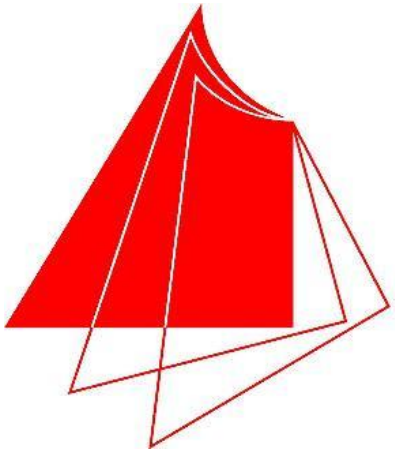
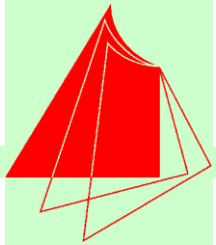


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

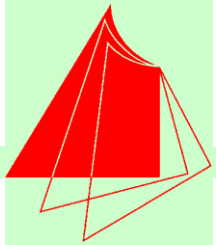
Labor
Systemnahes Programmieren

Prof. Dr. Thomas Fuchß
Fakultät für Informatik und Wirtschaftsinformatik –
Fachgebiet Informatik



Übersicht

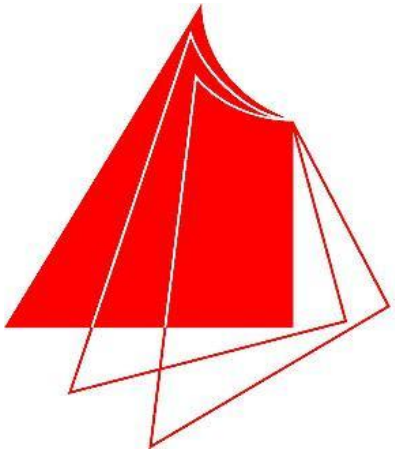
- **Allgemeines**
- **Teil I Lexikalische Analyse**
- **Teil II Syntaktische Analyse**
- **Teil III Prozesssynchronisation und Prozesskommunikation**



Allgemeines

- **Veranstaltungen**
  - jeweils mittwochs von 14.00 – 18.50 Uhr (LI 137)
  - Vorbesprechung der nächsten Aufgabe jeweils mittwochs um 14.00 Uhr im Seminarraum E 201

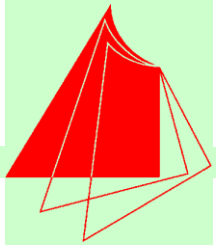
    Termine: 7.10., 11.11., 23.12.
- **Zeitplan**
  - Phase I 5 Termine (07.10. – 04.11.)
  - Phase II 6 Termine (11.11. – 16.12.)
  - Phase III 4 Termine (23.12. – 27.01.)
- **Werkzeuge und Sprachen**
  - C, C++



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Systemnahes Programmieren
Teil II
Syntaktische Analyse

Prof. Dr. Thomas Fuchß
Fakultät für Informatik und Wirtschaftsinformatik –
Fachgebiet Informatik



Literatur

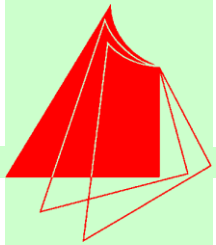
- A.V. Aho, R. Sethi und J.D. Ullmann.
  Compilerbau – 2nd Edition – München; Wien: Oldenburg, 1999

- N. Wirth.
  Grundlagen und Techniken des Compilerbaus - Bonn : Addison-Wesley, 1996.

- W. M. Waite und G. Goos.
  Compiler construction - New York : Springer, 1984

- B. Bauer und R. Höllerer.
  Übersetzung objektorientierter Programmiersprachen : Konzepte, abstrakte Maschinen und Praktikum "Java-Compiler"- Berlin; Heidelberg : Springer, 1998.

- D. Grune et. al.
  Modern compiler design - Chichester ; Weinheim : Wiley, 2000.



Ziel:

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

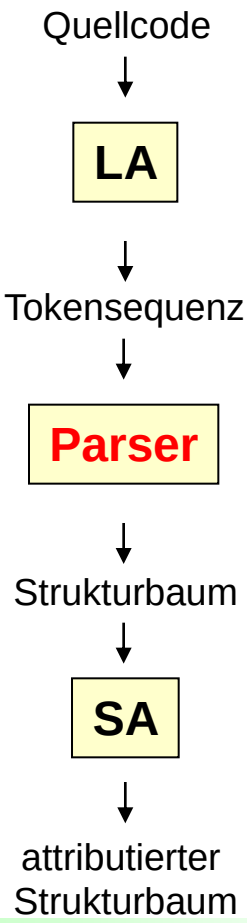
Insbesondere gilt es, das Prinzip des *rekursiven Abstiegs* verstehen und anwenden zu können.

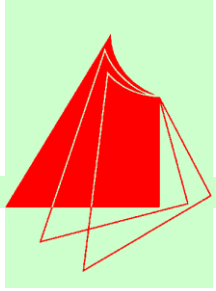
Die drei Teile der Analysephase

- **lexikalische Analyse**
  Zerlegung des Quellcodes in die Grundsymbole (Tokens) und Speichern und Weiterleiten von Informationen (Namen, Values).
  Token: Bezeichner, Schlüsselworte, Sonderzeichen, Zahlen

- **syntaktische Analyse**
  Überprüft die syntaktischen Spracheigenschaften und erzeugt den Strukturbaum
  (sind Ausdrücke korrekt *a = (b+c(;* **o.ä.)**

- **semantische Analyse:**
  bestimmt die statischen semantischen Eigenschaften des Programms und prüft deren Konsistenz
  - Gültigkeitsbereiche (Namensräume)
  - Typisierung (Ausdrücke, Variablen, ...)
  - Deklarationen





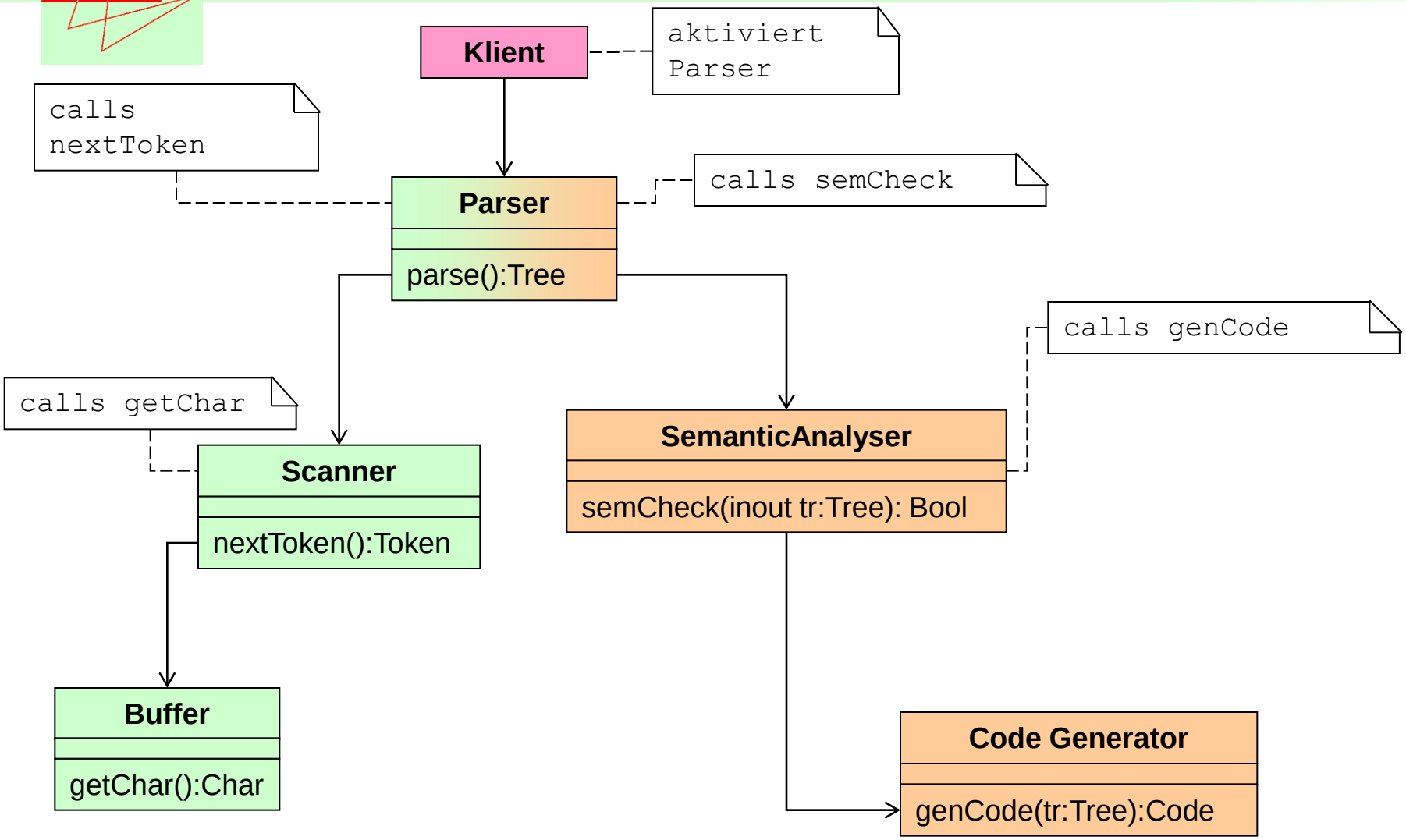
Was macht man mit der Tokensequenz?

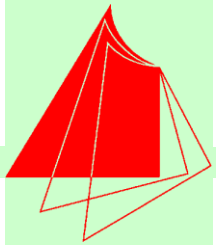
Zu jeder Programmiersprache gehören Regeln, die festlegen, wie die syntaktische Struktur wohlgeformter Pogramme auszusehen hat. Der Parser überprüft diese Regeln.

- Er fordert die Tokens vom Scanner an,
- prüft, ob die Reihenfolge der Tokens sinnvoll ist (den Regeln der Programmiersprache entspricht) und
- baut den Strukturbaum (Parse-Baum) auf, der dann in der semantischen Analyse zur Typprüfung weiterverarbeitet wird.

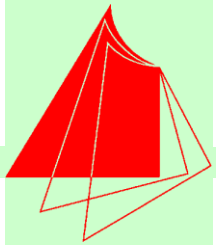
- Erkennt und behandelt Fehler.

Ablauf


Klient
aktiviert
Parser
calls
nextToken
Parser
parse():Tree
calls semCheck
calls genCode
calls getChar
Scanner
nextToken():Token
SemanticAnalyser
semCheck(inout tr:Tree): Bool
Buffer
getChar():Char
Code Generator
genCode(tr:Tree):Code



Was soll man tun?



Aufgabe

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

Hinweis:

- normaleTerminale (Schlüsselworte) 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

PROG ::= *DECLS STATEMENTS*

DECLS ::= *DECL* **;** *DECLS* | ε

DECL ::= *TYPE ARRAY* ***identifier***

ARRAY ::= **[** ***integer*** **]** | ε

TYPE ::= **int** | **float**

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

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

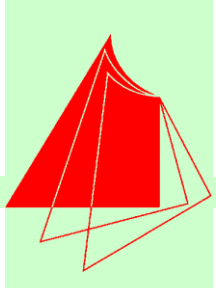
EXP ::= *EXP2 OP_EXP*

EXP2 ::= **(** *EXP* **)** | ***identifier*** *INDEX* | ***integer*** | ***real*** | **-** *EXP2* | **!** *EXP2* | **float** *EXP2*

INDEX ::= **[** *EXP* **]** | ε

OP_EXP ::= *OP EXP* | ε

OP ::= **+** | **-** | ***** | **/** | **<** | **=** | **&**



Aufgabe

b) Verwenden Sie hierzu den Scanner aus Aufgabe 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“ |

\+ ::= „+“

...

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

...

] ::= „]“

integer ::= digit {digit}

real ::= integer „.“ integer [„E“ [„+“ | „-“] integer]

identifier ::= letter {letter | digit}

write ::= „write“

read ::= „read“

if ::= „if“

else ::= „else“

while ::= „while“

int ::= „int“

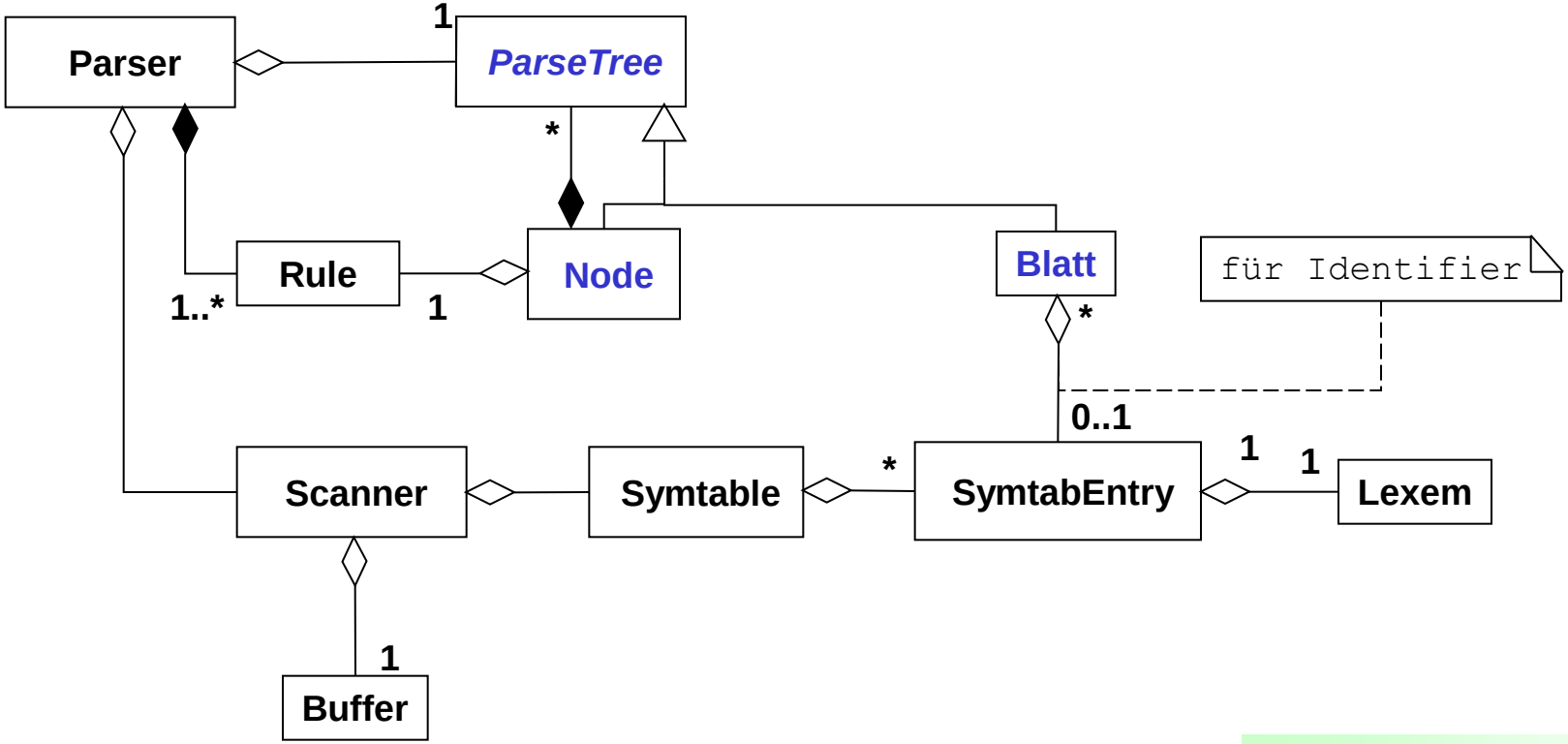
float ::= „float“

Aufgaben:

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

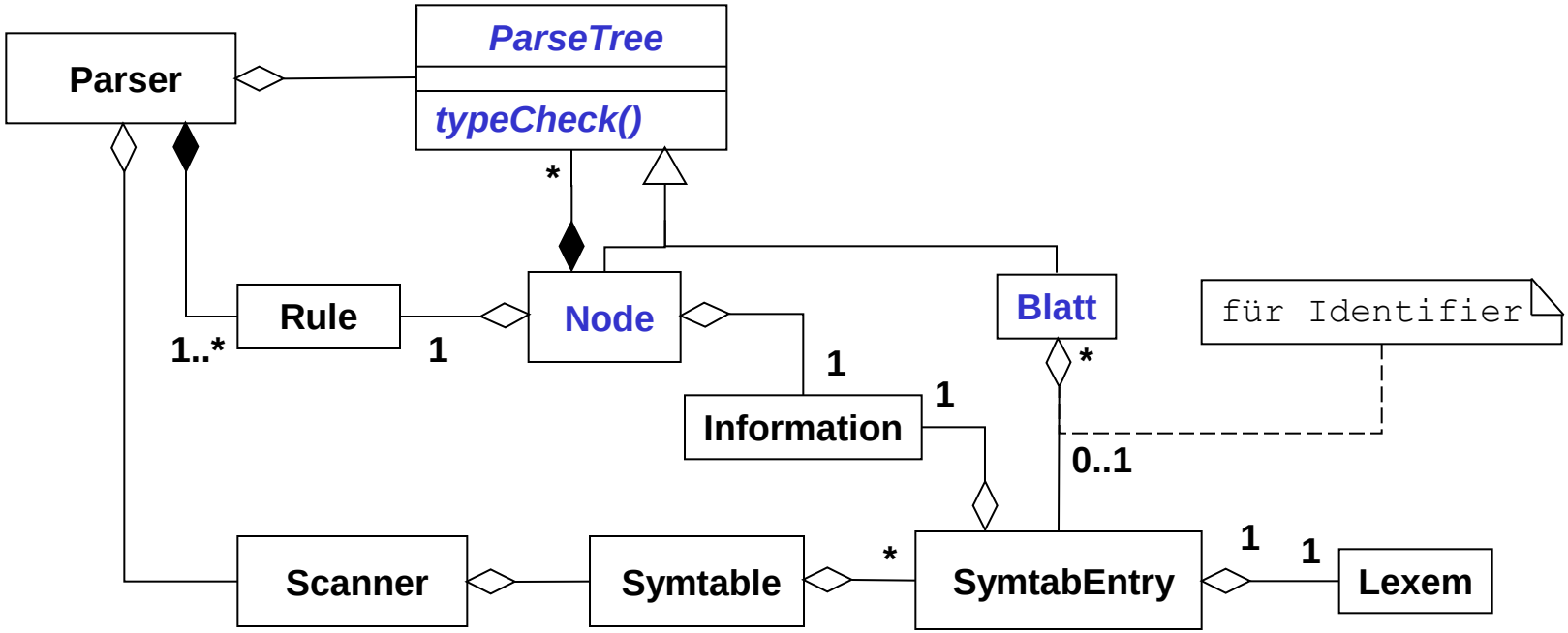
Erweitern Sie hierzu die Funktionen des Parsers, so dass sie nicht nur die Syntax überprüfen, sondern auch den Baum aufbauen.

