# Compilers

## *CS414-2017S-01*

## *Compiler Basics & Lexical Analysis*

David Galles

Department of Computer Science
University of San Francisco

# 01-0: Syllabus

- Office Hours
- Course Text
- Prerequisites
- Test Dates & Testing Policies
- Projects
  - Teams of up to 2
- Grading Policies
- Questions?

# 01-1: Notes on the Class

- I tend to talk quickly. Don't be afraid to ask me to slow down!
- We will cover some pretty complex stuff here, which can be difficult to get the first (or even the second) time. *ASK QUESTIONS*
- While specific questions are always preferred, "I don't get it" is always an acceptable question. I am always happy to stop, re-explain a topic in a different way.
  - If you are confused, I can *guarantee* that at least 1 other person in the class would benefit from more explanation

# 01-2: Notes on the Class

- Projects are non-trivial
  - Using new tools (lex, yacc)
  - Managing a large scale project
  - Lots of pointer chasing, with large, complex data structures

# 01-3: Notes on the Class

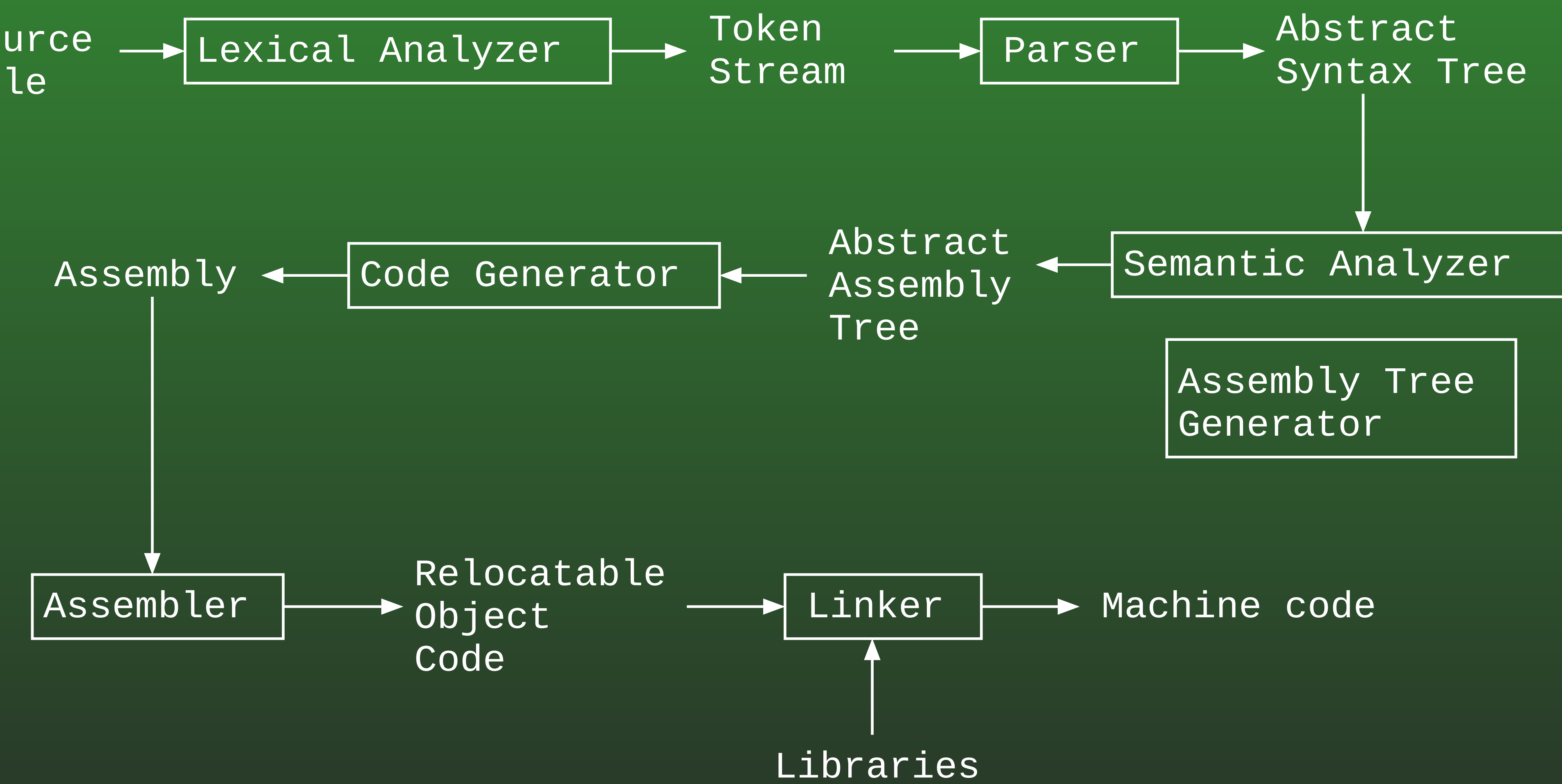
- Projects are non-trivial
  - Using new tools (lex, yacc)
  - Managing a large scale project
  - Lots of pointer chasing, with large, complex data structures
- *START EARLY!*
  - Projects will take longer than you think (especially starting with the semantic analyzer project)
- *ASK QUESTIONS!*

