  - Each subset X of the NFA's set of states makes transition on a given input symbol to the set of states that the NFA can reach on the given input from a state in X.



13

Convert NFA to DFA

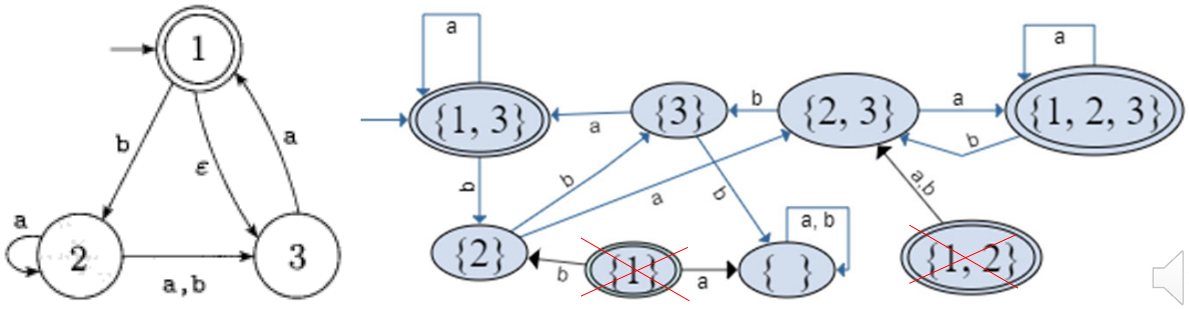
- Use the "set of subsets" construction to transform an NFA into DFA
  - The start state of the DFA is the set of states the NFA could be in before reading any input (those that can be reached from NFA's start state via ε-arrows)
  - A final state of the DFA is any set of states containing a final state of the NFA



14

Convert NFA to DFA

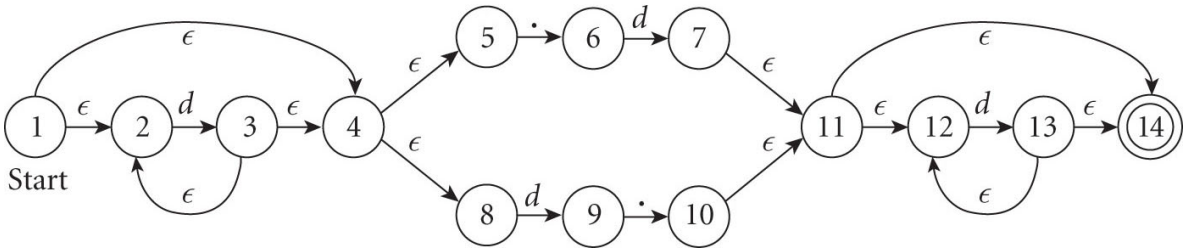
- Use the "set of subsets" construction to transform an NFA into DFA
  - The number of states in the DFA is exponential in the number of states in the NFA (if the NFA has n states, then the DFA has 2^n states)
  - We can remove states in the DFA that are unreachable from the DFA's start state
    - We can start with the DFA's start state, then keep adding only states that are reachable from existing states.



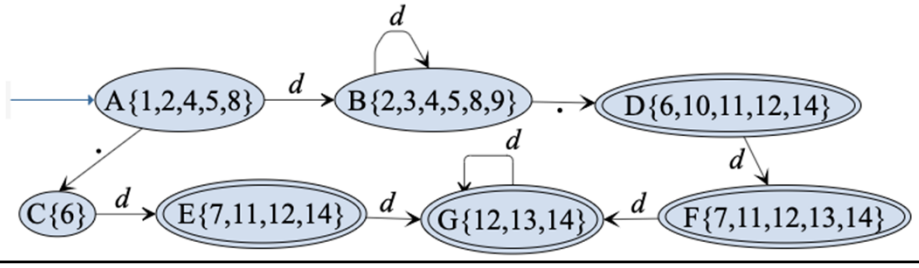
15

Convert NFA to DFA

- NFA for the regular expression: $d^*(.\ d \mid d\ .)\ d^*$



- DFA:



16

Representing DFA

- Two approaches to representing a DFA
  - Use control flow – a loop (`gotos`) and nested `case` (`switch`) statements
  - Use data structure -- a two-dimensional state transition table
- Handwritten DFA tend to use nested `case` statements
- Most automatically generated DFAs use tables
- Table-driven scanner is a driver program that uses two tables
  - One is the state transition table of the DFA
  - A 2nd table indicates, for each state, whether it's a final state, and if so, which token → if we pass such a state and hit an error state we can back up.

20

Nested `case` Statement Style of DFA

- General structure:

```
state := 1     //start state
loop
   read cur_char
   case state of
      1: case cur_char of
            ' ', '\t', '\n':…
            'a' …'z' : ….
             …
      2: case cur_char of
            …
            …
      n: case cur_char of
```

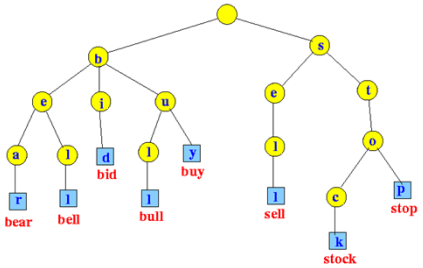
Set a new state or return from the scanner with the current token. (If the current character should not be part of that token, it is pushed back onto the input stream before returning.)

Need to modify this code to
- Handle keywords
- Peek ahead when a token can validly be extended by 2 or more additional characters

21

Handling Reserved Words

- Can write a DFA that distinguishes between keywords and identifiers
  - But it requires a *lot* of states!
- Most scanners treat keywords as "exception" to rules for identifiers
  - Before returning an identifier to the parser, the scanner looks it up in a *hash table* or *trie* (prefix tree) to make sure it isn't really a keyword.



Courtesy image from
http://www.mathcs.emory.edu/~cheung/Courses/253/Syllabus/Text/trie01.html

22

Handling "Longest Possible Token" Rule

- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
  - In Pascal/C, for example, when you have a **3** and you a see a dot **.**

    do you proceed (in hopes of getting **3.14**)?

    or

    do you stop (in fear of getting **3..5**)?
- In messier cases, a scanner may need to look an arbitrary distance ahead.
- We need 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

23

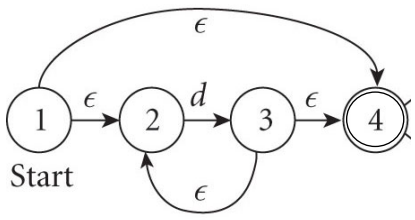
Lexical Errors

- Occur when the next input character may be
  - Neither an acceptable continuation of the current token
  - Nor the start of another token
- Lexical errors are relatively rare
- Handling lexical errors
  1. Throw away the current, invalid token
  2. Skip forward until a character is found than can legitimate begin a new token
  3. Restart the scanning algorithm
  4. Count on the error-recovery mechanism of the parser to cope with any cases in which the resulting sequence of tokens is not syntactically valid.

24

Practice for the Quiz

- Convert the following NFA to DFA using the "set of subsets" construction:



25