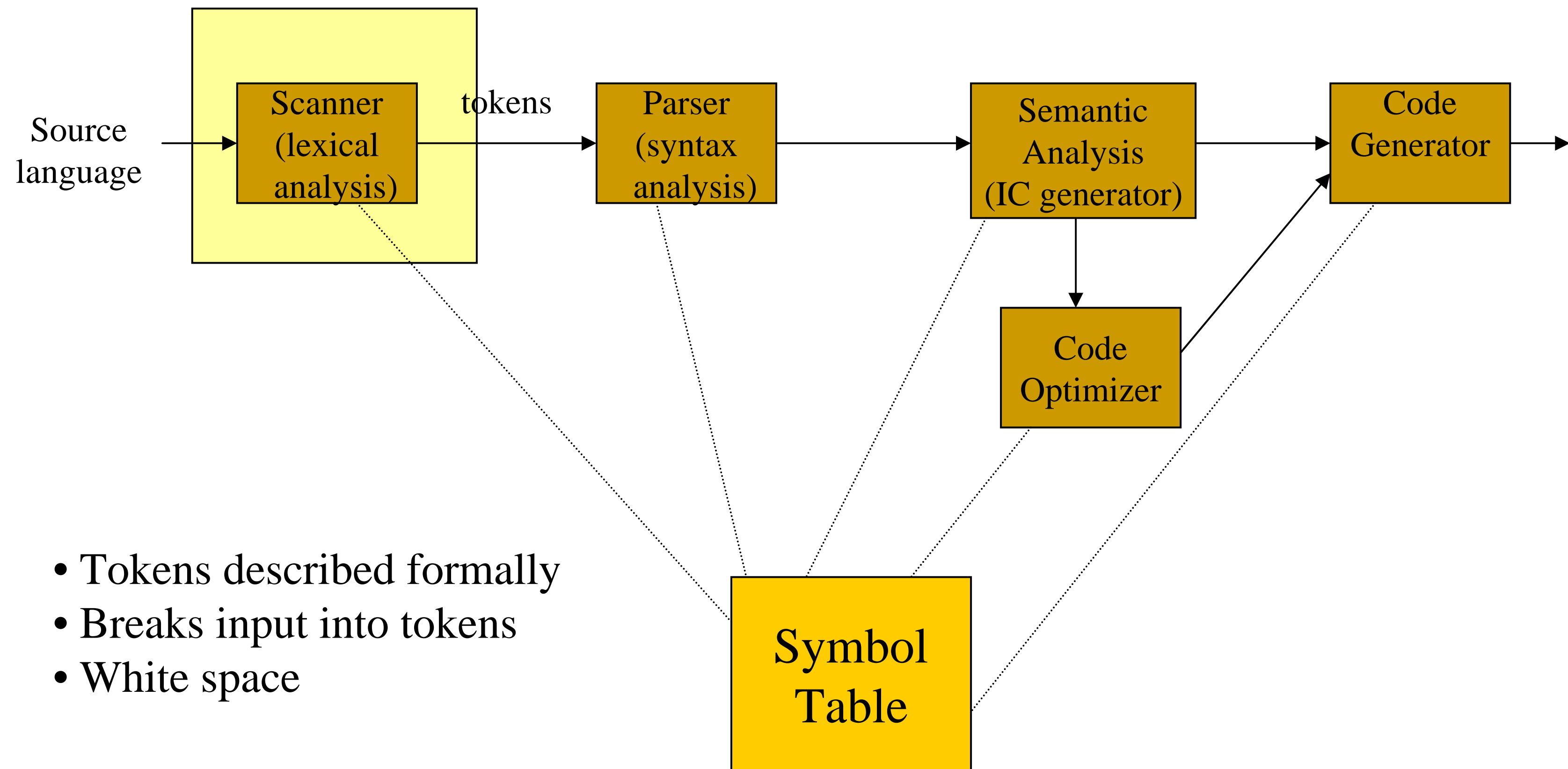
Lecture 2: Lexical Analysis & Lex Tool

長庚大學資訊工程學系　陳仁暉　助理教授
Tel: (03) 211-8800 Ext: 5990
Email: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/~jhchen

© All rights reserved. No part of this publication and file may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of Professor Jenhui Chen (E-mail: jhchen@mail.cgu.edu.tw).

