

Lecture 2: Software and computer language theory

- Review: Assembler
- Operating system
- (Higher) languages theory
- Programming paradigms
- Structures of a program
- Compilation and interpretation



Computer architecture



Which architecture is similar to those from Charles Babage and is realized in any modern computer? Give the units and explain them (each in one sentence)?

Review: Binary

```
int iSetTTYRaw (int iFiledescriptor)
{
    struct termio temp_mode;
    /* read settings */
    if (ioctl(iFiledescriptor, TCGETA, (char*) &temp_mode) < 0)
        return (-1);
    /* save for restoring later */
    tty_mode = temp_mode;
    /* turn off all input control */
    temp_mode.c_iflag = 0;
    /* disable output post-processing */
    temp_mode.c_oflag &= ~OPOST;
    /* disable signal generation, canonical input, echo, upper/lower output */
    temp_mode.c_lflag &= ~(ISIG | ICANON | XCASE | ECHO);
    /* clear char size, disable parity */
    temp_mode.c_cflag &= ~(CSIZE | PARENB);
    /* 8-bit chars */
    temp_mode.c_cflag |= CS8;
    /* min #char to satisfy read */
    temp_mode.c_cc[VMIN] = 1;
    /* 10'ths of seconds between chars */
    temp_mode.c_cc[VTIME] = 1;
    /* write settings */
    if (ioctl(iFiledescriptor, TCSETA, (char*) &temp_mode) < 0)
        return (-1);
    return (0);
}
```

Example code:
Terminal
manipulation

Review: Binary

```
int iSetTTYRaw (int iFiledescriptor)
{
    struct termio temp_mode;
    /* read settings */
    if (ioctl(iFiledescriptor, TCGETA, (char*) &temp_mode) < 0)
        return (-1);
    /* save for restoring later */
    tty_mode = temp_mode;
    /* turn off all input control */
    temp_mode.c_iflag = 0;
    /* disable output post-processing */
    temp_mode.c_oflag &= ~OPOST;
    /* disable signal generation, canonical input, echo, upper/lower output */
    temp_mode.c_lflag &= ~(ISIG | ICANON | XCASE | ECHO);
    /* clear char size, disable parity */
    temp_mode.c_cflag &= ~(CSIZE | PARENB);
    /* 8-bit chars */
    temp_mode.c_cflag |= CS8;
    /* min #char to satisfy read */
    temp_mode.c_cc[VMIN] = 1;
    /* 10'ths of seconds between chars */
    temp_mode.c_cc[VTIME] = 1;
    /* write settings */
    if (ioctl(iFiledescriptor, TCSETA, (char*) &temp_mode) < 0)
        return (-1);
    return (0);
}
```

Example code:
Terminal
manipulation

Set local mode of TTY
(from „Teletype Writer“)

& = bit by bit AND

| = bit by bit OR

~ = bit by bit NOT

Review: Binary

```
int iSetTTYRaw (int iFiledescriptor)
{
...
    /* disable signal generation, canonical input,
    echo, upper/lower output */
    temp_mode.c_lflag &= ~(ISIG | ICANON | XCASE | ECHO);
...
}
```

Example code:
Terminal
manipulation

```
=> temp_mode.c_lflag = temp_mode.c_lflag & (~(ISIG | ICANON | XCASE | ECHO));
```

ISIG	= 1	=	0000 0001	
ICANON	= 2	=	0000 0010	
XCASE	= 3	=	0000 0100	
ECHO	= 4	=	0000 1000	
(OR)	=		0000 1111	(bit by bit)
~ (NOT)	=		1111 0000	(bit by bit)

```
e.g. temp_mode.c_lflag = 1011 1010
                        1111 0000
                        & (AND)
                        1011 0000
```

Switch off flags



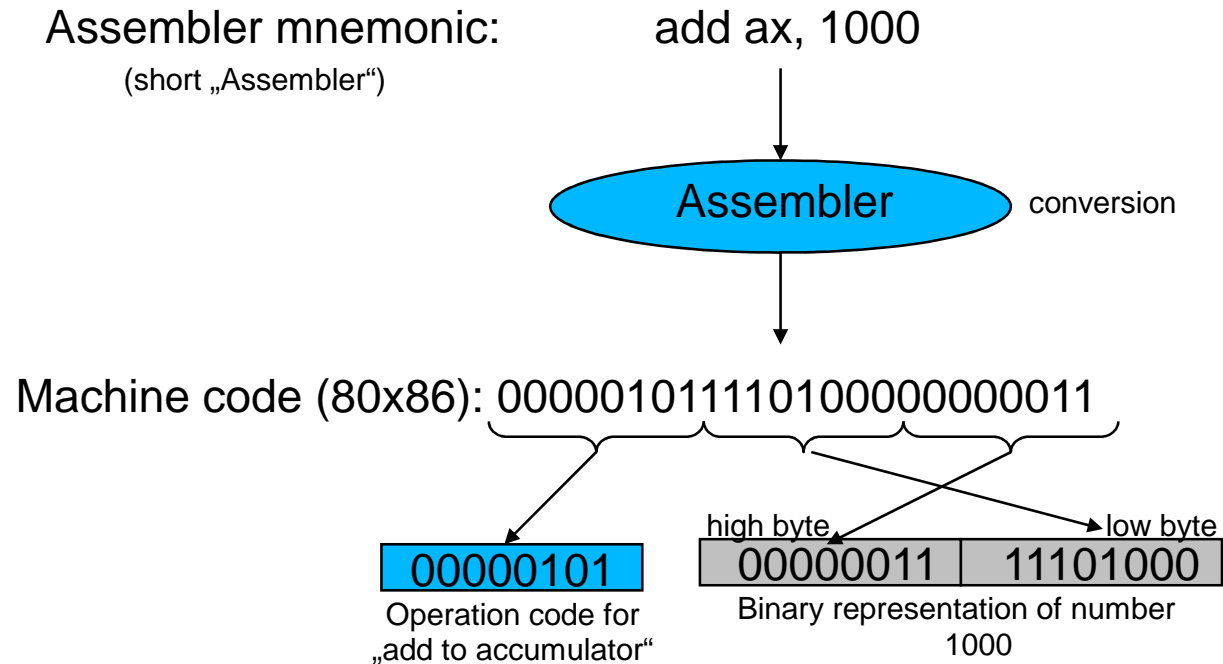
Review: Binary



How can you switch on the flag for ECHO?

Review: Assembler

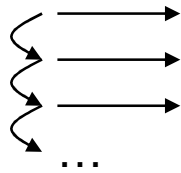
The machine instructions



Review: Assembler

The machine instruction sequence

Instruction counter



```
segment code

start:
mov ax, data
mov ds, ax

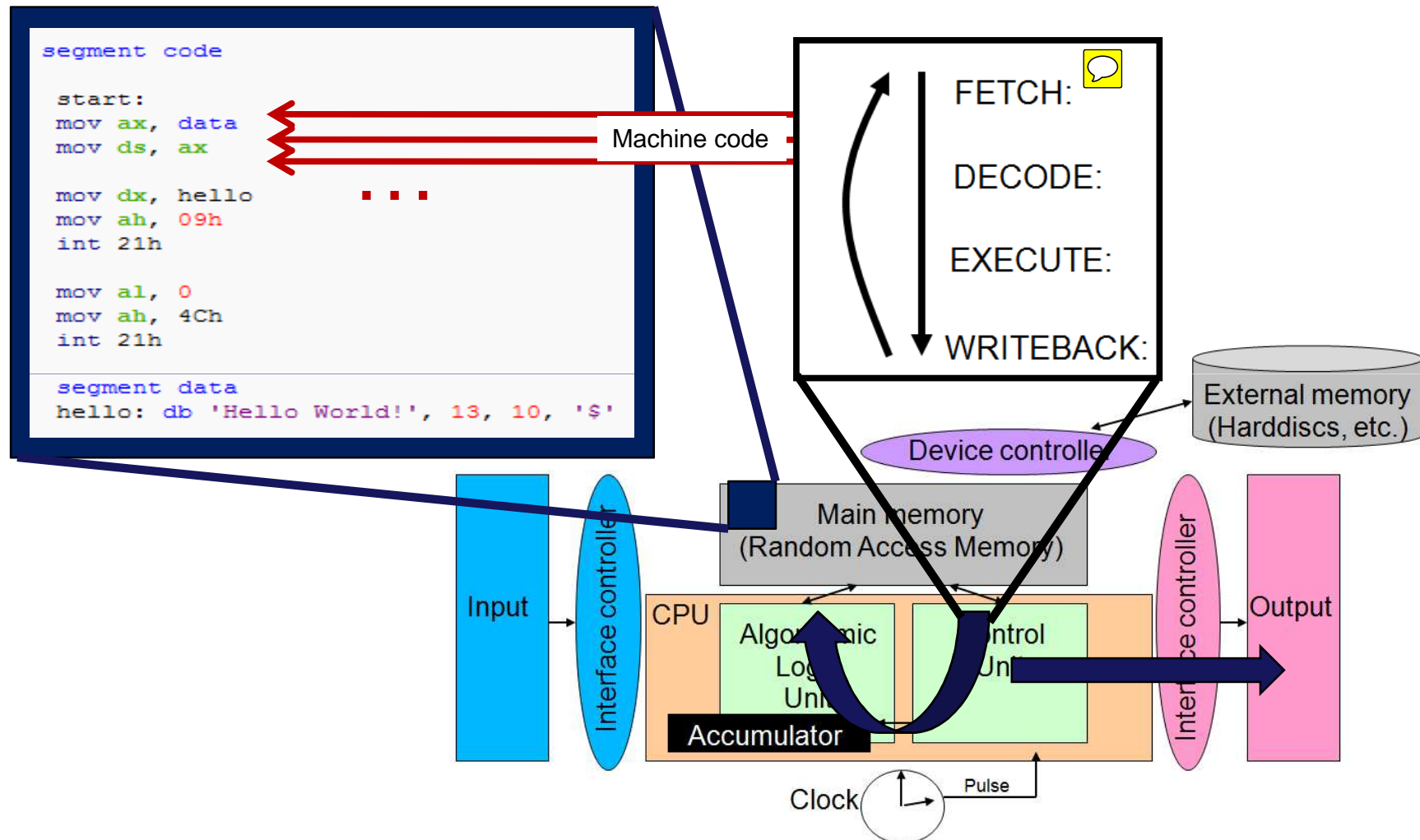
...

mov dx, hello
mov ah, 09h
int 21h

mov al, 0
mov ah, 4Ch
int 21h

segment data
hello: db 'Hello World!', 13, 10, '$'
```


Review: Assembler



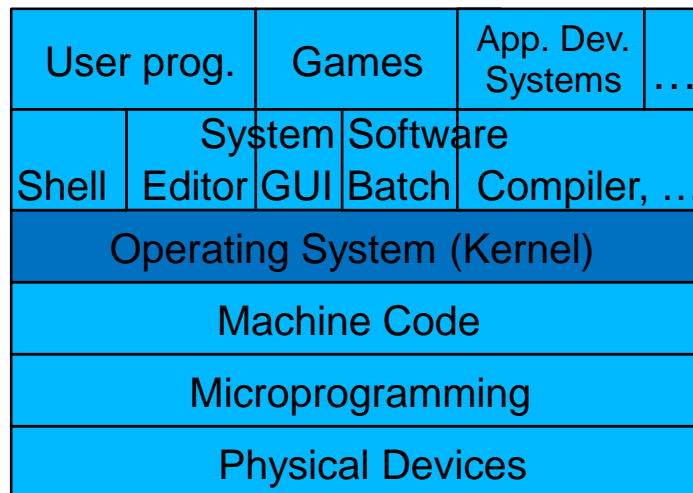
➡ Each user had to write his own software solutions for a specific problem including all system hardware specific requirements. In most cases there is no possibility to share resources automatically or to administrate the usage.

