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 Compiler I WS 2001/2002 / Slide 26

Objectives:

Understand lexical analysis subtasks

In the lecture:

Explain

- subtasks and their interfaces using slide CI-16,
- unusual notation of keywords,
- different forms of comments,
- sparation 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 Compiler I WS 2001/2002 / Slide 27

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

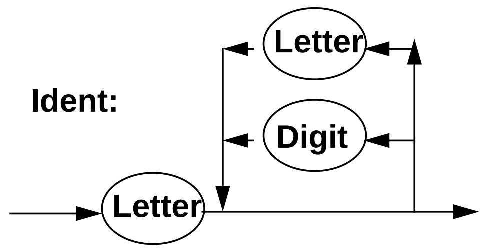
Ident = Letter (Letter | Digit)*

regular grammar ⟷ **regular expression**

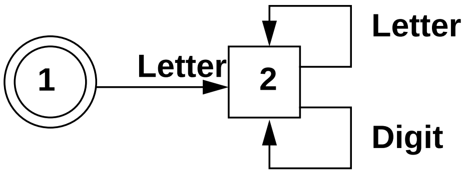
syntax diagram ⟷ **finite state machine**

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

Ident:



acceptor:



Lecture Compiler I WS 2001/2002 / Slide 28

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

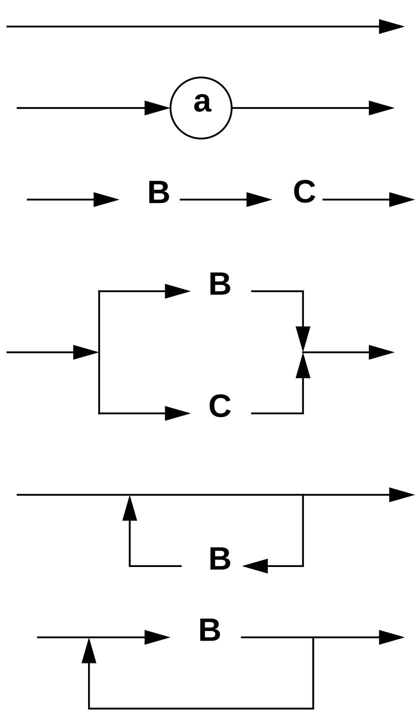
Questions:

- 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 Compiler I WS 2001/2002 / Slide 29

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 requires duplication of subexpressions. Give examples.
- Explain the analogy to control flows of programs with labels, jumps and loops.

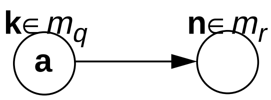
Construction of deterministic finite state machines

Syntax diagram	**deterministic finite state machine**
nodes, arcs	transitions, states
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*

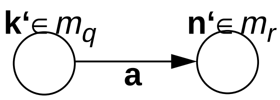
Construction:

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 — **initial state** *1*
3. **construct new sets of nodes (states)** and **transitions:** For a character *a* and a set m_q containing node *k* create set m_r with all nodes *n*, according to the following schema:

for



create



4. **repeat step 3** until no new sets of nodes can be created
5. a state *q* is a **final state** iff *0* is in m_q.

Lecture Compiler I WS 2001/2002 / Slide 30

Objectives:

Understand the method

In the lecture:

- Explain the idea with a small artificial example
- Explain the method using floating point numbers of Pascal (Slide CI-31)

Suggested reading:

Kastens / Übersetzerbau, Section 3.2

Assignments:

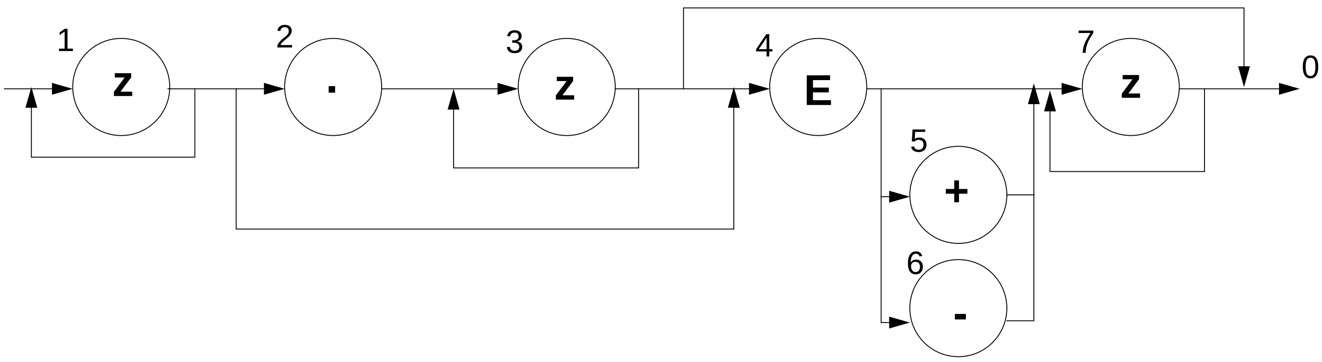
- Apply the method Exercise 6

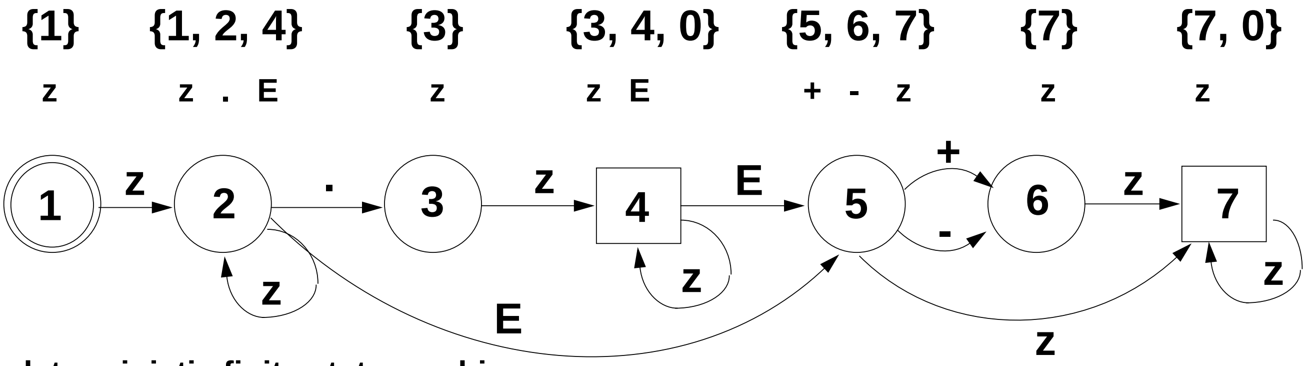
Questions:

- Why does the method yield deterministic automata?
- Describe roughly a simple technique which may yield non-deterministic automata.

Example: Floating point numbers in Pascal

Syntax diagram





deterministic finite state machine

Lecture Compiler I WS 2001/2002 / Slide 31

Objectives:

Understand the construction method

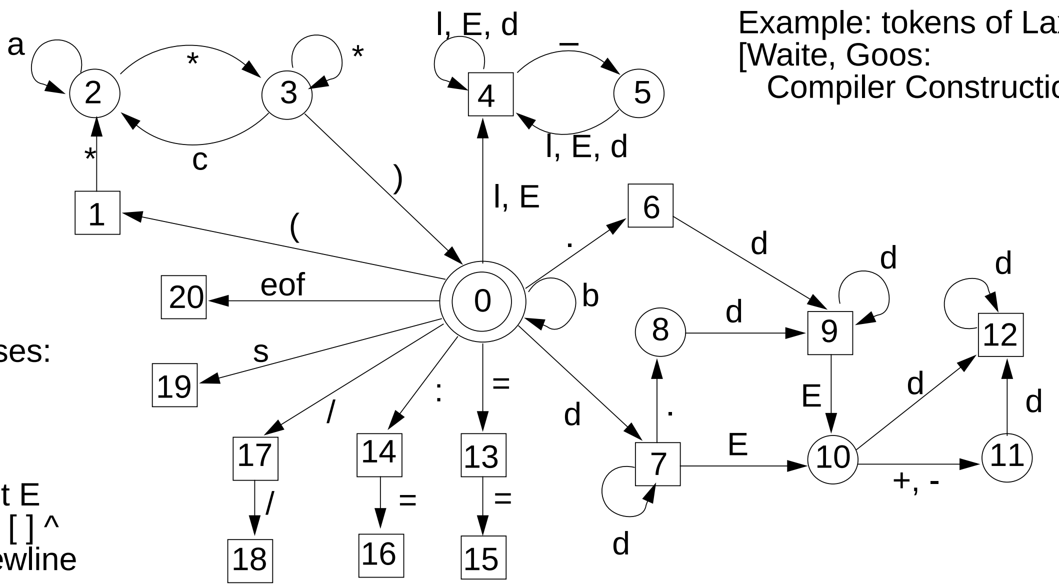
In the lecture:

The construction process of slide CI-30 is explained using this example.

Composition of token automata

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

- **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 Compiler I WS 2001/2002 / Slide 32

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

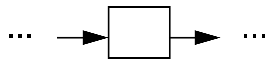
Questions:

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 Compiler I WS 2001/2002 / Slide 33

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:

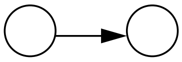
Kastens / Übersetzerbau, Section 3.2

Questions:

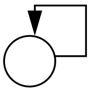
- Point out applications of the rule in the Lax automaton, which arose from the composition of sub-automata.
- Which tokens have to be separated by white space?

Scanner: Aspects of implementation

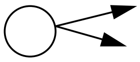
- **Runtime is proportional to the number of characters in the program**
- **Operations per character must be fast** - otherwise the Scanner dominates compilation time
- **Table driven** automata are too **slow:**
  Loop interprets table, 2-dimensional array access, branches
- **Directly programmed**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



sequence



repeat loop



branch

- **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 Compiler I WS 2001/2002 / Slide 34

Objectives:

Runtime efficiency is important

In the lecture:

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

Assignments:

- Generate directly programmed automata Exercise 7

Questions:

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

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 Compiler I WS 2001/2002 / Slide 35

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 Compiler I WS 2001/2002 / Slide 36

Objectives:

Know about some common generators

In the lecture:

Explain specific properties mentioned here.

Suggested reading:

Kastens / Übersetzerbau, Section 3.4

Assignments:

Use GLA and Lex Exercise 7