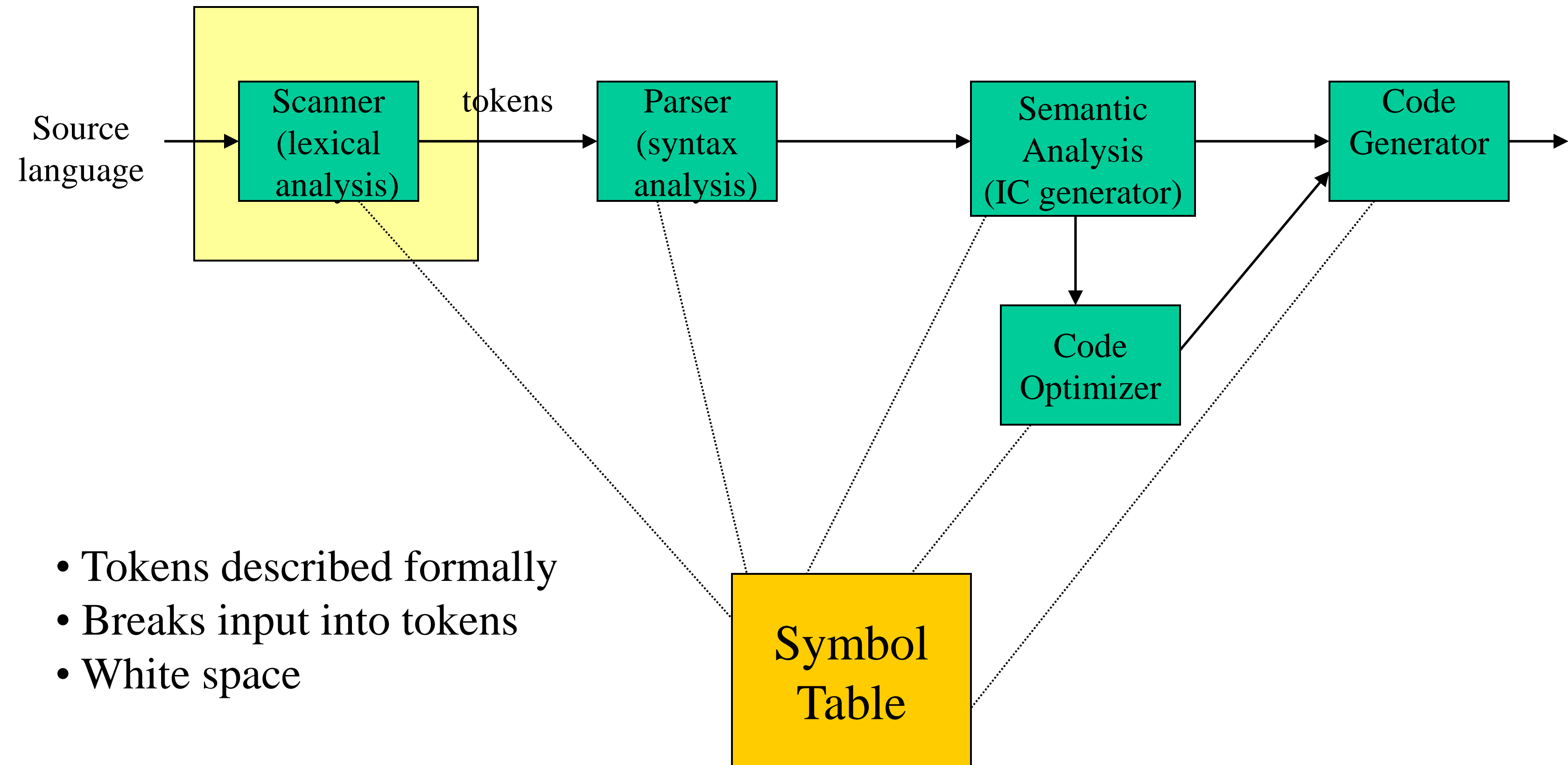
Lecture 2: Lexical Analysis

CS 540

George Mason University

Lexical Analysis - Scanning

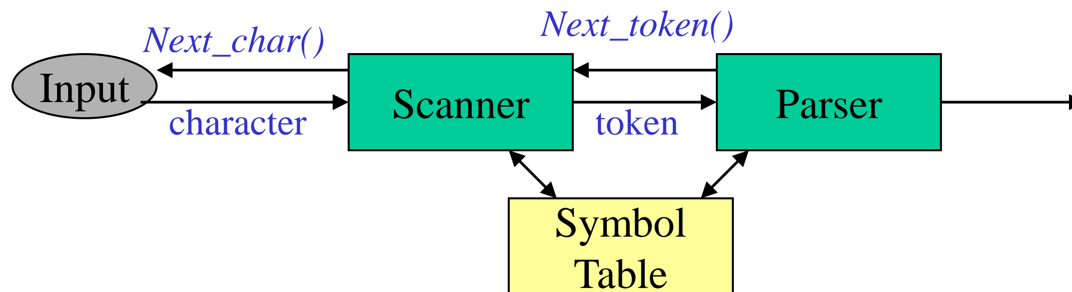


- Tokens described formally
- Breaks input into tokens
- White space

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>“hello”</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>

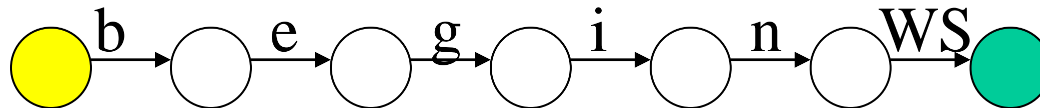
Implementing a Lexical Analyzer

