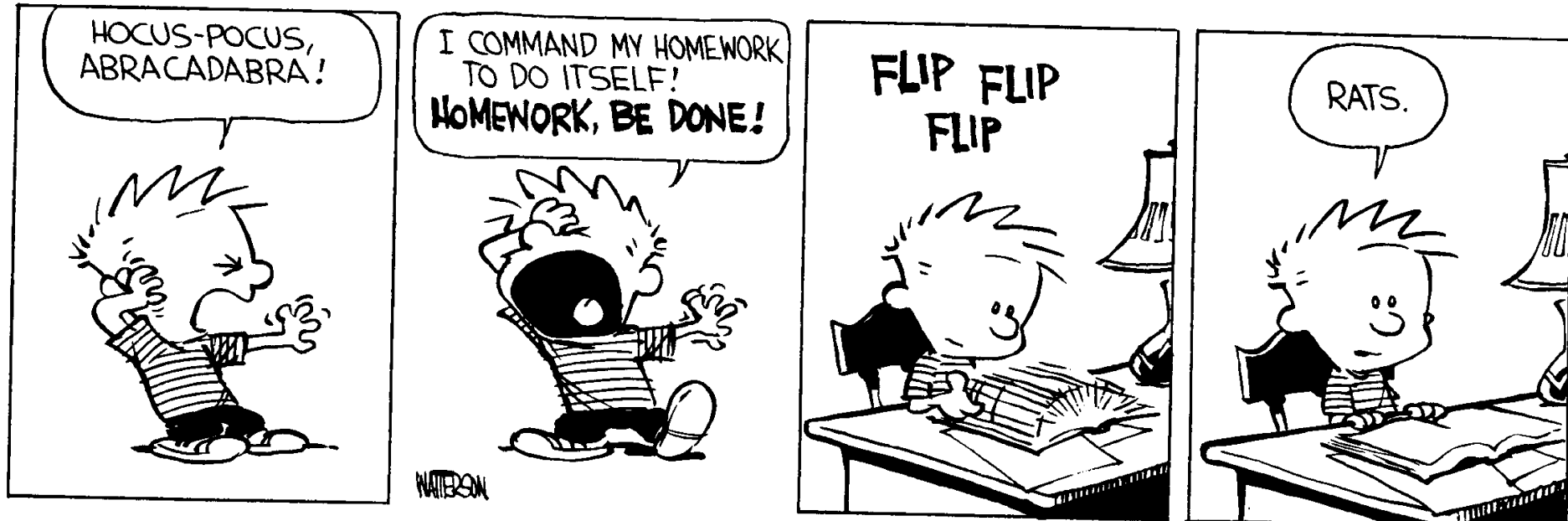


# Lexical Analysis

## Finite Automata

(Part 1 of 2)



# Cunning Plan

- Informal Sketch of Lexical Analysis
  - LA identifies tokens from input string
  - lexer : (char list)  $\rightarrow$  (token list)
- Issues in Lexical Analysis
  - Lookahead
  - Ambiguity
- Specifying Lexers
  - Regular Expressions
  - Examples

# One-Slide Summary

- Lexical analysis turns a stream of characters into a stream of tokens.
- Regular expressions are a way to specify sets of strings. We use them to describe tokens.

# Fold Batter Lightly ...

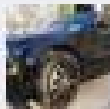
- `fold_left f a [1;...;n] == f (... (f (f a 1) 2)) n`
  - `fold_left (fun a e -> e :: a) [] [1;2;3] = [3;2;1]`
  - `fold_left (fun a e -> a @ [e]) [] [1;2;3] = [1;2;3]`
- `fold_right f [1;...;n] b == f 1 (f 2 (... (f n b)))`
  - `fold_right (fun a e -> e :: a) [1;2;3] [] = [1;2;3]`
  - `fold_right (fun e a -> a @ [e]) [1;2;3] [] = [3;2;1]`



## Internet Explorer

Question of the day: Which technological invention do you think has impacted our lives more - the telephone or the internet?

