2. Symbol specifications and lexical analysis

Notations of tokens is specified by regular expressions

Token classes: keywords (**`for`**, **`class`**), operators and delimiters (**`+`**, **`==`**, **`;`**, **`{`**), identifiers (**`getSize`**, **`maxint`**), literals (**`42`**, **`'\n'`**)

Lexical analysis isolates tokens within a stream of characters and encodes them:

Tokens

`int`	`count`	`=`	`0`	`;`	`double`	`sum`	`=`	`0.0`	`;`	`while`	`(`	`count`	`<`	`maxVect`	`)`	`{`	`sum`	`=`	`sum`	`+`	`vect`	`[`	`count`	`]`	`;`	`count`	`++`	`;`	`}`

Lexical Analysis

Input: ***Program represented by a sequence of characters***

Tasks:	**Compiler modul:**
	Input reader
Recognize and classify tokens Skip irrelevant characters	Scanner (central phase, finite state machine)
Encode tokens:	
	Identifier modul
Store token information	Literal modules
Conversion	String storage

Output: ***Program represented by a sequence of encoded tokens***

Avoid context dependent token specifications

Tokens should be **recognized in isolation**:

e. G. all occurrences of the identifier **a** get the same encoding:

```
{int a; ... a = 5; ... {float a; ... a = 3.1; ...}}
```

distinction of the two different variables would require information from semantic analysis

typedef problem in C:

The C syntax requires **lexical distinction** of type-names and other names:

```
typedef int *T; T (*B); X (*Y);
```

cause syntactically different structures: declaration of variable **B** and call of function **X**.
Requires feedback from semantic analysis to lexical analysis.

Identifiers in PL/1 may **coincide with keywords**:

```
if if = then then then := else else else := then
```

Lexical analysis needs feedback from syntactic analysis to distinguish them.

Token separation in FORTRAN:

„Deletion or insertion of blanks does not change the meaning.“

DO 24 K = 1,5 begin of a loop, 7 tokens

DO 24 K = 1.5 assignment to the variable **DO24K**, 3 tokens

Token separation is determined late.

Representation of tokens

Uniform encoding of tokens by triples:

Syntax code	**attribute**	**source position**
terminal code of the concrete syntax	value or reference into data module	to locate error messages of later compiler phases

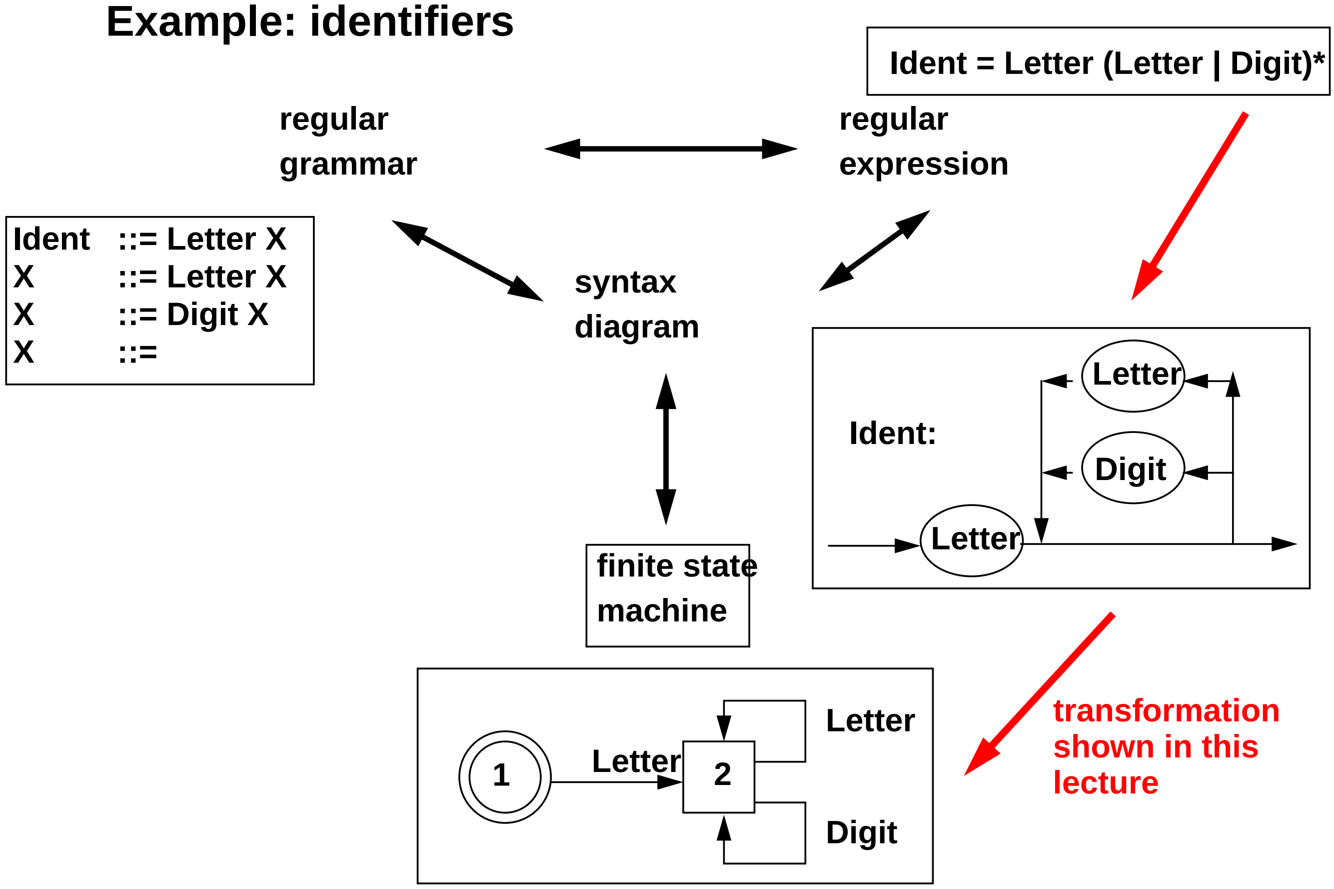
Examples:

```
double sum = 5.6e-5;
while (count < maxVect)
{ sum = sum + vect[count];
```

