CSE 340
Principles of Programming Languages

Lexical Analysis

Ayan Banerjee

Arizona State University
Adopted from slides by Adam Doupe

IRA A. FULTON SCHOOLS OF engineering
ARIZONA STATE UNIVERSITY

Language Syntax

- Programming Language must have a clearly specified syntax
- Programmers can learn the syntax and know what is allowed and what is not allowed
- Compiler writers can understand programs and enforce the syntax

Language Syntax

- Input is a series of bytes
  - How to get from a string of characters to program execution?
- We must first assemble the string of characters into something that a program can understand
- Output is a series of tokens

Language Syntax

- In English, we have an alphabet
  - a…z, ,, ., !, ?, …
- However, we also have a higher abstraction than letter in the alphabet
- Words
  - Defined in a dictionary
  - Categorized into
    - Nouns
    - Verbs
    - Adverbs
    - Articles
- Sentences
- Paragraphs

Language Syntax

- In a programming language, we also have an alphabet (the symbols that are important in the specific language)
  - a, z, ,, ., !, ?, <, >, ;, }, {, (,), …
- Just as in English, we create abstractions of the low-level alphabet
- Tokens
  - ==
  - <=
  - while
  - if
- Tokens are precisely specified using patterns

Strings

- Alphabet symbols together make a string
- We define a string over an alphabet Σ as a finite sequence of symbols from Σ
- ε is the empty string, an empty sequence of symbols
- Concatenating ε with a string s gives s
  - $\varepsilon s = s\, \varepsilon = s$
- In our examples, strings will be stylized differently, either
  - "in between double quotes"
  - *italic and dark blue*

Languages

- Σ represents the set of all symbols in an alphabet
- We define Σ^* as the set of all strings over Σ
  - Σ^* contains all the strings that can be created by combining the alphabet symbols into a string
- A language L over alphabet Σ is a set of strings over Σ
  - A language L is a subset of Σ^*
- Is Σ infinite?
- Is Σ^* infinite?
- Is L infinite?

Regular Expressions

- Tokens are typically specified using regular expressions
- Regular expressions are
  - Compact
  - Expressive
  - Precise
  - Widely used
  - Easy to generate an efficient program to match a regular expression

Regular Expressions

- We must first define the syntax of regular expressions
- A regular expression is either
  1. $\varnothing$
  2. ε
  3. a, where a is an element of the alphabet
  4. $R_1 \mid R_2$, where R_1 and R_2 are regular expressions
  5. $R_1 . R_2$, where R_1 and R_2 are regular expressions
  6. (R), where R is a regular expression
  7. R*, where R is a regular expression

Regular Expressions

- A regular expression defines a language (the set of all strings that the regular expression describes)
- The language L(R) of regular expression R is given by:
    1. $L(\varnothing) = \varnothing$
    2. $L(\boldsymbol{\varepsilon}) = \{\varepsilon\}$
    3. $L(a) = \{a\}$
    4. $L(R_1 \mid R_2) = L(R_1) \cup L(R_2)$
    5. $L(R_1 . R_2) = L(R_1) . L(R_2)$

$L(R_1 \mid R_2) = L(R_1) \cup L(R_2)$

Examples:

$L(a \mid b) = L(a) \cup L(b) = \{a\} \cup \{b\} = \{a, b\}$

$L(a \mid b \mid c) = L(a \mid b) \cup L(c) = \{a, b\} \cup \{c\} = \{a, b, c\}$

$L(a \mid \varepsilon) = L(a) \cup L(\varepsilon) = \{a\} \cup \{\varepsilon\} = \{a, \varepsilon\}$

$L(\varepsilon \mid \varepsilon) = \{\varepsilon\}$

$\{\varepsilon\} \neq \{\}$

$L(R_1 \, . \, R_2) = L(R_1) \, . \, L(R_2)$

Definition

For two sets A and B of strings:

$A \, . \, B = \{xy : x \in A \text{ and } y \in B\}$

Examples:

$A = \{aa, b\}$, $B = \{a, b\}$

$A \, . \, B = \{aaa, aab, ba, bb\}$

$ab \notin A \, . \, B$

$A = \{aa, b, \varepsilon\}$, $B = \{a, b\}$

$A \, . \, B = \{aaa, aab, ba, bb, a, b\}$

Operator Precedence

L(a | b . c)

What does this mean?

(a | b) . c or a | (b . c)

Just like in math or a programming language, we must define the operator precedence (* higher precedence than +)

a + b * c

(a + b) * c or a + (b * c)?

. has higher precedence than |

L(a | b . c) =

L(a) ∪ L(b . c) = {*a*} ∪ {*bc*} = {*a*, *bc*}

Regular Expressions

$L(\,(R)\,) = L(R)$

$L(\,(a \mid b)\,.\,c\,) =$

$L\,(a \mid b)\,.\,L\,(c) =$

$\{a, b\}\,.\,\{c\} =$

$\{ac, bc\}$

Kleene Star

$L(R^*) = ?$

$L\ (R^*) = \{\varepsilon\} \cup L(R) \cup L(R) \,.\, L(R) \cup L(R) \,.\, L(R) \,.\, L(R) \cup$
$L(R) \,.\, L(R) \,.\, L(R) \,.\, L(R) \ldots$

Definition

$L^0(R) = \{\varepsilon\}$

$L^i(R) = L^{i-1}(R) \,.\, L(R)$

$L(R^*) = \cup_{i \geq 0} L^i(R)$

$L(R^*) = \cup_{i \geq 0} L^i(R)$

Examples

$L(a \mid b^*) = \{a, \varepsilon, b, bb, bbb, bbbb, \ldots\}$

$L((a \mid b)^*) = \{\varepsilon\} \cup \{a, b\} \cup \{aa, ab, ba, bb\} \cup \{aaa, aab, aba, abb, baa, bab, bba, bbb\} \cup \ldots$

Tokens

letter = a | b | c | d | e | ... | A | B | C | D | E...