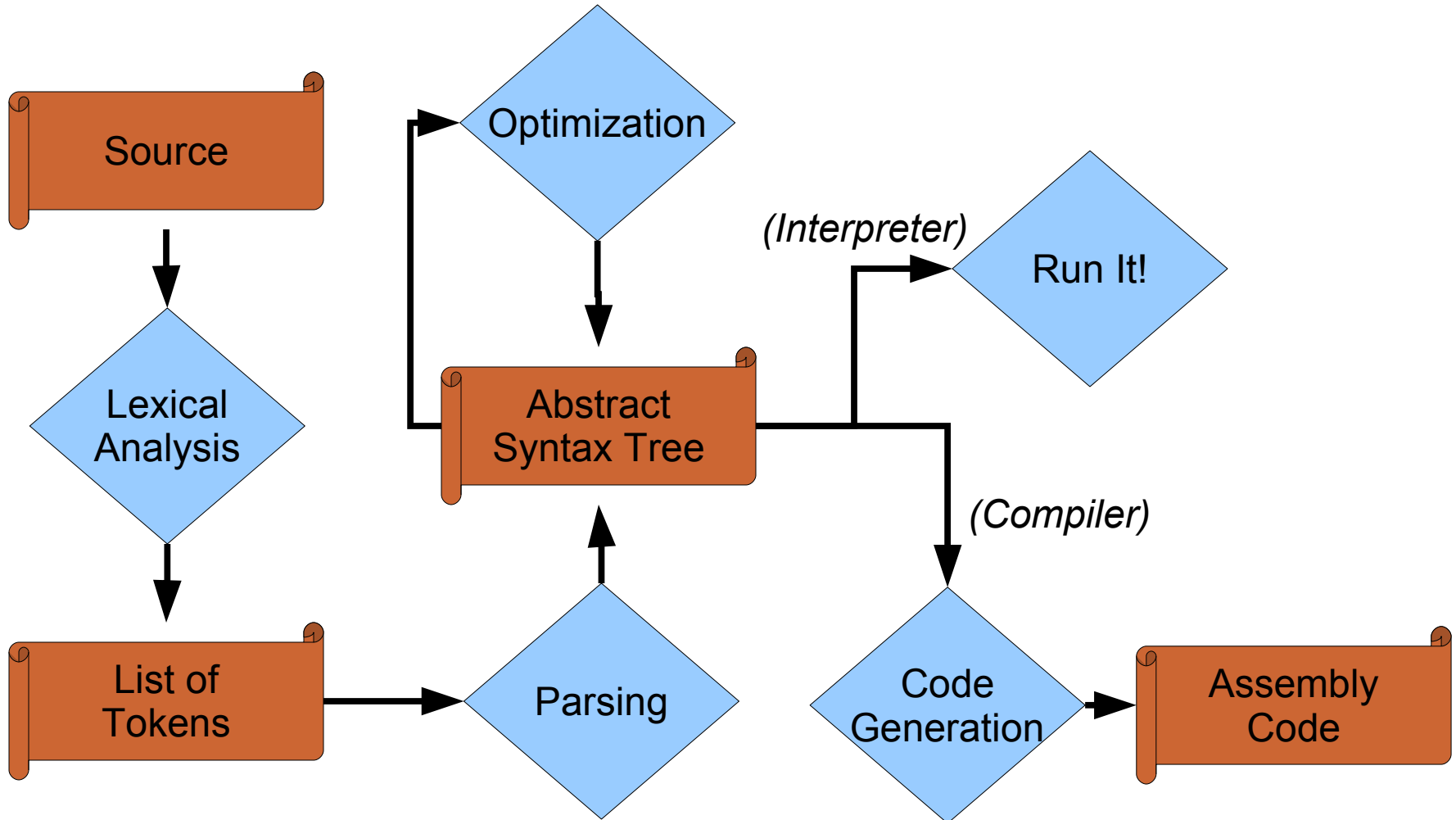
about a minute ago · Like · Comment



**Billy** You know you can post Polls on facebook now, right?  
IE, Always a little behind the times.

2 seconds ago · Like

# Interpreter and Compiler Structure



# Detail: Modern Reality

PA2

PA3

Regular Expressions

