Lexical Analysis

Textbook:Modern Compiler Design
Chapter 2.1

Extra Class

February 1 11-14
February 15 11-14
March 28 11-14

A motivating example

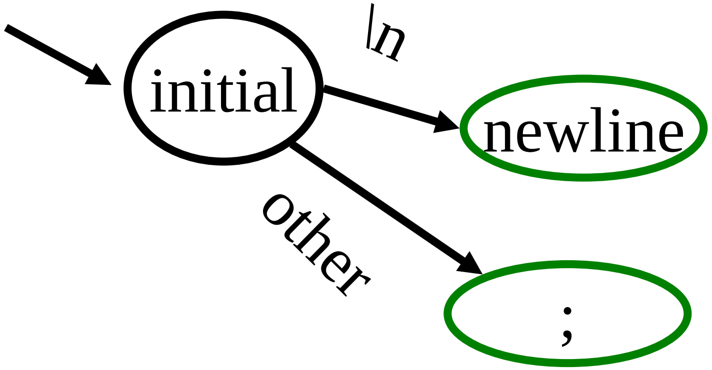
- Create a program that counts the number of lines in a given input text file

Solution (Flex)

```
            int num_lines = 0;
%%
\n      ++num_lines;
.       ;
%%
        main()
                {
                yylex();
                printf( "# of lines = %d\n", num_lines);
                }
```

Solution(Flex)

```
        int num_lines = 0;
%%
\n      ++num_lines;
.       ;
%%
      main()
          {
          yylex();
          printf( "# of lines = %d\n", num_lines);
          }
```



JLex Spec File

User code

- Copied directly to Java file

Possible source of javac errors down the road

%%

JLex directives

- Define macros, state names

DIGIT= [0-9]
LETTER= [a-zA-Z]

YYINITIAL

%%

Lexical analysis rules

- Optional state, regular expression, action
- How to break input to tokens
- Action when token matched

{LETTER}
({LETTER}|{DIGIT})*

Jlex linecount

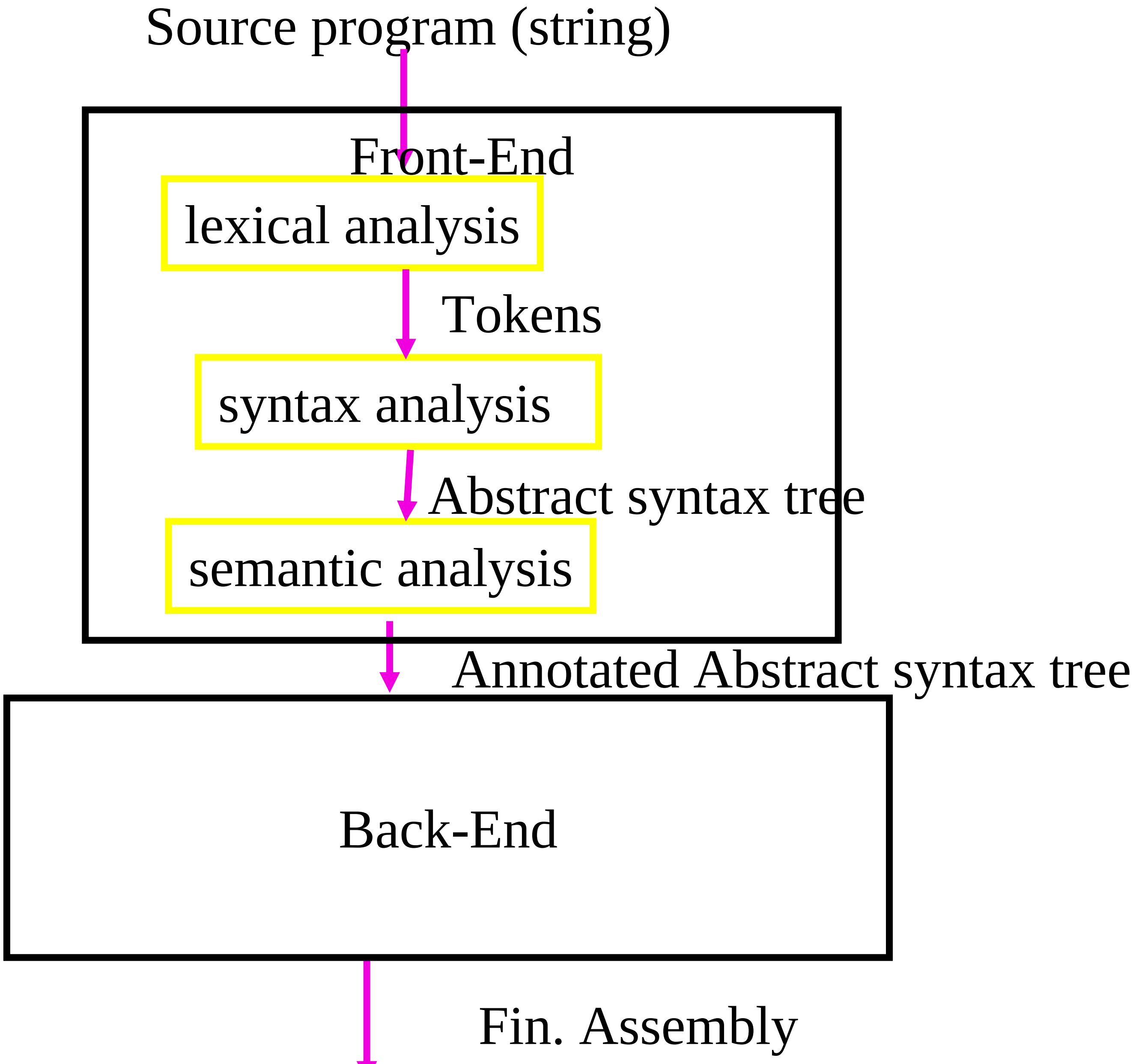
File: lineCount

```
import java_cup.runtime.*;
%%
%cup
%{
  private int lineCounter = 0;
%}

%eofval{
  System.out.println("line number=" + lineCounter);
  return new Symbol(sym.EOF);
%eofval}

NEWLINE=\n
%%
{NEWLINE} {
         lineCounter++;
}
[^{NEWLINE}] { }
```

