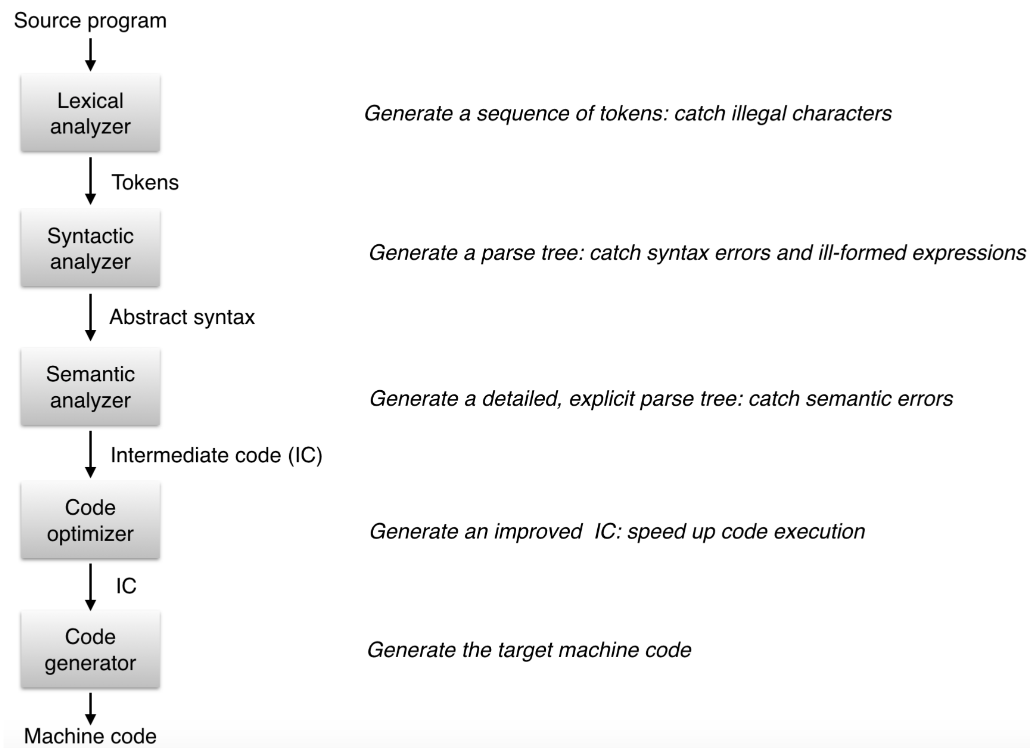
Syntax (continued)

Stages of Compilation



- Lexical analysis
    - takes source file as input
    - generate a sequence of valid tokens
    - character sequences that do not form valid tokens are discard, after generating an error message
- Syntactic analysis
    - takes a sequence of tokens as input
    - parses the token sequence, constructs a parse tree/abstract syntax tree according to the grammar
    - check syntax errors and ill-formed expressions
- Semantic analysis
    - takes parse tree/abstract syntax tree as input
    - generate intermediate code (more explicit, detailed parse tree where operators will generally be specific to the data type they are processing)
    - catch semantic errors like undefined variables, variable type conflicts, and implicit conversions
- Code optimization
    - take the intermediate code as input
    - identify optimizations that speed up code execution without changing the program functionality

- Code generator
    - converts the intermediate code into machine code
    - machine code is tailored to a specific machine, while intermediate code is general across platforms