Context-Free Grammar

Lexical Analyzer Generator

Parser Generator

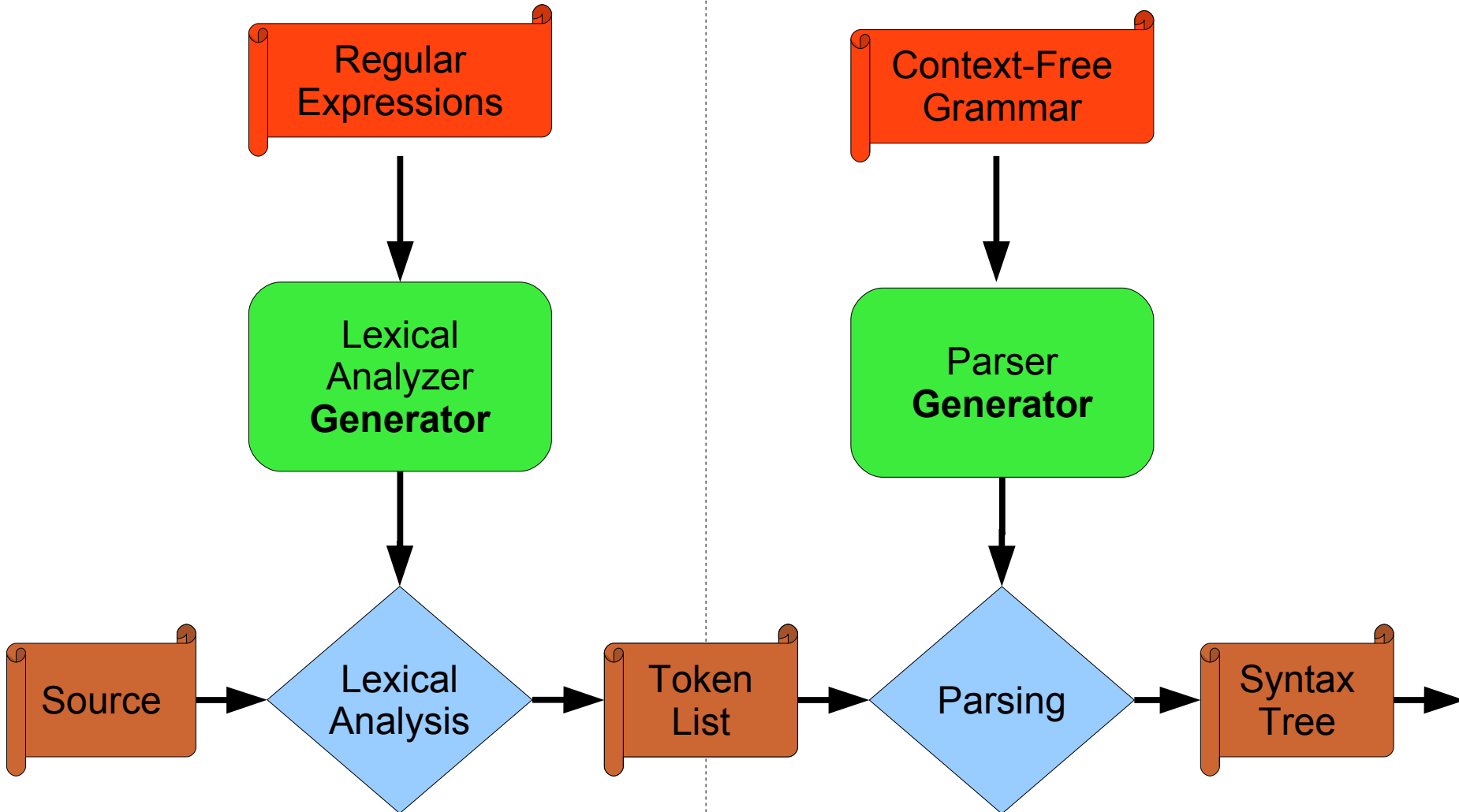
Source

Lexical Analysis

Token List

Parsing

Syntax Tree



# Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
    z = 0;
else
    z = 1;
```

- The input is just a sequence of characters:  

