## Reading Assignment

```
Fischer & LeBlanc

        Chapter  3

        Omit     3.4.1
```



## Lexical Analysis (scanning)

- Lexical analysis transforms a sequence of *characters* from one or more source files into a stream of language-specific *lexical tokens*.
- Lexical Analysis Tasks
  - Delete irrelevant information (whitespace, comments)
  - Determine lexical token boundaries in source stream
  - Transmit lexical tokens to next pass.
  - Deal with ill-formed lexical tokens, recover from lexical errors.
  - Process literal constants (optional).
  - Transmit source coordinates (file, line number) to next pass.



## Lexical Analysis

- Typical objects a lexical analyzer must deal with.
  - Identifiers
  - Reserved words or Keywords.
  - Literal constants: integer, real, string ...
  - Special characters: `+ − * / ( ) &` ...
  - Comments
- A programming language *should* be designed so that the lexical analyzer can determine lexical token boundaries easily.
- **Lexical analysis must be really efficient for good compiler performance.**
- In most languages, lexical tokens are designed to be easily separated based a left to right scan of the input stream.



## Token Separation

- Identifiers which could also be reserved words require special handling. Some possible solutions:
  - Table lookup using fast binary search or hash table.
  - Languages with keywords (e.g. Fortran, PL/I ) defer distinguishing identifiers and keywords to the syntax analysis pass.
  - Use some convention to distinguish identifiers from reserved words, e.g. if vs. **IF**.
- Some situations require look ahead in the lexical analyzer. Often one character, but sometimes (much) more. Examples:
  - Multiple character special tokens: `*` vs. `**`
  - Distinguishing integers and reals: `12345` vs. `12345E+3`
  - Distinguishing comments: `/ 100` vs. `/*`
  - String conventions, e.g. `'AB''CD'`
  - Bad language design, e.g.
    `'00110011'` vs. `'00110011'B` in PL/I
    `DO 10 I = 7` vs. `DO 10 I = 7 , 12` in Fortran

## Typical Lexical Analyzer

```
while( ! eof ) {
    switch( currentChar ) {
    case letter:        accumulate identifier
                        distinguish identifer
                           or reserved word
    case digit:         accumulate number
                        distinguish integer, float
    case quote:         accumulate string
    case commentStart:  discard comment
    case special:       accumulate special token
    case whiteSpace:    discard white space
    }
    if( tokenProduced )
         emit token
}
```



## Regular Expressions

- Regular expression notation is often a convenient way to precisely describe the lexical tokens in a language.
- The sets of strings defined by regular expressions are *regular sets*. Each regular expression defines a regular set.
- Let $V$ (*Vocabulary*) be a finite set of characters. Tokens are formed from the characters in $V$.
- $\lambda$ is the empty or null string.
- Concatenation is used to build strings from the characters in $V$.
- Characters may be quoted in single quotes **'** to avoid ambiguity. Mandatory for the meta characters: '(' ')' '|' '+' '*'



## Regular Expression Definition

1. $\emptyset$ is a regular expression denoting the empty set.
2. $\lambda$ is a regular expression denoting the set that contains only the empty string. Note $\emptyset \neq \lambda$.
3. A string of S of characters from $V$ is a regular expression denoting a set containing only S.
4. If A and B are regular expressions then so are:

| | |
|---|---|
| A \| B | Alternation |
| A B | Concatenation |
| $A^*$ | Kleene closure, zero or more copies of A |
| ( A ) | Parenthesis |

5. Useful Sets:

| | |
|---|---|
| $A^+$ | *One* or more strings in A |
| Not( A ) | Complement of A with respect to $V$ |
| $A^k$ | Concatenation of exactly k copies of strings from A |



## Regular Expression Examples

| | |
|---|---|
| ( 0 \| 1 \| 2 \| 3 \| 4 \| 5 \| 6 \| 7 \| 8 \| 9 ) | Digit |
| ( A \| B \| **...** \| X \| Y \| Z ) | Letter |
| ( Digit \| A \| B \| C \| D \| E \| F ) | Hex Digit |
| Digit* | Integer |
| ( '+' \| '-' \| $\lambda$ ) | OptSign |
| OptSign Integer | Signed integer |
| Letter ( Digit \| Letter \| '_' )* | Identifier |
| Digit$^+$ '.' Digit$^*$ ( 'E' OptSign Digit$^+$ \| $\lambda$ ) | Real Number |

## Finite Automata

- Finite automata (state machines) can be used to recognize tokens specified by a regular expression.
- A finite automaton (FA) is
  - A finite set of states
  - A set of transitions from one state to another.
    Transitions depend on input characters from the vocabulary $V$
  - A distinguished *start* state.
  - A set of final (*accepting*) states.
- Finite automata can be represented graphically using state transition diagrams. See Slide 53 for an example.
- If the FA has a *unique* transition for every ( state, input character ) pair then it is a *deterministic* FA (DFA).