Lecture 2: Software and computer language theory

- ✓ Review: Assembler
 - Operating system
 - (Higher) languages theory
 - Programming paradigms
 - Structures of a program
 - Compilation and interpretation

Operating system

Need of a basic system layer to abstract the hardware and to handle the administration of resources: the operating system

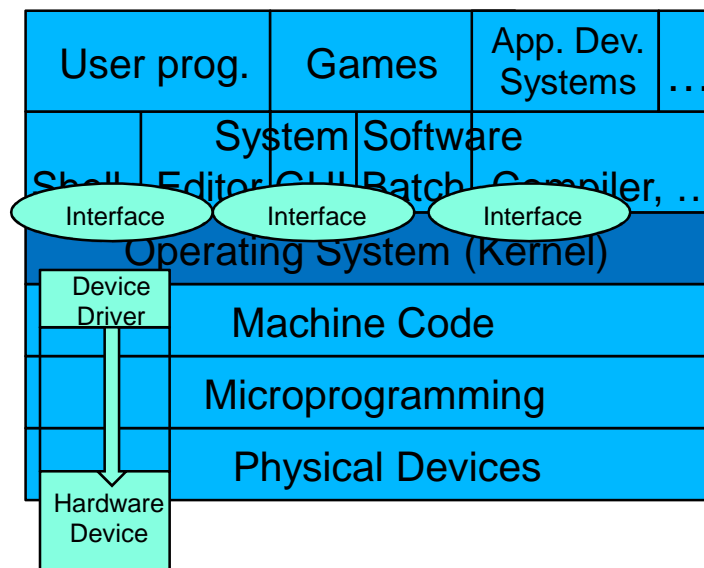


- . High-level abstraction of hardware with simple interface (system calls)
- . Application independence
- . Ressource sharing, critical section management
- . Memory management
- . Process management (multi-user, multi-tasking)
- . Security, Kernel-/Supervisor mode
- . Disk and file system
- . Device driver
- . Networking
- . Spooling (Simultaneous Peripheral Operation On Line)
- . Interrupt handling
-

e.g. MS DOS 1.0 (1981): single-user and single-tasking operating system consists of 4000 lines of Assembler code (modern operating systems are also programmed in C)

Operating system

Abstraction of the hardware

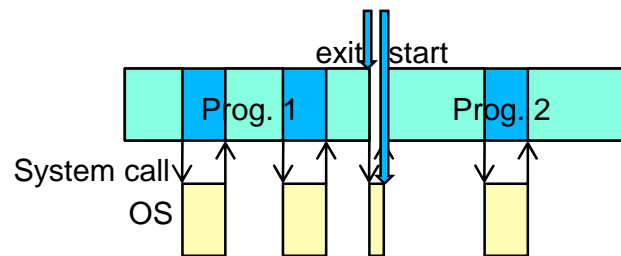


- The usage of a specific device is hidden in a driver with an easier interface (e.g. system calls to handle in-/output from/to serial interface without knowledge about addresses or handling of the RS232 UART hardware)
- Interface functions to manipulate system behaviour and to manage multiple access onto single hardware
- Interface functions to start, stop or administrate programs
- Basic input/output methods
- Interface methods for communication (e.g. Ethernet-TCP/IP-sockets)
- ...

Operating system

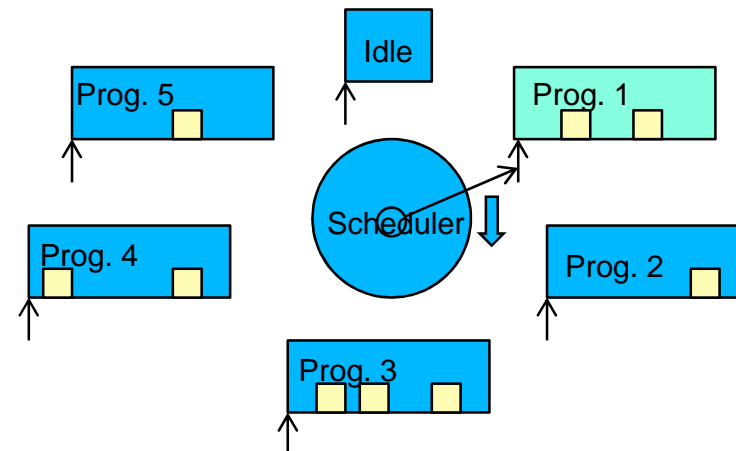
Administration of resources

Single program (on resource CPU)



- Only one program is in memory and processing
- Program calls system routines to contact devices
- A startup initializes the program into processing mode
- A shutdown removes program from processing mode

Multi program (on resource CPU)



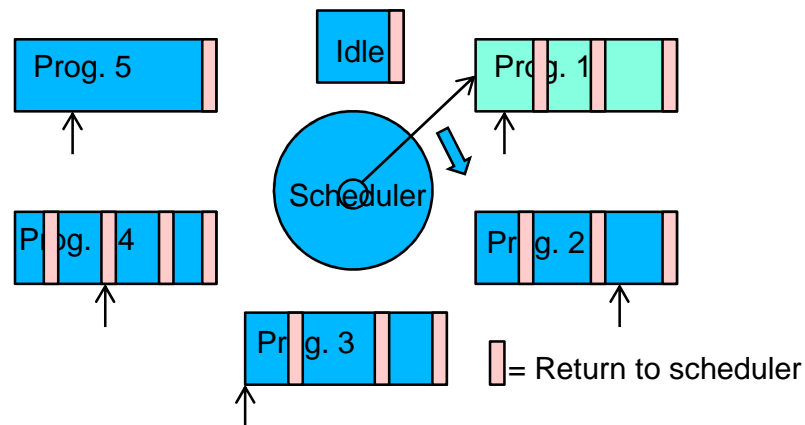
- Several programs are parallel in memory
- Alternating processing
- Programs are processes or tasks
- A scheduler does process management
- Programs activate system calls which are **non-reentrant** (critical sections)
- Critical sections are controlled with semaphore etc. (semaphore are variables which test&set within one cycle)
- An idle task is used when no program is valid
- Programs are in a batch queue

Operating system

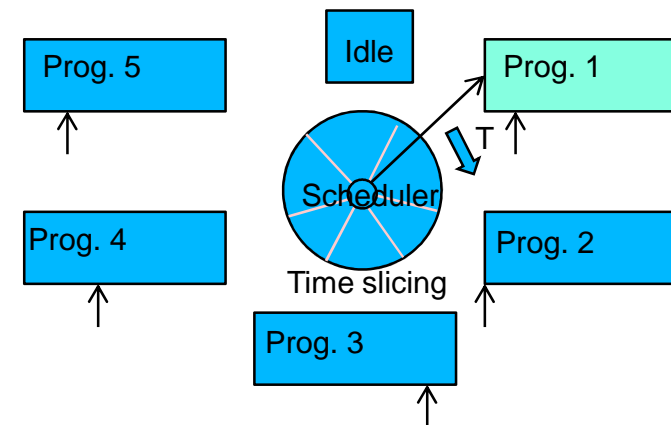
Administration of resources

Cooperative and preemptive multitasking

(decision criterion: no starvation, no deadlock, ...)



- Each program decides for its own to return
- The scheduler decides who is next
- Each program acts cooperative
- It can happen that some processes won't get resource (starvation)
- Several competing processes can block each other (deadlock)

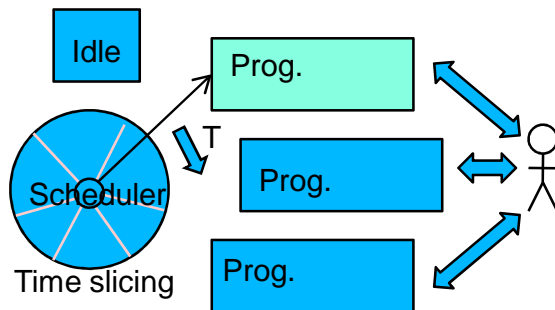


- The scheduler decides when and who is next
- It uses a timeslice T
- The resource is taken from process after T
- It prevents starvation
- In combination with timeout it can also prevent deadlocks

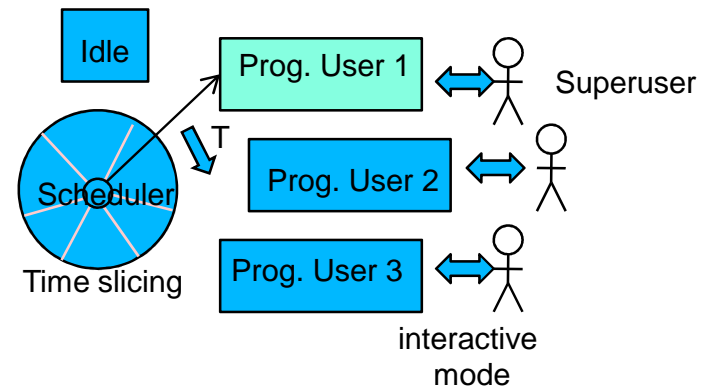
Operating system

Administration of resources

Single user and multi user mode (with right and security management, on resource computer)



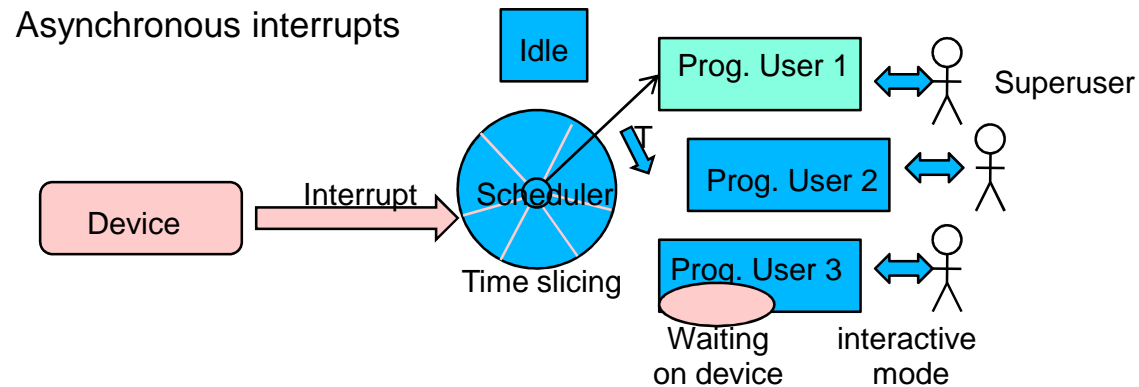
- There is only one user
- Each order has same security level
- User can do everything
- Interactive work



- There are several users
- Each user/order has specific rights
- Users must do authentication and autorisation
- There are users with specific high level rights
- There is an interactive and batch mode

