



**Hochschule Karlsruhe**  
**Technik und Wirtschaft**  
**UNIVERSITY OF APPLIED SCIENCES**

# **Labor**

## **Systemnahes Programmieren**

Prof. Dr. Thomas Fuchß

Hochschule Karlsruhe – Technik und Wirtschaft

Fakultät für Informatik und Wirtschaftsinformatik





# Übersicht

- **Veranstaltungen**
  - jeweils mittwochs von 14.00 – 19.00 Uhr (Li137 / E311)
- **Zeitplan**
  - Phase I (Scanner) 7 Termine (19.3. – 30.4.)
  - Phase II (Parser) 5 Termine (07.5. – 04.6.)
  - Phase III (Optimierung) 2 Termine (18.6. – 25.6.)
- **Werkzeuge und Sprachen**
  - C, C++ und **keine Datenstrukturen aus der STL**
  - **Eclipse CDT**
  - **Linux**



Hochschule Karlsruhe  
Technik und Wirtschaft  
UNIVERSITY OF APPLIED SCIENCES

# Systemnahes Programmieren Teil II Parser

Prof. Dr. Thomas Fuchß

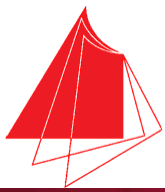
Hochschule Karlsruhe – Technik und Wirtschaft

Fakultät für Informatik und Wirtschaftsinformatik





- A.V. Aho, M.S. Lam, R. Sethi und J.D. Ullmann.  
Compiler – Prinzipien, Techniken und Werkzeuge – 2nd Edition – München: PEARSON STUDIUM, 2008
- M. Kerrisk.  
The Linux Programming Interface: A Linux and UNIX System Programming Handbook – No Starch Press, 2010
- N. Wirth.  
Grundlagen und Techniken des Compilerbaus – Addison-Wesley, 1996
- B. Bauer und R. Höllerer.  
Übersetzung objektorientierter Programmiersprachen : Konzepte, abstrakte Maschinen und Praktikum "Java-Compiler" – Springer, 1998
- D. Grune et. al.  
Modern compiler design – Wiley, 2000
- R. M. Stallman, R. McGrath, P. D. Smith  
GNU Make – Free Software Foundation, 2010([www.gnu.org/software/make/manual/](http://www.gnu.org/software/make/manual/))

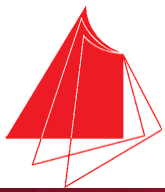


# Ziel:

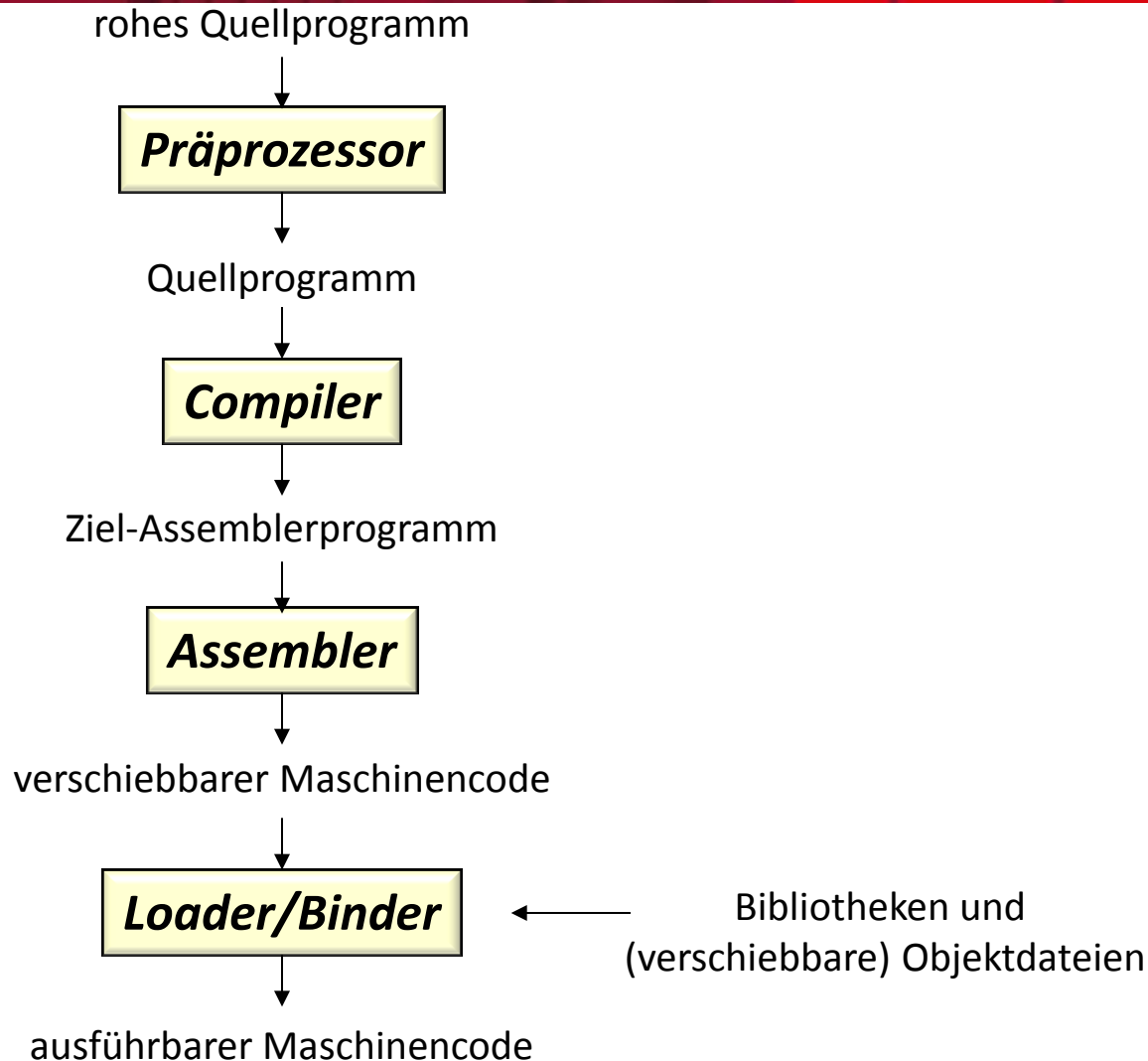
**Ziel der zweiten Laboraufgabe ist es, die Funktionsweise eines Parsers sowie dessen Einordnung innerhalb eines Compilers kennen zu lernen.**

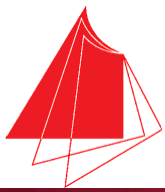
**Insbesondere gilt es, das Prinzip des *rekursiven Abstiegs* verstehen und anwenden zu können.**

**Semantische Analyse und beispielhafte Code-Erzeugung runden die Aufgabe ab.**



# Die Umgebung eines Compilers





# Das Analyse-Synthesemodell

**Der Übersetzungsprozess besteht aus zwei Teilen:**

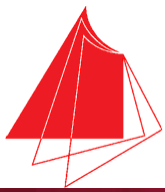
- **Analyse: (Frontend)**

Der Analyse-Teil zerlegt das Quellprogramm in seine Bestandteile und erzeugt eine Zwischendarstellung

- **Synthese: (Backend)**

Der Synthese-Teil konstruiert das gewünschte Zielprogramm aus der Zwischendarstellung

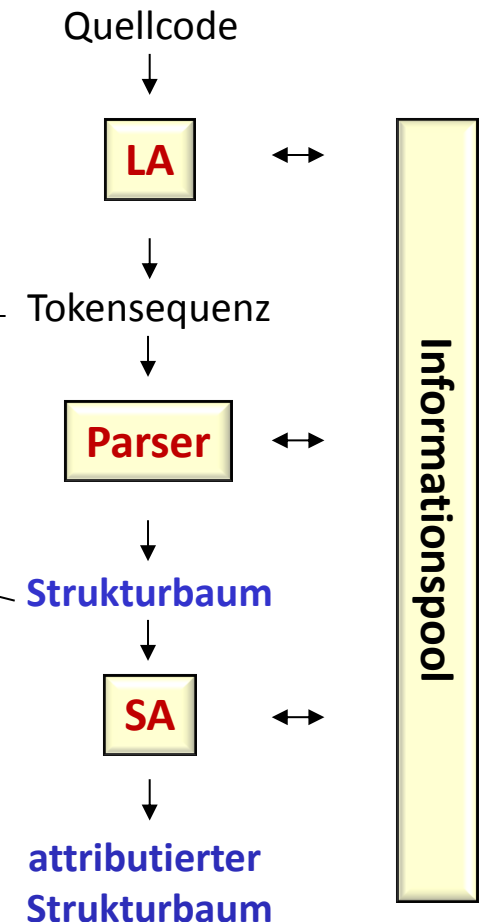
- Code-Erzeugung
- Optimierung



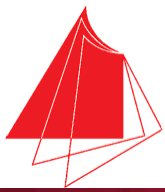
# Die Analyse

## Die Analyse besteht ihrerseits aus mehreren Teilaufgaben

- **Lexikalische Analyse:**
  - Zerlegung des Quellcodes in die Grundsymbole (Tokens)  
Bezeichner, Schlüsselworte, Sonderzeichen, Zahlen, ...
  - Speichern und Weiterleiten von Informationen (Namen, Values)
- **Syntaktische Analyse:**
  - Überprüfung der syntaktischen Spracheigenschaften  
Sind die Ausdrücke korrekt?  $a = b - / + c;$
  - Erstellung des Strukturbaums
- **Semantische Analyse:**
  - Bestimmung der statischen Semantik des Programms
  - Prüfung der Konsistenz
    - Gültigkeitsbereiche (Namensräume)
    - Typisierung (Ausdrücke, Variablen, ...)
    - Deklarationen





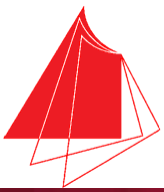


# Was macht man mit der Token-Sequenz?

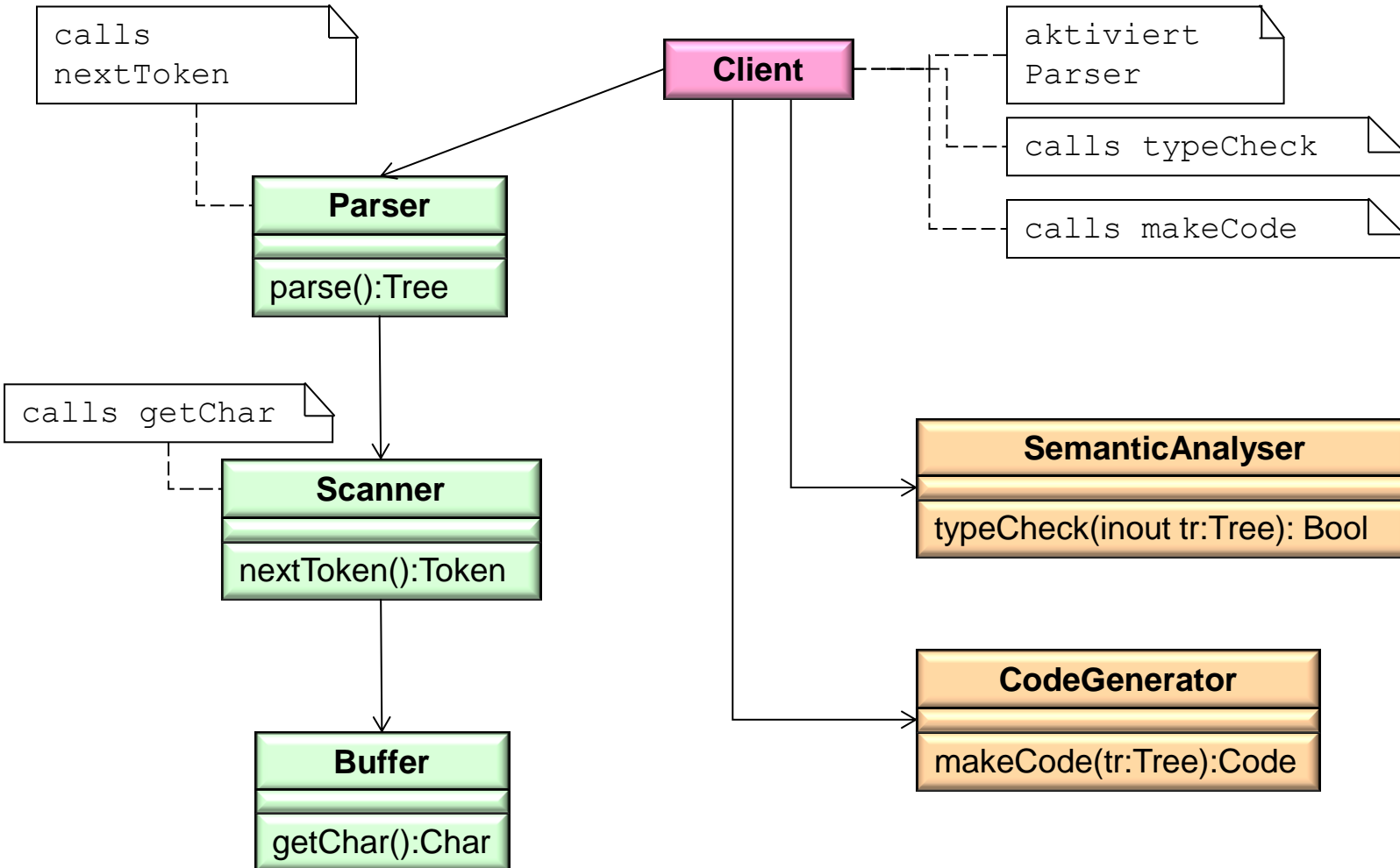
Zu jeder Programmiersprache gehören Regeln, die festlegen, wie die syntaktische Struktur wohlgeformter Programme auszusehen hat.

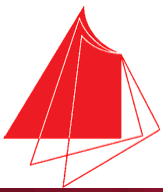
**Der Parser überprüft diese Regeln.**

- Der Parser fordert die Tokens vom Scanner an
- Der Parser prüft, ob die Reihenfolge der Tokens sinnvoll ist (den Regeln der Programmiersprache entspricht)
- Der Parser baut den Strukturbaum (Parse Tree) auf, der in der semantischen Analyse zur Typprüfung genutzt wird.
- Der Parser erkennt und behandelt Fehler.

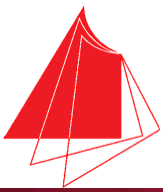


# Ablauf: (funktionale Referenzarchitektur)





# Was soll man tun?



# Aufgaben:

a) Schreiben Sie einen Parser für folgende Grammatik mit Startsymbol PROG.

PROG ::= DECLS STATEMENTS

DECLS ::= DECL ; DECLS |  $\varepsilon$

DECL ::= **int** ARRAY *identifizier*

ARRAY ::= [ *integer* ] |  $\varepsilon$

STATEMENTS ::= STATEMENT ; STATEMENTS |  $\varepsilon$

STATEMENT ::= *identifizier* INDEX = EXP | **print**( EXP ) | **read** ( *identifizier* INDEX ) | { STATEMENTS } |  
**if** ( EXP ) STATEMENT **else** STATEMENT |  
**while** ( EXP ) STATEMENT

EXP ::= EXP2 OP\_EXP

EXP2 ::= ( EXP ) | *identifizier* INDEX | *integer* | - EXP2 | ! EXP2

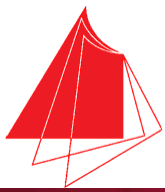
INDEX ::= [ EXP ] |  $\varepsilon$

OP\_EXP ::= OP EXP |  $\varepsilon$

OP ::= + | - | \* | / | < | > | == | != | &

## Hinweis:

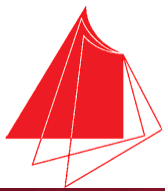
- normale Terminale (Schlüsselwörter) sind klein, **fett** und *rot*
- kleine, **fette**, *kursive* und blaue Terminale stehen für Konstanten bzw. Bezeichner (3, 3.14, x,...).
- Nichtterminale sind groß und *KURSIV* gedruckt



# Aufgaben:

b) Verwenden Sie hierzu den Scanner aus Teil 1 und ergänzen Sie die fehlenden Terminalsymbole.

```
digit      ::= „0“ | „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“
letter     ::= „A“ | „B“ | „C“ | ... | „Z“ | „a“ | „b“ | ... | „z“ |
sign+     ::= „+“
           ...
...     ::= „-“ „*“ „/“ „<“ „>“ „=“ „==“ „!=“ „!“ „&“ „;“ „:“ „(“ „)“ „{“ „}“ „[“
           ...
sign]     ::= „]“
integer   ::= digit {digit}
identifier ::= letter {letter | digit}
print     ::= „print“
read      ::= „read“
if        ::= „if“ | „IF“
else      ::= „else“ | „ELSE“
while     ::= „while“ | „WHILE“
int       ::= „int“
```

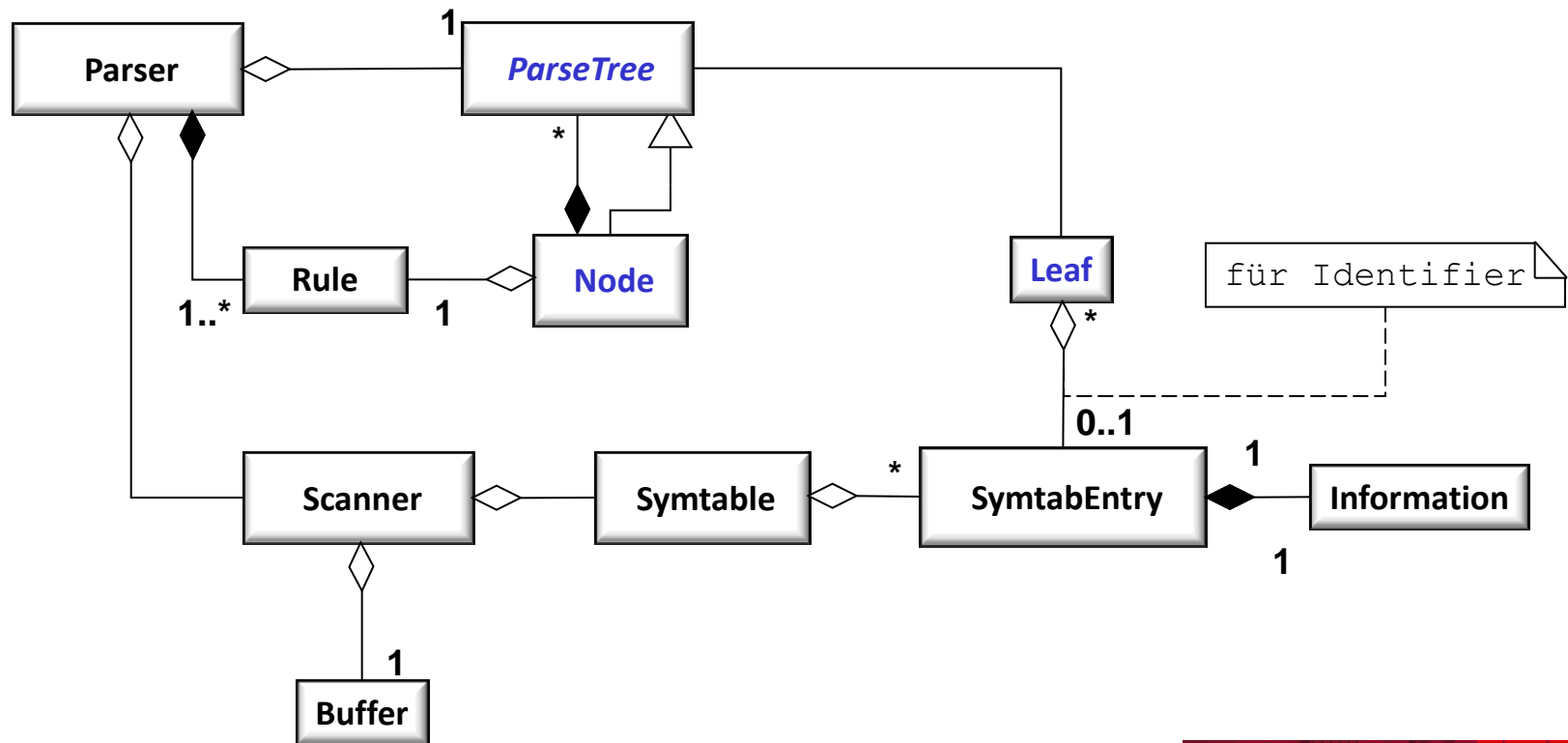


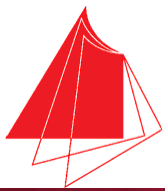
# Aufgaben:

## c) Erstellen Sie einen Strukturbaum (Parse-Baum).

Erweitern Sie hierzu die Funktionen des Parsers, so dass sie nicht nur die Syntax überprüfen, sondern auch den Baum aufbauen.

**(Bem.: Für jede erkannte Regel entsteht ein neuer Teilbaum)**





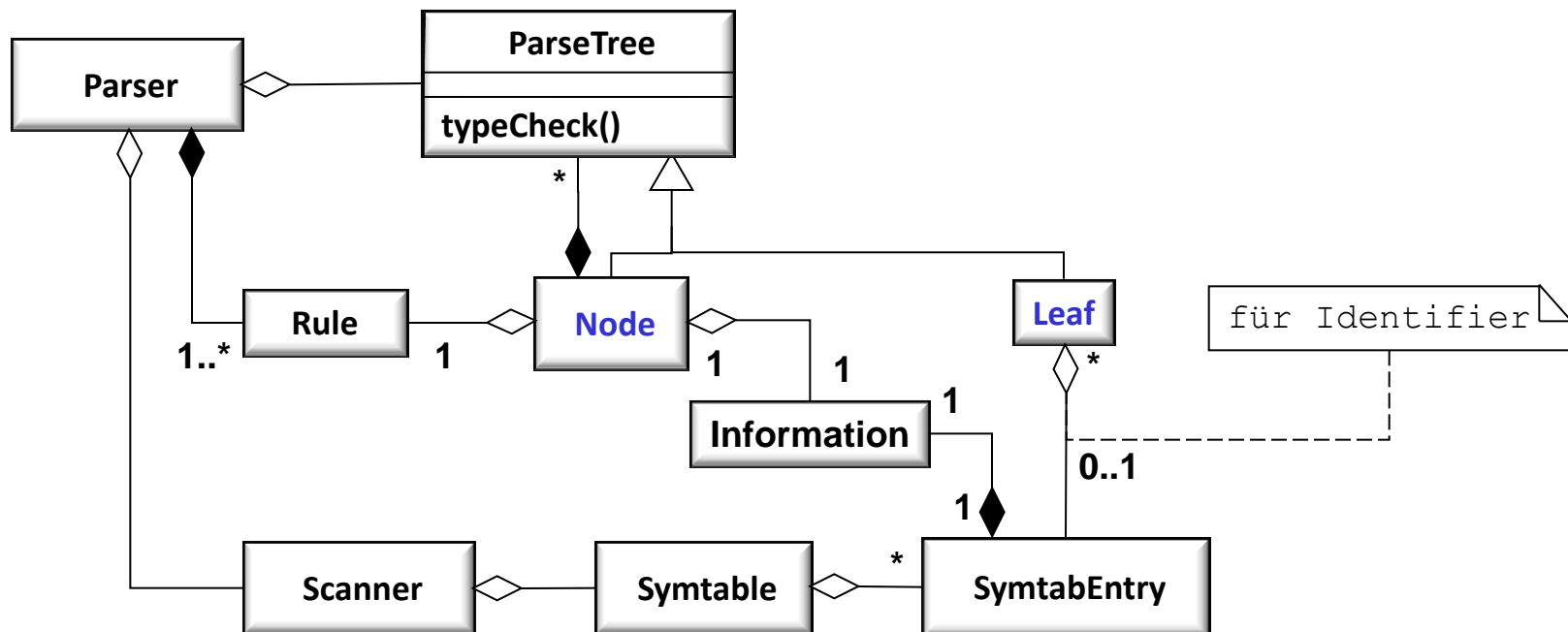
# Aufgaben:

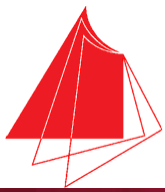
## d) Evaluieren Sie den Parse-Baum :

Ermitteln Sie hierzu zu jedem Knoten den entsprechenden Typ (gemäß Vorlage) und prüfen Sie, ob die Typen der Unterbäume zusammenpassen.