Outline

- Roles of lexical analysis
- What is a token
- Regular expressions and regular descriptions
- Lexical analysis
- Automatic Creation of Lexical Analysis
- Error Handling

Basic Compiler Phases



Source program (string)

Front-End

lexical analysis

Tokens

syntax analysis

Abstract syntax tree

semantic analysis

Annotated Abstract syntax tree

Back-End

Fin. Assembly

Example Tokens

Type	Examples
ID	foo n_14 last
NUM	73 00 517 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

Example Non Tokens

Type	Examples
comment	/* ignored */
preprocessor directive	#include <foo.h>
	#define NUMS 5, 6
macro	NUMS
whitespace	\t \n \b

Example

```
void match0(char *s) /* find a zero */
{
        if (!strncmp(s, “0.0”, 3))
                return 0. ;
}
```

VOID ID(match0) LPAREN CHAR DEREF ID(s)

RPAREN LBRACE IF LPAREN NOT ID(strncmp) LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3)

RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE EOF

Lexical Analysis (Scanning)

- input
  - program text (file)
- output
  - sequence of tokens
- Read input file
- Identify language keywords and standard identifiers
- Handle include files and macros
- Count line numbers
- Remove whitespaces
- Report illegal symbols
- [Produce symbol table]

Why Lexical Analysis

- Simplifies the syntax analysis
  - And language definition
- Modularity
- Reusability
- Efficiency

What is a token?

- Defined by the programming language
- Can be separated by spaces
- Smallest units
- Defined by regular expressions

A simplified scanner for C

```
Token nextToken()
{
char c ;
loop: c = getchar();
switch (c){
          case `  `:goto loop ;
          case `;`:  return SemiColumn;
          case `+`:  c = getchar() ;
                  switch (c) {
                     case `+': return PlusPlus ;
                     case '='  return PlusEqual;
                     default:  ungetc(c);
                               return Plus;                          }
           case `<`:
          case `w`:
 }
```

Regular Expressions

Basic patterns	Matching
x	The character x
.	Any character expect newline
[xyz]	Any of the characters x, y, z
R?	An optional R
R*	Zero or more occurrences of R
R+	One or more occurrences of R
R_1R_2	R_1 followed by R_2
$R_1\|R_2$	Either R_1 or R_2
(R)	R itself

Escape characters in regular expressions

- \ converts a single operator into text
  - a\+
  - (a\+*)+
- Double quotes surround text
  - “a+*”+
- Esthetically ugly
- But standard

Regular Descriptions

- EBNF where non-terminals are fully defined before first use

  letter →[a-zA-Z]
  digit →[0-9]
  underscore →_
  letter_or_digit → letter|digit
  underscored_tail → underscore letter_or_digit+
  identifier → letter letter_or_digit* underscored_tail

- token description
  - A token name
  - A regular expression

The Lexical Analysis Problem

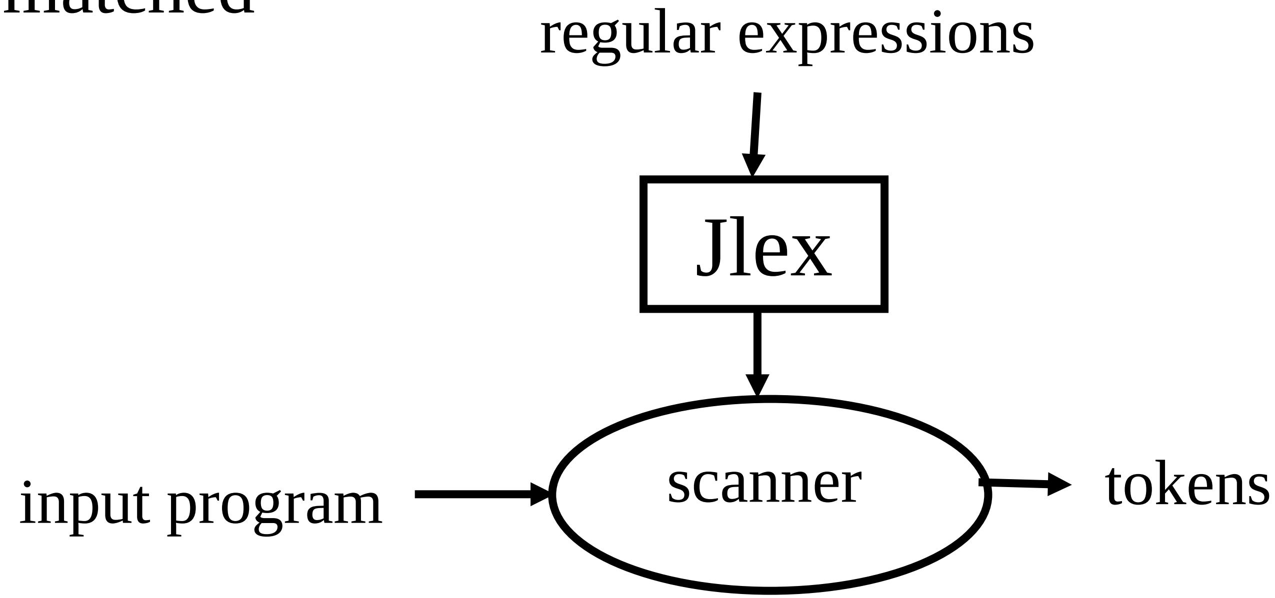
- Given
  - A set of token descriptions
  - An input string
- Partition the strings into tokens (class, value)
- Ambiguity resolution
  - The longest matching token
  - Between two equal length tokens select the first

