Lexical Analysis (Tokenizing)

COMP 3002
School of Computer Science
List of Acronyms

- RE - regular expression
- FSM - finite state machine
- NFA - non-deterministic finite automata
- DFA - deterministic finite automata
The Structure of a Compiler

- program text
- syntactic analyzer
  - tokenizer
  - parser
- intermediate representation
- code generator
- machine code
Purpose of Lexical Analysis

• Converts a character stream into a token stream

```c
int main(void) {
    for (int i = 0;
         i < 10;
         i++) {
        ... tokenizer
```
How the Tokenizer is Used

• Usually the tokenizer is used by the parser, which calls the `getNextToken()` function when it wants another token

• Often the tokenizer also includes a `pushBack()` function for putting the token back (so it can be read again)
Other Tokenizing Jobs

- Input reading and buffering
- Macro expansion (C's #define)
- File inclusion (C's #include)
- Stripping out comments
Tokens, Patterns, and Lexemes

- A token is a pair
  - token name (e.g., VARIABLE)
  - token value (e.g., "myCounter")

- A lexeme is a sequence of program characters that form a token
  - (e.g., "myCounter")

- A pattern is a description of the form that the lexemes of a token may take
  - e.g., character strings including A-Z, a-z, 0-9, and _
A History Lesson

- Usually tokens are easy to recognize even without any context, but not always
- A tough example from Fortran 90:

```
DO 5 I = 1.25
<variable, "D05I"> <assign> <number,"1.25">
```

```
DO 5 I = 1,25
<do> <number, "5"> <variable, "I">  
<assign> <number, "1"> <comma> <number, "25">
```
Lexical Errors

• Sometimes the current prefix of the input stream does not match any pattern
  – This is an error and should be logged

• The lexical analyzer may try to continue by
  – deleting characters until the input matches a pattern
  – deleting the first input character
  – adding an input character
  – replacing the first input character
  – transposing the first two input characters
Exercise

• Circle the lexemes in the following programs

```java
public static void main(String args[]) {
    System.println("Hello World!");
}

float max(float a, float b) {
    return a > b ? a : b;
}
```
**Input Buffering**

- Lexemes can be long and the `pushBack` function requires a mechanism for pushing them back.
- One possible mechanism (suggested in the textbook) is a double buffer.
- When we run off the end of one buffer we load the next buffer.

```java
class Buffer {
    public static void start(int index) {
        current = index;
        // ...}

    return (23); 
}
```
Tokenizing (so far)

- What a tokenizer does
  - reads character input and turns it into tokens
- What a token is
  - a token name and a value (usually the lexeme)
- How to read input
  - use a double buffer if some lookahead is necessary
- How does the tokenizer recognize tokens?
- How do we specify patterns?
Where to Next?

- We need a formal mechanism for defining the patterns that define tokens
- This mechanism is formal language theory
- Using formal language theory we can make tokenizers without writing any actual code
Strings and Languages

• An alphabet $\Sigma$ is a set of symbols
• A string $S$ over an alphabet $\Sigma$ is a finite sequence of symbols in $\Sigma$
• The empty string, denoted $\varepsilon$, is a string of length 0
• A language $L$ over $\Sigma$ is a countable set of strings over $\Sigma$
Examples of Languages

• The empty language $L = \emptyset$
• The language $L = \{\varepsilon\}$ containing only the empty string
• The set $L$ of all syntactically correct C programs
• The set $L$ of all valid variable names in Java
• The set $L$ of all grammatically correct English sentences
**String Concatenation**

- If $x$ and $y$ are strings then the *concatenation* of $x$ and $y$, denoted $xy$, is the string formed by appending $y$ to $x$

- Example
  - $x = "dog"
  - $y = "house"
  - $xy = "doghouse"

- If we treat concatenation as a "product" then we get *exponentiation*:
  - $x^2 = "dogdog"
  - $x^3 = "dogdogdog"
Operations on Languages

- We can form complex languages from simple ones using various operations

- Union: \( L \cup M \) (also denoted \( L \mid M \))
  - \( L \cup M = \{ s : s \in L \text{ or } s \in M \} \)

- Concatenation
  - \( LM = \{ st : s \in L \text{ and } t \in M \} \)

- Kleene Closure \( L^* \)
  - \( L^* = \{ L^i : i = 0, 1, 2, \ldots \} \)