Speichern Sie die Typ-Information im Knoten und für **Identifizier als Information in der Symboltabelle**.

Hinweis: Erweitern Sie hierzu die Klasse *ParseTree* um eine Operation **typeCheck**.



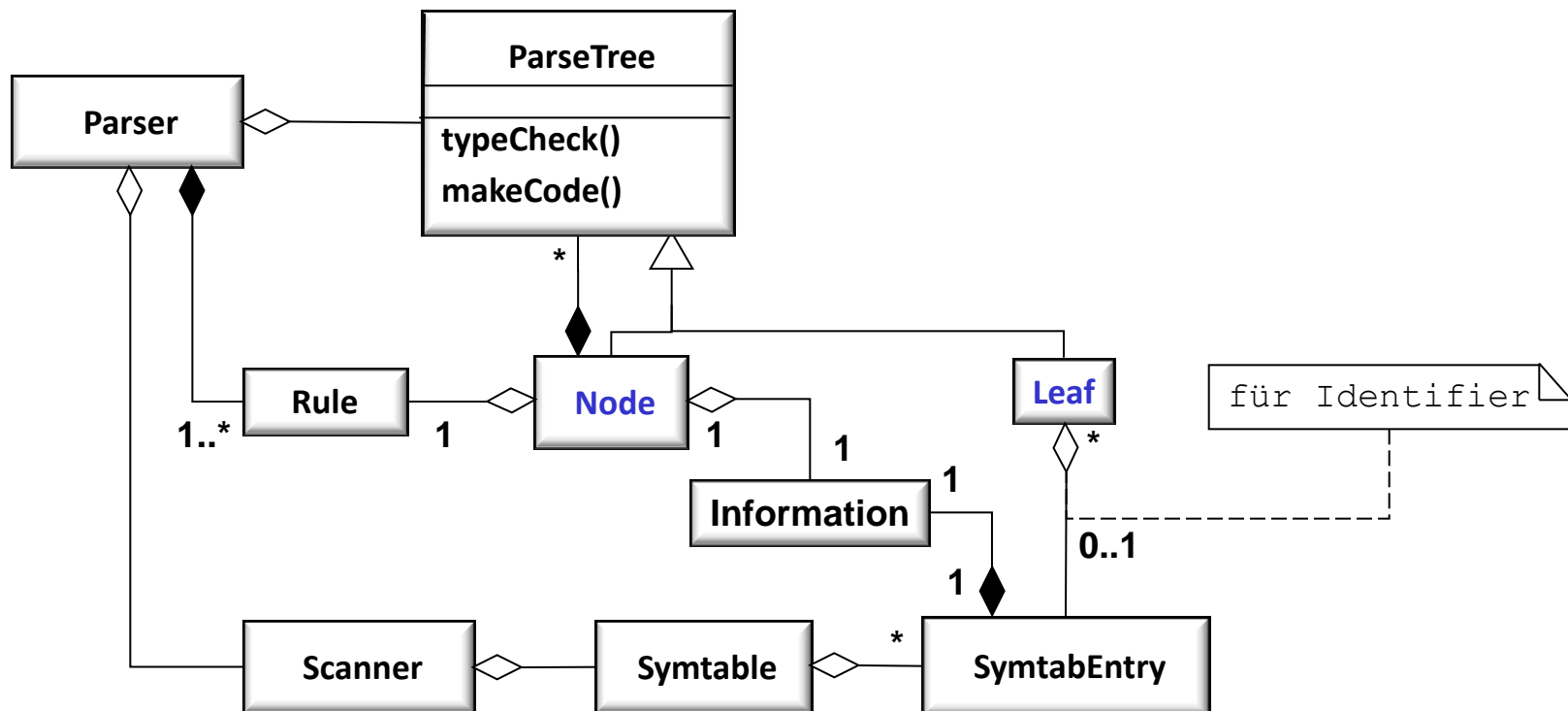


# Aufgaben:

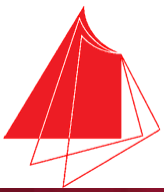
## e) Erzeugen Sie Code

Bestimmen Sie hierzu zu jedem Knoten das entsprechende Code-Segment (gemäß Vorlage) und speichern Sie dieses in einer Code-Datei (xxx.code) ab.

Hinweis: Erweitern Sie hierzu die Klasse *ParseTree* um eine Operation **makeCode**.







# Programmaufruf (I)

```
C:\> parser Parser-test.txt -c test.code
```

- „Parser-test.txt“ Eingabedatei mit dem zu parsenden Programm.

```
int n;  
int[3] m;  
n = 3 + 4;  
n = 3*n--5;  
print(n);
```

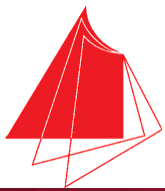
„Parser-test.txt“

```
C:\>parser parser-test.txt -c test.code
```

```
parsing ...
```

```
type checking ...
```

```
generate code ...
```



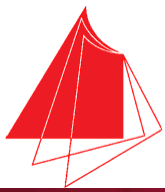
# Programmaufruf (II)

**Die resultierende Code-Datei hat dann etwa folgende Gestalt:**

Um Code-Files zu interpretieren verwenden Sie den zur Verfügung gestellten „Interpreter“.

```
> java -jar interpreter.jar
```

```
DS $n 1
DS $m 3
LC 3
LC 4
ADD
LA $n
STR
LC 3
LA $n
LV
LC 0
LC 5
SUB
SUB
MUL
LA $n
STR
LA $n
LV
PRI
NOP
STP
„test.code“
```



# Programmaufruf (III)

```
C:\> parser Parser-error.txt -c test.code
```

- „Parser-test.txt“ Eingabedatei mit dem zu parsenden Programm.
- Gefundene Fehler werden mit Angabe von Zeile, Spalte, Token auf „**stderr**“ ausgegeben.

```
...  
    n = 3 ) 4;  
...
```

„Parser-error.txt“

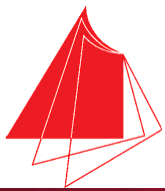
```
C:\>parser parser-error.txt
```

```
parsing ...
```

```
unexpected Token Line:      3   Column:      7   TokenRightParenthesis
```

```
stop
```

```
C:\>
```



# Programmaufruf (IV)

```
C:\> parser Parser-error.txt -c test.code
```

- „Parser-test.txt“ Eingabedatei mit dem zu parsenden Programm.
- Gefundene Fehler werden mit Angabe von Zeile, Spalte, Token auf „**stderr**“ ausgegeben.

```
...
```

```
    n = 3 + m;
```

```
...
```

„Parser-error.txt“

```
C:\>parser parser-error.txt
```

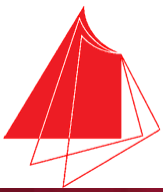
```
parsing ...
```

```
type checking ...
```

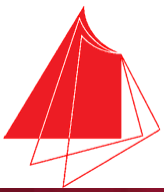
```
error Line: 3 Column: 9 not a primitive Type
```

```
stop
```

```
C:\>
```

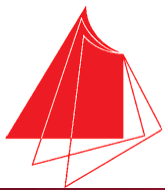


# Wie soll man das tun?



# Warum sehen die Regeln so seltsam aus?

```
PROG      ::=  DECLS STATEMENTS
DECLS     ::=  DECL ; DECLS | ε
DECL      ::=  int ARRAY identifier
ARRAY     ::=  [ integer ] | ε
STATEMENTS ::=  STATEMENT ; STATEMENTS | ε
STATEMENT ::=  identifier INDEX = EXP | print( EXP ) | read ( identifier INDEX ) | {STATEMENTS} |
               if ( EXP ) STATEMENT else STATEMENT |
               while ( EXP ) STATEMENT
EXP       ::=  EXP2 OP_EXP
EXP2      ::=  ( EXP ) | identifier INDEX | integer | - EXP2 | ! EXP2
INDEX     ::=  [ EXP ] | ε
OP_EXP    ::=  OP EXP | ε
OP        ::=  + | - | * | / | < | > | == | != | &
```



# Und nicht so?

*PROG* ::= *DECLS ; STATEMENTS* | *STATEMENTS ;* |  $\varepsilon$

*DECLS* ::= *DECL ; DECLS* | *DECL*

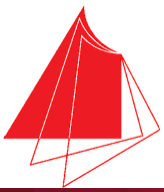
*DECL* ::= **int** [ *integer* ] *identifier* | **int** *identifier*

*STATEMENTS* ::= *STATEMENT ; STATEMENTS* | *STATEMENT*

*STATEMENT* ::= *identifier* = *EXP* | *identifier* [ *EXP* ] = *EXP* | {*STATEMENTS*}  
**print** ( *EXP* ) | **read** (*identifier*) | **read** (*identifier* [ *EXP* ] ) |  
**if** ( *EXP* ) *STATEMENT* **else** *STATEMENT* |  
**while** ( *EXP* ) *STATEMENT*

*EXP* ::= *EXP OP EXP* | ( *EXP* ) | *identifier* | *integer* | - *EXP* | ! *EXP*

*OP* ::= + | - | \* | / | < | > | == | != | &



# Wie überprüft man ein Wort?

Ist **id := id + id \* id** ein Element der Sprache unserer Grammatik ?

**Idee:**

**konstruktiv, man sucht eine Ableitung**

*s*

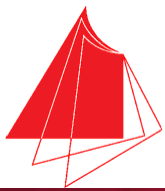
$\vdash^1 \text{id} := e$	$\vdash^1 \text{id} := e \text{ op } e$	$\vdash^1 \text{id} := e \text{ op } e \text{ op } e$
$\vdash^1 \text{id} := \text{id op } e \text{ op } e$	$\vdash^1 \text{id} := \text{id} + e \text{ op } e$	$\vdash^1 \text{id} := \text{id} + \text{id op } e$
$\vdash^1 \text{id} := \text{id} + \text{id} * e$	$\vdash^1 \text{id} := \text{id} + \text{id} * \text{id}$	

oder

$\vdash^1 \text{id} := e$	$\vdash^1 \text{id} := e \text{ op } e$	$\vdash^1 \text{id} := e \text{ op } e \text{ op } e$
$\vdash^1 \text{id} := e \text{ op } e \text{ op } \text{id}$	$\vdash^1 \text{id} := e \text{ op } e * \text{id}$	$\vdash^1 \text{id} := e \text{ op } \text{id} * \text{id}$
$\vdash^1 \text{id} := e + \text{id} * \text{id}$	$\vdash^1 \text{id} := \text{id} + \text{id} * \text{id}$	

...





# Wie findet man eine Ableitung?

**Man versucht sie zu konstruieren.**

## Der allgemeine LL-Akzeptor (Kellermaschine)

$G = (N, T, P, Z)$  kontextfreie Grammatik, bestehend aus:

Nichtterminalen, Terminalen, Produktionen, Startsymbol

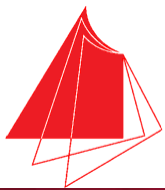
$$A_{LL}(G) = (\{\mathbf{q}\}, N, T, P_{LL}, Z\mathbf{q}, \mathbf{q})$$

$$P = \{t \mathbf{q} \mid t \vdash^1 \mathbf{q} \mid t \in T\} \quad \text{compare}$$

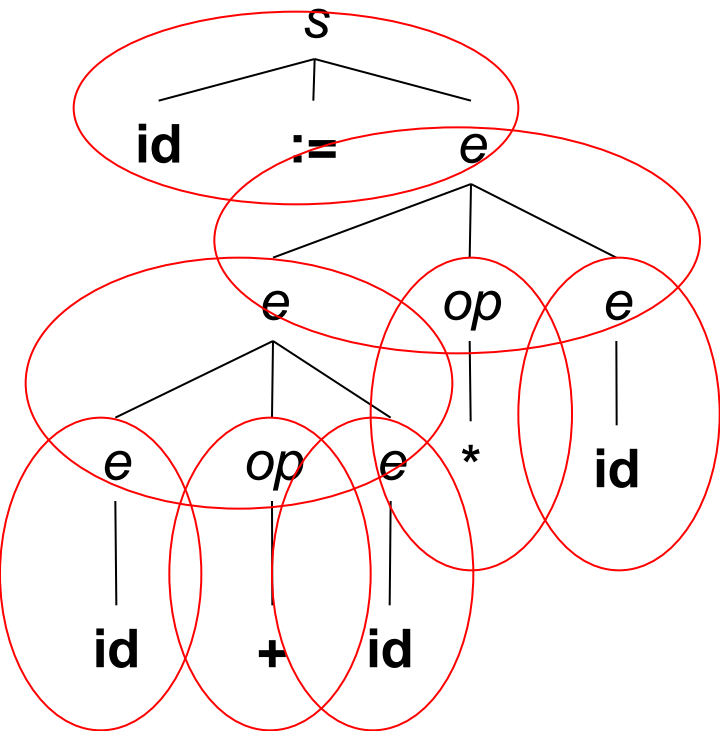
$$\cup \{B \mathbf{q} \vdash^1 b_n \dots b_1 \mathbf{q} \mid B ::= b_1 \dots b_n \in P\} \quad \text{produce}$$

**Zusammenhang:**  $w \in L(G)$  gdw.  $Z \mathbf{q} w \vdash \mathbf{q}$

**Nachteil: nicht deterministisch**



# Beispiel



$\underline{S} \text{ } q \text{ id} := \text{id} + \text{id} * \text{id}$

$\vdash^{-1} e := \text{id } q \text{ id} := \text{id} + \text{id} * \text{id}$

$\vdash^{-1} e := q := \text{id} + \text{id} * \text{id}$

$\vdash^{-1} \underline{e} \text{ } q \text{ id} + \text{id} * \text{id}$

$\vdash^{-1} e \text{ } op \text{ } \underline{e} \text{ } q \text{ id} + \text{id} * \text{id}$

$\vdash^{-1} e \text{ } op \text{ } e \text{ } op \text{ } \underline{e} \text{ } q \text{ id} + \text{id} * \text{id}$

$\vdash^{-1} e \text{ } op \text{ } e \text{ } op \text{ } id \text{ } q \text{ id} + \text{id} * \text{id}$

$\vdash^{-1} e \text{ } op \text{ } e \text{ } op \text{ } q + \text{id} * \text{id}$

$\vdash^{-1} e \text{ } op \text{ } e \text{ } + \text{ } q \text{ } + \text{id} * \text{id}$

$\vdash^{-1} e \text{ } op \text{ } \underline{e} \text{ } q \text{ id} * \text{id}$

$\vdash^{-1} e \text{ } op \text{ } id \text{ } q \text{ id} * \text{id}$

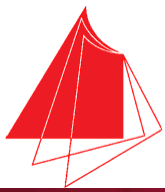
$\vdash^{-1} e \text{ } \underline{op} \text{ } q * \text{id}$

$\vdash^{-1} e \text{ } * \text{ } q * \text{id}$

$\vdash^{-1} \underline{e} \text{ } q \text{ id}$

$\vdash^{-1} id \text{ } q \text{ id}$

$\vdash^{-1} q$



# Wie wird die Ableitung eindeutig?

Im Allgemeinen gar nicht. Es gibt jedoch eine Reihe von Grammatiken, die eine eindeutige Ableitung ermöglichen. Z. B. kann **die Auswahl der Produce-Schritte über eine Vorausschau geklärt werden.**

**Auf was muss man bei einer Vorausschau achten?**



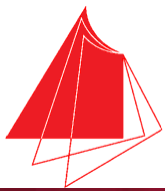
# Beispiel

- $S ::= A B \mid C ? S$
- $A ::= B C$
- $B ::= + A B \mid -$
- $C ::= \$ C D \mid \varepsilon$
- $D ::= ( S ) \mid \text{id}$

(S ist das Startsymbol)

**Frage:**

Gibt es eine Ableitung für „- -“ und „?- -“ oder „\$x- -“?



# Wie wird die Ableitung eindeutig?

Man betrachtet alle terminalen Worte, die aus einem Wort ( $w$ ) entstehen.

$$\text{First}(w) = \{y \in T^* \mid w \vdash y\}$$

Man betrachtet alle terminalen Worte, die nach einem Nichtterminal ( $A$ ) kommen können.

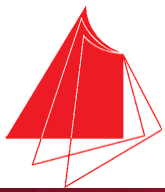
$$\text{Follow}(A) = \{y \mid y \in T^* \mid \exists z \in (N \cup T)^* \text{ mit } Z \vdash zAy\} \quad A \in N$$

Man prüft, ob für alternative Regeln ( $A ::= v$  und  $A ::= w$ ) eine eindeutige Entscheidung getroffen werden kann.

$$\text{First}(v, \text{Follow}(A)) \cap \text{First}(w, \text{Follow}(A)) = \{\}$$

$$\text{First}(r, \text{Follow}(A)) = \{y \in T^* \mid \exists z \in \text{Follow}(A) \text{ und } y \in \text{First}(rz)\} \quad A \in N, r \in (N \cup T)^*$$

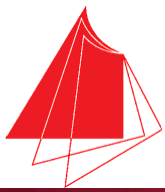
**Ideal ist, wenn man nicht die ganzen Wörter vergleichen muss, sondern nur ein Anfangsstück.**



# Die Konstruktion von $\text{First}_1(x)$

1. Für alle  $x \in T$  ist  $\text{First}_1(x) = \{x\}$
2. ist  $X ::= \varepsilon \in P$  dann ist auch  $\# \in \text{First}_1(X)$
3. ist  $X ::= y_1 \dots y_n \in P$  dann ist  $\text{First}_1(y_1 \dots y_n) \subseteq \text{First}_1(X)$
4. für jedes Terminal  $a$  gilt :  $a \in \text{First}_1(y_1 \dots y_n)$   
gdw.  
 $a \in \text{First}_1(y_1)$  oder  $a \in \text{First}_1(y_i)$  für  $i$  ( $1 < i \leq n$ ) und  $\# \in \text{First}_1(y_1)$  bis  $\text{First}_1(y_{i-1})$
5.  $\# \in \text{First}_1(y_1 \dots y_n)$  gdw.  $\# \in \text{First}_1(y_1)$  bis  $\text{First}_1(y_n)$

**$\text{First}_1(w)$  für  $w \in (N \cup T)^*$ , ist die kleinste Menge, die 1-5 erfüllt.**



# Die Konstruktion von $\text{Follow}_1(A)$

1.  $\# \in \text{Follow}_1(Z)$
2. ist  $B ::= vAw \in P$  dann ist  $\text{First}_1(w) \setminus \{\#\} \subseteq \text{Follow}_1(A)$
3. ist  $B ::= vA$  oder  $B ::= vAw \in P$  und  $\# \in \text{First}_1(w)$ ,  
dann ist  $\text{Follow}_1(B) \subseteq \text{Follow}_1(A)$

**$\text{Follow}_1(A)$  für  $A \in N$ , ist die kleinste Menge, die 1-3 erfüllt.**



# Beispiel

- $S ::= A B \quad | \quad C ? S$
- $A ::= B C$
- $B ::= + A B \quad | \quad -$
- $C ::= \$ C D \quad | \quad \varepsilon$
- $D ::= ( S ) \quad | \quad id$

$First_1(D) = \{ (, id \}$

$First_1(C) = \{ \$, \# \}$

$First_1(B) = \{ +, - \}$

$First_1(A) = First_1(B) = \{ +, - \}$

$First_1(S) = First_1(A) \cup First_1(C) \setminus \{ \# \} \cup \{ ? \} = \{ +, -, \$, ? \}$

$Follow_1(S) = \{ ), \# \}$

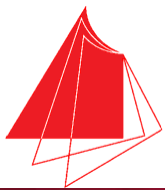
$Follow_1(A) = First_1(B) = \{ +, - \}$

$Follow_1(B) = Follow_1(S) \cup First_1(C) \setminus \{ \# \} \cup Follow_1(A) = \{ ), \$ +, -, \# \}$

$Follow_1(C) = \{ ? \} \cup Follow_1(A) \cup First_1(D) = \{ ?, +, -, (, id \}$

$Follow_1(D) = Follow_1(C) = \{ (, id, ?, +, - \}$





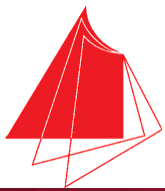
# Ein impliziter Keller

Statt des expliziten Kellers eines LL-Akzeptors kann man auch den impliziten Keller rekursiver Funktionen ausnutzen. Dieses Verfahren heißt „**rekursiver Abstieg**“, und bietet sich besonders für Parser an, die man von Hand programmiert.

**Für jedes Nichtterminal wird eine separate Funktion geschrieben:**

seien  $A ::= nBm$  und  $A ::= mmC \in P$  und die Grammatik SLL(1)

```
void A() {  
  if (token  $\in$  First1( $nBm$ , Follow1(A))) { // if (token == 'n')  
    next_token();  
    B();  
    if (token == 'm') { next_token(); } else { error(); }  
  }  
  else if (token  $\in$  First1( $mmC$ , Follow1(A))) { // if (token == 'm')  
    next_token();  
    if (token == 'm') { next_token(); } else { error(); }  
    C();  
  }  
  else error();  
}
```



# Aufgaben

a) Schreiben Sie einen Parser für folgende Grammatik mit Startsymbol PROG.

PROG ::= DECLS STATEMENTS

DECLS ::= DECL ; DECLS |  $\varepsilon$

DECL ::= **int** ARRAY *identifizier*

ARRAY ::= [ *integer* ] |  $\varepsilon$

STATEMENTS ::= STATEMENT ; STATEMENTS |  $\varepsilon$

STATEMENT ::= *identifizier* INDEX = EXP | **print**( EXP ) | **read** ( *identifizier* INDEX ) | { STATEMENTS } |  
**if** ( EXP ) STATEMENT **else** STATEMENT |  
**while** ( EXP ) STATEMENT

EXP ::= EXP2 OP\_EXP

EXP2 ::= ( EXP ) | *identifizier* INDEX | *integer* | - EXP2 | ! EXP2

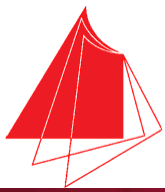
INDEX ::= [ EXP ] |  $\varepsilon$

OP\_EXP ::= OP EXP |  $\varepsilon$

OP ::= + | - | \* | / | < | > | == | != | &

## Hinweis:

- normale Terminale (Schlüsselwörter) sind klein, **fett** und *rot*
- kleine, **fette**, *kursive* und blaue Terminale stehen für Konstanten bzw. Bezeichner (3, 3.14, x,...).
- Nichtterminale sind groß und *KURSIV* gedruckt



# Aufgaben

b) Verwenden Sie hierzu den Scanner aus Teil 1 und ergänzen Sie die fehlenden Terminalsymbole.

**digit** ::= „0“ | „1“ | „2“ | „3“ | „4“ | „5“ | „6“ | „7“ | „8“ | „9“

**letter** ::= „A“ | „B“ | „C“ | ... | „Z“ | „a“ | „b“ | ... | „z“ |

**sign+** ::= „+“

...

**...** ::= „-“ „\*“ „/“ „<“ „>“ „=“ „==“ „!=“ „!“ „&“ „;“ „:“ „(“ „)“ „{“ „}“ „[“

...

