Lexical Analysis

Chapter 1, Section 1.2.1
Chapter 3, Section 3.1, 3.3, 3.4, 3.5
JFlex Manual
Inside the Compiler: Front End

- **Lexical analyzer (aka scanner)**
  - Converts ASCII or Unicode to a stream of tokens
  - Provides input to the syntax analyzer (aka parser), which creates a parse tree from the token stream
  - Usually the parser calls the scanner: `getNextToken()`

- **Other scanner functionality**
  - Removes comments: e.g. /* ... */ and // ...
  - Removes whitespaces: e.g., space, newline, tab
  - May add identifiers to the symbol table
  - May maintain information about source positions (e.g., file name, line number, character number) to allow more meaningful error messages
Basic Definitions

• **Token**: token name and optional attribute value
  – E.g. token name `if`, no attribute: the `if` keyword
  – E.g. token name `relop` (relational operator), attribute in `{ LT, LE, EQ, NE, GT, GE }`: represents `<, <=, =, <>, >, >=`
  – The token name is an abstract symbol that is a terminal symbol for the grammar in the parser

• Each token has a **pattern**: e.g. `id` has the pattern “letter followed by zero or more letters or digits”

• **Lexeme**: a sequence of input characters (ASCII or Unicode) that is an instance of the token: e.g. the characters `getPrice` for token `id`
Typical Categories of Tokens

• One token per reserved keyword; no attribute
• One token per operator or per operator group
• One token id for all identifiers; the attribute is a pointer to an entry in the symbol table
  – Names of variables, functions, user-defined types, ...
  – Symbol table has lexeme & source position where seen
• One token for each type of constant; attribute is the actual constant value
  – E.g. (int_const,5) or (string_const,"Alice")
• One token per punctuation symbol; no attribute
  – E.g. left_parenthesis, comma, semicolon
Specifying Patterns for Lexemes

• Standard terminology
  – **Alphabet**: a finite set of symbols (e.g., ASCII or Unicode)
  – **String** over an alphabet: a finite sequence of symbols
  – **Language**: countable set of strings over an alphabet

• Operations on languages
  – **Union**: \( L \cup M = \) all strings in \( L \) or in \( M \)
  – **Concatenation**: \( LM = \) all \( ab \) where \( a \) in \( L \) and \( b \) in \( M \)
  – \( L^0 = \{ \varepsilon \} \) and \( L^i = L^{i-1}L \)
  – **Closure**: \( L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \); **positive closure**: w/o \( L^0 \)

• Regular expressions: notation to express languages constructed with the help of such operations
Regular Expressions (1/2)

• Given some alphabet, a regular expression is
  – The empty string \( \varepsilon \); any symbol from the alphabet
  – \( r|s, rs, r^* \), and \( (r) \) where \( r \) and \( s \) are regular expressions
  – \( ^* \) has higher precedence than concatenation, which has higher precedence than \( | \); all are left-associative

• Each regular expression \( r \) defines a language \( L(r) \)
  – \( L(\varepsilon) = \{ \varepsilon \} \); \( L(a) = \{ a \} \) for alphabet symbol \( a \); \( L(r|s) = L(r) \cup L(s) \); \( L(rs) = L(r)L(s) \); \( L(r^*) = (L(r))^* \); \( L((r)) = L(r) \)

• Extended notation
  – \( r^+ = rr^* \); \( ^+ \) has same precedence & associativity as \( ^* \)
  – \( r? = r|\varepsilon \); \( ? \) has same precedence & associativity as \( */^+ \)
  – \([a-zA-Z_] \) = \( a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z \mid _\)
Regular Expressions (2/2)

• For convenience, give names to expressions

• Example: identifiers in C

\[
\text{letter}_- \rightarrow A | B | ... | Z | a | b | ... | z | _ \\
\text{digit} \rightarrow 0 | 1 | ... | 9 \\
\text{id} \rightarrow \text{letter}_- ( \text{letter}_- | \text{digit} )^* \\
\]

