



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

# **Labor**

## **Systemnahes Programmieren**

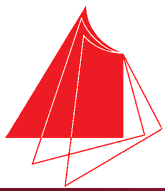
Prof. Dr. Oliver P. Waldhorst

Hochschule Karlsruhe – Technik und Wirtschaft

Fakultät für Informatik und Wirtschaftsinformatik

(basierend auf Materialien von Prof. Th. Fuchß)





# Übersicht

- **Veranstaltungen**
  - jeweils mittwochs von 14.00 (11.30) – 18.30 (17.00) Uhr (Li137)
- **Zeitplan**
  - Phase I (Scanner) 7 Termine (15.03.17 – 26.04.17)
  - Phase II (Parser) 8 Termine (03.05.17 – 28.06.17)
  - **Letzte Möglichkeit zur Abgabe ist Mittwoch, 28.06.17!**
- **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 I Lexikalische Analyse

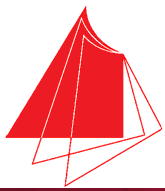
Prof. Dr. Oliver P. Waldhorst

Hochschule Karlsruhe – Technik und Wirtschaft

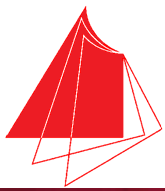
Fakultät für Informatik und Wirtschaftsinformatik

(basierend auf Materialien von Prof. Th. Fuchß)





- 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
- D. Grune und C. Jacobs.  
Parsing Techniques – A Practical Guide, 1990.  
([http://www.dickgrune.com/Books/PTAPG\\_1st\\_Edition/](http://www.dickgrune.com/Books/PTAPG_1st_Edition/))
- 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/))



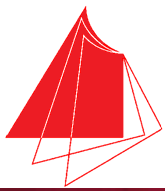
# Ziel:

**Ziel der ersten Laboraufgabe ist es, die Funktionsweise eines Scanners sowie dessen Einordnung innerhalb eines Compilers kennenzulernen.**

**Des Weiteren soll die Implementierung eines Scanners das Verständnis für dynamische Datenstrukturen und Zeiger (in C/C++) vertiefen.**

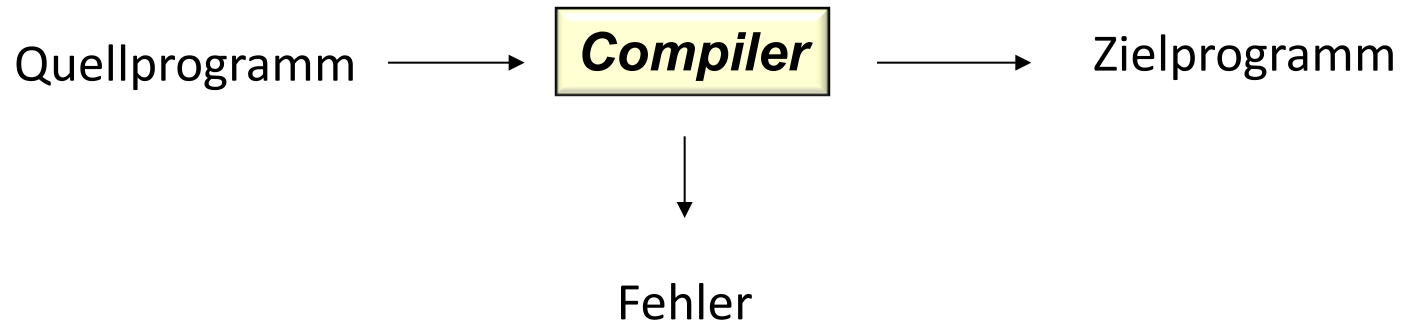


**kein Java !**

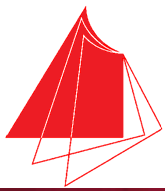


# Einführung: Was ist ein Compiler

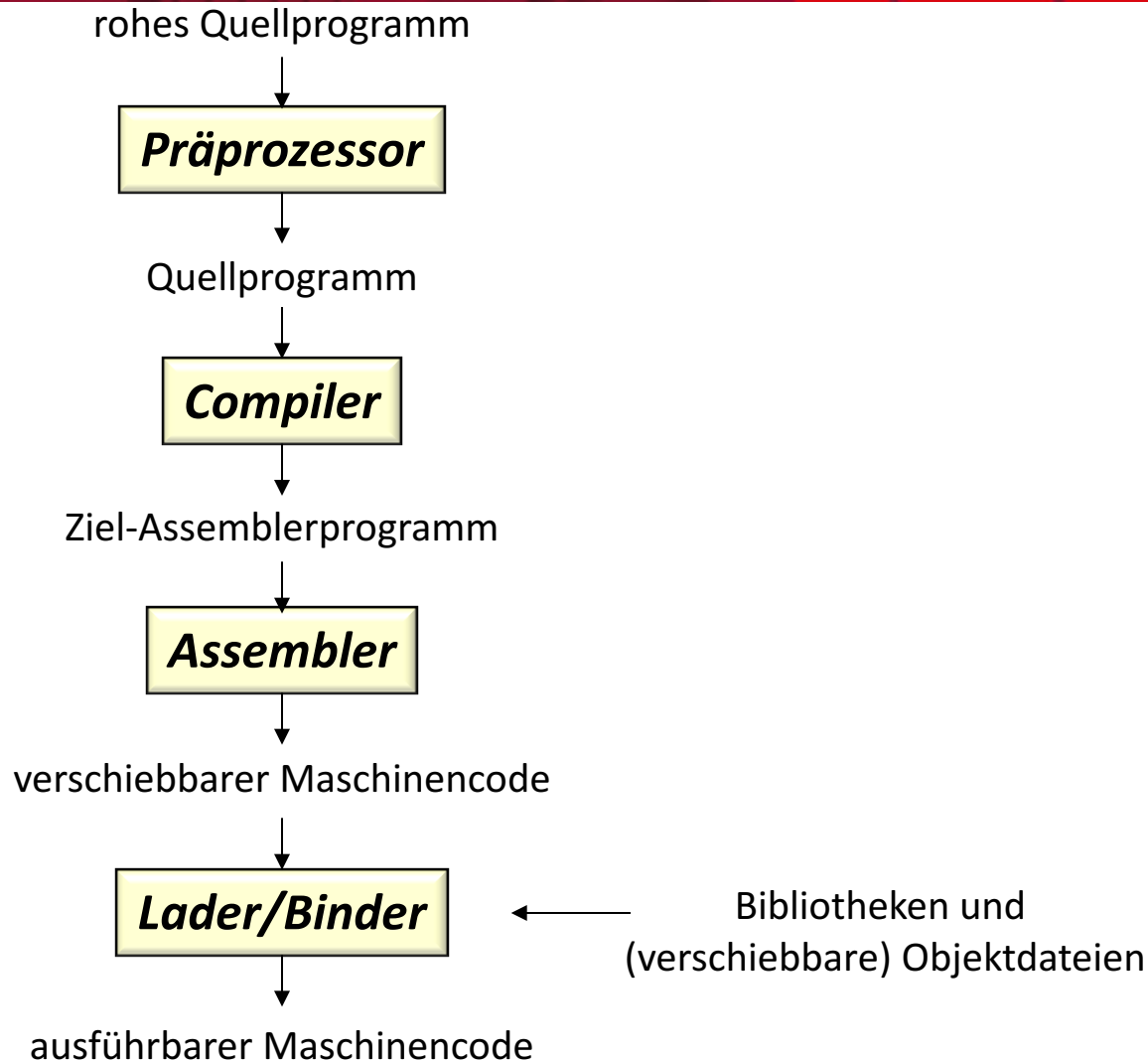
Ein Compiler ist ein Programm, das ein Programm einer bestimmten Sprache (Quellsprache) in ein äquivalentes Programm einer anderen Sprache (Zielsprache) übersetzt.

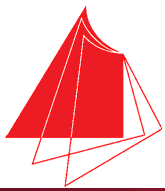


- C / C++ ➡ Maschinensprache
- Java ➡ Bytecode
- LaTeX ➡ dvi, pdf, ps
- XML ➡ (interne) Datenstrukturen
- ...



# Die Umgebung eines Compilers





# Das Analyse-Synthesemodell

**Der Übersetzungsprozess besteht aus zwei Teilen:**

- **Analyse: (Frontend)**

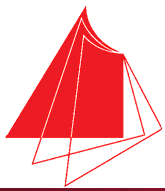
Der Analyse-Teil zerlegt das Quellprogramm in seine Bestandteile und erzeugt eine Zwischendarstellung

- **Synthese: (Backend)**

Der Synthese-Teil konstruiert das gewünschte Zielprogramm aus der Zwischendarstellung

