Compilers: Introduction and Scanners

a topic in

DM565 – Formal Languages and Data Processing

Kim Skak Larsen

Department of Mathematics and Computer Science (IMADA)
University of Southern Denmark (SDU)

kslarsen@imada.sdu.dk

September, 2023

Compilers

SDU

Typically, transforming high level constructs to low level constructs.

Ex: Compiling Java to Java bytecode or C to X86 Assembly.

There are many high-level languages, and more keep coming.

Many domain-specific languages require compiler technology, such as LaTeX, lex (flex), yacc (bison), html expansions, etc.

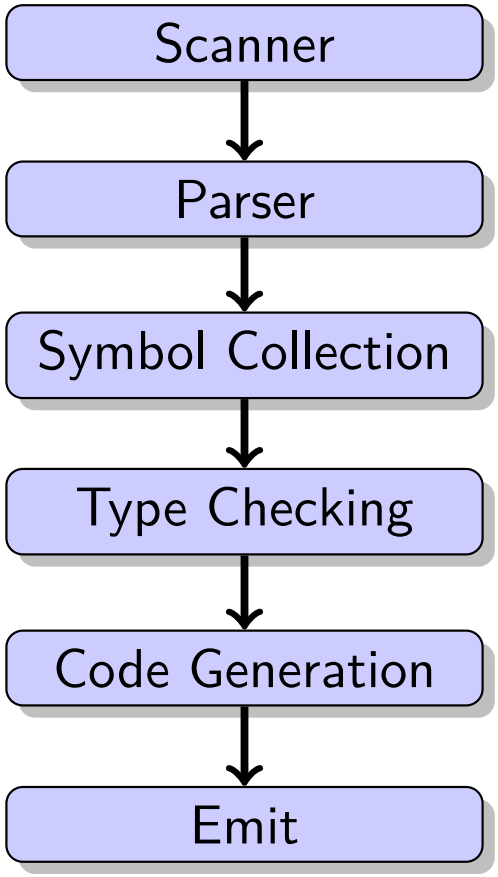
Many companies maintain their own collection of "compilers" for screen control, dbms interfaces, etc.

Jakob E. Bardram, Co-founder of Monsenso (on Nasdaq), September 14, 2021:

> *For a while, I thought that newer CS topics could replace older ones in the curricula. I was wrong! It's really important that they [the students] learn the classic material as well; compiler technology, for example.*

Compiler Phases

The Minimum

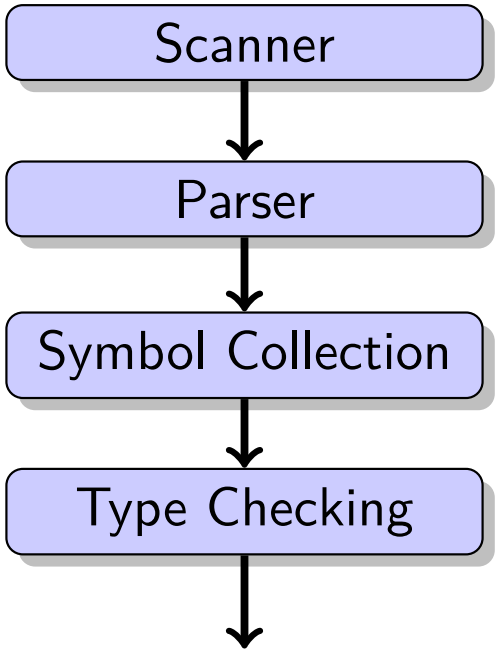


Compiler Phases

SDU

Front End

Analysis: "Ensuring that the input program is correct"



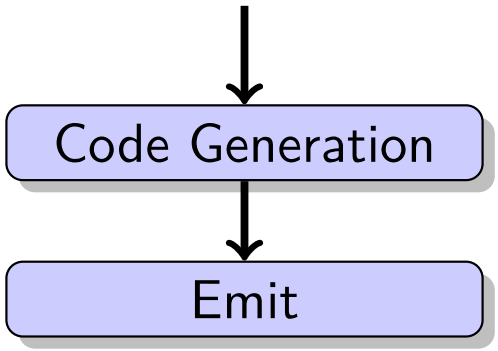
Weed phases can be inserted where required. They are for tasks *not* covered by the above, and therefore separate for modularity.