```
if (i == j)\n\tz = 0;\nelse\n\tz = 1;
```
- Goal: partition input strings into substrings
  - And **classify them** according to their role

# What's a Token?

- Output of lexical analysis is a list of tokens
- A **token** is a syntactic category
  - In English:
    - noun, verb, adjective, ...
  - In a programming language:
    - Identifier, Integer, Keyword, Whitespace, ...
- Parser relies on token distinctions:
  - e.g., identifiers are treated differently than keywords

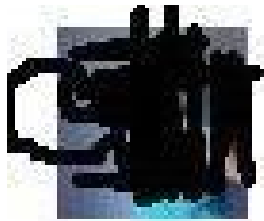


# Tokens

- Tokens correspond to **sets of strings**.
- **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/or tabs
- **OpenPar**: a left-parenthesis

# Lexical Analyzer: Build It!

- An implementation must do two things:
- Recognize substrings corresponding to tokens
- Return the value or lexeme of the token
  - The lexeme is the substring



[Redacted Name]

What have I done... :/



9 minutes ago via Mobile · Like · Comment



[Redacted Name] made a facebook status

6 minutes ago · Like · ↻ 2 people



[Redacted Name] Honestly shut up!

5 minutes ago · Like · ↻ 1 person

# Example

- Recall:

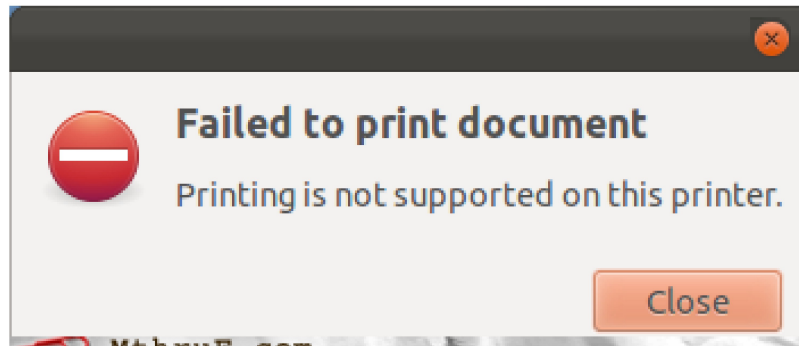
```
if (i == j)\n\tz = 0;\nelse\n\tz = 1;
```

- Token-lexeme pairs returned by the lexer:

- <Keyword, “if”>
- <Whitespace, “ ”>
- <OpenPar, “(”>
- <Identifier, “i”>
- <Whitespace, “ ”>
- <Relation, “==”>
- <Whitespace, “ ”>
- ...

# Lexical Analyzer: Implementation

- The lexer usually *discards* “uninteresting” tokens that don't contribute to parsing.
- Examples: Whitespace, Comments
  - Exception: which language cares about whitespace?
- Question: What happens if we remove all whitespace and comments *prior* to lexing?

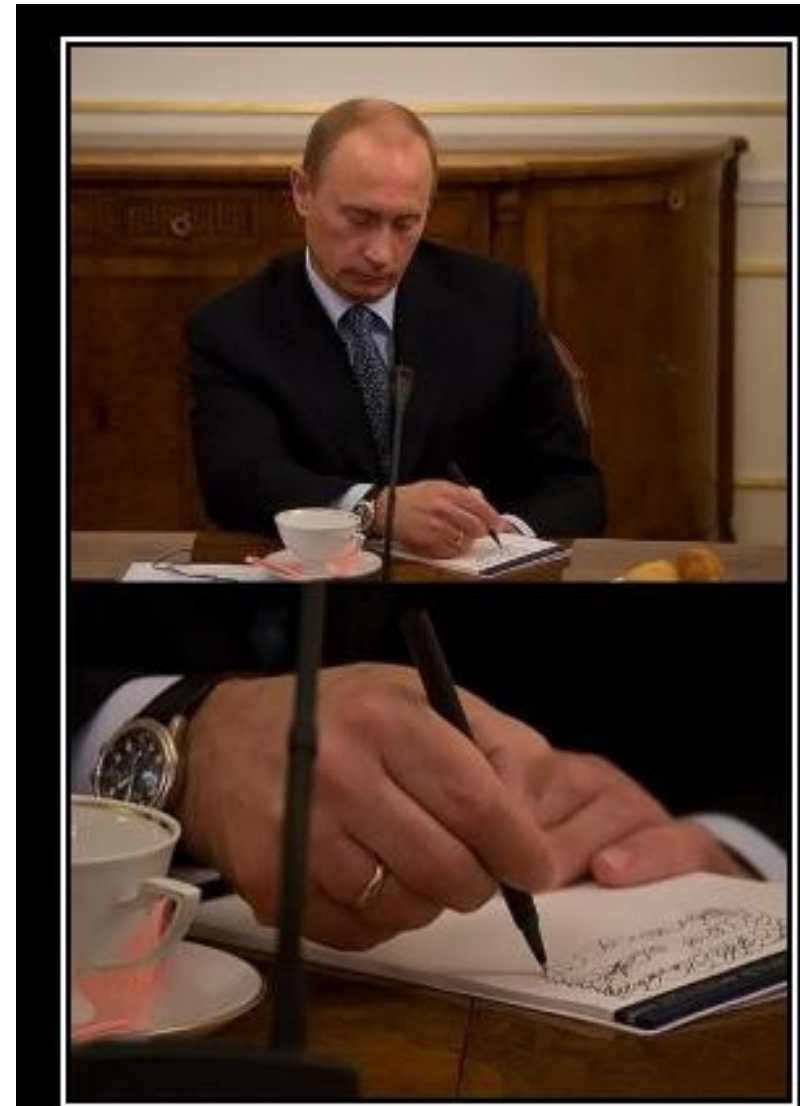


# Lookahead

- The goal is to partition the string. That is implemented by reading left-to-right, recognizing one token at a time.
- Lookahead may be required to decide where one token ends and the next token begins
  - Even our simple example has lookahead issues
  - **i** vs. **if**
  - **=** vs. **==**

# Still Needed

- A way to describe the lexemes of each token
  - Recall: lexeme = “the substring corresponding to the token”
- A way to resolve ambiguities
  - Is **if** two variables **i** and **f**?
  - Is **==** two equal signs **= =**?

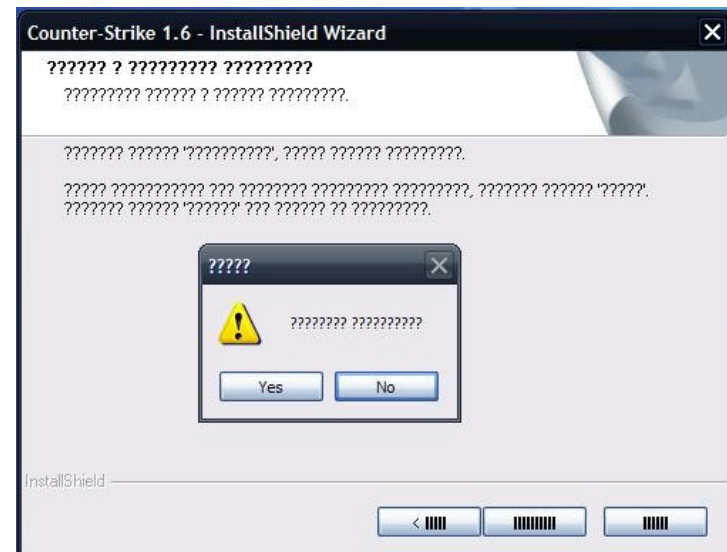


**FACT**

You never get away with pretend writing.

# Languages

- **Definition.** Let  $\Sigma$  be a set of characters. A language over  $\Sigma$  is a set of strings of characters drawn from  $\Sigma$ .  $\Sigma$  is called the alphabet.



# Examples of Languages

- Alphabet = English Characters
