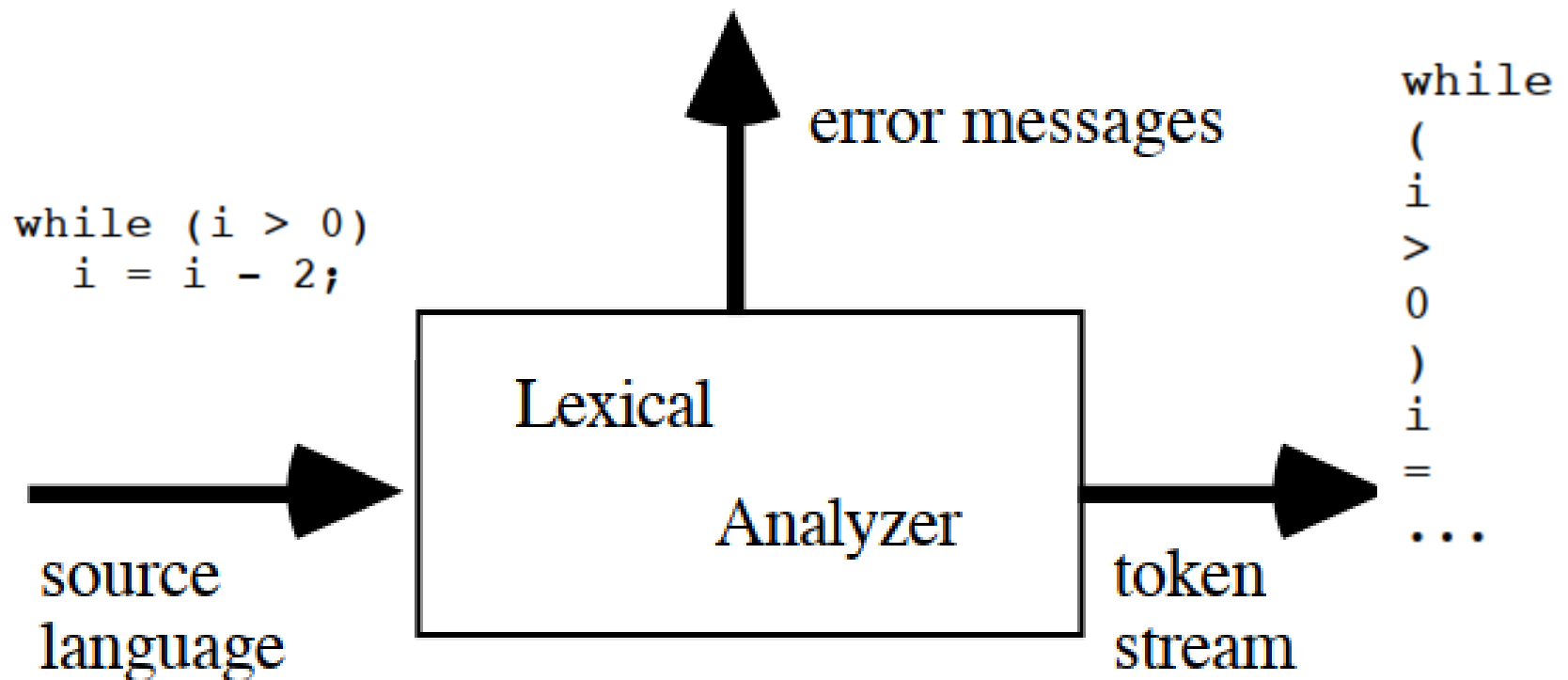# Lexical Analysis

## Lecture 3

# Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string


- Issues in lexical analysis
  - Lookahead
  - Ambiguities


- Specifying lexers (aka. scanners)
  - By regular expressions (aka. regex)
  - Examples of regular expressions

# Lexical Analysis

- **Lexical analysis** or **scanning** is the process where the stream of characters making up the source program is read from left to right and grouped into tokens.
- Tokens are sequences of characters with collective meaning.
- There are usually only a small number of tokens classes for a programming language: constants (integer, double, char, string, etc.), operators (arithmetic, relational, logical), punctuation, and reserved words.