- Code-Erzeugung
- Optimierung

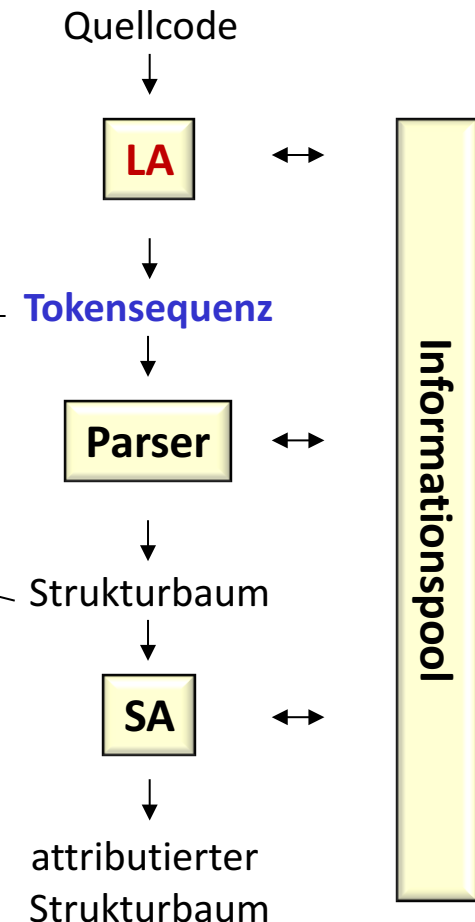


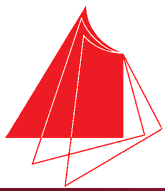


# Die Analyse

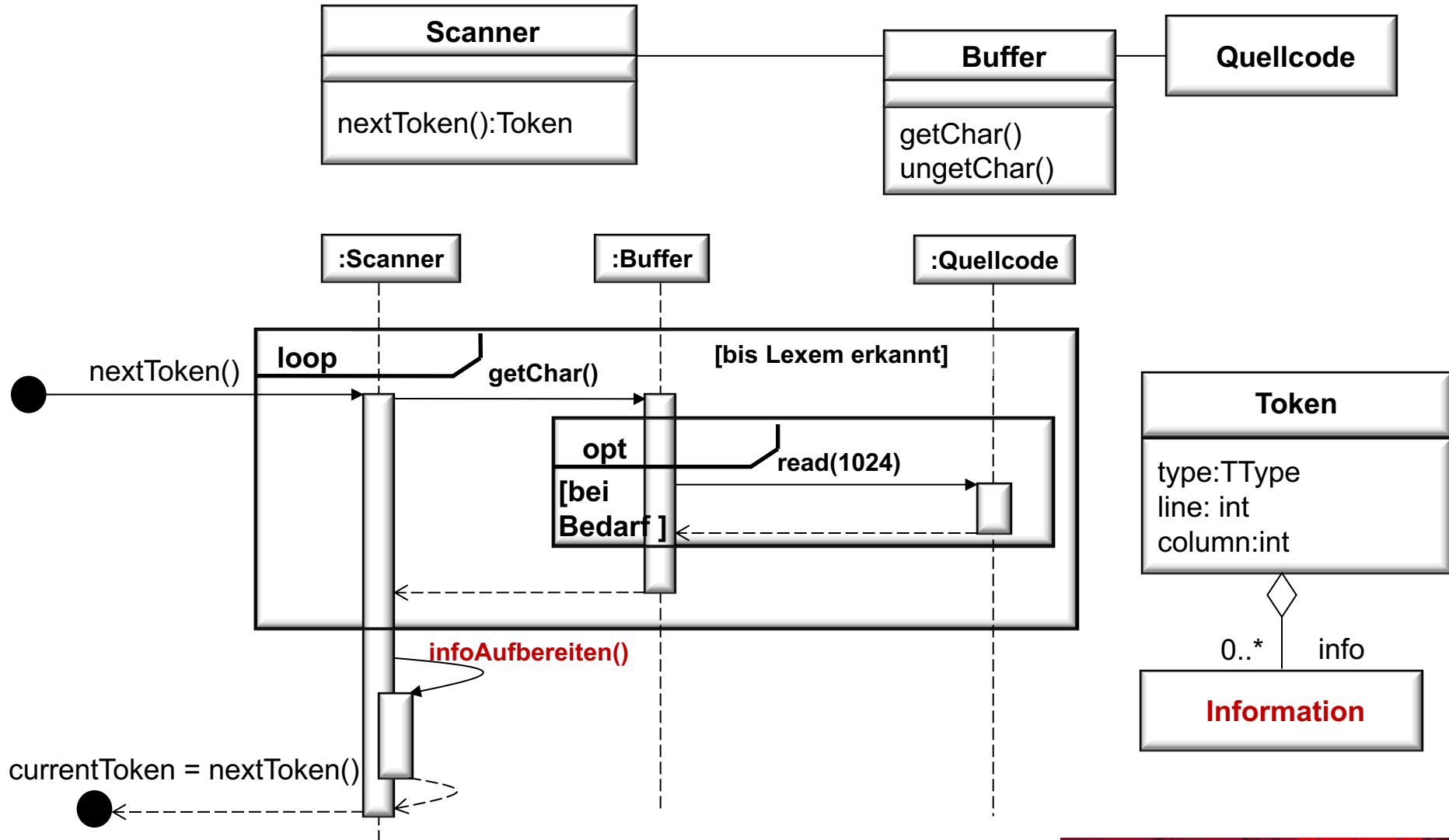
## Die Analyse besteht ihrerseits aus mehreren Teilaufgaben

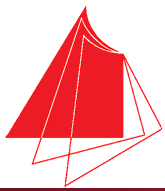
- **Lexikalische Analyse**
  - Zerlegung des Quellcodes in die Grundsymbole (Tokens)  
Bezeichner, Schlüsselworte, Sonderzeichen, Zahlen, ...
  - Speichern und Weiterleiten von Informationen (Namen, Values)
- **Syntaktische Analyse**
  - Überprüfung der syntaktischen Spracheigenschaften  
Sind die Ausdrücke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





# Arbeitsweise eines Scanners





# Lexikalische Analyse (Scanning)

Zerlegung des Quellcodes in seine Grundsymbole

**Programmiersprache**

`position := initial + rate * PI;`

**Textformatierer**

die lexikalische Analyse:

## Tokens

## Lexem

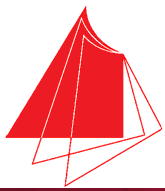
## Werte

1.	Bezeichner	position	—
2.	Zuweisungs-Zeichen	—	—
3.	Bezeichner	initial	—
4.	Plus-Zeichen	—	—
5.	Bezeichner	rate	—
6.	Multiplikations-Zeichen	—	—
7.	Real-Const	PI	3.14
8.	Semikolon	—	—

## Tokens

## Lexem

1.	Wort	die
2.	Leerraum	—
3.	Wort	lexikalische
4.	Leerraum	—
5.	Wort	Analyse
6.	Kolon	—



# Speichern von Werten

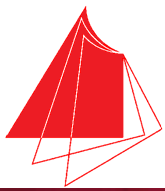
Eine der wesentlichen Aufgaben des Compilers ist es **Informationen** über die im Quellcode enthaltenen **Bezeichner (Identifier)** zu erfassen, zu verwalten und mit den entsprechenden Tokens zu verbinden.

Hierzu gehören: Typ, Name, Anzahl der Argumente usw.

Obwohl dies syntaktisch korrekt sein könnte, würde dies wohl kaum ein Compiler übersetzen.



```
int [3] x;  
int y;  
void p(int x, int y);  
...  
z := x[1] - 4;  
x[2] := 4;  
x := 3.5;  
Y:= p(x, y);
```

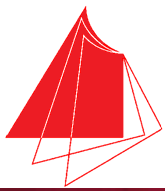


# Speichern von Werten

Diese Informationen zu verwalten ist Aufgabe der **Symboltabelle**.

Diese zu initialisieren und die Verbindung zum Token herzustellen ist Aufgabe des **Scanners**.

Dabei ist es entscheidend, dass vorhandene Informationen schnell extrahiert und neue Informationen schnell abgespeichert werden können.



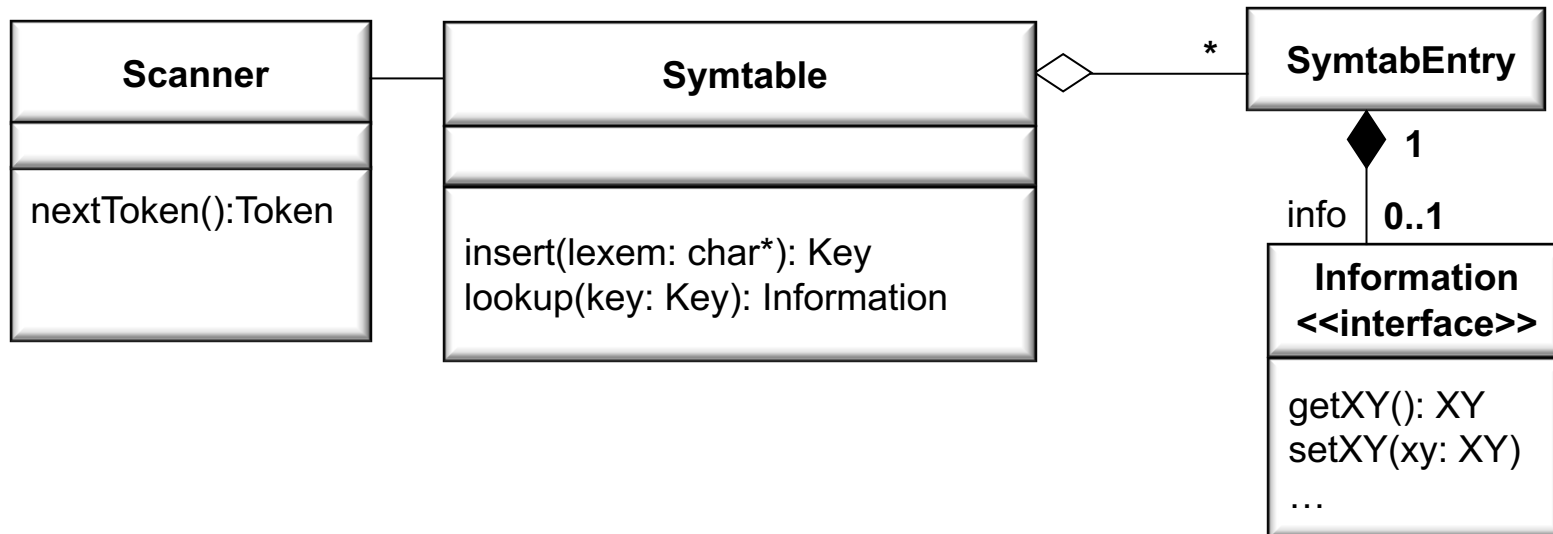
# Die Symboltabelle

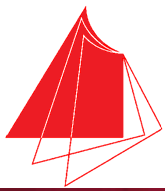
- In der Symboltabelle wird immer dann gesucht, wenn im Quelltext ein Lexem, das für einen Bezeichner steht, gefunden wurde.
- Änderungen treten in der Tabelle auf, wenn ein neuer Bezeichner oder neue Informationen über einen existierenden Bezeichner gefunden werden.

**Hierzu stehen zwei Operationen zur Verfügung:**

`insert(Lexem): Key`      */\* dieser **Key** wird im Token gespeichert (als Referenz auf die Informationsquelle)\*/*

`lookup(Key): Information`

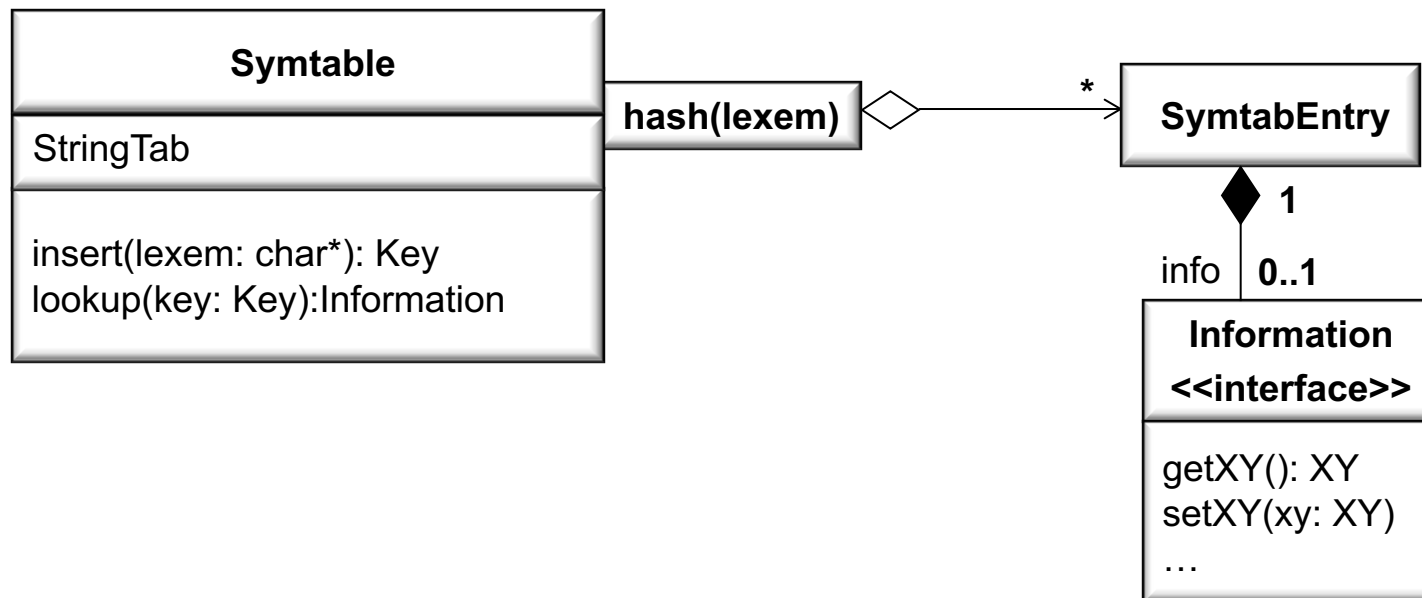


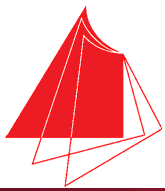


# Auffinden von Lexemen

Das **schnelle** Eintragen und Auffinden eines Lexems (Bezeichners), verbunden mit dem Extrahieren und Manipulieren der Information ist eine der zentralen Herausforderungen für jeden Compiler.

