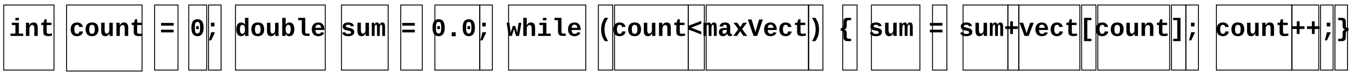
2. Symbol specifications and lexical analysis

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` `++` `;` `}`

Lecture Programming Languages and Compilers WS 2011/12 / Slide 201

Objectives:

Introduction of the task of lexical analysis

In the lecture:

Explain the example

Lexical Analysis

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

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

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

Lecture Programming Languages and Compilers WS 2011/12 / Slide 202

Objectives:

Understand lexical analysis subtasks

In the lecture:

Explain

- subtasks and their interfaces using example of PLaC-201,
- different forms of comments,
- sparation of tokens in FORTRAN,

Suggested reading:

Kastens / Übersetzerbau, Section 3, 3.3.1

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.

Lecture Programming Languages and Compilers WS 2011/12 / Slide 203

Objectives:

Recognize difficult specifications

In the lecture:

Explain

- isolated recognition and encoding of tokens,
- feedback of information,
- unusual notation of keywords,
- separation of tokens in FORTRAN,

Suggested reading:

Kastens / Übersetzerbau, Section 3, 3.3.1

Questions:

- Give examples of context dependent information about tokens, which the lexical analysis can not know.
- Some decisions on the notation of tokens and the syntax of a language may complicate lexical analysis. Give examples.
- Explain the typedef problem in C.

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

Lecture Programming Languages and Compilers WS 2011/12 / Slide 204

Objectives:

Understand token representation

In the lecture:

Explain the roles of the 3 components using the examples

Suggested reading:

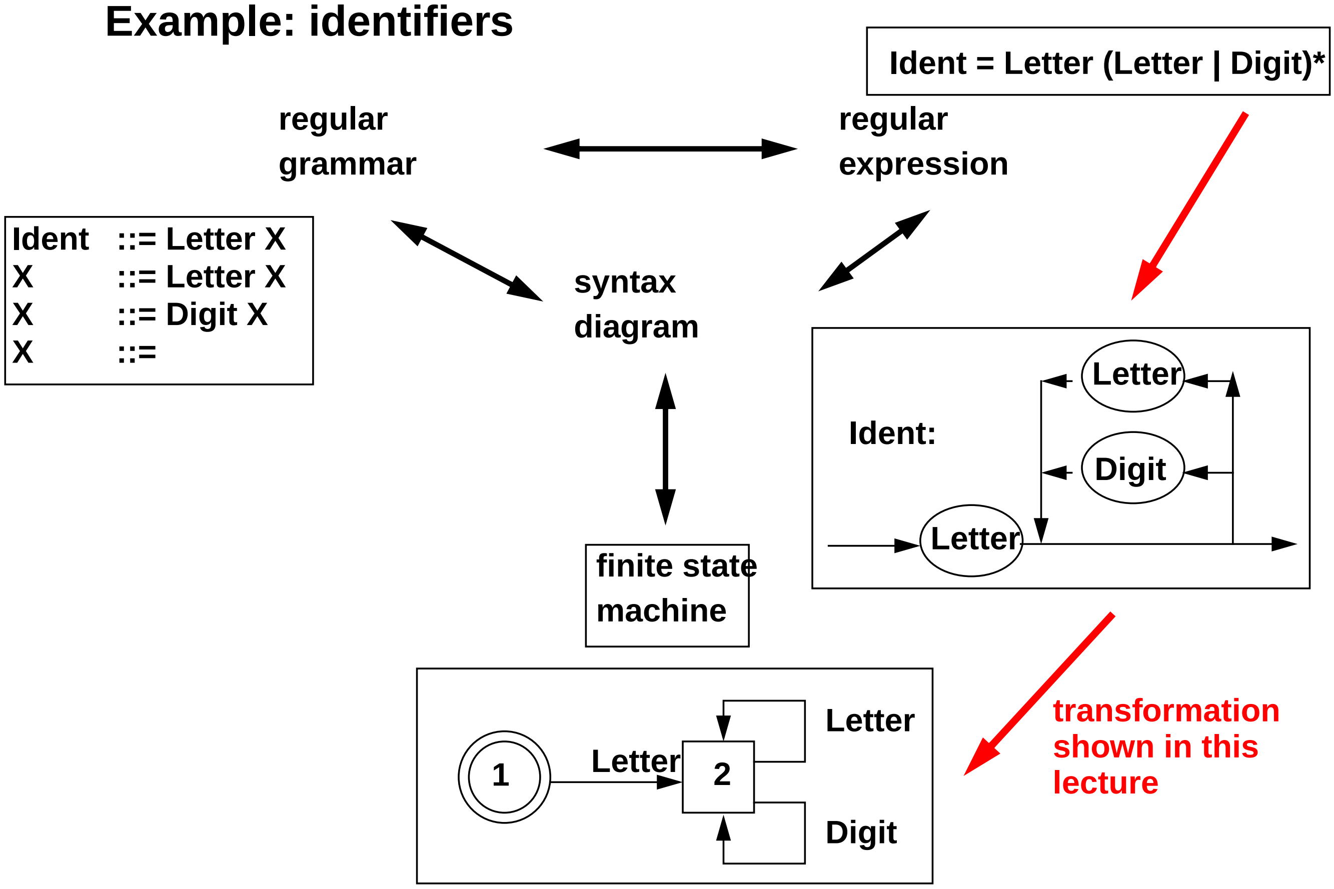
Kastens / Übersetzerbau, Section 3, 3.3.1