## Building Tables for Efficiency

- To save effort, the lexical analyzer can build tables of certain kinds of tokens, e.g. identifiers, string constants, other constants.
- These tables are saved on disk or in memory for use by later passes, typically for error messages and code generation.
- The token that gets passed on the the syntax analysis pass contains a field that identifies the type of token and the tokens index into the appropriate table.
- Indices that uniquely identify tokens are sufficient for most syntax and semantic analysis.



## What Makes Lexical Analysis Difficult

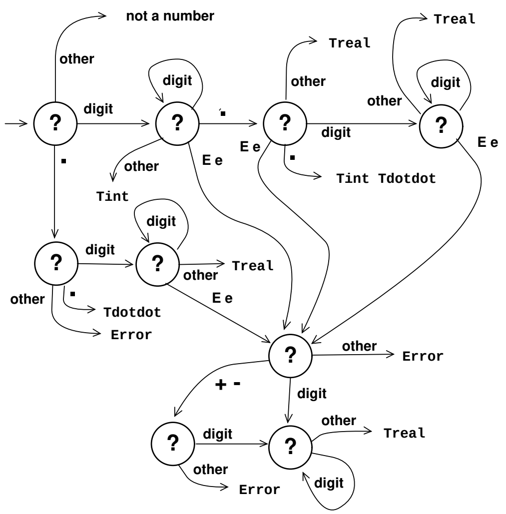
- Dealing with long tokens, e.g. 32K strings in PL/I
- Splitting tokens across input boundaries.
- Special handling in character strings. e.g. `\nnn` in C
- Context sensitive input, e.g. continuation cards.
- Processing numeric values close to the hardware limits for minimum and maximum values.
- Separation requiring lookahead.
  More than one character lookahead is usually painful.



## Example - Real Constants in Turing

- Reference Manual definition - literal real constant:
  - An *explicitRealConstant* consists of an optional plus or minus sign a significant digits part and an exponent part.
  - The significant digits part **...** consists of a sequence of one or more digits optionally containing a decimal point. The decimal point is allowed to follow the last digit as in 16. or precede the first digit as in .25
  - The exponent part consists of the letter `e` or `E` followed optionally by a plus or minus sign followed by one or more digits. If the significant figures [sic] part contains a decimal point then the exponent part is not required.
- Issues
  - Must have at least one digit in the significant digits part. Must have at least one digit in the exponent part.
  - Context sensitive to distinguish integer constants from real constants.
  - Context sensitive: exponent can be omitted *iff* there was a decimal point in the significant digits part.
  - Leading/trailing decimal point interacts with use of **..** for integer subrange. e.g. `3.4` vs. `3..4`
  - Handle leading sign as unary + or − during syntax analysis. Lexical analyzer can't easily distinguish `X - 3.5` from `X - -3.5` or `X - - -3.5` .
  - Must detect malformed constants like `3.4.5E+`
  - Arbitrary number of digits in exponent part is unrealistic. Exponent range is -38 .. + 38 on 32-bit machines. But must allow `1.0E+000000000000000003`

## Recognizer for Turing Numeric Constants






## Lexical Analyzer Optimization

- First design a *correct* lexical analyzer.
  *Then optimize it for speed.*
- Most important optimizations[a] :
  - Input in bulk, not character by character.
  - Touch each input character as few times as possible.
  - Don't copy input if you can avoid it.
  - Discard white space and comments efficiently.
    Often 50% .. 60% of the source.
  - Make reserved word testing as fast as possible.
  - Evaluate literal constants efficiently.
- Optimizations should not make lexical analyzer difficult to maintain or modify.