**!Aus diesem Grund wird die Symboltabelle meist als Hashtabelle realisiert!**

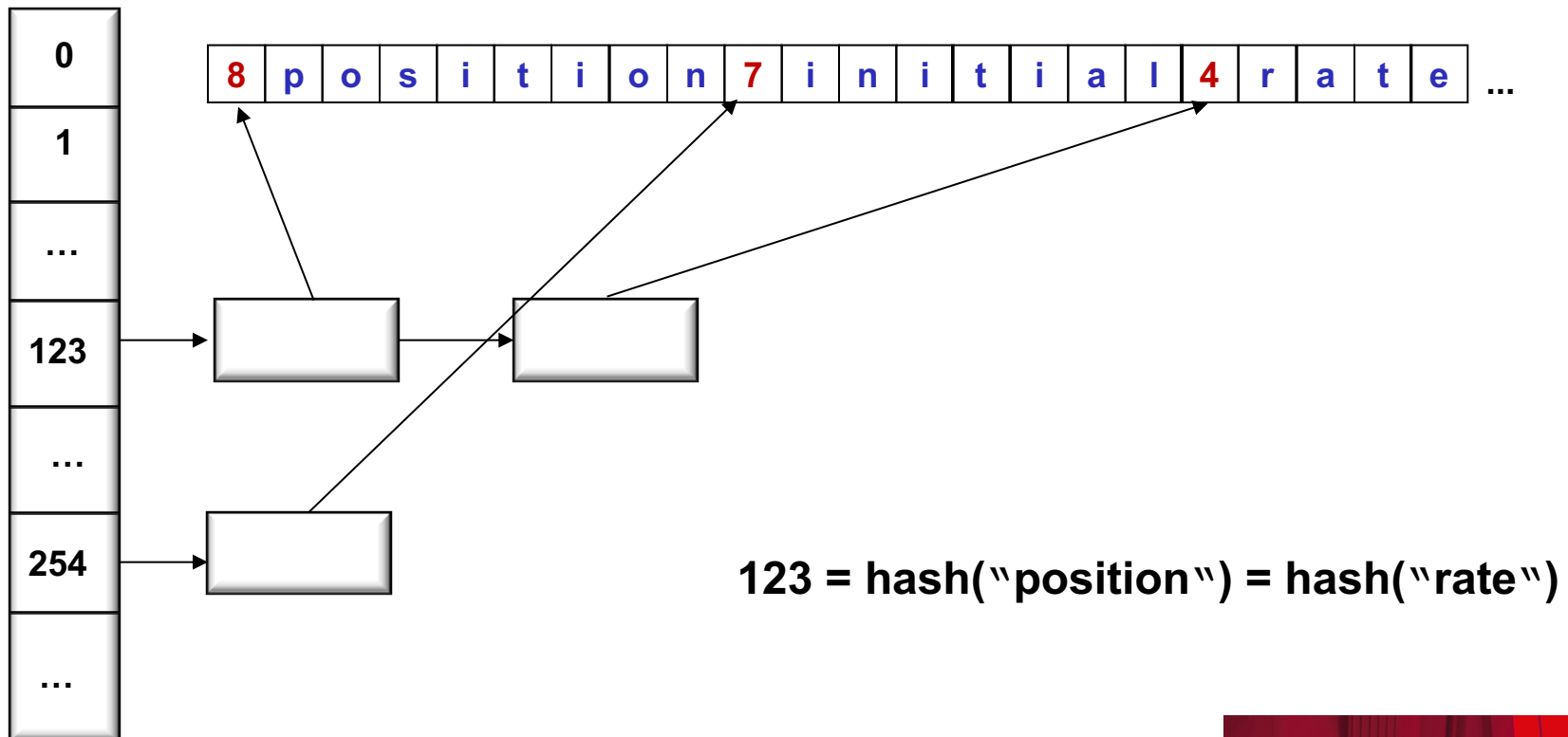




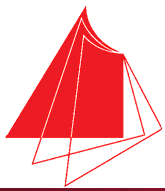
# Beispiel:

## Symboltabelle als Hashtabelle mit eigener Stringtabelle

Dies spart Zeit beim Anlegen von Objekten.





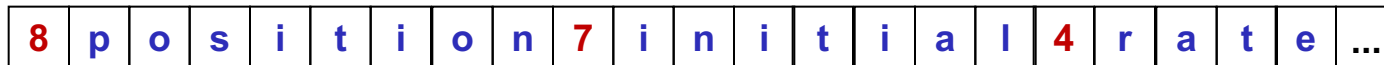


# Stringtabellen für Lexeme

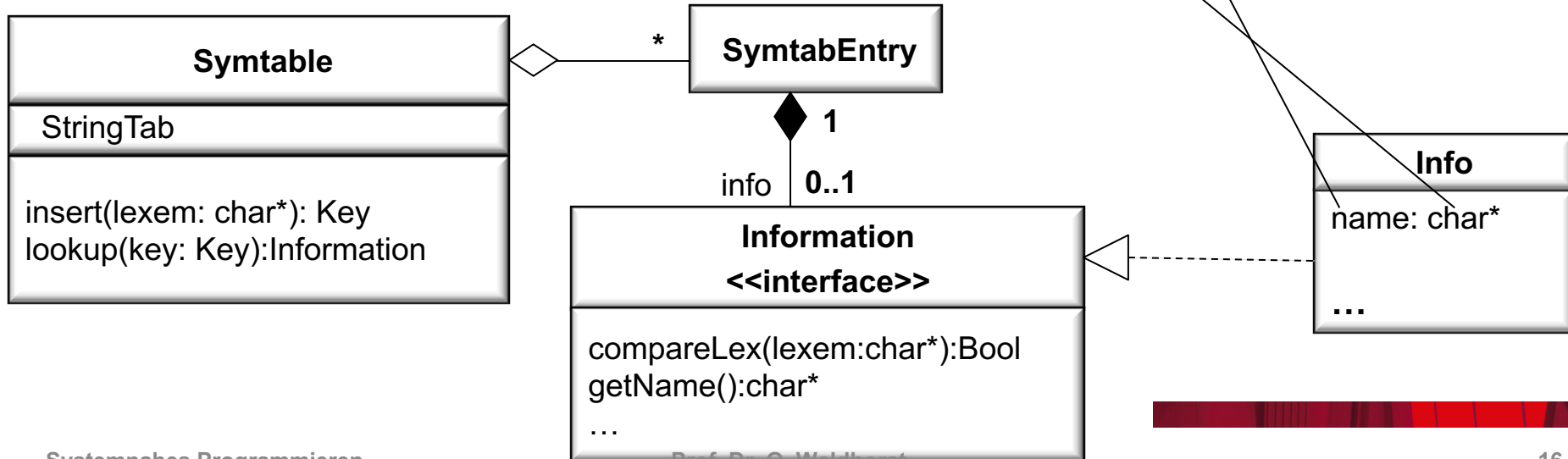
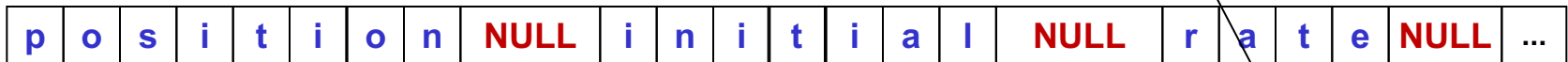
Alle Lexeme befinden sich in einem Char-Vektor (Stringtable)

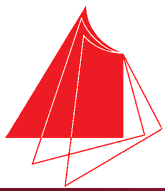
$$\text{position} = \text{initial} + \text{rate} * \text{PI}$$

mit Längenangabe



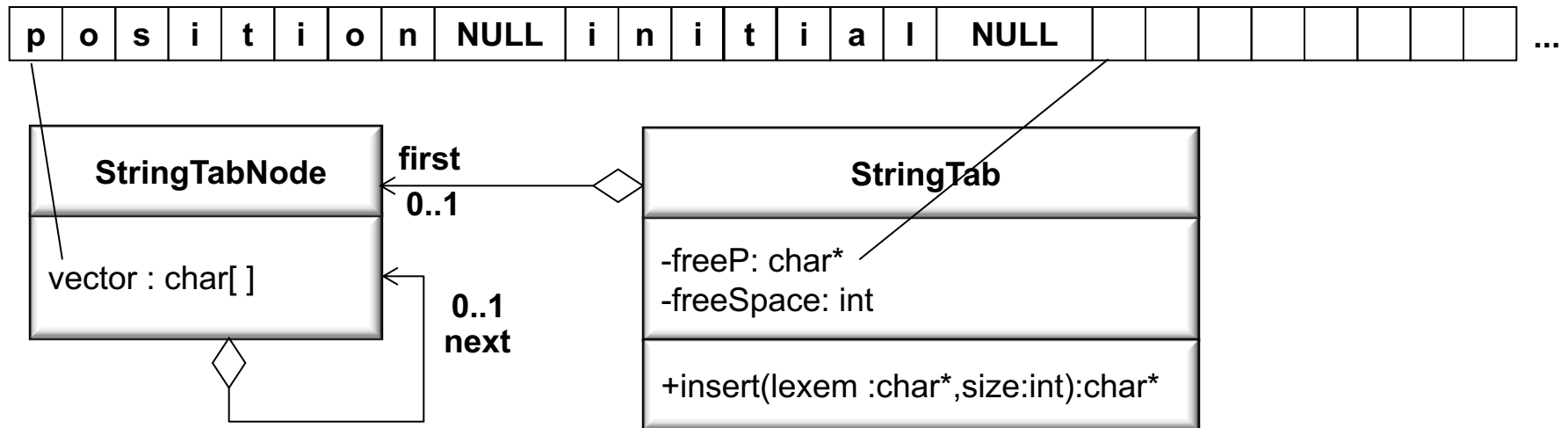
mit Endmarke



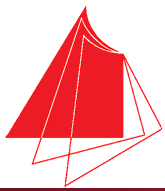


# Stringtabellen für Lexeme

**Auch diese Datenstrukturen müssen verwaltet werden:**

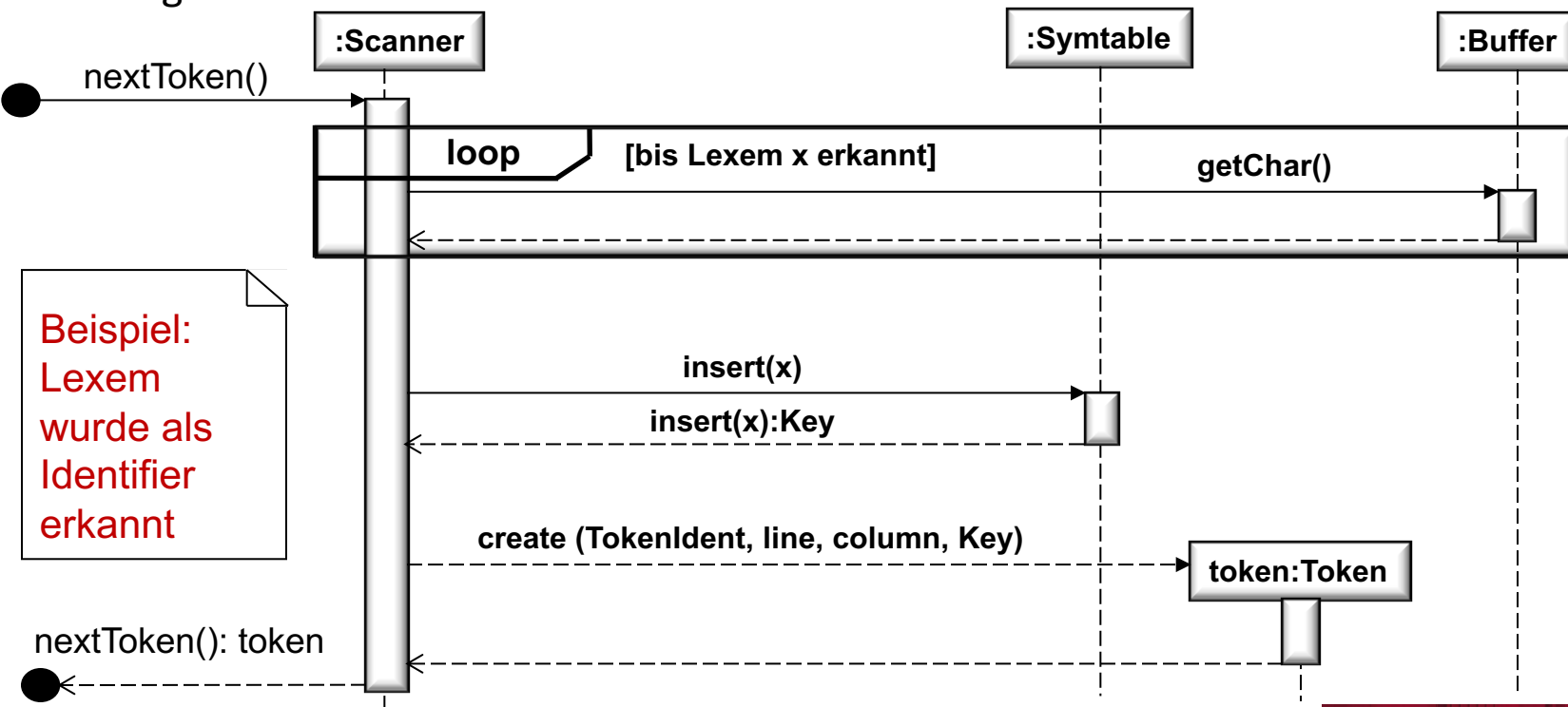


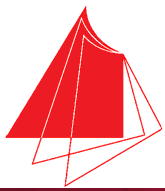
```
char* StringTab::insert(char* lexem, int size){
    char* tmp = this->freeP;
    if (size < this->freeSpace){
        memcpy(this->freeP,lexem,size+1);
        this->freeP[size] = '\0';
        this->freeP += size+1; this->freeSpace -= size+1;
    } else{ /* todo */ }
    return tmp;}
```