Compiler Phases

SDU

Back End

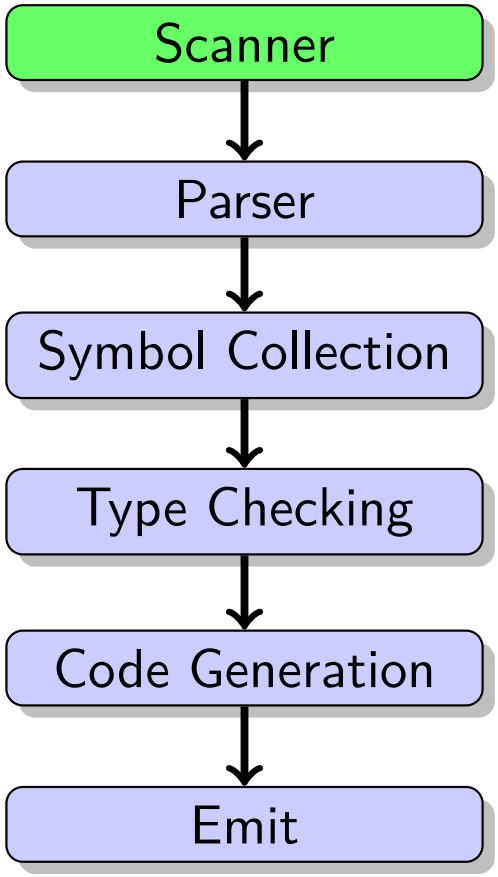
Synthesis: "Generating code for the correct input program"



Optimization phases can be inserted before code generation or after; important options include

- liveness analysis and register allocation
- peep-hole optimization
- garbage collection

Lexical Analysis: scanners



Lexical Analysis: scanners

SDU

Input to phase

A stream of characters (the user program).

Output from phase

A stream of lexical units.

Ex: **function**, **identifier** ("Fibonacci"), **(**, **identifier** ("n"), **:**, **int**, . . . , **LEQ**, **num** ("42"), . . .

Typically, want to ignore comments and whitespace (used as delimiters, but not output to next phase).

Overview of Lecture

1. How do we make software for this phase?
2. How do we use existing software for this (`flex`)?
3. How is it done in `scil`?

SDU

Overall Considerations

- Regular expressions is the most convenient formalism for specifying tokens: It is compact and we do not have to draw or specify large transition functions.
- DFAs are perfect for running the scanner: Simple, deterministic actions.
- Need: A tool that converts (a collection of) regular expressions to a DFA.
- A direct conversion is complicated; our tool will combine regular expressions into an NFA, which is then converted to a DFA.

Crafting a Scanner

Desired Functionality: Regular Expressions

a	An ordinary character stands for itself.
ϵ	The empty string.
	Another way to write the empty string.
$M \mid N$	Alternation, choosing from M or N.
$M \cdot N$	Concatenation, an M followed by an N.
MN	Another way to write concatenation.
M^*	Repetition (zero or more times).
M^+	Repetition, one or more times.
$M?$	Optional, zero or one occurrence of M.
$[\mathbf{a}-\mathbf{zA}-\mathbf{Z}]$	Character set alternation.
.	A period stands for any single character except newline.
`"a.+*"`	Quotation, a string in quotes stands for itself literally.

FIGURE 2.1. Regular expression notation.

Crafting a Scanner

Desired Functionality: Omni-Present Tokens

`if`	`{return IF;}`
`[a-z][a-z0-9]*`	`{return ID;}`
`[0-9]+`	`{return NUM;}`
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	`{return REAL;}`
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	`{ /* do nothing */ }`
`.`	`{error();}`

FIGURE 2.2. Regular expressions for some tokens.

SDU

Desired Functionality: Extent of Match

We do not want just *one* match; we want to split up the *entire* input into tokens using repeated, non-overlapping matches.

- Is `counter42` an identifier or an identifier follow by a number?
- Is `if42` an identifier or a keyword followed by a number?
- Is `if` an identifier or a keyword?