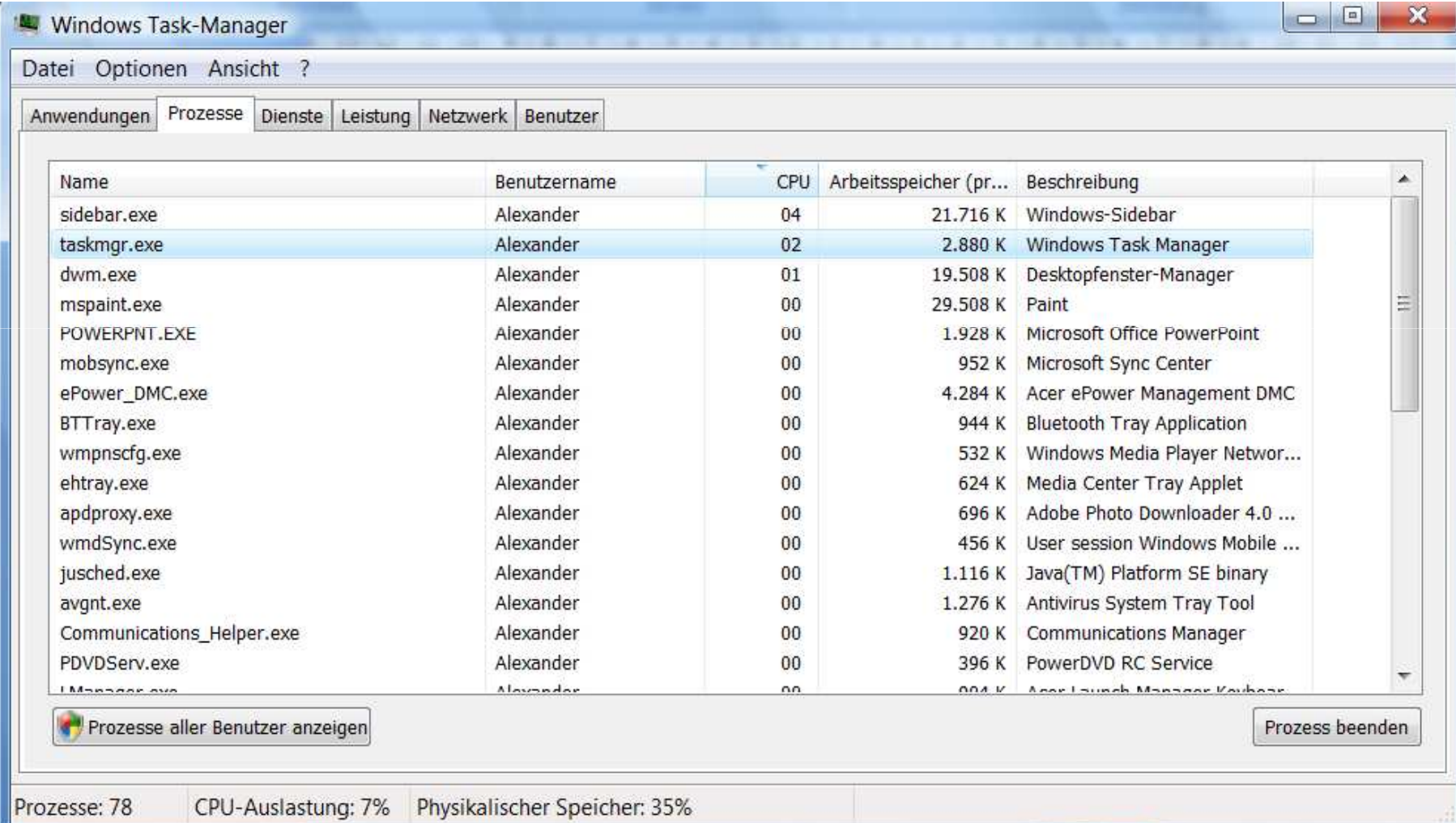
Operating system

Administration of resources



- Some devices send interrupts which start specific program sections
- In most cases interrupts are non-interruptable
- An interrupt allows to assign resource independent from time slice
- A interrupt service routine is activated

Operating system



The screenshot shows the Windows Task Manager window with the 'Prozesse' (Processes) tab selected. The window title is 'Windows Task-Manager'. The menu bar includes 'Datei', 'Optionen', 'Ansicht', and '?'. The tabs at the top are 'Anwendungen', 'Prozesse', 'Dienste', 'Leistung', 'Netzwerk', and 'Benutzer'. The main area displays a table of running processes. The status bar at the bottom shows 'Prozesse: 78', 'CPU-Auslastung: 7%', and 'Physikalischer Speicher: 35%'. At the bottom left, there is a button 'Prozesse aller Benutzer anzeigen' with a multi-colored icon. At the bottom right, there is a button 'Prozess beenden'.

Name	Benutzername	CPU	Arbeitsspeicher (pr...	Beschreibung
sidebar.exe	Alexander	04	21.716 K	Windows-Sidebar
taskmgr.exe	Alexander	02	2.880 K	Windows Task Manager
dwm.exe	Alexander	01	19.508 K	Desktopfenster-Manager
mspaint.exe	Alexander	00	29.508 K	Paint
POWERPNT.EXE	Alexander	00	1.928 K	Microsoft Office PowerPoint
mobsync.exe	Alexander	00	952 K	Microsoft Sync Center
ePower_DMC.exe	Alexander	00	4.284 K	Acer ePower Management DMC
BTTray.exe	Alexander	00	944 K	Bluetooth Tray Application
wmpnscfg.exe	Alexander	00	532 K	Windows Media Player Networ...
ehtray.exe	Alexander	00	624 K	Media Center Tray Applet
apdproxy.exe	Alexander	00	696 K	Adobe Photo Downloader 4.0 ...
wmdSync.exe	Alexander	00	456 K	User session Windows Mobile ...
jusched.exe	Alexander	00	1.116 K	Java(TM) Platform SE binary
avgnt.exe	Alexander	00	1.276 K	Antivirus System Tray Tool
Communications_Helper.exe	Alexander	00	920 K	Communications Manager
PDVDServ.exe	Alexander	00	396 K	PowerDVD RC Service
...

Operating system

Administration of resources

- **Management of the memory**

- Where are code and data located at the memory (address + offsets)

- Swapping onto virtual memory on disk

- Defining critical sections where only one process is allowed to access memory (semaphore)

- ...

- **Management of the hardware**

- Using abstracting drivers for each device

- Managing multiple access by „parallel“ processes (non-interruptable sections)

- **Management of the file system**

- Manage file access and life cycles

- Administrate directory trees

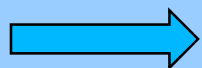
- Administrate file attributes to manage a logical order of creation and touch times

- **Interprocess communication**

- Manage communication between processes (pipes, signals, shared memories, etc.)

- Handle barriers for process synchronisation

- ...



Operating systems offers a high level of abstraction
which is used when programming

Operating system => Middleware

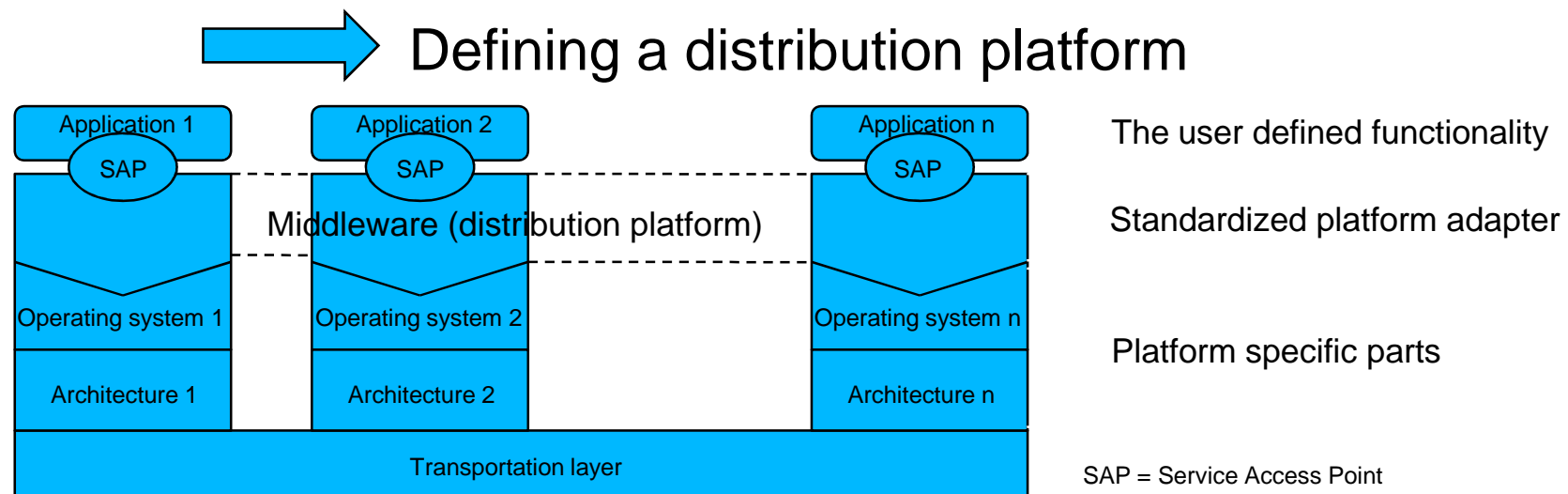
Higher level of abstraction

Distributed systems become more and more necessary.

There are several solutions to easy distributed communication.

They have a higher level of abstraction as commonness cause:

- . Communication is still quite complex
- . System calls are different at different platforms
- . There are several additional problems on distributed systems: distributed clocks, timing, logical and absolute order of events, finding services, ...





Operating system



Give five tasks an operating system kernel must do?

What is non-re-entrant?

What is the difference between cooperative and pre-emptive scheduling?

What is starvation in combination with scheduling at the operating system kernel?

What is an interrupt service routine?

What is the idea behind middleware? Why is middleware beneficial when using different operating systems?

