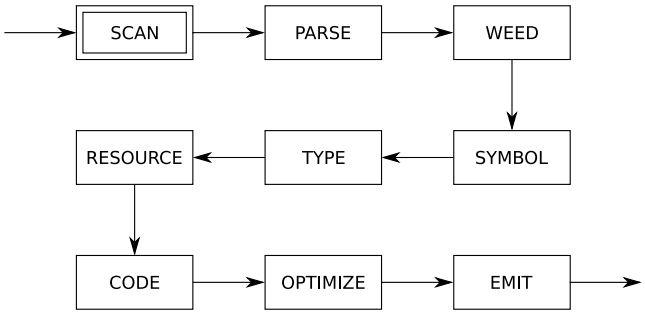
Scanning

COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 13:30-14:30, MD 279



Announcements (Friday, January 6th)

Facebook group:

- Useful for discussions/announcements
- Link on myCourses or in email

Milestones:

- Continue picking your group (3 recommended)
- Create a GitHub account, learn git as needed

Midterm:

- Either 1st or 2nd week after break on the Friday
- 1.5 hour “in class” midterm, so either 30 minutes before/after class. *Thoughts?*
- **Tentative date:** Friday, March 10th. Or the week after? *Thoughts?*

Readings

Textbook, Crafting a Compiler:

- Chapter 2: *A Simple Compiler*
- Chapter 3: *Scanning–Theory and Practice*

Modern Compiler Implementation in Java:

- Chapter 1: *Introduction*
- Chapter 2: *Lexical Analysis*

Flex tool:

- Manual - `https://github.com/westes/flex`
- Reference book, Flex & bison - `http://mcgill.worldcat.org/title/flex-bison/oclc/457179470`

Scanning:

- also called lexical analysis;
- is the first phase of a compiler;
- takes an arbitrary source file, and identifies meaningful character sequences.
- *note: at this point we do not have any semantic or syntactic information*

Overall:

- a scanner transforms a string of characters into a string of tokens.

An example:

```
var a = 5
if (a == 5)
{
    print "success"
}
```

```
tVAR
tIDENTIFIER: a
tASSIGN
tINTEGER: 5
tIF
tLPAREN
tIDENTIFIER: a
tEQUALS
tINTEGER: 5
tRPAREN
tLBRACE
tIDENTIFIER: print
tSTRING: success
tRBRACE
```

Review of COMP 330:

- Σ is an *alphabet*, a (usually finite) set of symbols;
- a *word* is a finite sequence of symbols from an alphabet;
- Σ^* is a set consisting of all possible words using symbols from Σ;
- a *language* is a subset of Σ^*.

An example:

- **alphabet:** Σ={0,1}
- **words:** {ϵ, 0, 1, 00, 01, 10, 11, . . . , 0001, 1000, . . . }
- **language:**
  - {1, 10, 100, 1000, 10000, 100000, . . . }: “1” followed by any number of zeros
  - {0, 1, 1000, 0011, 11111100, . . . }: *?!*

A regular expression:

- is a string that defines a language (set of strings);
- in fact, a *regular* language.

A regular language:

- is a language that can be accepted by a DFA;
- is a language for which a regular expression exists.

In a scanner, tokens are defined by *regular expressions*:

- $\emptyset$ is a regular expression [the empty set: a language with no strings]
- ε is a regular expression [the empty string]
- a, where $a \in \Sigma$ is a regular expression [Σ is our alphabet]
- if M and N are regular expressions, then $M \mid N$ is a regular expression
  [alternation: either M or N]
- if M and N are regular expressions, then $M \cdot N$ is a regular expression
  [concatenation: M followed by N]
- if M is a regular expression, then M^* is a regular expression
  [zero or more occurences of M]

What are M? and M^+?