**sign]** ::= „]“

**integer** ::= digit {digit}

**identifier** ::= letter {letter | digit}

**print** ::= „print“

**read** ::= „read“

**if** ::= „if“ | „IF“

**else** ::= „else“ | „ELSE“

**while** ::= „while“ | „WHILE“

**int** ::= „int“

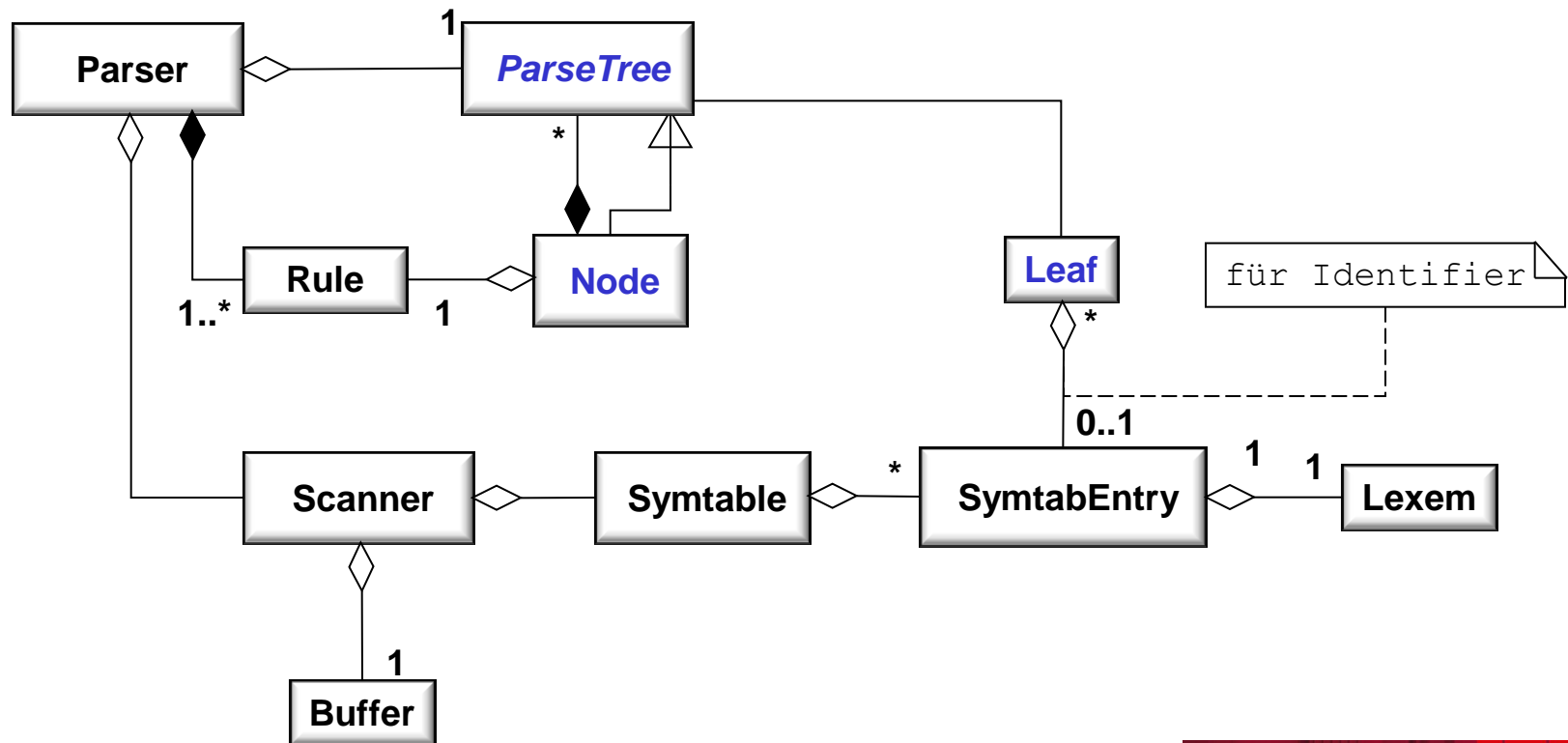


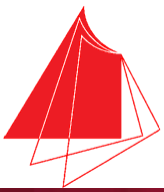
# Aufgaben

## c) Erstellen Sie einen Strukturbaum (Parse-Baum).

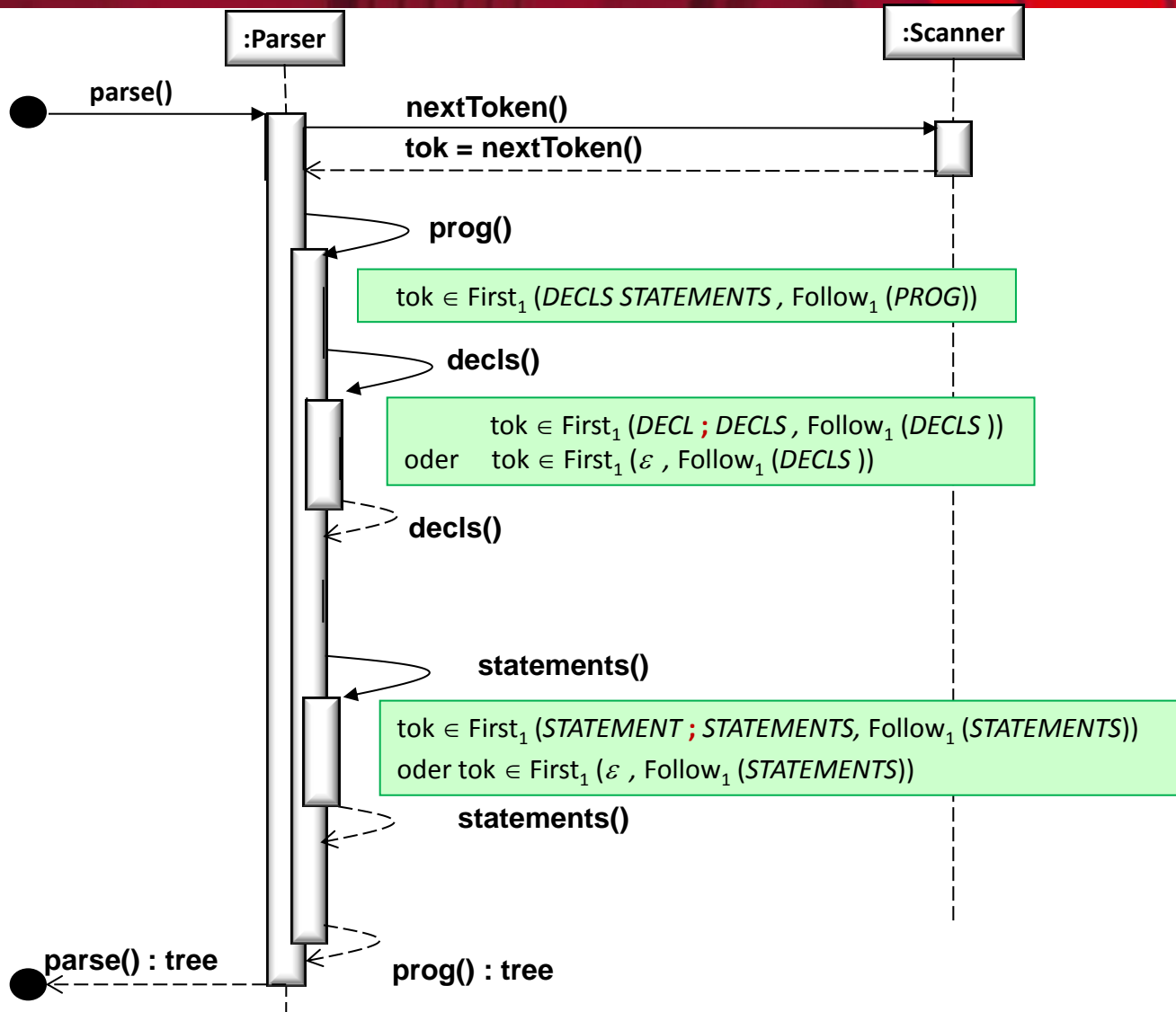
Erweitern Sie hierzu die Funktionen des Parsers, so dass sie nicht nur die Syntax überprüfen, sondern auch den Baum aufbauen.

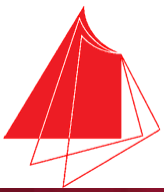
**(Bem.: Für jede erkannte Regel entsteht ein neuer Teilbaum)**



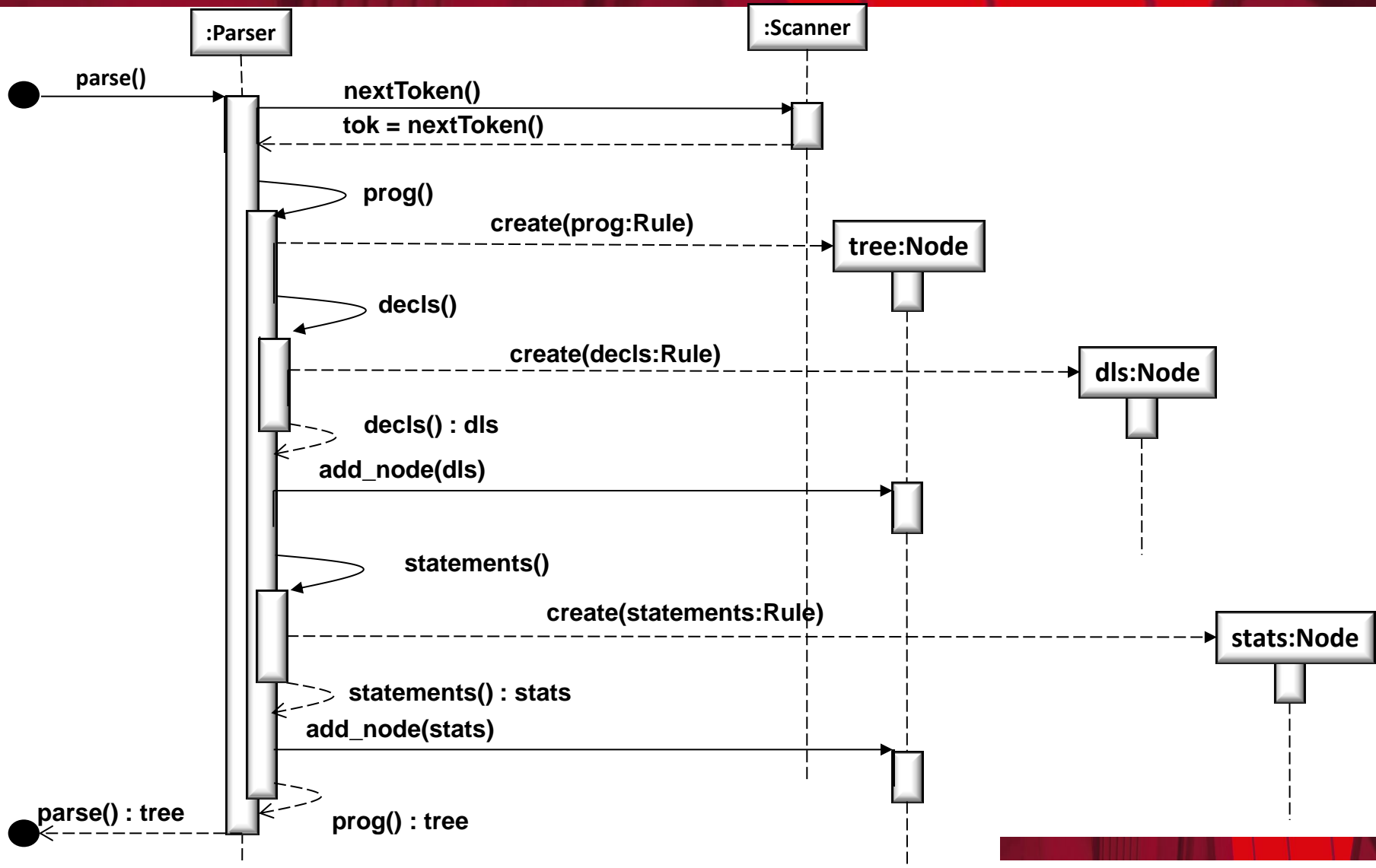


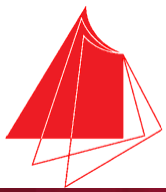
# Ein Szenario





# Ein Szenario





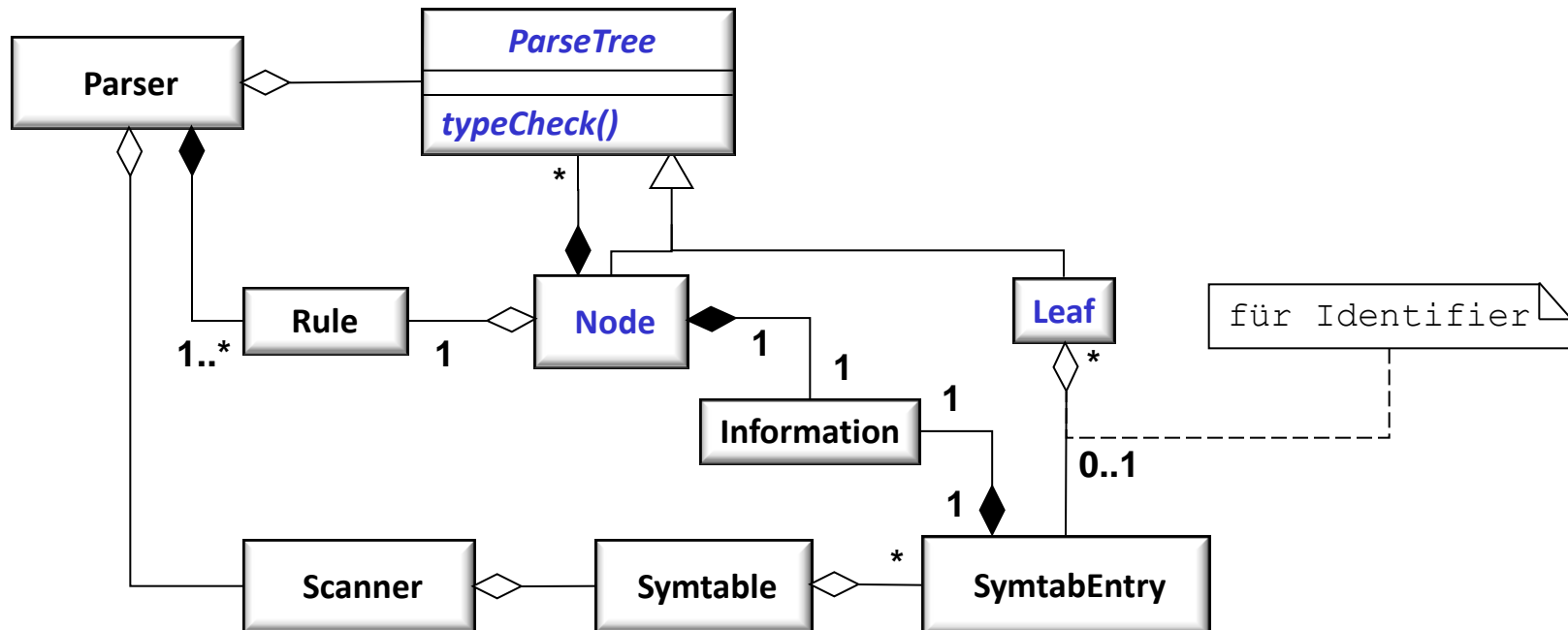
# Aufgaben

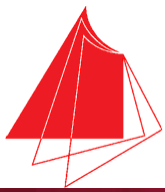
## d) Evaluieren Sie den Parse-Baum

Ermitteln Sie hierzu zu jedem Knoten den entsprechenden Typ (gemäß Vorlage) und prüfen Sie, ob die Typen der Unterbäume zusammenpassen.

Speichern Sie die Typ-Information im Knoten und für **Identifizier als Information in der Symboltabelle**.

Hinweis: Erweitern Sie hierzu die Klasse *ParseTree* um eine Operation **typeCheck**.





# Typisierung und Typ-Check

```
typeCheck (PROG ::= DECLS STATEMENTS){  
  typeCheck(DECLS); typeCheck(STATEMENTS);  
  this.type = noType;}
```

## Typ-Informationen:

intType , intArrayType, arrayType,  
noType, errorType,  
opPlus, opMinus, opMult, opDiv, opLess,  
opGreater, opEqual, opUnequal, opAnd

```
typeCheck (DECLS ::= DECL ; DECLS ){ typeCheck(DECL ); typeCheck(DECLS);  this.type = noType;}
```

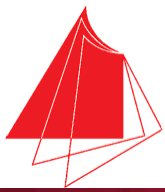
```
typeCheck (DECLS ::= ε){this.type = noType;}
```

```
typeCheck(DECL::= int ARRAY identifier){typeCheck(ARRAY);  
  if (getType(identifier)!=noType ) { error(„identifier already defined“);this.type=errorType;}  
  else if (ARRAY.type==errorType){ this.type = errorType;}  
  else { this.type = noType;  
    if (ARRAY.type == arrayType) store(identifier, intArrayType); // Typ-Information speichern  
    else store(identifier, intType);} }// Typ-Information speichern
```

```
typeCheck(ARRAY::=[integer]){  
  if(integer.value>0) this.type= arrayType ;  
  else {error(„no valid dimension“); this.type= errorType;}}
```

```
typeCheck(ARRAY ::= ε){ this.type = noType ;}
```





# Typisierung und Typ-Check

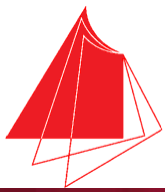
```
typeCheck (STATEMENTS ::= STATEMENT ; STATEMENTS){
    typeCheck(STATEMENT ); typeCheck(STATEMENTS); this.type = noType;}

typeCheck (STATEMENTS ::= ε){this.type = noType;}

typeCheck (STATEMENT ::= identifier INDEX = EXP ){ typeCheck(EXP); typeCheck(INDEX);
    if (getType(identifier) == noType) {error(„identifier not defined“); this.type = errorType; }
    else if ( EXP.type == intType && (
        ( getType(identifier) == intType && INDEX.type == noType)
        || ( getType(identifier) == intArrayType && INDEX.type == arrayType)) )
        this.type = noType;
    else { error(„incompatible types“); this.type = errorType; }}

typeCheck (STATEMENT ::= print( EXP ) ){ typeCheck(EXP); this.type = noType; }

typeCheck (STATEMENT ::= read( identifier INDEX ) ){typeCheck(INDEX);
    if (getType(identifier) == noType){error(„identifier not defined“); this.type = errorType; }
    else if ( ( (getType(identifier) == intType ) && INDEX.type == noType)
        || ( (getType(identifier) == intArrayType) &&INDEX.type==arrayType))
        this.type = noType;
    else { error(„incompatible types“); this.type = errorType; }}
```



# Typisierung und Typ-Check

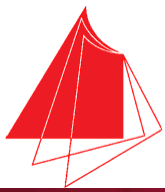
```
typeCheck (STATEMENT ::= { STATEMENTS } ){ typeCheck(STATEMENTS); this.type = noType; }
```

```
typeCheck (STATEMENT ::= if ( EXP ) STATEMENT else STATEMENT ){  
    typeCheck(EXP); typeCheck(STATEMENT ); typeCheck(STATEMENT );  
    if (EXP.type == errorType) this.type = errorType;  
    else this.type = noType; }
```

```
typeCheck (STATEMENT ::= while ( EXP ) STATEMENT){  
    typeCheck(EXP); typeCheck(STATEMENT );  
    if (EXP.type == errorType) this.type = errorType;  
    else this.type = noType; }
```

```
typeCheck(INDEX ::= [ EXP ] ) {  
    typeCheck(EXP);  
    if (EXP.type == errorType) this.type = errorType;  
    else this.type = arrayType; }
```

```
typeCheck(INDEX ::= ε) {this.type = noType ;}
```



# Typisierung und Typ-Check

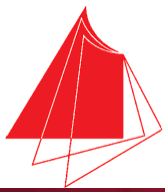
```
typeCheck (EXP ::= EXP2 OP_EXP ){  
  typeCheck(EXP2); typeCheck(OP_EXP);  
  if (OP_EXP.type == noType ) this.type = EXP2.type ;  
  else if (EXP2.type != OP_EXP.type) this.type = errorType;  
  else this.type = EXP2.type; }
```

```
typeCheck (EXP2 ::= ( EXP )){ typeCheck(EXP); this.type = EXP.type ; }
```

```
typeCheck (EXP2 ::= identifier INDEX ){  
  typeCheck(INDEX);  
  if (identifier.type == noType ){error(„identifier not defined“); this.type = errorType; }  
  else if ((getType(identifier) == intType && INDEX.type == noType)  
    this.type = getType(identifier);  
  else if ( getType(identifier) == intArrayType && INDEX.type == arrayType) this.type = intType;  
  else { error(„no primitive Type“); this.type = errorType; };
```

```
typeCheck (EXP2 ::= integer ){ this.type = intType ; }
```

```
typeCheck (EXP2 ::= - EXP2){ typeCheck(EXP2); this.type = EXP2.type ; }
```



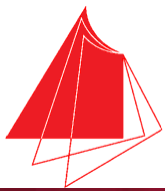
# Typisierung und Typ-Check

```
typeCheck (EXP2 ::= ! EXP2){  
  typeCheck(EXP2);  
  if (EXP2.type != intType){this.type = errorType; }  
  else this.type = intType; }
```

```
typeCheck (OP_EXP ::= OP EXP ){  
  typeCheck(OP); typeCheck(EXP); this.type = EXP.type ;}
```

```
typeCheck (OP_EXP ::=  $\epsilon$ ){this.type = noType;} 
```

```
typeCheck (OP ::= +){ this.type = opPlus; }  
typeCheck (OP ::= -){ this.type = opMinus; }  
typeCheck (OP ::= *){ this.type = opMult; }  
typeCheck (OP ::= /){ this.type = opDiv; }  
typeCheck (OP ::= <){ this.type = opLess; }  
typeCheck (OP ::= >){ this.type = opGreater; }  
typeCheck (OP ::= ==){ this.type = opEqual; }  
typeCheck (OP ::= !=){ this.type = opUnequal; }  
typeCheck (OP ::= &){ this.type = opAnd; }
```

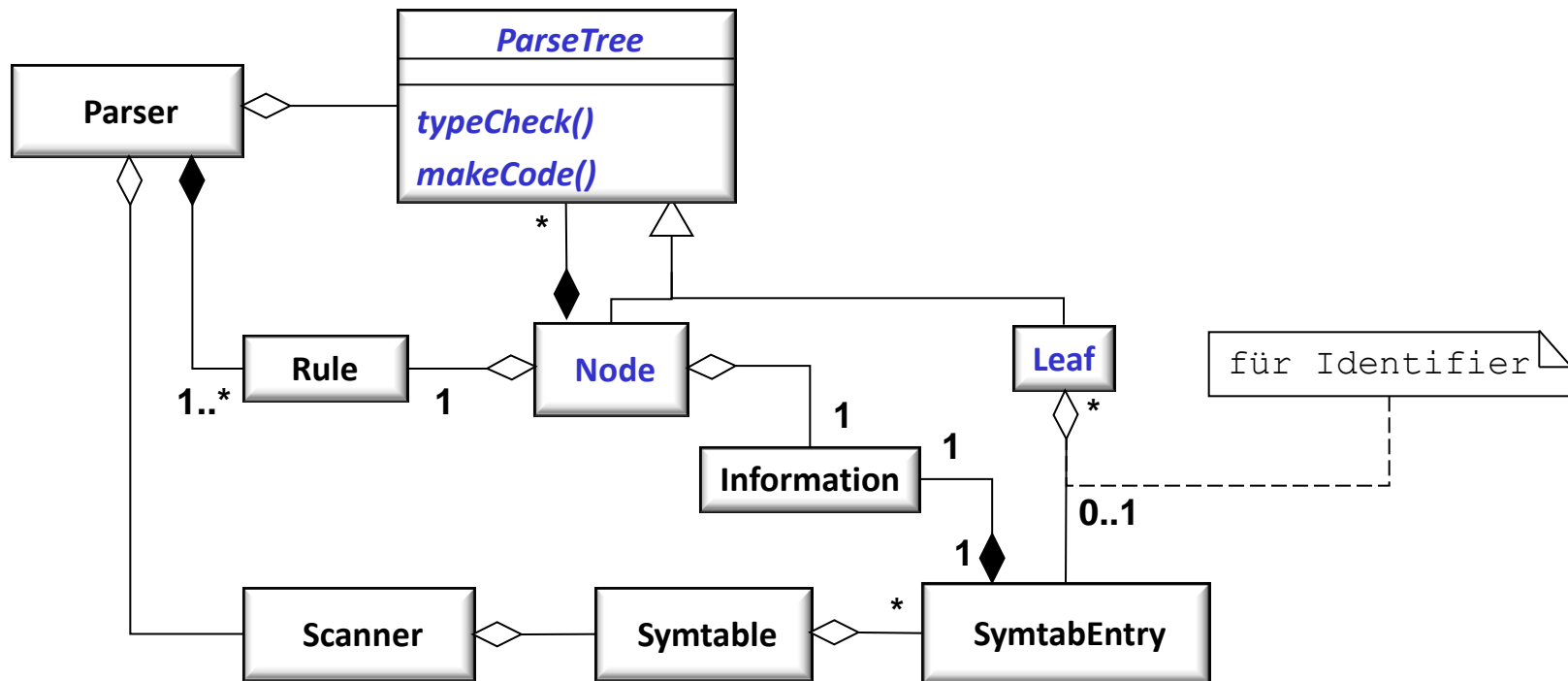


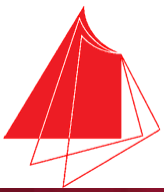
# Aufgaben:

## e) Erzeugen Sie Code

Bestimmen Sie hierzu zu jedem Knoten das entsprechende Code-Segment (gemäß Vorlage) und speichern Sie dieses in einer Code-Datei (xxx.code) ab.