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



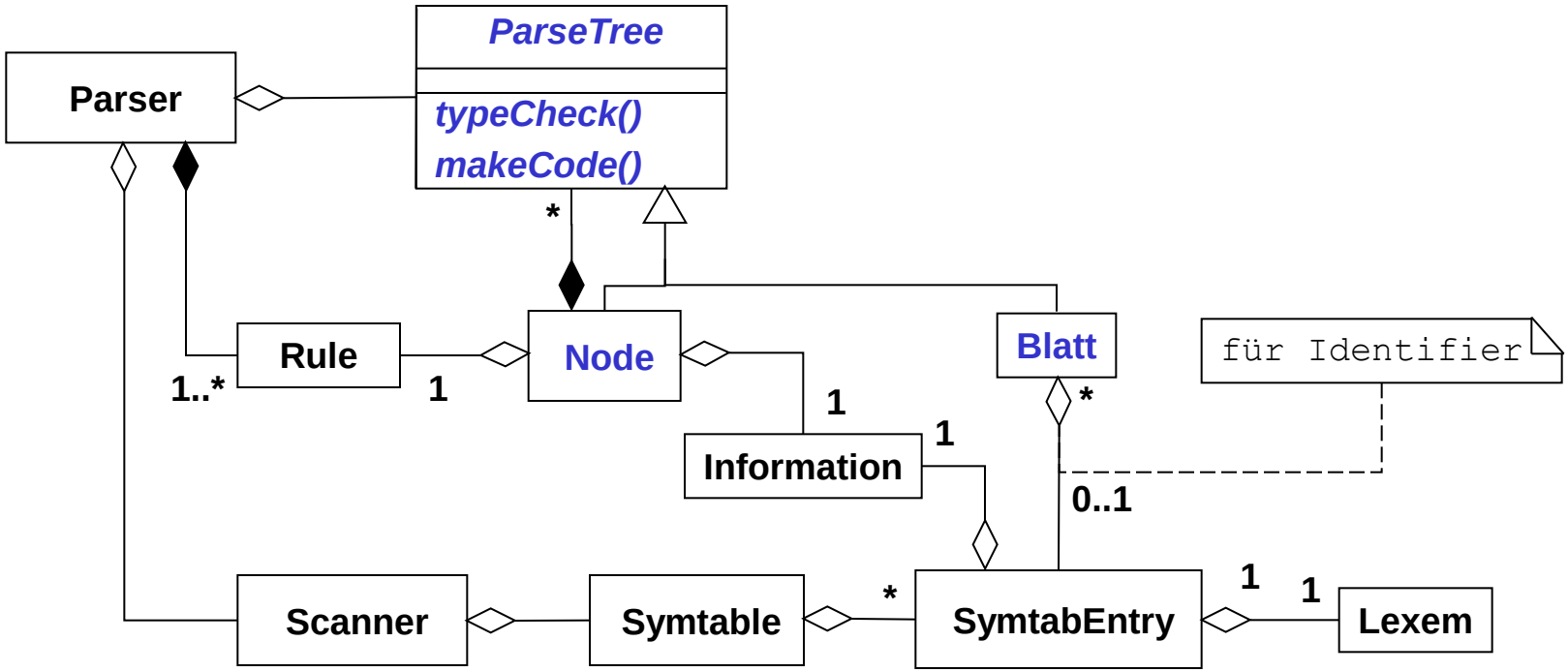
Aufgaben:

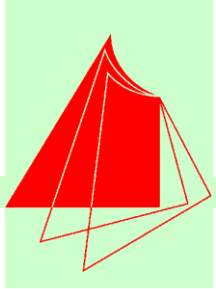
d) Evaluieren Sie den Parse-Baum – ermitteln Sie hierzu, zu jedem Knoten den entsprechenden Typ (gemäß Vorlage) und prüfen Sie, ob die Typen der Unterbäume zusammenpassen.
Speichern Sie die Typ-Information im Knoten und für **Identifier als Information in der Symboltabelle.**
Hinweis: Erweitern Sie hierzu 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)

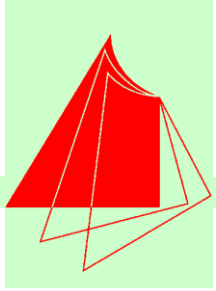
```
C:\> parser Parser-test.txt -c test.code
```

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

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

„Parser-test.txt “

```
C:\>parser parser-error.txt
parsing …
type check …
generate code …
```



Programmaufruf (II)

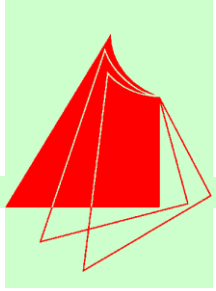
Die resultierende Code-Datei hat dann etwa folgende Gestalt:

```
DS n 1
LC 3
LC 4
ADI
LA n
STR
LC 3
LA n
LV
LC 0
LC 5
SBI
SBI
MLI
LA n
STR
LA n
LV
PRI
NOP
STP
```

„test.code “

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

```
C:\>interpreter.exe test.code
36
C:\>
```



Programmaufruf (III)

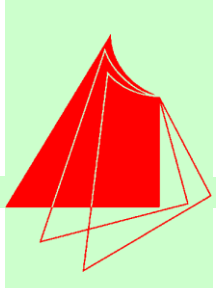
```
C:\> parser Parser-error.txt -c test.code
```

- „**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:     23   Column:     12  TokenRightParent
stop
C:\>
```



Programmaufruf (IV)

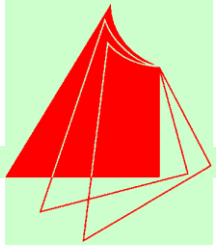
```
C:\> parser Parser-error.txt -c test.code
```

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

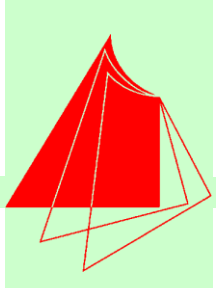
```
...
      n = 3 + 4.5;
...
```

„Parser-error.txt “

```
C:\>parser parser-error.txt
parsing …
type check …
error Line: 25 Column:  8  incompatible types
stop
C:\>
```



Wie soll man das tun?



Warum sehen die Regeln so seltsam aus?

PROG ::= *DECLS STATEMENTS*

DECLS ::= *DECL* **;** *DECLS* | ε

DECL ::= *TYPE ARRAY* ***identifier***

ARRAY ::= **[** ***integer*** **]** | ε

TYPE ::= **int** | **float**

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

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

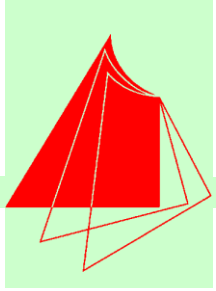
EXP ::= *EXP2 OP_EXP*

EXP2 ::= **(** *EXP* **)** | ***identifier*** *INDEX* | ***integer*** | ***real*** | **-** *EXP2* | **!** *EXP2* | **float** *EXP2*

INDEX ::= **[** *EXP* **]** | ε

OP_EXP ::= *OP EXP* | ε

OP ::= **+** | **-** | ***** | **/** | **<** | **=** | **&**



Und nicht so?

PROG ::= *DECLS* **;** *STATEMENTS* | *STATEMENTS* **;** | ε

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

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

TYPE ::= **int** | **float**

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

STATEMENT ::= ***identifier*** *INDEX* **=** *EXP* | ***identifier*** **[** *EXP* **]** **=** *EXP* | **{***STATEMENTS***}**

print (*EXP* **)** | **read (*identifier*** *INDEX* **)** | **read (*identifier*** **[** *EXP* **]** **)** |

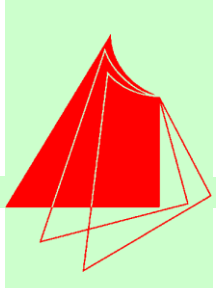
if (*EXP* **)** *STATEMENT* **else** *STATEMENT* |

while (*EXP* **)** *STATEMENT*

EXP ::= *EXP OP EXP* | **(** *EXP* **)** | ***identifier*** | ***integer*** | ***real*** |

- *EXP* | **!** *EXP* | **float** *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

s $\vdash^1$ **id := *e***
$\vdash^1$ **id := id *op e op e***
$\vdash^1$ **id := id + id * *e***

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

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

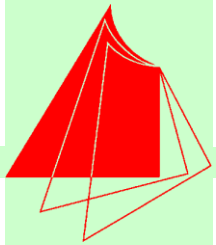
oder

s $\vdash^1$ **id := *e***
$\vdash^1$ **id := *e op e op* id**
$\vdash^1$ **id := *e* + id * id**

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

$\vdash^1$ **id := *e op e op e***
$\vdash^1$ **id := *e op* id * 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

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

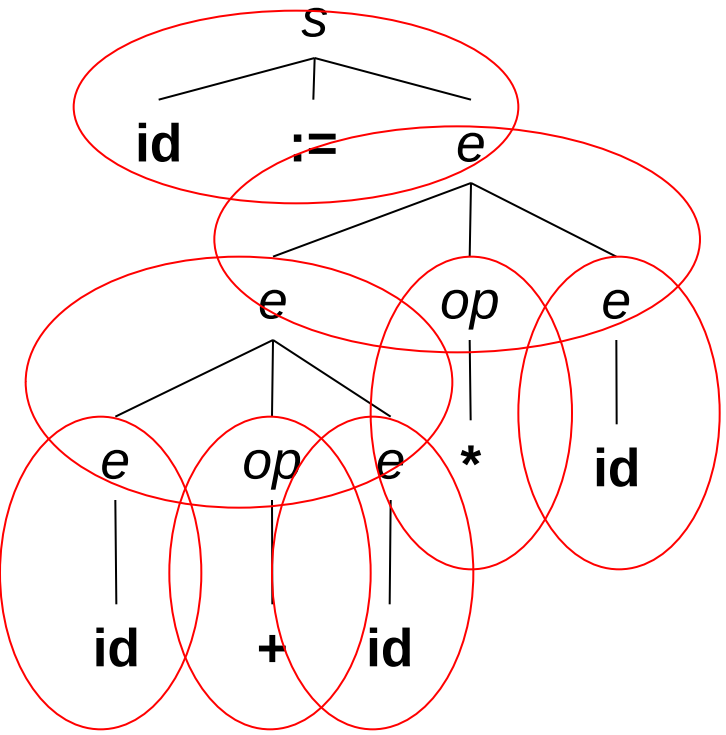
$P = \{t\ \mathbf{q}\ t \vdash^1 \mathbf{q} \mid t \in T\}$ compare Schritt

$\cup\ \{ B\ \mathbf{q} \vdash^1 b_n...b_1\mathbf{q} \mid B ::= b_1...b_n \in P\}$ produce Schritt

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

Nachteil: nicht deterministisch

Beispiel



S **q id := id + id * id**

$\vdash^1$ *e* **:= id q id := id + id * id**

$\vdash^1$ *e* **:= q := id + id * id**

$\vdash^1$ *e* **q id + id * id**

$\vdash^1$ *e op e* **q id + id * id**

$\vdash^1$ *e op e op e* **q id + id * id**

$\vdash^1$ *e op e op* **id q id + id * id**

$\vdash^1$ *e op e op* **q + id * id**

$\vdash^1$ *e op e* **+ q + id * id**

$\vdash^1$ *e op e* **q id * id**

$\vdash^1$ *e op* **id q id * id**

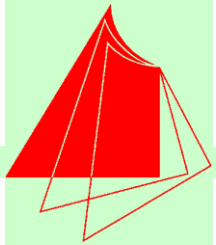
$\vdash^1$ *e op* **q * id**

$\vdash^1$ *e* *** q * id**

$\vdash^1$ *e* **q id**

$\vdash^1$ **id q id**

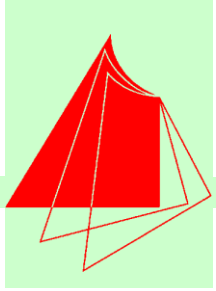
$\vdash^1$ **q**



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?



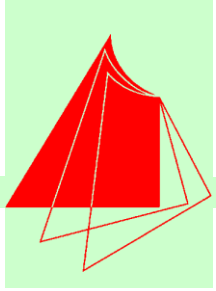
Beispiel

- *E* ::= *T R* | U **-** E
- *R* ::= **+** *T R* | **ε**
- *T* := *F U*
- *U*::= ***** *FU* | **ε**
- *F* ::= **(** *E* **)** | ***id***

(*E* ist das Startsymbol)

Frage:

Wie findet man eine Ableitung für x und -x ?



Wie wird die Ableitung eindeutig?

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

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

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

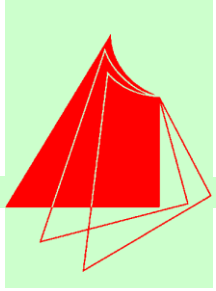
Follow(A) $= \{y \mid y \in T^* \mid \exists\, z \in (N \cup T)^* \text{ mit } Z \vdash zAy\}$ $A \in N$

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.



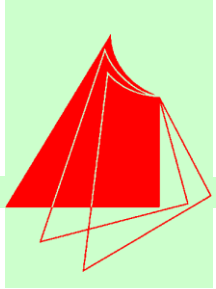
Die Konstruktion von $First_1(x)$

1. Für alle $\mathbf{x} \in T$ ist $First_1(\mathbf{x}) = \{\mathbf{x}\}$
2. ist $X ::= \varepsilon \in P$ dann ist auch $\# \in First_1(X)$
3. ist $X ::= y_1 ... y_n \in P$ dann ist $First_1(y_1 ... y_n) \subseteq First_1(X)$

4. für jedes Terminal a gilt : $a \in First_1(y_1 ... y_n)$
   gdw.
   $a \in First_1(y_1)$ oder $a \in First_1(y_i)$ für i $(1<i\leq n)$ und $\# \in First_1(y_1)$ bis $First_1(y_{i-1})$

5. $\# \in First_1(y_1 ... y_n)$ gdw. $\# \in First_1(y_1)$ bis $First_1(y_n)$

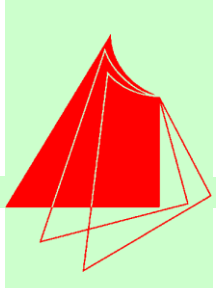
$First_1(w)$ für $w \in (N \cup T)^*$, ist die kleinste Menge, die 1-5 erfüllt.



Die Konstruktion von $Follow_1(A)$

1. $\# \in Follow_1(Z)$
2. ist $B ::= vAw \in P$ dann ist $First_1(w) \backslash \{\#\} \subseteq Follow_1(A)$
3. ist $B ::= vA$ oder $B ::= vAw \in P$ und $\# \in First_1(w)$, dann ist $Follow_1(B) \subseteq Follow_1(A)$

$Follow_1(A)$ für $A \in N$, ist die kleinste Menge, die 1-3 erfüllt.



Beispiel

- $E ::= T\ R \mid U\ \text{-}\ E$
- $R ::= \text{+}\ T\ R \mid \varepsilon$
- $T := F\ U$
- $U ::= \text{*}\ F U \mid \varepsilon$
- $F ::= (\ E\ ) \mid id$

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

$First_1(U) = \{*, \#\}$

$First_1(T) = First_1(F) = \{(, id\}$

$First_1(R) = \{+, \#\}$

$First_1(E) = First_1(T) \cup First_1(U) \setminus \{\#\} \cup \{-\} = \{(, id, *, -\}$

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

$Follow_1(R) = Follow_1(E) = \{), \#\}$

$Follow_1(T) = First_1(R) \setminus \{\#\} \cup Follow_1(E) \cup Follow_1(R) = \{+,), \#\}$

$Follow_1(U) = \{-\} \cup Follow_1(T) = \{-,), +, \#\}$

$Follow_1(F) = First_1(U) \setminus \{\#\} \cup Follow_1(T) \cup Follow_1(U) = \{*, -,), +, \#\}$

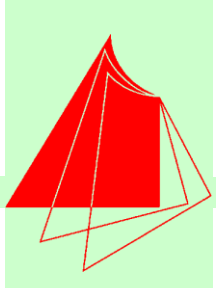
Ein impliziter Keller

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 $A ::= \mathbf{n}B\mathbf{m}$ und $A ::= \mathbf{mm}C \in P$ und die Grammatik SLL(1)

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



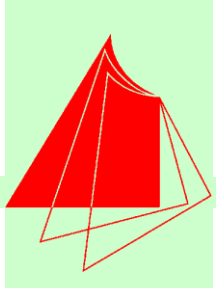
Aufgabe

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

Hinweis:

- normaleTerminale (Schlüsselworte) 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

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



Aufgabe

b) Verwenden Sie hierzu den Scanner aus Aufgabe 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“ |

+ ::= „+“

...

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

...

] ::= „]“

integer ::= digit {digit}

real ::= integer „.“ integer [„E“ [„+“ | „-“] integer]

identifier ::= letter {letter | digit}

write ::= „write“

read ::= „read“

if ::= „if“

else ::= „else“

while ::= „while“

int ::= „int“

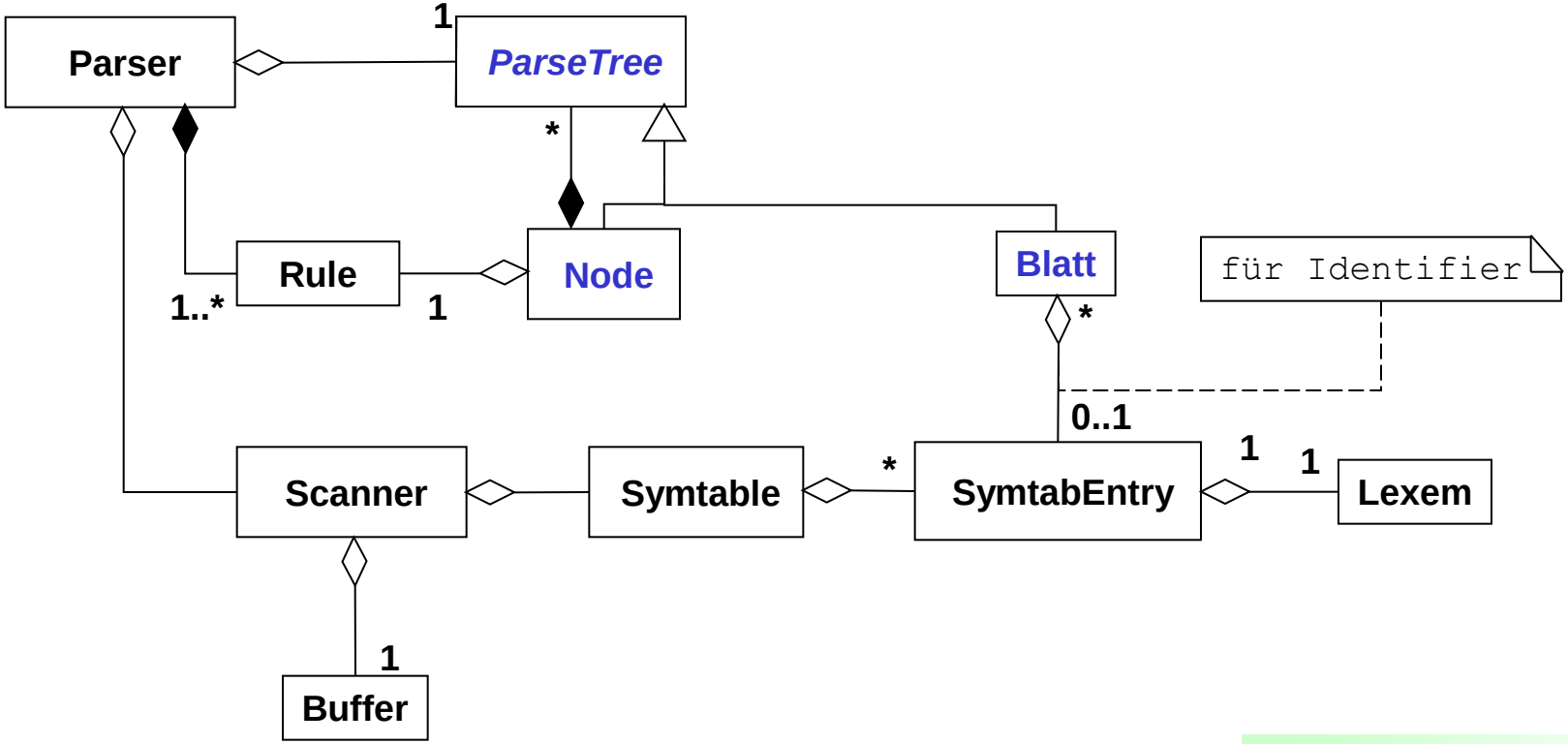
float ::= „float“

Aufgaben:

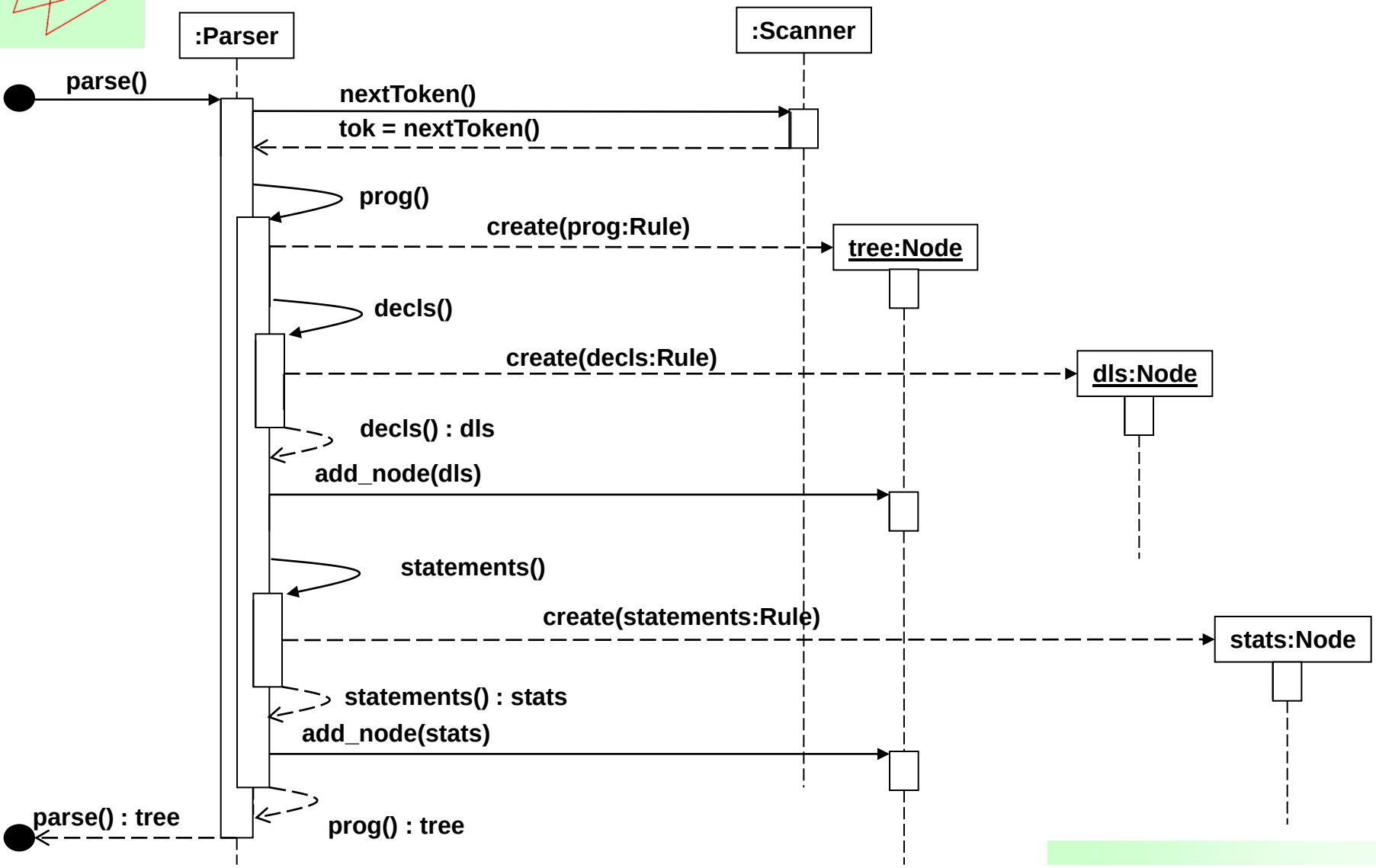
c) Erstellen Sie einen Strukturbaum (Parse-Baum).

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)

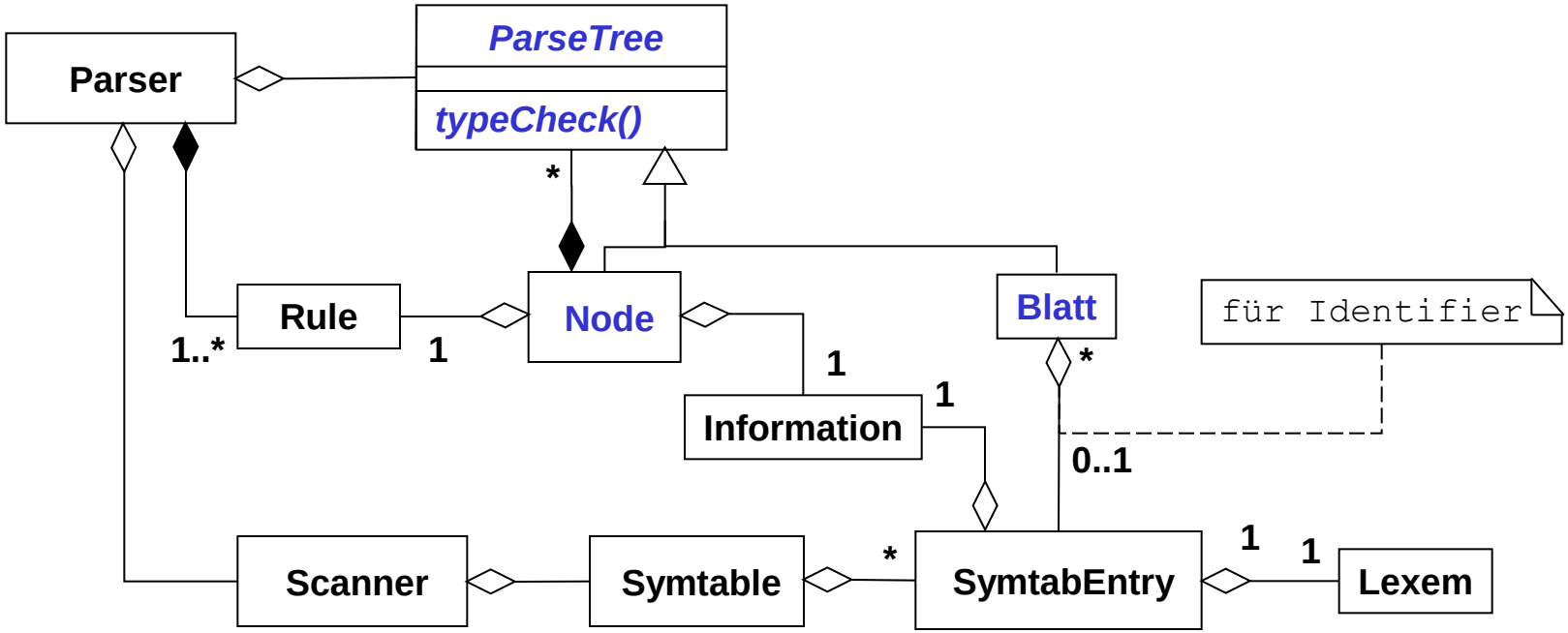


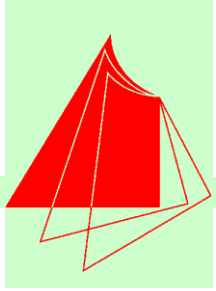
Ein Szenario



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 **Identifier als Information in der Symboltabelle.**
Hinweis: Erweitern Sie hierzu die Klasse *ParseTree* um eine Operation **`typeCheck`**.

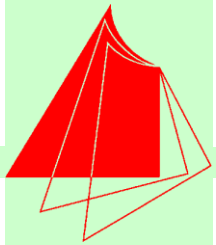




Typisierung und Typ-Check

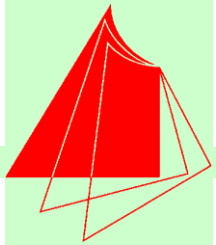
Typ-Informationen:
intType, realType, arrayType,
intArrayType, realArrayType,
noType, errorType,
opPlus, opMinus, opMult, opDiv,
opLess, opNeg, opAnd

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

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

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

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

typeCheck(ARRAY::=[integer]){ if(integer.value>0) this.type= arrayType}; else this.type= errorType;}
typeCheck(ARRAY ::= ε){ this.type = noType ;}

typeCheck(Type ::= int ) { this.type = intType;} typeCheck(Type ::= float ) { this.type = realType;}
```