Examples of regular expressions:

- Alphabet Σ={a,b}
- a* = {ϵ, a, aa, aaa, aaaa, . . . }
- (ab)* = {ϵ, ab, abab, ababab, . . . }
- (a|b)* = {ϵ, a, b, aa, bb, ab, ba, . . . }
- a*ba* = strings with exactly 1 “b”
- (a|b)*b(a|b)* = strings with at least 1 “b”

We can write regular expressions for the tokens in our source language using standard POSIX notation:

- simple operators: `"*"`, `"/"`, `"+"`, `"-"`
- parentheses: `"("`, `")"`
- integer constants: `0|([1-9][0-9]*)`
- identifiers: `[a-zA-Z_][a-zA-Z0-9_]*`
- white space: `[␣\t\n]+`

[. . .] define a *character class*:

- matches a single character from a set;
- allows ranges of characters to be “alternated”; and
- can be negated using “^” (i.e. `[^\n]`).

The wildcard character:

- is represented as “.” (dot); and
- matches all characters except newlines by default (in most implementations).

A scanner:

- can be generated using tools like `flex` (or `lex`), `JFlex`, `...`;
- by defining *regular expressions* for each type of token.

Internally, a *scanner* or *lexer*:

- uses a combination of *deterministic finite automata* (DFA);
- plus some glue code to make it work.

A *finite state machine* (FSM):

- represents a set of possible states for a system;
- uses transitions to link related states.

A *deterministic finite automaton* (DFA):

- is a machine which recognizes regular languages;
- for an input sequence of symbols, the automaton either *accepts* or *rejects* the string;
- it works *deterministically* - that is given some input, there is only one sequence of steps.

Background (DFAs) from textbook, “Crafting a Compiler”