DoubleToken		12, 1
Ident	138	12, 8
Assign		12, 12
FloatNumber	16	12, 14
Semicolon		12, 20
WhileToken		13, 1
OpenParen		13, 7
Ident	139	13, 8
LessOpr		13, 14
Ident	137	13, 16
CloseParen		13, 23
OpenBracket		14, 1
Ident	138	14, 3

Specification of token notations

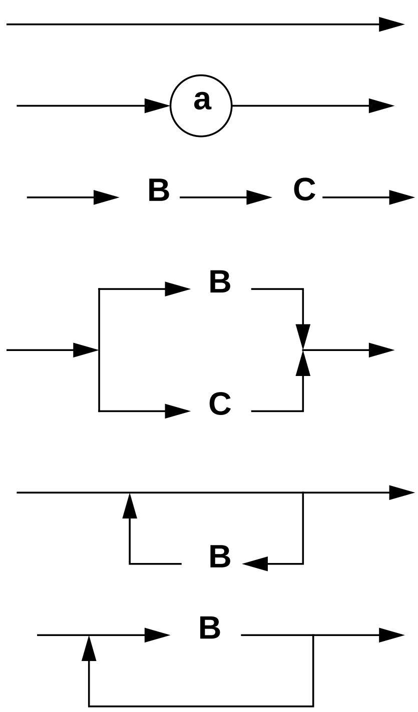
Example: identifiers



Regular expressions mapped to syntax diagrams

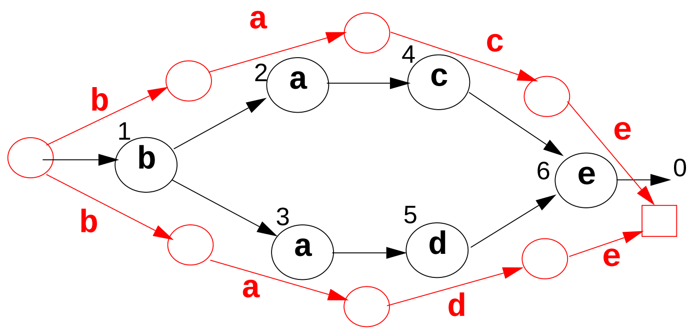
Transformation rules:

regular expression A	syntax diagram for A	
empty		empty
a		single character
B C		sequence
B | C		alternative
B*		repetition, may be empty
B^+		repetition, non-empty