• Example: unsigned numbers (integers or floating point) – optional fraction and exponent

\[
\text{digit} \rightarrow [0-9] \\
\text{digits} \rightarrow \text{digit}^* \\
\text{number} \rightarrow \text{digits} ( . \text{digits} )? ( E [+-]? \text{digits} )? \\
\]
Recognition of Tokens

- Finite automata (FA)
  - Automata that recognize strings defined by a regular expression
  - States, input symbols, transitions, start state, set of final states
  - Transitions between states occur on specific input symbols
  - Deterministic automata: only 1 transition per state on a specific input; no transitions on the empty string

RE for integers: 
\[ [0-9]^+ \]
Languages and Automata

• Language **recognized** by an automaton: the set of strings it **accepts**
  
  – Starting in the start state; transitions upon to the input symbols in the input string; finish in a final state

• Each pattern for a token gets its own FA

```
9
start

1 <

2 =

3 >

4 = return(relop,LE)

5 > return(relop,NE)

6 other backtrack; return(relop,LT)

7 = return(relop,GE)

8 > other backtrack; return(relop,GT)
```
Keywords, Identifiers, Numbers

• FA for the keyword **then**

• FA for id
  – Letter or _, followed by letter, digit, or _
  – Ends with something other than letter, digit, _
  – Backtrack

• FA for simple integers
  – One or more digits; end with non-digit; backtrack

• FA for floating point numbers, with exponents, etc.
  – see Fig 3.16
Implementing a Scanner

• The naïve approach:
  – Define an order in which to try each FA (e.g., first try the FA for each individual keyword, then the FA for id, then for numbers, etc.)

• For each FA: keep a state variable; big `switch` stmt

```java
state = 0;
while (true) {
    switch (state) {
        case 0: c = nextChar();
            if (c == '<') state = 1; else if (c == '=') state = 2;
            else if (c == '>') state = 3; else fail(); break;
        case 1: ... state = ...; break;
        ... }
```
The Easy Way

• Do the code generation automatically, using a generator of lexical analyzers
  – High-level description of regular expressions and corresponding actions
  – Automatic generation of transition tables, etc.
  – Sophisticated lexical analysis techniques – better that what you can hope to achieve manually

• C world: lex and flex; Java world: JLex and JFlex

• Can be used to generate
  – Standalone scanners
  – Scanners integrated with automatically-generated parsers (using yacc, bison, CUP, etc.)
Simple JFlex Example

[Assignment: get it from the course web page under “Resources” and run it – today!]

• Standalone text substitution scanner
  – Reads a name after the keyword name
  – Substitutes all occurrences of "hello" with "hello <name>!"

Everything above %% is copied in the resulting Java class (e.g., Java import, package, comments)

%%

%public The generated Java class should be public
%class Subst The generated Java class will be called Subst.java
%standalone Create a main method; no parser; unmatched text printed
%unicode Capable of handling Unicode input text (not only ASCII)
%

{ String name; Code copied verbatim into the generated Java class
%

%% Start rules and actions
"name " [a-zA-Z]+ Reg expr Returns the lexeme as String
[Hh] "ello"

{ name = yytext().substring(5); } { System.out.print(yytext() +" " +name +"!"); }
Rules (Regular Expressions) and Actions

• The scanner picks a regular expressions that matches the input and runs the action

• If several regular expressions match, the one with the longest lexeme is chosen
  – E.g., if one rule matches the keyword `break` and another rule matches the id `breaking`, the id wins

• If there are several “longest” matches, the one appearing earlier in the specification is chosen

• The action typically will create a new token for the matched lexeme
Regular Expressions in JFlex

- **Character**
  - Except meta characters `| ( ) { } [ ] < > \ . * + ? ^ $ / . " ~ !`

- **Escape sequence**
  - `\n \r \t \f \b \x3F` (hex ASCII) `\u2BA7` (hex Unicode)
  - **Warning**: `\n` is not end of line, but ASCII LF (line feed); use `\r | \n | \r\n` to match end of line