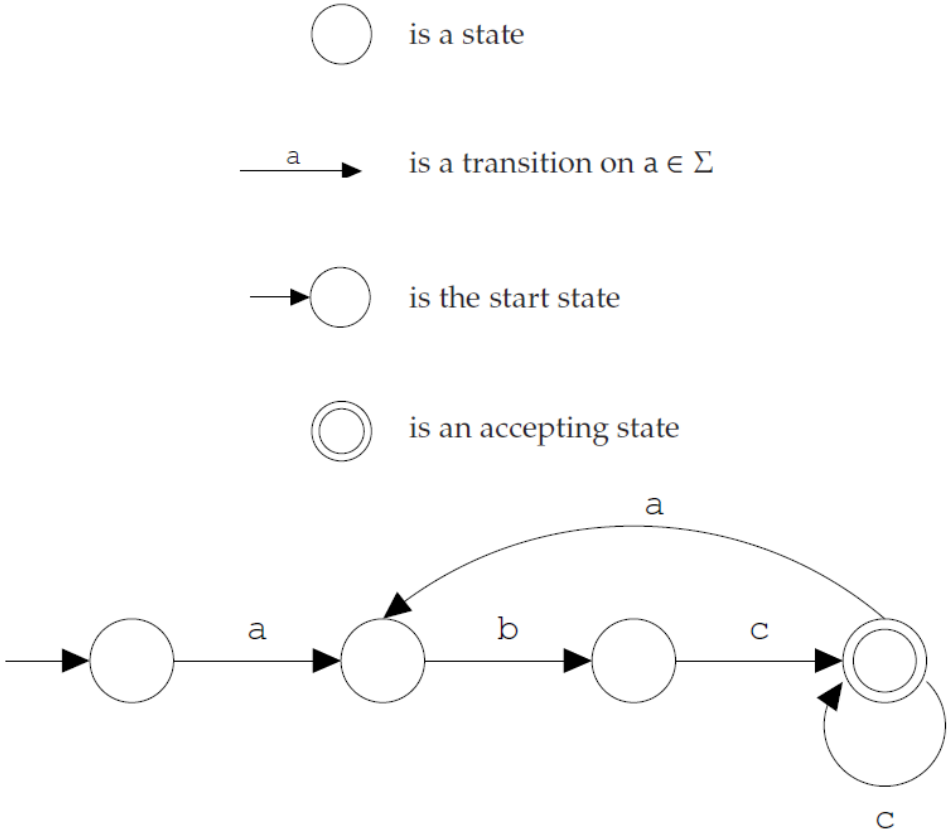
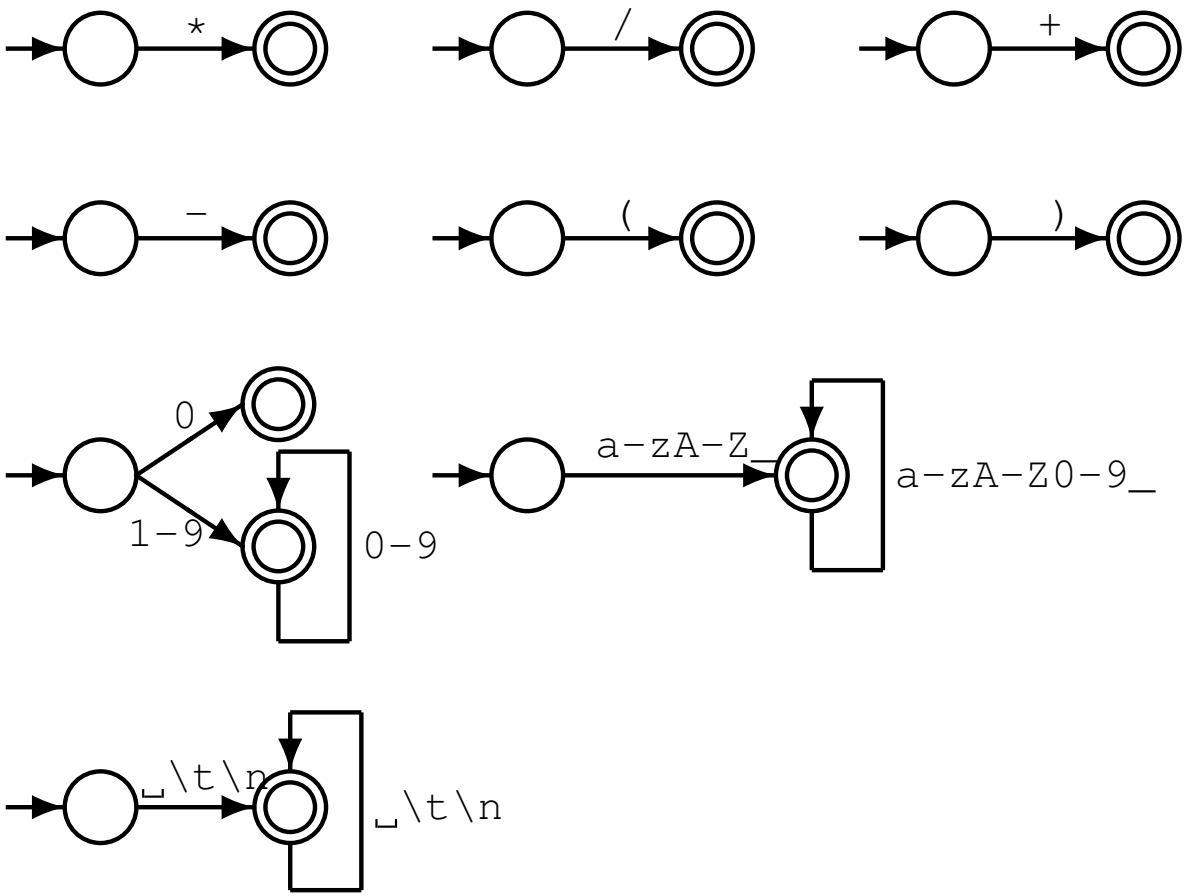


Figure 3.1: Components of a finite automaton drawing and their use to construct an automaton that recognizes $(a\,b\,c^{+})^{+}$.