# 01-4: What is a compiler?

```
ource Program  ──→  [ Compiler ]  ──→  Machine code
```

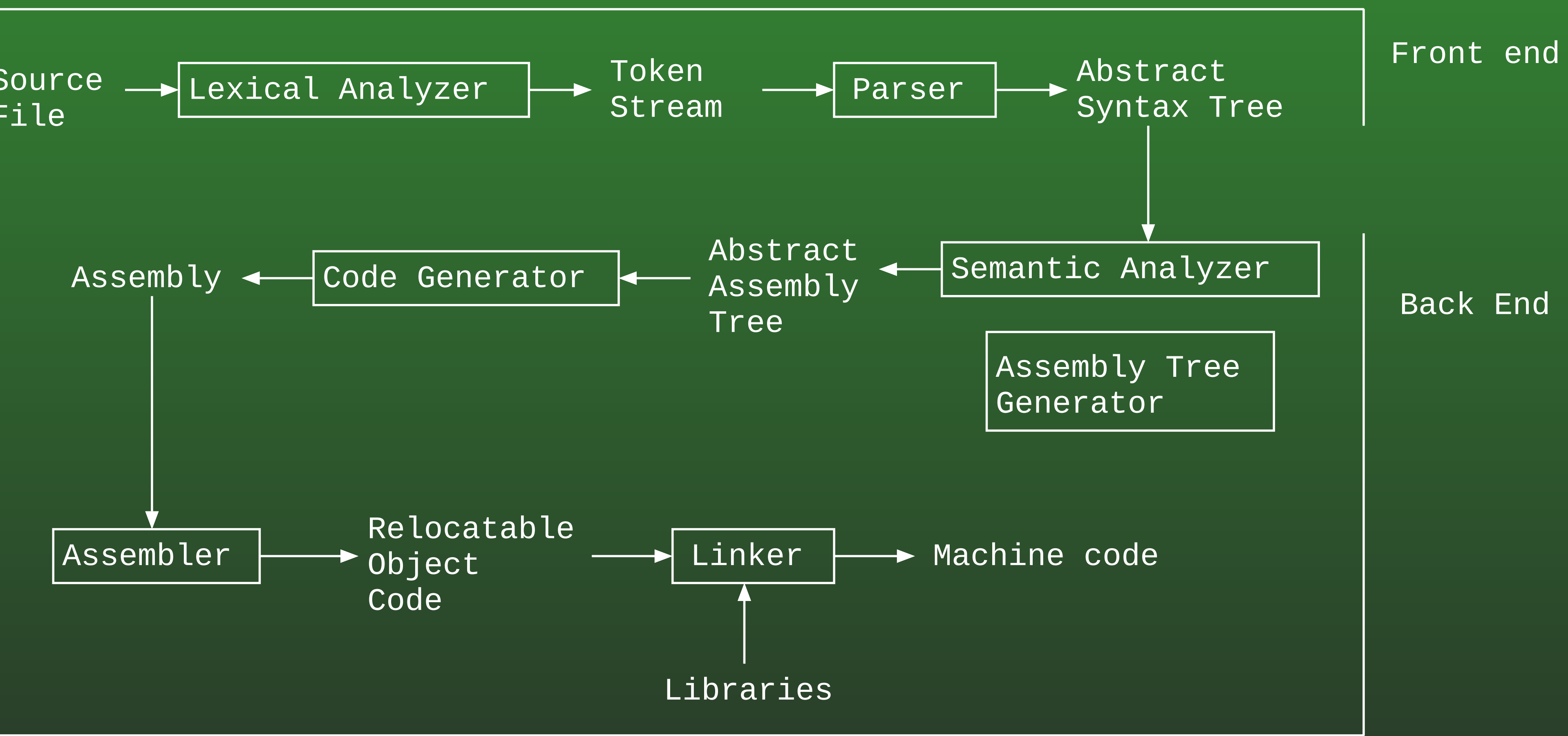
mplified View

# 01-5: What is a compiler?

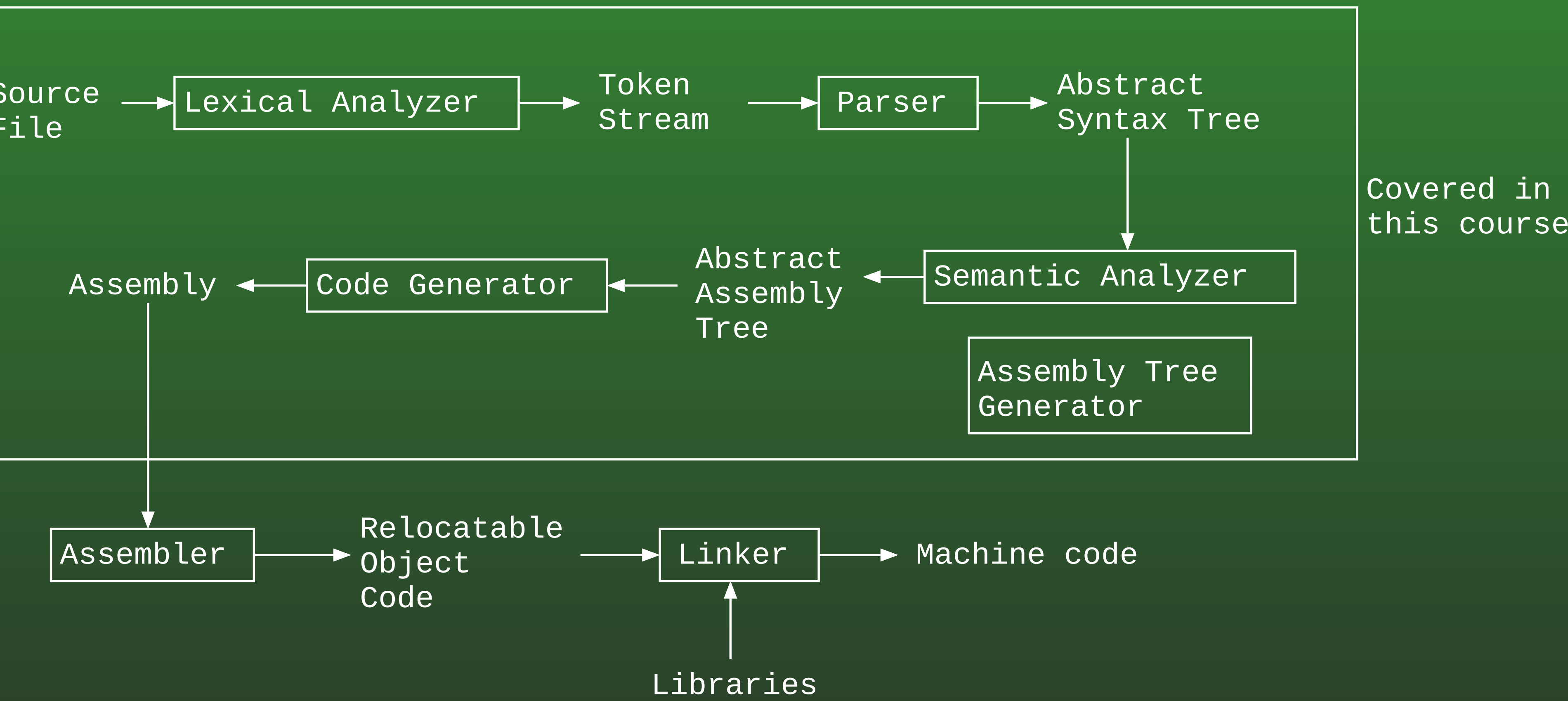



ore Accurate View

# 01-6: What is a compiler?



# 01-7: What is a compiler?



# 01-8: Why Use Decomposition?

# 01-9: Why Use Decomposition?

Software Engineering!