We resolve these issue with a prioritized list of decisions:

1. Longest match (from the input)
2. First match (in the definition file)

Crafting a Scanner

Desired Functionality: Omni-Present Tokens

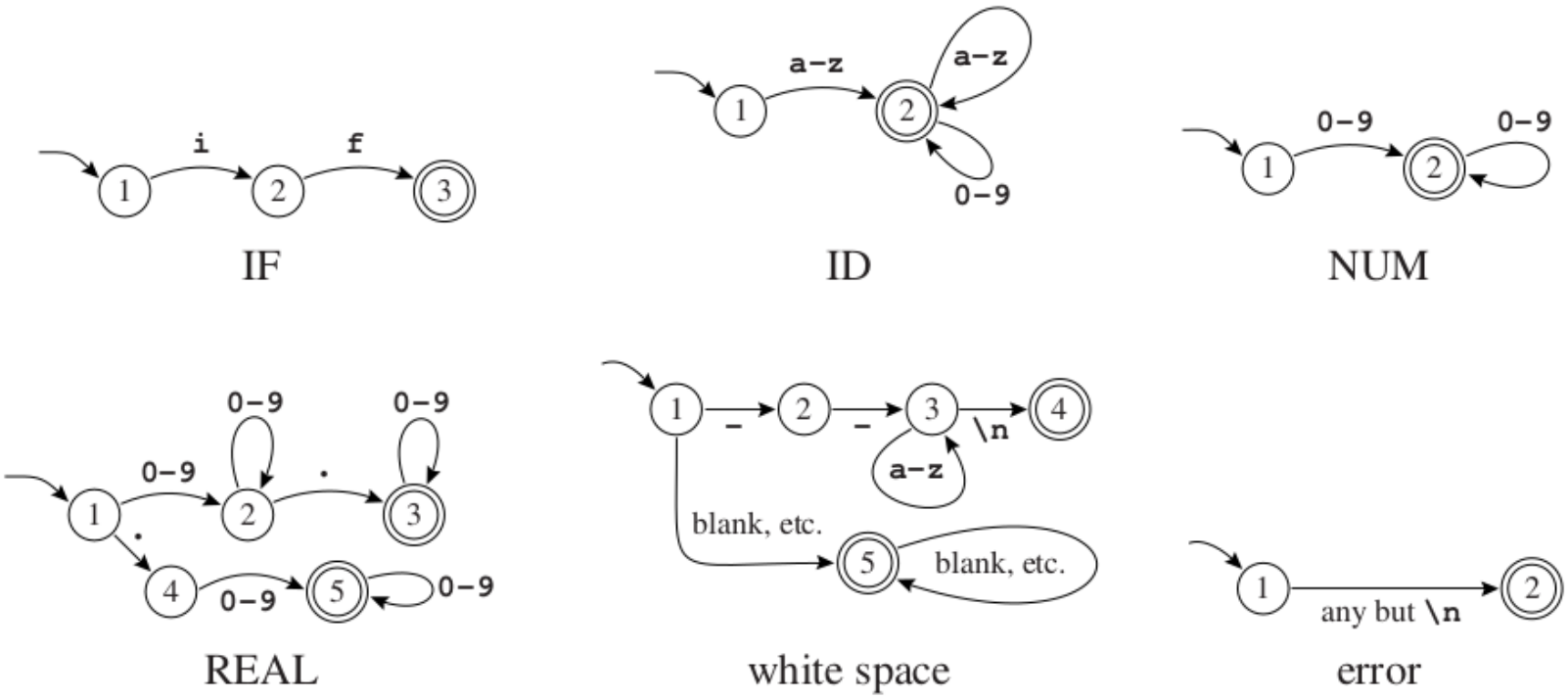


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

SDU

Ad Hoc Constructed DFA

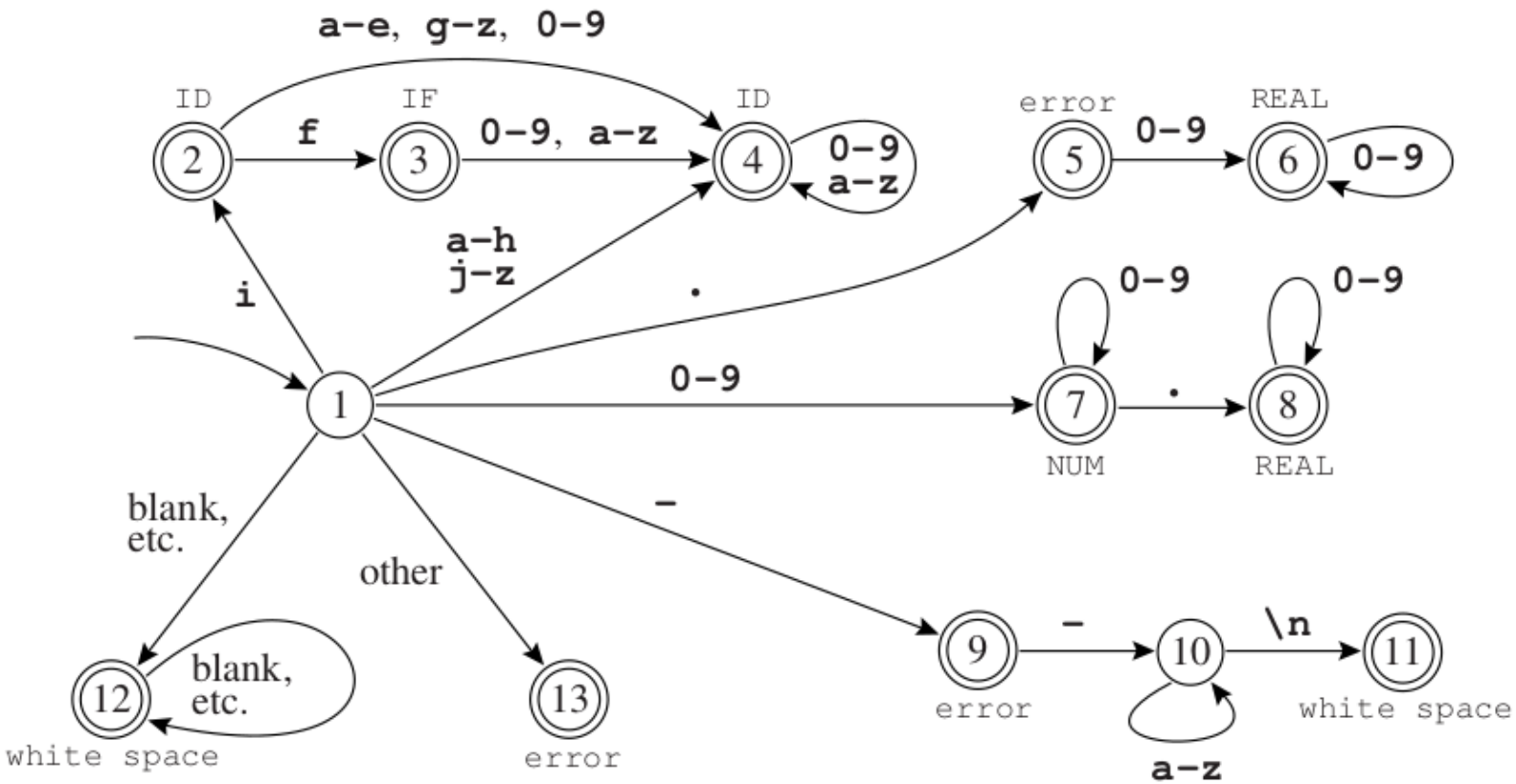


FIGURE 2.4. Combined finite automaton.

Crafting a Scanner

Running the DFA

Last Final	Current State	Current Input	Accept Action
0	1	⊥if --not-a-com	
2	2	|i⊥f --not-a-com	
3	3	|if⊤ --not-a-com	
3	0	|if⊤ ⊥--not-a-com	*return* IF
0	1	if⊤ --not-a-com	
12	12	if| ⊥--not-a-com	
12	0	if| ⊤-⊥not-a-com	*found white space; resume*
0	1	if ⊥--not-a-com	
9	9	if |-⊥-not-a-com	
9	10	if |-⊤-⊥not-a-com	
9	10	if |-⊤-n⊥ot-a-com	
9	10	if |-⊤-no⊥t-a-com	
9	10	if |-⊤-not⊥-a-com	
9	0	if |-⊤-not-⊥a-com	*error, illegal token '-'; resume*
0	1	if -⊥-not-a-com	
9	9	if -|-⊥not-a-com	
9	0	if -|-⊤n⊥ot-a-com	*error, illegal token '-'; resume*

FIGURE 2.5. The automaton of Figure 2.4 recognizes several tokens. The symbol | indicates the input position at each successive call to the lexical analyzer, the symbol ⊥ indicates the current position of the automaton, and ⊤ indicates the most recent position in which the recognizer was in a final state.