- **Character classes**
  - `[a0-3\n]` is `{a,0,1,2,3,\n}`; `[^a0-3\n]` is any char not in set
  - Predefined classes: e.g. `[:letter:]`, `[:digit:]`, `.` (all except `\n`)

- " ... " matches the exact text in double quotes
  - All meta characters but `\` and " lose their special meaning inside a string
Regular Expressions in JFlex

• \{ MacroName \}
  – A macro can be defined earlier, in the second part of the specification: e.g., LineTerminator = \r | \n | \r\n
  – In the third part, it can be used with \{LineTerminator\}

• Operations on regular expressions
  – a|b, ab, a*, a+, a?, !a, ~a, a{n}, a{n,m}, (a), ^a, a$, a/…,

• End of file: <<EOF>>

• **Assignment**: http://jflex.de/manual.html
  – Read “Lexical Specifications”, subsection “Lexical rules”
  – Read “A Simple Example: How to work with JFlex”
Lexical States

• Definition and use
  – In part 2: `%state STRING`
  – In part 3: `<STRING>` expr { action } or
    `<STRING>` { expr1 { action1 } expr2 { action2 } ... }

• Lexical states can be used to refine a specification
  – E.g. if the scanner is in STRING, only expressions that
    are preceded by `<STRING>` can be matched
  – A regular expression can depend on more than one
    state; matched if the scanner is in any of these states
  – YYINITIAL is predefined and is the starting state
  – If an expression has no state, it is matched in all states

• See example in online documentation
Interoperability with CUP (1/2)

• CUP is a parser generator; grammar given in x.cup

• Terminal symbols in grammar are encoded in a CUP-generated `sym.java`

```java
public class sym {
    public static final int MINUS = 4;
    public static final int NUMBER = 9; ...
}
```

• The CUP-generated parser (in `parser.java`) calls a method `next_token` on the scanner and expects to get back an object of `java_cup.runtime.Symbol`
  – A Symbol contains a token type (from `sym.java`) and optionally an Object with an attribute value, plus source location (start & end position)
Interoperability with CUP (2/2)

• Inside the lexical specification
  – import java_cup.runtime.Symbol;
  – Add %cup in part 2
  – Return instances of Symbol
    
    "-"                { return new Symbol(sym.MINUS); }  
    
    {IntConst}  { return new Symbol(sym.NUMBER,  
                  new Integer(Integer.parseInt(yytext()))})

• Workflow
  – Run JFlex to get Lexer.java
  – Run CUP to get sym.java and parser.java
  – Main.java: new parser(new Lexer(new FileReader(...)));  
  – Compile everything (e.g. javac Main.java)
  – Assignment: read & run calc example on the web page, under “Resources”
Project 1

- **simpleC** on web page: a tiny scanner and parser for a subset of C (the parser is fake – no real parsing)
- Project: extend the functionality to handle
  - All TODO comments in the specification file
  - Any keywords, operators, etc. needed to handle two small C programs (already preprocessed with `cpp`)
- **Do not** change `MyLexer.java` (a driver program) or `MySymbol.java` (helper class, extension of Symbol)
  - The output from `MyLexer` will be used for grading
- **Assignment**: start working on it today!
Constructing JFlex-like tools

• Well-known and investigated algorithms for
  – Generating non-deterministic finite automata (NFA) from regular expressions (Sect. 3.7.4)
  – “Running” a NFA on a given string (Sect. 3.7.2)
  – Generating deterministic finite automata (DFA) from NFA (Sect. 3.7.1)
  – Generating DFA from regular expressions (Sect. 3.9.5)
  – Optimizing DFA to reduce number of states (Sect. 3.9.6)
  – We will not cover these algorithms in this class

• Building an actual tool
  – Compile the spec (e.g., z.flex) to transition tables for a single NFA (new start node, $\varepsilon$-transitions to all NFA)
  – Run the NFA (or an equivalent DFA) on the input