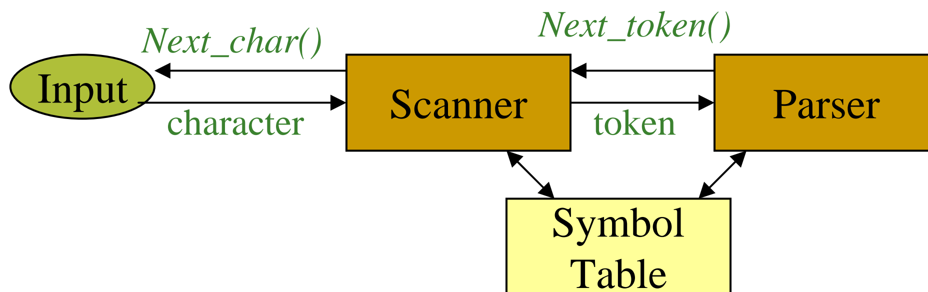
Lexical Analysis - Scanning



Lexical Analysis

INPUT: sequence of characters

OUTPUT: sequence of tokens



A lexical analyzer is generally a subroutine of parser:

- Simpler design
- Efficient
- Portable

Definitions

- **token** – set of strings defining an atomic element with a defined meaning
- **pattern** – a rule describing a set of string
- **lexeme** – a sequence of characters that match some pattern

Examples

<table>
<tr><th>Token</th><th>Pattern</th><th>Sample Lexeme</th></tr>
<tr><td>while</td><td>while</td><td>while</td></tr>
<tr><td>relation_op</td><td>= | != | < | ></td><td><</td></tr>
<tr><td>integer</td><td>(0-9)+</td><td>42</td></tr>
<tr><td>string</td><td>Characters between “ “</td><td>“I am here”</td></tr>
</table>

Input string: size := r * 32 + c

<token,lexeme> pairs:

- <id, size>
- <assign, :=>
- <id, r>
- <arith_symbol, *>
- <integer, 32>
- <arith_symbol, +>
- <id, c>

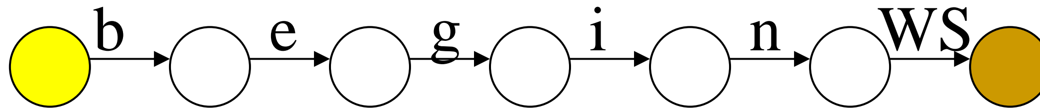
Implementing a Lexical Analyzer

Practical Issues:

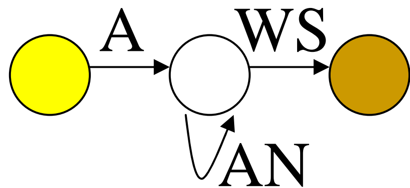
- Input buffering
- Translating RE into executable form
- Must be able to capture a large number of tokens with single machine
- Interface to parser
- Tools

Capturing Multiple Tokens

Capturing keyword "begin"