Practical Issues:

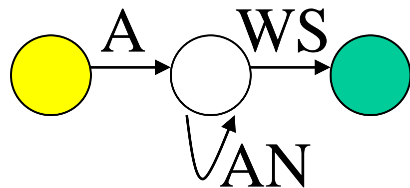
- Input buffering
- Translating RE into executable form
- Must be able to capture a large number of tokens with single machine
- Interface to parser
- Tools

Capturing Multiple Tokens

Capturing keyword “begin”



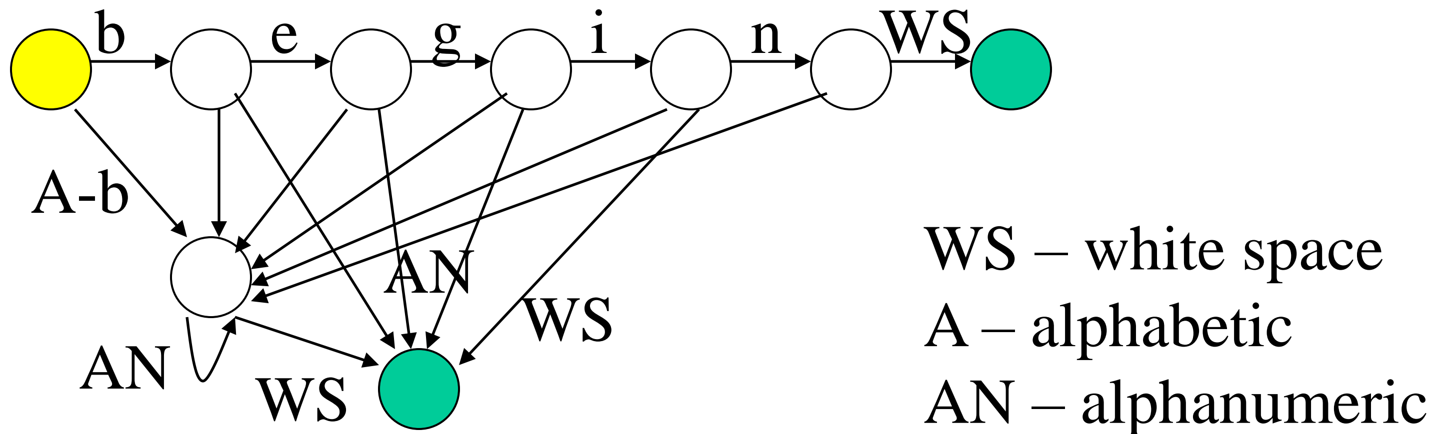
Capturing variable names



WS – white space
A – alphabetic
AN – alphanumeric

What if both need to happen at the same time?

