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()`

• Possible 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, column number) to allow more meaningful error messages
Basic Definitions

• **Token**: token name and optional attribute value
  – Token name `if`, no attribute: the `if` keyword
  – Token name `relop` (relational operator), attribute from `{ 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 is defined by a **pattern**: e.g., token `id` is defined by the pattern “letter followed by zero or more letters or digits”

• **Lexeme**: a sequence of input characters (ASCII or Unicode) that matches the pattern
  – the characters `getPrice` matches token `id`
Typical Categories of Tokens

• One token per reserved **keyword**; no attribute

• One token per **operator** or per **operator group**
  – E.g., **relop** for the group of all relational operators

• 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 Tokens

• Formal languages: basis for the design and implementation of programming languages

• **Alphabet**: finite set $T$ of symbols

• **String**: finite sequence of symbols
  – Empty string $\varepsilon$: sequence of length zero
  – $T^*$ - set of all strings over $T$ (incl. $\varepsilon$)
  – $T^+$ - set of all non-empty strings over $T$

• **Language**: set of strings $L \subseteq T^*$

• **Regular expressions**: notation to express regular languages
  – Traditionally used to specify the token patterns
General Formal Grammars

- **G = (N, T, S, P)**
  - Finite set of **non-terminal symbols** \( N \)
  - Finite set of **terminal symbols** \( T \)
  - Starting non-terminal symbol \( S \in N \)
  - Finite set of **productions** \( P \)
  - Describes a language \( L \subseteq T^* \)

- **Production:** \( x \rightarrow y \)
  - \( x \) is a non-empty sequence of terminals and non-terminals
  - \( y \) is a sequence of terminals and non-terminals

- **Applying a production:** \( uXv \Rightarrow uyw \)
Example: Non-negative Integers

• \( N = \{ I, D \} \)

• \( T = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \} \)

• \( S = I \)

• \( P = \{ \begin{align*} & I \rightarrow D, \\
& I \rightarrow DI, \\
& D \rightarrow 0, \\
& D \rightarrow 1, \\
& ..., \\
& D \rightarrow 9 \end{align*} \} \)
More Common Notation

I \rightarrow D \mid DI

- two production alternatives

D \rightarrow 0 \mid 1 \mid \ldots \mid 9

- ten production alternatives

- Terminals: 0 \ldots 9
- Starting non-terminal: I
  – Shown first in the list of productions
- Examples of production applications:
  
  \begin{align*}
  \text{step 1: } & I \Rightarrow DI \\
  \text{step 2: } & DI \Rightarrow DDI \\
  \text{step 3: } & DDI \Rightarrow D6I \\
  \text{step 4: } & D6I \Rightarrow D6D \\
  \text{step 5: } & D6D \Rightarrow 36D \\
  \text{step 6: } & 36D \Rightarrow 361
  \end{align*}
Languages and Grammars

• String derivation
  – $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$; denoted $w_1 \Rightarrow^* w_n$
  – If $n>1$, non-empty derivation sequence: $w_1 \Rightarrow w_n$

• Language generated by a grammar
  – $L(G) = \{ w \in T^* \mid S \Rightarrow^+ w \}$

• Fundamental theoretical characterization:
  Chomsky hierarchy (Noam Chomsky, MIT)
  – Regular languages $\subset$ Context-free languages $\subset$ Context-sensitive languages $\subset$ Unrestricted languages
  – Regular languages in compilers: for **lexical analysis**
  – Context-free languages in compilers: for **syntax analysis** (i.e., parsing)
Regular Grammars

• **Regular grammars** generate regular languages
  – All productions are \( A \rightarrow wB \) and \( A \rightarrow w \)
    • \( A \) and \( B \) are non-terminals; \( w \) is a sequence of terminals
    • This is a right-regular grammar
  – Or all productions are \( A \rightarrow Bw \) and \( A \rightarrow w \)
    • Left-regular grammar

• Example: \( L = \{ a^n b \mid n > 0 \} \) is a regular language
  – \( S \rightarrow A b \) and \( A \rightarrow a \mid A a \)

• \( l \rightarrow D \mid D l \) and \( D \rightarrow 0 \mid 1 \mid ... \mid 9 \): is this a regular grammar? Is the language itself regular?
Regular Expressions

• Instead of regular grammars, we often use regular expressions to specify regular languages

• Operations on languages
  – **Union**: $L \cup M = \text{all strings in } L \text{ or in } M$
  – **Concatenation**: $LM = \text{all } ab \text{ where } a \text{ in } L \text{ and } b \text{ in } M$
  – $L^0 = \{ \varepsilon \}$ and $L^i = L^{i-1}L$
  – **Closure**: $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$
  – **Positive closure**: $L^+ = L^1 \cup L^2 \cup \ldots$

• Regular expressions: notation to express languages constructed with the help of such operations
  – Example: $(0|1|2|3|4|5|6|7|8|9)^+$
Regular Expressions

- Given some alphabet, a **regular expression** is
  - The empty string \( \varepsilon \)
  - Any symbol from the alphabet
  - If \( r \) and \( s \) are regular expressions, so are \( r|s, rs, r^*, r^+, r? \), and \( (r) \)
  - \( *+/? \) have higher precedence than concatenation, which has higher precedence than \( | \)
  - All are left-associative
Regular Expressions

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

• Example: what is the language defined by $0(x|X)(0|1|...|9|a|b|...|f|A|B|...|F)^+$
Specification of Regular Languages

• Equivalent formalisms
  – Regular grammars
  – Regular expressions
  – Nondeterministic finite automata (NFA)
  – Deterministic finite automata (DFA)

• In compilers:
  – Regular expressions are used to specify the token patterns
  – Finite automata are used inside lexical analyzers to recognize lexemes that match the patterns
Implementing a Lexical Analyzer

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

• Examples: **lex** and **flex** for C, **JLex** and **Jflex** for Java

• Can be used to generate
  – Standalone scanners (i.e., have a “main”)
  – 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>!"

```
%%
%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
[Hh] "ello"

Returns the lexeme as String

{ name = yytext().substring(5); }
{ System.out.print(yytext()+" "+name+"!"); }

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

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

• Escape sequence
  – \n \r \t \f \b \x3F (hex ASCII) \u2BA7 (hex Unicode)

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

• " ... " matches the exact text in double quotes
  – All meta characters except \ 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”
Interoperability with CUP (1/2)

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

• Terminal symbols of the grammar are encoded in a CUP-generated class `sym.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 (javac Main.java)
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
  - Test carefully with 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, ε-transitions to all NFA)
  – Run the NFA (or an equivalent DFA) on the input