[a] See: W. M. Waite, The Cost of Lexical Analysis, Software–Practice and Experience, v.16, n.5, May 1986



## Example - Accumulating Identifiers

- Naive

```
char ch, iBuffer[200]  ;
int i ;
. . .
i = 0 ;
while( isLetter(ch) || isDigit(ch) || ch == '_') {
   iBuffer[i++] = ch ;
   ch = getchar() ;
}
/* identifier is in iBuffer[ 0 .. i - 1 ]  */
```

- Optimized

```
BOOL identifierChar[ 256 ] ;
char * iStart, * iEnd, *inPtr ;
. . .
iStart = inPtr ;
while( identifierChar[ *inPtr++ ] ) ;
iEnd = inPtr ;
/* Identifier is in input buffer iStart .. iEnd - 1  */
```



## Example - Table Driven Scanner

- Language with whitespace, integers, identifiers, strings , special characters, C style comments
- Make lexical analysis decisions with 1 character look ahead.
- Advance in input, save start of long tokens.
- Save input pointer on entry to each state.
  If appropriate emit accumulated token on return to state 1
- Notation:
  N - change to state N
  +N - advance input pointer and change to state N

## Table Driven Scanner

| State | Input Character blank | digit | letter | quote | Special | / | * | Purpose |
|---|---|---|---|---|---|---|---|---|
| 1 | +1 | 2 | 3 | +4 | 5 | +6 | 5 | blanks & dispatch |
| 2 | 1 | +2 | 1 | 1 | 1 | 1 | 1 | integers |
| 3 | 1 | +3 | +3 | 1 | 1 | 1 | 1 | identifiers |
| 4 | +4 | +4 | +4 | +1 | +4 | +4 | +4 | 'string' |
| 5 | 1 | 1 | 1 | 1 | +1 | 1 | +1 | special chars. |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | +7 | emit token '/' |
| 7 | +7 | +7 | +7 | +7 | +7 | +7 | +8 | comment body |
| 8 | +7 | +7 | +7 | +7 | +7 | +1 | 7 | end comment |



## Lexical Errors

- Lexical analyzer must deal with ill-formed and incomplete lexical tokens.
- Examples:
  - Incorrectly formed: 3.4.5 3.4E- 0x7FFG8
  - Oversize tokens: 11111111111111111111111111111111111111111111111111
    Supercalifragalisticexpialadocious
  - Numeric value out of range: 1.23456E+99 99999999999999999
  - Unterminated string or comment: "Hi There /* Obfuscate here
  - Illegal characters: ? #
- Error handling strategies:
  - Emit error message for lexical error
  - Repair strategies:
    Correct token if possible.
    Discard bad token or replace with "safe" value
    Stop run away strings, comments on line boundaries.



## Handling Inclusion

- Often lexical analyzer must implement source inclusion,
  e.g. **#include** in C or **include** in OOT.
- Processing of include directive
  - Search sequence of directories for the file to include.
  - If file is found, emit file name and path to output.
  - Switch lexical analyzer input source to the file.
  - Perform lexical analysis on the file, emiting tokens as needed.
  - At end of file switch lexical analyzer input source back to previous source (after the include directive).
- This process must be recursive to deal with nested includes.
  Pass <file name, line number> coordinates on to later passes.
- Production compilers optimize include handling to minimize file opening and input effort.



## Managing Source Program Coordinates

- The subsequent phases of the compiler need to know where each lexical token originated if there are to emit source code specific error messages.
  e.g., `Syntax Error on line 23 of file foo.c`
- Conceptually tag each lexical token with an identification of the source file and line number where it originated. Really ambitious systems include offset in the line.
- In practice, this makes lexical tokens too large and unwieldy.
  Most practical systems are based on transmitting only *changes* in source coordinates (as invisible internal lexical tokens) to the subsequent phases.
  Each phase must then include logic to keep track of the current source program coordinate.

## Source Coordinate Management Example

- Send a *new source file token* at
  - The start of compilation.
  - The start of processing an included file (e.g. #include)
    The lexical analyzer may get this information from a pre processor.
  - At the end of processing an included file (e.g. revert to previous file)
- Send a *new source line token* at
  - The start of compilation.
  - Each time the lexical analyzer moves to a new line containing a lexical token (optimizes whitespace deletion).
  - At the start of processing an included file.
  - At the end of processing an included file to revert to the previous files coordinate system.



## Scanner Generators - Lex & Flex

- Lex and Flex (Flexible Lex) are automated *scanner generators* that transform a regular expression description of a language's lexical tokens into source code for a scanner.
- The generated scanners are adequate for student projects and prototype compilers but are not efficient enough for production compilers.
- Lex and Flex use an old-fashioned interface that depends on shared global variables. This makes it difficult to include multiple instances of Lex or Flex in one system.
- A regular expression defines a class of lexical token.
  The compiler writer can specify an arbitrary piece of C code that gets executed when the token is recognized.
- Use Lex/Flex in cases where production quality efficiency isn't required and efficient generation of the lexical analyzer is important.



food for slide eater



food for slide eater