DFAs (for the previous example regexes):

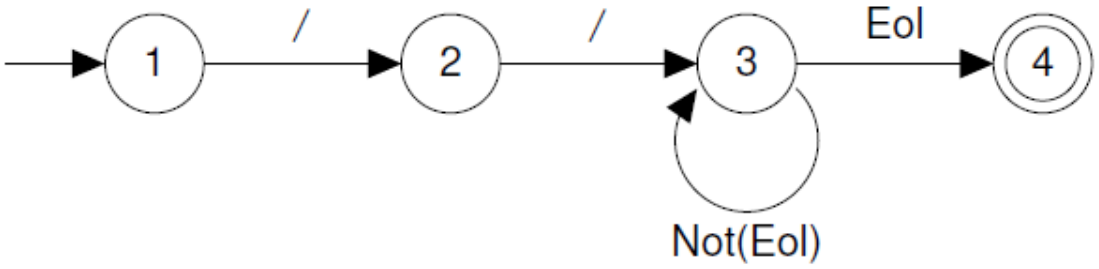


Try it yourself:

- Design a DFA matching binary strings divisible by 3. Use only 3 states.
- Design a regular expression for floating point numbers of form: {1., 1.1, .1} (a digit on at least one side of the decimal)
- Design a DFA for the language above language.

Background (Scanner Table) from textbook, “Crafting a Compiler”

(a)



(b)