- Language = English Sentences
  - Note: *Not* every string on English characters is an English sentence.
  - Example: xayenb sbe'
- Alphabet = ASCII characters
- Language = C Programs
  - Note: ASCII character set is different from English character set.



# Notation

- Languages are sets of strings
- We need some notation for specifying which sets we want
  - that is, which strings are in the set
- For lexical analysis we care about *regular languages*, which can be described using *regular expressions*.

# Regular Expressions

- Each regular expression is a notation for a regular language (a set of words)
  - You'll see the exact notation in minute!
- If **A** is a regular expression then we write **L(A)** to refer to the language denoted by **A**

ReportManager



Error Retrieving values in VB App.

Either you didn't enter the data properly, or the developer for this screwed up royally. I'm leaning towards 'A'.

OK

# Base Regular Expression

- Single character: 'c'
  - $L('c') = \{ \text{"c"} \}$  (for any  $c \in \Sigma$ )
- Concatenation:  $AB$ 
  - $A$  and  $B$  are other regular expressions
  - $L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \}$
- Example:  $L('i' 'f') = \{ \text{"if"} \}$ 
  - We abbreviate 'i' 'f' as 'if'

# Compound Regular Expressions

- Union

- $L(A \mid B) = \{ s \mid s \in L(A) \text{ or } s \in L(B) \}$

- Examples:

- $L('if' \mid 'then' \mid 'else') = \{ \text{“if”}, \text{“then”}, \text{“else”} \}$

- $L('0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9') = \textit{what?}$

- Fun Example:

- $L( ('0' \mid '1') ('0' \mid '1') ) = \{ \text{“00”}, \text{“01”}, \text{“10”}, \text{“11”} \}$

# Starz!

- So far we have only finite languages
- Iteration:  $A^*$ 
  - $L(A^*) = \{“”\} \cup L(A) \cup L(AA) \cup L(AAA) \dots$
- Examples:
  - $L('0'^*) = \{“”, “0”, “00”, “000”, “0000”, \dots\}$
  - $L('1''0'^*) = \{“1”, “10”, “100”, “1000”, \dots\}$
- Empty:  $\epsilon$ 
  - $L(\epsilon) = \{ “” \}$

The screenshot shows the Microsoft Help and Support interface. At the top, there is a search bar with the text "Search Microsoft.com for:" and a "Go" button. Below the search bar, the text "Help and Support" is displayed. At the bottom of the screenshot, a bold error message is visible: "Error Message: Your Password Must Be at Least 18770 Characters and Cannot Repeat Any of Your Previous 30689 Passwords".

**Error Message: Your Password Must Be at Least 18770 Characters and Cannot Repeat Any of Your Previous 30689 Passwords**

Q: Advertising (810 / 842)

- The United States Forest Service's ursine mascot first appeared in 1944. Give his catchphrase safety message.