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))
            || EXP.type == realType && (
                ( getType(identifier) == realType && INDEX.type == noType)
                || ( getType(identifier) == realArrayType && INDEX.type = arrayType))  this.type = noType;
  else { error(„incompatible types“); this.type = errorType; }}

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

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



Typisierung und Typ-Check

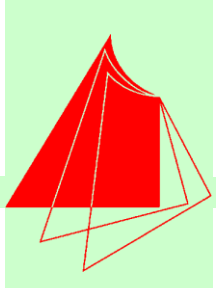
typeCheck (STATEMENT ::= **{** STATEMENTS **}**){ typeCheck(STATEMENTS); this.type = noType; }

typeCheck (STATEMENT ::= **if (**EXP **)** STATEMENT **else** STATEMENT){
  typeCheck(EXP); typeCheck(STATEMENT); typeCheck(STATEMENT);
  if (EXP.type != intType){error(„integer required“); this.type = errorType; }
  else this.type = noType; }

typeCheck (STATEMENT ::= **while (**EXP **)** STATEMENT){
  typeCheck(EXP); typeCheck(STATEMENT);
  if (EXP.type != intType){error(„integer required“); this.type = errorType; }
  else this.type = noType; }

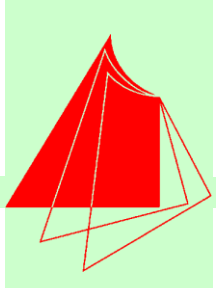
typeCheck(INDEX ::= **[** EXP **]**) {
 if (EXP.type == intType) this.type = arrayType;
 else error(„integer required“); this.type = errorType; }

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) {error(„incompatible types“); this.type = errorType; }
  else if  (EXP2.type==realType && OP_EXP.OP.type==opAand){error(„integer requiered“);this.type=errorType;}
  else if  (OP_EXP.OP.type==opLess || OP_EXP.OP.type==opEqual){this.type=intType;}
  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 || getType(identifier) == realType) && INDEX.type == noType)
            this.type = getType(identifier);
  else if ( getType(identifier) == intArrayType    && INDEX.type == arrayType) this.type = intType;
  else if ( getType(identifier) == realArrayType  && INDEX.type == arrayType) this.type = realType;
  else { error(„no primitive Type“); this.type = errorType; };}

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

typeCheck (EXP2 ::= real ){ this.type = realType ; }

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

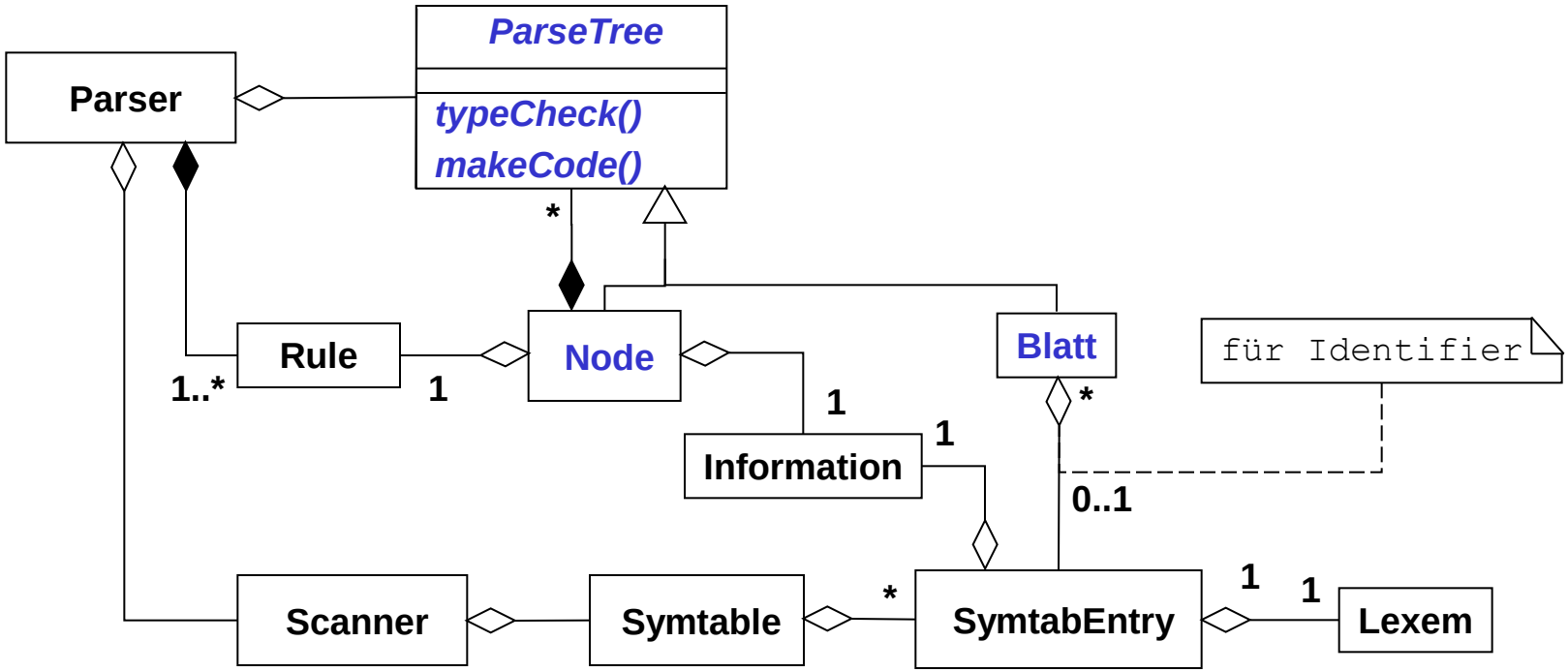


Typisierung und Typ-Check

```
typeCheck (EXP2 ::= ! EXP2){
   typeCheck(EXP2);
   if (EXP2.type != intType){error(„integer required“); this.type = errorType; }
   else this.type = intType; }

typeCheck (EXP2 ::= float EXP2){  typeCheck(EXP2);  this.type = realType;}

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

typeCheck (OP_EXP ::= ε){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 = opEqual; }
typeCheck (OP ::= &){  this.type = opAnd; }
```

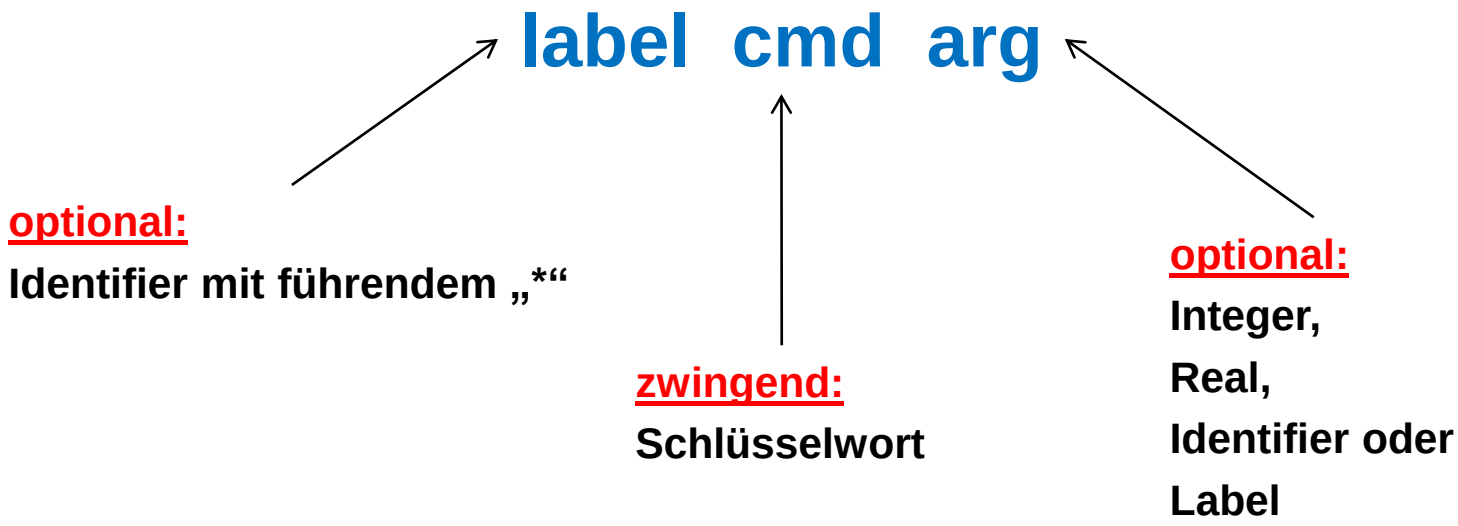
Aufgaben:

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

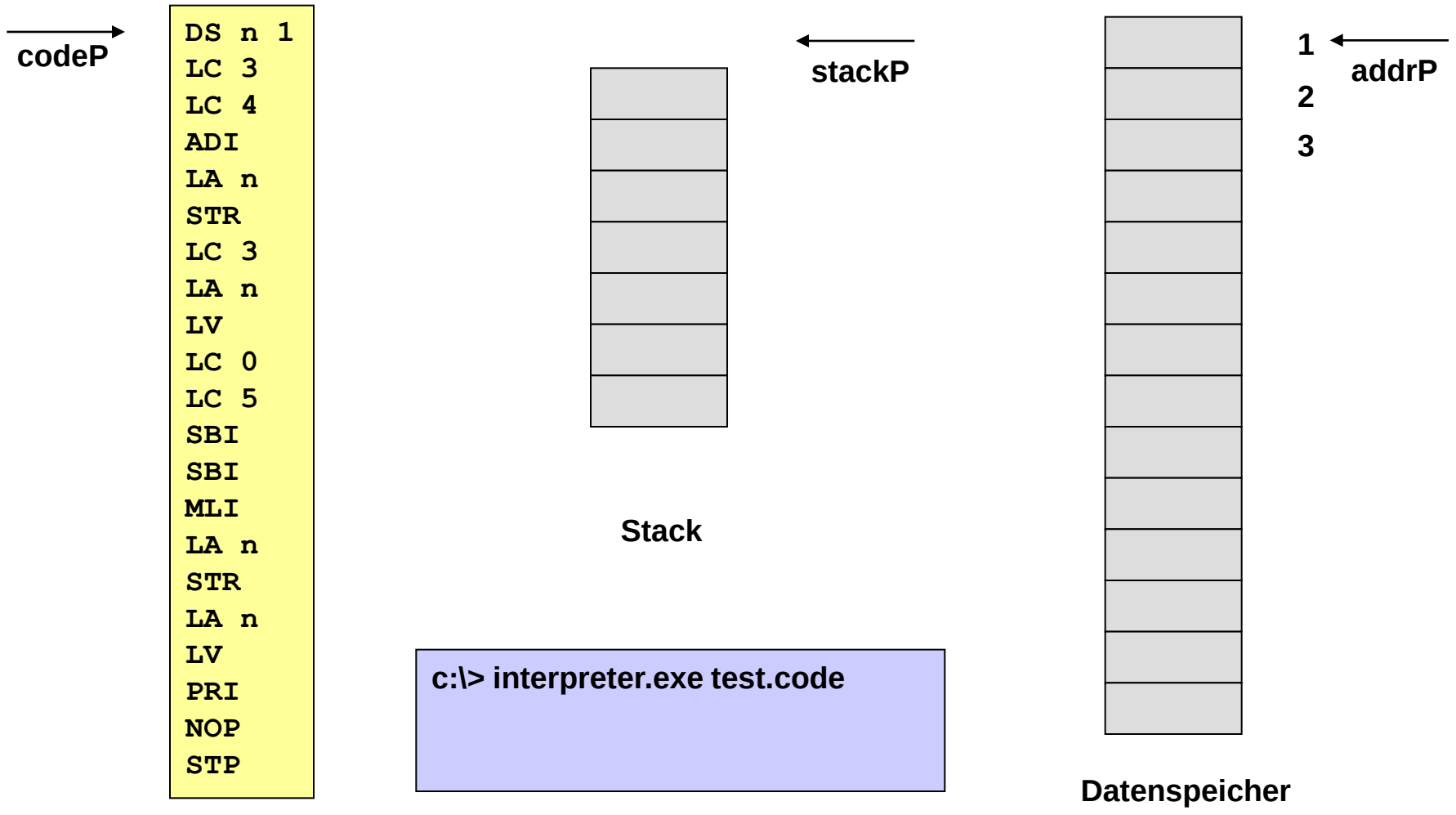
Kommandos mit Label und Argument



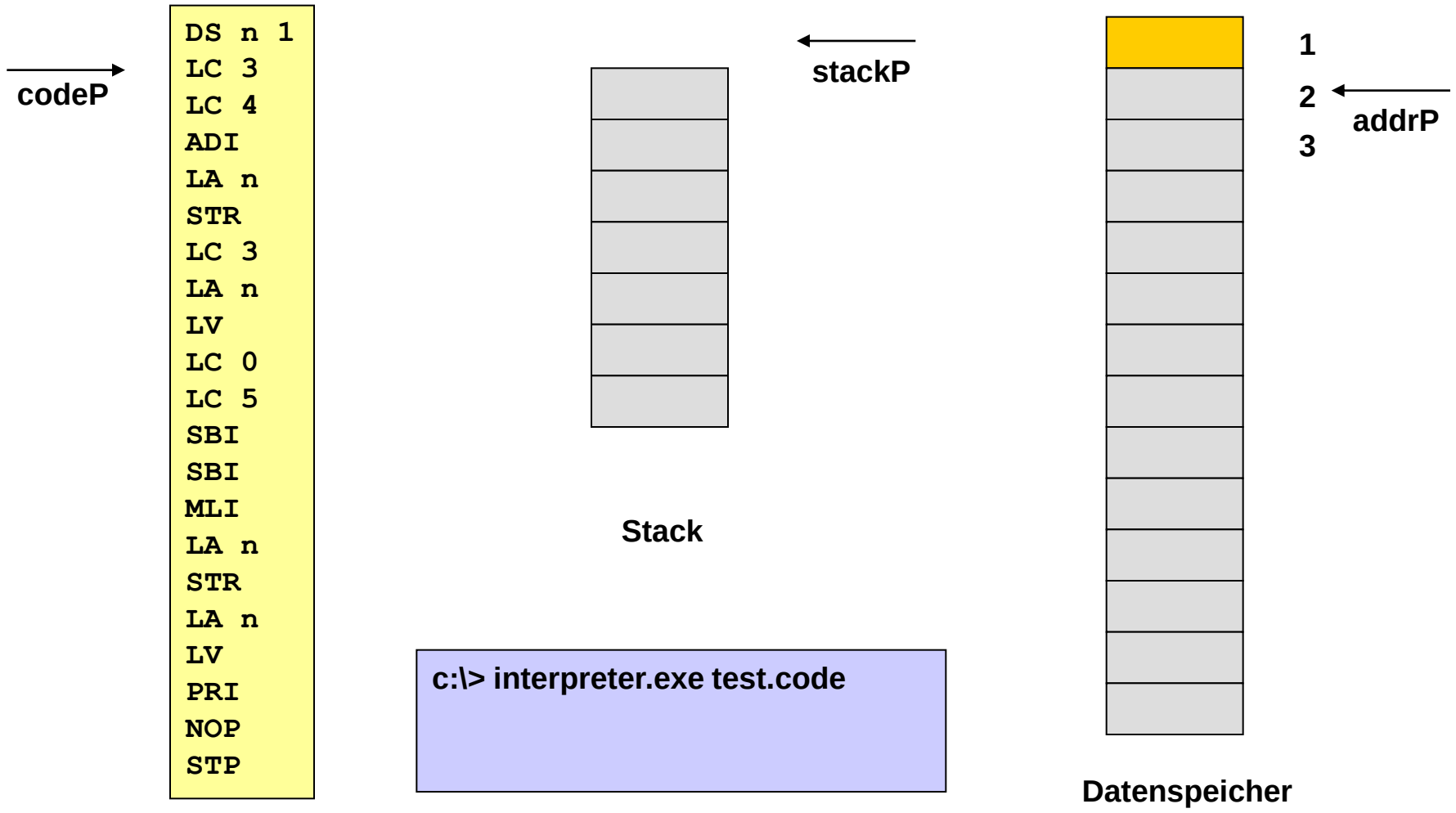
Ausnahme: Kommando zur Speicherreservierung