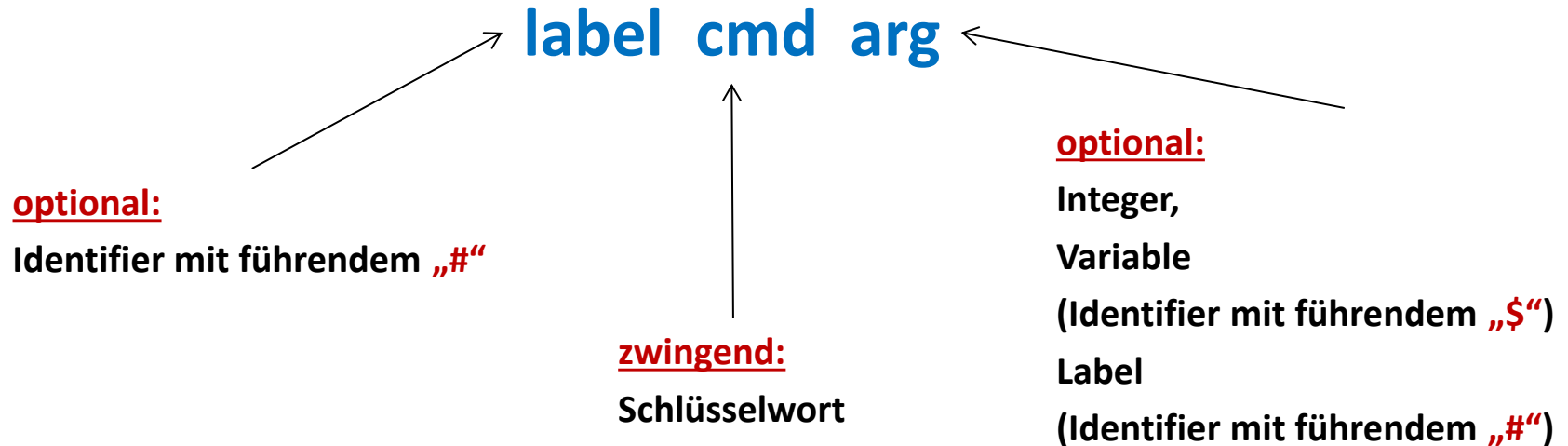
Hinweis: Erweitern Sie hierzu die Klasse *ParseTree* um eine Operation **makeCode**.