Lecture 2: Software and computer language theory

- ✓ Review: Assembler
- ✓ Operating system
 - (Higher) languages theory
 - Programming paradigms
 - Structures of a program
 - Compilation and interpretation

Language theory: start with assembler

Advantages of Assembler:

- Compared with former techniques, mnemonics are quite easier for programming
- Usage of all hardware possibilities of a specific computer architecture
- Very fast programs, cause of easy, directly into machine code converted instructions
- Very „slim“ codes

Disadvantages of Assembler:

- Not easily readable, especially when large codes
- Dependent on computer architecture (especially on CPU)
- Bad maintainability and portability to other architectures
- Costly in development



Need of higher, problemoriented languages

(Assembler in most cases used for operating systems and hardware drivers)

Language theory: basic terminology

Computer languages given by an alphabet, a syntax, a (formal) grammar and the semantics



Alphabet: Lexical set of symbols

Syntax: Set of production rules, which defines the combinations allowed on the alphabet

Grammar: Precise description of a formal language based on an alphabet and syntax using variables and starting at a dedicated starting point to generate the language

Semantics: Meaning and interpretation of the resulting word combinations for a following work flow on a computer architecture

Language theory: grammar

Formal definition of a grammar

A grammar G is a 4-tuple $G = (V, \Sigma, P, S)$

where

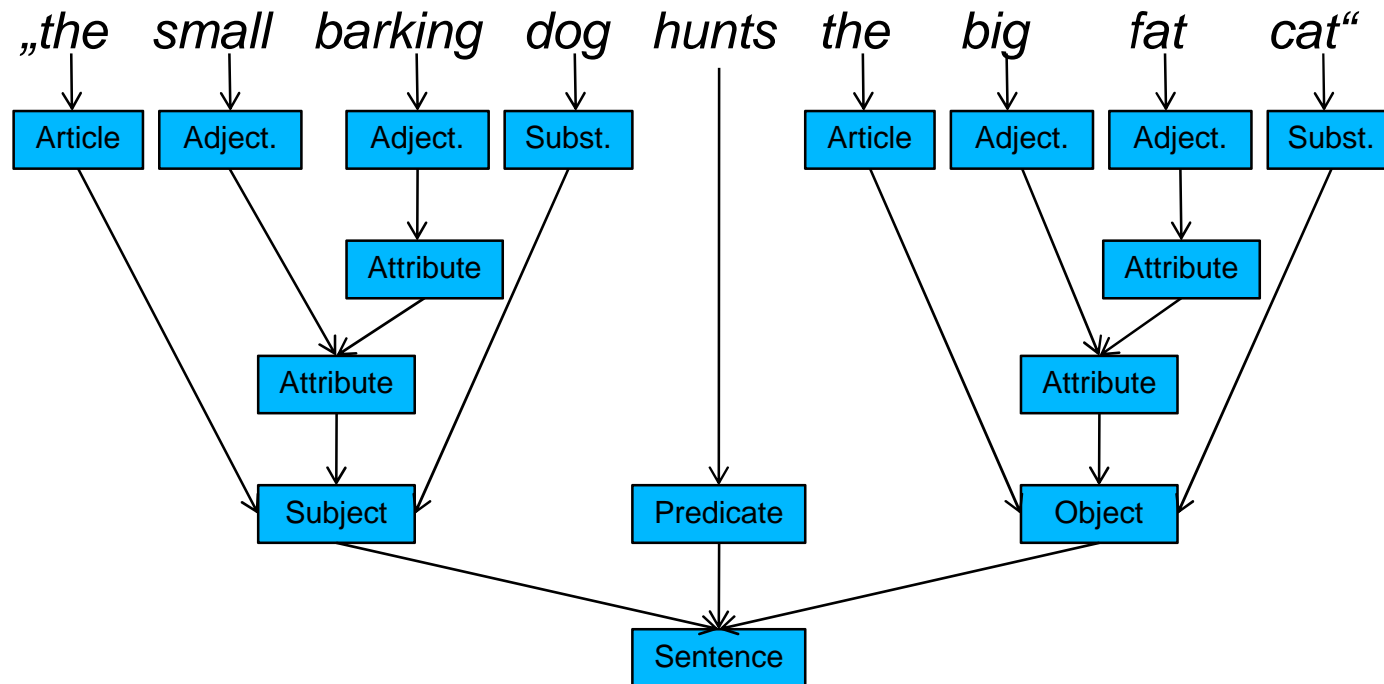
1. V is a finite set of *non-terminal* characters or variables. They represent different types of phrase or clause in the sentence.
2. Σ is a finite set of *terminals*, disjoint with V , which make up the actual content of the sentence.
3. S is the start variable, used to represent the whole sentence (or program). It must be an element of V .
4. P is a finite set of production rules (relations). The members of are called the *rules* or *productions* of the grammar.

We define the relation: $u \Rightarrow v$ (we say u yields v) with $u, v \in (V \cup \Sigma)^*$ such that there is a production rule $\exists(\alpha, \beta) \in R$ transforming $u = u_1 \alpha u_2$ into $v = u_1 \beta u_2$ where $u_1, u_2 \in (V \cup \Sigma)^*$. Thus v is the result of applying the rule (α, β) to u .

The resulting language is $L(G) = \{w \in \Sigma^* : S \xRightarrow{*} w\}$

Language theory: grammar

A simple example:




So it's possible to define rules of the form: left side \rightarrow right side
 e.g.: $\langle \text{Sentence} \rangle \rightarrow \langle \text{Subject} \rangle \langle \text{Predicate} \rangle \langle \text{Object} \rangle$

and it's possible to define with a finite grammar infinite languages
 e.g.: „The small barking dog hunts the big fat fat fat fat ... cat“ is possible

Language theory: grammar (example)

„the small barking dog hunts the big fat cat“

<u>Sentence</u>	-> <i>Subject Predicate Object</i>	
<i>Subject</i>	-> <i>Article Attribute Substantive</i>	
<i>Object</i>	-> <i>Article Attribute Substantive</i>	
<i>Attribute</i>	-> <i>Attribute Adjective</i>	
<i>Article</i>	-> „the“	
<i>Adjective</i>	-> „small“ „barking“ „big“ „fat“	
<i>Substantive</i>	-> „dog“ „cat“	
<i>Predicate</i>	-> „hunts“	

Sentence: Startvariable/-symbol

Subject: Non-terminal symbol

„small“: Terminal symbol

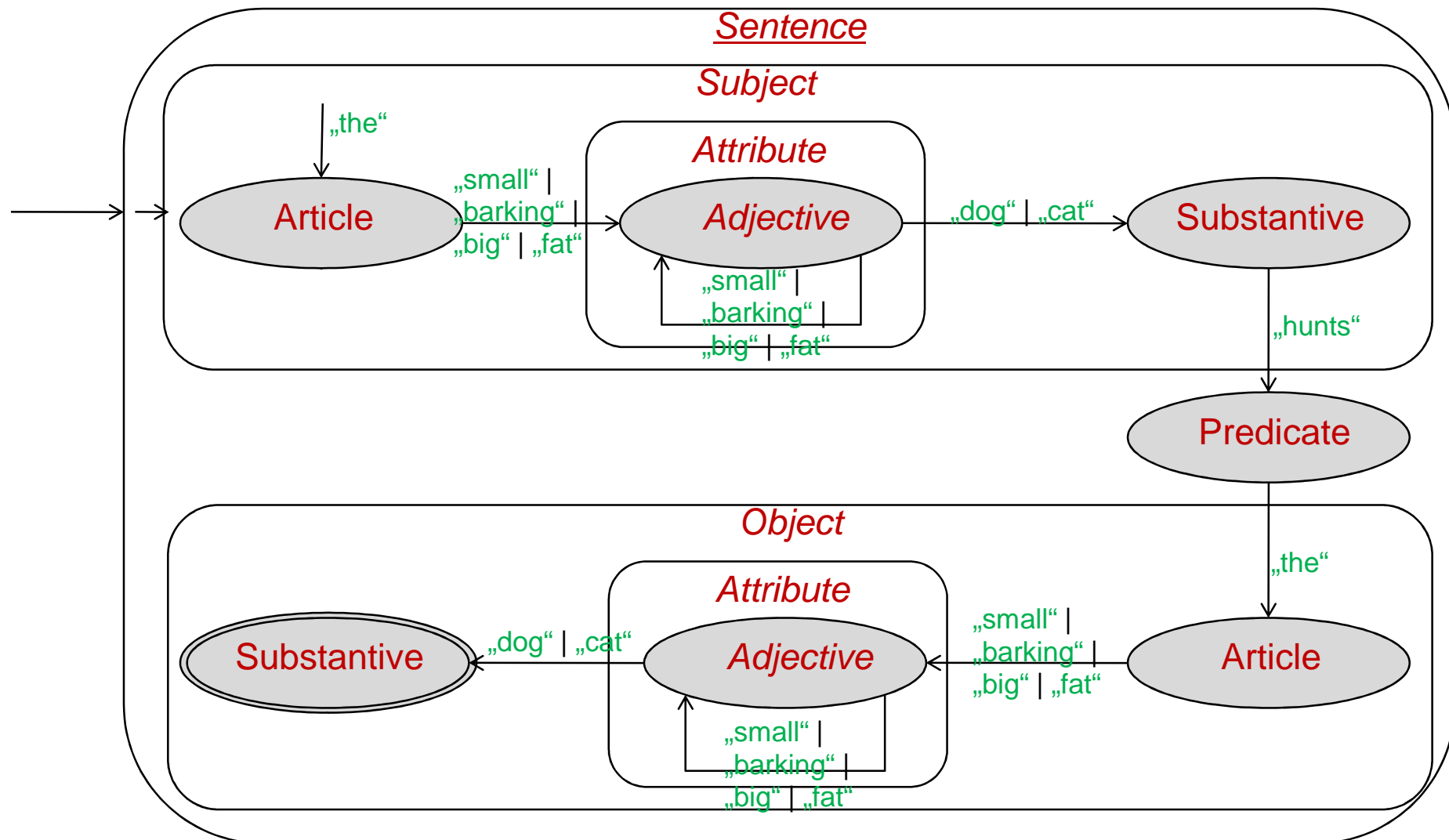
->: Production rule („replace left side by right side“)

<white space> : Concatenation

| : Or

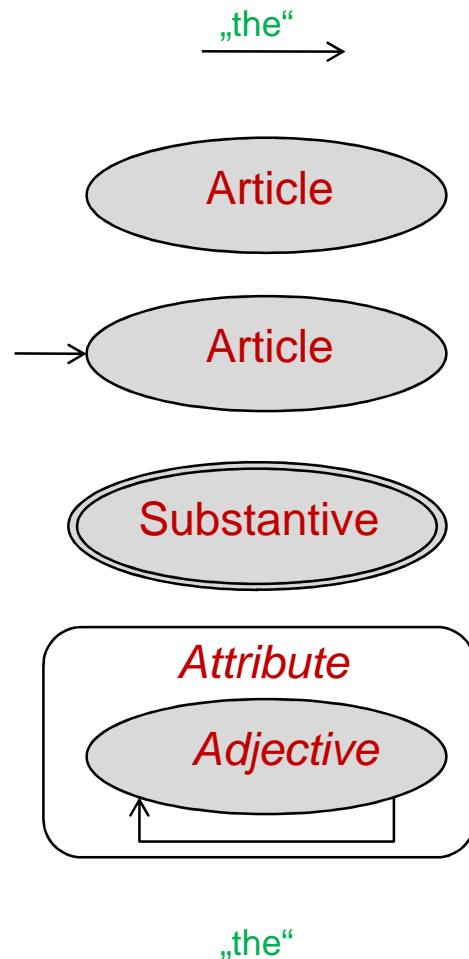
Language theory: grammar (parsing state graph)

„the small barking dog hunts the big fat cat“



Language theory: grammar (parsing graph)