DS Identifier Integer

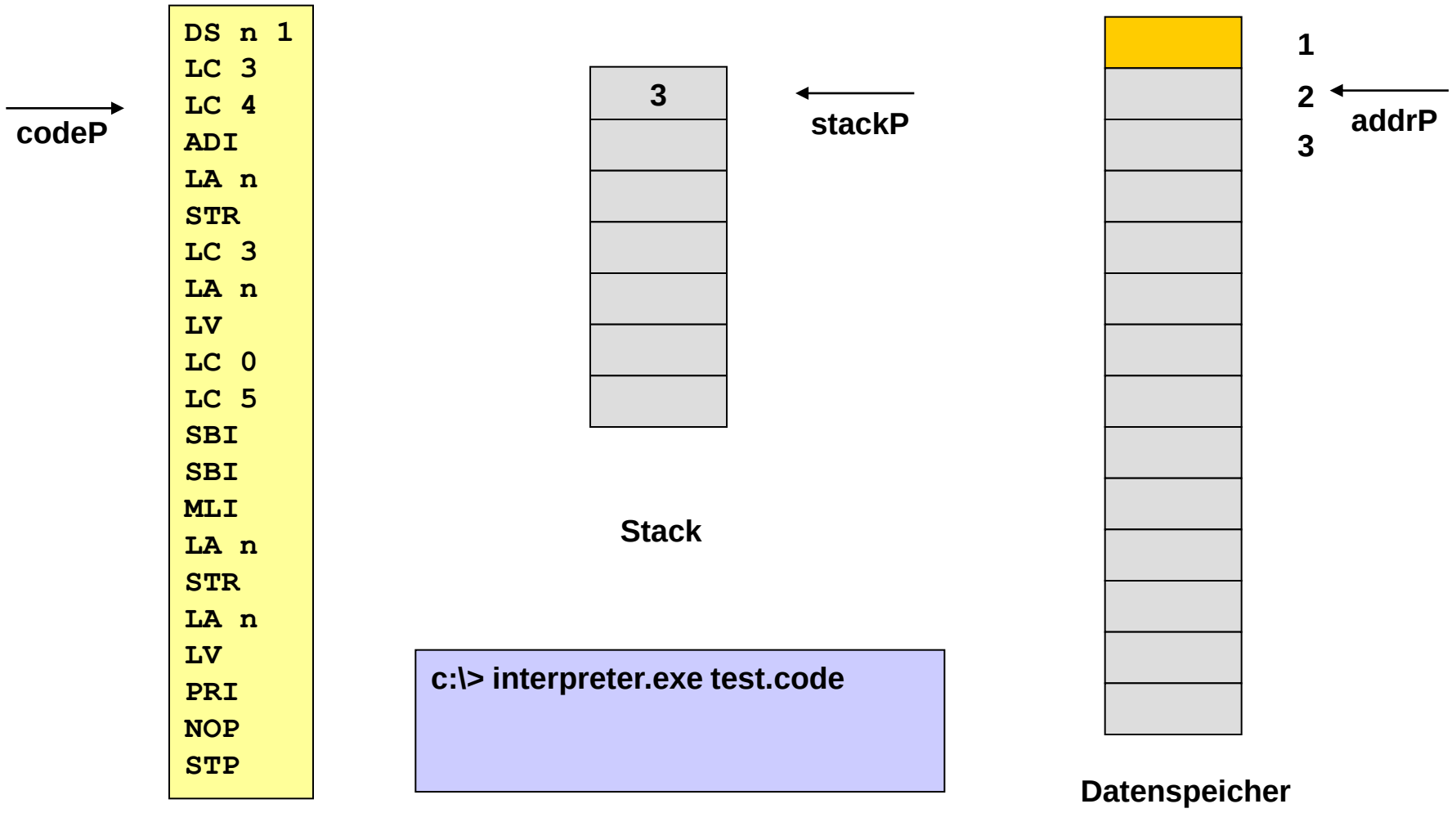
Eine einfache Stack-Maschine



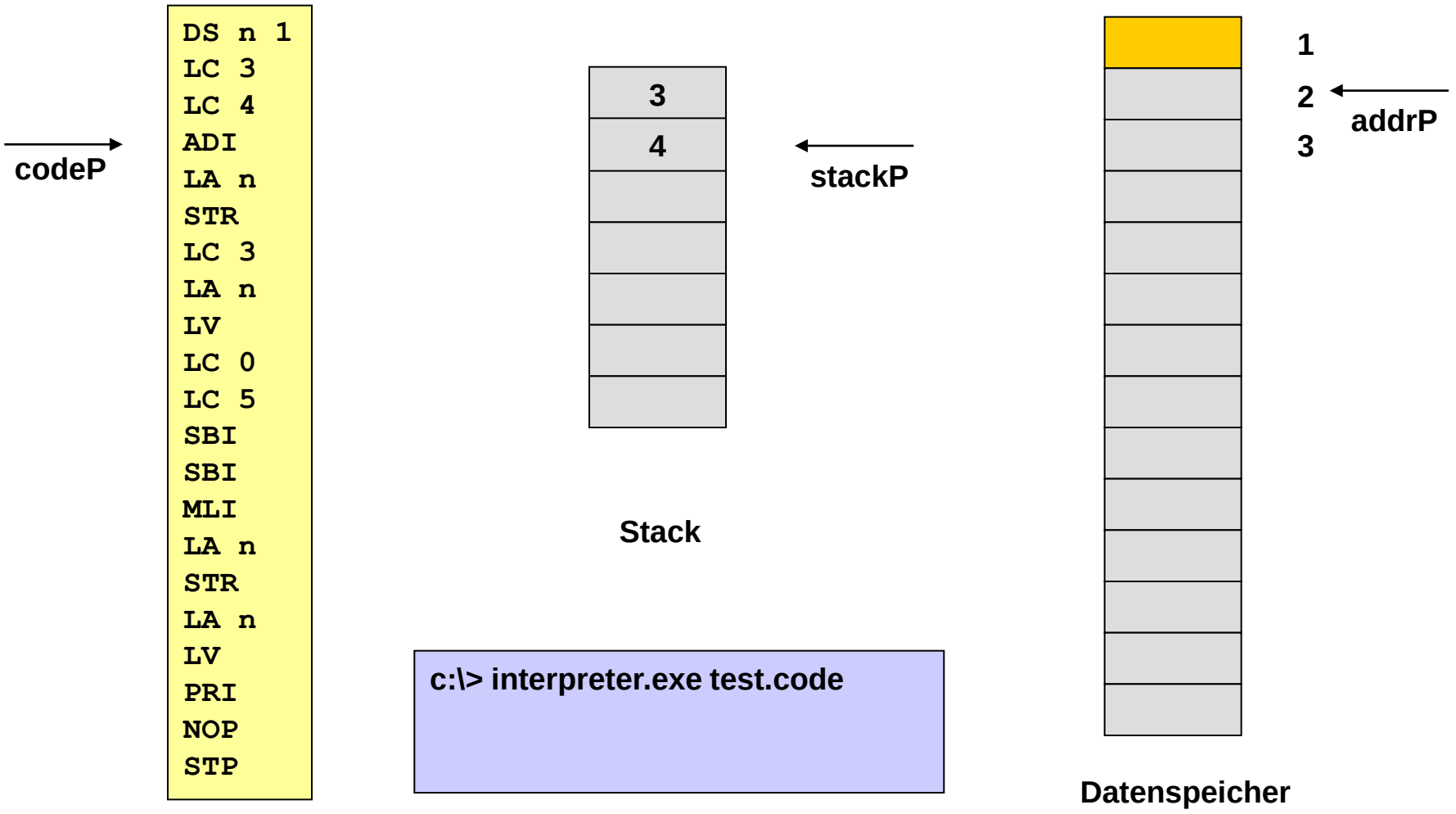
Eine einfache Stack-Maschine



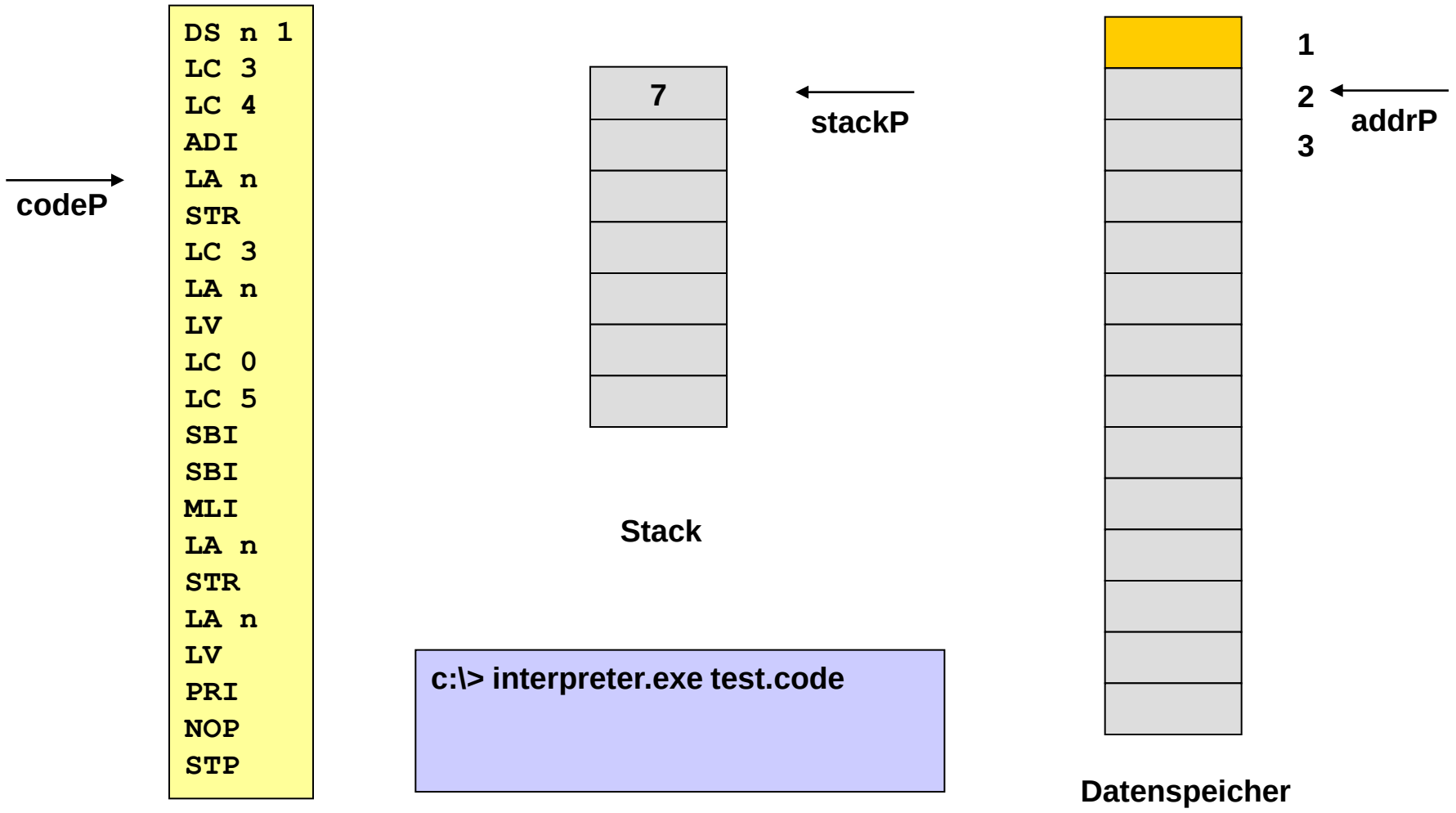
Eine einfache Stack-Maschine



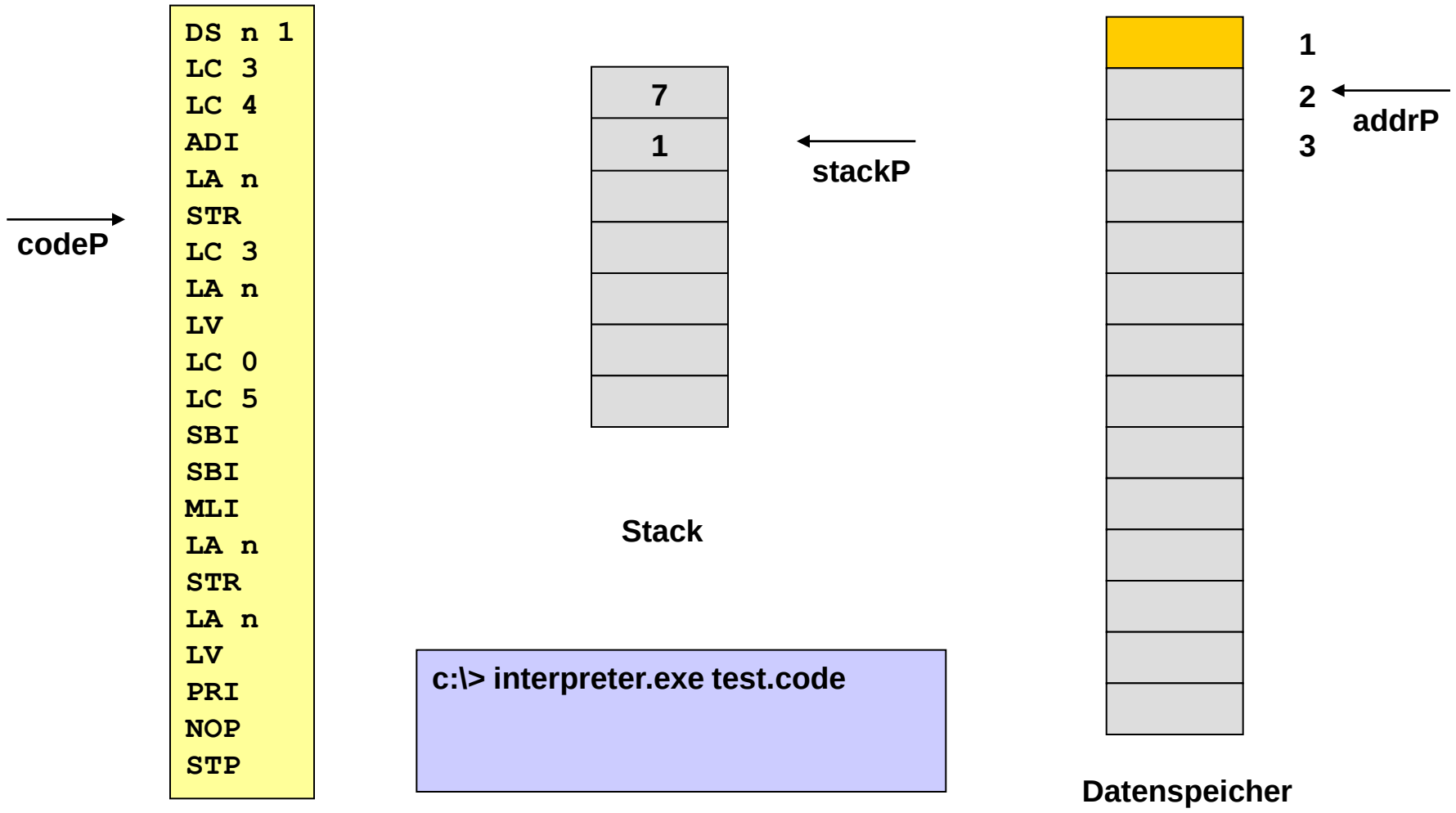
Eine einfache Stack-Maschine



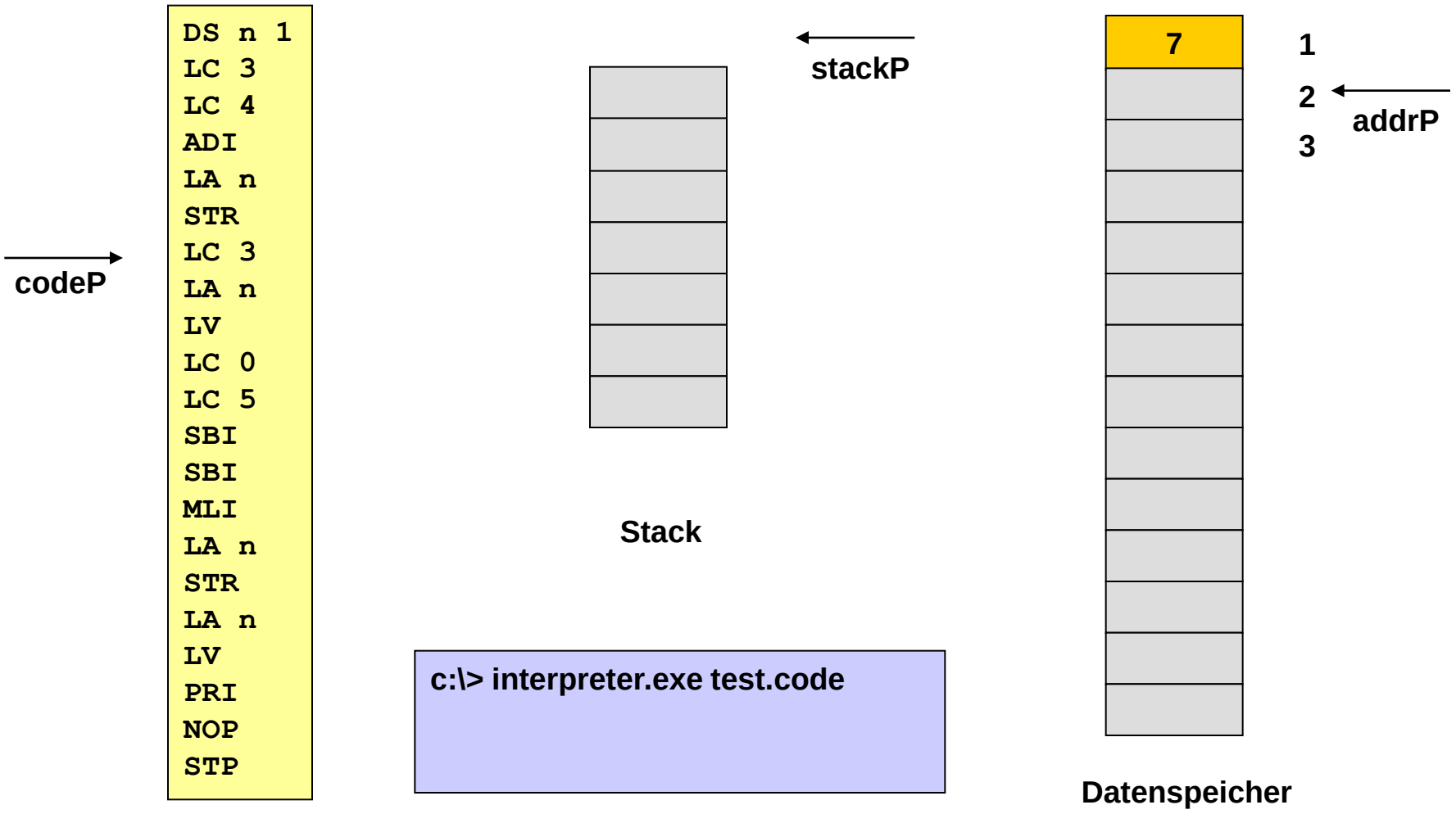
Eine einfache Stack-Maschine



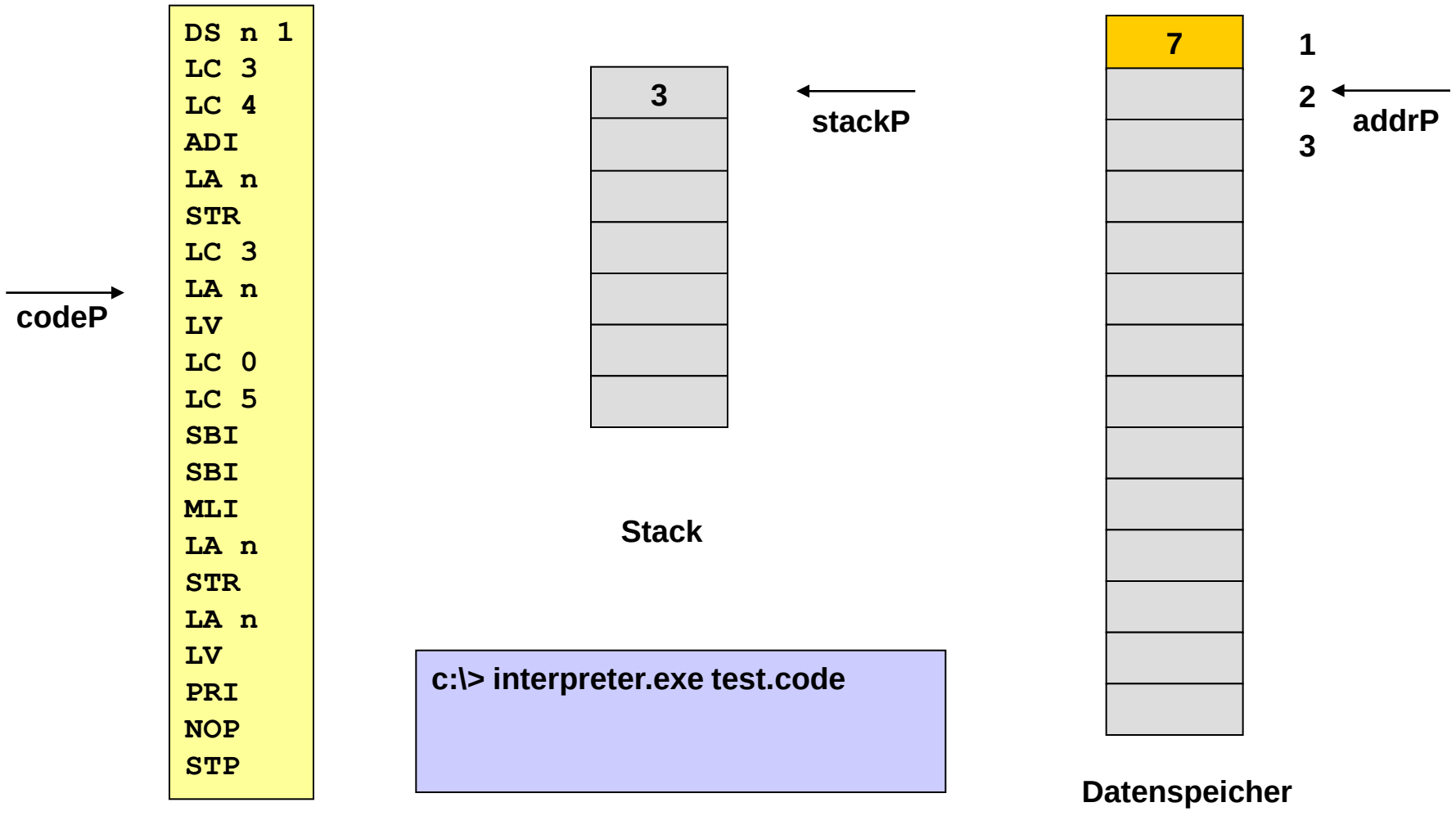
Eine einfache Stack-Maschine



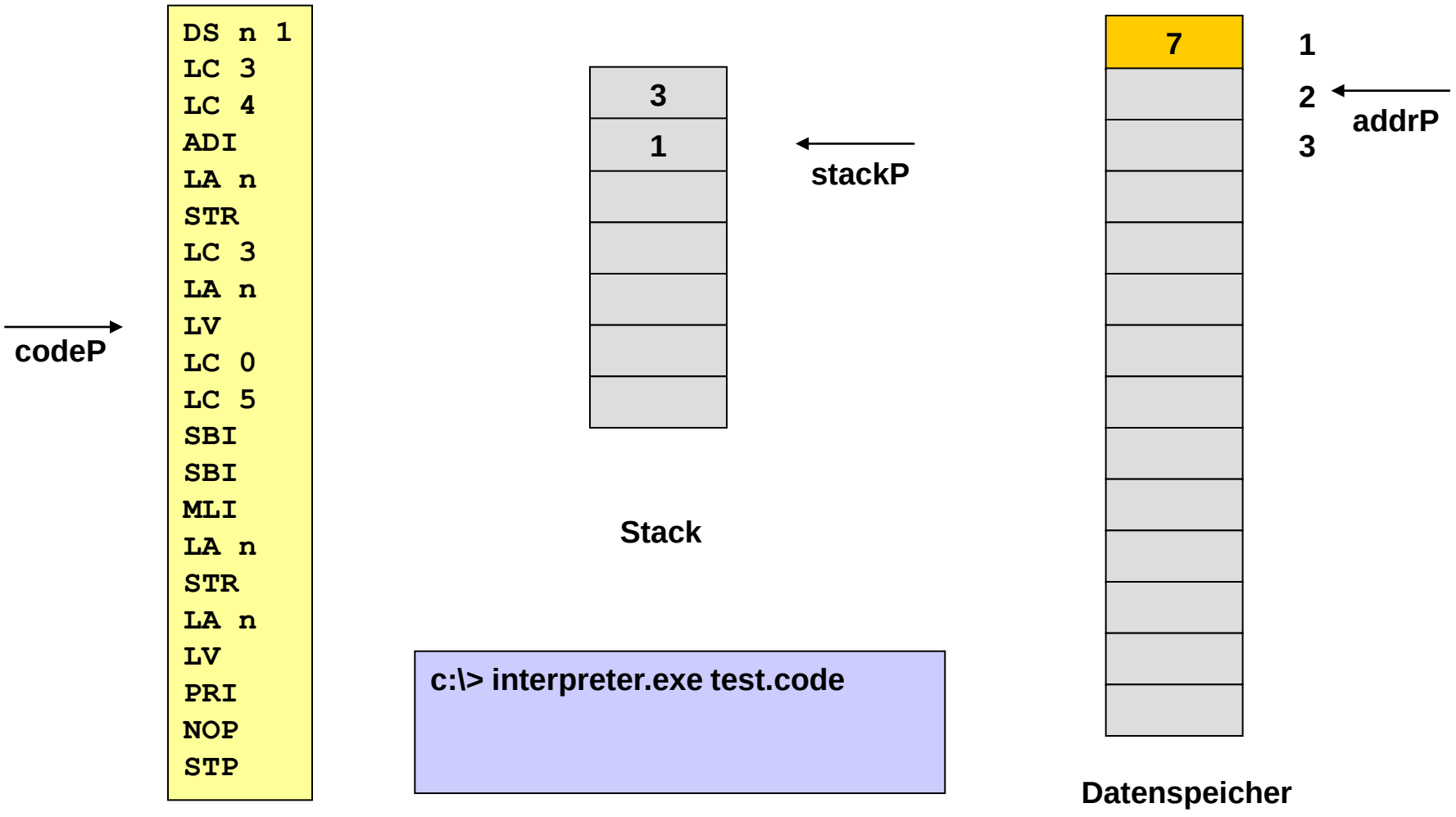
Eine einfache Stack-Maschine



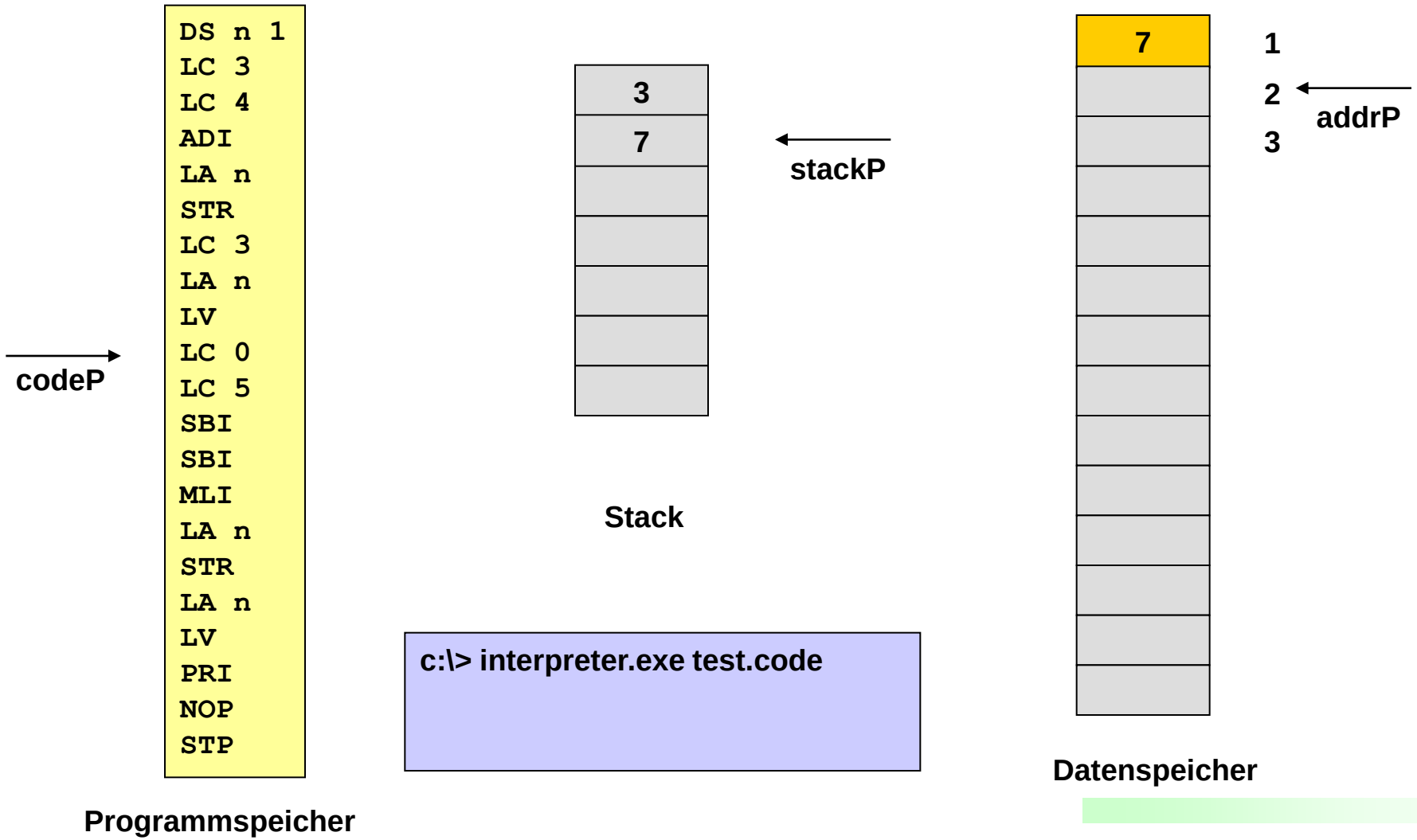
Eine einfache Stack-Maschine



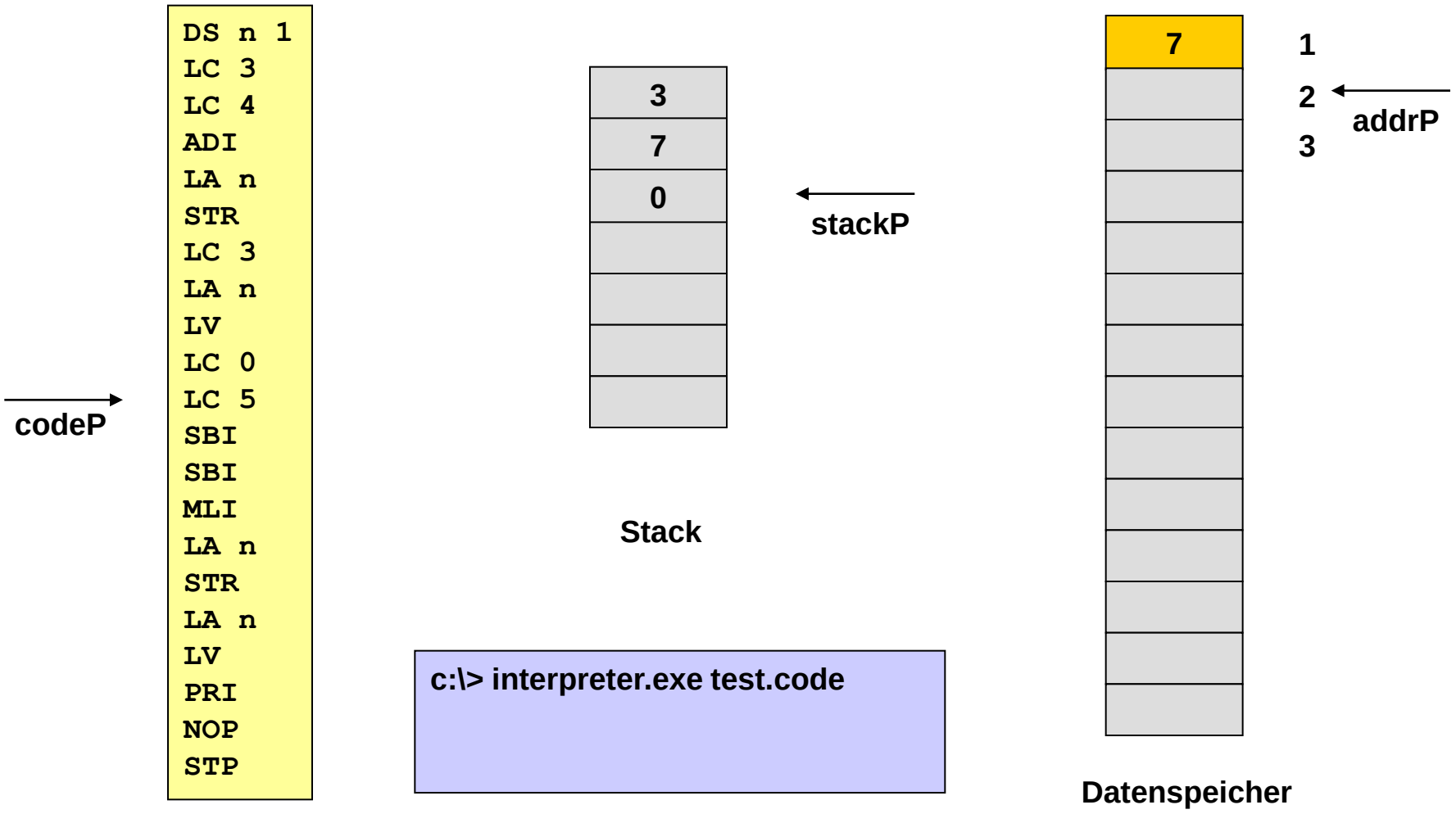
Eine einfache Stack-Maschine



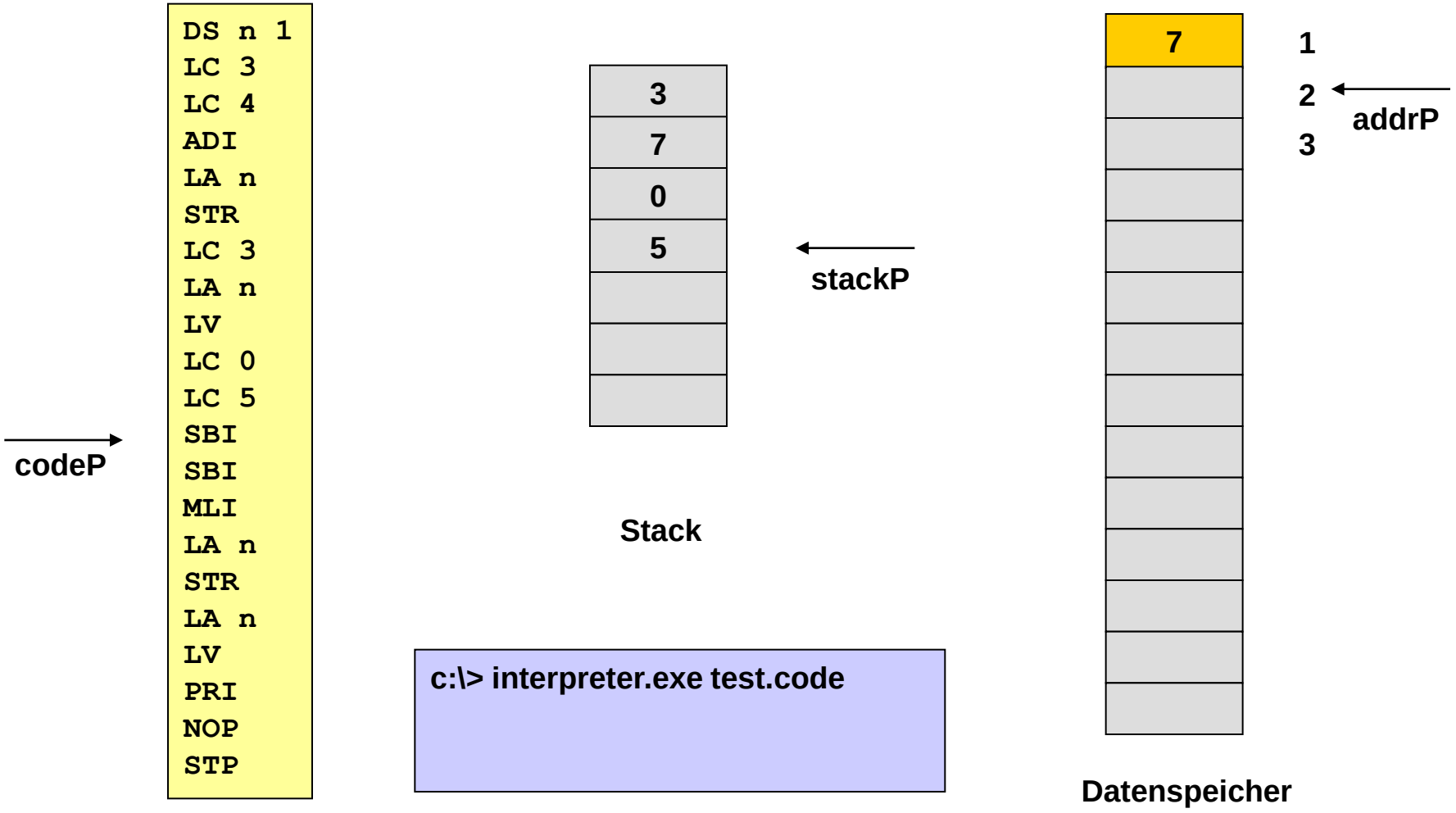
Eine einfache Stack-Maschine



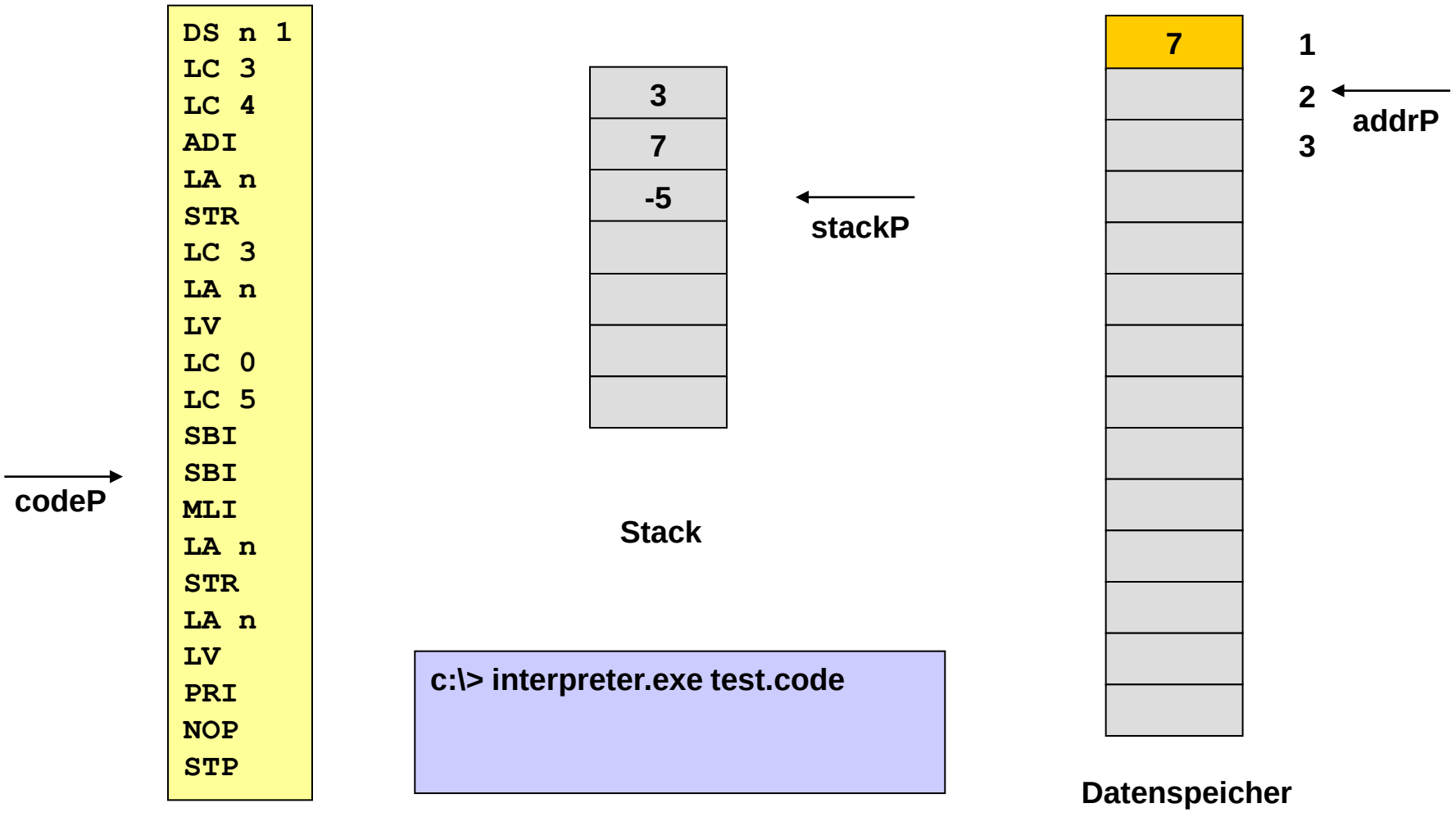
Eine einfache Stack-Maschine



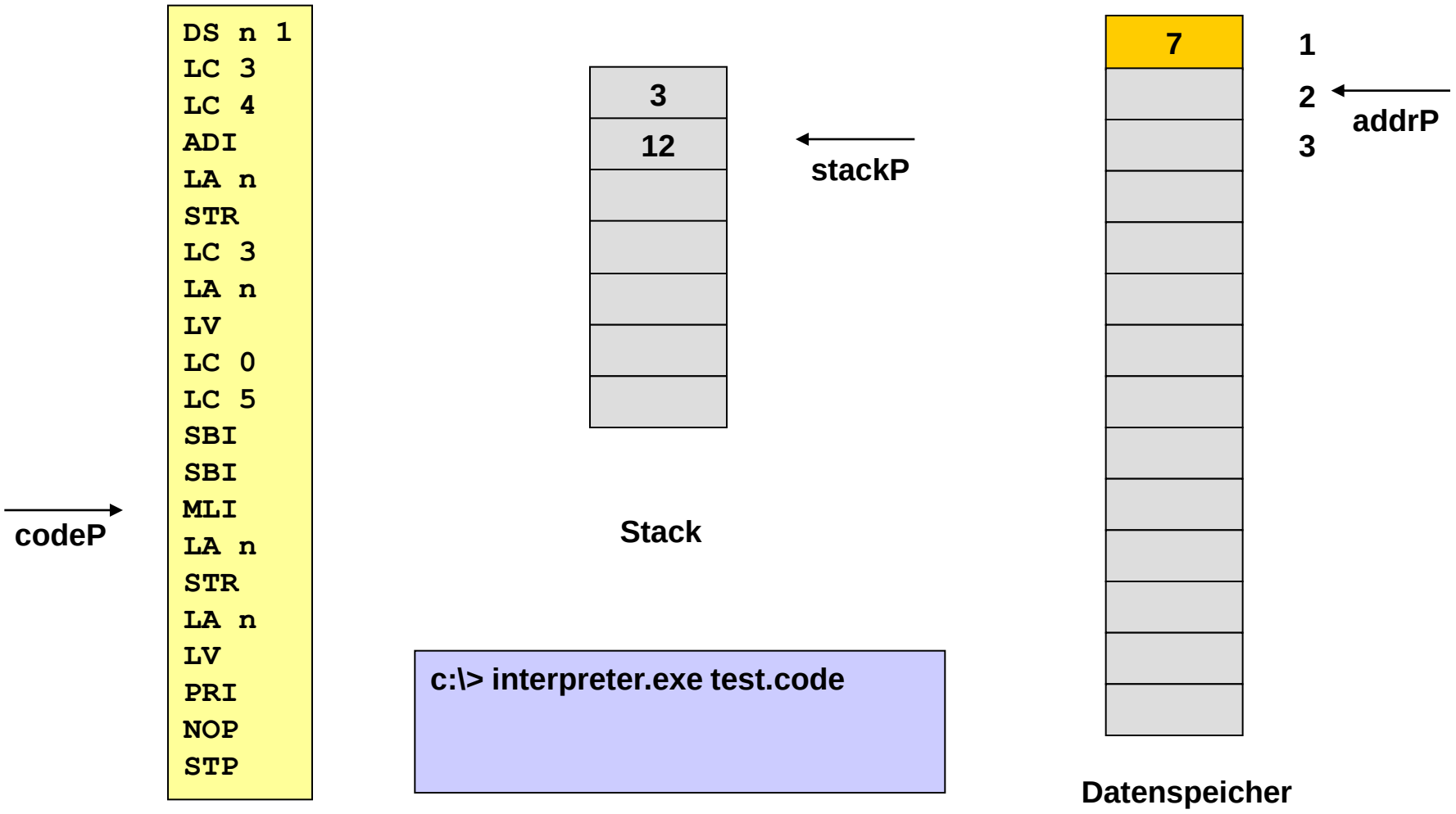
Eine einfache Stack-Maschine