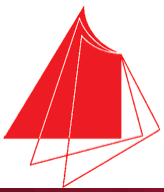


# Eine einfache Stack-Maschine

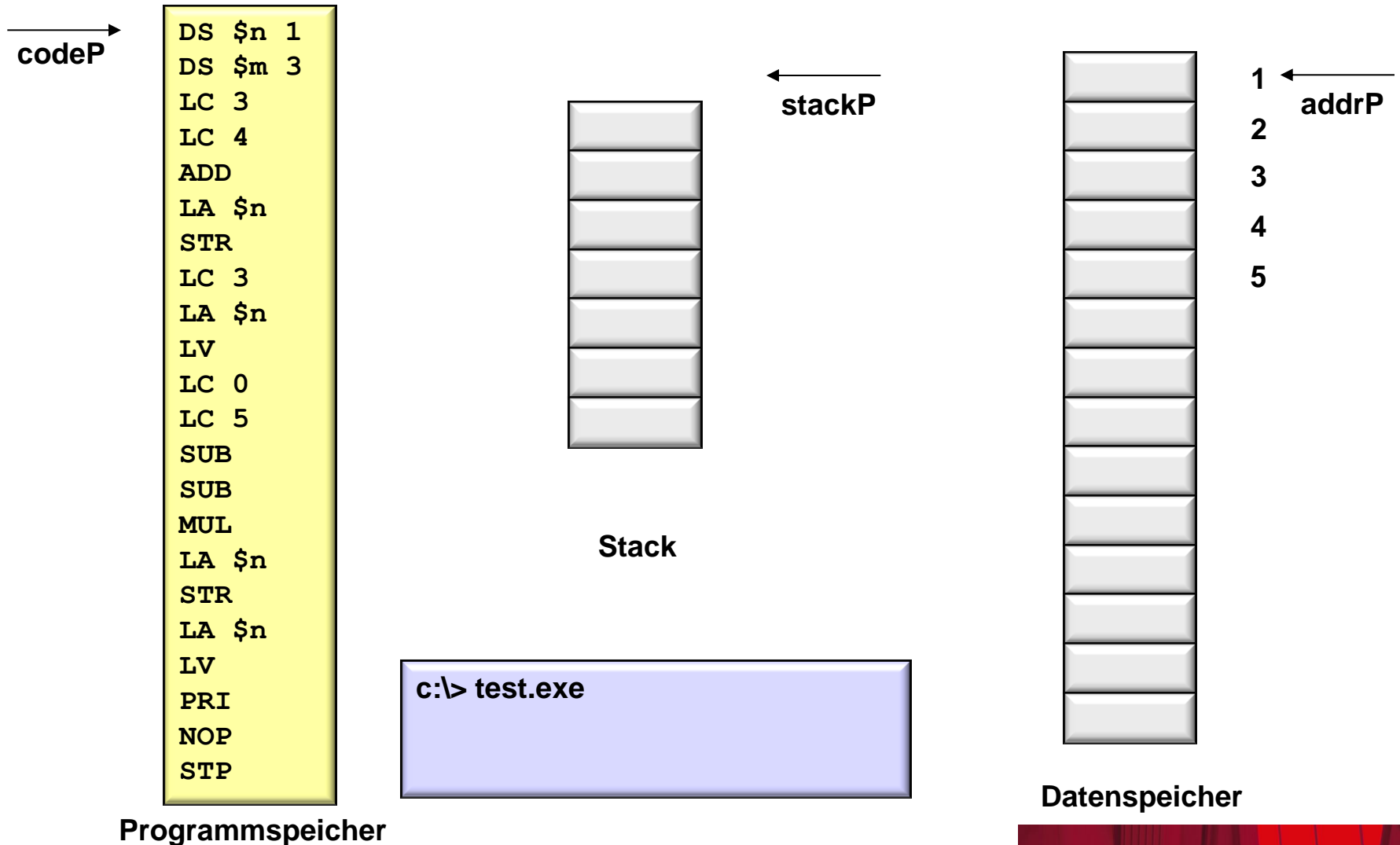
## Kommandos mit Label und Argument

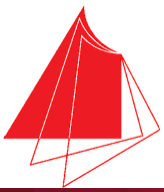


**Ausnahme: Kommando zur Speicherreservierung**  
**DS Variable Integer**

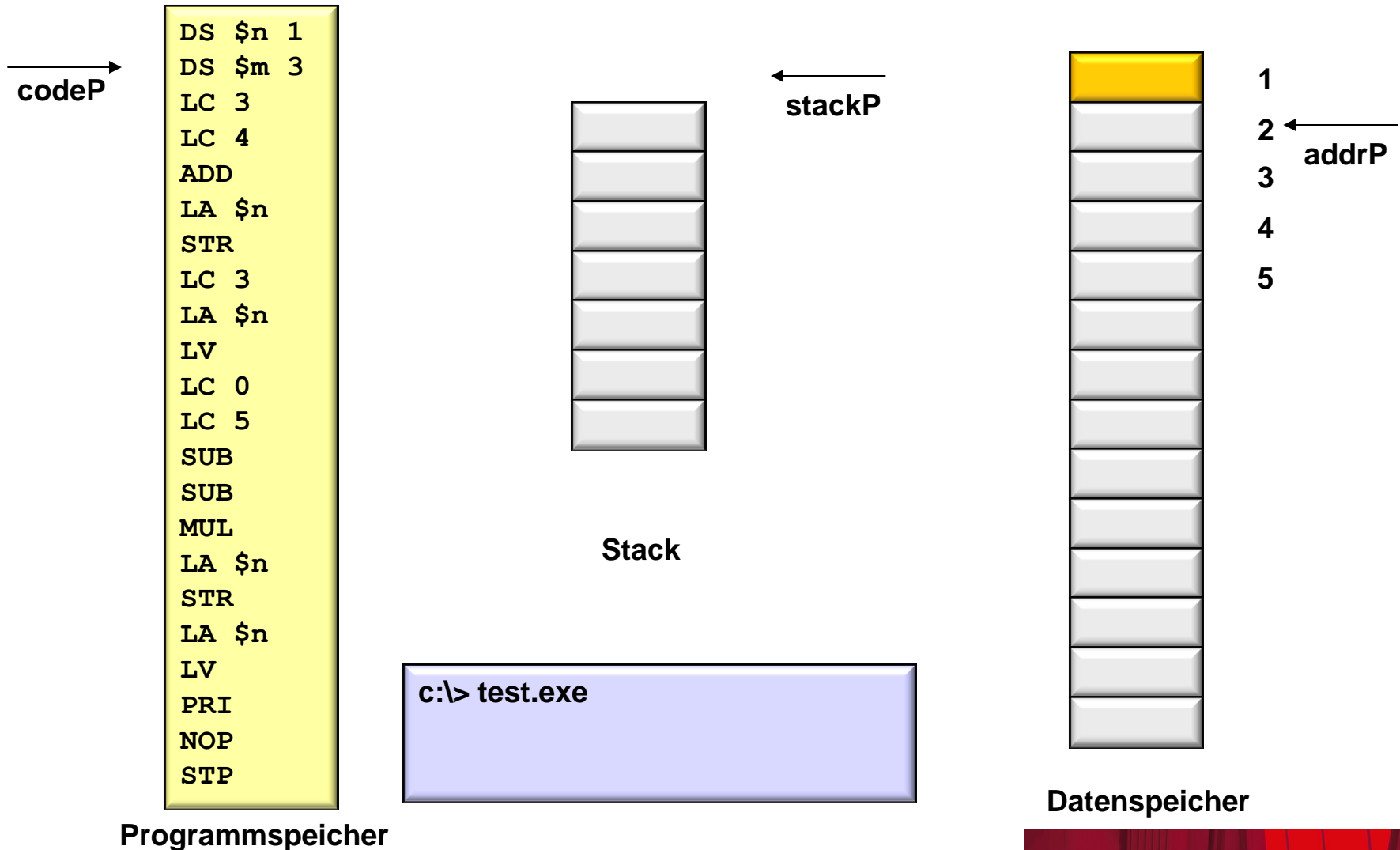


# Eine einfache Stack-Maschine

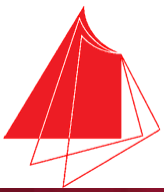




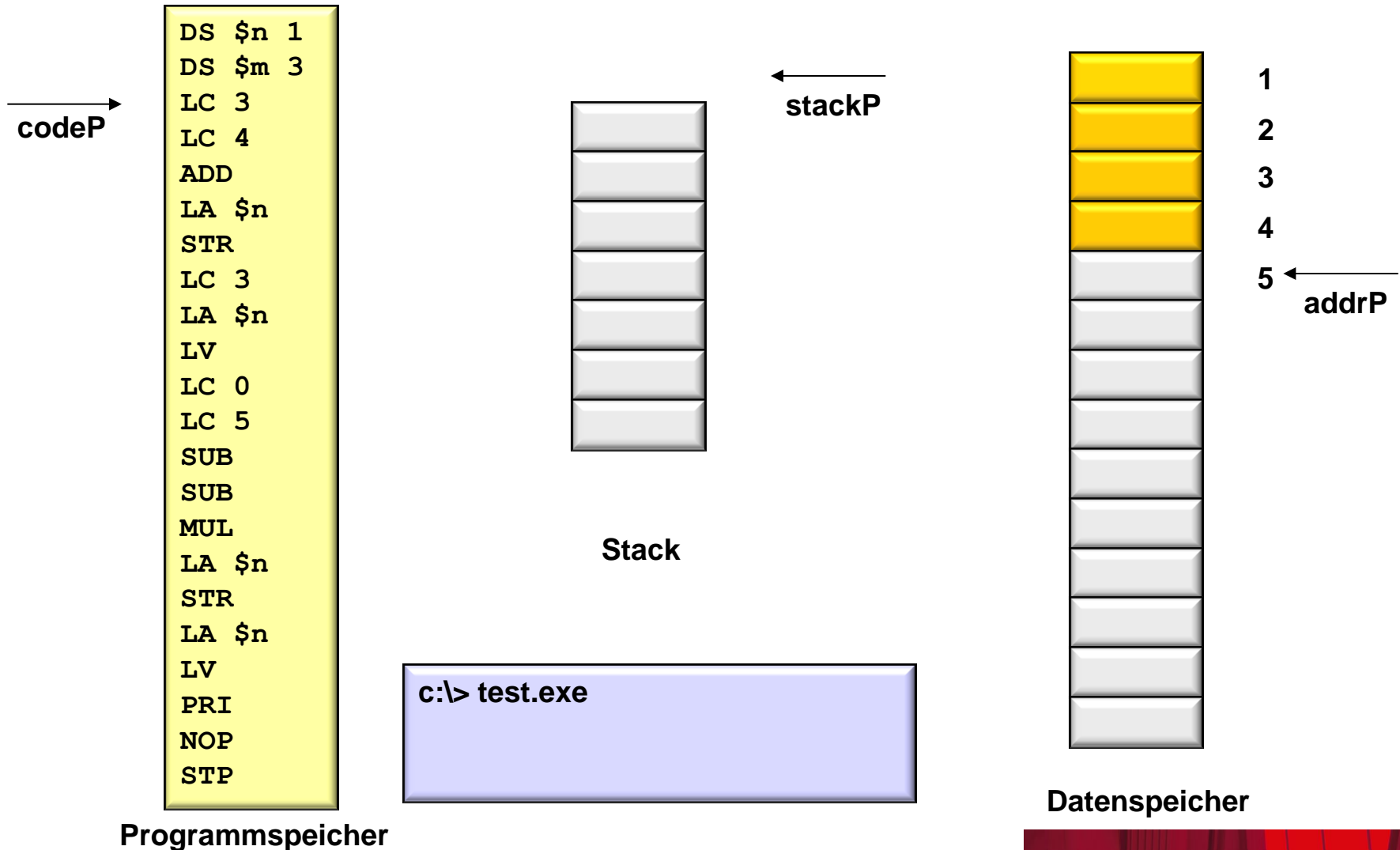
# Eine einfache Stack-Maschine

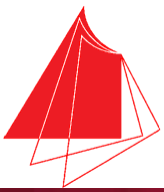




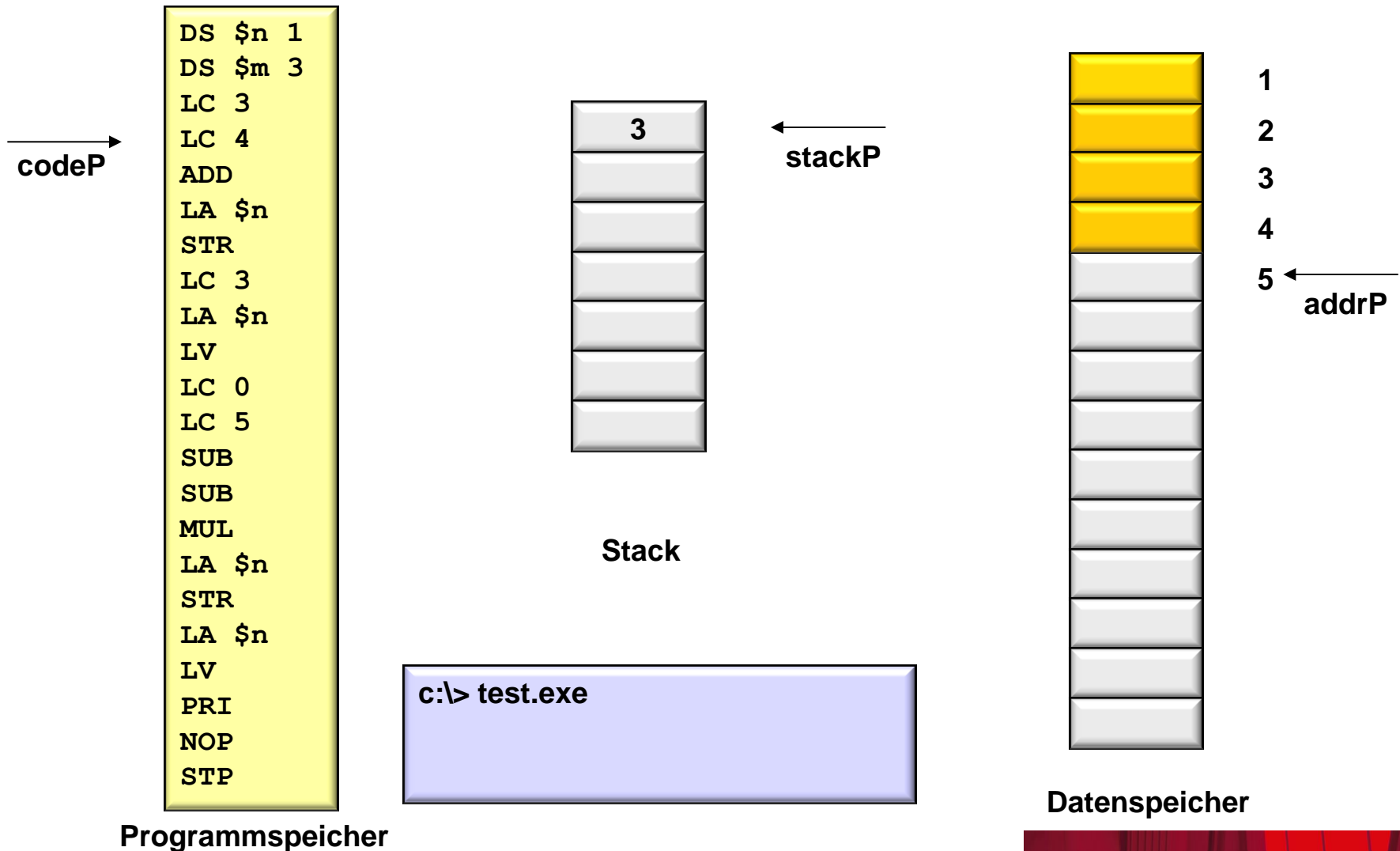


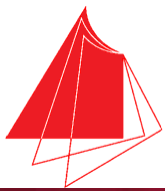
# Eine einfache Stack-Maschine



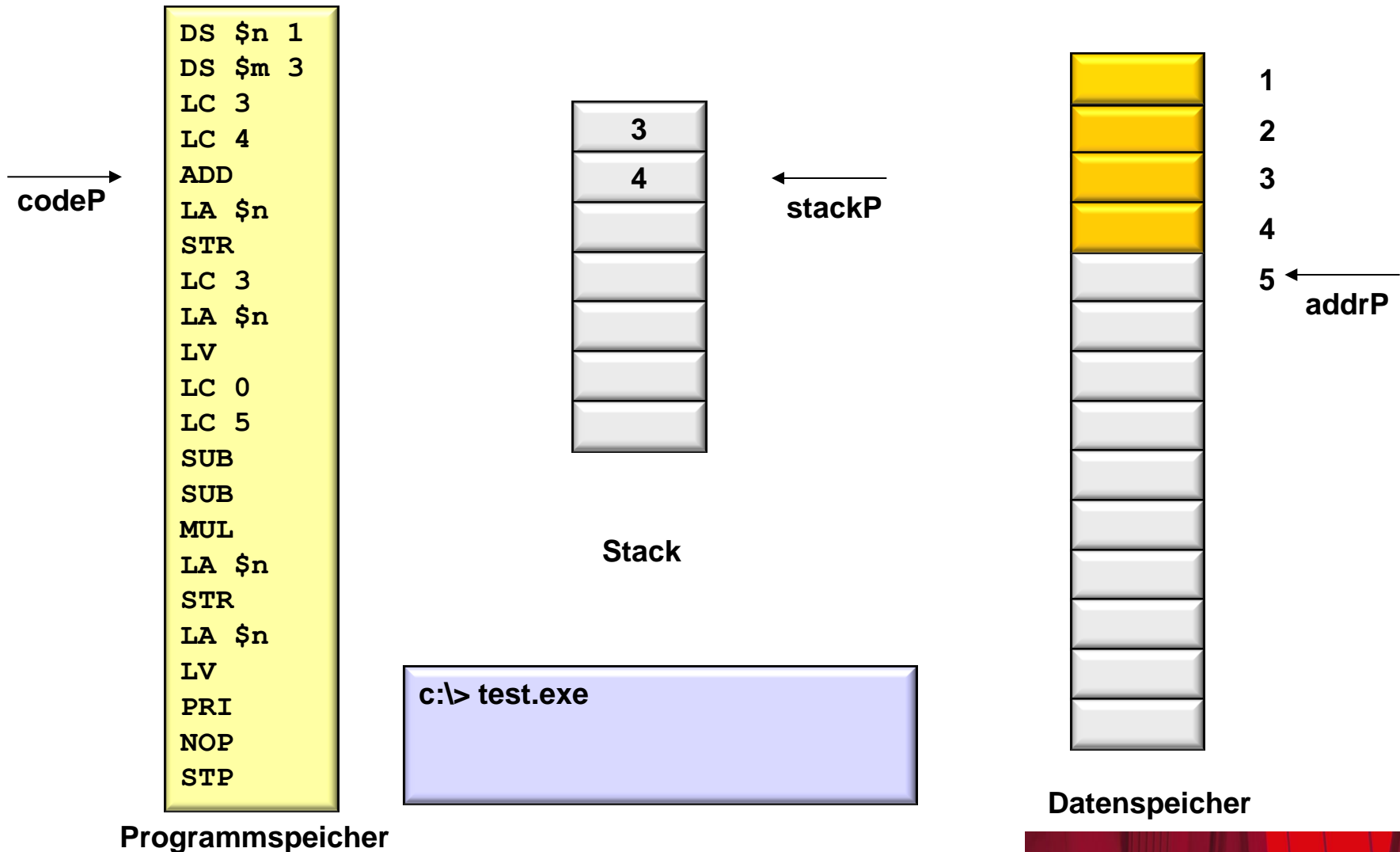


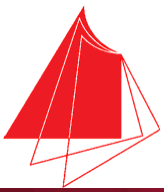
# Eine einfache Stack-Maschine



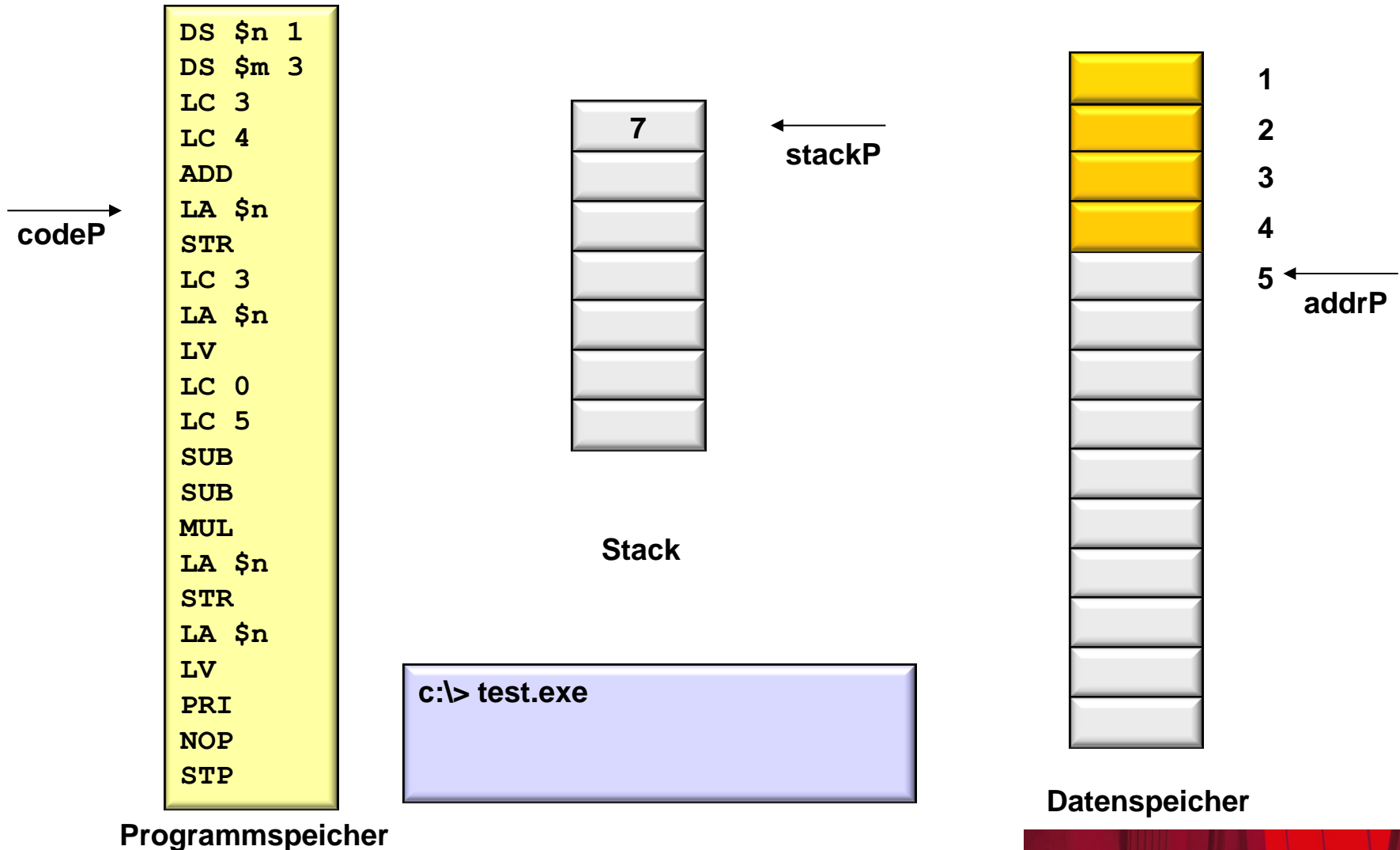


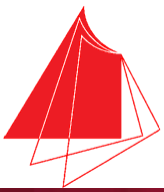
# Eine einfache Stack-Maschine



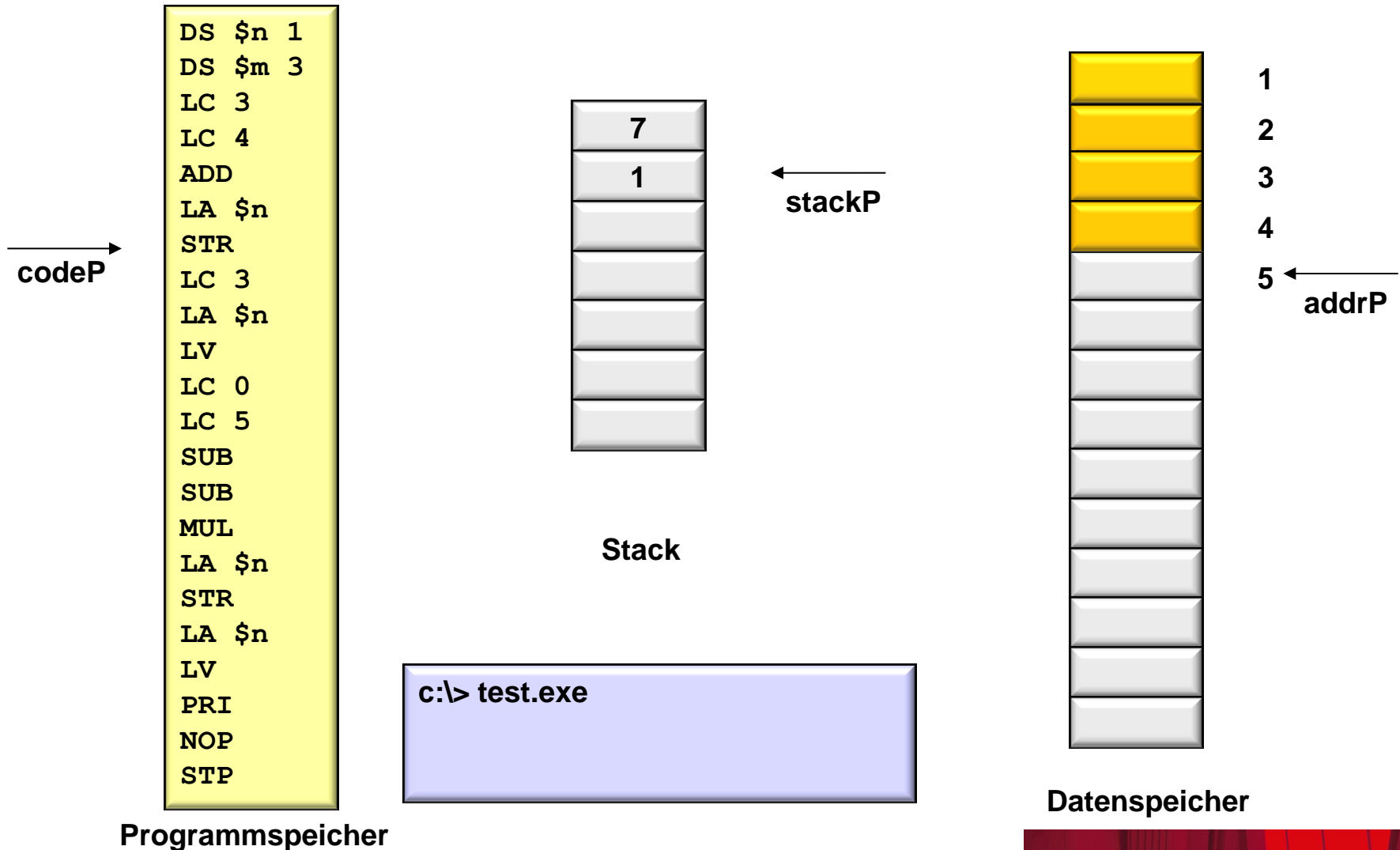


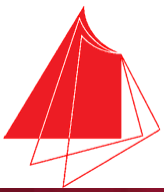
# Eine einfache Stack-Maschine



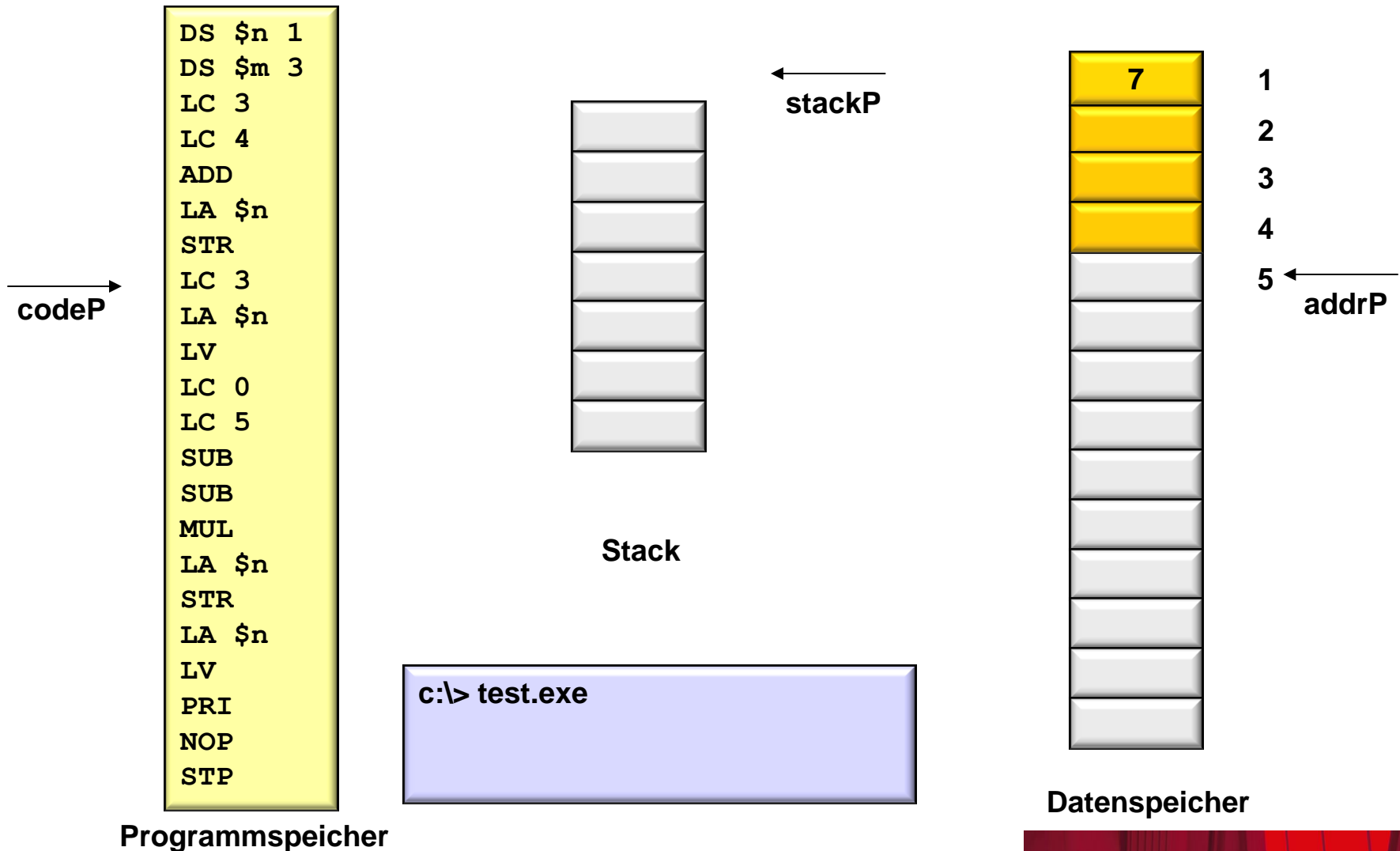


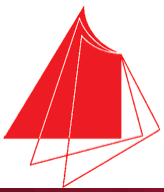