„the small barking dog hunts the big fat cat“



Edge (Lines) with label (token) „the“

Node (Vertex) with label „Article“

Start-Node (Vertex) with label „Article“:
Graph starts here

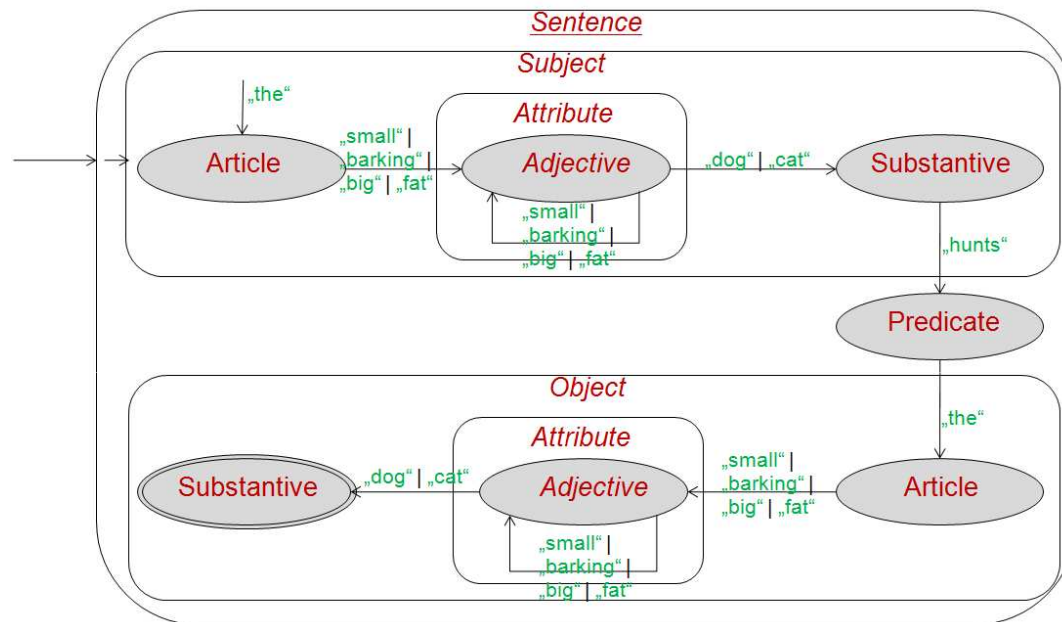
End-Node (Vertex) with label „Substantive“
Graph ends here

Combination-Node (Vertex) with label „Attribute“
Only for structuring the nodes into blocks

Token (detected word)

Language theory: grammar (parsing graph)

„the small barking dog hunts the big fat cat“



But which other sentences can be created by this grammar graph?

„the small small small barking dog hunts the big fat cat“

„the big barking dog hunts the small fat cat“

„the big fat dog hunts the small fat cat“

BUT ALSO:

~~„the big barking dog hunts the small big cat“~~

~~„the big fat dog hunts the small barking cat“~~

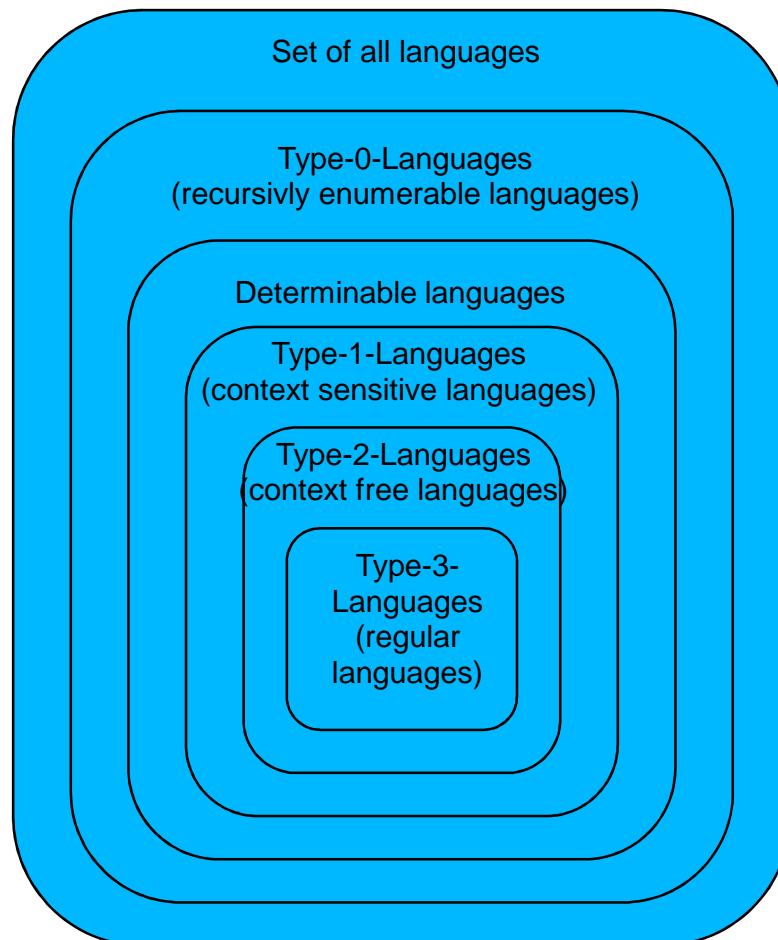
~~Semantics!?~~

But grammatical correct!

Language theory: Chomsky Hierarchy

The Chomsky Hierarchy of computer languages:

(defined by Avram Noam Chomsky 1956 and Marcel-Paul Schützenberger)



The Chomsky hierarchy consists of the following levels:

- Type-0 grammars (**unrestricted grammars**) include all formal grammars. They generate exactly all languages that can be recognized by a **Turing machine**. These languages are also known as the **recursively enumerable languages**. Note that this is different from the **recursive languages** which can be *decided* by an **always-halting Turing machine**.
- Type-1 grammars (**context-sensitive grammars**) generate the **context-sensitive languages**. These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with A a nonterminal and α , β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a **linear bounded automaton** (a nondeterministic Turing machine whose tape is bounded by a constant times the length of the input.)
- Type-2 grammars (**context-free grammars**) generate the **context-free languages**. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic **pushdown automaton**. Context free languages are the theoretical basis for the syntax of most **programming languages**.
- Type-3 grammars (**regular grammars**) generate the **regular languages**. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed (or preceded, but not both in the same grammar) by a single nonterminal. The rule $S \rightarrow \epsilon$ is also allowed here if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a **finite state automaton**. Additionally, this family of formal languages can be obtained by **regular expressions**. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

Note that the set of grammars corresponding to recursive languages is not a member of this hierarchy.

Every regular language is context-free, every context-free language is context-sensitive and every context-sensitive language is recursive and every recursive language is recursively enumerable. These are all proper inclusions, meaning that there exist recursively enumerable languages which are not recursive, recursive languages that are not context-sensitive, context-sensitive languages which are not context-free and context-free languages which are not regular.

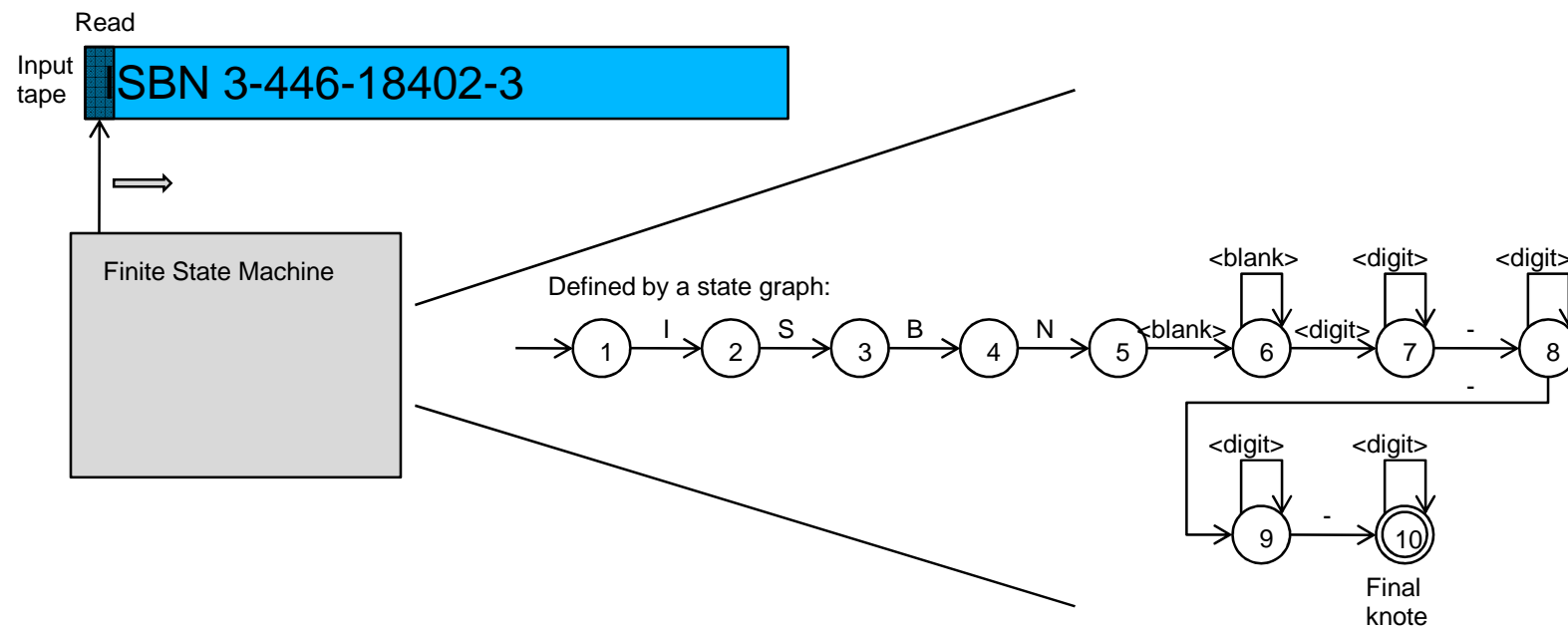
Language theory: Chomsky Hierarchy – type 3

Interpretation of a language with state machines (automaton):

Type-3-Languages (regular languages): Finite state machine

$A \rightarrow \alpha, A \rightarrow \alpha B$ with $A, B \in \text{variables}$ and $\alpha \in \text{terminals}$

e.g. regular expression for book ISBN codes: ISBN 3-446-18402-3



Notice: There is no memory for former states. The machine does its decisions only given by current state.