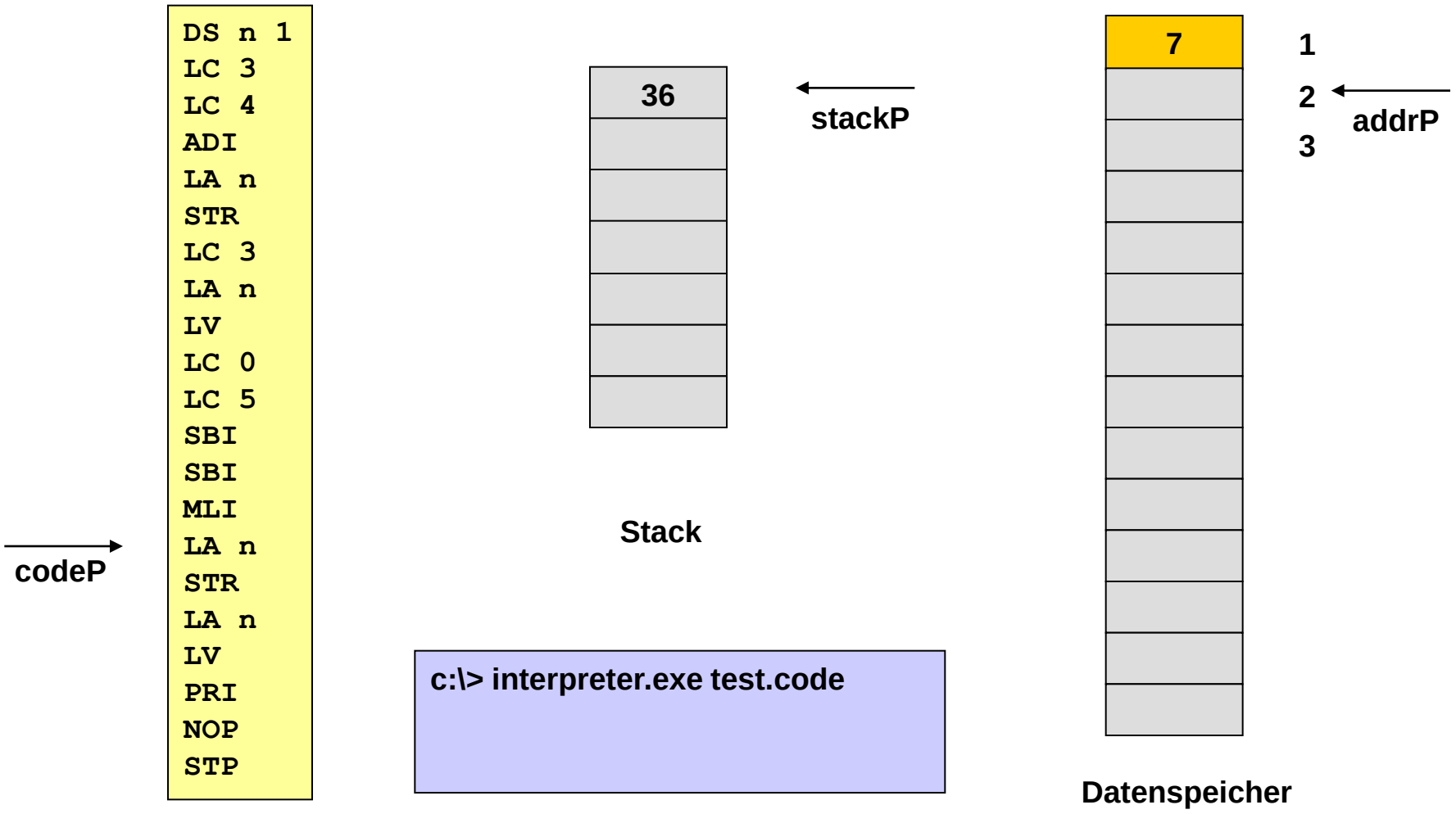
Eine einfache Stack-Maschine



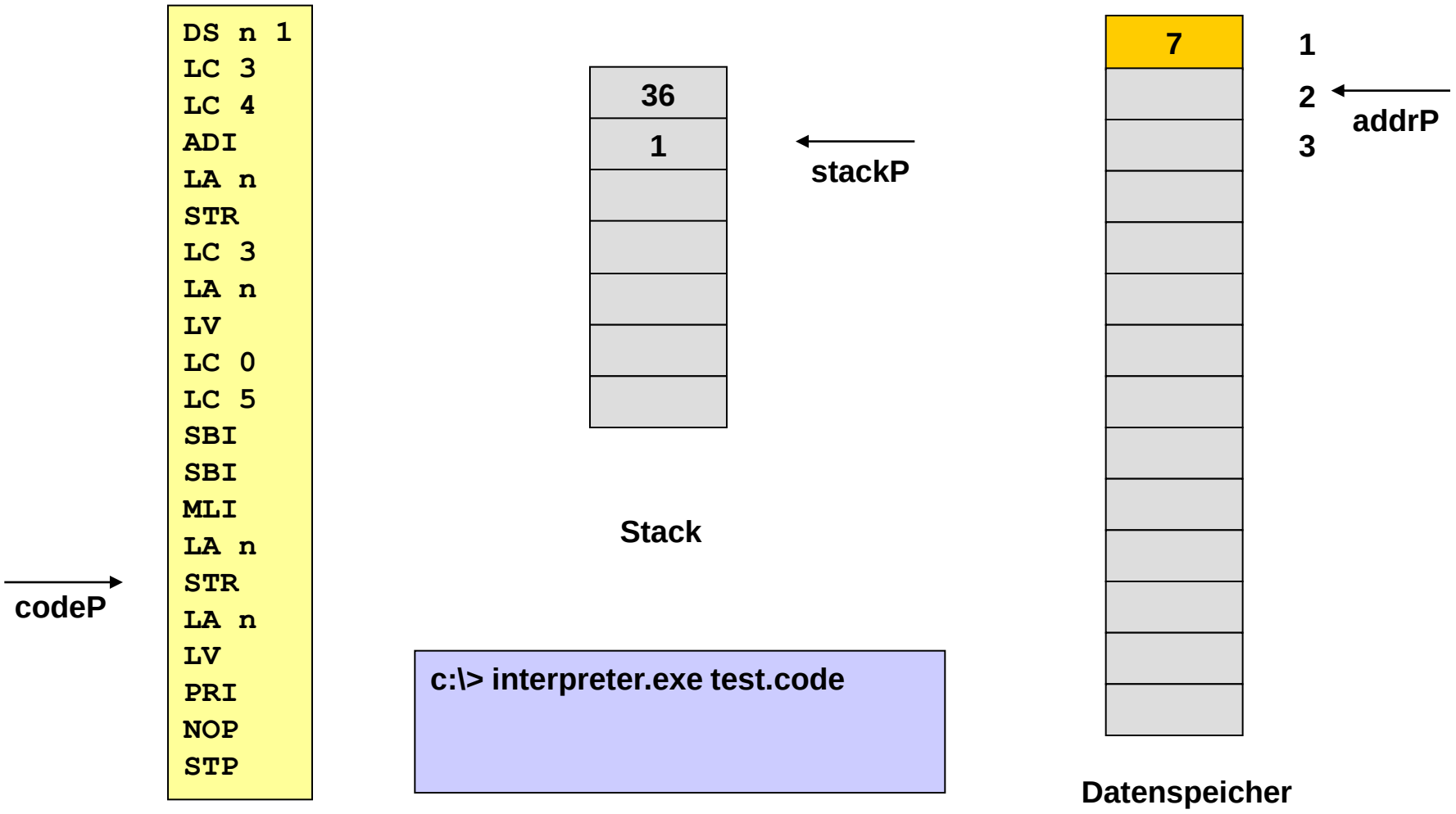
Eine einfache Stack-Maschine



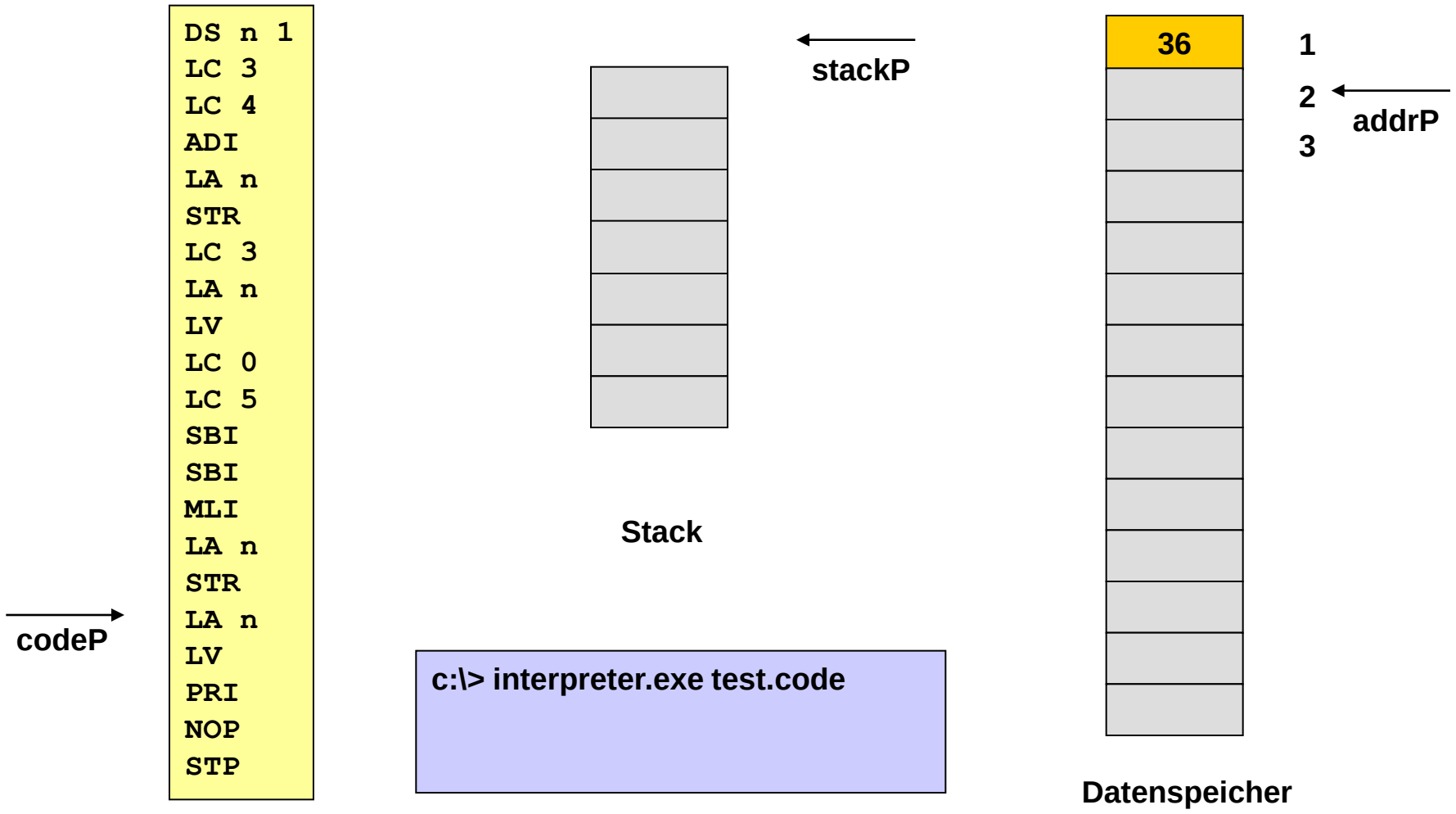
Eine einfache Stack-Maschine



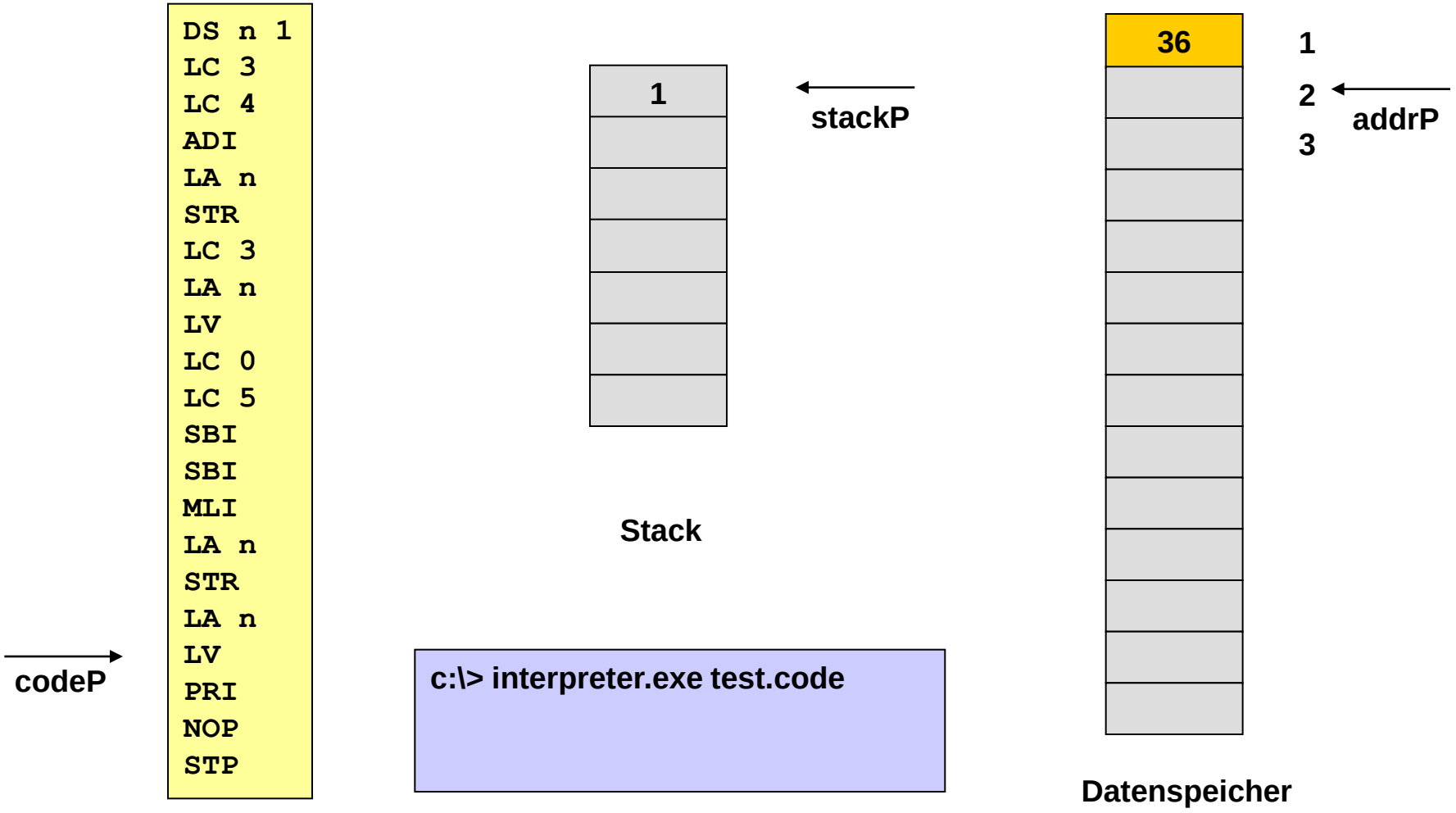
Eine einfache Stack-Maschine



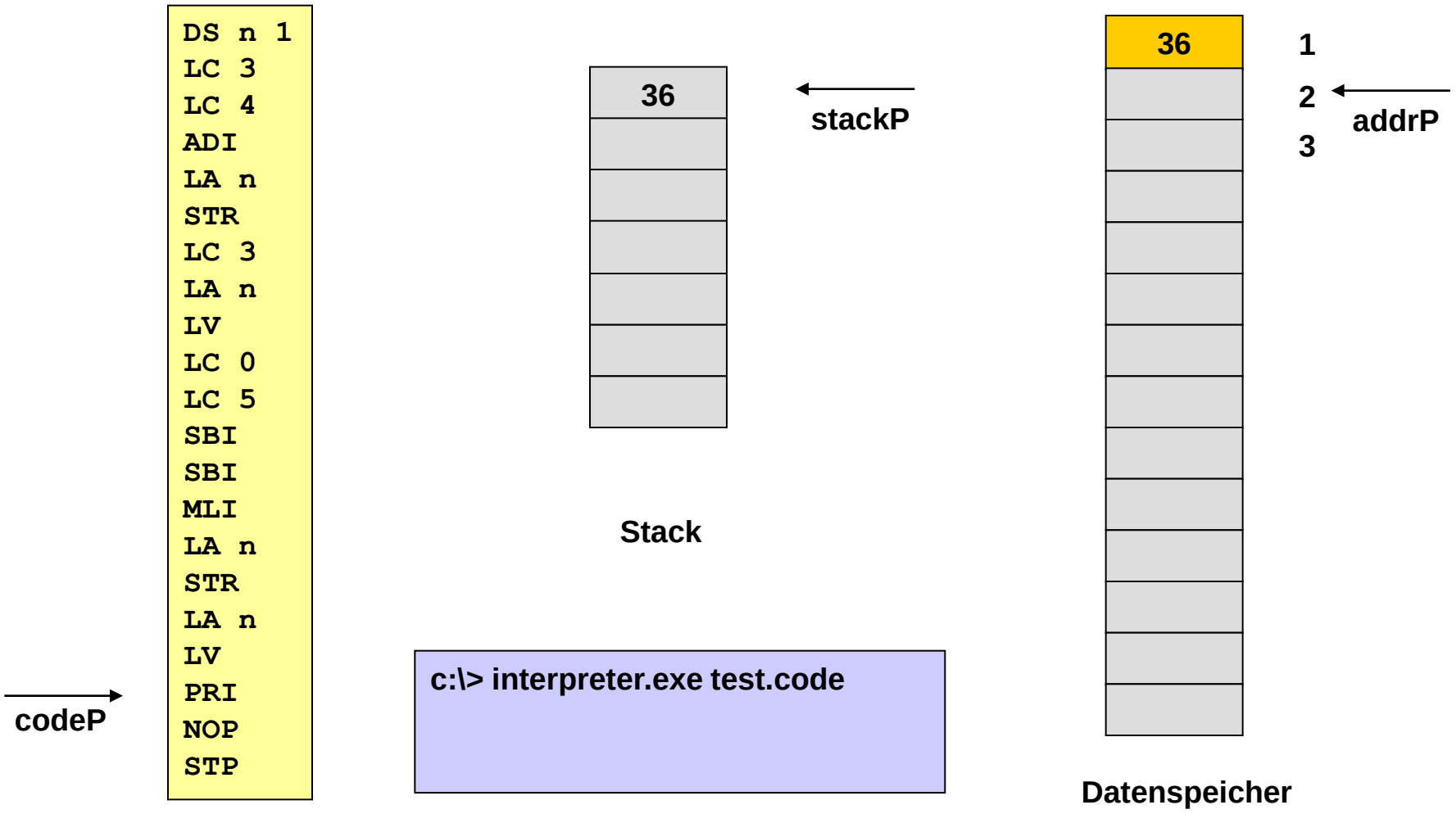
Eine einfache Stack-Maschine



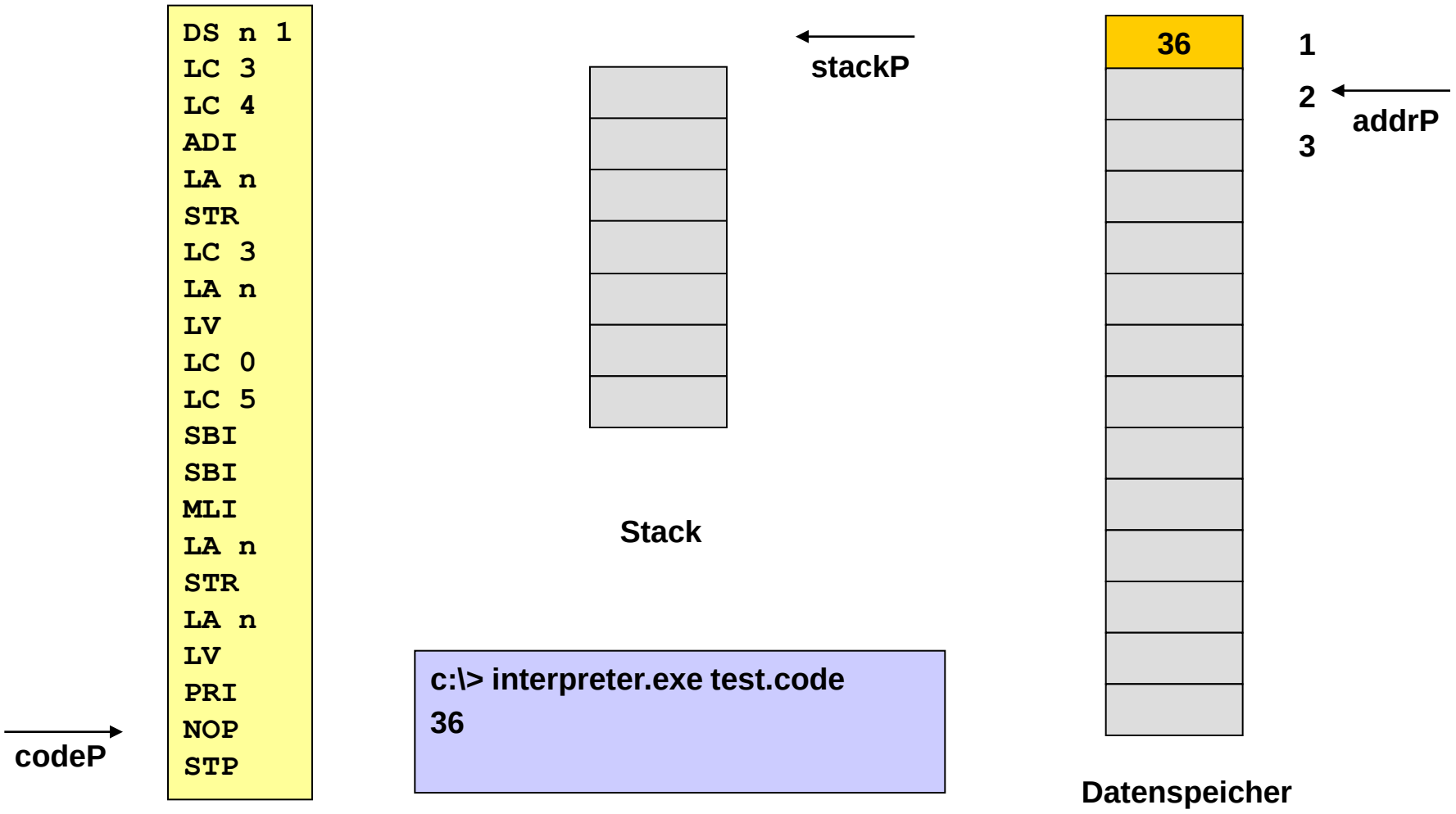
Eine einfache Stack-Maschine



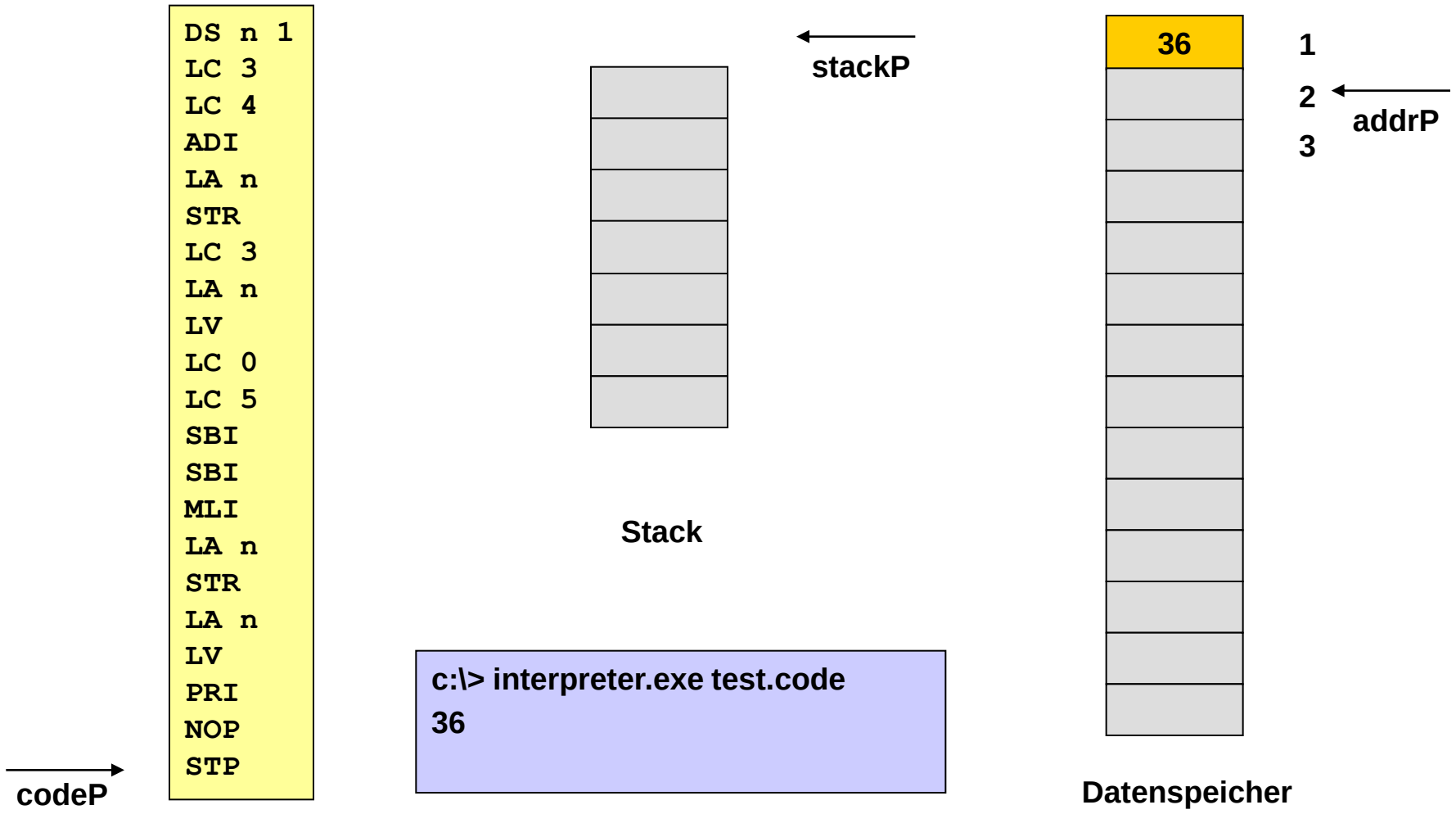
Eine einfache Stack-Maschine



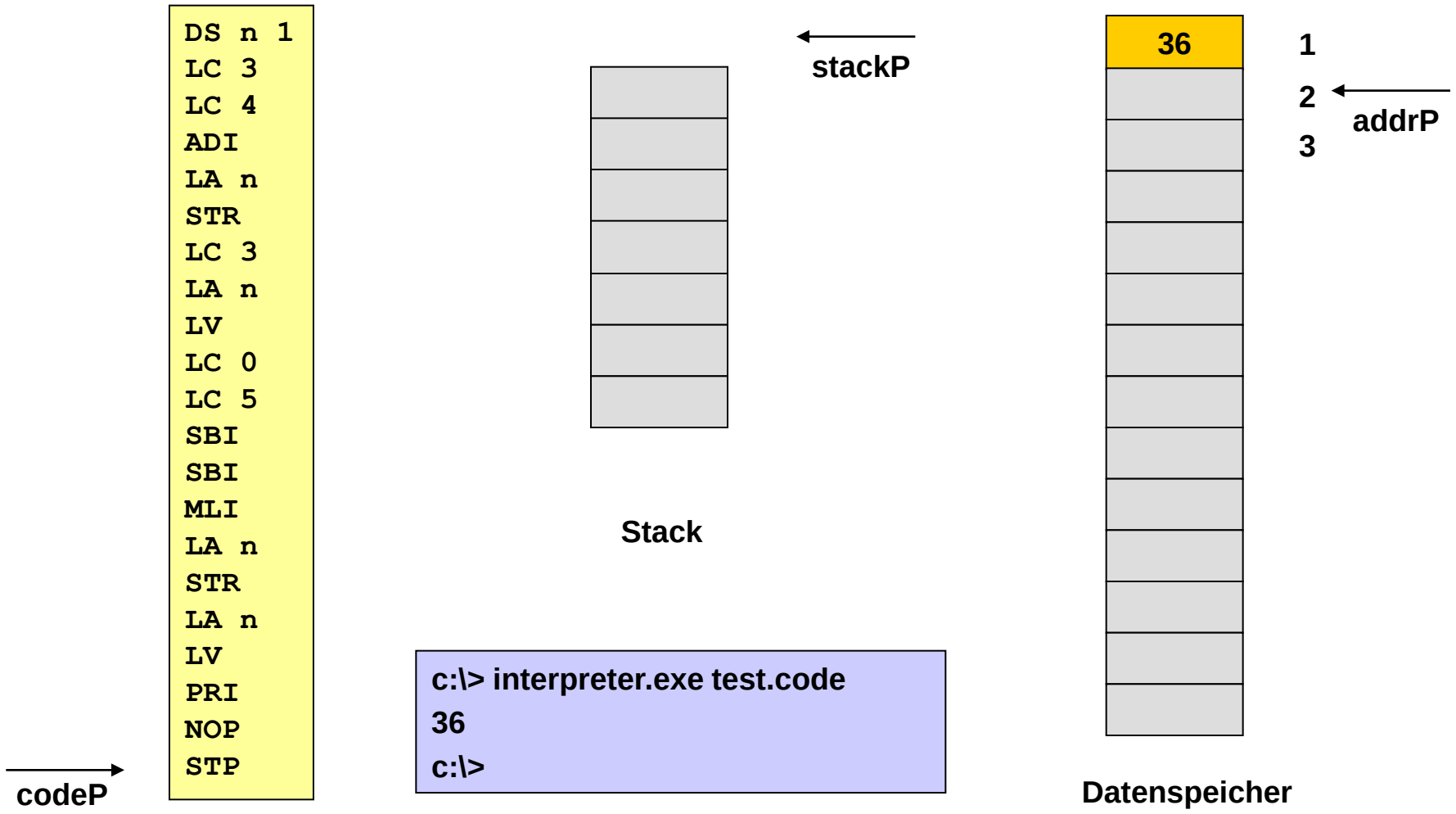
Eine einfache Stack-Maschine

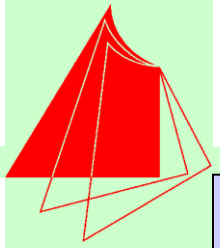


Eine einfache Stack-Maschine



Eine einfache Stack-Maschine





Maschinenbefehle

```
CMD | CMD argument | *label CMD | *label CMD argument
```

- **Arithmetik-Befehle ohne Argument**
  - Addition für Floats und Integer – **ADF** und **ADI**

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