Crafting a Scanner

Combine to NFA

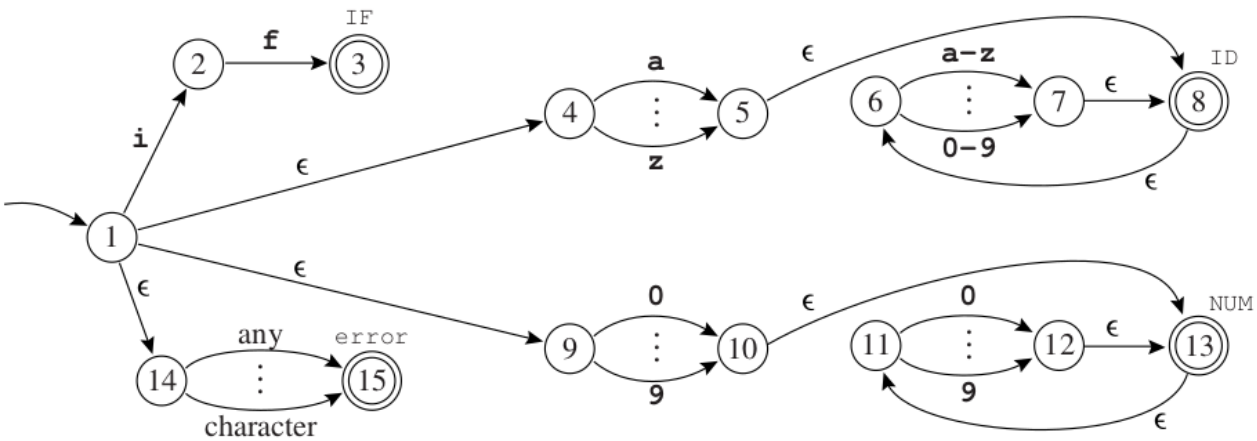


FIGURE 2.7. Four regular expressions translated to an NFA.

The approach is really very clean (the above has been postprocessed).

1. Make state names in the component DFAs unique.
2. Combine all components by introducing a new start state with ε-transitions to all start states for the individual components.
3. Mark the accepting states from each component with their token type.

