LEX1: Intro to Regexps

# Lexical Analysis

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

# Lexical Analysis

- Also called *scanning*, take input program *string* and convert into tokens
- Example:

```
double f = sqrt(-1);
```

```
T_DOUBLE      ("double")
T_IDENT    ("f")
T_OP       ("=")
T_IDENT    ("sqrt")
T_LPAREN   ("(")
T_OP       ("-")
T_INTCONSTANT ("1")
T_RPAREN   (")")
T_SEP   (";")
```

# Token Attributes

- Some tokens have attributes
  - T_IDENT "sqrt"
  - T_INTCONSTANT 1
- Other tokens do not
  - T_WHILE
- *Token*=T_IDENT, *Lexeme*="sqrt", *Pattern*
- Source code location for error reports

# Lexical errors

- What if user omits the space in “doublef”?
  - No lexical error, single token T_IDENT (“doublef”) is produced instead of sequence T_DOUBLE, T_IDENT(“f”)!
- Typically few lexical error types
  - E.g., illegal chars, opened string constants or comments that are not closed

# Lexical errors

- Lexical analysis should not disambiguate tokens,
  - e.g. unary op − versus binary op −
  - Use the same token T_MINUS for both
  - It's the job of the parser to disambiguate based on the context
- Language definition should not permit crazy long distance effects (e.g. Fortran)

| | |
|---|---|
| DO 5 I = 1,5 | T_DO T_INT(5) T_ID(I) … |
| DO 5 I = 1.5 | T_ID(DO5I) T_EQ … |

# Ad-hoc Scanners

# Implementing Lexers: Loop and switch scanners

- Ad hoc scanners
- Big nested switch/case statements
- Lots of getc()/ungetc() calls
  - Buffering; Sentinels for push-backs; streams
- Can be error-prone
- Changing or adding a keyword is problematic
- Have a look at an actual implementation of an ad-hoc scanner

# Implementing Lexers: Loop and switch scanners

- Another problem: how to show that the implementation actually captures all tokens specified by the language definition?
- How can we show correctness
- Key idea: separate the definition of tokens from the implementation
- Problem: we need to reason about patterns and how they can be used to define tokens (recognize strings).

# Specification of Patterns using Regular Expressions

# Formal Languages: Recap

- Symbols: a, b, c
- Alphabet : finite set of symbols $\Sigma = \{a, b\}$
- String: sequence of symbols bab
- Empty string: $\varepsilon$ Define: $\Sigma^{\varepsilon} = \Sigma \cup \{\varepsilon\}$
- Set of all strings: $\Sigma^*$
  - $\Sigma^0, \Sigma^1, \Sigma^2, \ldots \Sigma^n$
- (Formal) Language: a set of strings

$$\{ a^n b^n : n > 0 \}$$

# Regular Languages

- The set of regular languages: each element is a regular language
  - $R = \{R_1 , R_2 , \ldots, R_n,\ldots\}$
- Each regular language is an example of a (formal) language, i.e. a set of strings

  e.g. $\{ a^m b^n : m, n$ are positive integers $\}$

# Regular Languages

- Defining the set of all regular languages:
  - The empty set and {a} for all a in $\Sigma^{\varepsilon}$ are regular languages
  - If $L_1$ and $L_2$ and L are regular languages, then:

    $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$ (concatenation)

    $L_1 \cup L_2$ (union)

    $L^* = \cup_{i=0}^{\infty} L^i$ (Kleene closure)

    are also regular languages
  - There are no other regular languages

# Formal Grammars

- A formal grammar is a concise description of a formal language $\{\varepsilon, a, aa, \ldots\} = a^*$
- A formal grammar uses a specialized syntax
- For example, a **regular expression** is a concise description of a regular language

  *(a|b)*abb* : is the set of all strings over the alphabet {*a*, *b*} which end in *abb*
- We will use regular expressions (regexps) in order to define tokens in our compiler,
  - e.g. lexemes for string tokens are \" (Σ-{"})* \"

# Regular Expressions: Definition

- Every symbol of $\Sigma \cup \{ \varepsilon \}$ is a regular expression
  - E.g. if $\Sigma = \{a,b\}$ then ‘a’, ‘b’ are regexps
- If $r_1$ and $r_2$ are regular expressions, then the core operators to combine two regexps are
  - Concatenation: $r_1 r_2$, e.g. ‘ab’ or ‘aba’
  - Alternation: $r_1 | r_2$, e.g. ‘a|b’
  - Repetition: $r_1$*, e.g. ‘a*’ or ‘b*’
- No other core operators are defined
  - But other operators can be defined using the basic operators (as in lex regular expressions) e.g. a+ = aa*

# Lex regular expressions

<table>
<tr><th>Expression</th><th>Matches</th><th>Example</th><th>Using core operators</th></tr>
<tr><td>c</td><td>non-operator character c</td><td><code>a</code></td><td></td></tr>
<tr><td>\c</td><td>character c literally</td><td><code>*</code></td><td></td></tr>
<tr><td>"s"</td><td>string s literally</td><td><code>"**"</code></td><td></td></tr>
<tr><td>.</td><td>any character but newline</td><td><code>a.*b</code></td><td></td></tr>
<tr><td>^</td><td>beginning of line</td><td><code>^abc</code></td><td>used for matching</td></tr>
<tr><td>$</i></td><td>end of line</td><td><code>abc$</code></td><td>used for matching</td></tr>
<tr><td>[s]</td><td>any one of characters in string s</td><td><code>[abc]</code></td><td>(a|b|c)</td></tr>
<tr><td>[^s]</td><td>any one character not in string s</td><td><code>[^a]</code></td><td>(b|c) where □ = {a,b, c}</td></tr>
<tr><td>r*</td><td>zero or more strings matching r</td><td><code>a*</code></td><td></td></tr>
<tr><td>r+</td><td>one or more strings matching r</td><td><code>a+</code></td><td>aa*</td></tr>
<tr><td>r?</td><td>zero or one r</td><td><code>a?</code></td><td>(a|ε)</td></tr>
<tr><td>r{m,n}</td><td>between m and n occurences of r</td><td><code>a{2,3}</code></td><td>(aa|aaa)</td></tr>
<tr><td>$r_1r_2$</td><td>an $r_1$ followed by an $r_2$</td><td><code>ab</code></td><td></td></tr>
<tr><td>$r_1|r_2$</td><td>an $r_1$ or an $r_2$</td><td><code>a|b</code></td><td></td></tr>
<tr><td>(r)</td><td>same as r</td><td><code>(a|b)</code></td><td></td></tr>
<tr><td>$r_1/r_2$</td><td>$r_1$ when followed by an $r_2$</td><td><code>abc/123</code></td><td>used for matching</td></tr>
</table>