A Jlex specification of C Scanner

```
import java_cup.runtime.*;
%%
%cup
%{
  private int lineCounter = 0;
%}
Letter= [a-zA-Z_]
Digit= [0-9]
%%
”\t” { }
”\n”    { lineCounter++; }
“;”     { return new  Symbol(sym.SemiColumn);}
“++”   {return new  Symbol(sym.PlusPlus); }
“+=”   {return new  Symbol(sym.PlusEq); }
“+”     {return new  Symbol(sym.Plus); }
“while”  {return new  Symbol(sym.While); }
{Letter}({Letter}|{Digit})*
                {return new  Symbol(sym.Id, yytext() ); }
“<=”     {return new  Symbol(sym.LessOrEqual); }
“<”     {return new  Symbol(sym.LessThan); }
```

Jlex

- Input
  - regular expressions and actions (Java code)
- Output
  - A scanner program that reads the input and applies actions when input regular expression is matched



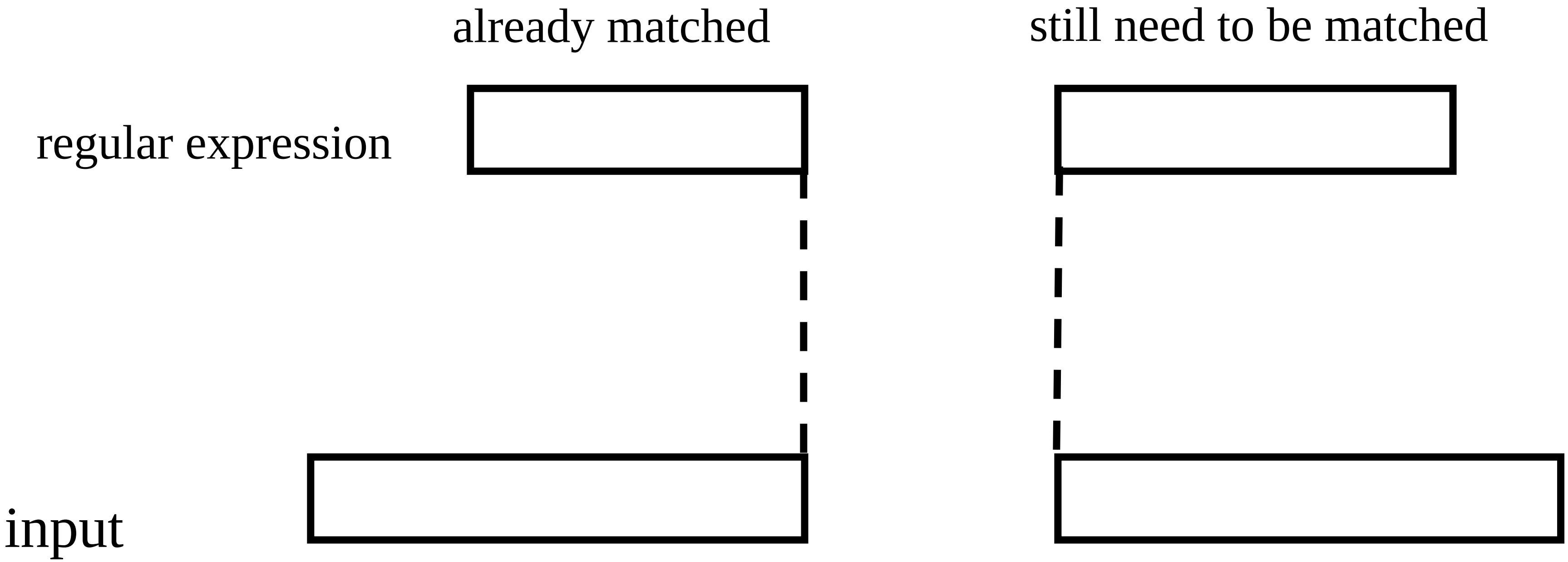
Naïve Lexical Analysis

```
SET the global token (Token .class, Token .length) to (0, 0);
FOR EACH Length SUCH THAT the input matches $T_1 \rightarrow R_1$
    IF LENGTH > TOKEN .length
        SET (Token .class, Token .length) TO ($T_1$, Length)
FOR EACH Length SUCH THAT the input matches $T_2 \rightarrow R_2$
    IF LENGTH > TOKEN .length
        SET (Token .class, Token .length) TO (T2, Length)
...
FOR EACH Length SUCH THAT the input matches $T_n \rightarrow R_n$
    IF LENGTH > TOKEN .length
        SET (Token .class, Token .length) TO ($T_n$, Length)
IF TOKEN .length = 0 handle non matching character
```

