CS143: Lexical Analysis

David L. Dill
Stanford University
Introduction

• PA1 (or PA1J) is assigned today.
  • Due in one week
  • I recommend starting immediately.

• Review notes on regular languages based on CS103
  • Includes DFA minimization – optional

• Notes on the material of lectures 1, 3, 4.
Lexical Analysis

- Interface to Rest of Compiler
- Formal Languages Concepts
- Regular Expressions
- Deterministic Finite Automata
- Theory vs. Practice
Interfaces to Rest of Compiler
Lexical analysis

letters $\rightarrow$ words

characess $\rightarrow$ Lexical Analyzer (lexer) $\rightarrow$ Tokens
if $x = y$ then $z = 1$, else $z = 2$
Interface

Characters → lexer → parser

tokens
Interface

Characters → lexer → parser

tokens are generated "on demand" for parser.
Interface

Characters $\rightarrow$ lexer $\rightarrow$ parser

tokens are generated "on demand" for parser.

getLex() $\rightarrow$ returns next token
if \( x == y \) then \( z = 1 \), else \( z = 2 \).

next input
if $x = y$ then $z = 1$, else $z = 2$.

next input

getlex() returns \[ \text{if } \text{token} \]
if \( x == y \) then \( z = 1 \), else \( z = 2 \)
if \( x = y \) then \( z = 1 \), else \( z = 2 \).

next input

\( \text{get lex}() \) returns \([ \text{if} \text{ token} \]

\( \text{get lex}() \) returns \([ x ]\)

(discards space after "if" and keeps going until it can return a token.)
Terminology

if \( x = y \) then \(* * *\)
Terminology

\[
\text{If } x == y \text{ then } \cdots
\]

\[\wedge\]

lexeme - input string for next token
Terminology

\[ \text{If } x = y \text{ then } \]  

\[ \text{getLex()} \rightarrow \text{if} \]

\[ \text{token data structure returned by getLex} \]

\[ \text{goes to parser} \]
Performance

Lexer used to be performance-critical

e.g. before 1990?

# of characters vs. # of tokens

Lexer had to deal with all of these
eater stages just had tokens & trees.
Performance

Never used to be performance critical

So we know how to do it FAST

Q: Why is the North American Antelope so speedy?
Formal Languages Concepts
Lexical Analysis Theory

Precise descriptions - Regular expressions

\[[a-z A-Z][a-z A-Z 0-9-]+\]

Automatic generation

Regular expressions $\rightarrow$ NFA $\rightarrow$ DFA

$\rightarrow$ lexical analyzer program
Formal language Concepts

**Alphabet** - A finite set of symbols (a.k.a characters).

\[ \Sigma \] - standard symbol for an alphabet.
Formal Language Concepts

Alphabet – A finite set of symbols (a.k.a characters).

\[ \Sigma \] – Standard symbol for an alphabet.

For compilers, a character set like ASCII, UTF8
Formal Language Concepts

String - finite sequence of characters

E.g. abc
Formal Language Concepts

1 \* 1 - length of the string \( x \)

\( \emptyset \) - empty string \( |\emptyset| = 0 \)
Formal Language Concepts

$x \cdot y$ - concatenation of strings
(also written $x y$)

$abc \cdot def = abcdef$
Formal Language Concepts

Formal language – set of strings over some alphabet $\Sigma$. 

Formal language Concepts

If $X, Y$ are languages over $\Sigma$, $X \cdot Y = \{xy \mid x \in X \land y \in Y\}$ is their concatenation.
Formal Language Concepts

If $X, Y$ are languages over $\Sigma$, $X \cdot Y = \{ xy \mid x \in X \land y \in Y \}$ is their concatenation.

Ex: $\{a, ab\} \cdot \{b, bb\} = \{ab, aabb, abbb\}$
Formal Language Concepts

Let $X$ be a language. Then $X^i$ is $X$ concatenated with itself $i$ times.

$X^0 = \{ \varepsilon \}$

$X^1 = X$

$X^2 = XX$

etc.
Formal Language Concepts

\[ X^* = X^0 U X^1 U X^2 U \ldots \]

is the Kleene closure of \( X. \)
Regular Expressions
Regular Expressions

Expression:
- $\varepsilon$
- $a \ (a \in \Sigma)$
- $R_1 \mid R_2$
- $R_1 \cdot R_2$
- $R_1^*$

Language:
- $L(\varepsilon) = \{ \varepsilon \}$
- $L(a) = \{ a \}$
- $L(R_1 \mid R_2) = L(R_1) \cup L(R_2)$
- $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$
- $L(R_1^*) = L(R_1)^*$
Regular Expressions

Also: Precedence

\[ * \rightarrow \rightarrow \rightarrow ]

"stickiest" like exponentiation
Regular Expressions

Also: Precedence

* \textbullet \textbullet \textbullet ! \uparrow

next stickiest
like multiplication
Regular Expressions

Also: Precedence

* , , ,

↑

least sticky
like addition
Regular Expressions

Also: Precedence

\[
* \quad \cdot \quad | \quad \]

Parentheses to enforce grouping

\[
((a1 b) (c1 d))^* \]

Notation used in tools is somewhat different (and more complex).
Abbreviations

$R^? \; - \; \text{"optional} \; R^n$

abbrev for $(R \cup \varepsilon)$

$R^+ \; - \; \text{positive closure (one or more)}$

abbrev for $RR^* \; \text{or} \; R^*R$. 