Capturing Multiple Tokens

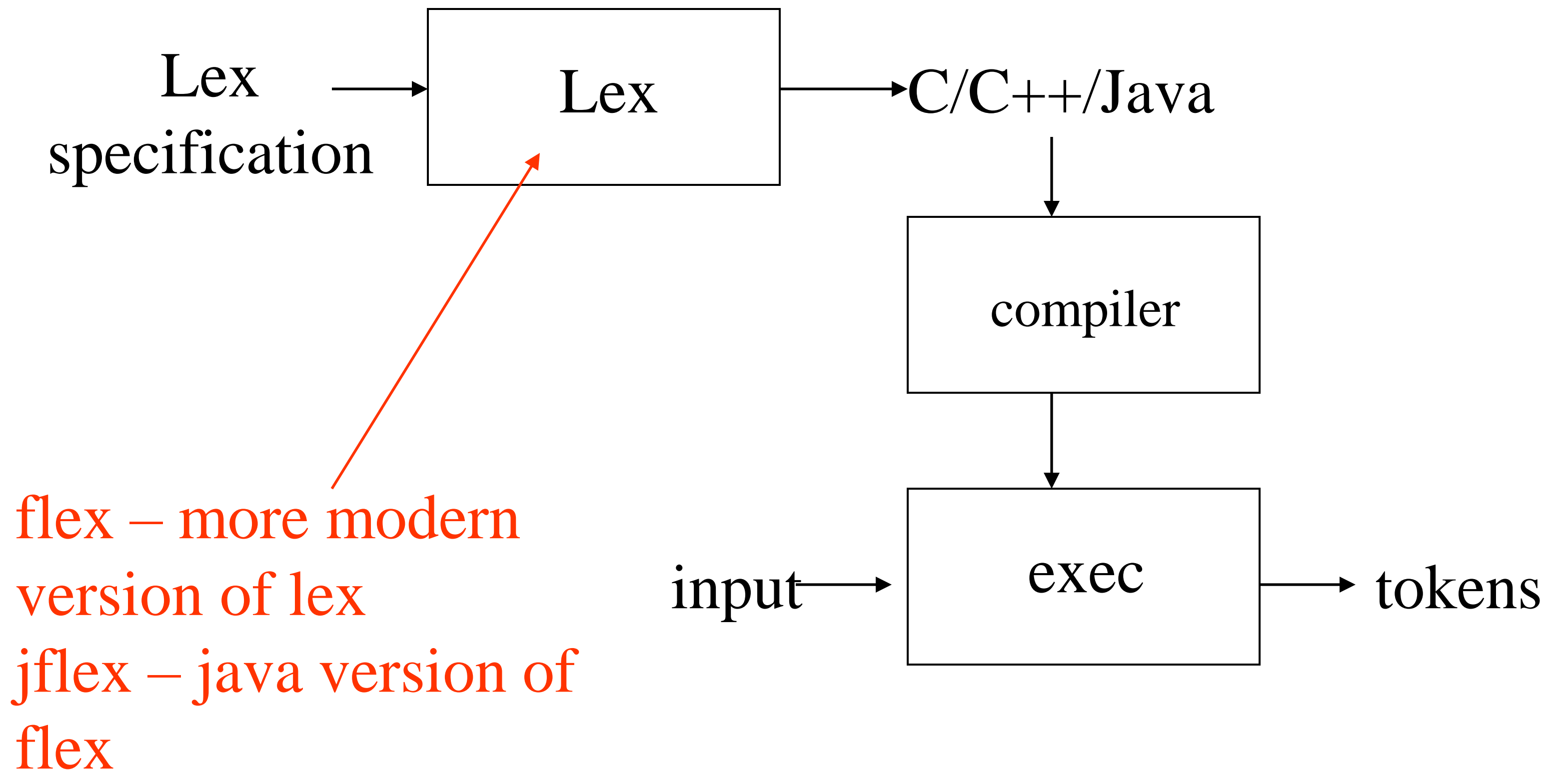


Machine is much more complicated – just for these two tokens!