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

ID = letter(letter | digit | _)*

a891_jksdbed

12ajkdfjb

Note that we've left out the . regular expression operator. It is implied when two regular expressions are next to each other, similar to x*y=xy in math.

Tokens

How to define a number?

NUM = digit*

132

ε

NUM = digit(digit)*

132

0

00000000000

pdigit = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

NUM = pdigit(digit)*

132

0

00000000000

Tokens

NUM = pdigit . (digit)* | 0

123

0

00000000

1901adb

Tokens

How to define a decimal number?

DECIMAL = NUM . \. . NUM

1.5

2.10

1.01

DECIMAL = NUM . \. . digit*

1.5

2.10

1.01

1.

DECIMAL = NUM . \. . digit . digit*

1.5

2.10

1.01

1.

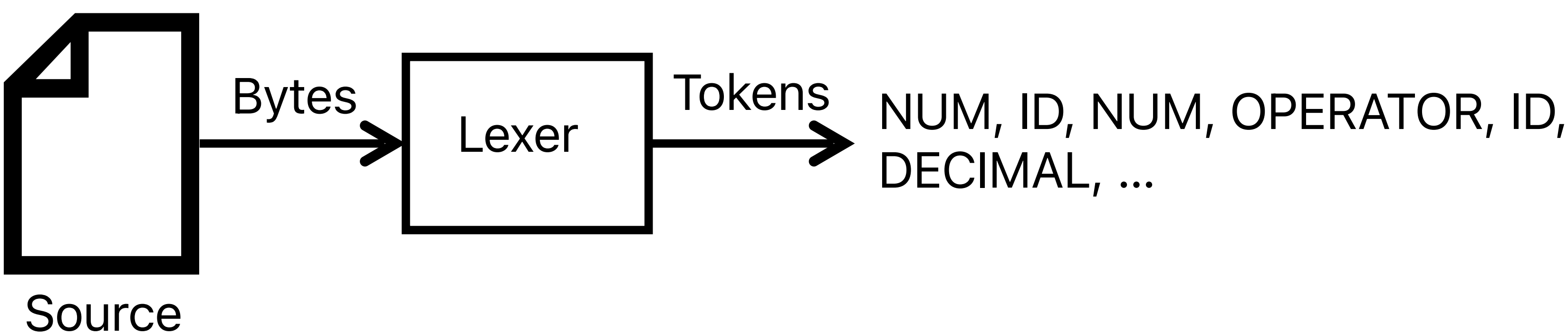
0.00

Note that here we mean a regular expression that matches the one-character string dot . However, to differentiate between the regular expression concatenation operator . and the character ., we escape the . with a \ (similar to strings where \n represents the newline character in a string). This means that we also need to escape \ with a \ so that the regular expression \\ matches the string containing the single character \

Lexical Analysis

- The job of the lexer is to turn a series of bytes (composed from the alphabet) into a sequence of tokens
  - The API that we will discuss in this class will refer to the lexer as having a function called getToken(), which returns the next token from the input steam each time it is called

- Tokens are specified using regular expressions



Lexical Analysis

Given these tokens:

ID = letter . (letter | digit | _)*

DOT = \.

NUM = pdigit . (digit)* | 0

DECIMAL = NUM . DOT . digit . digit*

What token does getToken() return on this string:

1.1abc1.2

NUM?

DECIMAL?

ID?

Longest Matching Prefix Rule

- Starting from the next input symbol, find the longest string that matches a token
- Break ties by giving preference to token listed first in the list

String	Matching	Potential	Longest Match
1.1abc1.2		All	
1.1abc1.2	NUM	DECIMAL, NUM	NUM, 1
1.1abc1.2		DECIMAL	NUM, 1
1.1abc1.2	DECIMAL	DECIMAL	DECIMAL, 3
1.1abc1.2			
1.1abc1.2		All	
abc1.2	ID	ID	ID, 1
abc1.2	ID	ID	ID, 2
abc1.2	ID	ID	ID, 3
abc1.2	ID	ID	ID, 4
abc1.2			ID, 4
abc1.2		All	
.2	DOT		DOT, 1
.2			DOT, 1
.2		All	
2	NUM	NUM	NUM, 1
2			NUM, 1

Mariner 1

Lexical Analysis

- In some programming languages, whitespace is not significant at all
  - In most programming language, whitespace is not always significant
    - (5 + 10) vs. (5+10)
- In Fortran, whitespace is ignored
- DO 15 I = 1,100
- DO 15 I = 1.100
- DO15I = 1.100
  - Variable assignment instead of a loop!