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

Background (Scanner Algorithm) from textbook, "Crafting a Compiler"

```
/* Assume CurrentChar contains the first character to be scanned */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    CurrentChar ← READ( )
if State ∈ AcceptingStates
then /* Return or process the valid token */
else /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.

A *non-deterministic finite automaton*:

- is a machine which recognizes regular languages;
- for an input sequence of symbols, the automaton either *accepts* or *rejects* the string;
- it works *non-deterministically* - that is given some input, there is potentially more than one path;
- an NFA accepts a string if at least one path leads to an accept.

Note: DFAs and NFAs are equally powerful.

Regular Expressions to NFA (1) from textbook, “Crafting a Compiler”

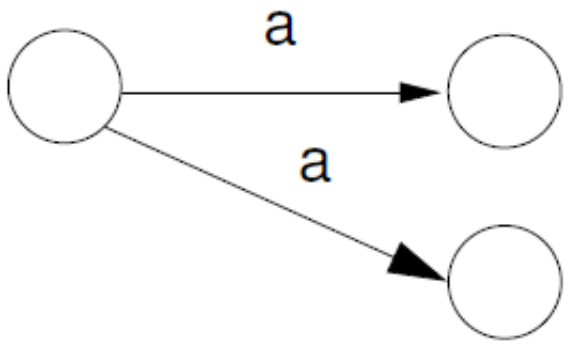


Figure 3.17: An NFA with two a transitions.

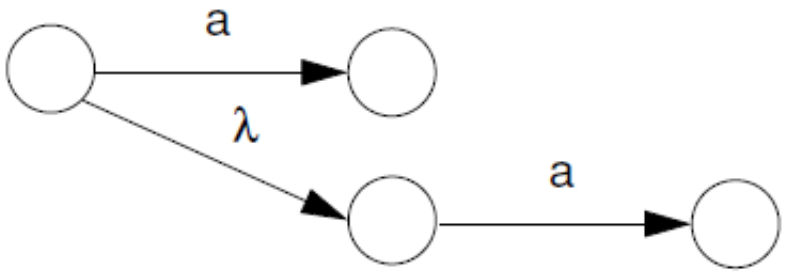


Figure 3.18: An NFA with a λ transition.

Regular Expressions to NFA (2) from textbook, ”Crafting a Compiler"

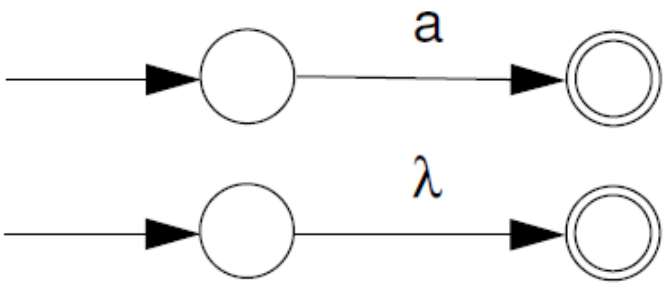


Figure 3.19: NFAs for a and λ.

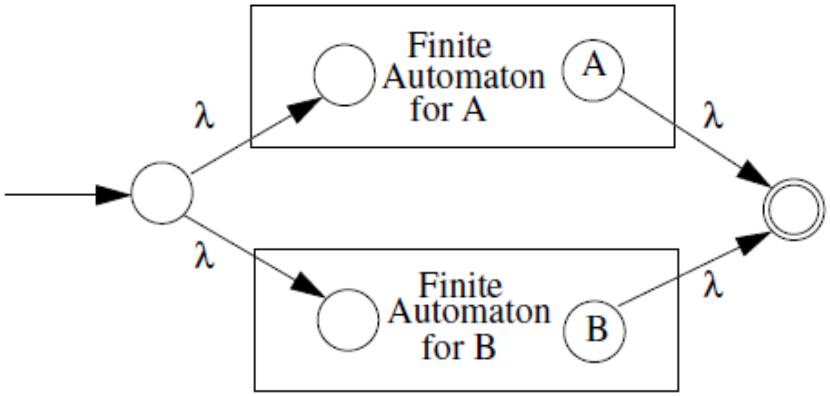


Figure 3.20: An NFA for $A \mid B$.

Regular Expressions to NFA (3) from textbook, ”Crafting a Compiler"

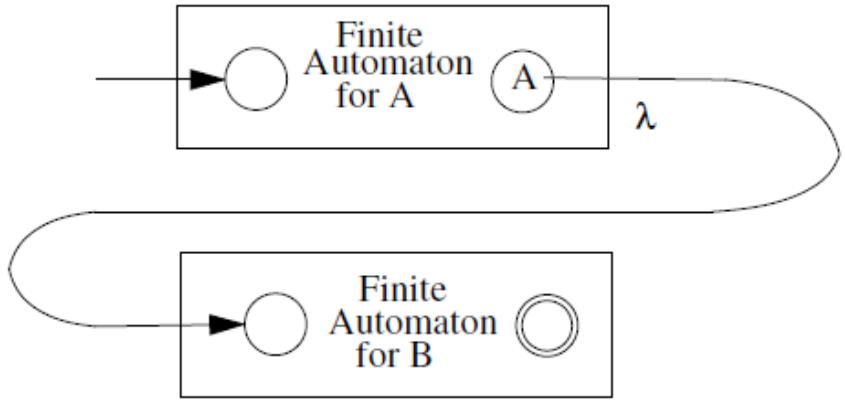


Figure 3.21: An NFA for AB.

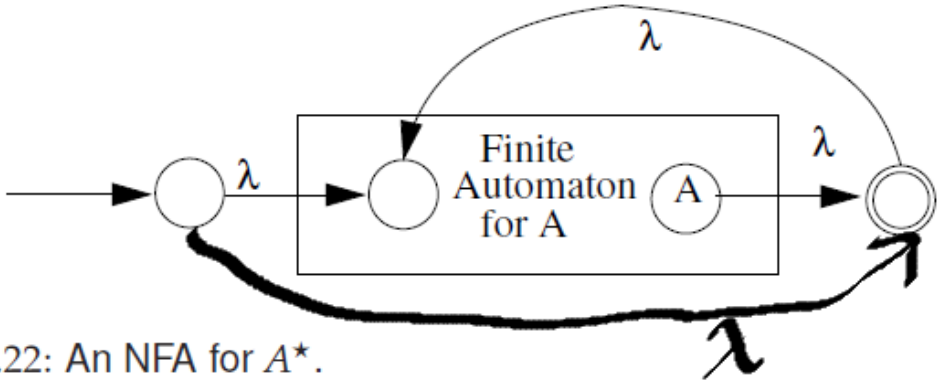


Figure 3.22: An NFA for $A^{\star}$.

How to go from regular expressions to DFAs?

1. `flex` accepts a list of regular expressions (regex);
2. converts each regex internally to an NFA (Thompson construction);
3. converts each NFA to a DFA (subset construction)
4. may minimize DFA

See “Crafting a Compiler", Chapter 3; or “Modern Compiler Implementation in Java", Chapter 2

What you should know:

1. Understand the definition of a regular language, whether that be: prose, regular expression, DFA, or NFA.

2. Given the definition of a regular language, construct either a regular expression or an automaton.

What you do not need to know:

1. Specific algorithms for converting between regular language definitions.

2. DFA minimization

Let's assume we have a collection of DFAs, one for each lex rule