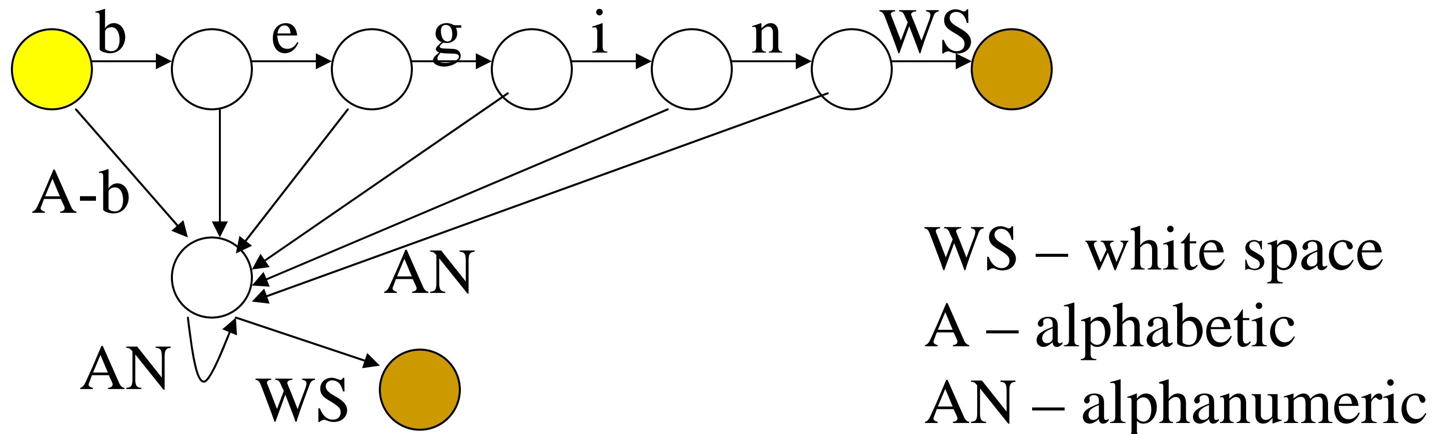
Capturing variable names



WS – white space
A – alphabetic
AN – alphanumeric

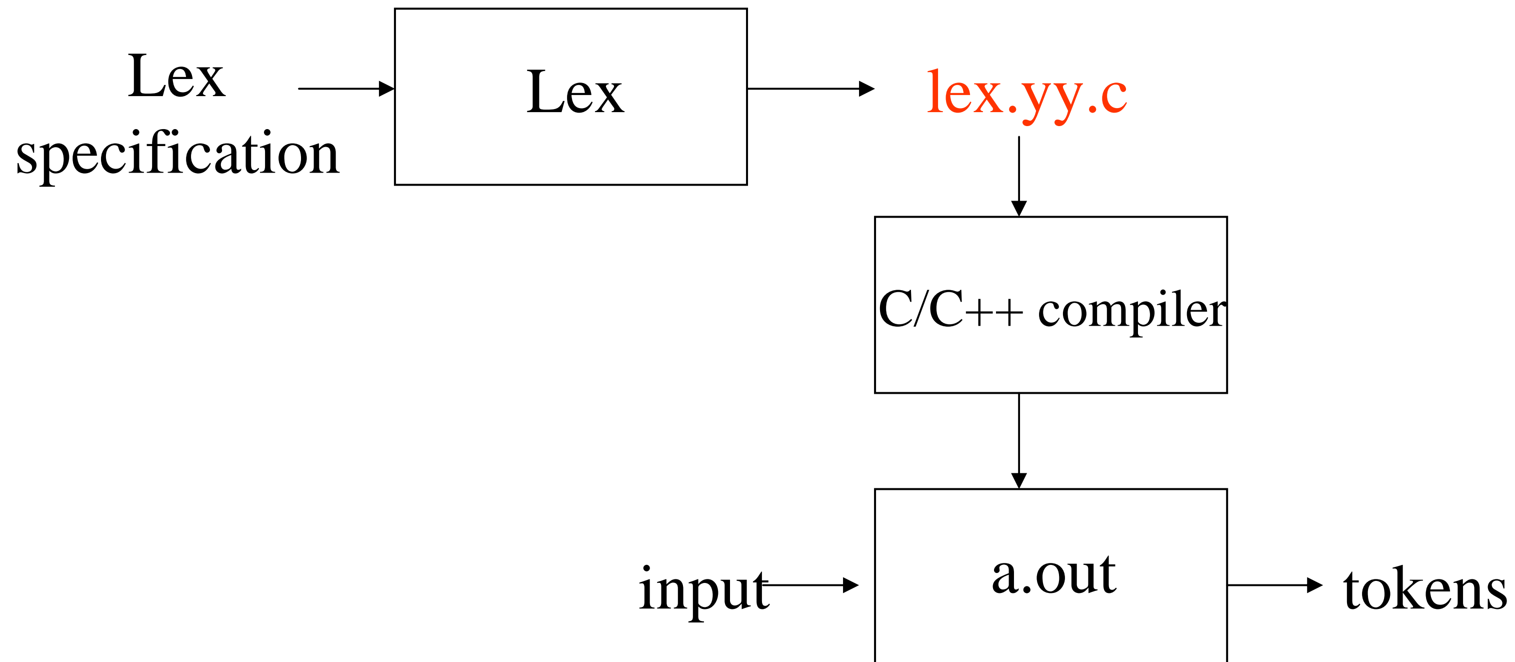
What if both need to happen at the same time?