# Natural Languages

- These languages, of which there are about 250, are often mutually intelligible and constitute a major branch of the Niger-Congo languages. They are spoken largely in central, east and southern Africa. Popular examples include Swahili, with 80 million speakers, Shona, with 11 million, and Zulu, with 10 million. They commonly use words such as *muntu* or *mutu* for “person”. Words such as bongos, chimpanzee, gumbo, jumbo, mambo, rumba and safari come from these languages.

## Q: Music (150 / 842)

- In this 1958 Sheb Wooley song the pigeon-toed title character wears short shorts and wants to get a job in a rock'n'roll band playing the horn, but is perhaps best known for his skin tone and non-standard diet.



# Example: Keyword

- Keyword: “else” or “if” or “begin” or ...

'else' | 'if' | 'begin' | ...

(Recall: 'else' abbreviates 'e' 'l' 's' 'e')



You have entered invalid data in your Security Image

Please do not use any of the following characters or words: 'SELECT FROM' 'DELETE FROM' 'UPDATE SET' 'INSERT INTO' DROP NULL .. --

OK

# Example: Integers

- Integer: a non-empty string of digits

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

number = digit digit\*

Abbreviation:  $A^+ = A A^*$

# Example: Identifier

- Identifier: string of letters or digits, starting with a letter

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

ident = letter ( letter | digit )\*

Is (letter\* | digit\*) the same?

# Example: Whitespace

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

`(' ' | '\t' | '\n') +`

or

`(' ' | '\t' | '\n' | '\r') +`

# Example: Phone Numbers

- Regular expressions are everywhere!
- Consider: (434) 924-1021

$\Sigma$  = {0, 1, 2, 3, ..., 9, (, ), -}

area = digit digit digit

exch = digit digit digit

phone = digit digit digit digit

number = '(' area ')' exch '-' phone

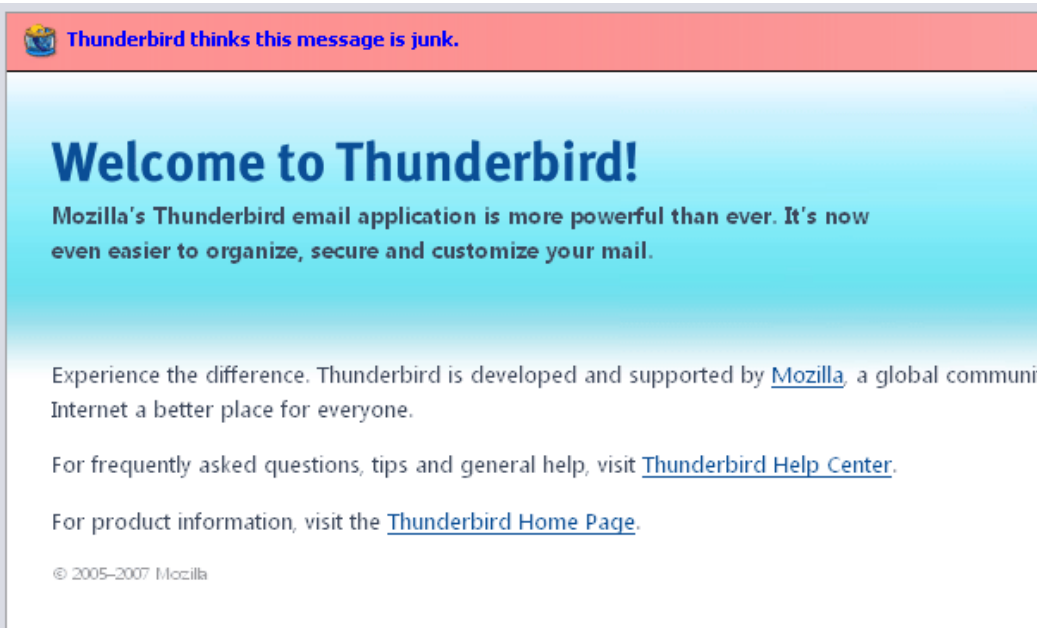
# Example: Email Addresses

- Consider `weimer@cs.virginia.edu`

$\Sigma$  = {a, b, ..., z, ., @}

name = letter+

address = name '@' name ('.' name)\*



# Regex Summary

- Regular expressions describe many useful languages
- Next: Given a string  $s$  and a regexp  $R$ , is  
 $s \in L(R)$
- But a yes/no answer is not enough!
- Instead: partition the input into lexemes
- We will adapt regular expression to this goal

# Subsequent Outline

- Specifying lexical structure using regexps
- Finite Automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of Regular Expressions
  - Regexp -> NFA -> DFA -> Tables
  - The tables are the heart of the lexer, which is just a while loop that takes in the current input character and looks up the new state in the transition table.