- Positive Closure \( L^+ \)
  - \( L^* = \{ L^i : i = 1, 2, 3, \ldots \} \)
Some Example

• $L = \{ A, B, C, \ldots Z, a, b, c, \ldots z \}$
• $D = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
• $L \cup D$
• $LD$
• $L^4$
• $L^*$
• $L(L \cup D)^*$
• $D^+$
Regular Expressions

- Regular expressions provide a notation for defining languages
- A regular expression $r$ denotes a language $L(r)$ over a finite alphabet $\Sigma$
- Basics:
  - $\varepsilon$ is a RE and $L(\varepsilon) = \{ \varepsilon \}$
  - For each symbol $a$ in $\Sigma$, $a$ is a RE and $L(a) = \{ a \}$
Regular Expression Operators

- Suppose $r$ and $s$ are regular expressions
- Union (choice)
  - $(r)|(s)$ denotes $L(r) \cup L(s)$
- Concatenation
  - $(r)(s)$ denotes $L(r)L(s)$
- Kleene Closure
  - $r^*$ denotes $(L(r))^*$
- Parenthesization
  - $(r)$ denote $L(r)$
  - Used to enforce specific order of operations
Order of Operations in REs

• To avoid too many parentheses, we adopt the following conventions
  – The * operator has the highest level of precedence and is left associative
  – Concatenation has second highest precedence and is left associative
  – The | operator has lowest precedence and is left associative
Binary Examples

• For the alphabet $\Sigma = \{ a, b \}$
  – $a|b$ denotes the language $\{ a, b \}$
  – $(a|b)(a|b)$ denotes the language $\{ aa, ab, ba, bb \}$
  – $a^*$ denotes $\{ \epsilon, a, aa, aaa, aaaa, ... \}$
  – $(a|b)^*$ denotes all possible strings over $\Sigma$
  – $a|a^*b$ denotes the language $\{ a, b, ab, aab, aaab, ... \}$
**Regular Definitions**

- REs can quickly become complicated
- *Regular definitions* are multiline regular expressions
- Each line can refer to any of the preceding lines *but not to itself or to subsequent lines*

```plaintext
letter_ = A|B|...|Z|a|b|...|z|_

digit   = 0|1|2|3|4|5|6|7|8|9

id      = letter_(letter_ |digit)*
```
Regular Definition Example

- Floating point number example
  - Accepts 42, 42.314159, 42.314159E+23, 42E+23, 42E23, ...

digit \equiv 0|1|2|3|4|5|6|7|8|9

digits \equiv digit \, digit^*

optionalFraction \equiv . \, digits \mid \epsilon

optionalExponent \equiv (E \, (+|-|\epsilon) \, digits) \mid \epsilon

number \equiv digits \, optionalFraction \, optionalExponent
Exercises

• Write regular definitions for
  – All strings of lowercase letters that contain the five vowels in order
  – All strings of lowercase letters in which the letters are in ascending lexicographic order
  – Comments, consisting of a string surrounded by /* and */ without any intervening */
Extension of Regular Expressions

- There are also several time-saving extensions of REs
  - One or more instances
    - \( r^+ = rr^* \)
  - Zero or one instance
    - \( r? = r | \epsilon \)
- Character classes
  - \([abcdef] = (a | b | c | d | e | f)\)
  - \([A-Za-z] = (A | B | C | ... | Y | Z | a | b | c | ... | y | z)\)
- Others
  - See page 127 of the text for more common RE shorthands
Some Examples

digit   = [0-9]
digits  = digit+
number  = digits (. digits)? (E[+-]? digits)?

letter_ = [A-Za-z_]
digit   = [0-9]
variable= letter_ (letter|digit)*
Recognizing Tokens

• We now have a notation for patterns that define tokens
• We want to make these into a tokenizer
• For this, we use the formalism of finite state machines
An FSM for Relational Operators

- relational operators <, >, <=, >=, ==, <>
**FSM for Variable Names**

- `letter_ = [A-Za-z_]`
- `digit   = [0-9]`
- `variable= letter_ (letter|digit)*`
FSM for Numbers

• Build the FSM for the following:

\[
\begin{align*}
\text{digit} & = [0-9] \\
\text{digits} & = \text{digit}^+ \\
\text{number} & = \text{digits} (. \text{digits})? ((E|e) \text{digits})? \\
\end{align*}
\]
**NumReader.java**

- Look at NumReader.java example
  - Implements a token recognizer using a switch statement
The Story So Far