  - Subtraktion für Floats und Integer – **SBF** und **SBI**

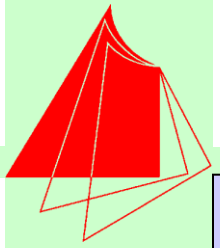
```
codeP++;
*(stackP-1) = *(stackP-1)
              - *stackP;
stackP--;
```

  - Multiplikation für Floats und Integer – **MLF** und **MLI**

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

  - Division für Floats und Integer – **DVF** und **DVI**

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



Maschinenbefehle

```
CMD | CMD argument | *label CMD | *label CMD argument
```

- **Vergleiche ohne Argument**
  - Kleiner für Floats und Integer – **LSF** und **LSI**

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

  - Gleich für Floats und Integer – **EQF** und **EQI**

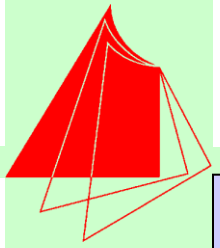
```
codeP++;
if (*(stackP-1) == *stackP)
     *(stackP-1) = 1;
else *(stackP-1) = 0;
stackP--;
```

- **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 identifier`**

```
codeP += 2; stackP++;
*stackP =addr(identifier);
```

  - Laden einer Konstante (int oder real) (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 Floats oder Integers – **`PRF`** und **`PRI`**

```
codeP ++;
println(*stackP);
stackP--;
```

  - Einlesen eines Floats oder Integers – **`RDF`** und **`RDI`**

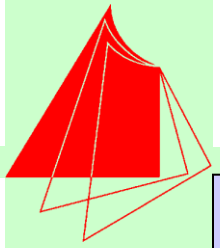
```
codeP ++; stackP++;
*stackP = read();
```

- **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
```

- **Konvertierung ohne Argument**
  - Konvertiert eine Integer in einen Float – **FLT**

```
codeP++;
*stackP == (float) *stackP;
```

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

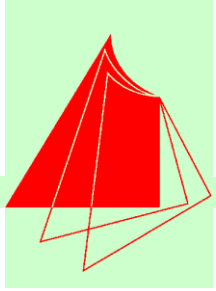
```
codeP+=3;
addr(identifier) = addrP;
addrP += size;
```

- **Was noch fehlt**
  - Nichts tun (ohne Argument) – **NOP**

```
codeP++;
```

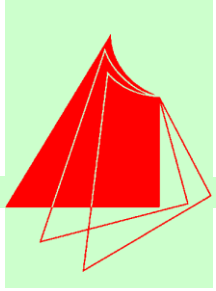
  - Stoppen (ohne Argument) – **STP**