# Eine einfache Stack-Maschine



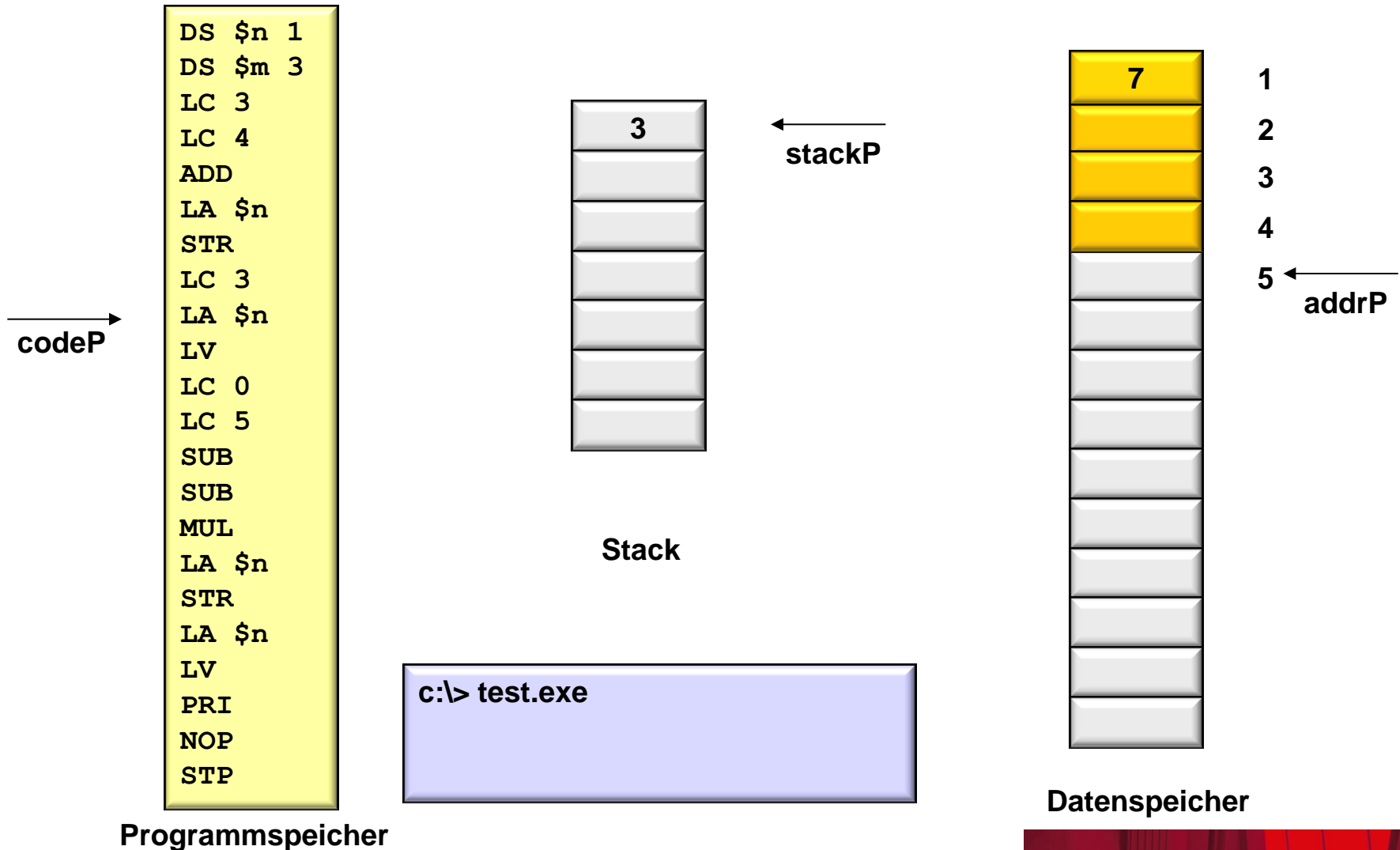


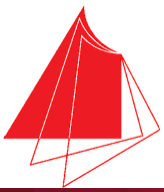
# Eine einfache Stack-Maschine



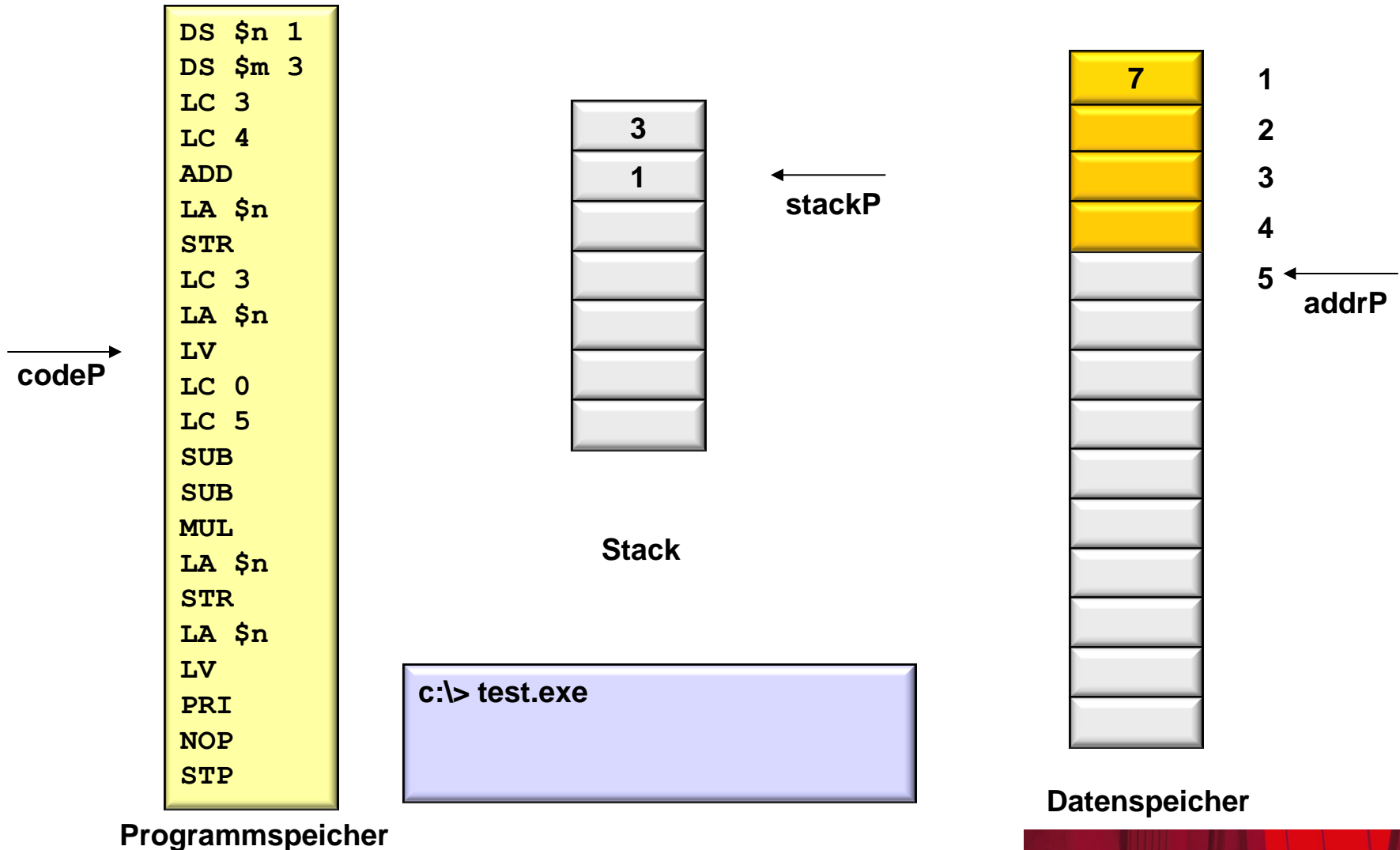


# Eine einfache Stack-Maschine

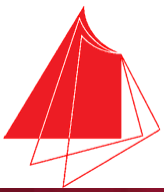




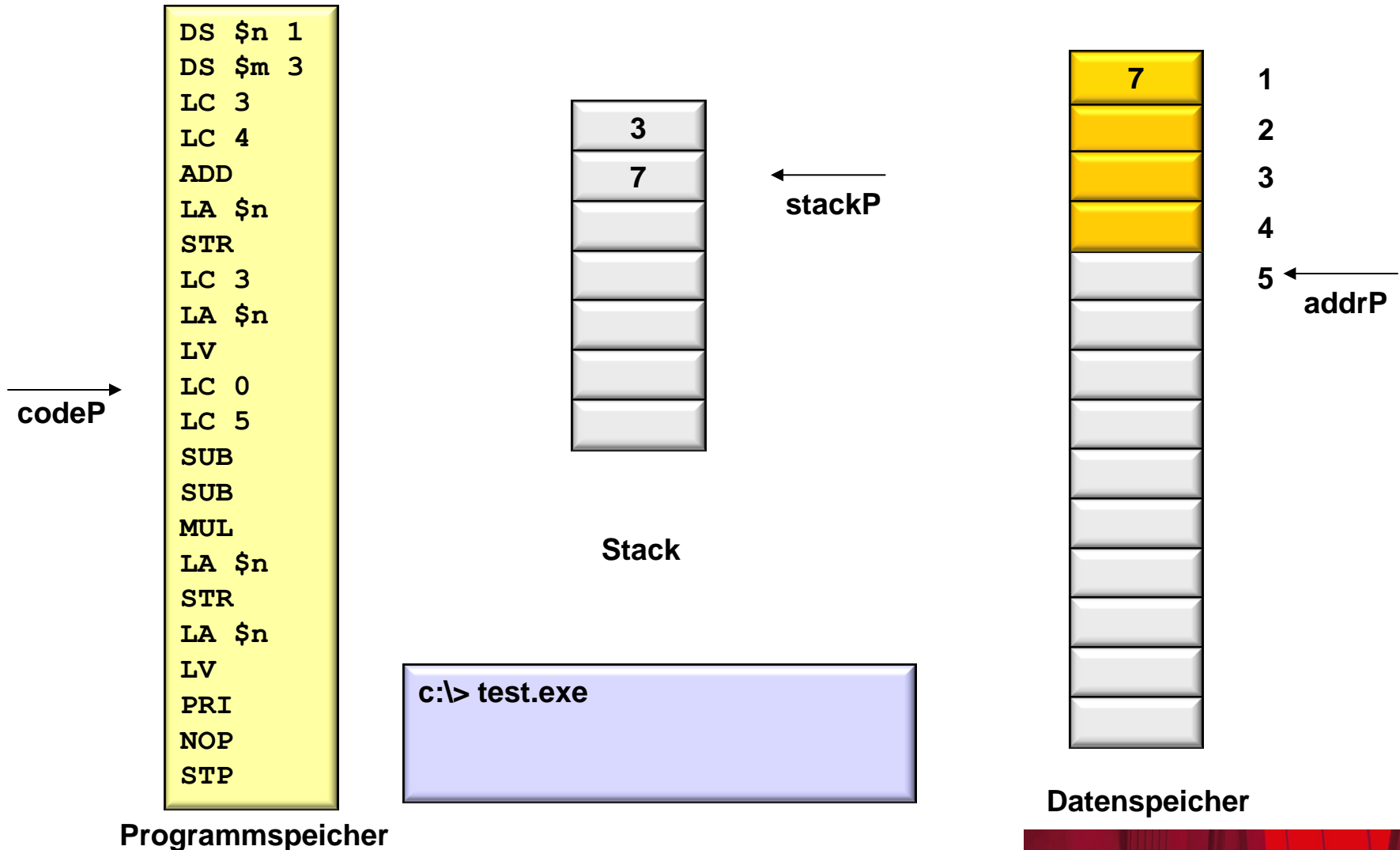
# Eine einfache Stack-Maschine

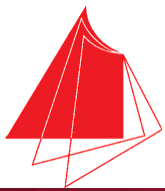




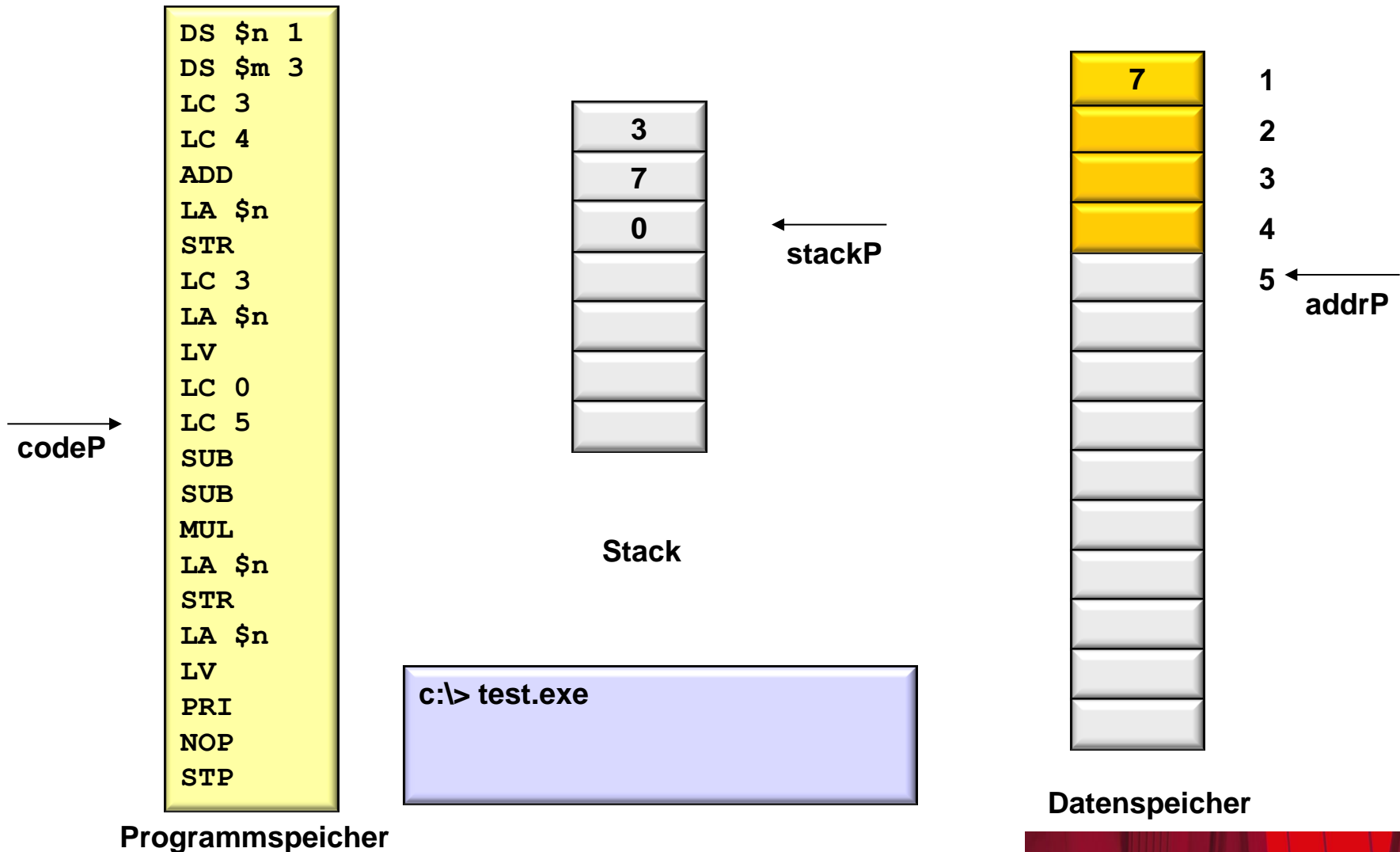


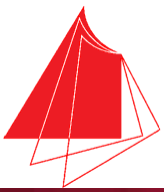
# Eine einfache Stack-Maschine



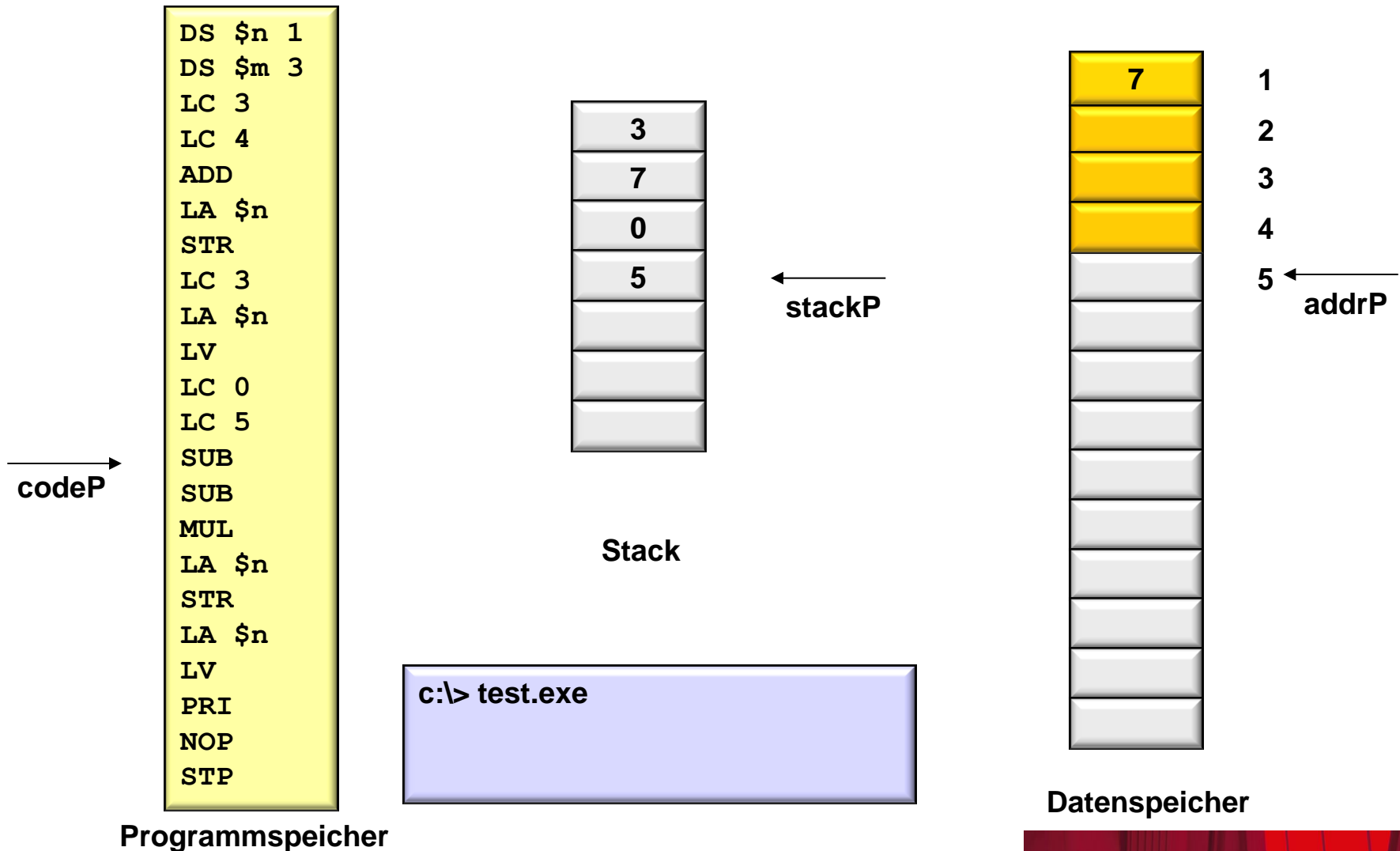


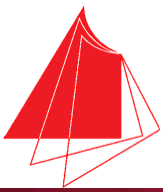
# Eine einfache Stack-Maschine



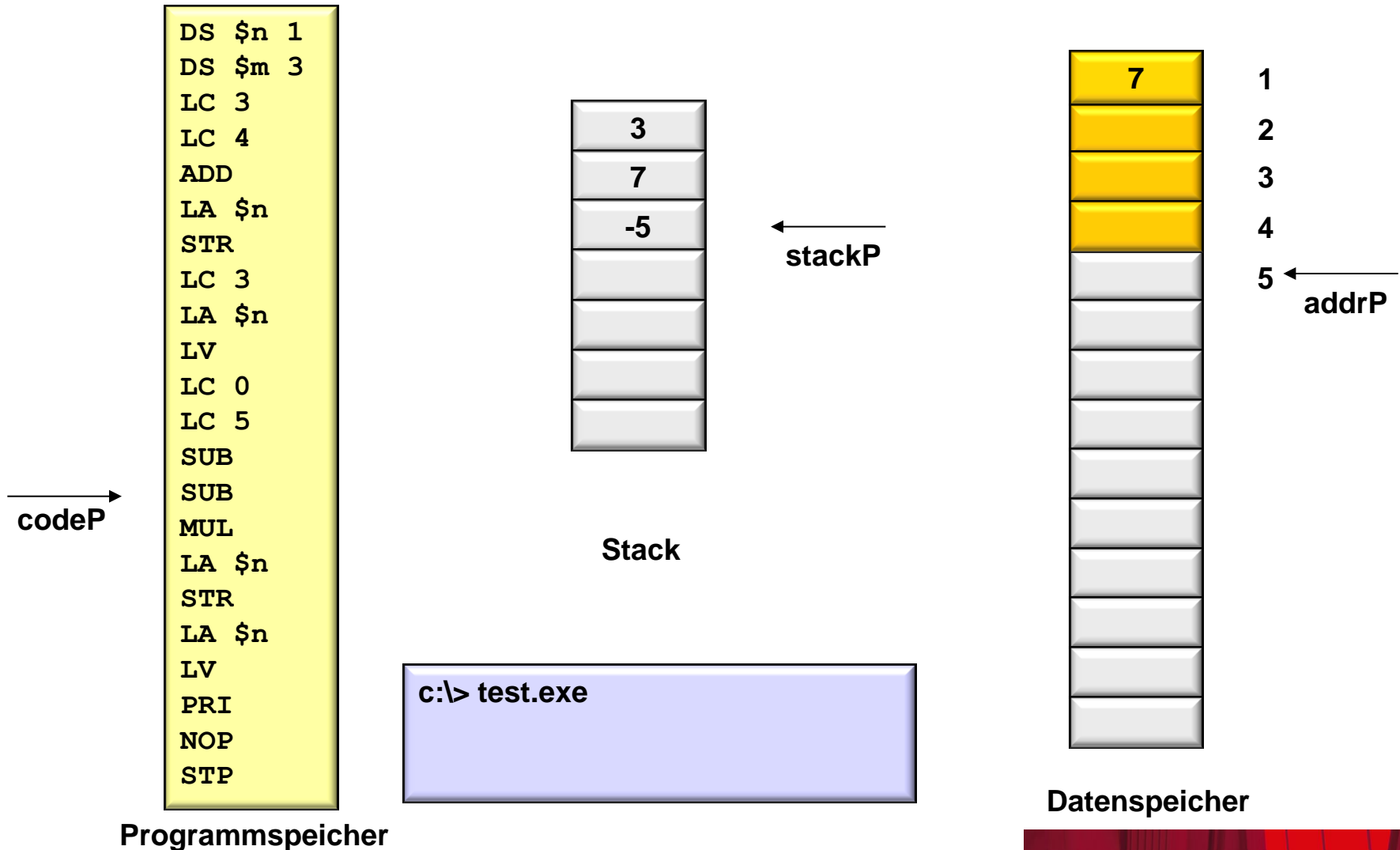


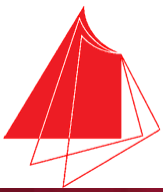
# Eine einfache Stack-Maschine



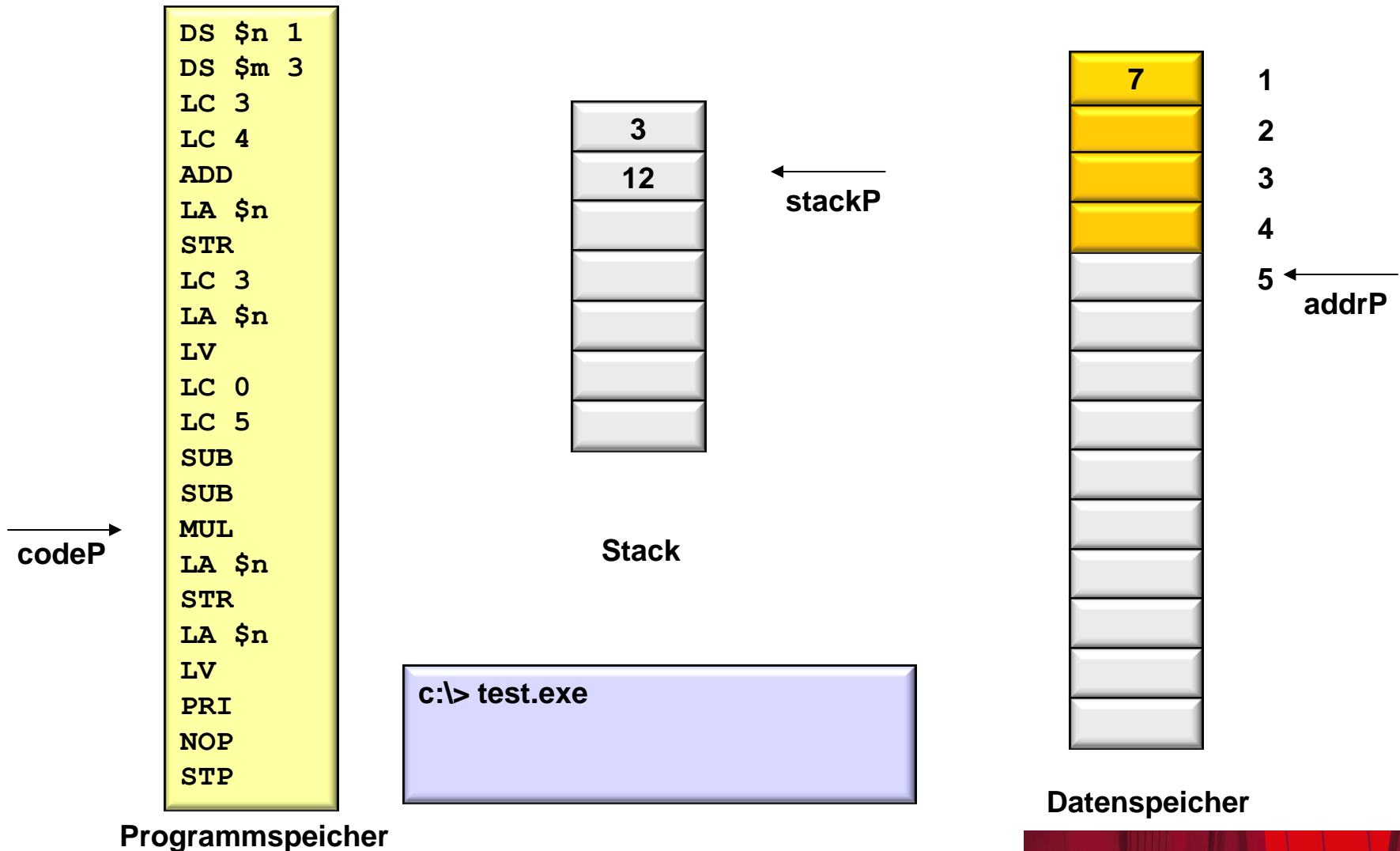


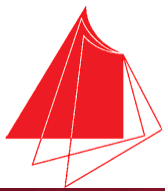
# Eine einfache Stack-Maschine



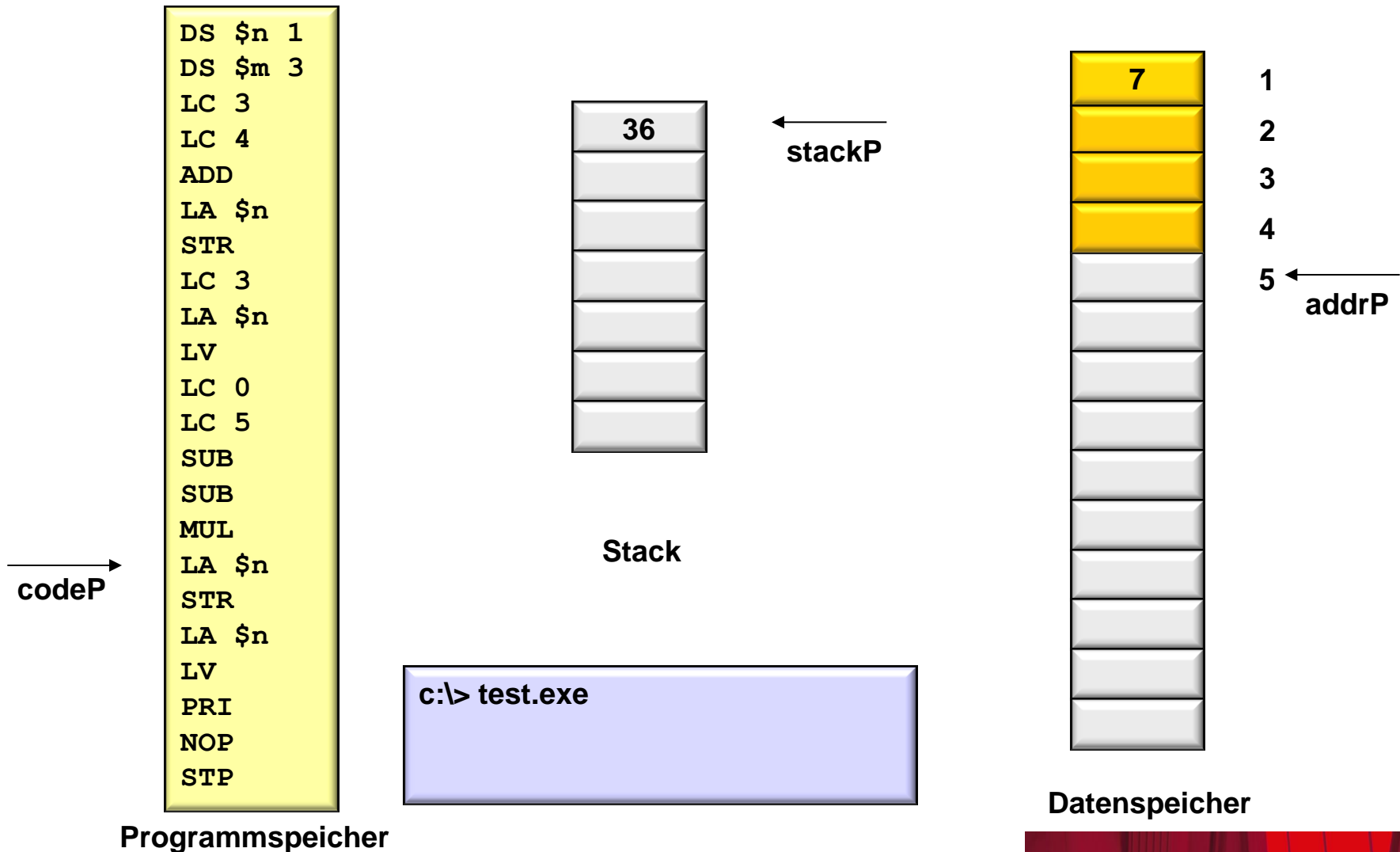


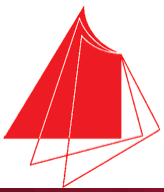
# Eine einfache Stack-Maschine



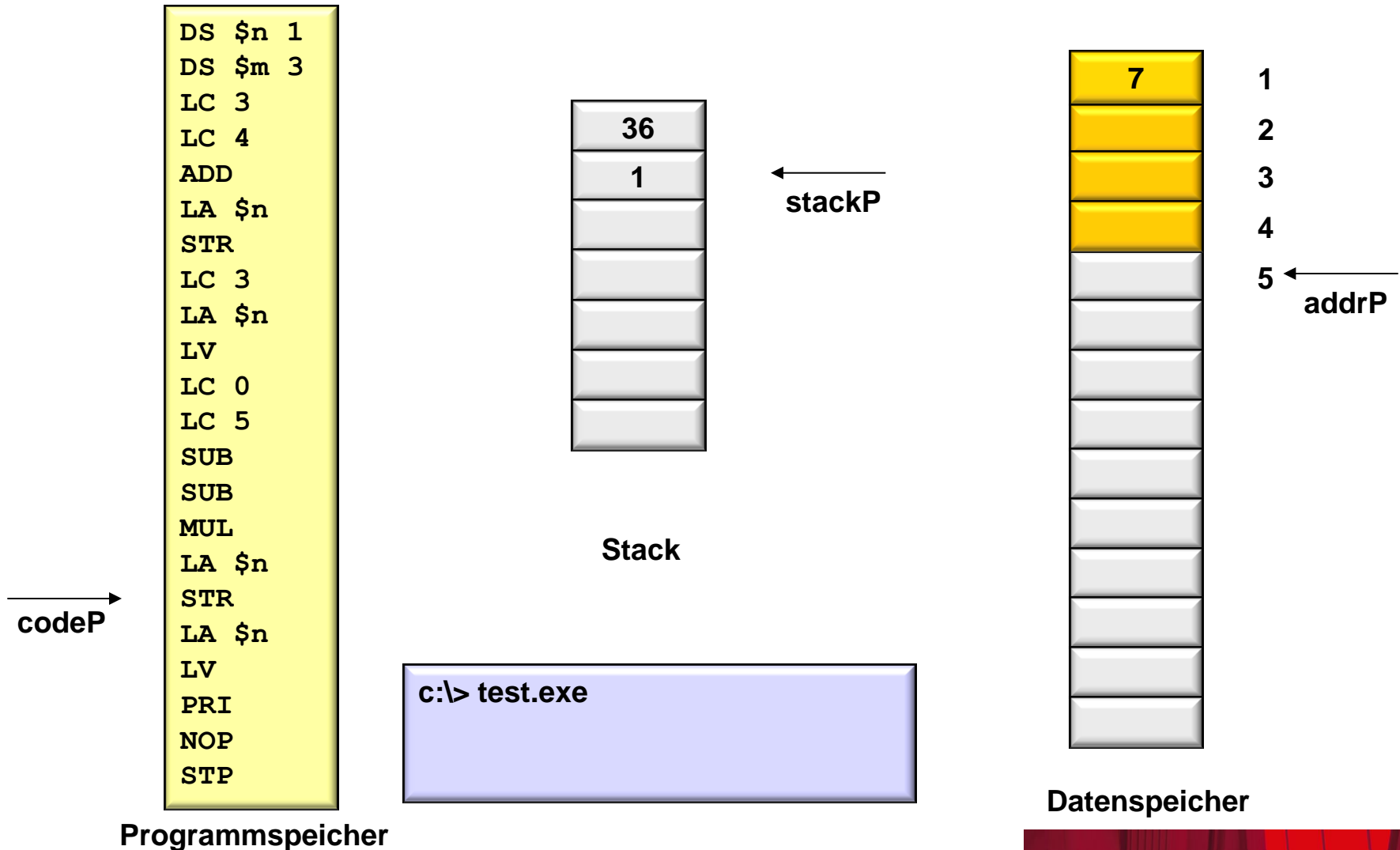