# Wie erkennt man ein Lexem?

**Nochmals zur Aufgabe:** Der Scanner liest nach der Aufforderung `nextToken()` solange mittels `getChar()` das nächste Zeichen, bis er ein Zeichen oder einen Bezeichner erkannt hat. Ist dieses ein Bezeichner, wird über die Symboltabelle mittels `insert()` eine eindeutige Referenz (`inofKey`) auf das eigentliche Informationsobjekt angefordert. Falls der Bezeichner neu war, wird dieses erzeugt und mit dem Bezeichner verknüpft; ansonsten lediglich dessen Referenz zurückgeliefert.



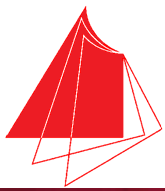


# Wie erkennt man ein Lexem?

Die Menge aller gültigen Zeichen und Bezeichner einer Programmiersprache ist üblicherweise eine reguläre Sprache.

**digit** ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
**letter** ::= A | B | C | ... | Z | a | b | ... | z  
**sign...** ::= + | - | : | \* | < | > | = | := | == | ! | && | ; | ( | ) | { | } | [ | ]  
**integer** ::= digit digit\*  
**identifizier** ::= letter (letter | digit)\*  
**if** ::= if | IF  
**while** ::= while | WHILE

Die gültigen Symbole bilden die reguläre Sprache  
 $L(\text{sign+} \mid \dots \mid \text{sign}] \mid \text{integer} \mid \text{identifizier} \mid \text{if} \mid \text{while})$



Zu jedem regulären Ausdruck  $s$  kann ein endlicher Automat  $A(s)$  konstruiert werden, der die Sprache  $L(s)$  akzeptiert.

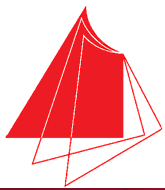
Das heißt:

- Nach Abarbeiten eines beliebigen Wortes aus  $L(s)$  befindet sich der Automat in einem Finalzustand.
- Nach Abarbeiten eines beliebigen Wortes, das nicht aus  $L(s)$  ist, befindet sich der Automat nicht in einem Finalzustand.

**Solche Automaten bezeichnet man auch als Akzeptoren.**

Ein Akzeptor  $A$  ist ein Fünftupel  $(Q, S, P, i, F)$

- $Q$  die Menge der Zustände (endlich)
- $S$  ist die Menge der Eingabesymbole (endlich)
- $P$  Menge von Zustandsübergängen  $qa ::= p$  bzw.  $q ::= p$   $q, p \in Q$  und  $a \in S$
- $i$  Startzustand ( $i \in Q$ )
- $F$  ist die Menge der Finalzustände ( $F \subseteq Q$ )



# Darstellung von Automaten

**A:**

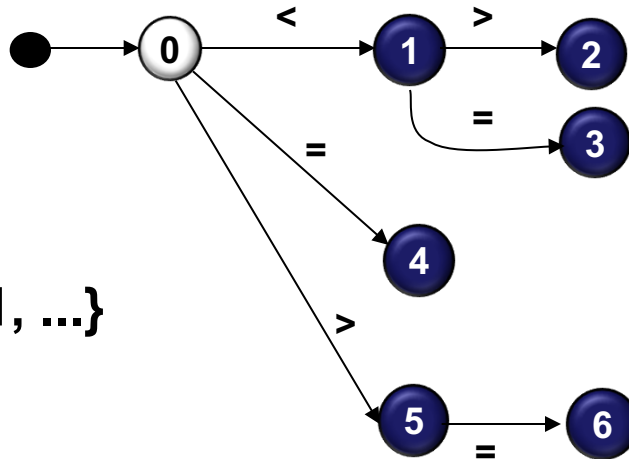
$Q = \{0, \dots, 6\}$

$S = \{<, =, >\}$

$P = \{0 < ::= 1, \dots\}$

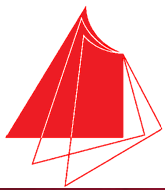
$i = 0$

$F = \{1, \dots, 6\}$



Der Automat **A** akzeptiert  $L(\mathbf{A}) = \{<, >, =, <=, >=, <>\}$ .  
Dies entspricht der durch den regulären Ausdruck **ra** definierten Sprache  $L(\mathbf{ra})$ .

$\mathbf{ra} ::= < \mid > \mid = \mid <= \mid >= \mid <>$

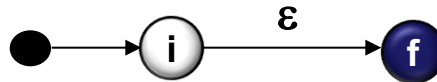


# Konstruktion von Automaten

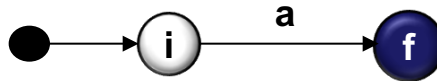
**Eingabe:** Ein regulärer Ausdruck  $r$  über dem Alphabet .

**Ausgabe:** Ein Automat  $A(r)$  mit  $L(A(r)) = L(r)$ .

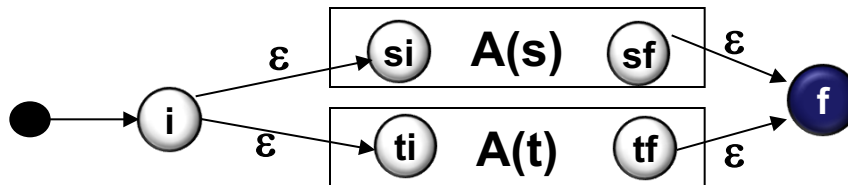
•  $r ::= \varepsilon$



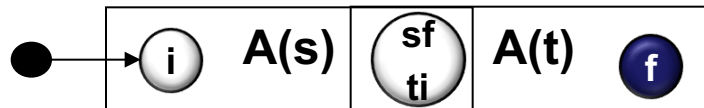
•  $r ::= a$  für  $a \in X$



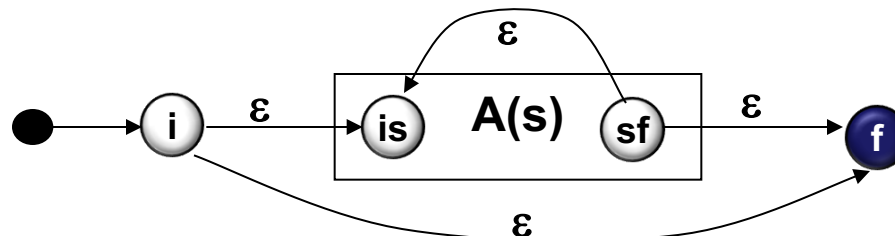
•  $r ::= s \mid t$

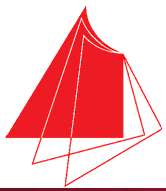


•  $r ::= st$



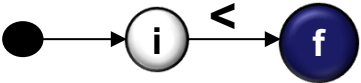
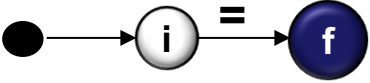
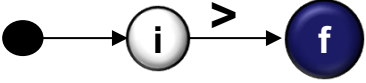
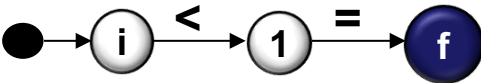
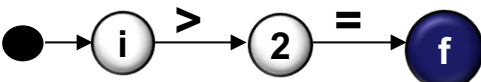

•  $r ::= s^*$



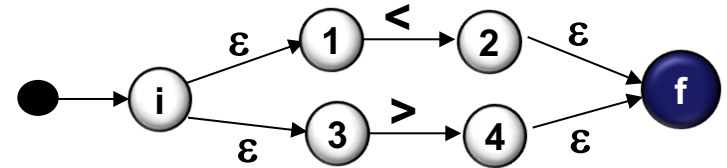


# Bsp.:

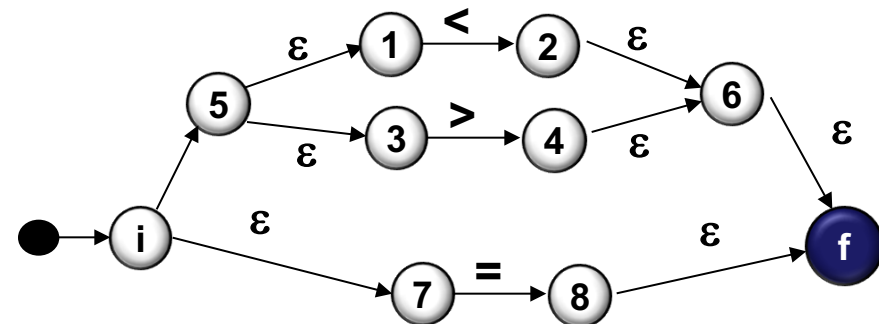
$ra ::= < \mid > \mid = \mid <= \mid >= \mid <>$

- $r ::= <$  
- $r ::= =$  
- $r ::= >$  
- $r ::= <=$  
- $r ::= >=$  
- $r ::= <>$  

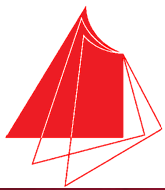
- $r ::= < \mid >$



- $r ::= (< \mid >) \mid =$







# Bsp.:

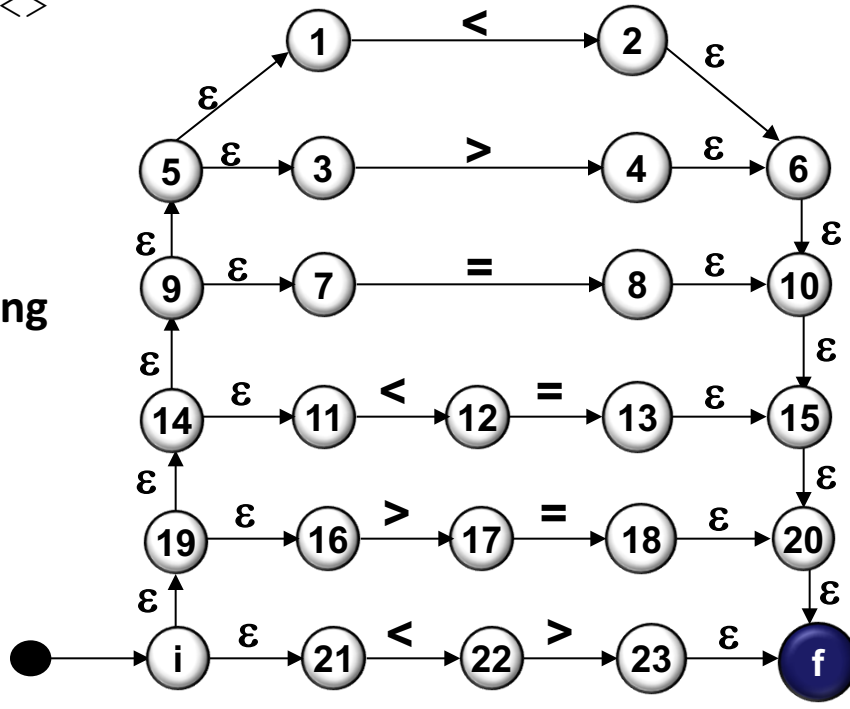
$ra ::= < \mid > \mid = \mid <= \mid >= \mid <>$

$ra ::= ((((<|>)|=)|<=)|>=)|<>$

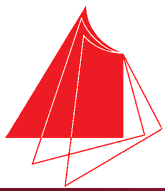
**Problem:** Welchen Übergang soll man wählen?