Automatic Creation of Efficient Scanners

- Naïve approach on regular expressions (dotted items)
- Construct non deterministic finite automaton over items
- Convert to a deterministic
- Minimize the resultant automaton
- Optimize (compress) representation

Dotted Items



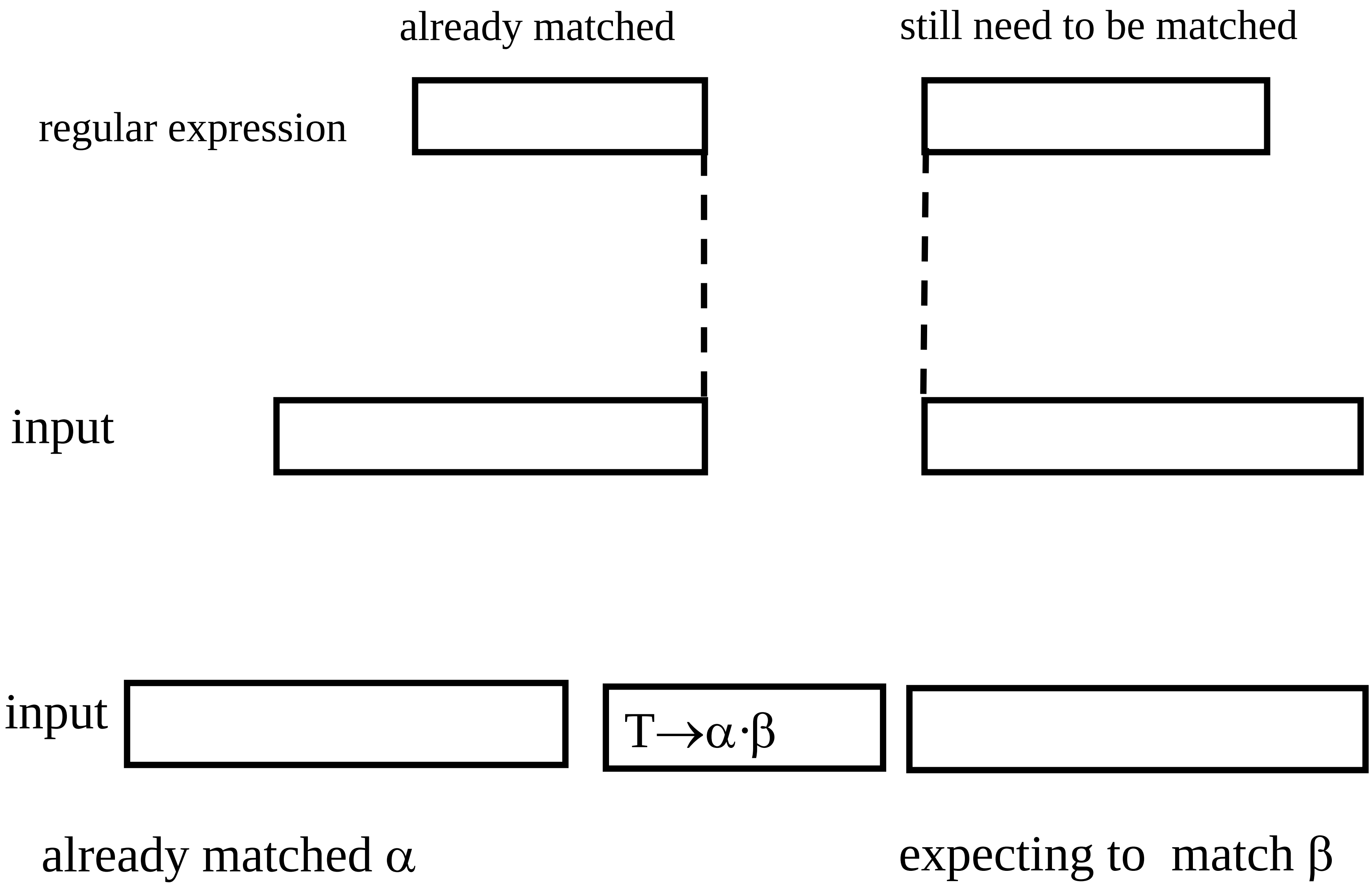
already matched

still need to be matched

regular expression

input

Dotted Items



already matched

still need to be matched

regular expression

input

input

T→α·β

already matched α

expecting to match β

Example

- $T \to a+\ b+$
- Input ‘aab’
- After parsing ‘aa’
  - $T \to a+ \cdot b+$

Item Types

- Shift item
  - · In front of a basic pattern
  - A → (ab)+ · c (de|fe)*
- Reduce item
  - · At the end of rhs
  - A → (ab)+ c (de|fe)* ·
- Basic item
  - Shift or reduce items

Character Moves

- For shift items character moves are simple

$T \rightarrow \alpha \cdot c\ \beta \quad c \quad \Rightarrow \quad c \quad T \rightarrow \alpha\ c \cdot \beta$

$\text{Digit} \rightarrow \cdot [0\text{-}9] \quad 7 \quad \Rightarrow \quad 7 \quad T \rightarrow [0\text{-}9] \cdot$

ε Moves

- For non-shift items the situation is more complicated
- What character do we need to see?
- Where are we in the matching?
  $T \rightarrow \cdot\ a^*$
  $T \rightarrow \cdot\ (a^*)$