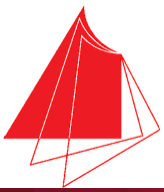
# Eine einfache Stack-Maschine



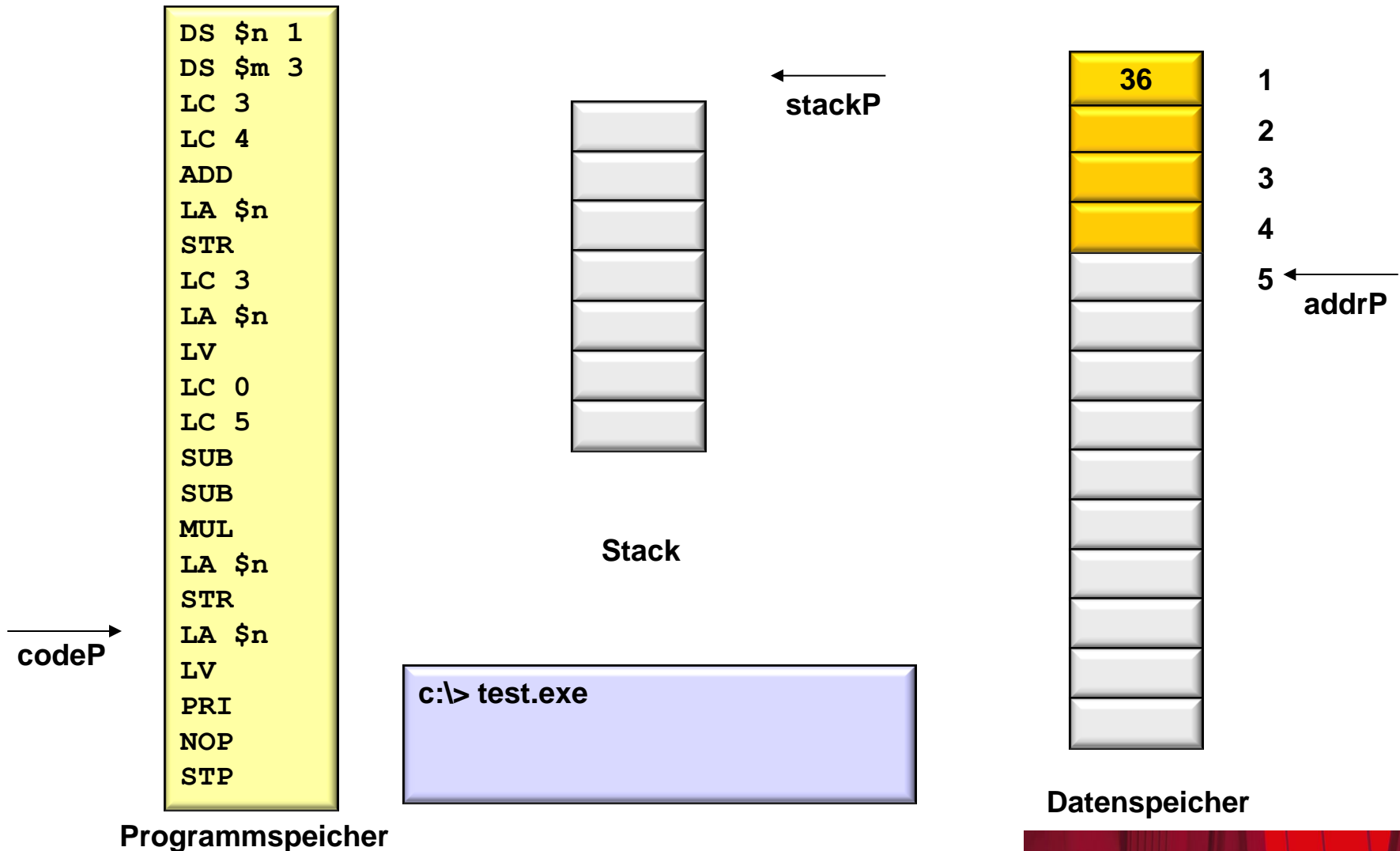


# Eine einfache Stack-Maschine

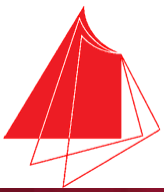




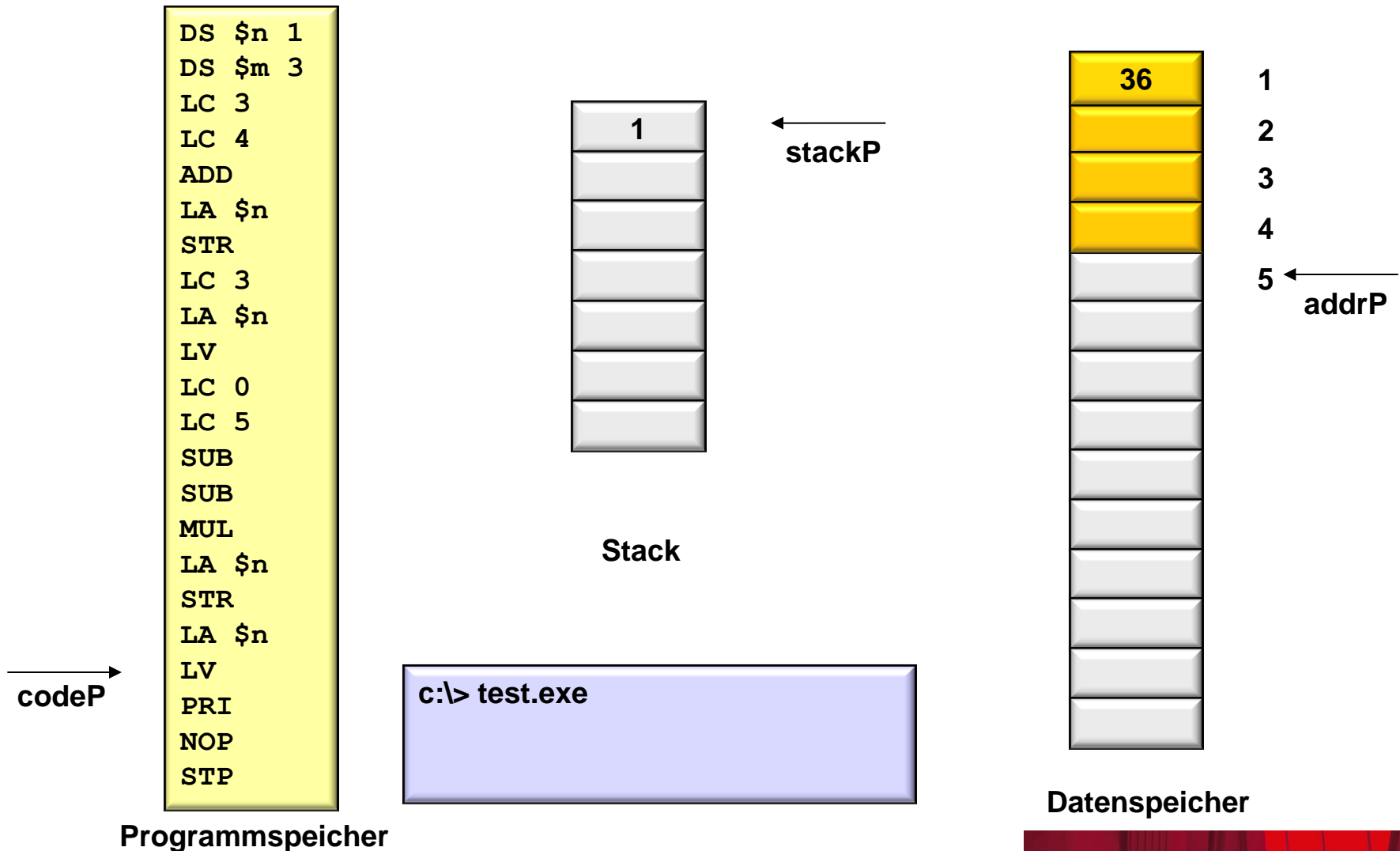
# Eine einfache Stack-Maschine

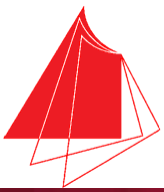




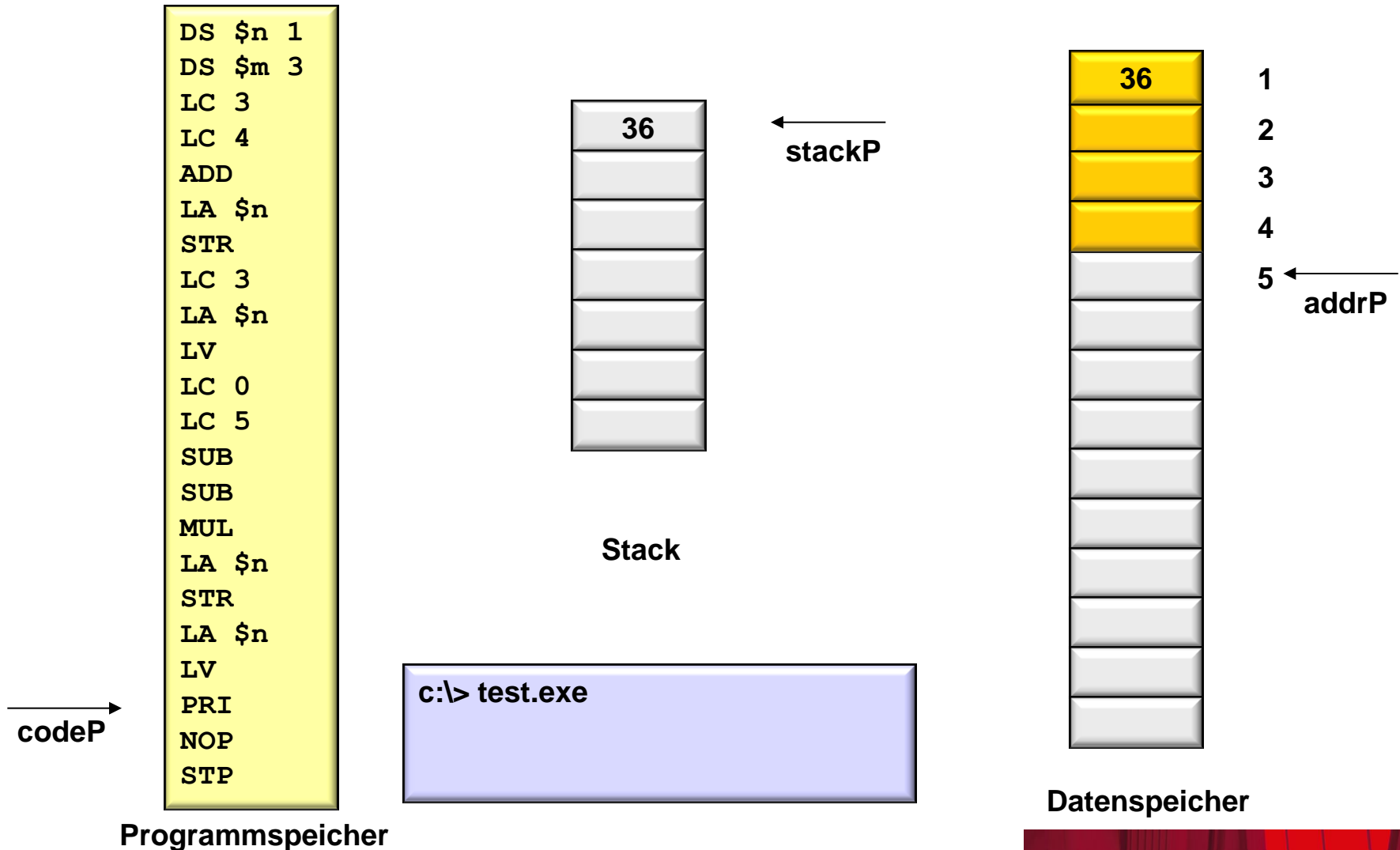


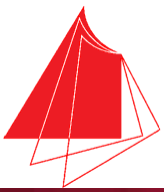
# Eine einfache Stack-Maschine



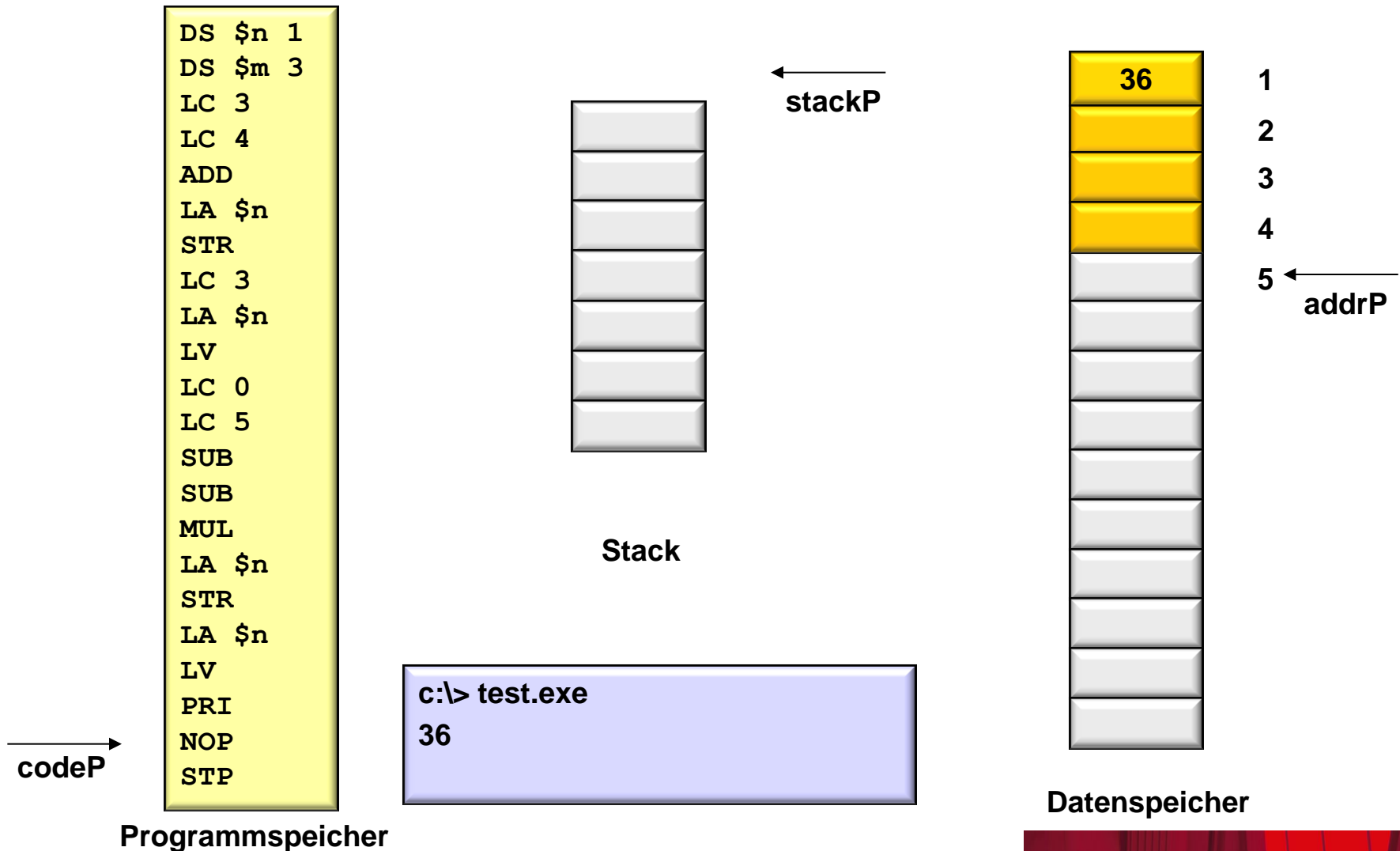


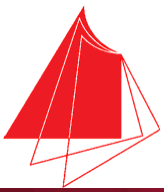
# Eine einfache Stack-Maschine



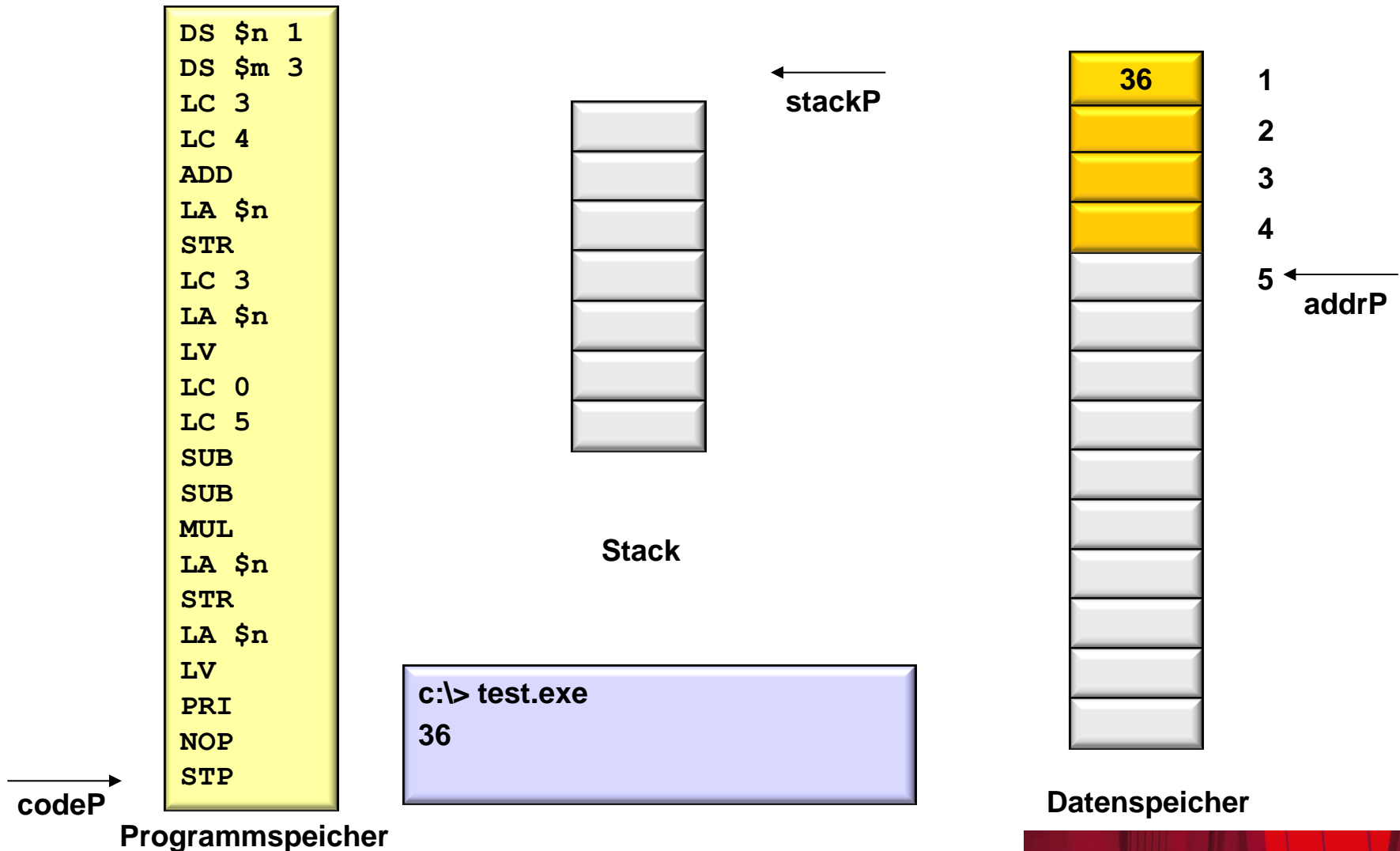


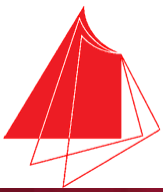
# Eine einfache Stack-Maschine



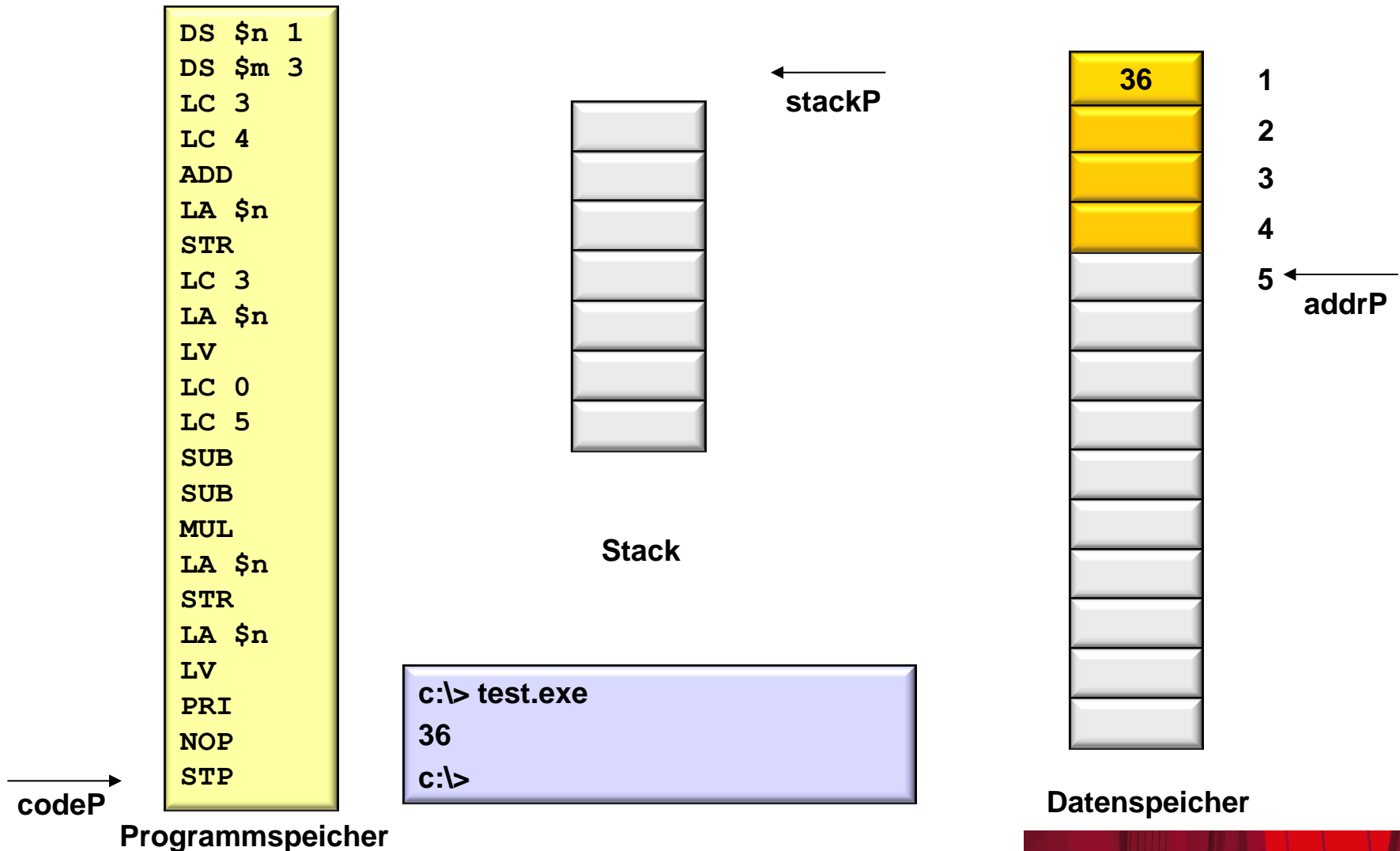


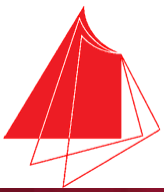
# Eine einfache Stack-Maschine





# Eine einfache Stack-Maschine





# Maschinenbefehle

CMD | CMD argument | label CMD | label CMD argument

- **Arithmetik-Befehle ohne Argument**

- Addition für Integer

– **ADD**

```
codeP++;  
*(stackP-1) = *(stackP-1)  
              + *stackP;  
stackP--;
```