Crafting a Scanner

Convert the NFA to a DFA

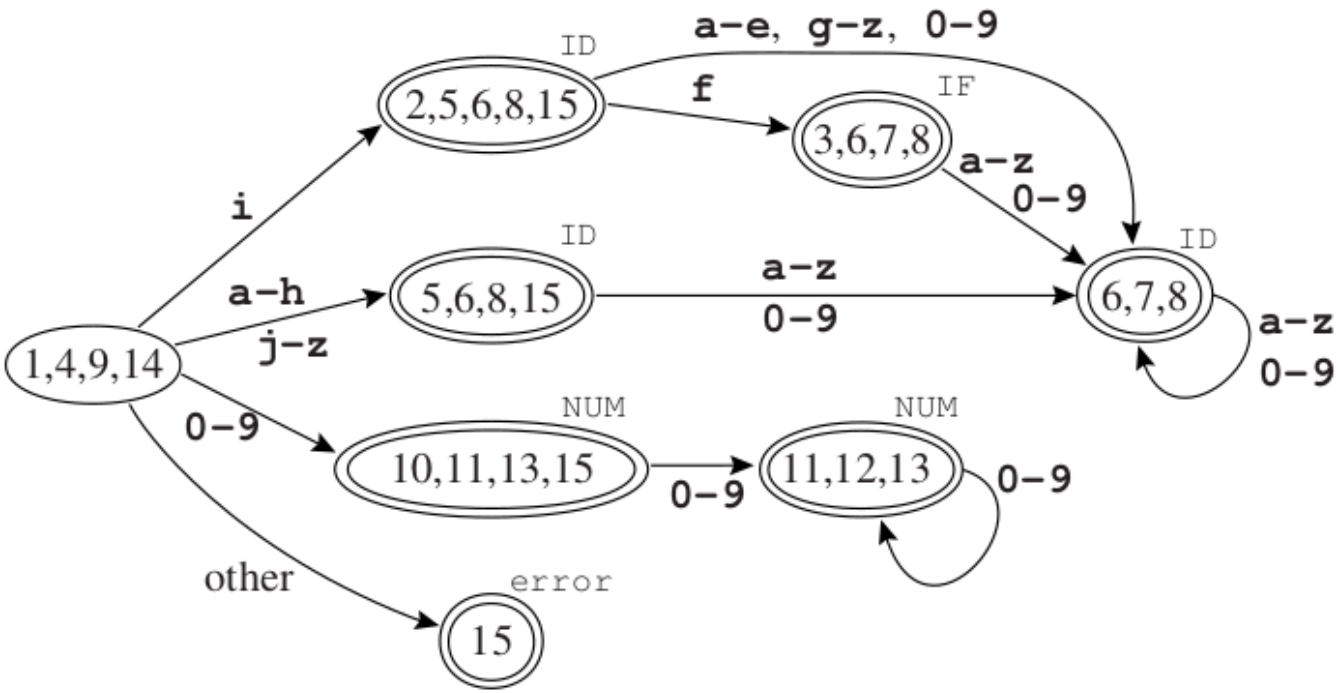


FIGURE 2.8. NFA converted to DFA.

Mark accept states with the token type listed first among all the accept states from the NFA that now make up the set of states in the DFA.

Flex

Fast **lex**ical analyser generator

Tool available for many programming languages: C, Java, ...

Same functionality available in Python with native syntax.

Flex

Format

```
%{
  C DEFINITIONS
%}
  FLEX DEFINITIONS
%%
  REGULAR EXPRESSIONS AND ACTIONS
%%
  C CODE
```

Special variables

- `yytext` – last matched string
- `yyleng` – length of last matched string
- `yylval` – associated value to the parser, e.g., when the token is INT, the value is passed on via `yylval`

SDU

Flex Example

- `first.l`

`%option noyywrap` tells `flex` that there is only one input file.

SDU

How to Run Flex

```
> flex FILENAME.l                -- makes lex.yy.c
> gcc lex.yy.c                   -- makes a.out
> ./a.out < INPUTFILE            -- running on the input in INPUT_FILE
```

The above is for a stand-alone application using `flex`. Later, we will see how to combine the `flex` scanner with the next phase of a compiler.

SDU

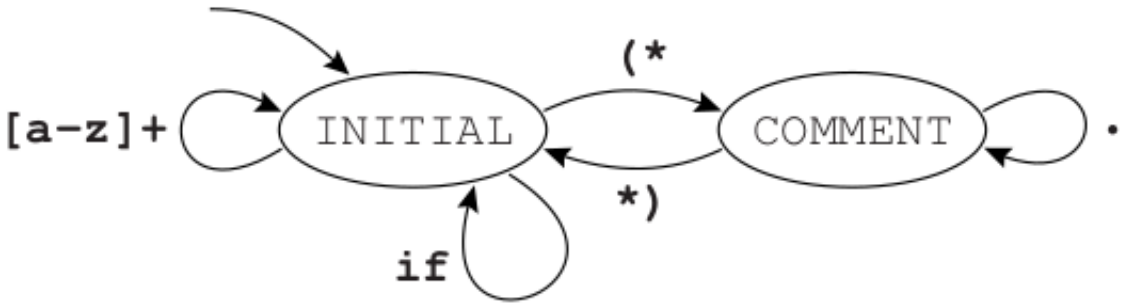
Flex Examples

- `counting.l`
- `weird.l`

Flex

Syntax rules varies from program parts to comments, strings, embedded database code, etc.

In `flex`, we can move between different states to use completely separate rule sets.



SDU

Realizing Multiple Flex States

State specifiers are first on the line, and the functionaly is realized via the keyword `BEGIN`.

```
    ⋮ the usual preamble ...
%Start INITIAL COMMENT
%%
<INITIAL>if       {ADJ; return IF;}
<INITIAL>[a-z]+ {ADJ; yylval.sval=String(yytext); return ID;}
<INITIAL>"(*"     {ADJ; BEGIN COMMENT;}
<INITIAL>.        {ADJ; EM_error("illegal character");}
<COMMENT>"*)"     {ADJ; BEGIN INITIAL;}
<COMMENT>.        {ADJ;}
.                 {BEGIN INITIAL; yyless(1);}
```

`yyless(1)` tells `flex` to back up one already read input character.

Scanner Construction in Python

In Python, the same functionality known from `flex` is realized using a more native Python code style using the module `ply.lex`.

See how it is done in `scil`...