ε Moves for Repetitions

- Where can we get from $T \rightarrow \alpha \cdot (R)^* \beta$?
- If R occurs zero times

  $T \rightarrow \alpha\ (R)^* \cdot \beta$
- If R occurs one or more times

  $T \rightarrow \alpha\ (\cdot R)^*\ \beta$
  - When R ends $\alpha\ (R \cdot)^*\ \beta$
    - $\alpha\ (R)^* \cdot \beta$
    - $\alpha\ (\cdot R)^*\ \beta$

ε Moves

$T \rightarrow \alpha \cdot (R)^* \beta$ $\Rightarrow$ $T \rightarrow \alpha (R)^* \cdot \beta$

$T \rightarrow \alpha (\cdot R)^* \beta$

$T \rightarrow \alpha (R \cdot)^* \beta$ $\Rightarrow$ $T \rightarrow \alpha (R)^* \cdot \beta$

$T \rightarrow \alpha (\cdot R)^* \beta$

$T \rightarrow \alpha \cdot (R_1|R_2)\ \beta$ $\Rightarrow$ $T \rightarrow \alpha (.R_1|R_2)\ \beta$

$T \rightarrow \alpha (R_1|\ .R_2)\ \beta$

$T \rightarrow \alpha (R_1\ .\ |R_2)\ \beta$ $\Rightarrow$ $T \rightarrow \alpha (R_1\ |R_2) \cdot \beta$

$T \rightarrow \alpha (R_1\ |R_2\ .\ )\ \beta$ $\Rightarrow$ $T \rightarrow \alpha (R_1\ |R_2) \cdot \beta$

Input ‘3.1;’

I → [0-9]+

F →[0-9]*’.’[0-9]+

I → ·([0-9])+

I → (.[0-9])+

I → ([0-9] .)+

I → (.[0-9])+

I → ([0-9])+.

Input ‘3.1;’

I → [0-9]+

F →[0-9]*’.’[0-9]+

F → ·([0-9])*’.’([0-9])+

F → (·[0-9])*’.’([0-9])+

F → ([0-9])* ·’.’([0-9])+

F → ([0-9] ·)*’.’([0-9])+

F → (· [0-9])*’.’([0-9])+

F → ([0-9])* ·’.’([0-9])+

F → ([0-9])*’.’ ·([0-9])+

F → ([0-9])*’.’ (·[0-9])+

F → ([0-9])*’.’ ([0-9] ·)+

F → ([0-9])*’.’ (· [0-9])+

F → ([0-9])*’.’ ([0-9])+ ·

Concurrent Search

- How to scan multiple token classes in a single run?

Input ‘3.1;’

I → [0-9]+

F →[0-9]*’.’[0-9]+

I → ·([0-9])+

F → ·([0-9])*’.’([0-9])+

I → (·[0-9])+

F → (·[0-9])*’.’([0-9])+

F → ([0-9])* ·’.’([0-9])+

I → ([0-9] ·)+

F → ([0-9] ·)*’.’([0-9])+

I → (·[0-9])+

I → ([0-9])+ ·

F → (·[0-9])*’.’([0-9])+

F → ([0-9])* ·’.’([0-9])+

F → ([0-9])*’.’ ·([0-9])+

A Non-Deterministic Finite State Machine

- Add a production $S' \rightarrow T_1 \mid T_2 \mid \dots \mid T_n$
- Construct NDFA over the items
  - Initial state $S' \rightarrow \cdot\, (T_1 \mid T_2 \mid \dots \mid T_n)$
  - For every character move, construct a character transition

    $<T \rightarrow \alpha \cdot c\, \beta, c> \Rightarrow T \rightarrow \alpha\ \ c \cdot \beta$
  - For every ε move construct an ε transition
  - The accepting states are the reduce items
  - Accept the language defined by T_i

ε Moves

$T \rightarrow \alpha \cdot (R)^* \ \beta \quad \Rightarrow \quad T \rightarrow \alpha (R)^* \cdot \beta$

$\qquad\qquad\qquad\qquad\qquad\qquad T \rightarrow \alpha (\cdot R)^* \ \beta$

$T \rightarrow \alpha (R \cdot)^* \ \beta \quad \Rightarrow \quad T \rightarrow \alpha (R)^* \cdot \beta$

$\qquad\qquad\qquad\qquad\qquad\qquad T \rightarrow \alpha (\cdot R)^* \ \beta$

$T \rightarrow \alpha \cdot (R_1 | R_2) \ \beta \quad \Rightarrow \quad T \rightarrow \alpha (.R_1 | R_2) \ \beta$