- Smaller units are easier to write, test and debug
- Code Reuse
  - Writing a suite of compilers (C, Fortran, C++, etc) for a new architecture
  - Create a new language – want compilers available for several platforms

# 01-10: Lexical Analysis

- Converting input file to stream of tokens

```
void main() {
  print(4);
}
```

# 01-11: Lexical Analysis

- Converting input file to stream of tokens

```
void main() {
  print(4);
}
```

```
IDENTIFIER(void)
IDENTIFIER(main)
LEFT-PARENTHESIS
RIGHT-PARENTHESIS
LEFT-BRACE
IDENTIFIER(print)
LEFT-PARENTHESIS
INTEGER-LITERAL(4)
RIGHT-PARENTHESIS
SEMICOLON
RIGHT-BRACE
```

# Lexical Analysis

## Brute-Force Approach

- Lots of nested if statements

```
if (c = nextchar() == 'P') {
   if (c = nextchar() == 'R') {
      if (c = nextchar() == 'O') {
         if (c = nextchar() == 'G') {
           /*  Code to handle the rest of either
               PROGRAM or any identifier that starts
               with PROG
           */
         } else if (c == 'C') {
           /*  Code to handle the rest of either
               PROCEDURE or any identifier that starts
               with PROC
           */

       ...
```

# 01-13: Lexical Analysis

Brute-Force Approach

- Break the input file into words, separated by spaces or tabs
  - This can be tricky – not all tokens are separated by whitespace
  - Use string comparison to determine tokens

# 01-14: Deterministic Finite Automata

- Set of states
- Initial State
- Final State(s)
- Transitions

DFA for else, end, identifiers

Combine DFA

# 01-15: DFAs and Lexical Analyzers

- Given a DFA, it is easy to create C code to implement it
- DFAs are easier to understand than C code
  - Visual – almost like structure charts
- ... However, creating a DFA for a complete lexical analyzer is still complex

# 01-16: Automatic Creation of DFAs

We'd like a tool:

- Describe the tokens in the language
- Automatically create DFA for tokens
- Then, automatically create C code that implements the DFA

We need a method for describing tokens

# 01-17: Formal Languages

- **Alphabet** $\Sigma$: Set of all possible symbols (characters) in the input file
  - Think of $\Sigma$ as the set of symbols on the keyboard
- **String** $w$: Sequence of symbols from an alphabet
- **String length** $|w|$ Number of characters in a string: |car| = 3, |abba| = 4
  - **Empty String** $\epsilon$: String of length 0: $|\epsilon| = 0$
- **Formal Language**: Set of strings over an alphabet

Formal Language $\neq$ Programming language – Formal Language is only a set of strings.

# 01-18: Formal Languages

Example formal languages:

- Integers $\{0, 23, 44, \ldots\}$
- Floating Point Numbers $\{3.4, 5.97, \ldots\}$
- Identifiers {foo, bar, . . .}

# 01-19: Language Concatenation

- **Language Concatenation** Given two formal languages $L_1$ and $L_2$, the concatenation of $L_1$ and $L_2$, $L_1L_2 = \{xy|x \in L_1, y \in L_2\}$

For example:
{fire, truck, car} {car, dog} =
{firecar, firedog, truckcar, truckdog, carcar, cardog}

# 01-20: Kleene Closure

Given a formal language $L$:

$$
\begin{array}{rcl}
L^0 & = & \{\epsilon\} \\
L^1 & = & L \\
L^2 & = & LL \\
L^3 & = & LLL \\
L^4 & = & LLLL
\end{array}
$$

$$L^* = L^0 \bigcup L^1 \bigcup L^2 \bigcup \ldots \bigcup L^n \bigcup \ldots$$

# 01-21: Regular Expressions

Regular expressions are use to describe formal languages over an alphabet $\Sigma$:

| Regular Expression | Language |
|---|---|
| $\epsilon$ | $L[\epsilon] = \{\epsilon\}$ |
| $\mathtt{a} \in \Sigma$ | $L[\mathtt{a}] = \{\mathtt{a}\}$ |
| $(MR)$ | $L[MR] = L[M]L[R]$ |
| $(M\|R)$ | $L[(M\|R)] = L[M] \bigcup L[R]$ |
| $(M*)$ | $L[(M*)] = L[M]*$ |

# 01-22: r.e. Precedence

From highest to Lowest:

Kleene Closure *

Concatenation

Alternation |

ab*c|e = (a(b*)c) | e

# 01-23: Regular Expression Examples

| | |
|---|---|
| (a\|b)* | all strings over {a,b} |
| (0\|1)* | binary integers (with leading zeroes) |
| a(a\|b)*a | all strings over {a,b} that begin and end with a |
| (a\|b)*aa(a\|b)* | all strings over {a,b} that contain aa |
| b*(abb*)*(a\|$\epsilon$) | all strings over {a,b} that do not contain aa |