Real lexer (handcoded)

- http://cs.gmu.edu/~white/CS540/lexer.cpp
- Comes from C# compiler in Rotor
- >950 lines of C++

Lex – Lexical Analyzer Generator



Lex Specification (flex)

```
%{
int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]+
%%
{word}      {wordCount++; charCount += strlen(yytext); }
[\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

Lex Specification (jflex)

```
import java.io.*;
%%
%class ex1
%unicode
%line
%column
%standalone
%{
static int charCount = 0, wordCount = 0, lineCount = 0;
public static void main(String [] args) throws IOException
{
    ex1 lexer = new ex1(new FileReader(args[0]));
    lexer.yylex();
    System.out.println("Characters: " + charCount +
            " Words: " + wordCount +" Lines: " +lineCount);
}
%}
%type Object    //this line changes the return type of yylex into Object
word = [^ \t\n]+
%%
{word}      {wordCount++; charCount += yytext().length(); }
[\n]        {charCount++; lineCount++; }
.           {charCount++; }
```

Definitions – Code, RE

Rules – RE/Action pairs

Lex definitions section

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

- C/C++/Java 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 += strlen(yytext); }
[\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
  – + - one or more
  – ? – zero or one
  – {*number*}, {*number*,*number*}

- Other operators
  - . – matches any character except newline
  - ^ - matches beginning of line
  - $ - matches end of line
  - / - trailing context
  - () – grouping
  - { } – RE definitions

Lex Operators

Highest: closure

concatenation

alternation

Special lex characters:

- \ / * + > “ { } . $ ( ) | % [ ] ^

Special lex characters inside [ ]:

- \ [ ] ^

Examples

- a.*z
- (ab)+
- [0-9]{1,5}
- (ab|cd)?ef = *abef,cdef,ef*
- -?[0-9]\.[0-9]

Lex Actions

Lex actions are C (C++, Java) 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 (not all versions have this)

In Java:
yytext() and
yytext().length()

User Subroutines

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

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

How Lex works

Lex works by processing the file one character at a time, trying to match a string starting from that character.

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

Once it matches a string, it starts from the character after the string

Lex Matching Rules

1. Lex *always* attempts to match the longest possible string.

```
beg     {…}
begin   {…}
in      {…}
```

Input ‘begin’ can match either of the first two rules.
The second rule will be chosen because of the length.

Lex Matching Rules

2. If two rules are matched (the matched strings are same length), the first rule in the specification is used.

```
begin       {… }
[a-z]+      {…}
```

Input ‘begin’ can match both rules – the first one will be chosen

Lex Example: Extracting white space

```
%{
#include <stdio.h>
%}
%%
[ \t\n]          ;
.                {ECHO;}
%%
```

To compile and run above (simple.l):

flex simple.l
gcc lex.yy.c –ll
a.out < input

flex simple.l
g++ -x c++ lex.yy.c –ll
a.out < input

-lfl on some systems

Lex Example: Extracting white space (Java)

```
%%
%class ex0
%unicode
%line
%column
%standalone
%%
[^ \t\n]       {System.out.print(yytext());}
.                       {}
[\n]                    {}
```

name of class to build

To compile and run above (simple.l):

```
java -jar ~cs540/JFlex.jar simple.l
javac ex0.java
java ex0 inputfile
```

Input:

This is a file

of stuff we want   to extract all

white space from

Output:

Thisisafileofstuffwewantoextractallwhitespacefrom

Lex (C/C++)

- Lex always creates a file 'lex.yy.c' with a function yylex()
- -ll directs the compiler to link to the lex library (-lfl on some systems)
- 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 2: Unix wc

```
%{ int charCount=0, wordCount=0, lineCount=0;
%}
word   [^ \t\n]+
%%
{word}   {wordCount++; charCount += strlen(yytext); }
[\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 States

```
%{
int ctr = 0;
int linect = 1;
%}
%s COMMENT
%%
<INITIAL>.              ECHO;
<INITIAL>[\n]           {linect++; ECHO;}
<INITIAL>"/*"           {BEGIN COMMENT; ctr = 1;}
<COMMENT>.              ;
<COMMENT>[\n]           linect++;
<COMMENT>"*/"           {if (ctr == 1) BEGIN INITIAL;
                          else ctr--;
                        }
<COMMENT>"/*"           {ctr++;}
%%
```