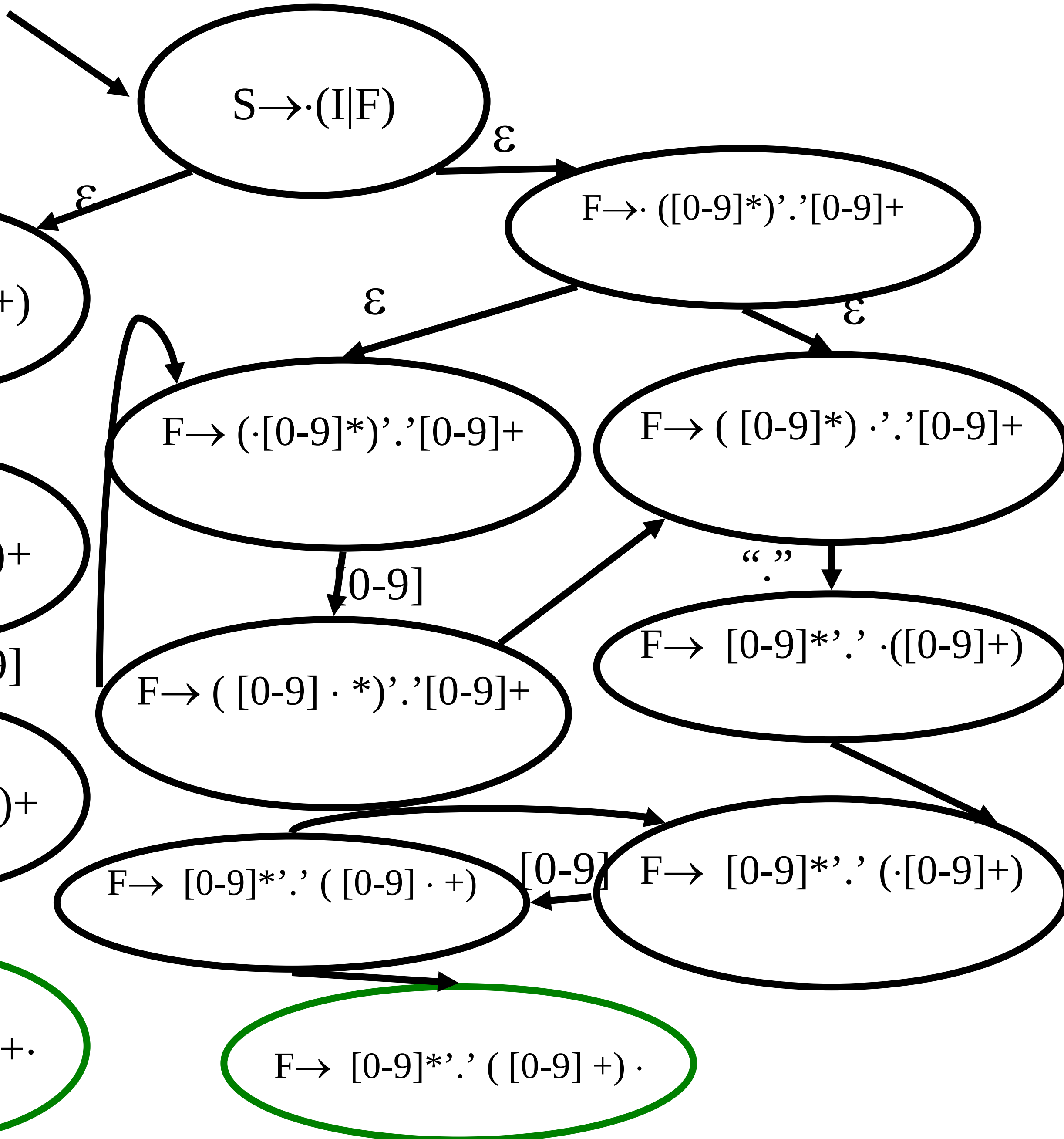
$\qquad\qquad\qquad\qquad\qquad\qquad T \rightarrow \alpha (R_1 | \ .R_2) \ \beta$

$T \rightarrow \alpha (R_1 \ . \ | R_2) \ \beta \quad \Rightarrow \quad T \rightarrow \alpha (R_1 | R_2) \cdot \beta$

$T \rightarrow \alpha (R_1 | R_2 \ . \ ) \ \beta \quad \Rightarrow \quad T \rightarrow \alpha (R_1 | R_2) \cdot \beta$