# Lexical Specification (1)

- Select a set of tokens
  - Number, Keyword, Identifier, ...
- Write a regexp for the lexemes of each token
  - Number = `digit+`
  - Keyword = `'if' | 'else' | ...`
  - Identifier = `letter ( letter | digit ) *`
  - OpenPar = `'('`
  - ...

# Lexical Specification (2)

- Construct  $R$ , matching all lexemes for all tokens:

$R = \text{Keyword} \mid \text{Identifier} \mid \text{Number} \mid \dots$

$R = R_1 \mid R_2 \mid R_3 \mid \dots$

- Fact: if  $s \in L(R)$  then  $s$  is a lexeme
  - Furthermore,  $s \in L(R_j)$  for some  $j$
  - This  $j$  determines the token that is reported

# Lexical Specification (3)

- Let the input be  $x_1 \dots x_n$ 
  - Each  $x_i$  is in the alphabet  $\Sigma$
- For  $1 \leq i \leq n$ , check
  - $x_1 \dots x_i \in L(R)$
- If so, it must be that
  - $x_1 \dots x_i \in L(R_j)$  for some  $j$
- Remove  $x_1 \dots x_i$  from the input and restart

# Lexing Example

- R = Whitespace | Integer | Identifier | Plus
- Parse “f +3 +g”
  - “f” matches R, more precisely Identifier
  - “ “ matches R, more precisely Whitespace
  - “+” matches R, more precisely Plus
  - ...
  - The token-lexeme pairs are
    - <Identifier, “f”>
    - <Whitespace, “ “>
    - <Plus, “+”> ...



In the future, we'll just drop whitespace.

# Ambiguities (1)

- There are ambiguities in the algorithm
- Example:
  - $R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid \text{Plus}$
- Parse “foo+3”
  - “f” matches R, more precisely Identifier
  - But also “fo” matches R, and “foo”, but not “foo+”
- How much input is used?
  - Maximal Munch rule: Pick the longest possible substring that matches R

# Ambiguities (2)

- $R = \text{Whitespace} \mid \text{'new'} \mid \text{Integer} \mid \text{Identifier}$
- Parse “new foo”
  - “new” matches  $R$ , more precisely 'new'
  - but also Identifier - which one do we pick?
- In general, use the rule listed **first**.
  - No, really.
- So we must list 'new' (and other keywords) before Identifier.

# Error Handling

- $R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid '+'$
- Parse “=56”
  - No prefix matches R: not “=”, nor “=5”, nor “=56”
- Problem: we can't just get stuck and die
- Solution:
  - Add a rule matching all “bad” strings
  - Put it last
- Lexer tools allow the writing of:
  - $R = R1 \mid R2 \mid \dots \mid Rn \mid \text{Error}$

# Summary



- Regular expressions provide a concise notation for **string patterns**
- Their use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Requiring only a single pass over the input
  - And few operations per character (table lookup)