**Bsp.:**  $<=$

Übergang 1  $\Rightarrow$  2  
oder 11  $\Rightarrow$  12  
oder 21  $\Rightarrow$  22



Wie wird aus diesem Automat der vorherige Automat?



# Wiederholtes Zusammenfassen

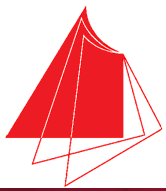
**Man konstruiert wiederholt verschiedene Mengen.**

$\text{Closure}(t) = \{s \mid s \text{ ist von } t \text{ nur über } \epsilon\text{-Übergänge erreichbar oder } s=t\}$

$\text{Closure}(T) = \bigcup_{t \in T} \text{Closure}(t)$

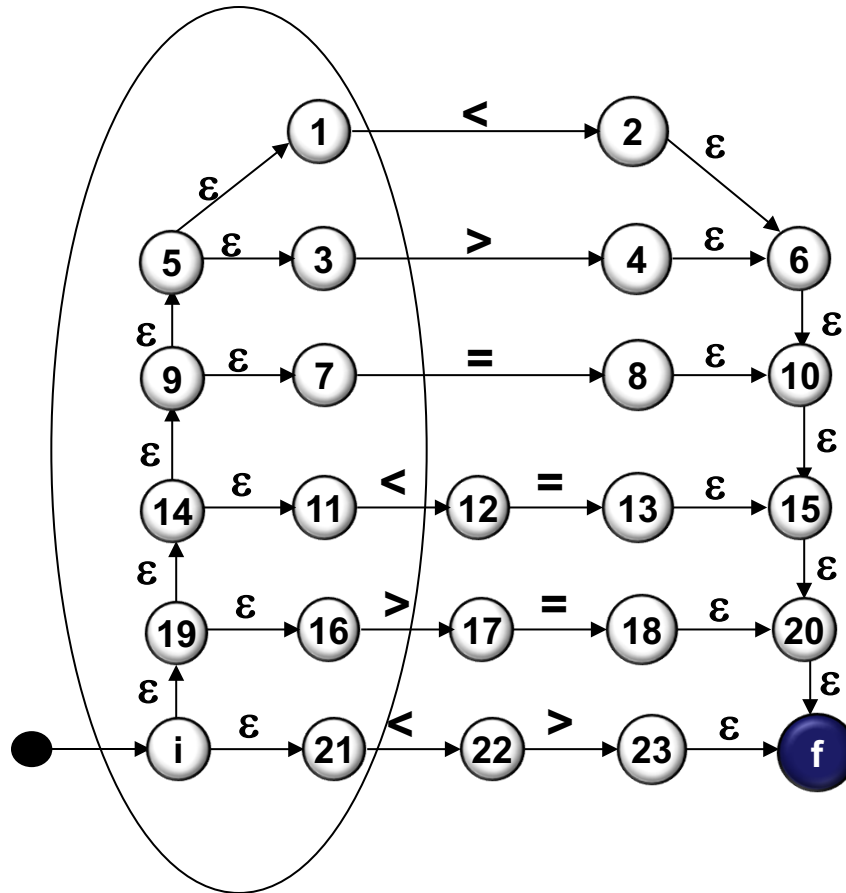
$\text{Move}(S,a) = \{t \mid \text{es gibt eine Regel } sa ::= t \in P \text{ und } s \in S\}$

1. Neuer Startzustand  $O = \text{Closure}(i)$ .
2. Bestimme für alle Symbole  $s$   $\text{Closure}(\text{Move}(O,s))$ .  
(Daraus ergeben sich maximal so viele neue Zustände wie vorhandene Symbole)
3. Wiederhole 2 für alle neuen Zustände (neuen Mengen) solange, bis keine neuen Zustände mehr hinzukommen.
4. Bestimme die neuen Finalzustände.  
(Alle Zustände (Mengen), die einen alten Finalzustand enthalten.)



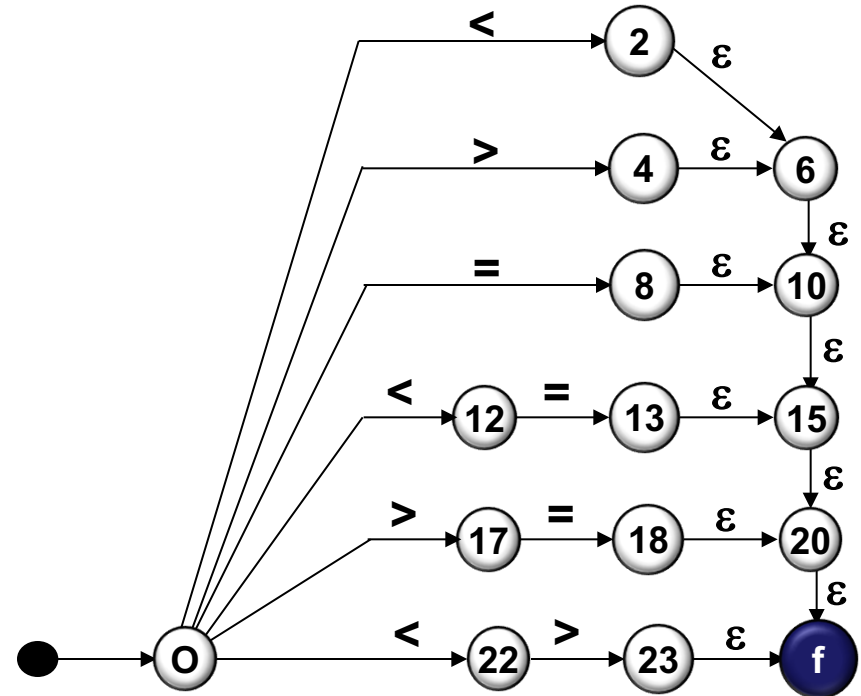
# Beispiel

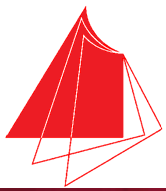
1.  $O = \{i, 1, 3, 5, 7, 9, 11, 14, 16, 19, 21\}$





$c = \text{Closure}(\text{Move}(O, =)) = \{8, 10, 15, 20, f\}$





# Beispiel

$O = \{i, 1, 3, 5, 7, 9, 11, 14, 16, 19, 21\}$

$a = \{2, 6, 10, 12, 15, 20, 22, f\}$

$\text{Closure}(\text{Move}(a, <)) = \{\}$

$d = \text{Closure}(\text{Move}(a, >)) = \{23, f\}$

$e = \text{Closure}(\text{Move}(a, =)) = \{13, 15, 20, f\}$

$b = \{4, 6, 10, 15, 17, 20, f\}$

$\text{Closure}(\text{Move}(b, <)) = \{\}$

$\text{Closure}(\text{Move}(b, >)) = \{\}$

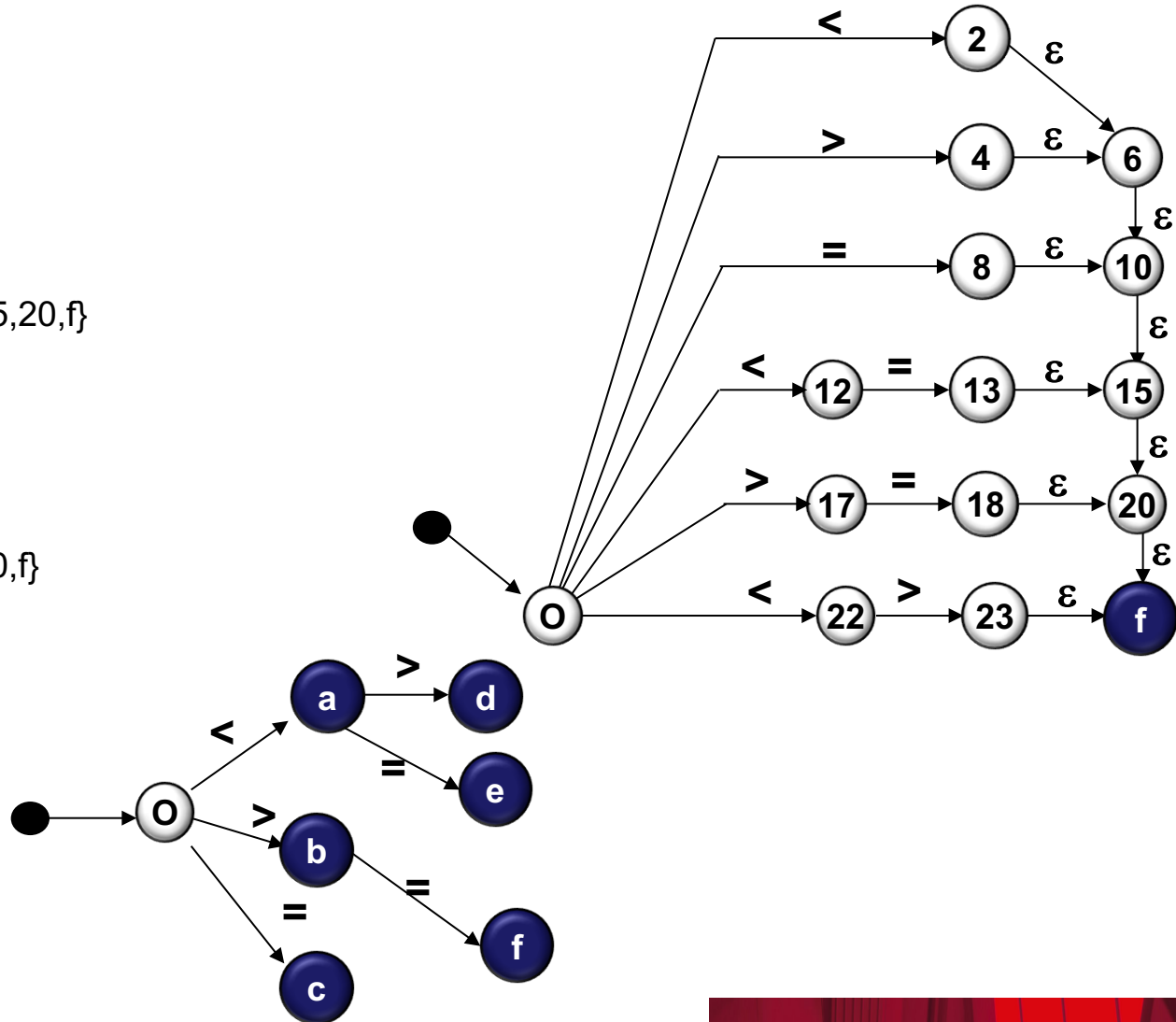
$f = \text{Closure}(\text{Move}(b, =)) = \{18, 20, f\}$