Questions:

- What are the requirements for the encoding of identifiers?
- How does the identifier module meet them?
- Can the values of integer literals be represented as attribute values, or do we have to store them in a data module? Explain! Consider also cross compilers!

Specification of token notations

Example: identifiers



Ident = Letter (Letter | Digit)*

regular grammar

regular expression

syntax diagram

finite state machine

Ident ::= Letter X
X ::= Letter X
X ::= Digit X
X ::=

transformation shown in this lecture

Lecture Programming Languages and Compilers WS 2011/12 / Slide 205

Objectives:

Equivalent forms of specification

In the lecture:

- Repeat calculi of the lectures "Modellierung" and "Berechenbarkeit und formale Sprachen".
- Our strategy: Specify regular expressions, transform into syntax diagrams, and from there into finite state machines

Suggested reading:

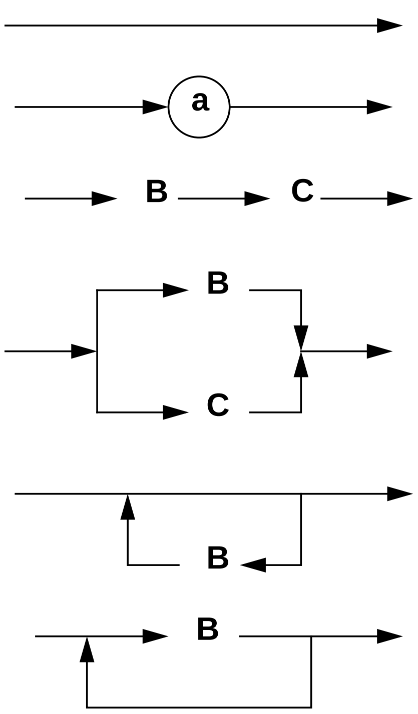
Kastens / Übersetzerbau, Section 3.1

Questions:

- Give examples for Unix tools which use regular expressions to describe their input.

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

Lecture Programming Languages and Compilers WS 2011/12 / Slide 206

Objectives:

Construct by recursive substitution

In the lecture:

- Explain the construction for floating point numbers of Pascal.

Suggested reading:

Kastens / Übersetzerbau, Section 3.1

Assignments:

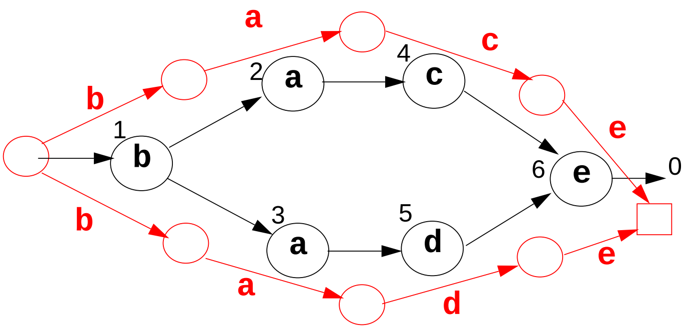
- Apply the technique Exercise 6

Questions:

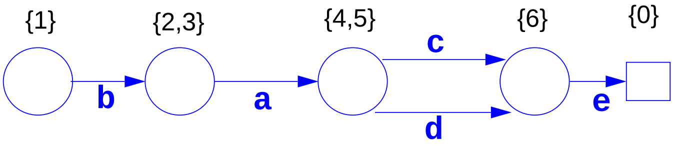
- If one transforms syntax diagrams into regular expressions, certain structures of the diagram require duplication of subexpressions. Give examples.
- Explain the analogy to control flows of programs with labels, jumps and loops.