```
reg_expr1     ->     DFA1
reg_expr2     ->     DFA2
...
reg_rexpn     ->     DFAn
```

How do we decide which regular expression should match the next characters to be scanned?

Given DFAs $D_1, \ldots, D_n$, ordered by the input rule order, the behaviour of a `flex`-generated scanner on an input string is:

```
while input is not empty do
    s_i := the longest prefix that D_i accepts
    l := max{|s_i|}
    if l > 0 then
        j := min{i : |s_i| = l}
        remove s_j from input
        perform the j^th action
    else (error case)
        move one character from input to output
    end
end
```

- The *longest* initial substring match forms the next token, and it is subject to some action
- The *first* rule to match breaks any ties
- Non-matching characters are echoed back

Why the “longest match” principle?

Example: keywords

```
...
import                  return tIMPORT;
[a-zA-Z_][a-zA-Z0-9_]*  return tIDENTIFIER;
...
```

Given a string “`importedFiles`”, we want the token output of the scanner to be

```
tIDENTIFIER(importedFiles)
```

and not

```
tIMPORT tIDENTIFIER(edFiles)
```

Because we prefer longer matches, we get the right result.

Why the "first match" principle?

Example: keywords

```
...
continue                return tCONTINUE;
[a-zA-Z_][a-zA-Z0-9_]*  return tIDENTIFIER;
...
```

Given a string "`continue foo`", we want the token output of the scanner to be

```
tCONTINUE tIDENTIFIER(foo)
```

and not

```
tIDENTIFIER(continue) tIDENTIFIER(foo)
```

"First match" rule gives us the right answer: When both `tCONTINUE` and `tIDENTIFIER` match, prefer the first.

When "first longest match" (flm) is not enough, look-ahead may help.

FORTRAN allows for the following tokens:

```
.EQ., 363, 363., .363
```

flm analysis of `363.EQ.363` gives us:

```
tFLOAT(363) E Q tFLOAT(0.363)
```

What we actually want is:

```
tINTEGER(363) tEQ tINTEGER(363)
```

To distinguish between a `tFLOAT` and a `tINTEGER` followed by a ".", `flex` allows us to use look-ahead, using `'/'`:

```
363/.EQ. return tINTEGER;
```

A look-ahead matches on the full pattern, but only processes the characters before the `'/'`. All subsequent characters are returned to the input stream for further matches.

Another example taken from FORTRAN, FORTRAN ignores whitespace

1. `DO5I = 1.25` $\rightsquigarrow$ `DO5I=1.25`

   in C, these are equivalent to an assignment:

   ```
   do5i = 1.25;
   ```

2. `DO 5 I = 1,25` $\rightsquigarrow$ `DO5I=1,25`

   in C, these are equivalent to looping:

   ```
   for(i=1;i<25;++i){...}
   ```

   (`5` is interpreted as a line number here)

To get the correct token output:

1. flm analysis correct:

   ```
   tID(DO5I) tEQ tREAL(1.25)
   ```

2. flm analysis gives the incorrect result. What we want is:

   ```
   tDO tINT(5) tID(I) tEQ tINT(1) tCOMMA tINT(25)
   ```

   But we cannot make decision on `tDO` until we see the comma, look-ahead comes to the rescue:

   ```
   DO/({letter}|{digit})*=({letter}|{digit})*, return tDO;
   ```

Announcements (Monday, January 9th)

Facebook group:

- Useful for discussions/announcements
- Link on myCourses or in email

Milestones:

- Learn `flex`, `bison`, `SableCC`
- Assignment 1 out Wednesday
- Continue forming your groups

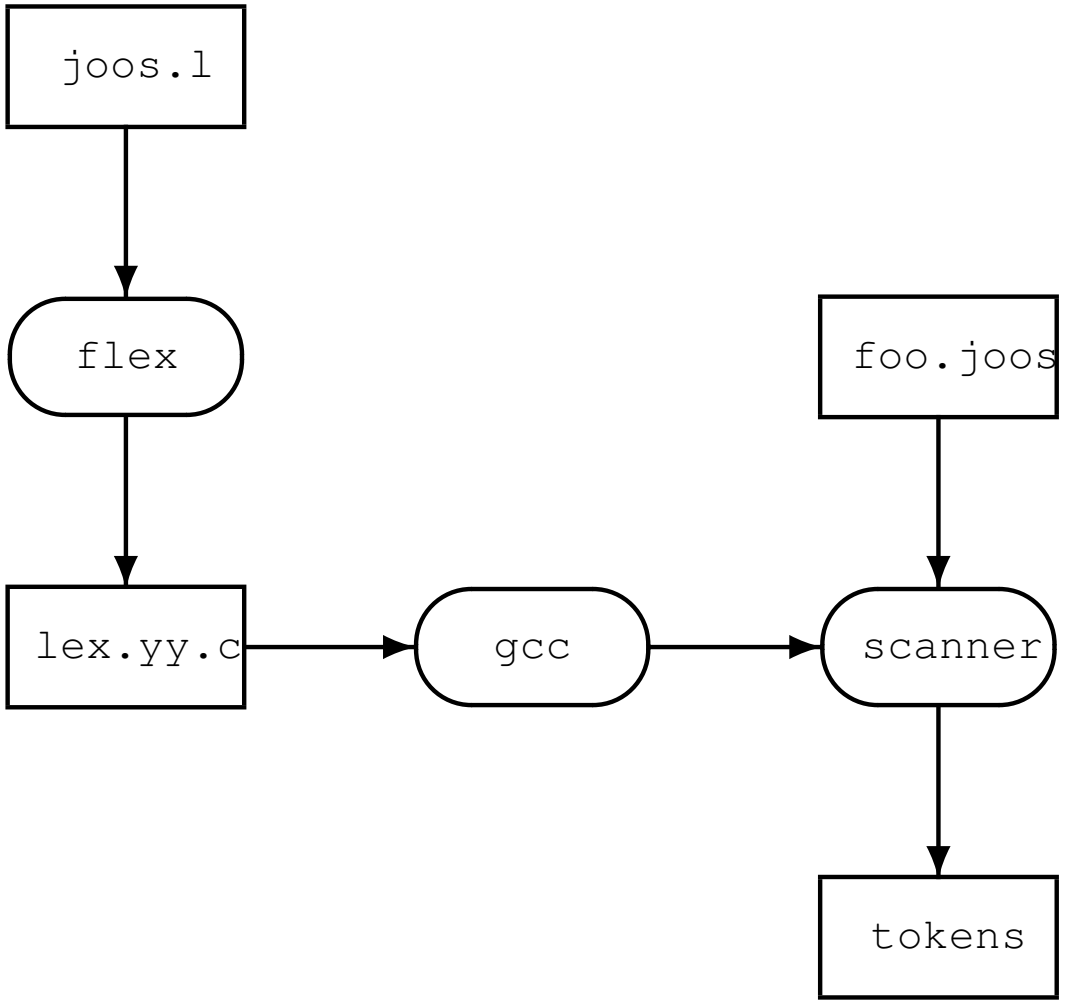
Midterm:

- **Friday, March 17th**
- 1.5 hour “in class” midterm. You have the option of either 13:00-14:30 or 13:30-15:00.

Introduce yourselves! (no, not joking)

- Name
- Major/year
- If grad student, research area
- *Any other fun facts we should know...*

In practice, we use tools to generate scanners. Using `flex`:



A flex file:

- is used to define a scanner implementation;
- has 3 main sections divided by %%:
  1. Declarations, helper code
  2. Regular expression rules and associated actions
  3. User code
- and saves much effort in compiler design.