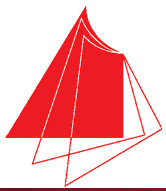
$c = \{8, 10, 15, 20, f\}$

$\text{Closure}(\text{Move}(b, <)) = \{\}$

$\text{Closure}(\text{Move}(b, >)) = \{\}$

$\text{Closure}(\text{Move}(b, =)) = \{\}$





# Beispiel

$O = \{i, 1, 3, 5, 7, 9, 11, 14, 16, 19, 21\}$

$a = \{2, 6, 10, 12, 15, 20, 22, f\}$

$b = \{4, 6, 10, 15, 17, 20, f\}$

$c = \{8, 10, 15, 20, f\}$

$d = \{23, f\}$

$\text{Closure}(\text{Move}(d, <)) = \{\}$

$\text{Closure}(\text{Move}(d, >)) = \{\}$

$\text{Closure}(\text{Move}(d, =)) = \{\}$

$e = \{13, 15, 20, f\}$

$\text{Closure}(\text{Move}(e, <)) = \{\}$

$\text{Closure}(\text{Move}(e, >)) = \{\}$

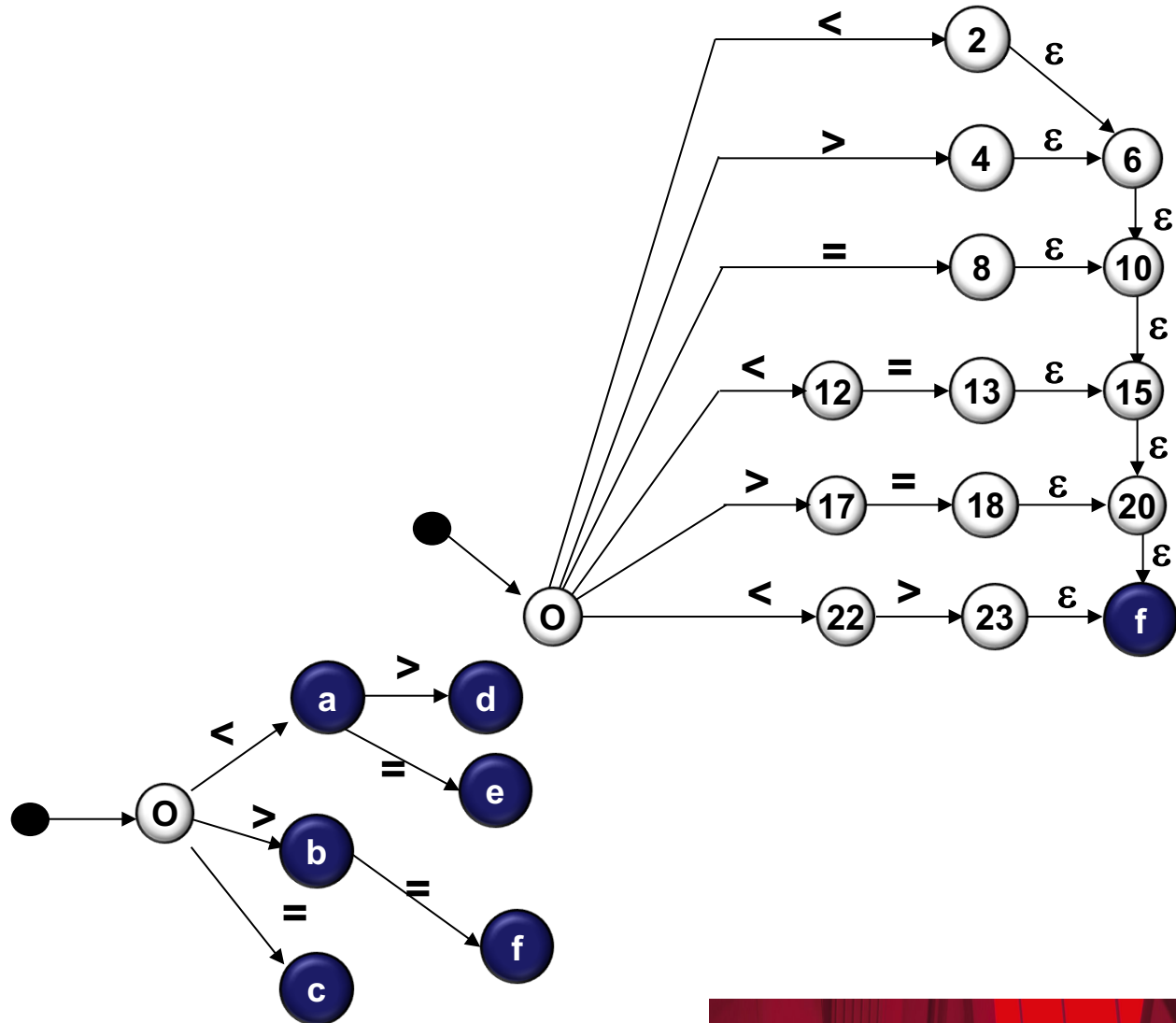
$\text{Closure}(\text{Move}(e, =)) = \{\}$

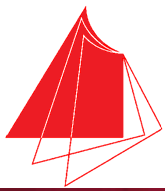
$f = \{18, 20, f\}$

$\text{Closure}(\text{Move}(f, <)) = \{\}$

$\text{Closure}(\text{Move}(f, >)) = \{\}$

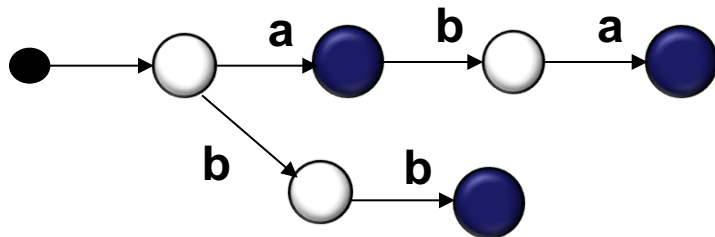
$\text{Closure}(\text{Move}(f, =)) = \{\}$





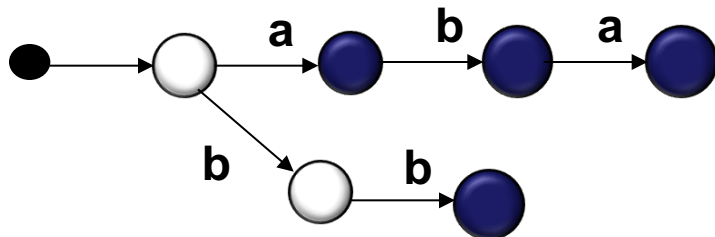
# Aufbereiten der Eingabe

Falls ein Text aus mehreren Worten besteht, ist es in der Regel notwendig, dass der Scanner einige Zeichen vorausschauen muss, bevor ein Wort erkannt werden kann.



Welche Lexeme verbergen sich hinter der Folge abb?

Die Lexeme a und bb.

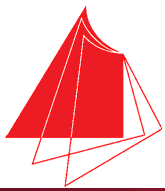


Welche Lexeme verbergen sich hinter der Folge abb?

Das Lexem ab und ein Fehler.