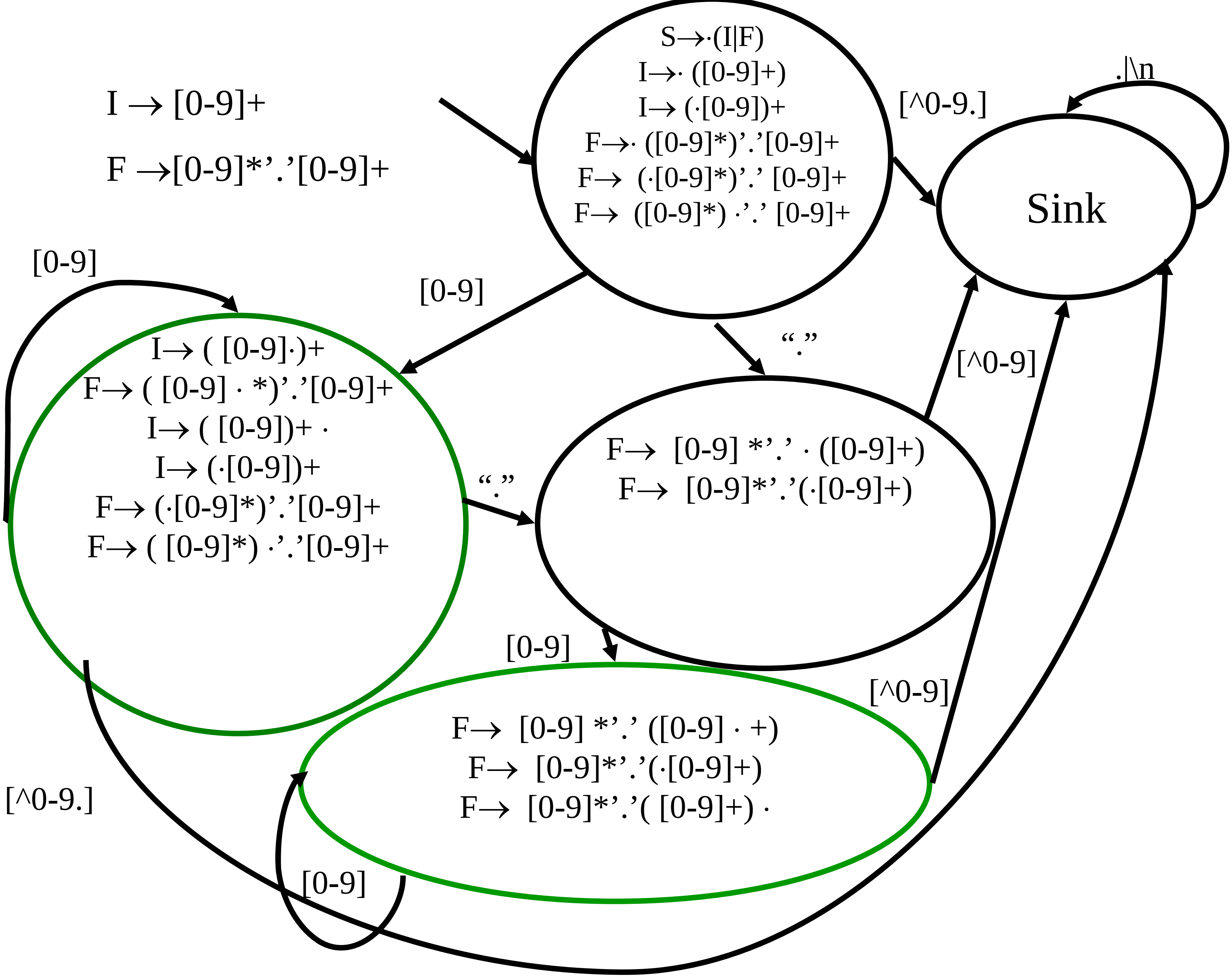
I → [0-9]+

F →[0-9]*’.’[0-9]+


S→·(I|F)
ε
F→· ([0-9]*)’.’[0-9]+
ε
I→· ([0-9]+)
ε
ε
ε
I→ (·[0-9])+
F→ (·[0-9]*)’.’[0-9]+
F→ ([0-9]*) ·’.’[0-9]+
[0-9]
“.”
[0-9]
I→ ([0-9]·)+
F→ ([0-9] · *)’.’[0-9]+
F→ [0-9]*’.’ ·([0-9]+)
F→ [0-9]*’.’ ([0-9] · +)
[0-9]
F→ [0-9]*’.’ (·[0-9]+)
I→ ([0-9])+·
F→ [0-9]*’.’ ([0-9] +) ·

Efficient Scanners

- Construct Deterministic Finite Automaton
  - Every state is a set of items
  - Every transition is followed by an ε-closure
  - When a set contains two reduce items select the one declared first
- Minimize the resultant automaton
  - Rejecting states are initially indistinguishable
  - Accepting states of the same token are indistinguishable
- Exponential worst case complexity
  - Does not occur in practice
- Compress representation


I → [0-9]+
F →[0-9]*'.'[0-9]+
S→·(I|F)
I→· ([0-9]+)
I→ (·[0-9])+
F→· ([0-9]*)'.'[0-9]+
F→ (·[0-9]*)'.' [0-9]+
F→ ([0-9]*) ·'.' [0-9]+
[^0-9.]
.|\n
Sink
[0-9]
[0-9]
"."
[^0-9]
I→ ([0-9]·)+
F→ ([0-9] · *)'.'[0-9]+
I→ ([0-9])+ ·
I→ (·[0-9])+
F→ (·[0-9]*)'.'[0-9]+
F→ ([0-9]*) ·'.'[0-9]+
"."
F→ [0-9] *'.' · ([0-9]+)
F→ [0-9]*'.'(·[0-9]+)
[0-9]
[^0-9]
F→ [0-9] *'.' ([0-9] · +)
F→ [0-9]*'.'(·[0-9]+)
F→ [0-9]*'.'([0-9]+) ·
[^0-9.]
[0-9]

