Example: 3 - 1 + 2

\[
Expr \rightarrow Expr + Term | Expr - Term | Term \\
Term \rightarrow Integer \\
Integer \rightarrow Digit \{Digit\} \\
Digit \rightarrow 0 | \cdots | 9
\]

\[
Expr \\
\downarrow \\
Expr + Term \\
\downarrow \\
Expr - Term \\
\downarrow \\
Term \\
\downarrow \\
Integer \\
\downarrow \\
Digit 2 \\
\downarrow \\
Digit 1 \\
\downarrow \\
Digit 3
\]

\[ (3 - 1) + 2 \]
Example: $3 - 1 + 2$ (cont.)

\[
\begin{align*}
Expr & \rightarrow Term + Expr \mid Term - Expr \mid Term \\
Term & \rightarrow Integer \\
Integer & \rightarrow Digit\{Digit\} \\
Digit & \rightarrow 0 \mid \cdots \mid 9
\end{align*}
\]

```
Expr
  Term
    Integer
      Digit
        3
  -
    Expr
      Term
        Integer
          Digit
            1
      +
        Expr
          Term
            Integer
              Digit
                2
              3 - (1 + 2)
```
Example: Expression

\[
\begin{align*}
Expr & \rightarrow Expr\ Op\ Expr | (Expr) | Integer \\
Op & \rightarrow + | - | * | / \\
Integer & \rightarrow Digit\{Digit\} \\
Digit & \rightarrow 0 | \cdots | 9
\end{align*}
\]

3 - 1 + 2
Ambiguous Grammars

A grammar is *ambiguous* if its language contains *at least one* string with *two or more distinct parse trees*.

- Unambiguous grammar is preferred
- Ambiguous grammar is tolerable
  - Simplify the rules in a grammar
Example: Dangling else

\[
\text{ifStatement} \rightarrow \text{if (Expression)Statement} \mid \text{if (Expression)Statement else Statement}
\]

\[
\text{Statement} \rightarrow \text{Assignment} \mid \text{ifStatement} \mid \text{Block}
\]

\[
\text{Block} \rightarrow \{\text{Statement}\{\text{Statement}\}}
\]

if \ (x < 0)
  if \ (y < 0)
    y = y + 1;
  else
    y = 0;
Example: Dangling else

✦ **Approach 1**: language specifications (C/C++)

Every `else` clause is associated with the **textually closest preceding unmatched if** statement.

✦ **Approach 2**: expanding the BNF grammar (Java)

\[
\text{IfThenStatement} \rightarrow \text{if (Expression) Statement} \\
\text{IfThenElseStatement} \rightarrow \text{if (Expression) StatementNotShortIf} \\
\text{else Statement} \\
\text{StatementNotShortIf}: \text{all statements except IfThenStatement}
\]

✦ **Approach 3**: closing keywords

```bash
#!/bin/bash

if [ conditional expression1 ]
then
    statement1
    statement2
else
    if [ conditional expression2 ]
    then
        statement3
    fi
fi
```
Stages of Compilation

Source program

Lexical analysis
- Generate a sequence of tokens: catch illegal characters

Tokens

Syntactic analysis
- Generate a parse tree: catch syntax errors and ill-formed expressions

Abstract syntax

Semantic analysis
- Generate a detailed, explicit parse tree: catch semantic errors

Intermediate code (IC)

Code optimizer
- Generate an improved IC: speed up code execution

Intermediate code (IC)

Code generator
- Generate the target machine code

Machine code
Lexical Analysis

- An important component:
  - For a non-optimizing compiler, lexical analysis can take up to 75% of compilation time.

- Transform a source file from a sequence of characters to a sequence of tokens:
  - **Identifiers**: strings made up of letters and digits
  - **Literals**: Integers, Floats, Chars
  - **Keywords**: bool char else false float if int main true while
  - **Operators**: + - * / && || = ==
  - **Punctuation**: ; . { } ( )

- Tools for generating lexical analyzers: **Lex**, **Flex**
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

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>A set of alternatives</td>
<td>[ae]</td>
<td>apple (a, e)</td>
</tr>
<tr>
<td>\</td>
<td>An escaped character</td>
<td>\d</td>
<td>4G (4)</td>
</tr>
<tr>
<td>{ name }</td>
<td>A reference to a name</td>
<td>{ letter }</td>
<td></td>
</tr>
<tr>
<td></td>
<td>An or between expressions</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>()</td>
<td>Group tokens</td>
<td>th(e</td>
<td>is)</td>
</tr>
<tr>
<td>.</td>
<td>Any single character except line breaks</td>
<td>a.e</td>
<td>water (ate)</td>
</tr>
<tr>
<td>*</td>
<td>Zero or more occurrences</td>
<td>\d*.\d</td>
<td>.3, 12.5, 139.7</td>
</tr>
<tr>
<td>+</td>
<td>One or more occurrences</td>
<td>go+</td>
<td>good, god</td>
</tr>
<tr>
<td>?</td>
<td>Zero or one occurrence</td>
<td>rai?n</td>
<td>rain, ran</td>
</tr>
</tbody>
</table>
Fast Lexical Analyzer (Flex)

Flex file: tmp.yy

Definitions
%%
Rules
%%
User code

flex tmp.yy

lex.yy.c

gcc lex.yy.c -lfl

a.out

./a.out input.txt
Flex File - Definitions

• Define macros used in the rule section

• Can include C code to define variables used in the rule implementations

DIGIT \[0-9\]
ID \[a-z][a-z0-9\]
Rules

- Rules are tested in the order they appear in the file (specify priority)
- Each rule is defined by a regular expression and C code
- The regular expressions 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-brace on the same line as the rule

```
[0-9]+          printf( "number %10d\n", atoi(yytext) );
.              /* skip all other input */
```

```
{DIGIT}+        printf( "number %10d\n", atoi(yytext) );
.              /* skip all other input */
```
Flex File - User Code

- User code
  - Appended to the end of the C file generated by Flex
  - Include `main` function in the user code section

```c
int main(int argc, char *argv[]) {
    if ( argc > 1 )
        yyin = fopen( argv[1], "r" );
    else
        yyin = stdin;
    yylex();
}
```
Sample Code

```c
/**
 * Read in from either stdin or a specified file and
 * print out a list of all the integers in the file
 * Lecture 5 code
 *
 * flex -o lec5.yy.c lec5.yy
 * gcc -o lec5 lec5.yy.c -lfl
 */

DIGIT       [0-9]

%

{DIGIT}+     printf("number: %10d\n", atoi(yytext));
.           /* skip all other input */
%

int main( int argc, char *argv[] ) {
    if (argc > 1)
        yyin = fopen( argv[1], "r" );
    else
        yyin = stdin;
    yylex();
}