---

Lexical Analysis

- Take a source file as the input, generate a sequence of valid tokens. Discard invalid characters after generating an error message.
- Token
    - Identifiers
        - variable names, function names, labels
    - Literals
        - numbers (e.g. Integers and Floats), characters, true and false
    - Keywords
        - bool char else false float if int main true while
    - Operators
        - for example, + - / * && || ==
    - Punctuation
        - for example, ; . {} ()
- Tokenization, or lexical analysis, is simply conversion from a string of characters or whatever input format is being used to a sequential string of symbols.
- Do not do syntax checking, but can identify improperly define identifiers.
    - In another word, it handles at least part of all of the rules that have a terminal on the right side.
    - In the case of something like an if statement, it converts the string if into a symbol that represents the keyword.
- It is not a trivial part of compiler.
    - Takes a significant percentage of time in compilation. Up to 75% of the time for a non-optimizing compiler.
    - Most compilers separate tokenization, or lexical analysis, from syntactic analysis and program generation.
- Because tokenization is such a common process, there are some nice tools for generating lexical analyzers automatically based on a description of the token grammar.
    - Examples include lex and **flex** (fast lexical analyzer generator, written in C around 1987), both are freely available. Flex is faster than lex. We use flex in this course.
    - These tools permit you to write the lexical syntax components of a language as a set of rules, generally based on regular expressions.

Regular Expressions

- Regular expressions are a language on their own designed to compactly represent a set of strings as a single expression.
- Special characters in regular expressions
    - **[]**: used to specify a set of alternatives *(different from EBNF)*
        - [AEIOU]: one uppercase vowel
        - T[ao]p: tap, top
    - \: used as an escape character to permit use of other special character
        - \d: one digit from 0 to 9.

    - \s: whitespace
    - Write an regex to match all CS courses. [CS\s\d\d\d matches CS XXX]
    - BUT this didn't work in flex when Stephanie tried it, don't use this in the context of flex.
- **.** : matches almost any character except line breaks
    - a.e: water, ate, gate
- ***** : match the prior expression zero or more times
    - \. : decimal point
    - Write a regex to match floating point values with one digit after the decimal points. [\d*\.\d: .3, 12.5, 139.9]
- **-** : range indicator
    - [a-z]: one lowercase letter from a to z
- **^**: negates an expression when inside brackets, permits you to specify strings that don't include a certain expression, or is the start of the string otherwise
    - [^0-9]: matches any character that is not a digit
    - ^a: matches strings start with a
- **$**: the end of the string
    - the end$: this is the end
    - Write a regex that can match any number between 1000 and 9999.
    - ^[1-9][0-9][0-9][0-9]$
    - create a text.txt with following numbers, one on each line, `1231 21 5 57 0101 100001 1000a`; key in the command `egrep "^[1-9][0-9][0-9][0-9]$" text.txt`; this command returns the matches in text.txt.
- **()** : group tokens *(different from EBNF)*
    - th(e|is) : the, this
- **+** : match the prior expression one or more times
    - html tags: <html> </html>, <h1></h1>, <div id="block1"></div>
    - <[A-Za-z][A-Za-z0-9]*>: matches HTML tags without any attributes
    - <[A-Za-z0-9]+>: matches HTML tags without any attributes, but can have invalid tag like <1>
    - <[^< >]+> : matches HTML tags without regard to attributes
- **{min, max}** : specify how many times a token can be repeated, min >=0 minimum number of matches, max >= min maximum number of matches. If {min, } the maximum number of matches is infinite. {min} repeat exactly min times.
    - {0, } same as *, {1, } same as +
    - ^[1-9][0-9]{3}$: matches a number between 1000 and 9999
    - ^[1-9][0-9]{2,4}$: matches a number between 100 and 99999
- **?** : makes an expression lazy (first possible completion) instead of greedy (largest possible completion)
    - \w{3,5}? : "app" in "apple"
    - \w: word character (ASCII letter, digit, or unicode)
    - {3,5}: three to five times
    - ?: once or none

To test your understanding of regular expressions, you can use the Mac Terminal/Unix program *egrep*. Egrep stands for “Extended Global Regular Expressions Print” and, given a regular expression, it searches a file line by line and reports which lines match the regular expression.

http://www.cs.columbia.edu/~tal/3261/fall07/handout/egrep_mini-tutorial.htm

For example, in class I wrote the file courses.txt with the following 4 lines in it:

CS333
CS125
ECE345
CS232
CS232L

And then searched it for

all CS courses (CS followed by 3 digits):

```
egrep “CS[0-9][0-9][0-9]” courses.txt
```

which printed out

```
CS333
CS125
CS232
CS232L
```

as did a shorter command that indicates 3 digits

```
egrep "CS[0-9]{3}" courses.txt
```