Regular Definitions

Allow new named abbreviations for regular expressions
Example: Java floating point (partial)

\[\begin{align*}
\text{Exp}^+: & \quad ((e|E)(+|-)? \text{digit}^+) \\
\text{Suffix}: & \quad (f|F|d|D) \\
\text{digit}^+ & \cdot \text{digit}^* \text{Exp}^? \text{Suffix}^? \\
1 & \cdot \text{digit}^+ \text{Exp}^? \text{Suffix}^? \\
1 & \text{digit}^+ \text{Exp}^ \text{Suffix}^? \\
1 & \text{digit}^+ \text{Exp}^? \text{Suffix}^?
\end{align*}\]
Deterministic Finite Automata
Deterministic Finite Automata

Good for implementing lexers.

*Input string* → $\text{accept}$ (action after reading entire string) → $\text{reject}$
DFA

\( Q \) - finite set of states

\( \Sigma \) - alphabet

\( S : Q \times \Sigma \to Q \) - next state function

\( q_0 \in Q \) - start state

\( F \subseteq Q \) - accepting states.
Example: Identifiers

\[
\text{letter} \ (\text{letter} \ | \ \text{digit} \ (-))^* \\
\text{digit} \ [0-9] \\
\text{letter} \ [a-zA-Z]
\]

Diagram:

- Start state \( q_0 \)
- Transition: \( \text{letter} \rightarrow q_1 \)
- Transition: \( \text{digit} \rightarrow q_1 \)
- Transition: \( - \rightarrow \text{underbar} \)
- Accept state \( q_1 \)
Example: Identifiers

\[ \text{letter} \ (\text{letter} \mid \text{digit} \mid -)^* \]

At most one arrow from each state on each symbol ("deterministic")

OK to have no arrow on symbol (S partial function)
Example: Identifiers

letter (letter | digit | -) *

Input: ab

\[
q_0 \xrightarrow{a} q_1 \xrightarrow{1} q_1 \xrightarrow{b} q_1
\]
Example: Identifiers

\[ \text{letter} \ (\text{letter} \mid \text{digit} \mid \_)^* \]

Input: a1b

Start state: \( q_0 \)

Transition:
- \( q_0 \xrightarrow{a} q_1 \)
- \( q_1 \xrightarrow{1} q_1 \)
- \( q_1 \xrightarrow{b} q_2 \)

Last state is accepting, \( a1b \) is identifier.
Automatic Conversion

Regular expression $\rightarrow$ NFA with $\varepsilon$ transitions

$\rightarrow$ DFA

$\rightarrow$ minimized DFA

Review notes posted
Example: Modula-2 Real Numbers
digit* digit* (E (+|-) ? digit*)?

DFA?
Example: Modula-2 Real Numbers

digit\* digit\* (E (+1-)? digit\*)?
Theory vs. Practice
Theory

Practice

1. Lexeme is prefix of input
2. Multiple token types
Combined Automaton

Real: \( \text{digit}^+ \cdot \text{digit}^* \ (E \ (+/-) \ ? \ \text{digit}^+) \).

Int: \( \text{digit}^+ \)

[Diagram of a finite state machine with transitions labeled d and E, and an accept state for INT]
Combined Automaton

Real: digit* · digit* (E (+|-) ? digit*).

Int: digit*
Input: 12.34+6

Longest lexeme rule ("maximal munch")

Tokens: [1, 2.3, 4, 6]

Tokens: [12, 34, ?]
Input: 12.34+6

(next char to read)
Input: 1 2 3 4 + 6

[start_char] nextchar

(advanced)
Input: $12.34 + 6$

(last char) (end of longest lexeme so far = "1" in this case)

(start char) next char
Input: 1 2.3 4 + 6

longest so far: “12”
Input: 12.34+6

longest so far: "12.0"
Input: 12.34+6

longest so far: "12.3"

start char

next char

last char

\[\rightarrow d \rightarrow \text{INT} \rightarrow d \rightarrow \text{R} \rightarrow d \rightarrow \text{E} \rightarrow d \rightarrow \text{R} \rightarrow d \rightarrow + \rightarrow -\]
Input: 12.34+6

longest so far: "12.34"

start char

next char

last char
Input: 12.34+6

longest so far: "12.34"

start_char

next_char

last_char

no arrow for "+"
Input: 1 2.34+6

Start char

Last char

Next char

Longest so far: "12.34"

No arrow for "+"
Input: 1 2.3 4+6

Ready for next lexeme

start_char
next_char

Diagram:  
- Input arrow to INT
- INT to R
- R to E
- E to + and -
Input: 12.34+6

"Look ahead" — had to read "+" to see that end of lexeme was previous char.

Look ahead = max(next char - last char)
Look ahead for this example is at least 1. Is it more?
Unbounded lookahead?

\[ a(a b^* a) \]

Input: \( aa b b b b b b b b b b b b b ... \ b \)

May need to read whole file to find first lexeme!
C Comments

/* ... */

Has unbounded lookahead if handled with a single pattern.

If closing "*/" is missing, tokenizes the comment: /* ...

Saves string unnecessarily.
Start Conditions

A better way to handle comments

When you see "/*"

Use new DFA

Patterns:
  - any single char — discard
  - "*/" — discard, return to main DFA
Rule Ordering

`ID letter(letter | digit | _)`
Rule Ordering

if

ID

letter(letter | digit | _)

label for first rule in file wins.
Rule Ordering

if if ← special case first

ID letter (letter | digit | _ ) ← default

← label for first rule in file wins.