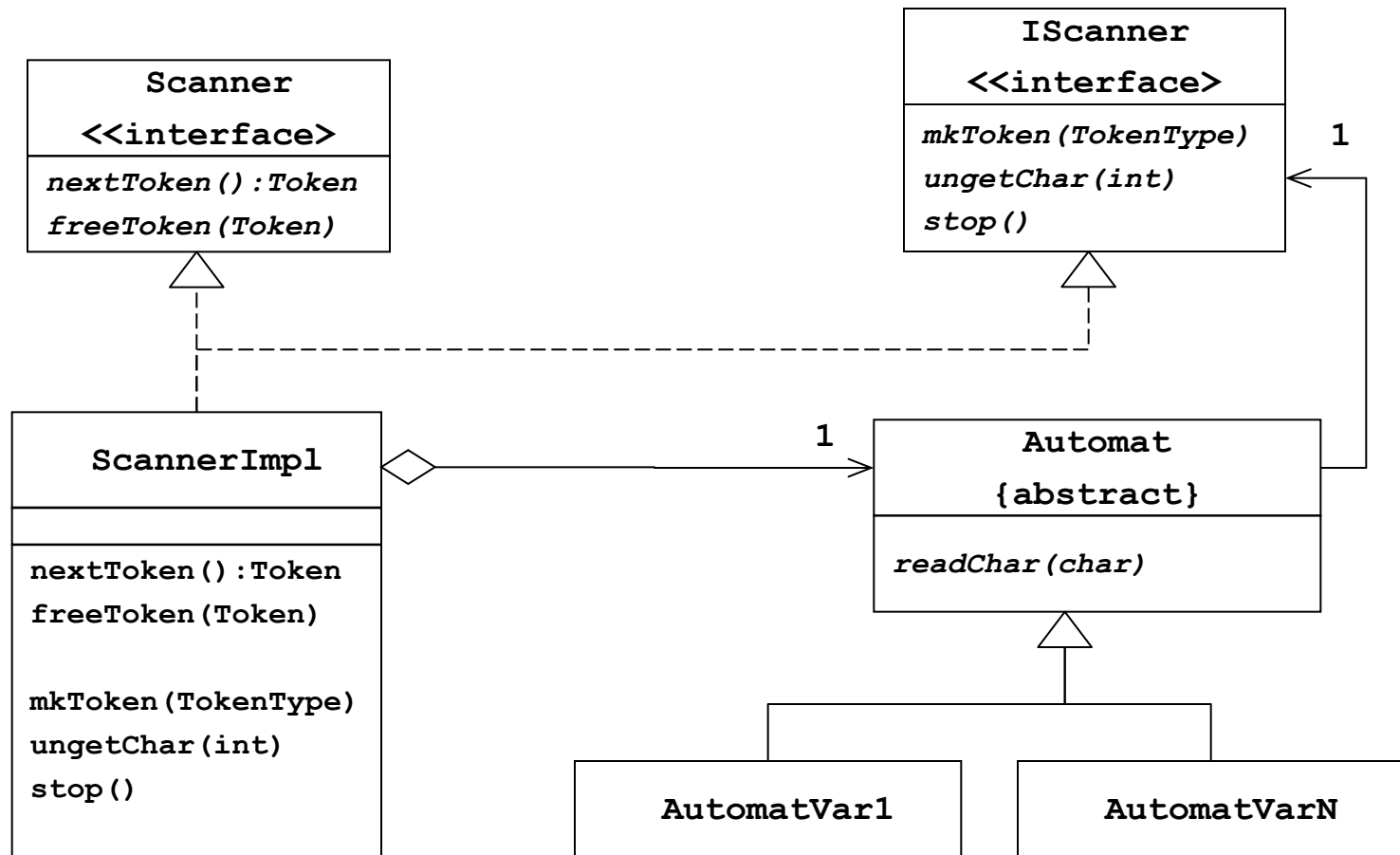


**Die Eingabe muss gepuffert werden.**

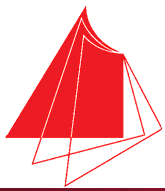


# Realisierung von Automaten

## Eine allgemeine und flexible Struktur



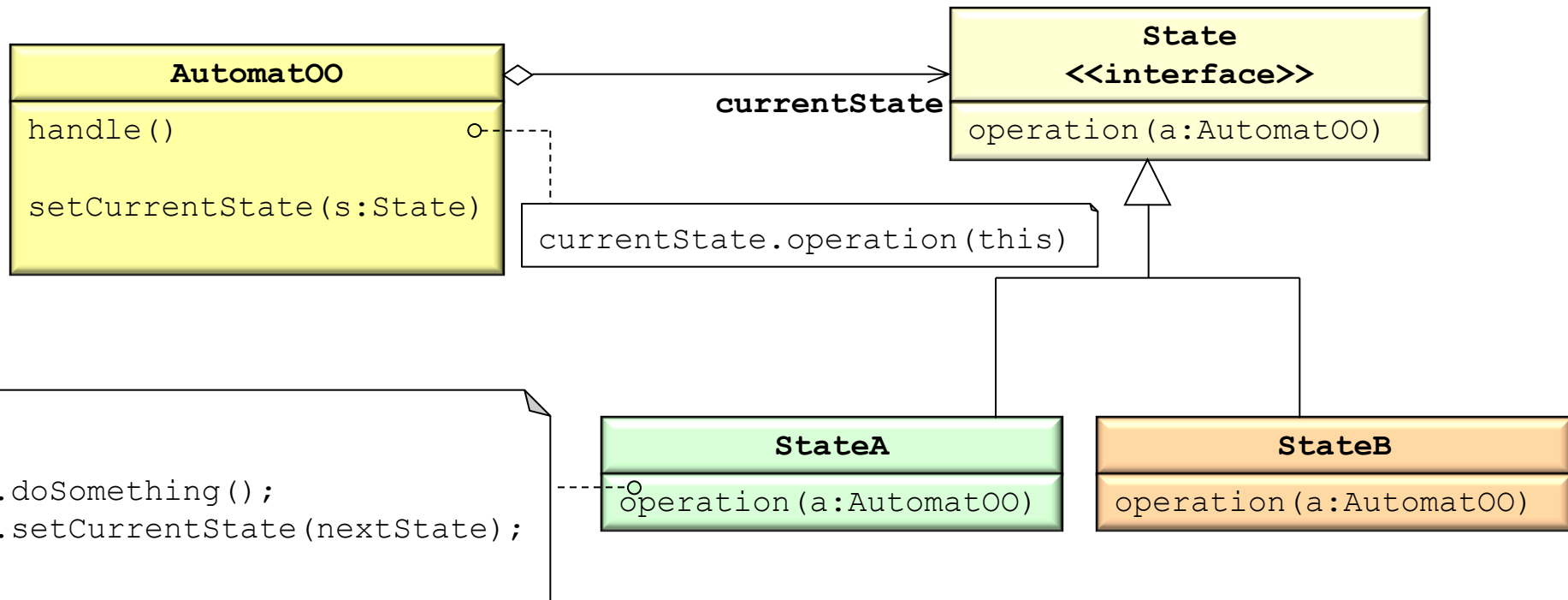


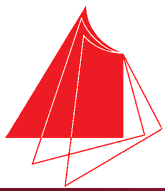


# Der objektorientierte Automat

## State-Pattern

Es ermöglicht einem Objekt sein Verhalten zu ändern,  
wenn der interne Zustand sich ändert.

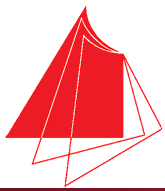




# Implementieren

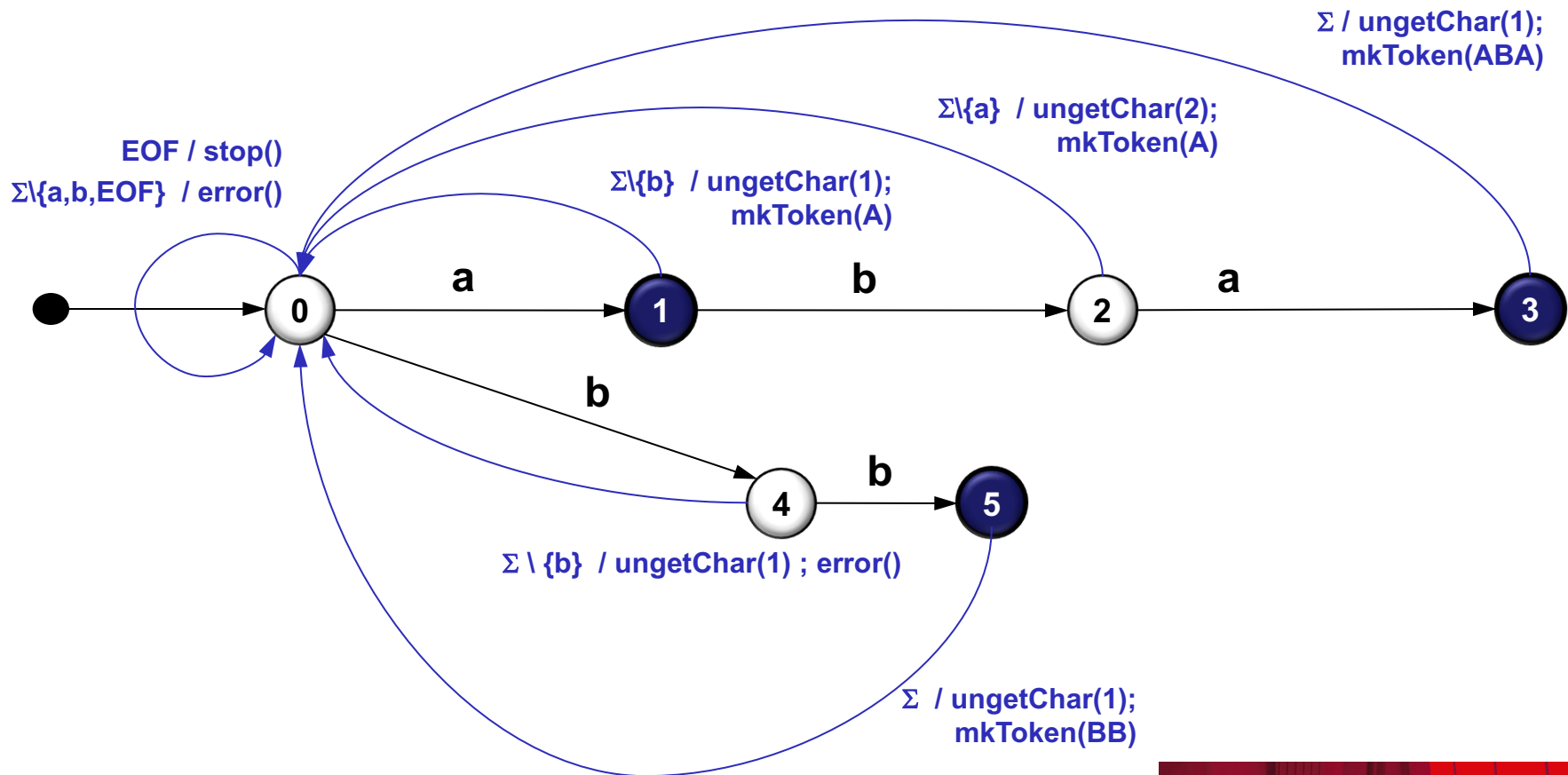
```
class State{
public:
    virtual void read(char c, AutomatOO* m) =0;
};
class State0: public State{ /*...*/}
...
class State5: public State{ /*...*/}

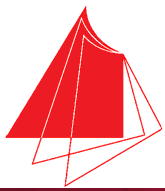
void State4::read(char c, AutomatOO* automat){
    switch (c) {
        case 'b' :    automat->setState(State5::makeState()); break;
        case '\0' :    automat->setState(InitialState::makeState());
                        automat->ungetChar(1);
                        automat->error(); break;
        default:
            automat->setState(InitialState::makeState());
            automat->error();
    }
}
```



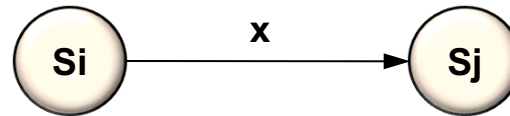
# Operationen bestimmen

**Der Automat muss vervollständigt werden und die auszuführenden Aktionen sind zu ergänzen.**





# Zustandsübergangstabelle



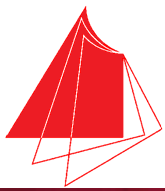
Zustand

Eingabe

Zustand				
Eingabe			Si	
	x		Sj	

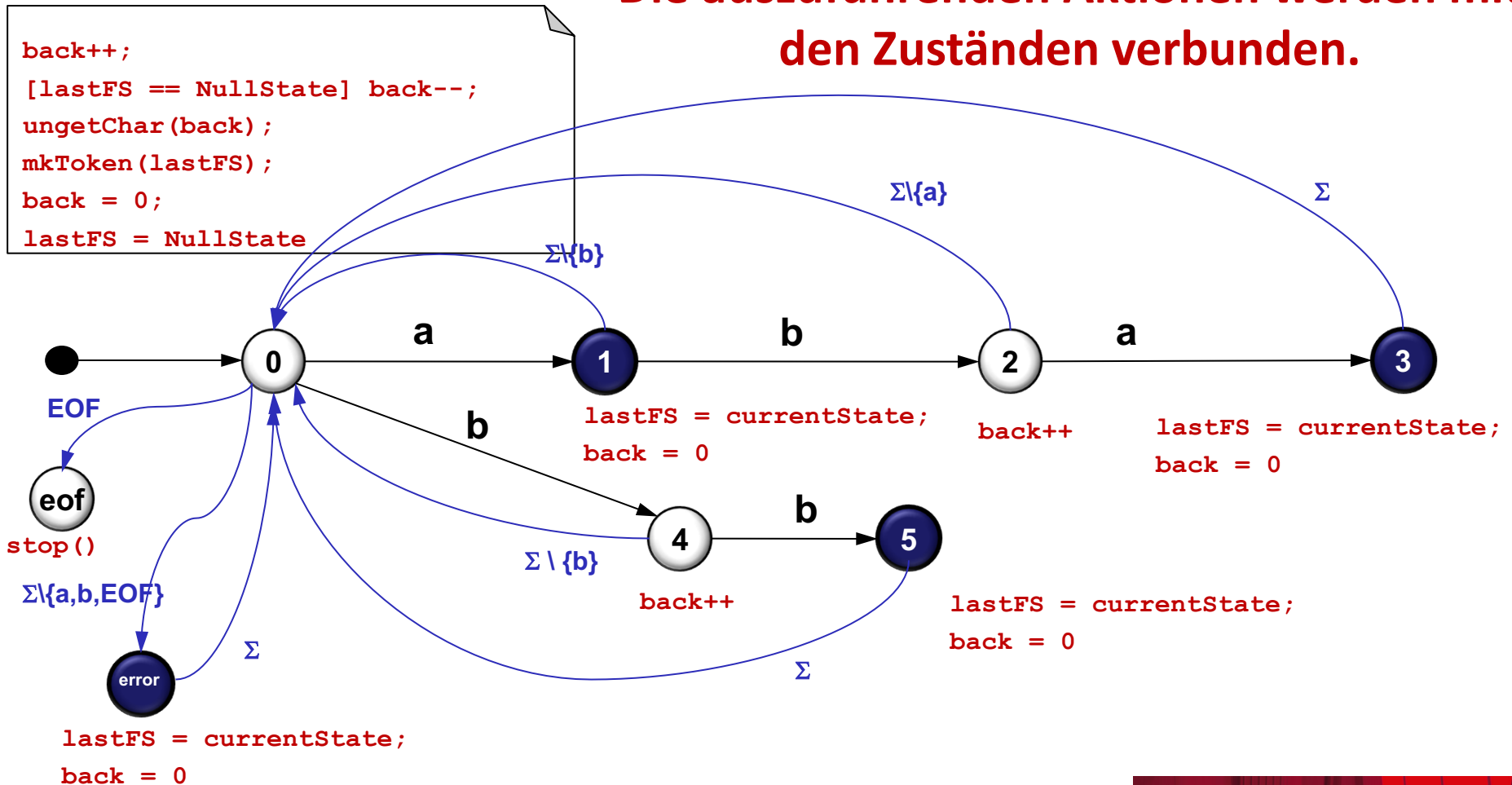
Folgezustand

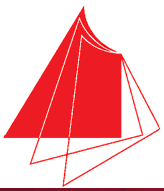
`nextState = stateMatrix[currentState][currentChar]`



# Zustände ergänzen

Der Automat muss vervollständigt werden.  
Die auszuführenden Aktionen werden mit  
den Zuständen verbunden.