We the searched for all CS courses that weren’t labs (i.e. didn’t have an “L” at the end).

```
egrep "CS[0-9][0-9][0-9]$" courses.txt
```

which printed out

```
CS333
CS125
CS232
```

Then we searched for classes at the 100- and 200-level

```
egrep “CS[13][0-9][0-9]” courses.txt
```

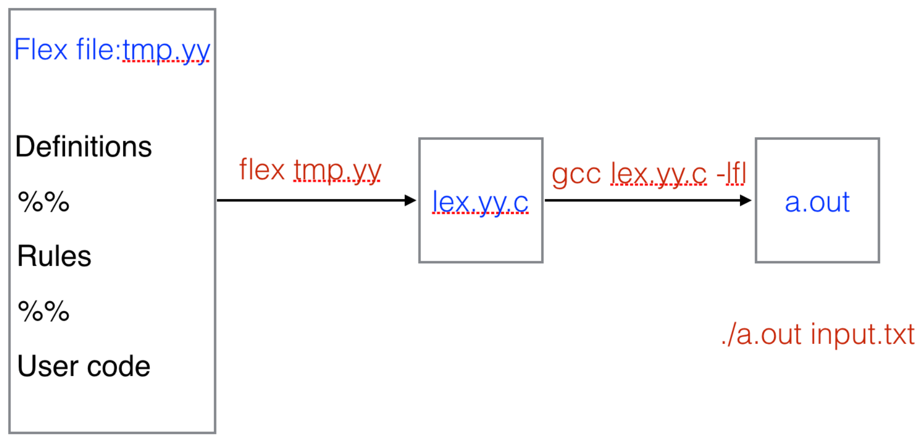
which printed out

```
CS333
CS125
```

Once you have mastered individual regular expressions, you can move on to using them within the context of a parser.

Flex

- Flex makes use of regular expressions to define lexical tokens.
- A lexical parser is defined by a set of rules. Each rule is a regular expression followed by C code that expresses an action (including doing nothing) when flex finds a string matching the regular expression.
- The rules are tested in the order in which they appear in the flex file, which allows you to specify priority.
- Text that does not match any rule is passed along to the output.
- A flex has three parts: definitions, rules, and user code. They are separated by the expression %%.



- Definitions
  - Define macros to be used in the rule section
  - Include C code to define variables used in the rule implementations
- Rules
  - Defined by a regular expression and C code
  - The regular expression must not be indented
  - The code for a rule has to start on the same line as the regular expression
  - Multi-line code needs to be inside a block ({…}), with at least the opening curly-bracket on the same line as the rule
- User code
  - Appended to the end of the C file generated by flex.
  - If you put nothing at the end of the C file, you need to write your own main function, link it with the flex output file (lex.yy.c, by default), and call the function yylex() inside your code.
  - Alternatively, you can put the main function in the user code section of the flex file.

Simple String Replacement

```
/**
 * Hello World: replace "blah" with "hello world"
 * flex -o hello.yy.c hello.yy
 * gcc -o hello hello.yy.c -ll
 * echo "blah and another blah" | ./hello
 */

%%

blah    printf("hello world");

%%

int main ( int argc, char *argv[] ) {

        yylex();

        return 0;
}
```

Sample for reading input file

```
/**
 * Read in from a specified file and
 * print out a list of all the integers in the file
 *
 * flex -o test.yy.c test.yy
 * gcc -o test test.yy.c -ll
 */

        int count = 0; // the whitespace here is important.(the declaration is tabbed in)

DIGIT      [0-9]

%%

{DIGIT}+       { printf("number: %10d\n", atoi(yytext));
                 /*yytex a special character available to the C code
                       contains the text corresponding to the current token:
                       the text matched by the regular expression */
                 }
\n             count++;
.              /* skip all other input */

%%
int main( int argc, char *argv[] ) {

   if (argc > 1)
      yyin = fopen( argv[1], "r" ); //where yylex reads its input

   yylex(); // a function of flex that read input till it is exhausted
   printf("There are %d lines in the file.\n", count);
   return 0;
}
```