Capturing Multiple Tokens



Machine is much more complicated – just for these two tokens!

Lex – Lexical Analyzer Generator



Lex Specification

```
%{ int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]
%%
{word}     {wordCount++; charCount += yyleng; }
[\n] {charCount++; lineCount++;}
.              {charCount++;}
%%
main() {
   yylex();
   printf("Characters %d, Words: %d, Lines: %d\n",charCount,
wordCount, lineCount);
}
```

Definitions – Code, RE

Rules – RE/Action pairs

User Routines

A Lex file

… define …
%%
… rules …
%%
… code …

```
%{
 int charcount=0,linecount=0;
%}

%%

. charcount++;
\n {linecount++; charcount++;}

%%
int main()
{
  yylex();
  printf("There were %d characters in %d lines\n",
          charcount,linecount);
  return 0;
}
```

Lex definitions section

```
%{ int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]
```

- C/C++ code:
  - Surrounded by %{… %} delimiters
  - Declare any variables used in actions
- RE definitions:
  - Define shorthand for patterns:

    digit   [0-9]

    letter   [a-z]

    *ident   {letter}({letter}|{digit})**
  - Use shorthand in RE section: {*ident*}

Lex Regular Expressions

```
{word}    {wordCount++; charCount += yyleng; }
[\n]      {charCount++; lineCount++;}
.                   {charCount++;}
```

- Match explicit character sequences
  - integer, “+++”, \<\>
- Character classes
  - [abcd]
  - [a-zA-Z]
  - [^0-9] – matches non-numeric

- Alternation
  - twelve | 12
- Closure
  - * - zero or more
  - + - at least one or more
  - ? – zero or one
  - {*number*}, {*number*,*number*}

Character	**Meaning**
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches *any* character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*****	Match *zero* or more occurrences of the preceding pattern.
+	Matches *one* or more occurrences of the preceding pattern.
?	Matches *zero or one* occurrences of the preceding pattern.
$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.
****	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation.
|	Logical OR between expressions.
"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	Groups a series of regular expressions.

Lex Matching Rules

- Lex *always* attempts to match the longest possible string.
- If two rules are matched (and match strings are same length), the first rule in the specification is used.

Lex Operators

Highest: closure

concatenation

alternation

Special lex characters:

- \ / * + > “ { } . $ ( ) | % [ ] ^

Special lex characters inside [ ]:

- \ [ ] ^

Examples

- joke[rs] ➔ matches {joker, jokes}
- A{1,2}lias? ➔ {Alias, AAlias, Alia, AAlia}
- a.*z ➔ {az, a!z, a#z, a.z, a..z, aaz, aaaz, …}
- (ab)+ ➔ {ab, abab, ababab, …}
- [0—9]{1,5} ➔ { 0, 1, …, 9, 00001, …, 99999}
- (ab|cd)?ef ➔ {abef, cdef, ef}
- -?[0-9]\.[0-9]

