REs for Keywords

- It is easy to define a RE that describes all keywords
  
  \[ \text{Key} = \text{`if'} | \text{`else'} | \text{`for'} | \text{`while'} | \text{`int'} | \ldots \]

- These can be split in groups if needed
  
  \[ \text{Keyword} = \text{`if'} | \text{`else'} | \text{`for'} | \ldots \]
  
  \[ \text{Type} = \text{`int'} | \text{`double'} | \text{`long'} | \ldots \]

- The choice depends on what the next component (i.e., the parser) would like to see

REs for Numbers

- Straightforward representation for integers
  
  \[ \text{digits} = \text{`0'} | \text{`1'} | \text{`2'} | \text{`3'} | \text{`4'} | \text{`5'} | \text{`6'} | \text{`7'} | \text{`8'} | \text{`9'} \]
  
  \[ \text{integer} = \text{digits}^* \]

- RE systems allow the use of `-' for ranges, sometimes with `[` and `]`
  
  \[ \text{digits} = \text{[0-9]+} \]

- Floating point numbers are much more complicated
  
  \[ \text{2.00, .12e-12, 312.00001E+12, 4, 3.141e-12} \]

- Here is one attempt
  
  \[ (\text{`+`} | \text{`-`} | \epsilon) (\text{digit}^* \text{`.`}^? | \text{digits}^* (\text{`e历e'}) (\text{`E历e'}) (\text{`+`} | \text{`-`} | \epsilon) (\text{digit}^+)) \]

- Note the difference between meta-character and language-characters
  
  `+' versus `+`, `-' versus `-`, `('` versus `{`, etc.

- Often books/documentations use different fonts for each level of language

RE for Identifiers

- Here is a typical description
  
  \[ \text{letter} = \text{a-z | A-Z} \]
  
  \[ \text{ident} = \text{letter} (\text{letter} | \text{digit} | \text{`_'}^*) \]

  - Starts with a letter
  
  - Has any number of letter or digit or `_' afterwards

- In C: \( \text{ident} = (\text{letter} | \text{`_'}^*) (\text{letter} | \text{digit} | \text{`_'}^*) \)
**RE for Phone Numbers**

- **Simple RE**
  - digit = 0-9
  - area = digit digit digit
  - exchange = digit digit digit
  - local = digit digit digit digit
  - phononenumber = '(' area ')' '?' exchange ('-'|' ') local

- The above describes the $10^{3+3+4}$ strings of the L(phononenumber) language

---

**Regular Expression Practice**

- Write regular expressions for
  - All strings over alphabet \{a,b,c\}
  - All strings over alphabet \{a,b,c\} that contain substring ‘abc’
  - All strings over alphabet \{a,b,c\} that consist of one or more a’s, followed by two b’s, followed by whatever sequence of a’s and c’s
  - All strings over alphabet \{a,b,c\} such that they contain at least one of substrings ‘abc’ or ‘cba’

---

**Regular Expression Practice**

- Write regular expressions for
  - All strings over alphabet \{a,b,c\}
  - All strings over alphabet \{a,b,c\} that contain substring ‘abc’
  - All strings over alphabet \{a,b,c\} that consist of one or more a’s, followed by two b’s, followed by whatever sequence of a’s and c’s
  - All strings over alphabet \{a,b,c\} such that they contain at least one of substrings ‘abc’ or ‘cba’

---

**Automaton Examples**

- This automaton accepts input ‘if’
Automaton Examples

- This automaton accepts strings that start with a 0, then have any number of 1’s, and end with a 0
- Note the natural correspondence between automata and REs: 01’0
- Question: can we represent all REs with simple automata?
- Answer: yes
- Therefore, if we write a piece of code that implements arbitrary automata, we have a piece of code that implements arbitrary REs, and we have a lexer!
  - Not _this_ simple, but close

Example REs and DFA

- Say we want to represent RE ‘a’b’c’d’*e’ with a DFA

Example REs and NFA

- ‘a’b’c’d’*e’: much simpler with a NFA
- With \( \epsilon \)-transitions, the automaton can ‘choose’ to skip ahead, non-deterministically

Example REs and NFA

- ‘a’+b’+c’+d’+e’: easy modification
- But now we have multiple choices for a given character at each state!
  - e.g., two ‘a’ arrows leaving n
Automaton vs. RE Practice

- Write REs for the following NFAs

![Diagrams of NFAs]

Automaton vs. RE Practice

- Write REs for the following NFAs

\[ a^+ b^* a \]
\[ a^* b^* (\epsilon | ab^*) \]
\[ a^* b^* (ab^* a | bab^+) \]

A Grammar for Expressions

<table>
<thead>
<tr>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr \rightarrow Expr \ Op \ Expr</td>
</tr>
<tr>
<td>Expr \rightarrow Number \</td>
</tr>
<tr>
<td>Identifier \rightarrow Letter \</td>
</tr>
<tr>
<td>Letter \rightarrow a-z</td>
</tr>
<tr>
<td>Op \rightarrow &quot;+&quot; \</td>
</tr>
<tr>
<td>Number \rightarrow Digit \ Number \</td>
</tr>
<tr>
<td>Digit \rightarrow 0 \</td>
</tr>
</tbody>
</table>

Derivations as Trees

- A convenient and natural way to represent a sequence of derivations is a syntactic tree or parse tree.

- Example: Expr \rightarrow Expr \ Op \ Expr \rightarrow Number \ Op \ Expr \rightarrow Digit \ Number \ Op \ Expr \rightarrow 3 \ Number \ Op \ Expr \rightarrow 34 \ Op \ Expr \rightarrow 34 \* \ Expr \rightarrow 34 \* \ Identifier \rightarrow 34 \* \ Letter \ Identifier \rightarrow 34 \* \ a \ Identifier \rightarrow 34 \* \ a \ Letter \rightarrow 34 \* \ ax
Derivations as Trees

- In the parser, derivations are implemented as trees
- Often, we draw trees without the full derivations
- Example:

```
Expr
  Expr
    Number
      34
  Op
    Identifier
      ax

Expr
  Expr
    Op
      Expr
```

Grammar Practice

Consider the CFG:

- \( S \rightarrow (L) \mid a \)
- \( L \rightarrow L, S \mid S \)

Draw parse trees for:

- \((a, a)\)
- \((a, ((a, a), (a, a)))\)
Grammar Practice

- Write a CFG for the language of well-formed parenthesized expressions
  - (), ((())), ()(), (()()), etc.: OK
  - ()(), )(, ((()), (((, etc.: not OK

P → () | PP | (P)

Grammar Practice

- Is the following grammar ambiguous?
  A → A “and” A | “not” A | “0” | “1”

Yes!