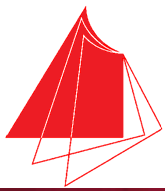




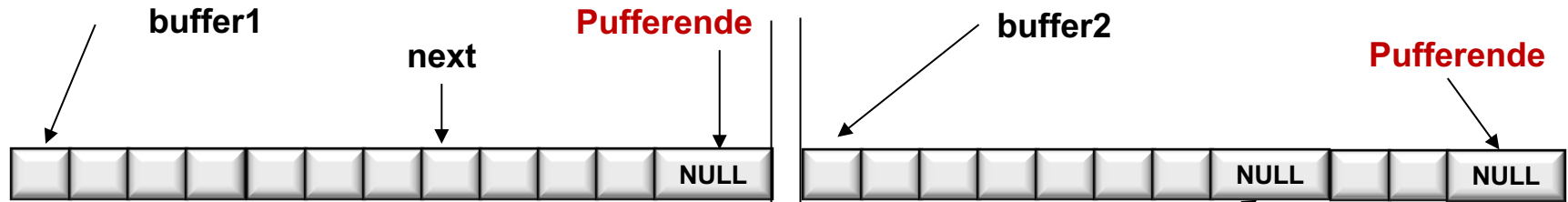
# Implementieren

```
AutomatMatrix::AutomatMatrix() { /* Matrix erstellen */  
    this->currentState      = STATE_START;  
    this->lastFinalState    = NULL_STATE;  
    this->back              = 0; }
```

```
void AutomatMatrix::read(char currentChar) {  
    this->currentState =  
        this->stateMatrix[this->currentState][ (int) (unsigned char)currentChar];  
  
    if (this->currentState & IS_FINAL) { // final State  
        this->lastFinalState = this->currentState;  
        this->back = 0;  
    } else {  
        this->back++;  
        if (this->currentState == STATE_START) {  
            if (this->lastFinalState == NULL_STATE) this->back--;  
            ungetChar(this->back);  
            mkToken(this->lastFinalState);  
            this->back = 0;  
            this->lastFinalState = NULL_STATE;  
        } else if (this->currentState == STATE_EOF) stop();  
    }  
}
```



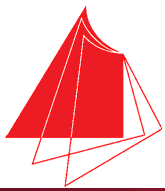
# Ein einfaches Pufferschema



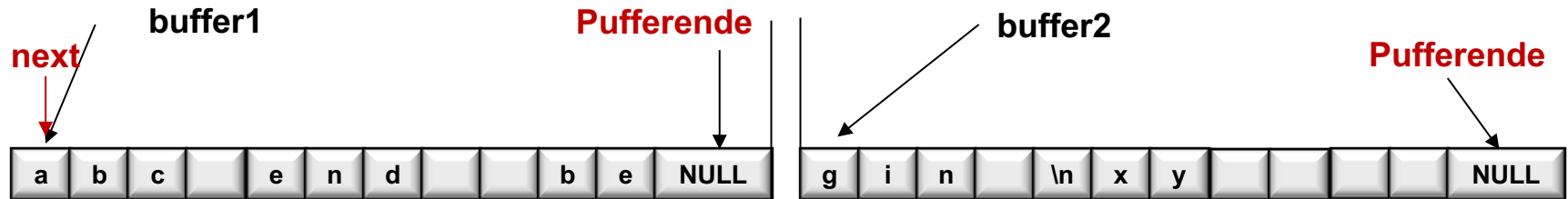
```
if (*next == NULL) {
    if (next == &buffer1[size]) /* am Ende der ersten Hälfte*/
    {
        // lade die zweite Hälfte neu;
        next = buffer2;
    }
    else
        if (next == &buffer2[size]) /* am Ende der ersten Hälfte*/
        {
            // lade die erste Hälfte neu;
            next = buffer1;
        }
    else // lexikalische Analyse beenden
}
else next++;
```

**Datenende**

**Pufferende und  
Datenende  
werden durch  
das gleiche  
Zeichen  
dargestellt  
(z.B.: NULL)**



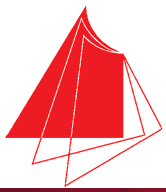
# Ein einfaches Pufferschema



```
if (*next == NULL) {
    if (next == &buffer1[size]) /* am Ende der ersten Hälfte*/
    {
        // lade die zweite Hälfte neu;
        next = buffer2;
    }
    else
        if (next == &buffer2[size]) /* am Ende der ersten Hälfte*/
        {
            // lade die erste Hälfte neu;
            next = buffer1;
        }
        else // lexikalische Analyse beenden
        }
    else next++;
```

**Pufferende und  
Datenende  
werden durch  
das gleiche  
Zeichen  
dargestellt  
(z.B.: NULL)**

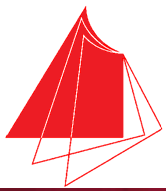




# Aufgabe:

Schreiben Sie einen Scanner, der in einer gegebenen Datei alle Worte der regulären Sprache **L(sign+ | ... | sign) | integer | identifier | if | while** findet und Worte, die nicht zu dieser Sprache gehören, als fehlerhaft markiert.

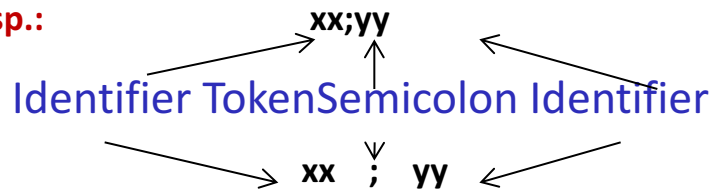
<b>digit</b>	::= 0   1   2   3   4   5   6   7   8   9
<b>letter</b>	::= A   B   C   ...   Z   a   b   ...   z
<b>sign...</b>	::= +   -   :   *   <   >   =   :=   ==   !   &&   ;   (   )   {   }   [   ]
<b>integer</b>	::= digit digit*
<b>identifier</b>	::= letter (letter   digit)*
<b>if</b>	::= if  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



# Aufgabe: Hinweise

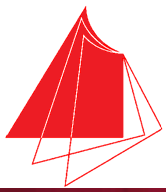
Eine Trennung der einzelnen Worte mittels Whitespace (Leerzeichen, Tabs oder Zeilenumbrüche) ist nicht notwendig aber erlaubt.

Bsp.:



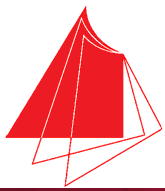
Kommentare sind zugelassen und sollen wie ein Whitespace behandelt werden.  
Sie trennen Worte und werden nicht als Token erfasst.

- Kommentare beginnen mit „:“ und enden auf „:“ oder erstrecken sich bis zum Dateiende.
- Innerhalb eines Kommentars dürfen beliebige ASCII-Zeichen stehen (natürlich nicht „:“).
- Kommentare kann man nicht schachteln.



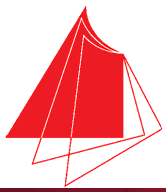
# Aufgabe: Details

- **Machen Sie sich über die Lösung der Aufgabe Gedanken und zerlegen Sie die Aufgabe in mindestens zwei Teile:**
  - **Puffer (I/O)**
  - **Automat , Scanner und Tokens**
  - **Symboltabelle (inkl. Hashtabelle und String-Tabelle)**
- **Entwickeln Sie die Software unter Eclipse.**
  - **Entwerfen Sie geeignete Schnittstellen in Form Header-Dateien, so dass der Parser diese später einbinden kann.**



# Automat und Scanner

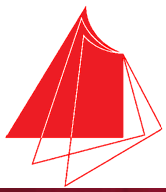
- Erstellen und implementieren Sie einen Automaten.  
(**behandeln Sie Schlüsselworte und Identifizier gleich**)
- Für jedes erkannte Lexem ist ein Token zu erstellen, das **Zeile**, **Spalte** und **Typ** enthält.
  - Erkannte **Identifizier** sind in die Symboltabelle einzutragen und ein Verweis auf deren **Informationskomponente** (**vorerst: Lexem und Typ**) ist im Token zu speichern.
  - Für erkannte **Integer** ist deren **Wert** zu bestimmen und im Token zu speichern.  
**Nutzen Sie hierfür die Funktion strtol.**  
Zur Erkennung von Bereichsüberschreitungen nutzen Sie **error.h** und **errno.h**.
- Bei Symbolen, die nicht zur Sprache gehören, ist ein **Fehlertoken** zu erzeugen, das neben Zeile und Spalte auch das fehlerhafte Zeichen enthält.



# Symoltabelle

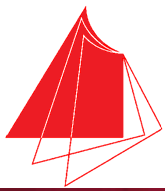
- Realisieren Sie die **Kollisionsauflösung** durch **Verkettung**.
- Zur **Vorbelegung** der Symoltabelle implementieren Sie die Operation **initSymbols()**, die die Symoltabelle mit Schlüsselworten füllt.

```
void Symtable::initSymbols() { insert("while",TokenWhile); ... }
```



# Puffer

- Zur besseren Verarbeitung puffern Sie die Eingabe.
- Verwenden Sie zur Eingabe **ausschließlich** die Funktionen Ihres Puffers.
- Realisieren Sie den Dateizugriff mittels der C/C++ Bibliothek **fstream**.



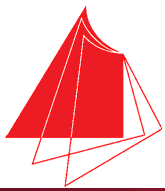
# main

Zum späteren Testen schreiben Sie ein Programm, das zwei Dateien entgegennimmt, einen Scanner erzeugt, Tokens anfordert und diese in eine Datei ausgibt, bis die Eingabe vollständig abgearbeitet wurde.

**Dieses Programm wird dann von der Console / Shell aus aufgerufen!**

```
int main(int argc, char* argv[]){
    . . .
    if (argc < 1) return EXIT_FAILURE;
    Symtab* stab = new Symtab();
    Scanner* s    = new Scanner(argv[1], stab);
    Token* t;
    while (t = s->nextToken()){
        //Token ausgeben
    }
    return EXIT_SUCCESS
}
```

„Test.cpp“



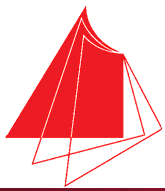
# Aufgabe: Beispiel

```
user> ./scanner scanner-test.txt out.txt
processing ...
unknown Token Line: 3 Column: 1 Symbol: ?
unknown Token Line: 5 Column: 19 Symbol: %
stop
user>
```

```
X:=3+4;
:* eine einfache Aufgabe !! *:
?y := 7 = := X : (X - 4);
Z := ((3 + 4 - 6);
Resultat := : X y %
„scanner-test.txt“
```

- „**scanner-test.txt**“ ist eine Eingabedatei.
- „**out.txt**“ ist eine Ausgabedatei, in die die gefundenen Tokens unter Angabe von Zeile, Spalte und ggf. Lexem bzw. Value geschrieben werden.
- Gefundene Fehler werden mit Angabe von **Zeile**, **Spalte** und **Symbol** auf „**stderr**“ ausgegeben.





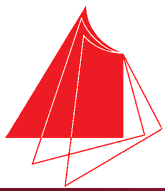
# Aufgabe: Beispiel

scanner-test.txt

```
X:=3+4;  
:* eine einfache Aufgabe !! *:  
?y := 7 = := X : (X - 4);  
Z := ((3 + 4 - 6);  
Resultat := : X y %
```

out.txt

Token Identifier	Line: 1 Column: 1	Lexem: X
Token Assign	Line: 1 Column: 2	
Token Integer	Line: 1 Column: 4	Value: 3
Token Plus	Line: 1 Column: 4	
Token Integer	Line: 1 Column: 5	Value: 4
Token Semicolon	Line: 1 Column: 6	
Token Identifier	Line: 3 Column: 2	Lexem: y
...		
Token Identifier	Line: 5 Column: 17	Lexem: y



# Dokumentation & Bewertung

## Scanner und Parser sind jeweils zu dokumentieren

- Aufgabenstellung (in eigenen Worten)
- Lösungsansatz und Umsetzung
- Programmausführung und Testfälle
- ...

**Jeweils ca. 10 Seiten pro Aufgabenteil**

## Bewertung

- |                         |                       |
|-------------------------|-----------------------|
| • Scanner               | max. 10 Punkte        |
| • Scanner Dokumentation | max. 10 Punkte        |
| • Parser                | max. 10 Punkte        |
| • Parser Dokumentation  | max. 10 Punkte        |
| • <b>Gesamt</b>         | <b>max. 40 Punkte</b> |
- **Zum Bestehen müssen 2/3 der Punkte ( $\approx 27$  Punkte) erreicht werden**