Language theory: Chomsky Hierarchy – type 3

Type-3-Languages (regular languages): Finite state machine

$A \rightarrow \alpha, A \rightarrow \alpha B$ with $A, B \in$ variables and $\alpha \in$ terminals

e.g. regular expression for book ISBN codes: ISBN 3-446-18402-3

Regular expressions (e.g. in language Perl as a simple example)

`/^[I][S][B][N][]+\d+[-]\d+[-]\d+[-]\d+$/` (with `\d` = digit 0,1,2,3,...,9)

```
\w Match a "word" character (alphanumeric plus "_")
\W Match a non-word character
\s Match a whitespace character
\S Match a non-whitespace character
\d Match a digit character
\D Match a non-digit character
```

<code>\</code> Quote the next metacharacter	<code>*</code> Match 0 or more times
<code>^</code> Match the beginning of the line	<code>+</code> Match 1 or more times
<code>.</code> Match any character (except newline)	<code>?</code> Match 1 or 0 times
<code>\$</code> Match the end of the line (or before newline at the end)	<code>{n}</code> Match exactly n times
<code> </code> Alternation	<code>{n,}</code> Match at least n times
<code>()</code> Grouping	<code>{n,m}</code> Match at least n but not more than m times
<code>[]</code> Character class	

Language theory: Chomsky Hierarchy – type 2

Type-2-Languages (context free languages): Pushdown automaton (stack machine)

A \rightarrow combination of non-terminals and terminals

e.g. ETF-grammar for arithmetical expressions with optional brackets

$E \rightarrow T \mid E+T$

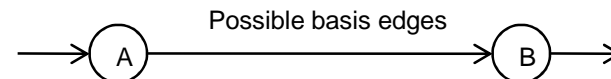
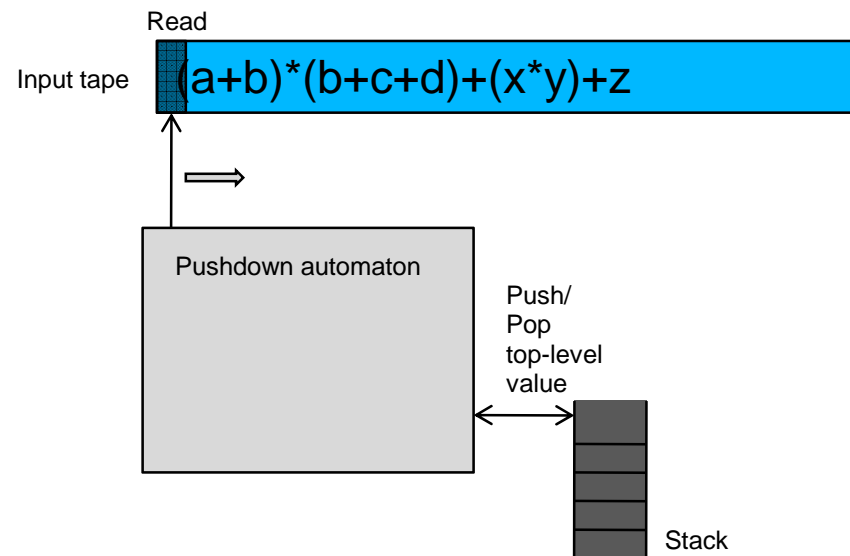
$T \rightarrow F \mid T * F$

$F \rightarrow (E) \mid a \mid b \mid c \mid d \mid \dots \mid z$

Non-terminal symbols: E, T, F

Terminal symbols: $+, *, (,), a, b, c, \dots, z$

Startsymbol: E



Read input and change state (similar to type-3-grammar)

Read input, push to stack and don't change state

Read input, push to stack and change state

Read input, pop top-level element from stack and don't change state

Read input, pop top-level element from stack and change state

Pop top-level element from stack and don't change state

Pop top-level element from stack and change state

\Rightarrow State changes depend on stack content, which changes during processing

Language theory: Chomsky Hierarchy – type 2

Type-2-Languages (context free languages): Pushdown automaton (stack machine)

$A \rightarrow$ combination of non-terminals and terminals

e.g. ETF-grammar for arithmetical expressions with optional brackets

$E \rightarrow T \mid E+T$

$T \rightarrow F \mid T * F$

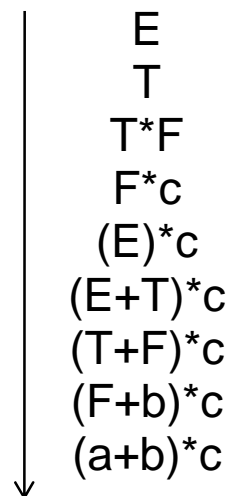
$F \rightarrow (E) \mid a \mid b \mid c \mid d \mid \dots \mid z$

Non-terminal symbols: E, T, F

Terminal symbols: $+, *, (,), a, b, c, \dots, z$

Startsymbol: E

e.g.: create mathematical expression „ $(a+b)^*c$ “





Language theory: Chomsky Hierarchy – type 2



Given the ETF -grammar.

ETF -grammar:

Non-terminal symbols: E, T, F

*Terminal symbols: $+, *, (,), a, b, c, \dots, z$*

Startsymbol: E

Productionrules: $E \rightarrow T \mid E+T$

$T \rightarrow F \mid T^*F$

$F \rightarrow (E) \mid a \mid b \mid c \mid d \mid \dots \mid z$

Draw the production tree to create the following expression $(a+b)^(b+c)$.*

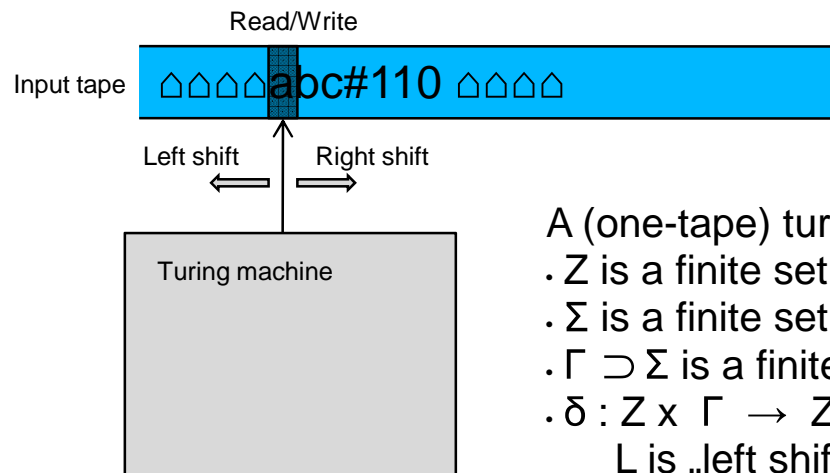
(advice for final test: task type could be similar with changed expression)

Language theory: Chomsky Hierarchy – type 1/ type 0

Interpretation of a language with turing machines:

Type-0- and Type-1-Languages (recursively enumerable and context sensitive languages):
turing machine (with not bounded or bounded tape)

$\alpha A \beta \rightarrow \alpha \gamma \beta$ with $A \in$ non-terminal and α, γ, β combinations of terminals and non-terminals



A (one-tape) turing machine is a 7-tuple $M = (Z, \Sigma, \Gamma, \delta, z_0, \triangle, E)$

- . Z is a finite set of states
- . Σ is a finite set of the tape alphabet (input)
- . $\Gamma \supset \Sigma$ is a finite set of working alphabet
- . $\delta : Z \times \Gamma \rightarrow Z \times \Gamma \times \{L, R, N\}$ is the transition function where
 L is „left shift“, R is „right shift“ and N is „no shift“
- . $z_0 \in Z$ is the start state
- . $\triangle \in \Gamma - \Sigma$ is the „blank symbol“
- . $E \subset Z$ a finite set of final or accepting states

Turing machine deyscribed
 by Alan Turing 1936

Language theory: Chomsky Hierarchy – type 1/ type 0

Interpretation of a language with turing machines:

Type-0- and Type-1-Languages (recursively enumerable and context sensitive languages):
turing machine (with not bounded or bounded tape)

$\alpha A \beta \rightarrow \alpha \gamma \beta$ with $A \in$ non-terminal and α, γ, β combinations of terminals and non-terminals

e.g.: $M = (\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \triangle\}, \delta, z_0, \triangle, \{z_e\})$

$\delta(z_0, 0) = (z_0, 0, R)$

$\delta(z_0, 1) = (z_0, 1, R)$

$\delta(z_0, \triangle) = (z_1, \triangle, L)$

Start is here always on the
most left sign (not \triangle)

$\delta(z_1, 0) = (z_2, 1, L)$

$\delta(z_1, 1) = (z_1, 0, L)$

$\delta(z_1, \triangle) = (z_e, 1, N)$

$\delta(z_2, 0) = (z_2, 0, L)$

$\delta(z_2, 1) = (z_2, 1, L)$

$\delta(z_2, \triangle) = (z_e, \triangle, R)$

current state
↑
read sign
↑
next state
↑
write sign
↑
movement
↑

Language theory: Chomsky Hierarchy – type 1/ type 0

Interpretation of a language with turing machines:

Type-0- and Type-1-Languages (recursively enumerable and context sensitive languages):
turing machine (with not bounded or bounded tape)

$\alpha A \beta \rightarrow \alpha \gamma \beta$ with $A \in$ non-terminal and α, γ, β combinations of terminals and non-terminals

e.g.: $M = (\{z_0, z_1, z_2, z_e\}, \{0,1\}, \{0,1,\triangle\}, \delta, z_0, \triangle, \{z_e\})$