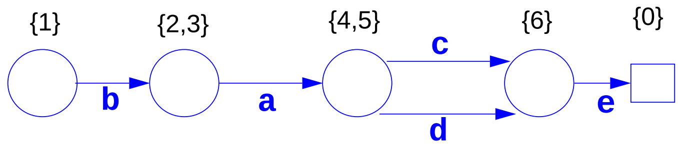


Naive transformation

1. Transform a **syntax diagram into a non-det. FSM** by naively exchanging nodes and arcs



2. Transform a non-det. FSM into a det. FSM:
   Merge equivalents sets of nodes into nodes.



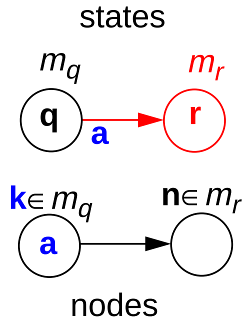
Syntax diagram	**deterministic finite state machine**
set of nodes m_q	state *q*
sets of nodes m_q and m_r connected with the same character *a*	transition *q* ---> *r* with character *a*

Construction of deterministic finite state machines

Syntax diagram	**deterministic finite state machine**
set of nodes m_q	state *q*
sets of nodes m_q and m_r connected with the same character *a*	transitions *q* ---> *r* with character *a*

Construction:

1. **enumerate nodes**; exit of the diagram gets the number *0*
2. **initial set of nodes** m_1 contains all nodes that are reachable from the begin of the diagram; m_1 represents the **initial state 1**.
3. **construct new sets of nodes (states)** and **transitions:**
   - chose state *q* with m_q, chose a character *a*
   - consider the set of nodes with character *a*, s.t. their labels *k* are in m_q.
   - consider all nodes that are directly reachable from those nodes; let m_r be the set of their labels
   - create a state *r* for m_r and a transition from *q* to *r* under *a.*



4. **repeat step 3** until no new states or transitions can be created
5. a state *q* is a **final state** iff *0* is in m_q.

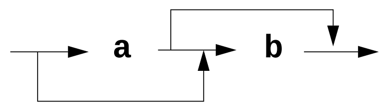
Properties of the transformation

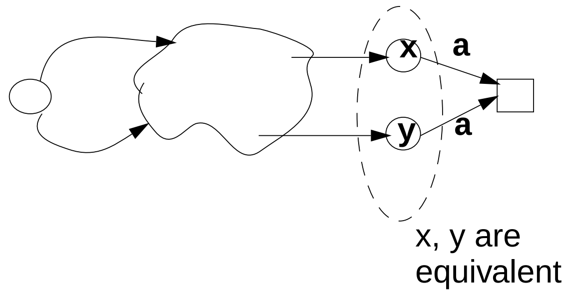
1. **Syntax diagrams** can express languages **more compact** than regular expressions can:
   A regular expression for { a, ab, b} needs more than one occurrence of a or b -
   a syntax diagram doesn't.