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

Lecture Programming Languages and Compilers WS 2011/12 / Slide 207

Objectives:

Understand the transformation method

In the lecture:

- Explain the naive idea with a small artificial example

Suggested reading:

Kastens / Übersetzerbau, Section 3.2

Assignments:

- Apply the method Exercise 6

Questions:

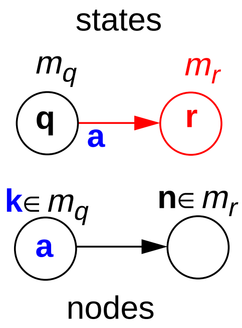
- Why does the naive method may yield non-deterministic automata?

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 $q \dashrightarrow r$ 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.



Lecture Programming Languages and Compilers WS 2011/12 / Slide 207a

Objectives:

Understand the transformation method

In the lecture:

- Explain the method using floating point numbers of Pascal (PLaC-2.8)
- Recall the method presented in the course "Modellierung".

Suggested reading:

Kastens / Übersetzerbau, Section 3.2

Assignments:

- Apply the method Exercise 6

Questions:

- Why does the method yield deterministic automata?

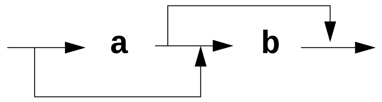
Properties of the transformation

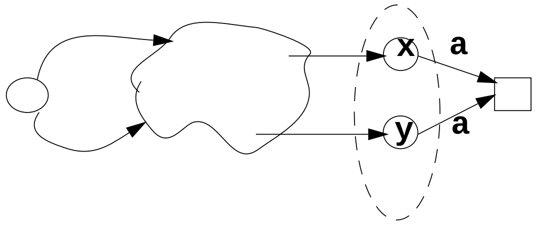
1. **Syntax diagrams** can express languages **more compact** than regular expressions can:
   A regular expression for { a, ab, b} needs more than one occurrence of a or b -
   a syntax diagram doesn't.

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`





x, y are equivalent

Lecture Programming Languages and Compilers WS 2011/12 / Slide 207b

Objectives:

Understand the transformation method

In the lecture:

- Explain the properties.
- Recall the algorithm.

Suggested reading:

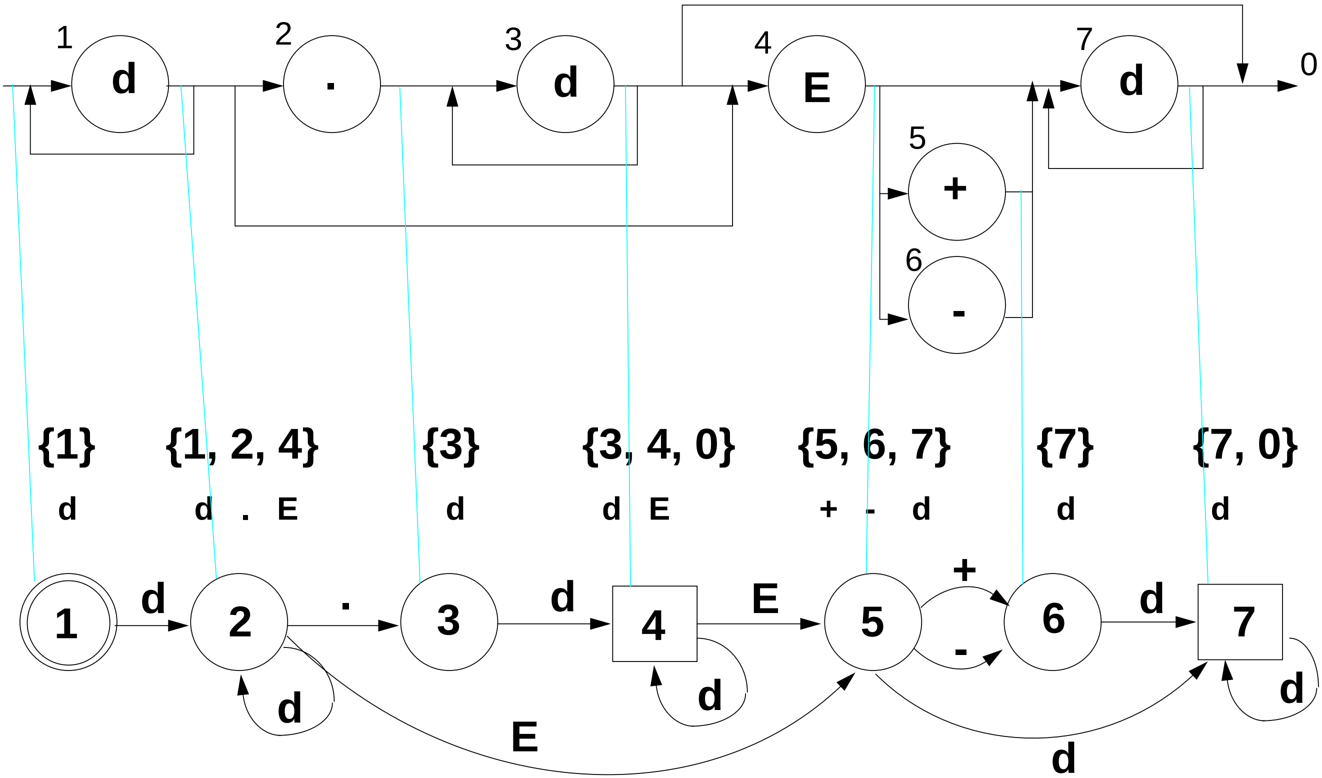
Kastens / Übersetzerbau, Section 3.2

Assignments:

- Apply the method Exercise 6

Example: Floating point numbers in Pascal

Syntax diagram



deterministic finite state machine

Lecture Programming Languages and Compilers WS 2011/12 / Slide 208

Objectives:

Understand the construction method

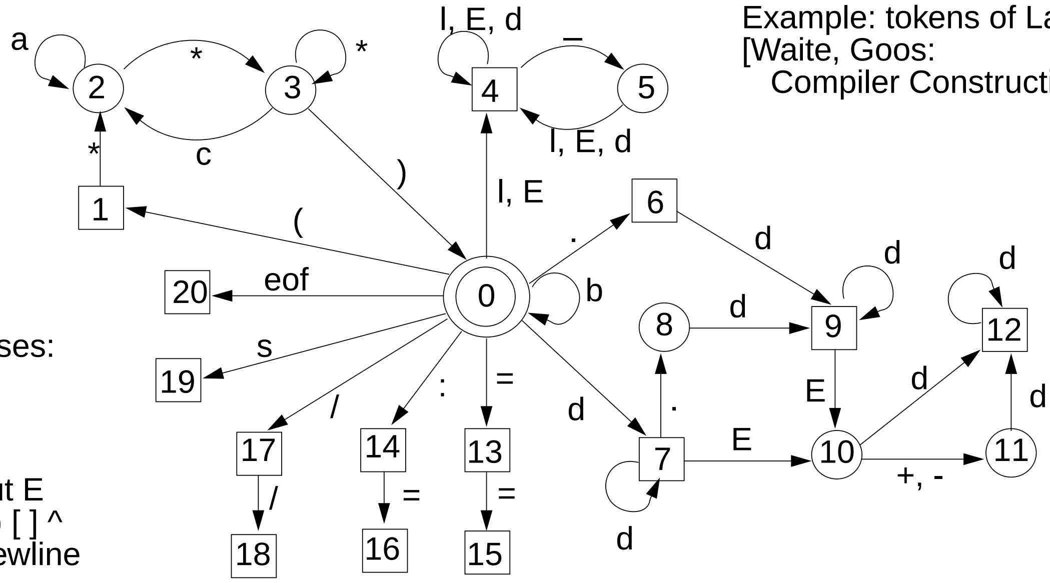
In the lecture:

The construction process of the previous slide is explained using this example.

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

Lecture Programming Languages and Compilers WS 2011/12 / Slide 209

Objectives:

Construct a multi-token automaton

In the lecture:

Use the example to

- discuss the composition steps,
- introduce the abbreviation by character classes,
- to see a non-trivial complete automaton.

Suggested reading:

Kastens / Übersetzerbau, Section 3.2

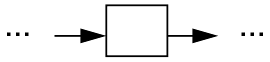
Questions:

Describe the notation of Lax tokens and comments in English.

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!

Lecture Programming Languages and Compilers WS 2011/12 / Slide 210

Objectives:

Understand the consequences of the rule

In the lecture:

- Discuss examples for the rule of the longest match.
- Discuss different cases of token separation.

