Scanning

COMP 520: Compiler Design (4 credits)

Professor Laurie Hendren

hendren@cs.mcgill.ca

Readings

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 - `http://flex.sourceforge.net/manual/`
- Reference book, Flex & bison - `http://mcgill.worldcat.org/title/flex-bison/oclc/457179470`

Background (1), from "Crafting a Compiler"

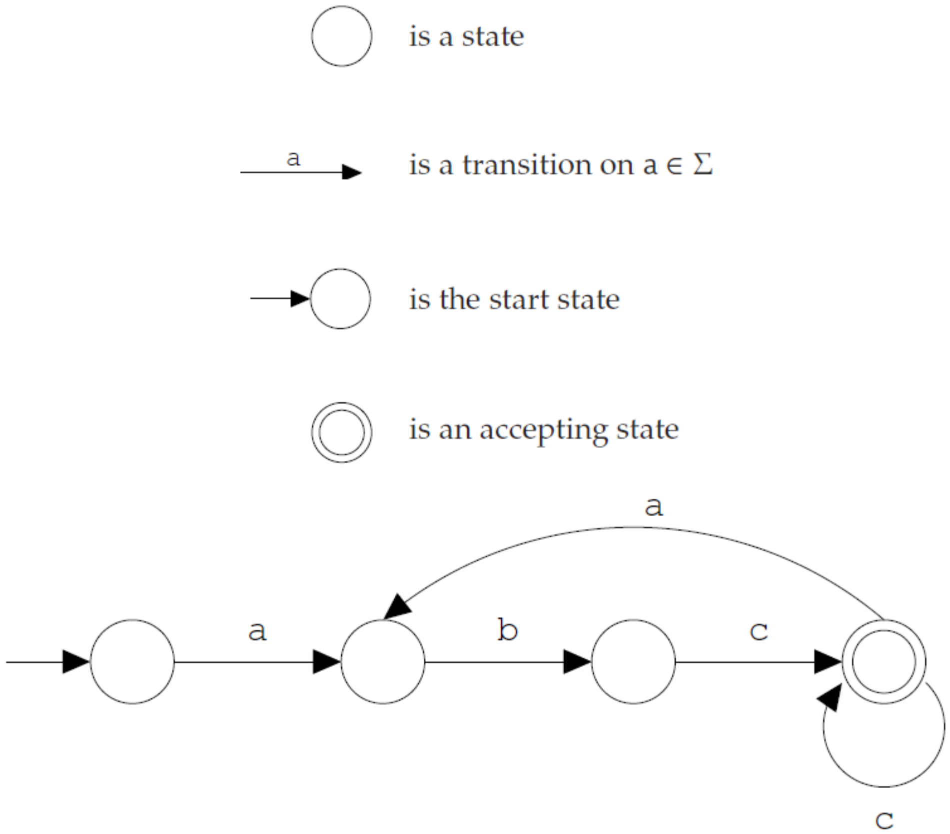
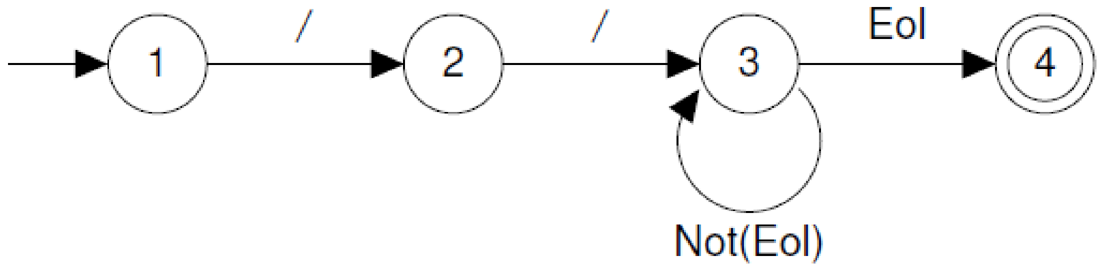


Figure 3.1: Components of a finite automaton drawing and their use to construct an automaton that recognizes $(a\,b\,c^+)^+$.

Background (2) , from ”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 (3), from "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.

Tokens are defined by *regular expressions*:

- $\emptyset$, the empty set: a language with no strings
- ε, the empty string
- a, where $a \in \Sigma$ and Σ is our alphabet
- $M|N$, alternation: either M or N
- $M \cdot N$, concatenation: M followed by N
- M^*, zero or more occurences of M

where M and N are both regular expressions.