```
/* includes and other arbitrary C code. copied to the scanner verbatim */
%{
%}
/* helper definitions */
DIGIT [0-9]

%%
/* regex + action rules come after the first %% */
RULE        ACTION

%%
/* user code comes after the second %% */
main () {}
```

```
$ cat print_tokens.l # flex source code

/* includes and other arbitrary C code */
%{
#include <stdio.h> /* for printf */
%}
/* helper definitions */
DIGIT [0-9]
/* regex + action rules come after the first %% */
%%
[ \t\n]+        printf ("white space, length %i\n", yyleng);
"*"             printf ("times\n");
"/"             printf ("div\n");
"+"             printf ("plus\n");
"-"             printf ("minus\n");
"("             printf ("left parenthesis\n");
")"             printf ("right parenthesis\n");

0|([1-9]{DIGIT}*) printf ("integer constant: %s\n", yytext);
[a-zA-Z_][a-zA-Z0-9_]* printf ("identifier: %s\n", yytext);
%%
/* user code comes after the second %% */
main () {
    yylex ();
}
```

Sometimes a token is not enough, we need the value as well:

- want to capture the value of an identifier; or
- need the value of a string, int, or float literal.

In these cases, `flex` **provides:**

- `yytext`: the scanned sequence of characters;
- `yylval`: a user-defined variable from the parser (`bison`) to be returned with the token; and
- `yyleng`: the length of the scanned sequence.

```
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.stringconst = (char *)malloc(strlen(yytext)+1);
    printf(yylval.stringconst,"%s",yytext);
    return tIDENTIFIER;
}
```

Using `flex` to create a scanner is really simple:

```
$ vim print_tokens.l

$ flex print_tokens.l

$ gcc -o print_tokens lex.yy.c -lfl
```

Running this scanner with input:

```
a*(b-17) + 5/c
```

```
$ echo "a*(b-17) + 5/c" | ./print_tokens
```

our `print_tokens` scanner outputs:

```
identifier: a
times
left parenthesis
identifier: b
minus
integer constant: 17
right parenthesis
white space, length 1
plus
white space, length 1
integer constant: 5
div
identifier: c
white space, length 1
```

Count lines and characters:

```
%{
    int lines = 0, chars = 0;
%}

%%
\n      lines++; chars++;
.       chars++;

%%
main () {
    yylex ();
    printf ("#lines = %i, #chars = %i\n", lines, chars);
}
```

Getting (better) position information in `flex`:

- is easy for line numbers: option and variable `yylineno`; but
- is more involved for character positions.

If position information is useful for further compilation phases:

- it can be stored in a structure `yylloc` provided by the parser (`bison`); but
- **must** be updated by a user action.

```
typedef struct yyltype
{
    int first_line, first_column, last_line, last_column;
} yyltype;


%{
    #define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;
%}
%option yylineno

%%
. { printf("Error: (line %d) unexpected char '%s'\n", yylineno, yytext); exit(1); }
```

Actions in a `flex` file can either:

- do nothing – ignore the characters;
- perform some computation, call a function, etc.; and/or
- return a token (token definitions provided by the parser).

```
%{
    #include <stdlib.h>    /* for atoi   */
    #include <stdio.h>     /* for printf */
    #include "lang.tab.h"  /* for tokens */
%}

%%
[aeiouy]       /* ignore */
[0-9]+         printf ("%i", atoi (yytext) + 1);
'\\n'          {  yylval.rune_const = '\n';
                  return tRUNECONST;          }
%%
main () {
    yylex ();
}
```

Summary

- a scanner transforms a string of characters into a string of tokens;
- scanner generating tools like `flex` allow you to define a regular expression for each type of token;
- internally, the regular expressions are transformed to a deterministic finite automata for matching;
- to break ties, matching uses 2 principles: “longest match” and “first match”.