Suggested reading:

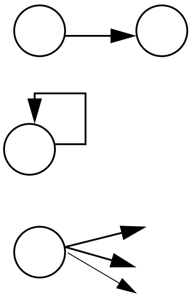
Kastens / Übersetzerbau, Section 3.2

Questions:

- Point out applications of the rule in the Lax automaton, which arose from the composition of sub-automata.
- Which tokens have to be separated by white space?

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.

Lecture Programming Languages and Compilers WS 2011/12 / Slide 211

Objectives:

Runtime efficiency is important

In the lecture:

- Advantages of directly programmed automata. Compare to table driven.

Suggested reading:

Kastens / Übersetzerbau, Section 3.3

Assignments:

- Generate directly programmed automata Exercise 7

Questions:

- Are there advantages for table-driven automata? Check your arguments carefully!

Characteristics of Input Data

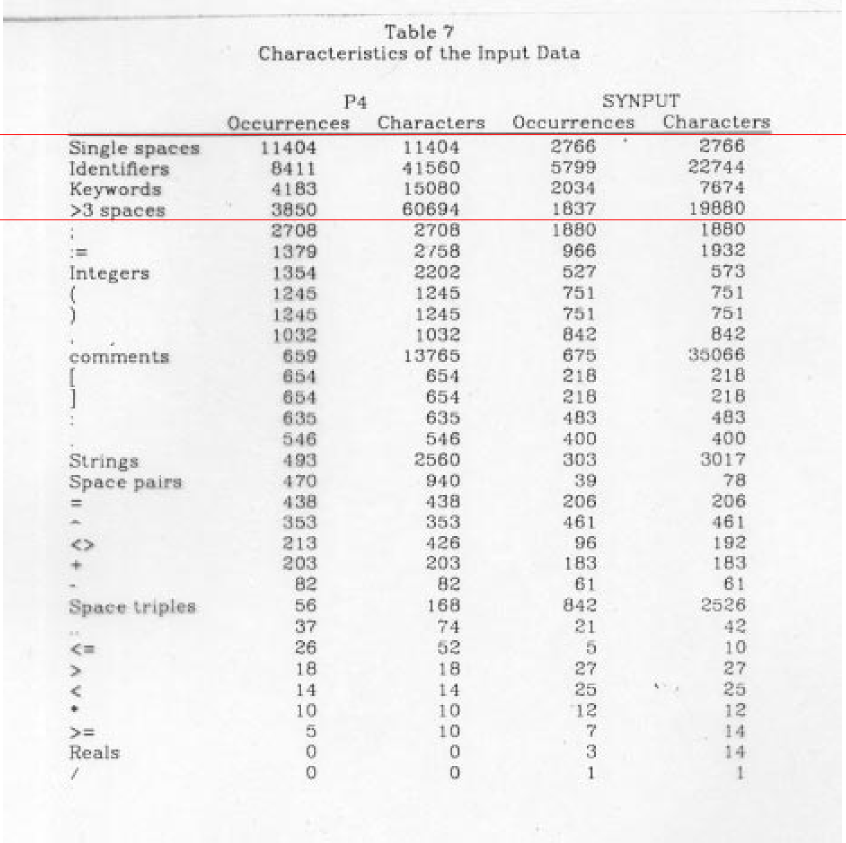


Table 7
Characteristics of the Input Data

	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.

Lecture Programming Languages and Compilers WS 2011/12 / Slide 211b

Objectives:

Profile how characters contribute to tokens

In the lecture:

- Measurements on occurrences of symbols: Single spaces, identifiers, keywords, squences of spaces are most frequent. Comments contribute most characters.

Suggested reading:

Kastens / Übersetzerbau, Section 3.3

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

Lecture Programming Languages and Compilers WS 2011/12 / Slide 212

Objectives:

Safe and efficient standard implementations are available

In the lecture:

- Give reasons for the implementation techniques.
- Show different representations of floating point numbers.
- Escape characters in strings need conversion.

Suggested reading:

Kastens / Übersetzerbau, Section 3.3

Questions:

- Give examples why the analysis phase needs to know values of integral literals.
- Give examples for representation of literals and their conversion.

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

---

Lecture Programming Languages and Compilers WS 2011/12 / Slide 213

Objectives:

Know about the most common generators

In the lecture:

Explain specific properties mentioned here.

Suggested reading:

Kastens / Übersetzerbau, Section 3.4

Assignments:

Use GLA and Lex Exercise 7