$\delta(z_0, 0) = (z_0, 0, R)$

$\delta(z_0, 1) = (z_0, 1, R)$

$\delta(z_0, \triangle) = (z_1, \triangle, L)$

$\delta(z_1, 0) = (z_2, 1, L)$

$\delta(z_1, 1) = (z_1, 0, L)$

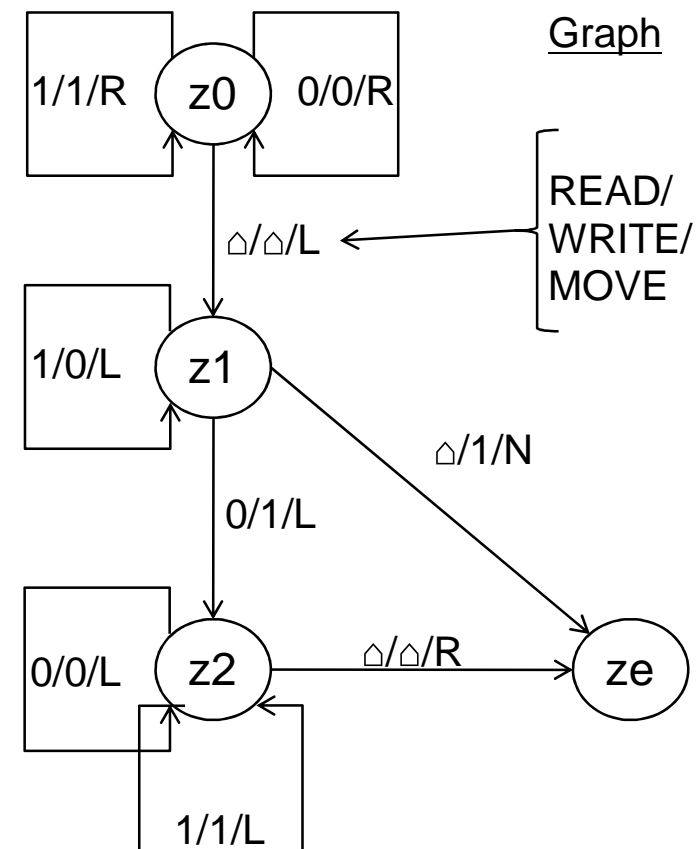
$\delta(z_1, \triangle) = (z_e, 1, N)$

$\delta(z_2, 0) = (z_2, 0, L)$

$\delta(z_2, 1) = (z_2, 1, L)$

$\delta(z_2, \triangle) = (z_e, \triangle, R)$

current state
read sign
next state
write sign
movement

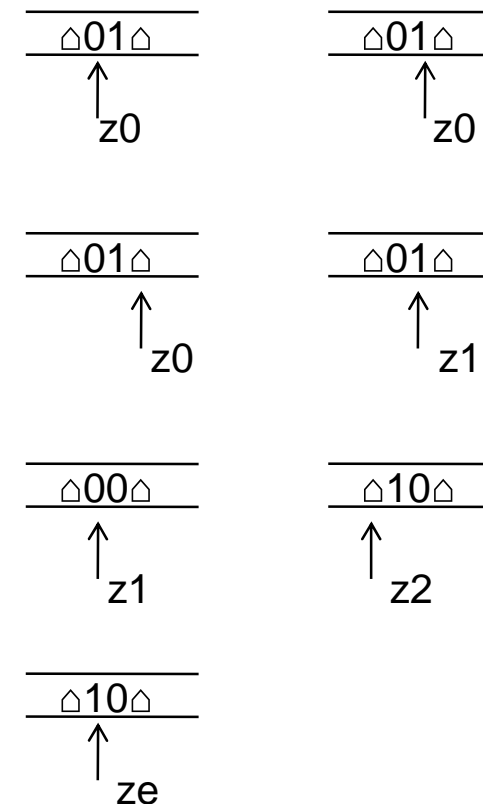
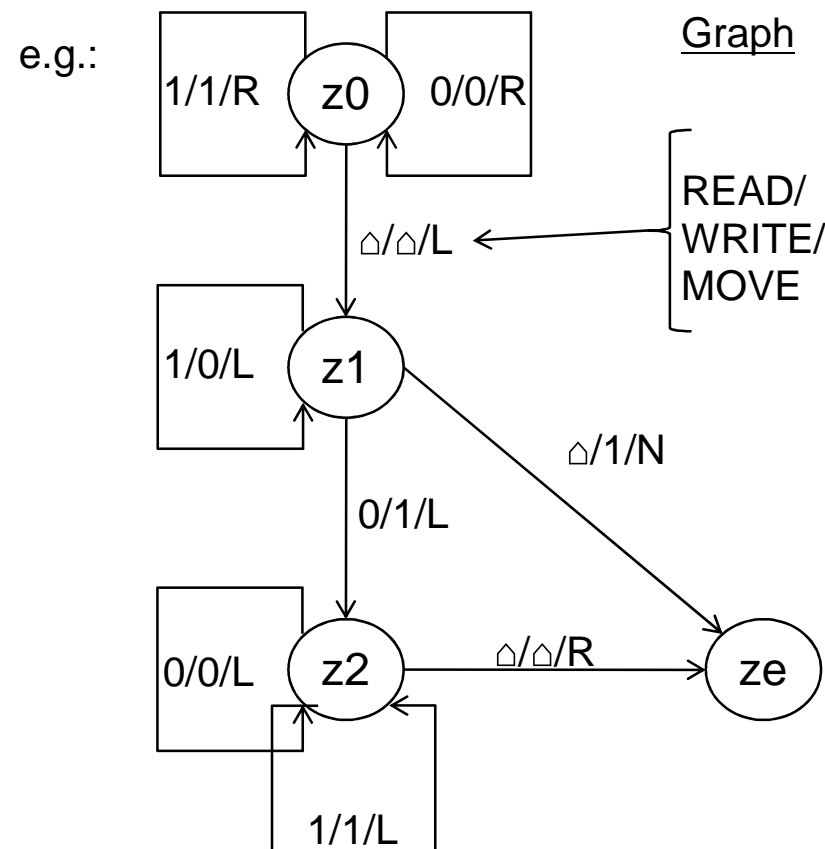


Language theory: Chomsky Hierarchy – type 1/ type 0

Interpretation of a language with turing machines:

Type-0- and Type-1-Languages (recursively enumerable and context sensitive languages):
turing machine (with not bounded or bounded tape)

$\alpha A \beta \rightarrow \alpha \gamma \beta$ with $A \in \text{non-terminal}$ and α, γ, β combinations of terminals and non-terminals



Language theory: Chomsky Hierarchy – type 1/ type 0

Interpretation of a language with turing machines:

Type-0- and Type-1-Languages (recursively enumerable and context sensitive languages):
turing machine (with not bounded or bounded tape)

$\alpha A \beta \rightarrow \alpha \gamma \beta$ with $A \in$ non-terminal and α, γ, β combinations of terminals and non-terminals

e.g.: $M = (\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \triangle\}, \delta, z_0, \triangle, \{z_e\})$

$\delta(z_0, 0) = (z_0, 0, R)$

$\delta(z_0, 1) = (z_0, 1, R)$

$\delta(z_0, \triangle) = (z_1, \triangle, L)$

$\delta(z_1, 0) = (z_2, 1, L)$

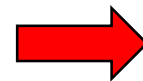
$\delta(z_1, 1) = (z_1, 0, L)$

$\delta(z_1, \triangle) = (z_e, 1, N)$

$\delta(z_2, 0) = (z_2, 0, L)$

$\delta(z_2, 1) = (z_2, 1, L)$

$\delta(z_2, \triangle) = (z_e, \triangle, R)$



Interpret a given binary representation of a number and add 1.



Language theory: Chomsky Hierarchy – type 1/ type 0



Remember the turing machine for adding 1 to a given binary number and write the complete path through the automat when # 111# (# = blank symbol) is the starting input on the tape. (advice for final test: task type could be similar with changed input tape)

$$\begin{aligned}
 M = (&\{z_0, z_1, z_2, z_e\}, \{0, 1\}, \{0, 1, \#\}, \delta, z_0, \#, \{z_e\}) \\
 &\delta(z_0, 0) = (z_0, 0, R) \\
 &\delta(z_0, 1) = (z_0, 1, R) \\
 &\delta(z_0, \#) = (z_1, \#, L) \\
 &\\
 &\delta(z_1, 0) = (z_2, 1, L) \\
 &\delta(z_1, 1) = (z_1, 0, L) \\
 &\delta(z_1, \#) = (z_e, 1, N) \\
 &\\
 &\delta(z_2, 0) = (z_2, 0, L) \\
 &\delta(z_2, 1) = (z_2, 1, L) \\
 &\delta(z_2, \#) = (z_e, \#, R)
 \end{aligned}$$



Language theory: Turing machine



Draw a state diagram for the following turing machine, which converts unary number representation (like $|||||$ for 5_{dec}) into a binary one (like 101 for 5_{dec}). (advice for final test: task type could be similar with changed turing machine, compare state diagram of exercise solution of lecture 2)

$M = (\{s_0, s_1, s_2, s_3\}, \{0, 1, X\}, \{#, , 0, 1, X\}, T,$			$s_0,$	$\#,$	$s_3)$		
states		input alphabet	working alphabet	transitions	start state	blank	final state
$T(s_0,)$	\Rightarrow	(s_1, X, L)					
$T(s_0, \#)$	\Rightarrow	$(s_2, \#, L)$					
$T(s_0, X)$	\Rightarrow	(s_0, X, R)					
$T(s_0, 0)$	\Rightarrow	$(s_0, 0, R)$					
$T(s_0, 1)$	\Rightarrow	$(s_0, 1, R)$					
$T(s_1, \#)$	\Rightarrow	$(s_0, 1, R)$					
$T(s_1, 0)$	\Rightarrow	$(s_0, 1, R)$					
$T(s_1, 1)$	\Rightarrow	$(s_1, 0, L)$					
$T(s_1, X)$	\Rightarrow	(s_1, X, L)					
$T(s_2, X)$	\Rightarrow	$(s_2, \#, L)$					
$T(s_2, 1)$	\Rightarrow	$(s_3, 1, N)$					
$T(s_2, 0)$	\Rightarrow	$(s_3, 0, N)$					

Lecture 2: Software and computer language theory

- ✓ Review: Assembler
- ✓ Operating system
- ✓ (Higher) languages theory
 - Programming paradigms
 - Structures of a program
 - Compilation and interpretation

Programming paradigms

Programming paradigms for problem oriented computer languages

- **Sequential, unstructured programming**
 - *Sequence of instructions with jumps to dedicated code positions*
 - *E.g. Assembler*
- **Procedural programming with structured programming as subset**
 - *Code is splitted into several, reusable sections called procedures or functions with own scopes, which can be called at given code positions*
 - *Logical procedure units are combined to modules*
 - *Jumps (like goto) are not allowed*
 - *E.g. Pascal, C*
- **Object oriented programming**
 - *Using objects (defined with class structures) and messages (interface method calls) to design applications*
 - *Extended techniques, like inheritance, modularity, polymorphism and encapsulation*
 - *E.g. Smalltalk, C++, Java*
- **Aspect oriented programming**
 - *Cross cutting concern as additional functionality which is not immediatly relevant for the functionality of a software itself but very important for developement, like error prevention, simulation and code investigation*
 - *Working with aspects as additional descriptions to the classes*
 - *E.g. extensions to C++*
- **Dataflow driven programming**
 - *State changes in data flows cause the execution of functionality*
 - *Message passing*
 - *E.g. NI LabView*

Lecture 2: Software and computer language theory

- ✓ Review: Assembler
- ✓ Operating system
- ✓ (Higher) languages theory
- ✓ Programming paradigms
 - Structures of a program
 - Compilation and interpretation

Structures of a program

Elements of a problem oriented computer language:

- Memory elements:
 - Data types, user defined data types: *integer, float, boolean, ..., MyType of a structure, ..., string, ...*
 - Static elements (stack elements)
 - Variables: *a, b, i, iIndex, Previous, ...*
 - Constants: *1, 2, 3, ..., 2.5, 10.3, ..., PI defined as 3.14....*
 - Variable combinations: arrays (*a[10], {1,2,3,4},...*), structures/mixed types (*struct of {int, float, string} e.g. {2, 100.5, "Temperature 2"}*)
 - Dynamic elements (heap elements, not always included)
 - Pointer: *"int * ptr" as pointer to int variable*
 - Memory allocation and freeing: *allocate heap memory for 7 integer variables*
 - Referencing/dereferencing: *go to following memory section "ptr++", get value to which ptr points to "*ptr"*
- Operators:
 - Basic operators: *=, +, -, *, /, %*
 - Additional operators: *[], ., ->, *, ...*

Structures of a program

Elements of a problem oriented computer language:

- Application work flow control:
 - Conditions: *if (condition) then ... else ..., switch (value) case ... case ...*
 - Loops: *while (condition) do ..., for value from startvalue to endvalue do ...*
- Input/Output
 - Console: *read from standard in, write to standard out*
 - File system: *open file, read line from file, read block, write line, write block, close file, ...*
- Source code structuring
 - Scopes, namespaces: *{...}, beg ... end, ...*
 - Functions, procedures, subroutines: *ret func (parameters) { ... }*
 - Modules: *separate code into several files*

Structures of a program

Additional elements of a (“problem oriented”) object oriented computer language:

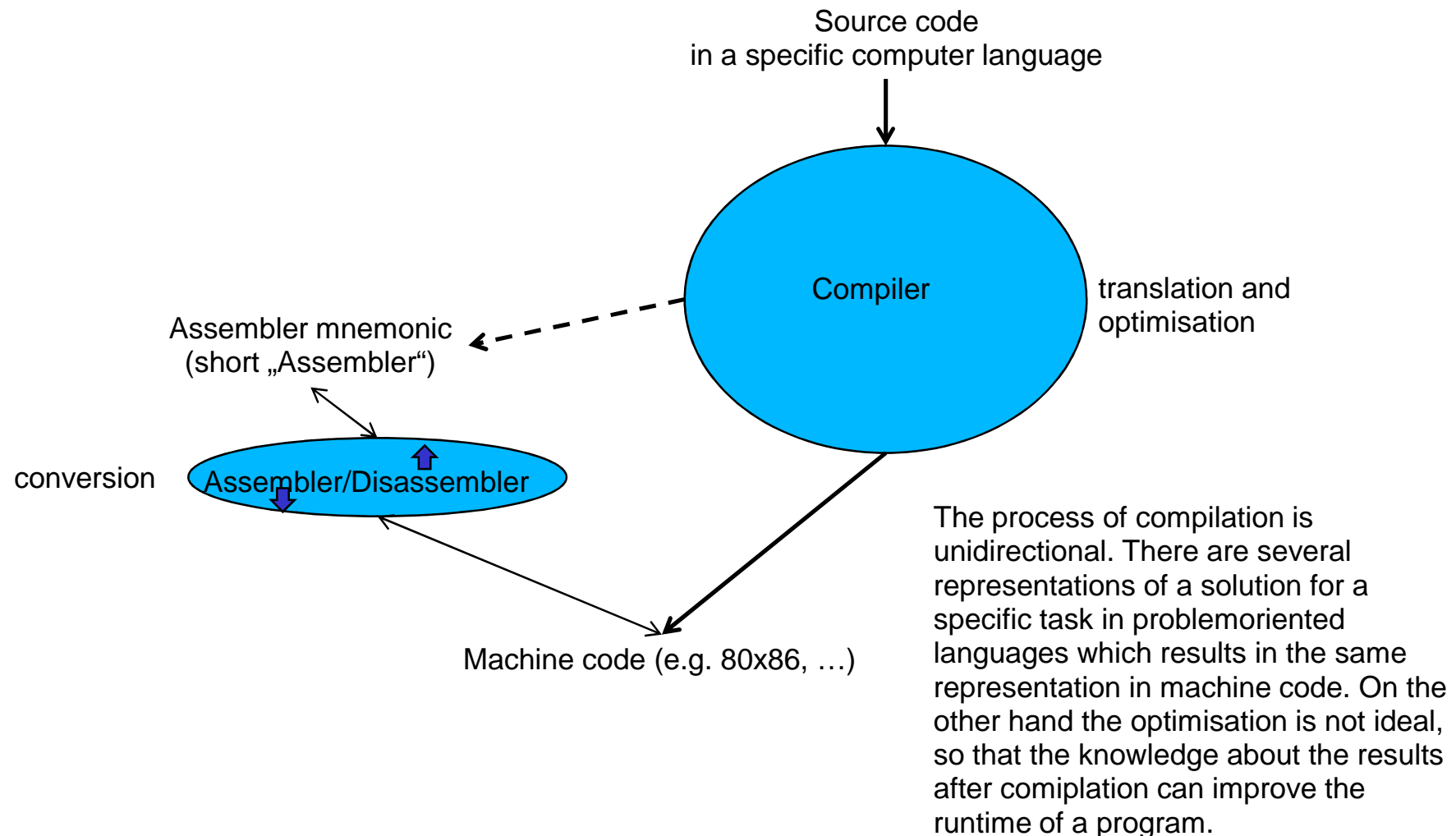
- Additional concepts of higher languages (object oriented)
 - Higher types like classes as object structures: *class A {internal variables, ..., internal methods on variables, ...}*
 - Inheritance of characteristics: *class passenger_car inherits basic characteristics from class car, class van inherits basic characteristics from class car, ...*
 - Operator overloading: *operator + acts specific on specific class*
 - Higher namespaces and access concepts: *named namespace xyz protects scope elements, so that nobody outside can activate them*
 - Templates: *container like lists, vectors, queues, ..., which can deal with different internal types*
 - Error handling: *throw exceptions if error occurs, ...*
 - ...

Lecture 2: Software and computer language theory

- ✓ Review: Assembler
- ✓ Operating system
- ✓ (Higher) languages theory
- ✓ Programming paradigms
- ✓ Structures of a program
 - Compilation and interpretation

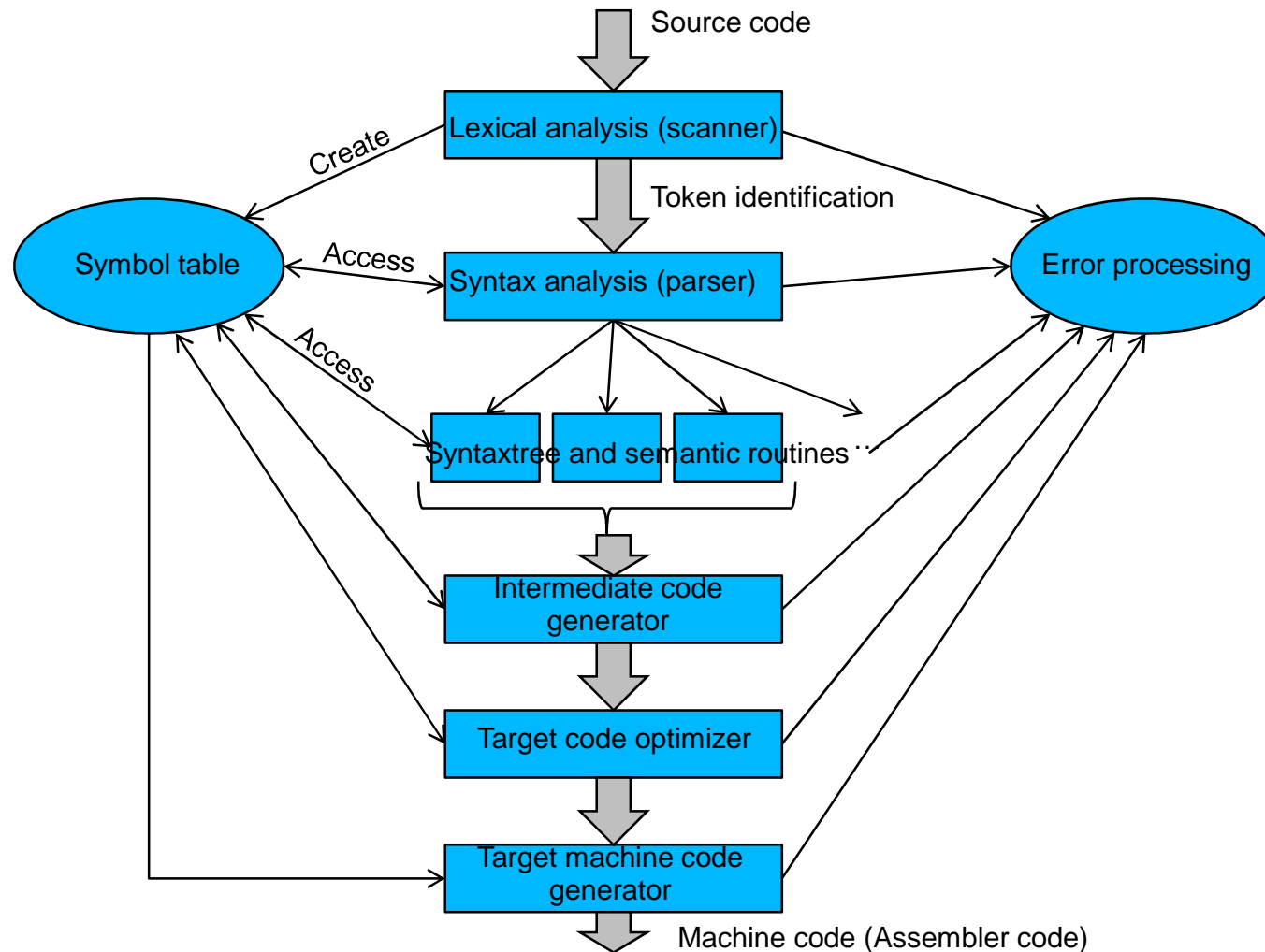
Compilation and interpretation: machine code

From source code to machine instructions



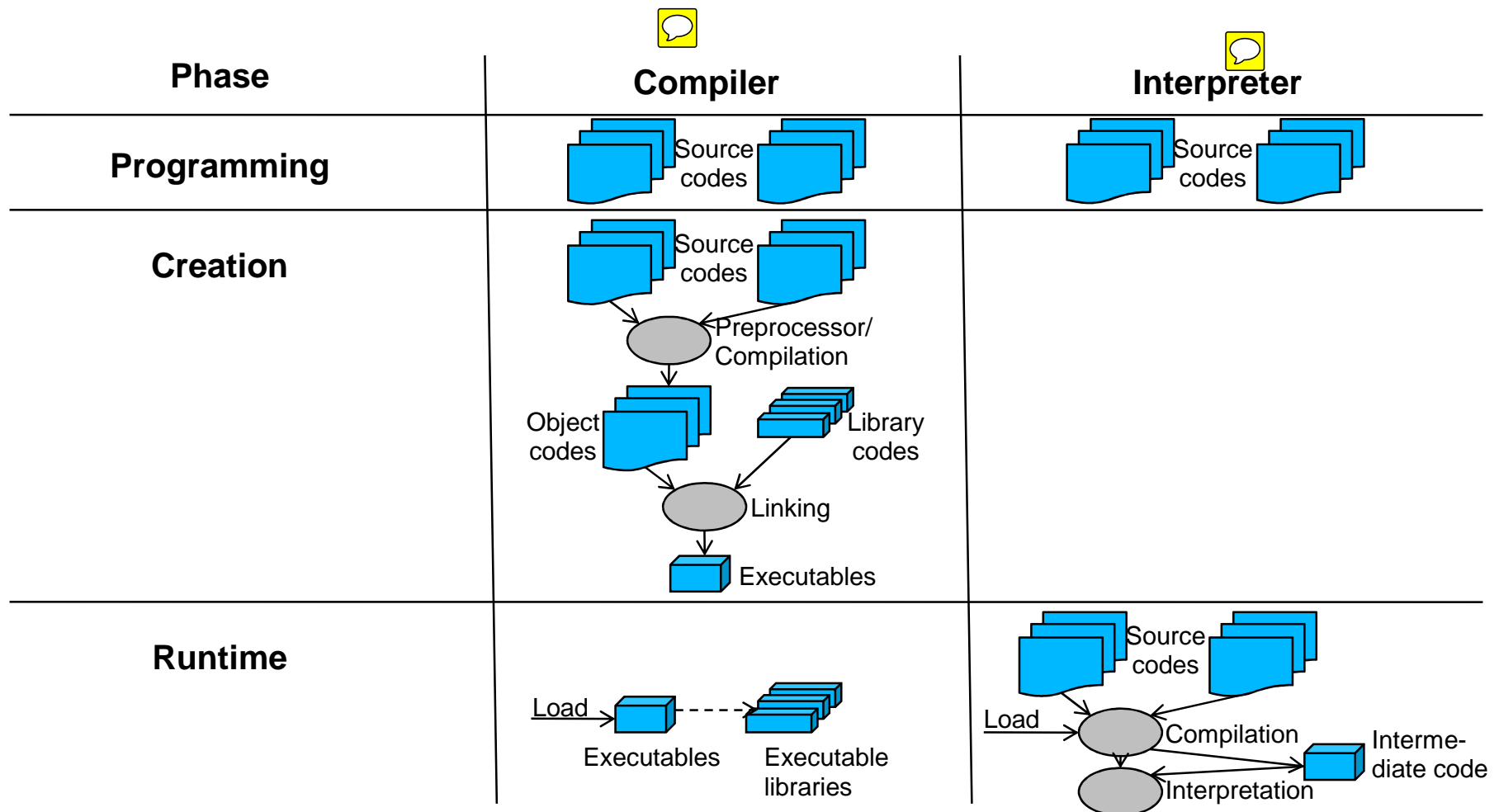
Compilation and interpretation: compilation

Workflow of a compilation



Compilation and interpretation: compiler vs. interpreter

Compiler vs. interpreter



Matlab (I)

Thank you!