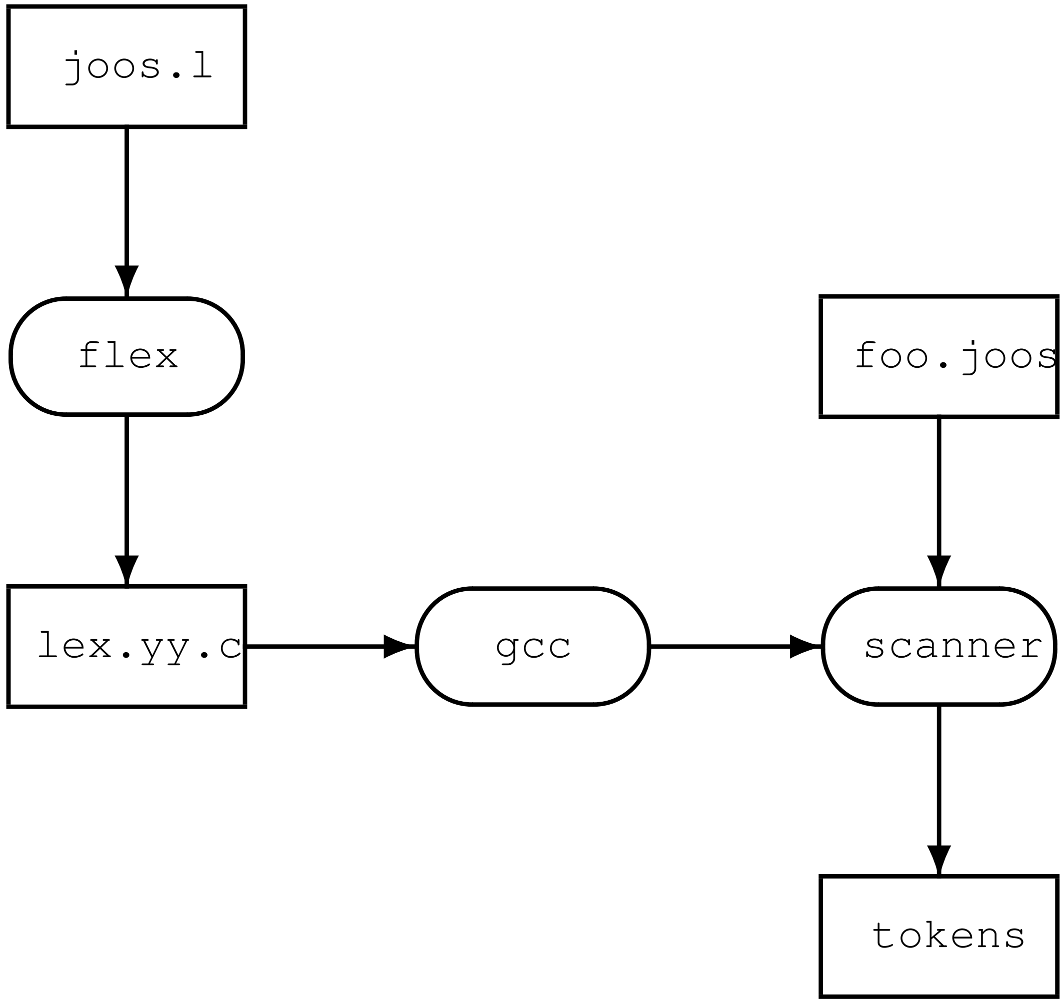
What are M? and M^+?

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]+`

A *scanner* or *lexer* transforms a string of characters into a string of tokens:

- uses a combination of *deterministic finite automata* (DFA);
- plus some glue code to make it work;
- can be generated by tools like `flex` (or `lex`), `JFlex`, ...


joos.l
flex
lex.yy.c
gcc
foo.joos
scanner
tokens

How to go from regular expressions to DFAs?

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

(see “Crafting a Compiler” , ch 3; or “Modern Compiler Implementation in Java”, Ch. 2)

Regular Expressions to NFA (1) from text, "Crafting a Compiler"

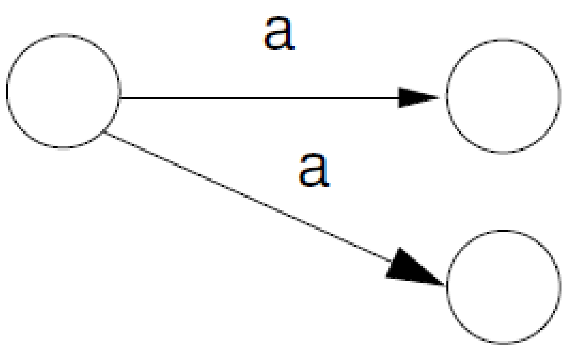


Figure 3.17: An NFA with two a transitions.

