

# CMPT 379

## Compilers

Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

# 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, use only if
  - Your language's lexical structure is very simple
  - The tools do not provide what you need for your token definitions
- 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:  $\epsilon$       Define:  $\Sigma^\epsilon = \Sigma \cup \{\epsilon\}$
- Set of all strings:  $\Sigma^*$       cf. *The Library of Babel*, Jorge Luis Borges
- (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
- Each regular language is an example of a (formal) language, i.e. a set of strings  
e.g.  $\{ a^m b^n : m, n \text{ are +ve integers} \}$

# 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)
    - $L^* = \bigcup_{i=0}^{\infty} L^i$  (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
- 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  $\text{"}(\Sigma\text{"})^*\text{"}$

# 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

Expression	Matches	Example	Using core operators
$c$	non-operator character $c$	$a$	
$\backslash c$	character $c$ literally	$\backslash *$	
$"s"$	string $s$ literally	$"**"$	
$.$	any character but newline	$a.*b$	
$\wedge$	beginning of line	$\wedge abc$	used for matching
$\$$	end of line	$abc\$$	used for matching
$[s]$	any one of characters in string $s$	$[abc]$	$(ablc)$
$[^s]$	any one character not in string $s$	$[^a]$	$(blc)$ where $\Sigma = \{a,b,c\}$
$r^*$	zero or more strings matching $r$	$a^*$	
$r^+$	one or more strings matching $r$	$a^+$	$aa^*$
$r?$	zero or one $r$	$a?$	$(a\epsilon)$
$r\{m,n\}$	between $m$ and $n$ occurrences of $r$	$a\{2,3\}$	$(aala aa)$
$r_1 r_2$	an $r_1$ followed by an $r_2$	$ab$	
$r_1/r_2$	an $r_1$ or an $r_2$	$a b$	
$(r)$	same as $r$	$(a b)$	
$r_1/r_2$	$r_1$ when followed by an $r_2$	$abc/123$	used for matching