# Lexical Analysis



while (i > 0)
  i = i - 2;

source
language

Lexical

Analyzer

error messages

token
stream

while
(
i
>
0
)
i
=
...

# Lexical Analysis

- Example:
  if (i == j)
      Z = 0;
  else
      Z = 1;

- The input is just a string of characters:
  \tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

- Goal: Partition input string into substrings
  – Where the substrings are called tokens

# What's a Token?

- A syntactic category
  - In English:
    noun, verb, adjective, …

  - In a programming language:
    Identifier, Integer, Keyword, Whitespace, …
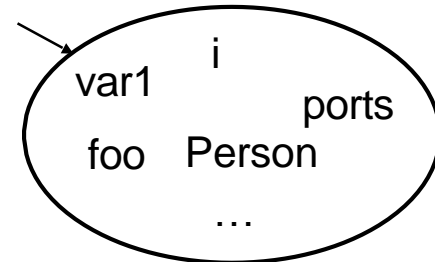
# Tokens

- A token class corresponds to a set of strings

Infinite set

- Examples
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: "else" or "if" or "begin" or …
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

var1   i   ports
foo   Person
…

# What are Tokens For?

- Classify program substrings according to role

- Lexical analysis produces a stream of tokens
  … which is input to the parser

- Parser relies on token distinctions
  – An identifier is treated differently than a keyword

# Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens

  - Tokens describe all items of interest
    - Identifiers, integers, keywords

  - Choice of tokens depends on
    - language
    - design of parser

# Example

- Recall
  \t if (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

- Useful tokens for this expression:
  Integer, Keyword, Relation, Identifier, Whitespace, (, ), =, ;

- N.B., (, ), =, ; above are tokens, not characters

# Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token class


- Recall:
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: "else" or "if" or "begin" or …
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

# Lexical Analyzer: Implementation

- An implementation must do two things:

   1. Classify each substring as a token

   2. Return the value or lexeme (value) of the token
      – The lexeme is the actual substring
      – From the set of substrings that make up the token

- The lexer thus returns token-lexeme pairs
   – And potentially also line numbers, file names, etc. to improve later error messages

# Lexical Analyzer: Implementation

- The lexer usually discards "uninteresting" tokens that don't contribute to parsing.

- Examples: Whitespace, Comments

# True Crimes of Lexical Analysis

- Is it as easy as it sounds?

- Sort of… if you do not make it hard!

- Look at some history

# Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant

- E.g., VAR1 is the same as VA    R1

- A terrible design!

- Historical footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators

# FORTRAN Example

- Consider
  - DO 5  I = 1,25
  - DO 5  I = 1.25

# Lexical Analysis in FORTRAN (Cont.)

- Two important points:

  1. The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time

  2. "Lookahead" may be required to decide where one token ends and the next token begins

# Lookahead

- Even our simple example has lookahead issues
    - i vs. if
    - = vs. ==

# Lexical Analysis in C++

- Unfortunately, the problems continue today

- C++ template syntax:

  Foo<Bar>

- C++ stream syntax:

  cin >> var;

- But there is a conflict with nested templates:

  Foo<Bar<Bazz>>

  Closing templates, not stream

20

# Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme

- Left-to-right scan => lookahead sometimes required

# **Next**

- We still need
  - A way to describe the lexemes of each token

  - A way to resolve ambiguities
    - Is if two variables i and f?
    - Is == two equal signs = =?

# Regular Languages

- There are two primary methods for implementing a scanner. The first is a program that is hard-coded to perform the scanning tasks. The second uses regular expression and finite automata theory to model the scanning process.

- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

## Languages

**Symbol:** an abstract entity that we shall not define formally (such as "point" in geometry). Letters, digits and punctuation are examples of symbols.