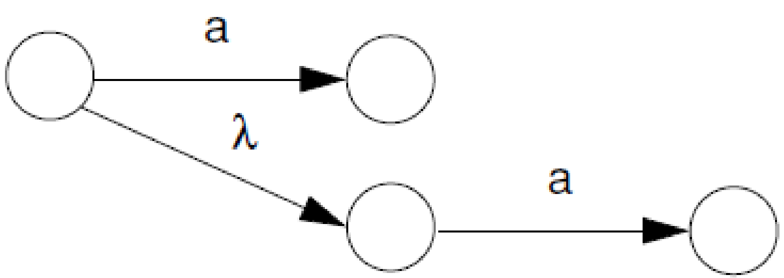


Figure 3.18: An NFA with a λ transition.

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

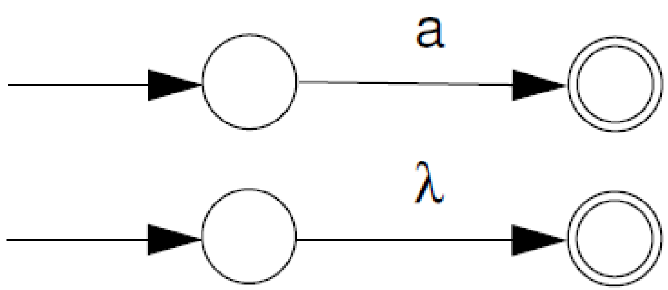


Figure 3.19: NFAs for a and λ.

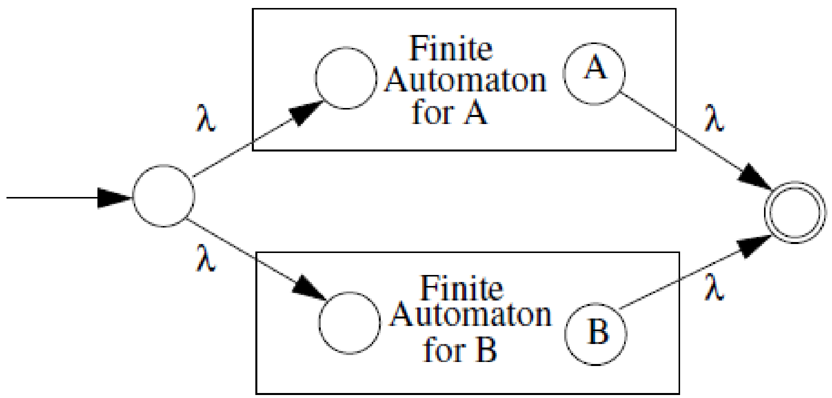


Figure 3.20: An NFA for $A \mid B$.

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

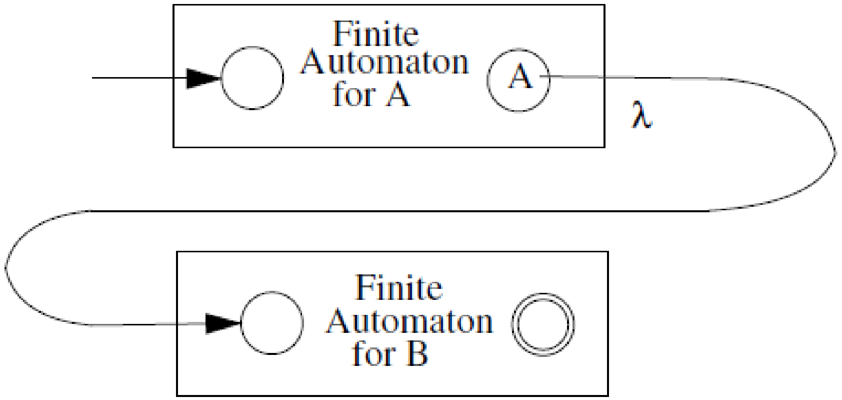


Figure 3.21: An NFA for AB.