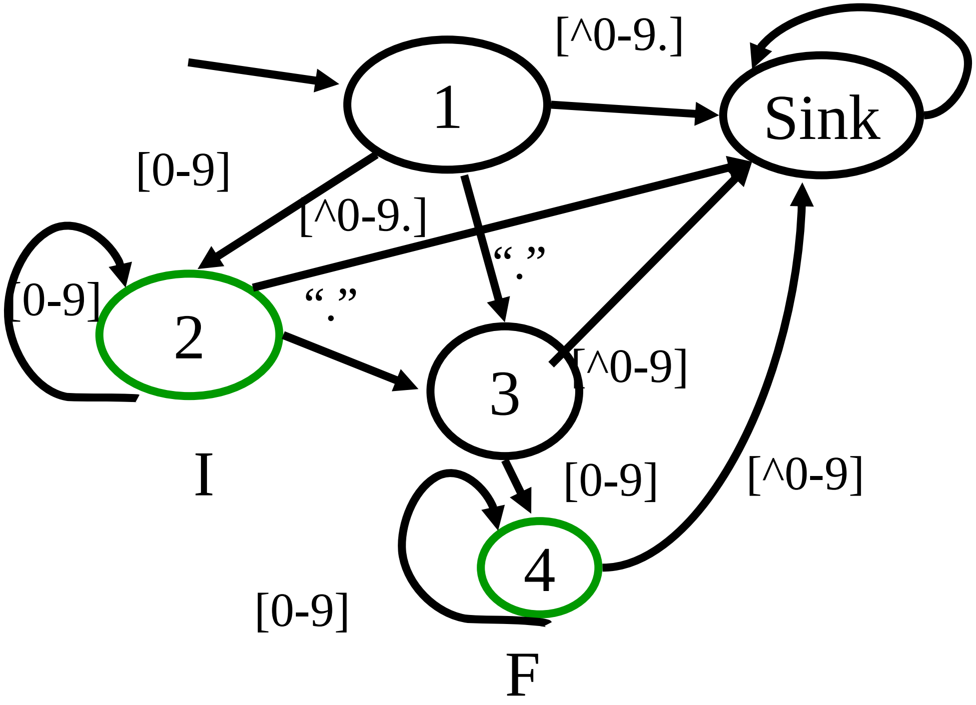
A Linear-Time Lexical Analyzer

```
IMPORT Input Char [1..];
Set Read Index To 1;
```

```
Procedure Get_Next_Token;
    set Start of token to Read Index;
    set End of last token to uninitialized
    set Class of last token to uninitialized
    set State to Initial
    while state /= Sink:
        Set ch to Input Char[Read Index];
        Set state = δ[state, ch];
        if accepting(state):
            set Class of last token to Class(state);
            set End of last token to Read Index
        set Read Index to Read Index + 1;
    set token .class to Class of last token;
    set token .repr to char[Start of token .. End last token];
    set Read index to End last token + 1;
```

Scanning “3.1;”

input	state	next state	last token
↓3.1;	1	2	I
3 ↓.1;	2	3	I
3. ↓1;	3	4	F
3.1 ↓;	4	Sink	F



The Need for Backtracking

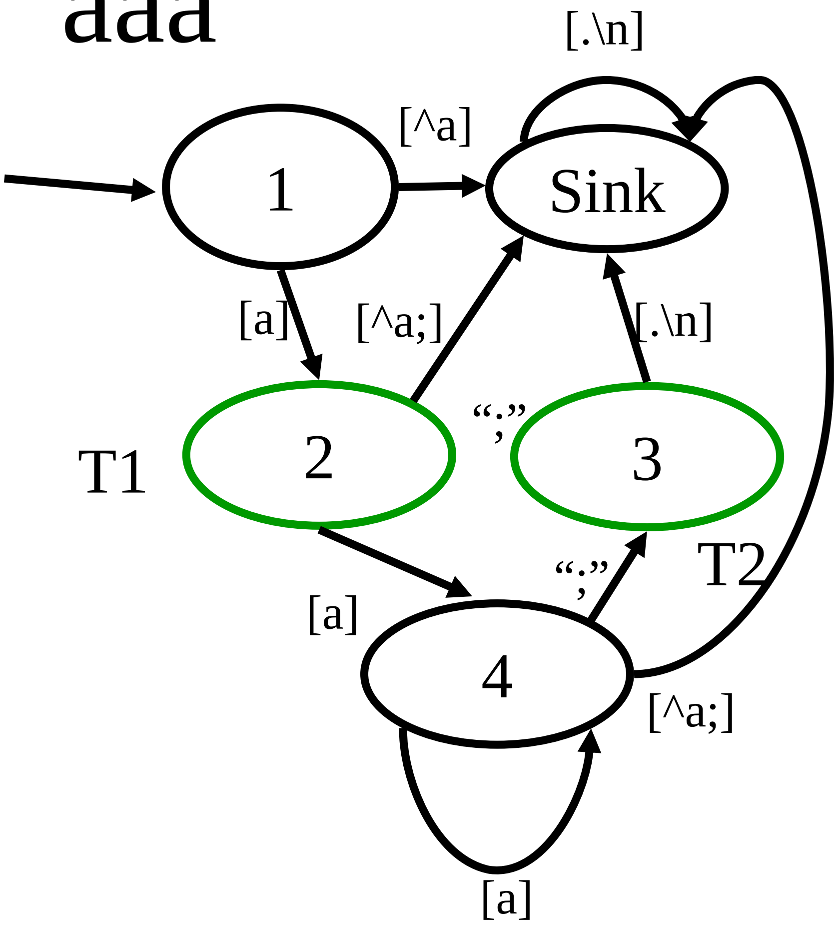
- A simple minded solution may require unbounded backtracking
  $T_1 \rightarrow a+;$
  $T_2 \rightarrow a$
- Quadratic behavior
- Does not occur in practice
- A linear solution exists

Scanning “aaa”

T1 → a+”;”

T2 →a

input	state	next state	last token
↓aaa$	1	2	T2
a ↓ aa$	2	4	T2
a a ↓ a$	4	4	T2
a a a ↓ $	4	Sink	T2



Error Handling

- Illegal symbols
- Common errors

Missing

- Creating a lexical analysis by hand
- Table compression
- Symbol Tables
- Handling Macros
- Start states
- Nested comments

Summary

- For most programming languages lexical analyzers can be easily constructed automatically
- Exceptions:
  - Fortran
  - PL/1
- Lex/Flex/Jlex are useful beyond compilers