• We can write tokens types as regular expressions
• We want to convert these REs into (deterministic) finite automata (DFAs)
• From the DFA we can generate code
  – A single while loop containing a large switch statement
    • Each state in \( S \) becomes a case
  – A table mapping \( S \times \Sigma \rightarrow S \)
    • (current state, next symbol) \( \rightarrow \) (new state)
  – A hash table mapping \( S \times \Sigma \rightarrow S \)
  – Elements of \( \Sigma \) may be grouped into character classes
NumReader2.java

- Look at NumReader2.java example
  - Implements a tokenizer using a hashtable
Automatic Tokenizer Generators

- Generating FSMs by hand from regular expressions is tedious and error-prone
- Ditto for generating code from FSMs
- Luckily, it can be done automatically

Regular expressions \(\xrightarrow{\text{lex}}\) NFA \(\xrightarrow{\text{NFA2DFA}}\) tokenizer
Non-Deterministic Finite Automata

- An NFA is a finite state machine whose edges are labelled with subsets of $\Sigma$
- Some edges may be labelled with $\epsilon$
- The same labels may appear on two or more outgoing edges at a vertex
- An NFA accepts a string $s$ if $s$ defines any path to any of its accepting states
NFA Example

- NFA that accepts apple or ape
**NFA Example**

- NFA that accepts any binary string whose 4 last value is 1
From Regular Expression to NFA

- Going from a RE to a NFA with one accepting state is easy
- \( \epsilon \)
- \( a \)
Union

- $r | s$

 FSM for $r$

 FSM for $s$

start
Concatenation

- rs

![Diagram showing two FSMs for r and s connected at the start state](image-url)
Kleene Closure

- $r^*$

![Finite State Machine (FSM) Diagram for Kleene Closure]

start

FSM for r

$\varepsilon$

$\varepsilon$

$\varepsilon$
NFA to DFA

• So far
  – We can express token patterns as RE
  – We can convert REs to NFA

• NFAs are hard to use
  – Given an NFA $F$ and a string $s$, it is difficult to test if $F$ accepts $s$

• Instead, we first convert the NFA into a deterministic finite automaton
  – No $\varepsilon$ transitions
  – No repeated labels on outgoing edges
Converting an NFA into a DFA

• Converting an NFA into a DFA is easy but sometimes expensive
• Suppose the NFA has $n$ states $1,\ldots,n$
• Each state of the DFA is labelled with one of the $2^n$ subsets of $\{1,\ldots,n\}$
• The DFA will be in a state whose label contains $i$ if the NFA could be in state $i$
• Any DFA state that contains an accepting state of the NFA is also an accepting state
NFA 2 DFA - Sketch of Algorithm

- Step 1 - Remove duplicate edge labels by using ε transitions
NFA 2 DFA

• Step 2: Starting at state 0, start expanding states
  – State \( i \) expands into every state reachable from \( i \) using only \( \varepsilon \)-transitions
  – Create new states, as necessary for the neighbours of already-expanded states
  – Use a lookup table to make sure that each possible state (subset of \( \{1,\ldots,n\} \)) is created only once
Example

• Convert this NFA into a DFA
Example

• Convert this NFA into a DFA
From REs to a Tokenizer

• We can convert from RE to NFA to DFA
• DFAs are easy to implement
  – Using a switch statement or a (hash)table
• For each token type we write a RE
• The lexical analysis generator then creates a NFA (or DFA) for each token type and combines them into one big NFA
From REs to a Tokenizer

- One giant NFA captures all token types
- Convert this to a DFA
  - If any state of the DFA contains an accepting state for more than 1 token then something is wrong with the language specification
Summary

• The Tokenizer converts the input character stream into a token stream
• Tokens can be specified using REs
• A software tool can be used to convert the list of REs into a tokenizer
  – Convert each RE to an NFA
  – Combine all NFAs into one big NFA
  – Convert this NFA into a DFA and the code that implements this DFA
**Other Notes**

- REs, NFAs, and DFAs are equivalent in terms of the languages they can define.
- Converting from NFA to DFA can be expensive:
  - An $n$-state NFA can result in a $2^n$ state DFA.
- None of these are powerful enough to parse programming languages but are usually good enough for tokens:
  - Example: the language $\{ a^n b^n : n = 1, 2, 3, \ldots \}$ is not recognizable by a DFA (why?)