**Alphabet:** a finite set of symbols out of which we build larger structures. An alphabet is typically denoted using the Greek sigma $\sum$, e.g., $\sum = \{0,1\}$.

**Strings:** a finite sequence of symbols from a particular alphabet juxtaposed. For example: a, b, c, are symbols and abcb is a string.

# Languages

**Empty Strings:** denoted ε (or sometimes ∂) is the string consisting of zero symbols.
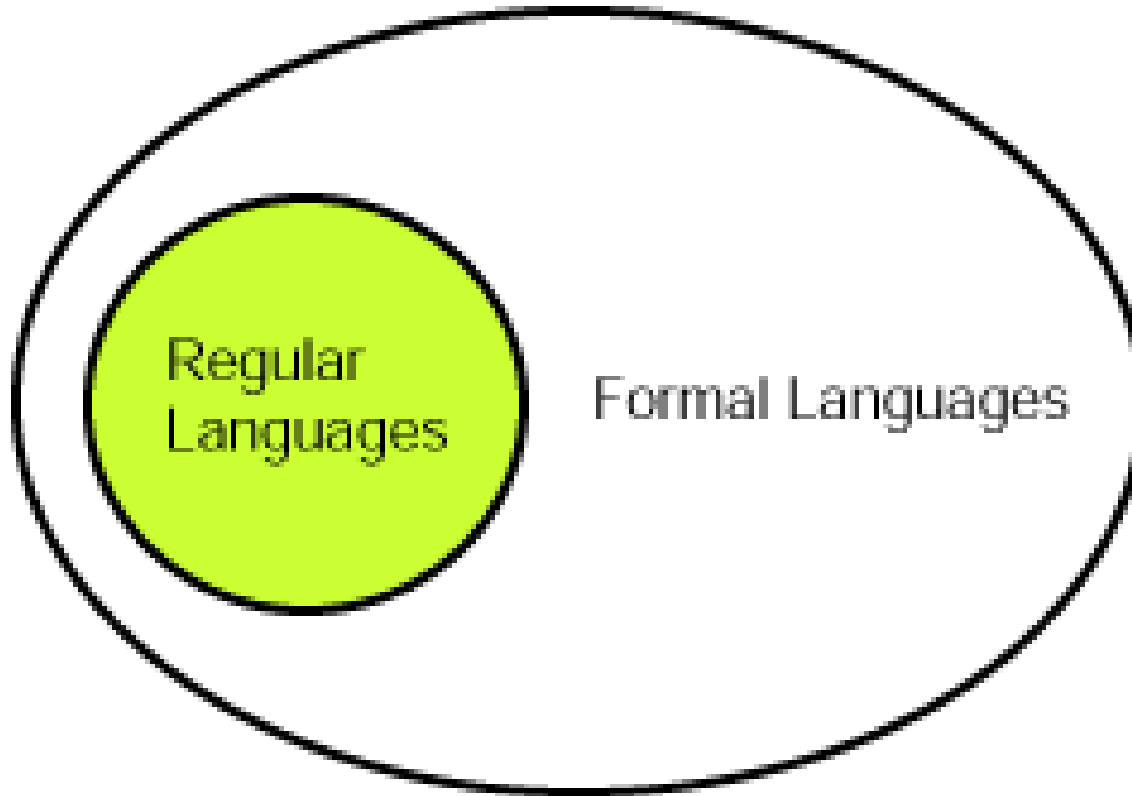
**Language:** set of strings.

**Regular Language∑\*:** A language (a set of strings) that can be described by a finite automaton is called a regular language.

For example, the set of legal Java identifiers is a regular language. The set of floating point literals (strings like 8.63 or -12.8E3) is a regular language.

# Examples of Languages

- Alphabet = English characters

- Language = English sentences

- Not every string of English characters is an English sentence

- Alphabet = ASCII

- Language = C programs

- Note: ASCII character set is different from English character set

# Formal Languages



Regular Languages

Formal Languages

# Notation

- Languages are sets of strings.