2. The FSM resulting from a transformation of PLaC 2.7a may have m**ore states than necessary**.

3. There are transformations that **minimize the number of states** of any FSM.

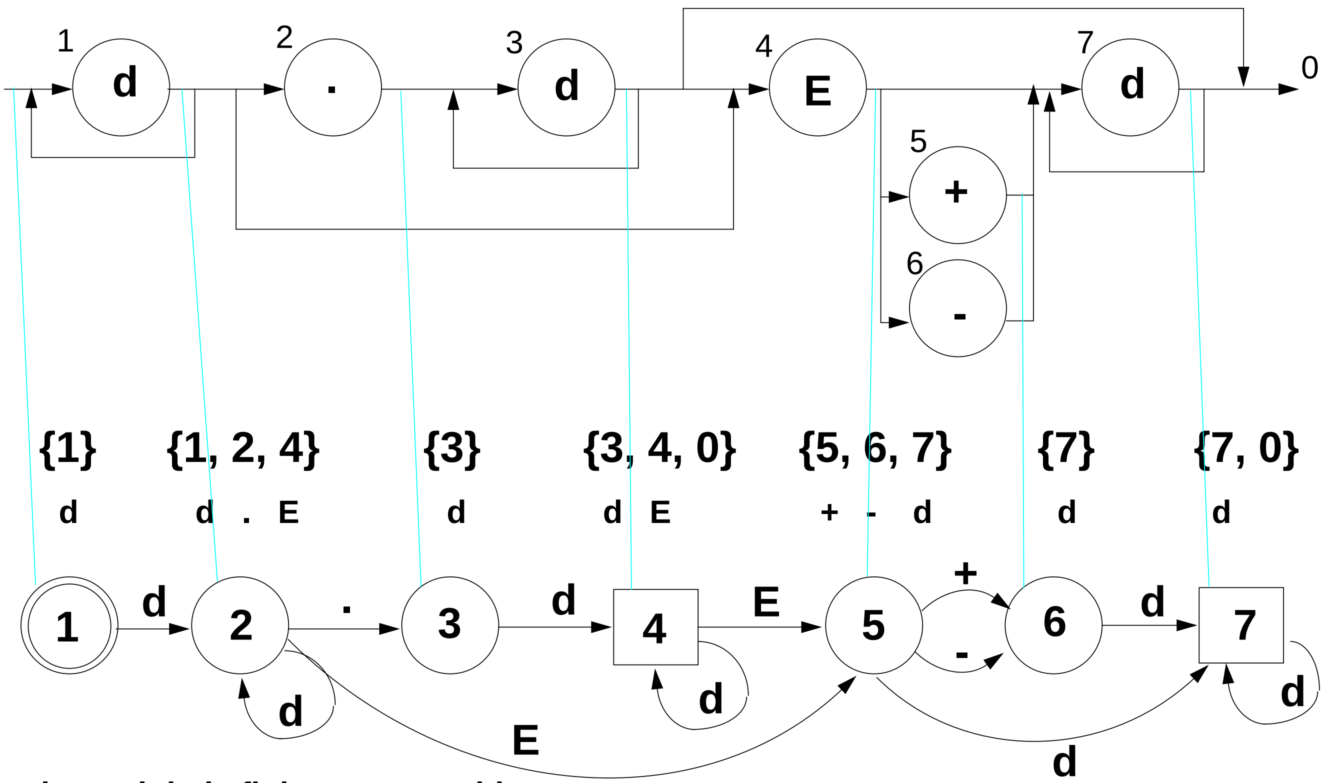
(a (| b)) | b





Example: Floating point numbers in Pascal

Syntax diagram



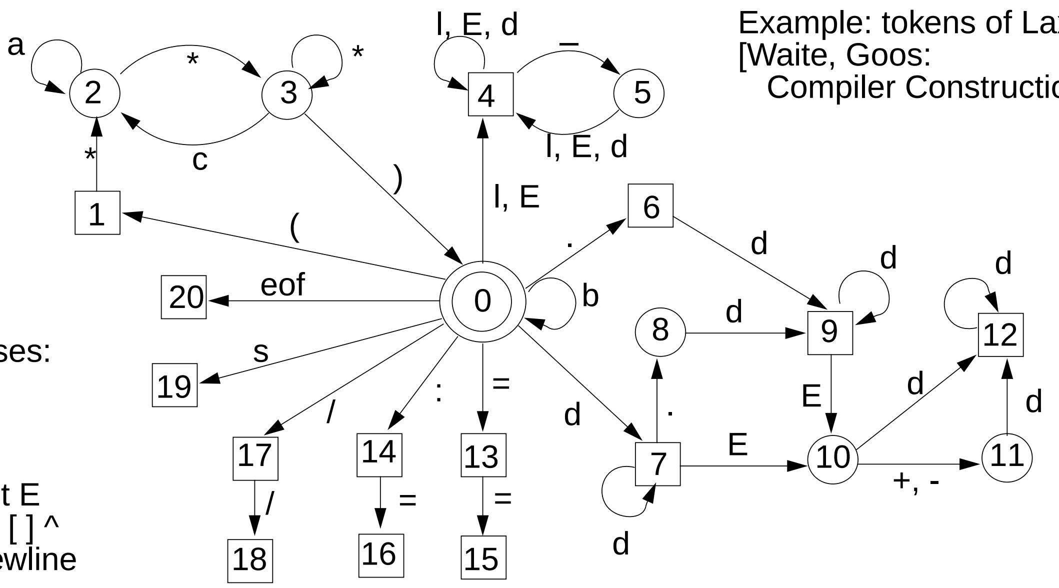
deterministic finite state machine

Composition of token automata

Construct one finite state machine for each token. Compose them forming a single FSM:

- **Identify the initial states of the single automata** and identical structures evolving from there (transitions with the same character and states).
- **Keep the final states of single automata distinct,** they classify the tokens.
- **Add automata for comments and irrelevant characters** (white space)

Example: tokens of Lax [Waite, Goos: Compiler Construction]



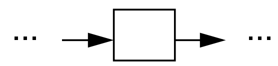
character classes:

a all but *
c all but * or)
d digits
l all letters but E
s + - * < > ; ,) [ ] ^
b blank tab newline

Rule of the longest match

An automaton may contain **transitions from final states:**

When does the automaton stop?