- Subtraktion für Integer

– **SUB**

```
codeP++;  
*(stackP-1) = *(stackP-1)  
              - *stackP;  
stackP--;
```

- Multiplikation für Integer

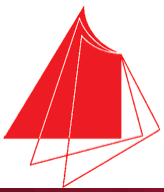
– **MUL**

```
codeP++;  
*(stackP-1) = *(stackP-1)  
              * *stackP;  
stackP--;
```

- Division für Integer

– **DIV**

```
codeP++;  
*(stackP-1) = *(stackP-1)  
              / *stackP;  
stackP--;
```



# Maschinenbefehle

CMD | CMD argument | label CMD | label CMD argument

- **Vergleiche ohne Argument**

- Kleiner für Integer
  - **LES**

```
codeP++;
if (*(stackP-1) < *stackP)
    *(stackP-1) = 1;
else *(stackP-1) = 0;
stackP--;
```
- Gleich für Integer
  - **EQU**

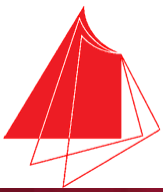
```
codeP++;
if (*(stackP-1) == *stackP)
    *(stackP-1) = 1;
else *(stackP-1) = 0;
stackP--;
```

- **Logische Operationen ohne Argument**

- Konjunktion für Integer – **AND**

```
codeP++;
if (*(stackP-1) != 0
    && *stackP != 0)
    *(stackP-1) = 1;
else *(stackP-1) = 0;
stackP--;
```
- Negation für Integer – **NOT**

```
codeP++;
if (*(stackP) == 0 )
    *stackP = 1;
else *stackP = 0;
```



# Maschinenbefehle

CMD | CMD argument | label CMD | label CMD argument

## Laden, Speichern

- Laden einer Speicheradresse (mit Argument) – **LA variable**  
`codeP += 2; stackP++;`  
`*stackP = addr(varianble);`
- Laden einer Konstante (int) (mit Argument) – **LC c**  
`codeP += 2; stackP++;`  
`*stackP = c;`
- Laden eines gespeicherten Werts (ohne Argument) – **LV**  
`codeP++;`  
`*stackP = **stackP;`
- Speichern eines Werts (ohne Argument) – **STR**  
`codeP++;`  
`**stackP = *(stackP-1);`  
`stackP -= 2;`

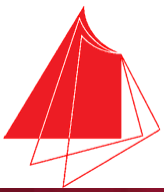
## Einlesen und Drucken ohne Argument

- Drucken eines Integers – **PRI**  
`codeP ++;`  
`println(*stackP);`  
`stackP--;`
- Einlesen eines Integers – **REA**  
`codeP ++; stackP++;`  
`*stackP = read();`

## Sprünge mit Argument

- Unbedingter Sprung – **JMP #label**  
`codeP = *label`  
`// springt an die mit`  
`// *label markierte Codezeile`
- Bedingter Sprung – **JIN #label**  
`if(*stackP == 0)`  
`codeP = *label;`  
`else codeP += 2;`  
`stackP--;`





# Maschinenbefehle

CMD | CMD argument | label CMD | label CMD argument

- Speicher reservieren mit Argument – **DS identifier size**

```
codeP+=3;  
addr(identifier) = addrP;  
addrP += size;
```

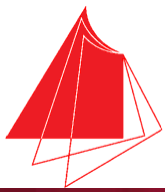
- Was noch fehlt

- Nichts tun (ohne Argument) – **NOP**

```
codeP++;
```

- Stoppen (ohne Argument) – **STP**

```
exit();
```



# Code-Erzeugung: makeCode

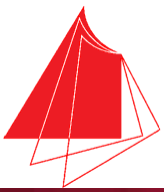
```
makeCode (PROG ::= DECLS STATEMENTS){  
  makeCode(DECLS); makeCode(STATEMENTS);  
  code << " STP " ;}
```

```
makeCode (DECLS ::= DECL ; DECLS ){  
  makeCode(DECL ); makeCode(DECLS); }
```

```
makeCode (DECLS ::=  $\epsilon$ ){ }
```

```
makeCode(DECL::= int ARRAY identifier){  
  code << " DS " << "$" << getLexem(identifier); makeCode(ARRAY)}
```

```
makeCode(ARRAY ::= [ integer ] ) { code << getValue(integer) ;}  
makeCode(ARRAY ::=  $\epsilon$  ) {code << 1;}
```



# Code-Erzeugung: makeCode

```
makeCode (STATEMENTS ::= STATEMENT ; STATEMENTS){  
    makeCode(STATEMENT ); makeCode(STATEMENTS);}
```

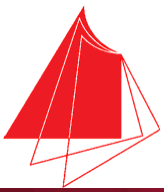
```
makeCode (STATEMENTS ::=  $\epsilon$ ){code << " NOP "};}
```

```
makeCode (STATEMENT ::= identifier INDEX = EXP ){ makeCode(EXP);  
    code << " LA " << "$" << getLexem(identifier) ; makeCode(INDEX); code << " STR "; }
```

```
makeCode (STATEMENT ::= write( EXP ) ){ makeCode(EXP);  
    code << " PRI "; }
```

```
makeCode (STATEMENT ::= read( identifier INDEX ) ){  
    code << " REA ";  
    code << " LA " << "$" << getLexem(identifier); makeCode(INDEX); code << " STR "; }
```

```
makeCode (STATEMENT ::= { STATEMENTS } ){ makeCode(STATEMENTS); }
```



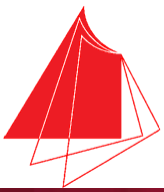
# Code-Erzeugung: makeCode

```
makeCode (STATEMENT ::= if ( EXP ) STATEMENT else STATEMENT ){  
  makeCode(EXP);  
  code << " JIN " << "#" << label1; // label1 ist neu  
  makeCode(STATEMENT );  
  code << " JMP " << "#" << label2; // label2 ist neu  
  code << "#" << label1 << " NOP ";  
  makeCode(STATEMENT );  
  code << "#" << label2 << " NOP ";;}
```

```
if (exp) stat1  
else stat2;  
  
=  
If (!exp) goto m1;  
stat1; goto m2;  
m1: stat2;  
m2: ...
```

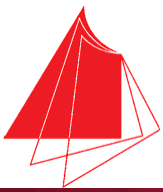
```
makeCode (STATEMENT ::= while ( EXP ) STATEMENT){  
  code << "#" << label1 << " NOP "; // label1 ist neu  
  makeCode(EXP);  
  code << " JIN " << "#" << label2; // label2 ist neu  
  makeCode(STATEMENT );  
  code << " JMP " << "#" << label1;  
  code << "#" << label2 << " NOP ";;}
```

```
while (exp) stat;  
  
=  
m1: If (!exp) goto m2;  
stat; goto m1;  
m2: ...
```



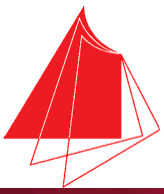
# Code-Erzeugung: makeCode

```
makeCode (EXP ::= EXP2 OP_EXP ){  
  if (OP_EXP.type == noType ) makeCode(EXP2);  
  else if (OP_EXP.OP.type == opGreater) {  
    makeCode(OP_EXP); makeCode(EXP2); code << "LES";}  
  else if (OP_EXP.OP.type == opUnequal) {  
    makeCode(EXP2); makeCode(OP_EXP); code << "NOT";}  
  else { makeCode(EXP2); makeCode(OP_EXP); } }  
  
makeCode (INDEX ::= [ EXP ]){makeCode(EXP); code << " ADD ";}  
makeCode (INDEX ::= ε){}  
  
makeCode (EXP2 ::= ( EXP )){ makeCode(EXP); }
```



# Code-Erzeugung: makeCode

```
makeCode (EXP2 ::= identifier INDEX){  
    code << " LA " << "$" << getLexem(identifier) ; makeCode(INDEX); code << " LV " ;}  
  
makeCode (EXP2 ::= integer ){ code << " LC " << getValue(integer) ;}  
  
makeCode (EXP2 ::= - EXP2){  
    code << " LC " << 0  
    makeCode(EXP2);  
    code << " SUB " ;  
}  
  
makeCode (EXP2 ::= ! EXP2){  
    makeCode(EXP2);  
    code << " NOT " ;  
}
```

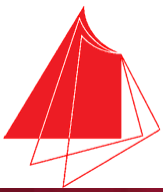


# Code-Erzeugung: makeCode

```
makeCode (OP_EXP ::= OP EXP ){  
  makeCode(EXP);  
  makeCode(OP); }
```

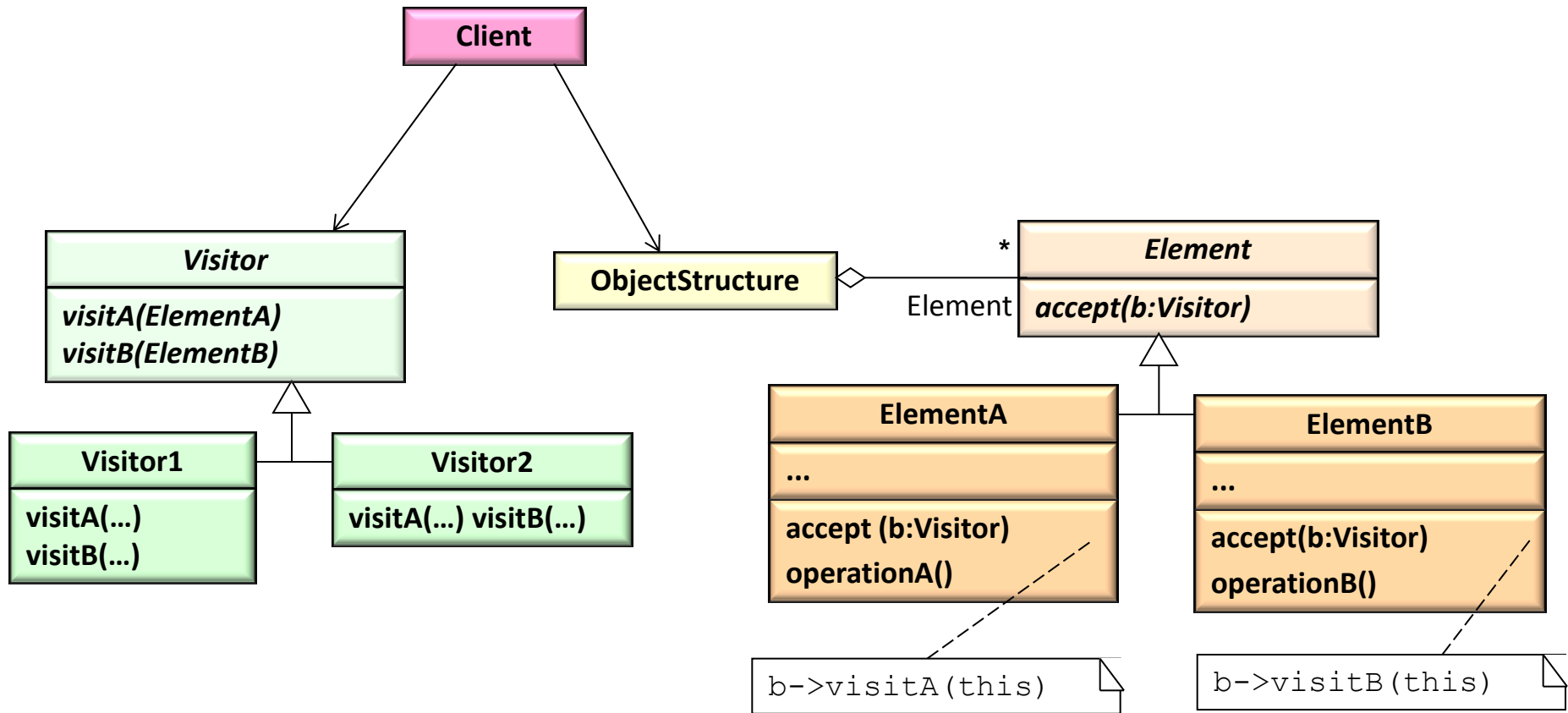
```
makeCode (OP_EXP ::= ε){}
```

```
makeCode (OP ::= +) { code << "ADD";}  
makeCode (OP ::= -) { code << "SUB";}  
makeCode (OP ::= *) { code << "MUL";}  
makeCode (OP ::= /) { code << "DIV";}  
makeCode (OP ::= <) { code << "LES";}  
makeCode (OP ::= >) { ; } // siehe EXP ::= EXP2 OP_EXP  
makeCode (OP ::= ==) { code << "EQU";}  
makeCode (OP ::= !=) { code << "EQU"; } // siehe EXP ::= EXP2 OP_EXP  
makeCode (OP ::= &) { code << "AND";}
```

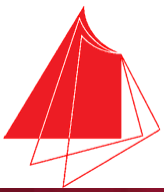


# Alternativen: 1. Ein Besucher

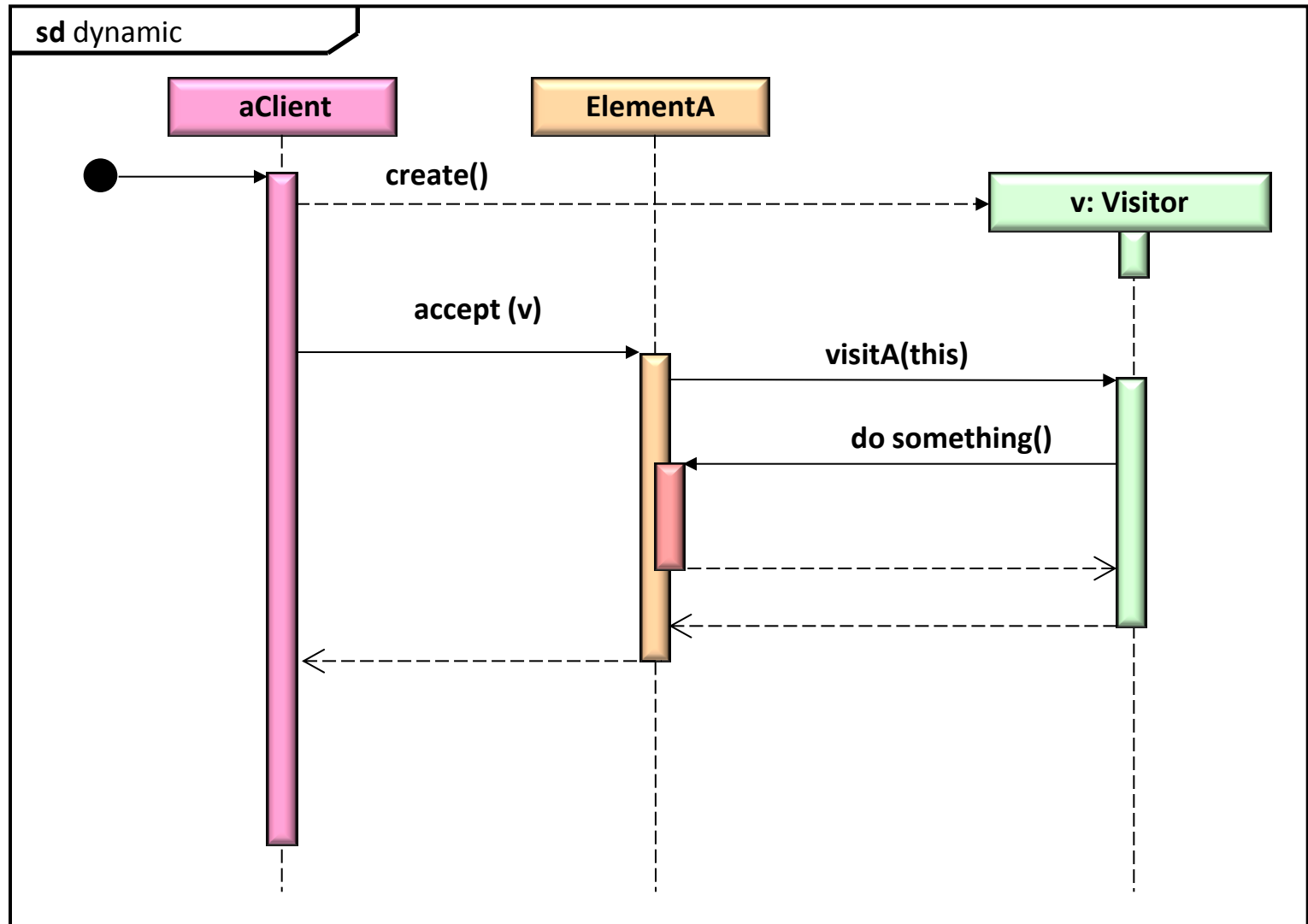
Auskopplung der Funktionalität über einen Besucher:





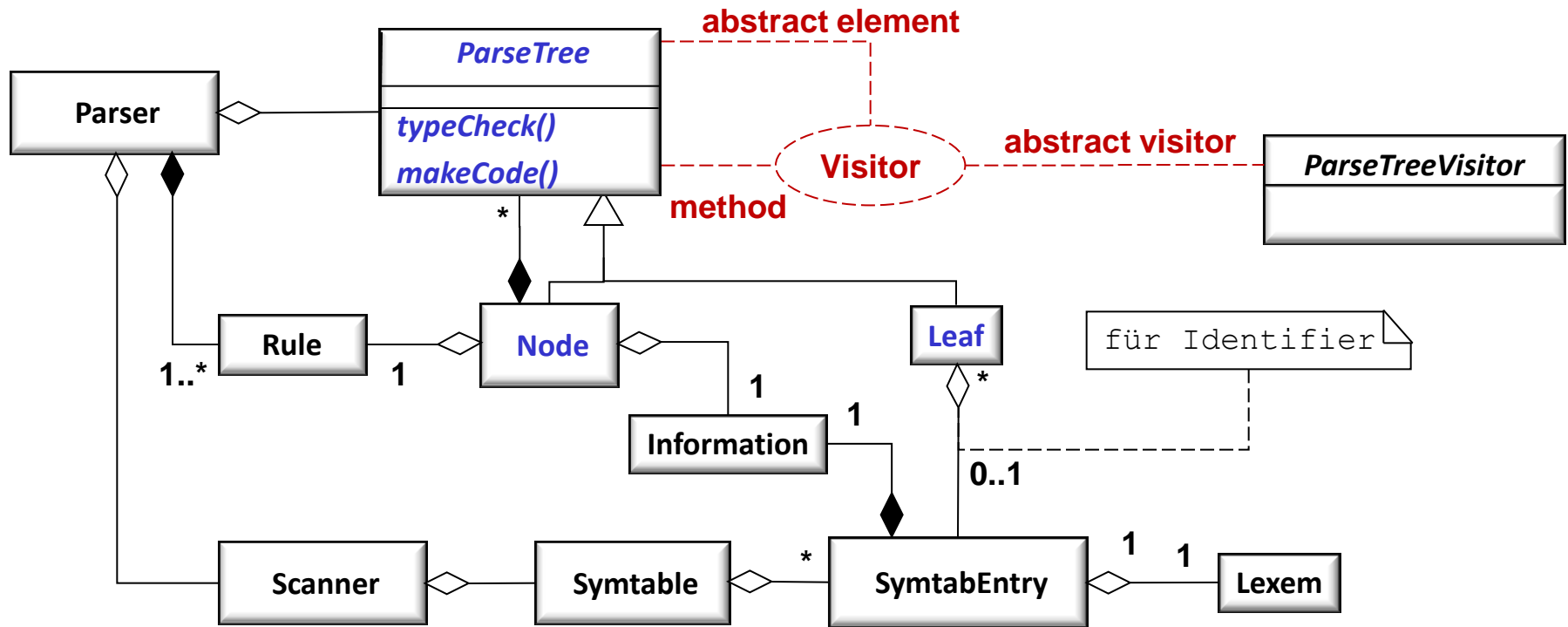


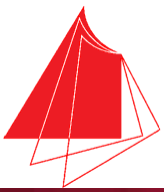
# Dynamisches Verhalten



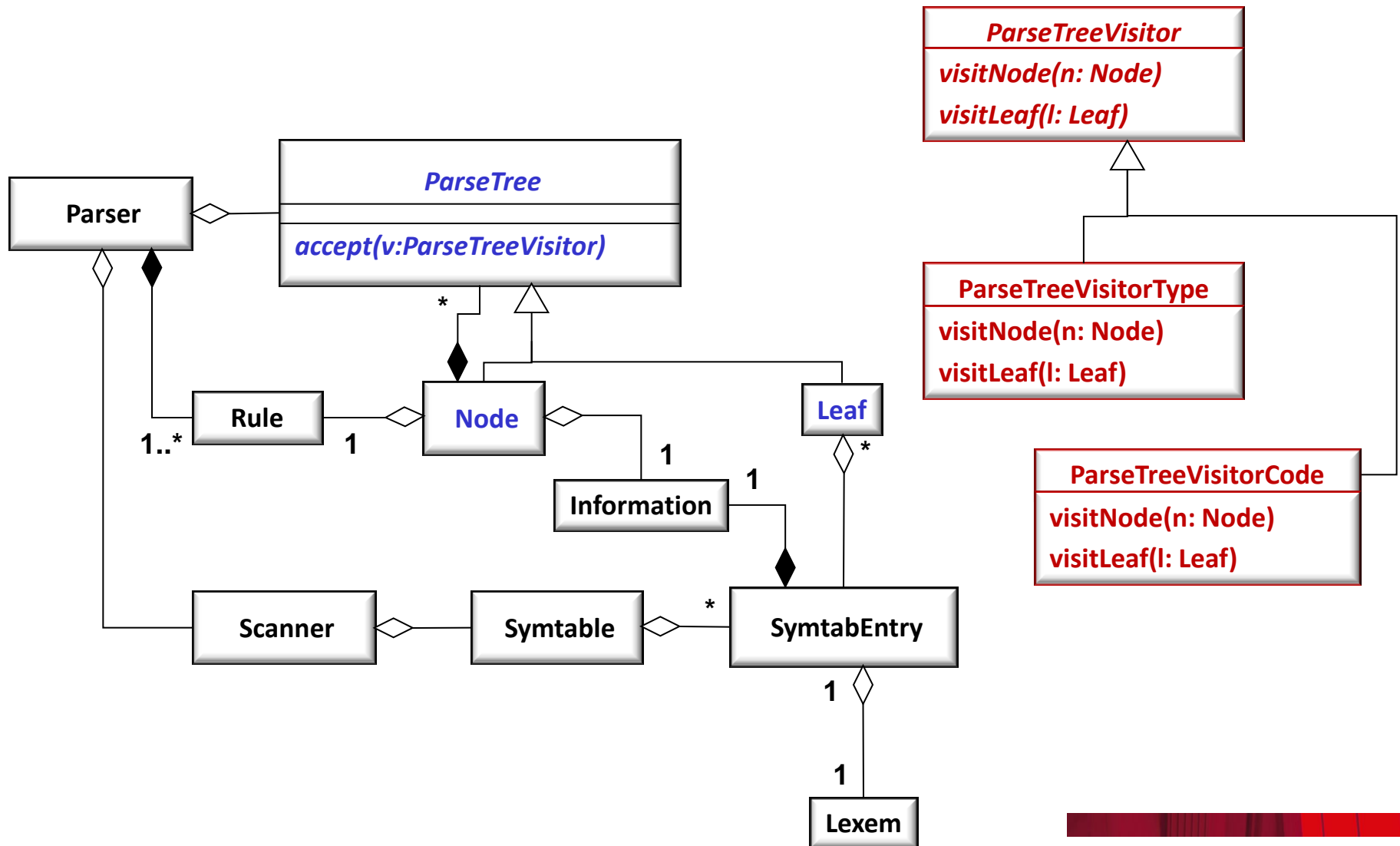


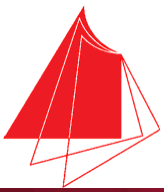
# Zuordnungen





# Das erwartete Ziel





# Alternativen: 2. Differenzierte Knoten-Klassen