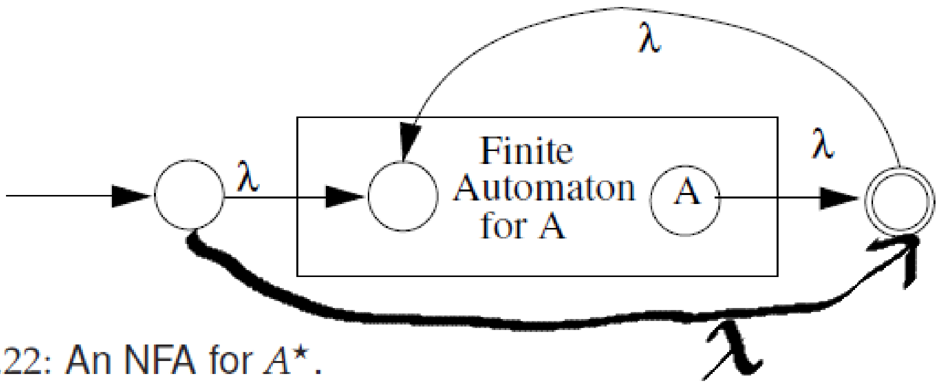
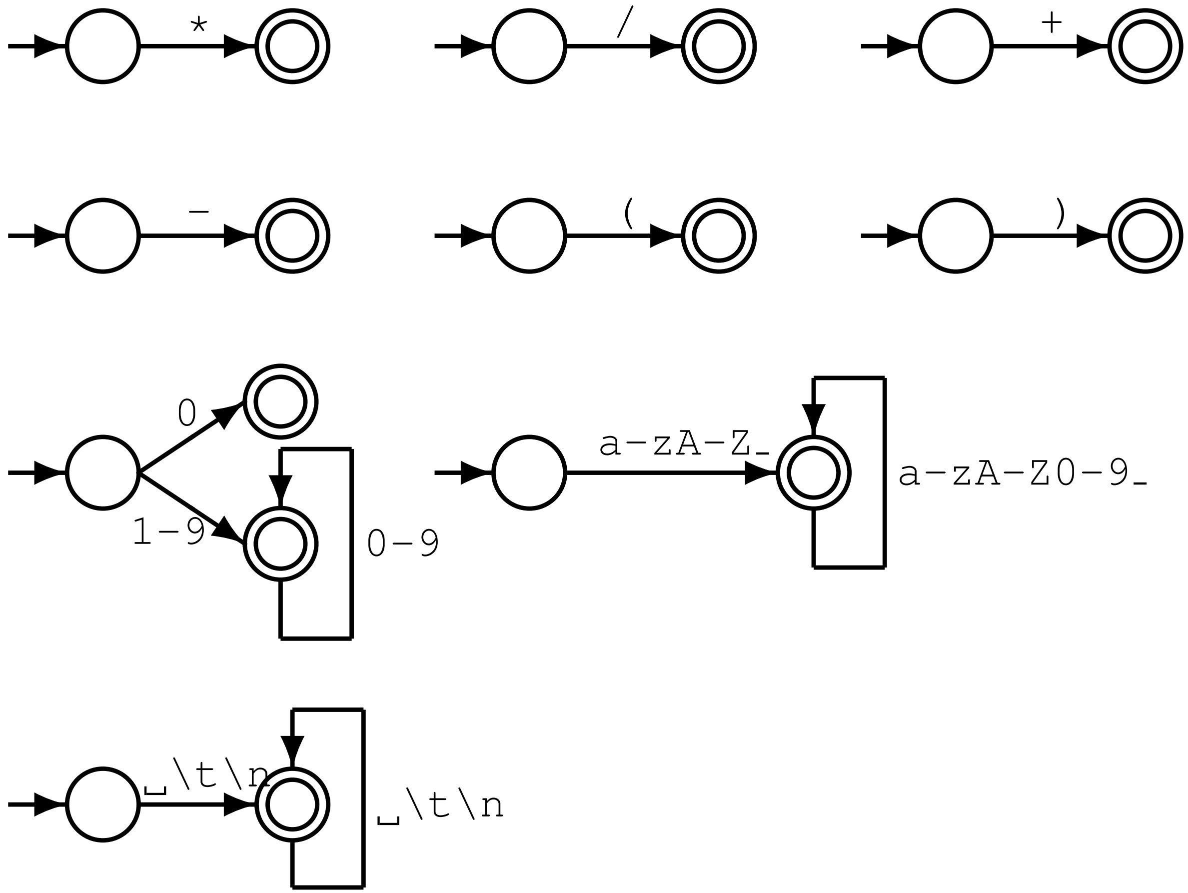


Figure 3.22: An NFA for $A^\star$.



Some DFAs

Each DFA has an associated *action*.

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

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

Want to match `‘‘importedFiles’’` as `tIDENTIFIER(importedFiles)` and not as `tIMPORT tIDENTIFIER(edFiles)`.

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

Why the “first match” principle?

Again — Example: keywords

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

Want to match ``continue foo'' as tCONTINUE tIDENTIFIER(foo) and not as 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)`

`flex` allows us to use look-ahead, using `'/'`:

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

Another example taken from FORTRAN, FORTRAN ignores whitespace

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

   in C: `do5i = 1.25;`

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

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

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

Case 1: flm analysis correct:

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

Case 2: want:

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

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

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

```
$ 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 ();
}
```

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

```
$ emacs print_tokens.l

$ flex print_tokens.l

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

When 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);
}
```

Remove vowels and increment integers:

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

%%
[aeiouy]        /* ignore */
[0-9]+          printf ("%i", atoi (yytext) + 1);

%%
main () {
  yylex ();
}
```