Phases of compilation

1) scanning - turn character stream into stream of tokens - identifies operators, constants...

```c
int main(int argc, char **argv) {
    printf("Hello world \n");
}
```

2) parsing - checks syntax, builds parse tree

3) semantic analysis - type checking, declaration

4) platform independent code improvement (optimal)

5) target generation

6) platform specific code improvement (registers, instruction choice... )
formal representation of what tokens look like

Java identifier: not a reserved word
starts w/ letter or underscore, followed
by any number of letters, digits, underscores

regular expressions:
1) any character is a regular expression
   \( x \) or \( ^{\text{\textquotedblright}x\text{\textquotedblright}} \)

2) if \( \alpha, \beta \) are regular expressions
   \( \alpha | \beta \) is too
   \( x | y \) \{"x", "y"\}

3) \( \alpha \beta \)
   \( x y \) \{"xy"\}

   \( (\alpha | \beta)^* \) \{"x^2", "y^2"\}

4) \( \alpha^+ \)
   one or more
   \( x^* \) \{"x", "xx", "xxx", "\ldots"\}

5) \( \alpha^* \)
   \( (x|y)^* \) \{"x", "y", "xy", "yx", "xx", "yy", "\ldots"\}

5) \( \epsilon \)
   \{"\"\}
other parts of syntax: line length
line position (Fortran)
indentation (Python)

letter: a | b | c | ... | z | A | B | ... | Z
digit: 0 | 1 | 2 | ... | 9
id: (letter | _) (letter | digit | _)*
signed-int: (+| -|) digit digit* ± 8 X

(not "")

escapes: \ | \n | \t | " | "... "
non-quote: letter | digit | - | \ | ... | escapes

quoted-string: "" non-quote + ""
can’t do (")" with regular expressions
but can do with context free grammars

CFL

arithmetic expressions (+ - \* / [ ]

\[6\*7 + 8\*9 \quad 6 + 7\] 

recursive nature of CFLs gives power but makes recognizing conforming strings more difficult

\text{expr} \rightarrow \text{num} | \text{id} | \text{expr op expr} | (\text{expr}) | -\text{expr} \\
\text{num} \rightarrow \text{see above (RE)} \\
\text{op} \rightarrow \text{\* | / | + | -}