Rule of the longest match:

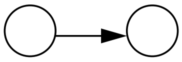
- **The automaton continues as long as there is a transition with the next character.**
- **After having stopped it sets back to the most recently passed final state.**
- **If no final state has been passed an error message is issued.**

Consequence: Some kinds of tokens have to be separated explicitly.

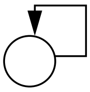
Check the concrete grammar for tokens that may occur adjacent!

Scanner: Aspects of implementation

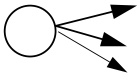
- **Runtime is proportional to the number of characters in the program**
- **Operations per character must be fast** - otherwise the Scanner dominates compilation time
- **Table driven** automata are too **slow:**
  Loop interprets table, 2-dimensional array access, branches
- **Directly programmed** automata is **faster;** transform **transitions into control flow:**



sequence



repeat loop



branch, switch

- **Fast loops** for sequences of irrelevant **blanks**.
- Implementation of **character classes**:
  bit pattern or indexing - avoid slow operations with sets of characters.
- **Do not copy characters** from input buffer - maintain a pointer into the buffer, instead.

Characteristics of Input Data

Table 7
Characteristics of the Input Data

	P4		SYNPUT	
	Occurrences	Characters	Occurrences	Characters
Single spaces	11404	11404	2766	2766
Identifiers	8411	41560	5799	22744
Keywords	4183	15080	2034	7674
>3 spaces	3850	60694	1837	19880
;	2708	2708	1880	1880
:=	1379	2758	966	1932
Integers	1354	2202	527	573
(	1245	1245	751	751
)	1245	1245	751	751
,	1032	1032	842	842
comments	659	13765	675	35066
[	654	654	218	218
]	654	654	218	218
:	635	635	483	483
.	546	546	400	400
Strings	493	2560	303	3017
Space pairs	470	940	39	78
=	438	438	206	206
^	353	353	461	461
<>	213	426	96	192
+	203	203	183	183
-	82	82	61	61
Space triples	56	168	842	2526
..	37	74	21	42
<=	26	52	5	10
>	18	18	27	27
<	14	14	25	25
*	10	10	12	12
>=	5	10	7	14
Reals	0	0	3	14
/	0	0	1	1

significant numbers of characters

W. M. Waite:
The Cost of Lexical Analysis.
Software- Practice and Experience,
16(5):473-488, May 1986.

Identifier module and literal modules

- **Uniform interface for all scanner support modules:**
  Input parameters: pointer to token text and its length;
  Output parameters: syntax code, attribute

- **Identifier module encodes identifier occurrences bijective (1:1), and recognizes keywords**
  Implementation: hash vector, extensible table, collision lists

- **Literal modules for floating point numbers, integral numbers, strings**

  Variants for representation in memory:
  token text; value converted into compiler data; value converted into target data

  Caution:
  Avoid overflow on conversion!
  Cross compiler: compiler representation may differ from target representation

- **Character string memory:**
  stores strings without limits on their lengths,
  used by the identifier module and the literal modules

Scanner generators

generate the central function of lexical analysis

GLA	University of Colorado, Boulder; component of the Eli system
Lex	Unix standard tool
Flex	Successor of Lex
Rex	GMD Karlsruhe

Token specification: regular expressions

GLA	library of precoined specifications; recognizers for some tokens may be programmed
Lex, Flex, Rex	transitions may be made conditional

Interface:

GLA	as described in this chapter; cooperates with other Eli components
Lex, Flex, Rex	actions may be associated with tokens (statement sequences) interface to parser generator Yacc

Implementation:

GLA	directly programmed automaton in C
Lex, Flex, Rex	table-driven automaton in C
Rex	table-driven automaton in C or in Modula-2
Flex, Rex	faster, smaller implementations than generated by Lex