# 01-24: r.e. Shorthand

| | | |
|---|---|---|
| [abcd] | = | (a\|b\|c\|d) |
| [b-g] | = | [befg] = (b\|e\|f\|g) |
| [b-gM-O] | = | [befgMNO] = (b\|e\|f\|g\|M\|N\|O) |
| M? | = | (M \| $\epsilon$) |
| M+ | = | MM* |
| . | = | Any character except newline |
| "vc" | = | The string vc exactly |
| 4"."5 | = | String 4.5 (not 4<any>5) |

# 01-25: r.e. Shorthand Examples

| Regular Expression | Langauge |
|---:|---|
| if | {if} |
| [a-z][0-9a-z]* | Set of legal identifiers |
| [0-9] | Set of integers (with leading zeroes) |
| ([0-9]+"."[0-9]*)\|<br>([0-9]*"."[0-9]+) | Set of real numbers |

# 01-26: Lexical Analyzer Generator

Lex is a lexical analyzer generator

- Input: Set of regular expressions (each of which describes a type of token in the language)
- Output: A lexical analyzer, which reads an input file and separates it into tokens

# 01-27: Structure of lex file

```
%{

/* C declarations */

%}

/* Lex definitions */

%%

/* Regular expressions and actions */
```

# 01-28: How Lex Works

- Lex creates a function int yylex(), which does the following:

```
while(true) {
      <find a rule that matches the next
         section of the input>

      <execute action associated with
         that rule>
}
```

- Thus, a call to yylex() will keep matching rules and performing actions, until an action contains a return statement

# 01-29: Lex Example

```
%{
#include "tokens.h"
%}

%%

for         { return FOR; }
if          { return IF; }
```

# 01-30: Lex Example

```
%{
#include "tokens.h"
%}

%%

for         { return FOR; }
if          { return IF; }
","         { return COMMA; }
```

# 01-31: Lex Example

```
%{
#include "tokens.h"
%}

%%

for         { return FOR; }
if          { return IF; }
","         { return COMMA; }
" "         {  }
\n          {  }
```

# 01-32: Lex Example

```
%{
#include "tokens.h"
%}

%%

for         { return FOR; }
if          { return IF; }
","         { return COMMA; }
" "         {  }
\n          {  }
[0-9]+      { return INTEGER_LITERAL; }
```

# 01-33: Lex Example

```
%{
#include <string.h>
#include "tokens.h"
%}
%%

for         { return FOR; }
if          { return IF; }
","         { return COMMA; }
" "         {  }
\n          {  }
[0-9]+      { yylval.integer_value.value =
                 atoi(yytex);
              return INTEGER_LITERAL;    }
```

# 01-34: Lex Loop

Lex creates a function yylex, which does the following:

- Find regular expression that matches the input
- Execute the action associated with that regular expression
- Repeat, until a return statement is reached – return the appropriate value

# Lex Matching

What if more than one regular expression matches?

- Find the longest possible match
  - if382 returns an IDENTIFIER, not an IF followed by an INTEGER_LITERAL
- If two matches are the same length, match the string that appears earliest in the file.
  - if returns an IF, not an IDENTIFIER (as long as the rule for IDENTIFIER appears after the rule for IF)

# 01-36: Lex States

- We can label each regular expression/action pair with a “state”
- Unlabeled regular expression/action pairs are assumed to be in the default INITIAL state
- Lex will only match regular expressions for the current state
- Can switch between states with a BEGIN action

# 01-37: Using Lex States

```
%{
#include "tokens.h"
%}

%x COMMENT

%%
else                    { return ELSE; }
for                     { return FOR; }
" "                     { }
\n                      { }
"/*"                    { BEGIN(COMMENT); }
<COMMENT>"*/"           { BEGIN(INITIAL); }
<COMMENT>\n             { }
<COMMENT>.              { }
```

# 01-38: Token Positions

```
%{
#include "tokens.h"
#include <string.h>
int current_line_number = 1;
void newline() {
  current_line_number++;
}
%}
%%
else                    { yylval.line_number = current_line_number;
                          return ELSE; }
" "                     {     }
\n                      { newline(); }
";"                     { yylval.line_number = current_line_number;
                          return SEMICOLON; }
[a-zA-Z][a-zA-Z0-9]*    { yylval.string_value.line_number = current_line_number;
                          yylval.string_value.value = malloc(sizeof(char) *
                                                          (strlen(yytext)+1));
                          strcpy(yylval.string_value.value,yytext);
                          return IDENTIFIER; }
```

# 01-39: Random Details

- tokens.h
- Skeleton .lex file
  - Note Current_Line is defined in errors.h
- errors.h file
- main (driver) program
- makefile