Lex Actions

Lex actions are C (C++) code to implement some required functionality

- Default action is to echo to output
- Can ignore input (empty action)
- ECHO – macro that prints out matched string
- *yytext* – matched string
- *yyleng* – length of matched string

User Subroutines

```
main() {
    yylex();
    printf("Characters %d, Words: %d, Lines: %d\n",charCount,
wordCount, lineCount);
}
```

- C/C++ code
- Copied directly into the lexer code
- User can supply ‘main’ or use default

Lex

- Lex always creates a file 'lex.yy.c' with a function yylex()
- -ll directs the compiler to link to the lex library
- The lex library supplies external symbols referenced by the generated code
- The lex library supplies a default main:

  main(int ac,char **av) {return yylex(); }

Lex Example: Extracting white space

```
%{
int yylex(void);  // make C++ happy
%}
%%
[ \t\n]                  ;
.                        {ECHO;}
%%
```

To compile and run above (example.l):

```
lex example.l                      flex simple.l
cc lex.yy.c –o first -ll
gcc lex.yy.c –ll                   g++ -x c++ lex.yy.c –ll
a.out < input
```

Input:

This is a file

of stuff we want   to extract all

white space from

Output:

Thisisafileofstuffwewantoextractallwhitespacefrom

Lex Example 2: Unix wc

```
%{ int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]
%%
{word}   {wordCount++; charCount += yyleng; }
[\n]{charCount++; lineCount++;}
.          {charCount++;}
%%
main() {
   yylex();
   printf("Characters %d, Words: %d, Lines: %d\n",charCount, wordCount,
    lineCount);
}
```

Lex Example 3: Extracting tokens

```
%%
and                     return(AND);
array                   return(ARRAY);
begin                   return(BEGIN);
…
\[                      return(‘[‘);
“:=“                    return(ASSIGN);
[a-zA-Z][a-zA-Z0-9_]*   return(ID);
[+-]?[0-9]+             return(NUM);
[ \t\n]                 ;
%%
```

Uses for Lex

- **Transforming Input** – convert input from one form to another (example 1). *yylex*() is called once; return is not used in specification
- **Extracting Information** – scan the text and return some information (example 2). *yylex*() is called once; return is not used in specification.
- **Extracting Tokens** – standard use with compiler (example 3). Uses return to give the next token to the caller.

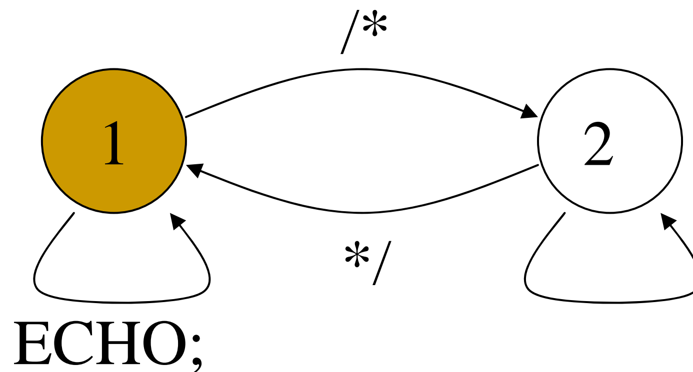
Lex States

- Regular expressions are compiled to state machines.
- Lex allows the user to explicitly declare multiple states.

  %s COMMENT
- Default initial state INITIAL (0)
- Actions for matched strings may be different for different states

Lex State Example

Problem: Want to discard comments surrounded by /*… */ from the input.



Lex State Example

Discard comments surrounded by /*… */ from the input.

```
%%
<INITIAL>.                    {ECHO;}
<INITIAL>"/*"          {BEGIN COMMENT;}
<COMMENT>.             ;
<COMMENT>"*/"                 {BEGIN INITIAL;}
%%
```