```
exit();
```



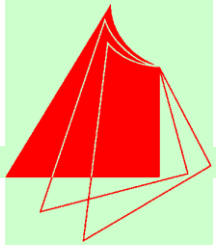
Code-Erzeugung: `makeCode`

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

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

makeCode (DECLS ::= ε){ }

makeCode(DECL::= TYPE ARRAY identifier){
   code << " DS " << getLexem(indentifier); makeCode(ARRAY)}

makeCode(Type ::= int ) {}
makeCode(Type ::= float ) {}
makeCode(ARRAY ::= [ integer ] ) { code << getValue(integer);}
makeCode(ARRAY ::= ε ) {code << 1;}
```



Code-Erzeugung: makeCode

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

makeCode (STATEMENTS ::= ε){code << " NOP ";}

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

makeCode (STATEMENT ::= print( EXP ) ){ makeCode(EXP);
   if (EXP.type == intType) code << " PRI "
   else code << " PRF "; }

makeCode (STATEMENT ::= read( identifier INDEX ) ){
    if (getType(identifier) == intType || getType(identifier) == intArrayType)  code << " RDI ";
   else code << " RDF ";
   code << " LA " << getLexem(identifier); makeCode(INDEX);  code << " STR "; }

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



Code-Erzeugung: makeCode

makeCode (STATEMENT ::= **if (**EXP **)** STATEMENT **else** STATEMENT){

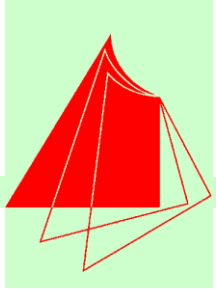
```
  makeCode(EXP);
  code << " JIN " << "*" << marke1; // marke1 ist neu
  makeCode(STATEMENT );
  code << " JMP " << "*" << marke2; // marke2 ist neu
  code << "*" << marke1 << " NOP ";
  makeCode(STATEMENT );
   code << "*" << marke2 << " NOP ";}
```

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

makeCode (STATEMENT ::= **while (**EXP **)** STATEMENT){

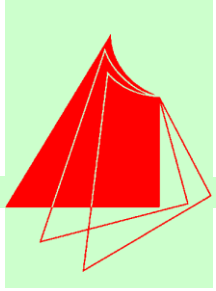
```
 code << "*" << marke1 << " NOP "; // marke1 ist neu
 makeCode(EXP);
 code << " JIN " << "*" << marke2; // marke2 ist neu
 makeCode(STATEMENT );
 code << " JMP " << "*" << marke1;
 code << "*" << marke2 << " NOP ";}
```

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



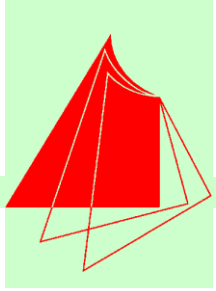
Code-Erzeugung: `makeCode`

```
makeCode (EXP ::= EXP2 OP_EXP ){
  makeCode(EXP2);
  if (OP_EXP.type == noType ) return ;
  makeCode(OP_EXP);
  code <<  // richtigen Operand wählen: ADI, ADF, ….
}

makeCode (INDEX ::= [ EXP ]){makeCode(EXP); code << " ADI ";}
makeCode (INDEX ::= ε){}

makeCode (EXP2 ::= ( EXP )){ makeCode(EXP); }

makeCode (EXP2 ::= identifier INDEX){
     code << " LA  " << getLexem(identifier) ; makeCode(INDEX);  code << " LV ";}

makeCode (EXP2 ::= integer ){ code << " LC  " << getValue(integer);}

makeCode (EXP2 ::= real ){code << " LC  " << getValue(real) ;}
```



Code-Erzeugung: `makeCode`

```
makeCode (EXP2 ::= - EXP2){
  code << " LC  " << 0              // oder 0.0;
  makeCode(EXP2);
  code << " SBI  ";                 // oder SBF
}

makeCode (EXP2 ::= ! EXP2){
  makeCode(EXP2);
  code << " NOT  ";
}

makeCode (EXP2 ::= float EXP2){
   makeCode(EXP2);
   code << " FLT ";
}
```

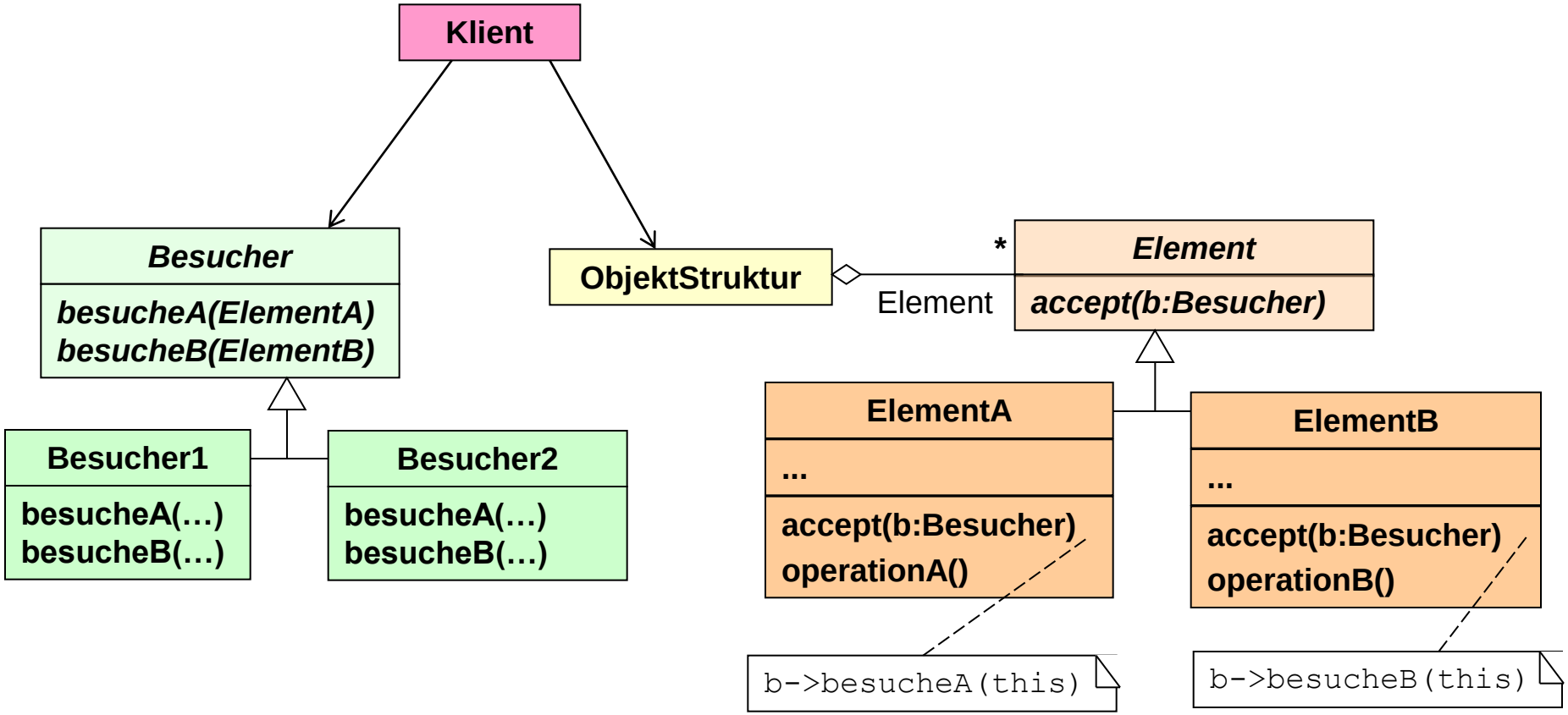


Code-Erzeugung: `makeCode`

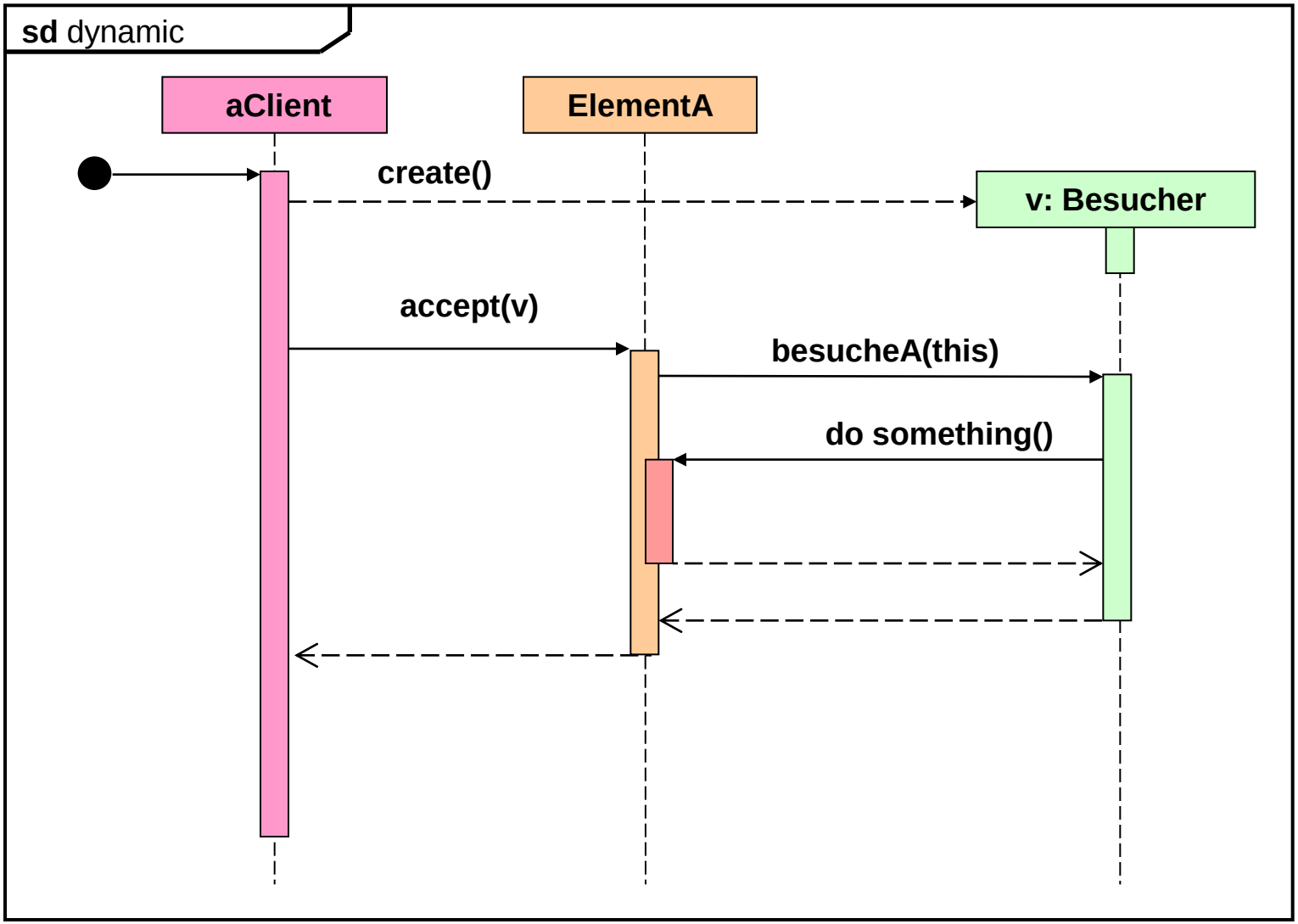
```
makeCode (OP_EXP ::= OP EXP ){
  makeCode(EXP);
  makeCode(OP); }

makeCode (OP_EXP ::= ε){}

makeCode (OP ::= +){ }  //siehe EXP ::= EXP2 OP_EXP
makeCode (OP ::= -){ }
makeCode (OP ::= *){ }
makeCode (OP ::= /){ }
makeCode (OP ::= <){ }
makeCode (OP ::= =){ }
makeCode (OP ::= &){ }
```

Alternativen:

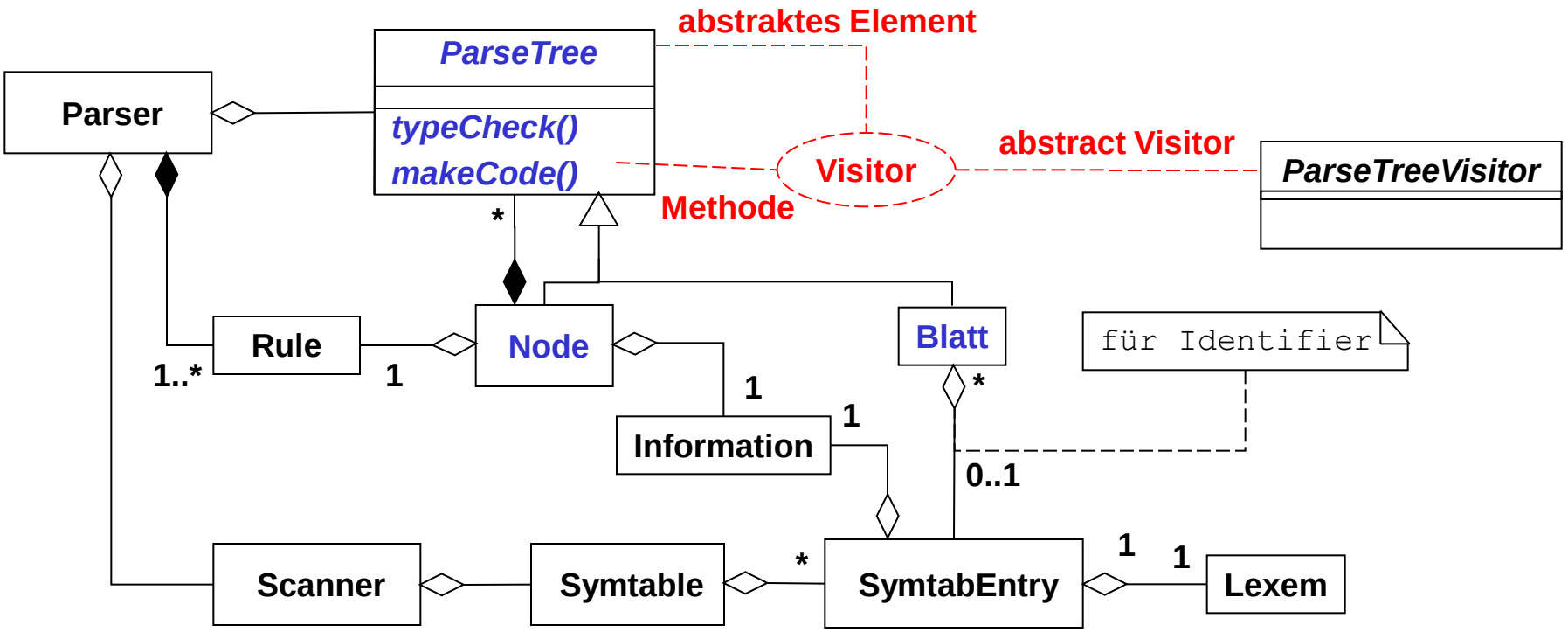
Auskopplung der Funktionalität über einen Besucher:



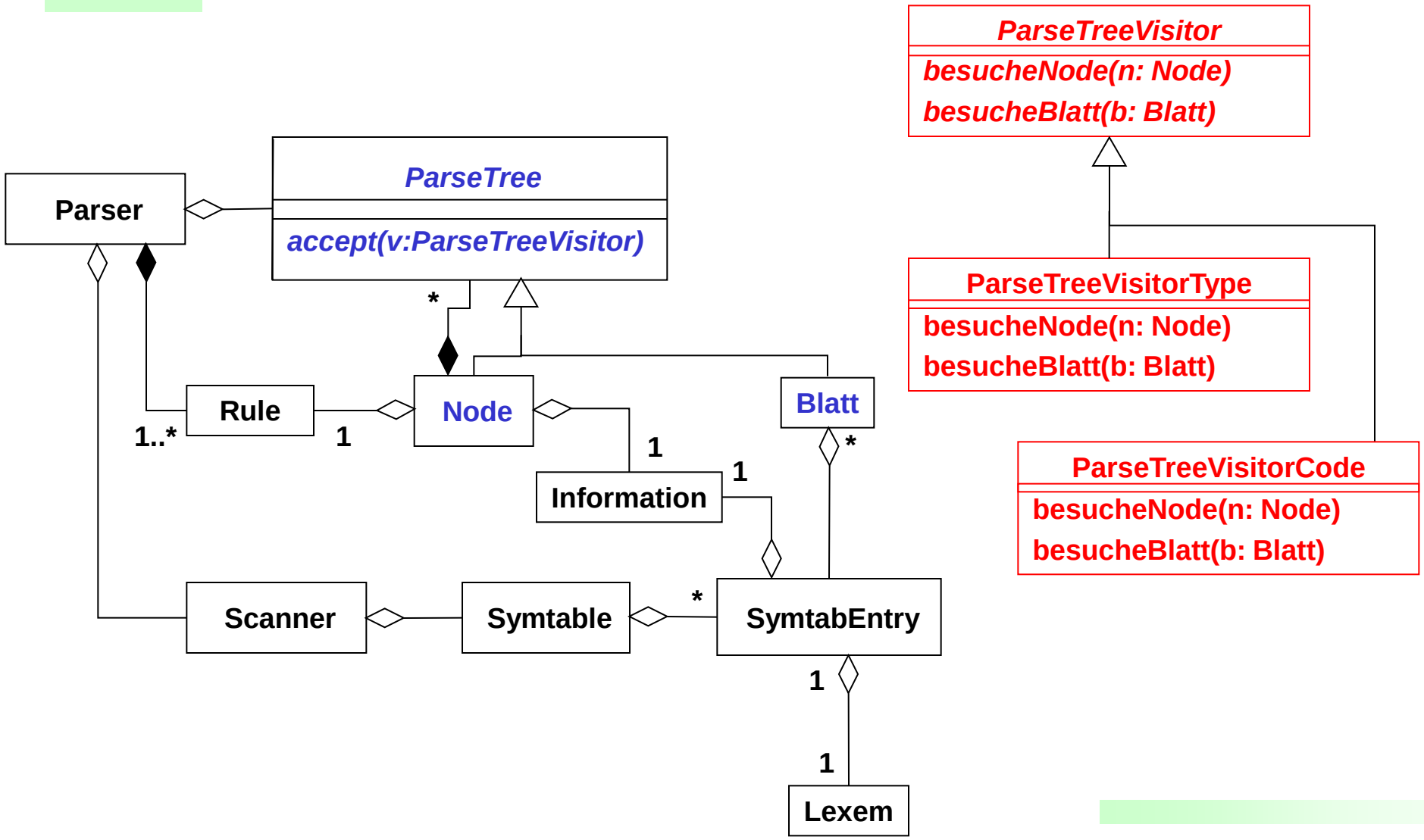
Dynamisches Verhalten

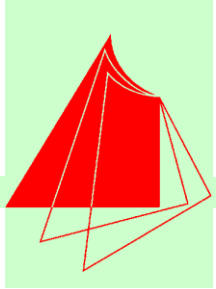


Zuordnungen



Das erwartete Ziel





Programmaufruf (I)

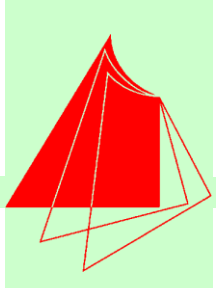
```
C:\> parser Parser-test.txt -c test.code
```

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

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

„Parser-test.txt “

```
C:\>parser parser-error.txt
parsing …
type check …
generate code …
```



Programmaufruf (II)

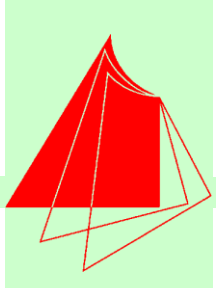
Die resultierende Code-Datei hat dann etwa folgende Gestalt:

```
DS n
LC 3
LC 4
ADI
LA n
STR
LC 3
LA n
LV
LC 0
LC 5
SBI
SBI
MLI
LA n
STR
LA n
LV
PRI
NOP
STP
```

„test.code “

Um Code-Files zu interpretieren verwenden Sie den „Interpreter“

```
C:\>interpreter.exe test.code
36
C:\>
```



Programmaufruf (III)

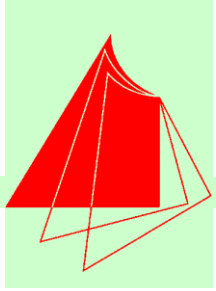
```
C:\> parser Parser-error.txt -c test.code
```

- „**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:     23   Column:     12  TokenRightParent
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 + 4.5;
...
```

„Parser-error.txt “

```
C:\>parser parser-error.txt
parsing …
type check …
error Line: 25 Column:  8  incompatible types
stop
C:\>
```