- Need some notation for specifying which sets we want

- The standard notation for regular languages is regular expressions.

# Atomic Regular Expressions

- Single character

$$'c\,' = \{\,''c''\,\}$$

- Epsilon

$$\varepsilon = \{\,''''\,\}$$

Not the empty set, but set with a single, empty, string.

# Compound Regular Expressions

- Union

$$A + B = \{ s \mid s \in A \text{ or } s \in B \}$$

- Concatenation

$$AB = \{ ab \mid a \in A \text{ and } b \in B \}$$

- Iteration

$$A^* = \cup_{i \geq 0} A^i \text{ where } A^i = \underbrace{AA \ldots A}_{i \text{ times}}$$

# Regular Expressions

- **Def.** The regular expressions over (alphabets)Σ are the smallest set of expressions including

- R $= \varepsilon$

$$| \: 'c' \quad \text{where } c \in \sum$$

$$| \: A + B$$

$$| \: AB$$

$$| \: A^* \quad \text{where } A \text{ is a rexp over } \sum$$

# Syntax vs. Semantics

- To be careful, we distinguish syntax and semantics.

$$L(\varepsilon) \quad = \quad \{""\}$$

$$L('c') \quad = \quad \{"c"\}$$

$$L(A+B) \quad = \quad L(A) \cup L(B)$$

$$L(AB) \quad = \quad \{ab \mid a \in L(A) \text{ and } b \in L(B)\}$$

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

# Segue

- Regular expressions are simple, almost trivial
  - But they are useful!


- We will describe tokens in regular expressions

# Example: Keyword

Keyword: "else" or "if" or "begin" or …

'else' + 'if' + 'begin' + . . .

Abbreviation: 'else' = 'e' 'l' 's' 'e'

# Example: Integers

Integer: a non-empty string of digits

$$\text{digit} \quad = \quad '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'$$

# Example: Integers

Integer: a non-empty string of digits

digit $=$ '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'

integer $=$ digit digit$^*$

Abbreviation: $A^+ = AA^*$

Abbreviation: [0-2] = '0' + '1' + '2'

# Example: Identifier

Identifier: strings of letters or digits, starting with a letter

letter    =  'A' + . . . + 'Z' + 'a' + . . . + 'z'

# Example: Identifier

Identifier: strings of letters or digits, starting with a letter

letter      =  'A' + . . . + 'Z' + 'a' + . . . + 'z'

identifier =  letter (letter + digit)*

Is (letter* + digit*) the same as (letter + digit)*?

# Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

$$(' ' + '\backslash n' + '\backslash t')^{+}$$

# Example: Phone Numbers

- Regular expressions are all around you!
- Consider (650)-723-3232

$$\Sigma \qquad\qquad = \quad \text{digits} \ \cup \ \{-,(,)\}$$

$$\text{exchange} \qquad = \quad \text{digit}^3$$

$$\text{phone} \qquad\qquad = \quad \text{digit}^4$$

$$\text{area} \qquad\qquad = \quad \text{digit}^3$$

$$\text{phone\_number} \quad = \quad \text{'(' area ')-' exchange '-' phone}$$

# Example: Email Addresses

- Consider <u>anyone@xyz.edu</u>

$$\Sigma \quad = \quad \text{letters} \ \cup \{.,@\}$$

$$\text{name} \quad = \quad \text{letter}^+$$

$$\text{address} \quad = \quad \text{name '@' name '.' name}$$

# Example: Unsigned Pascal Numbers

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

digits = digit$^+$

opt_fraction = ('.' digits) + ε

opt_exponent = ('E' ('+' + '-' + ε) digits) + ε

num = Digits + opt_fraction +
opt_exponent

# Other Examples

- File names
- Grep tool family

# **Summary**

- Regular expressions describe many useful languages
  - We will look at non-regular languages next week

- Regular languages are a language specification
  - We still need an implementation

- Next time: Given a string s and a regx R, is

$$s \in L(R)?$$