Lexical Analysis

- what tokens occur in source text
- considerations
  - fixed length tokens (e.g. `=`, `<=`, `.`)
  - variable length tokens (e.g. words, literals)
    - delimiters
  - separators (e.g. spaces, comments, *eol*)
    - treat *eol* as character
  - end of input
    - serves as delimiter
    - end of program must occur at *eof*
    - treat *eof* as token
- intermediate code

Lexical Errors

- incorrectly formed tokens
- characters which don't begin tokens
- error messages
  - source listing
- multi-pass compilers & error messages
  - error table
  - error tokens

Input Filter

- `SourceReader` class
- Decorator (filter) design pattern
- transforms input chars into desired form
  - *eol*
  - *eof*
- line numbers & character positions
- source listing
- single symbol lookahead
  - e.g. for < vs <=
  - line handling

Token Representation

- tokens are text so maybe `String`
- tokens have a position in the input
  - line and character
- `Token` class
  - string representation
  - position
  - kind
    - e.g. identifier
    - `TokenKind` class

Scanner Output

- usually interested only in the presence (or absence) of token
- sometimes interested in other info as well, e.g.
  - numeric literal: value
  - identifier: actual name
- scanner locates tokens
  - token kind
  - string representation
  - other info
- single pass compiler
  - parser can access `Token` as needed
- multi-pass compiler
  - produces a file of `Token`

Scanning

- `TokenReader`
- single character lookahead
  - variable length tokens
  - tokens beginning with same character
  - scanner one character ahead:
    - `SourceReader` reads first character
    - current character: `getCurChar`
    - always read next character: `nextChar`
- basic algorithm
  - look at next (current) character & determine token kind
  - invoke routine to scan that kind

- separator chars
  - may precede any token
  - must be skipped
- comments
  - are separators (but not single chars)
  - if begin with unique char can be handled while skipping separators
  - if not must be handled while scanning tokens introduced by that char (use recursion or loop to restart scan after comment)
  - must not pass *eof* when scanning comment (i.e. *eof* is a comment delimiter)

Scanning Routines

- simple tokens
  - fixed length, unique 1st char (e.g. &)
  - fixed length, non-unique 1st character (e.g. > and >=)
- variable length tokens
  - identifiers & reserved words
  - numeric literals
  - string literals
  - words symbols
    - keywords
    - identifiers
    - boolean literals
  - match longest conforming substring, (e.g. number) – *eol*, *eof*
- poorly formed tokens
  - string & char literals – *eol*, *eof*

Main Class

- `Compiler`
- opens files
  - source
  - listing
  - code
- for testing
  - repeatedly gets next token until EOF

Testing

- error free compiler
- systematically developed test sets
- correct test results determined before testing
- choice of test output
  - enough to determine if successful
  - indicate where the source of the error is
  - compact enough to allow verification
  - display `Token`
    - indicates correctness
    - infinite loop or abort

- verify all statements
  - each branch of conditional
  - each loop at least once through
  - for scanner
    - each token input at least once
    - test both ends of ranges (e.g. `'0'` & `'9'` for numbers)
    - if case sensitive, test both cases
- test searches (keywords)
  - both identifiers & reserved words
  - upper and lower case
- test error detection and recovery
- test cases documented and saved