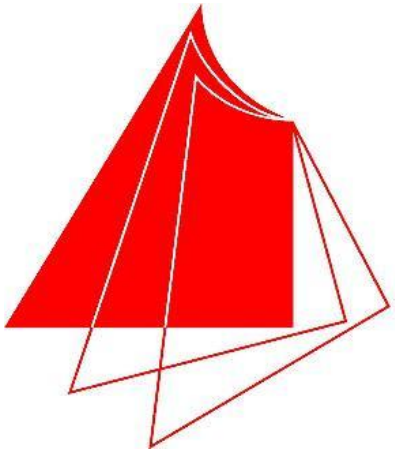
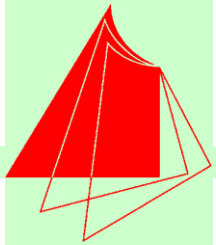


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

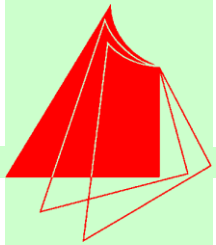
Labor
Systemnahes Programmieren

Prof. Dr. Thomas Fuchß
Hochschule Karlsruhe – Technik und Wirtschaft
Fakultät für Informatik und Wirtschaftsinformatik
Fachgebiet Informatik



Übersicht

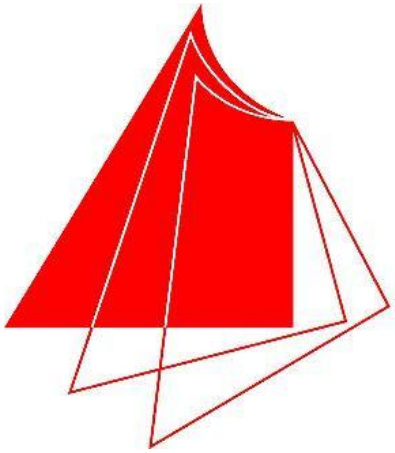
- **Allgemeines**
- **Teil I Lexikalische Analyse**
- **Teil II Syntaktische Analyse**
- **Teil III Prozesssynchronisation und Prozesskommunikation**



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++



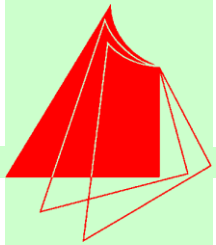
Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Systemnahes Programmieren
Teil I
Lexikalische Analyse

Prof. Dr. Thomas Fuchß

Hochschule Karlsruhe – Technik und Wirtschaft
Fakultät für Informatik und Wirtschaftsinformatik

Fachgebiet Informatik



Literatur

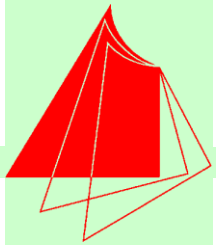
- A.V. Aho, R. Sethi und J.D. Ullmann.
  Compilerbau – 2nd Edition –Oldenburg, 1999

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

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

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

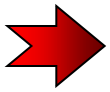
- D. Grune et. al.
  Modern compiler design - Wiley, 2000



Ziel:

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

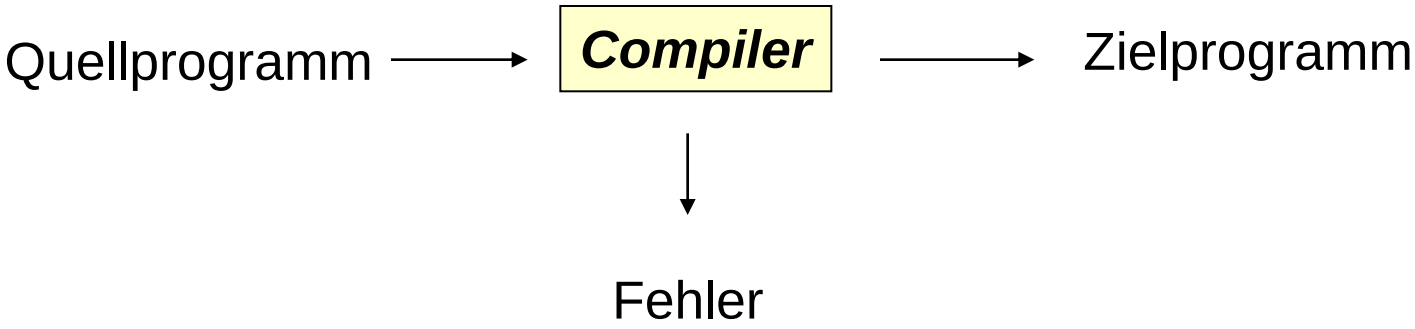
Des Weiteren soll die Implementierung eines Scanners das Verständnis für dynamische Datenstrukturen und Zeiger (in C/C++) vertiefen.



kein Java !

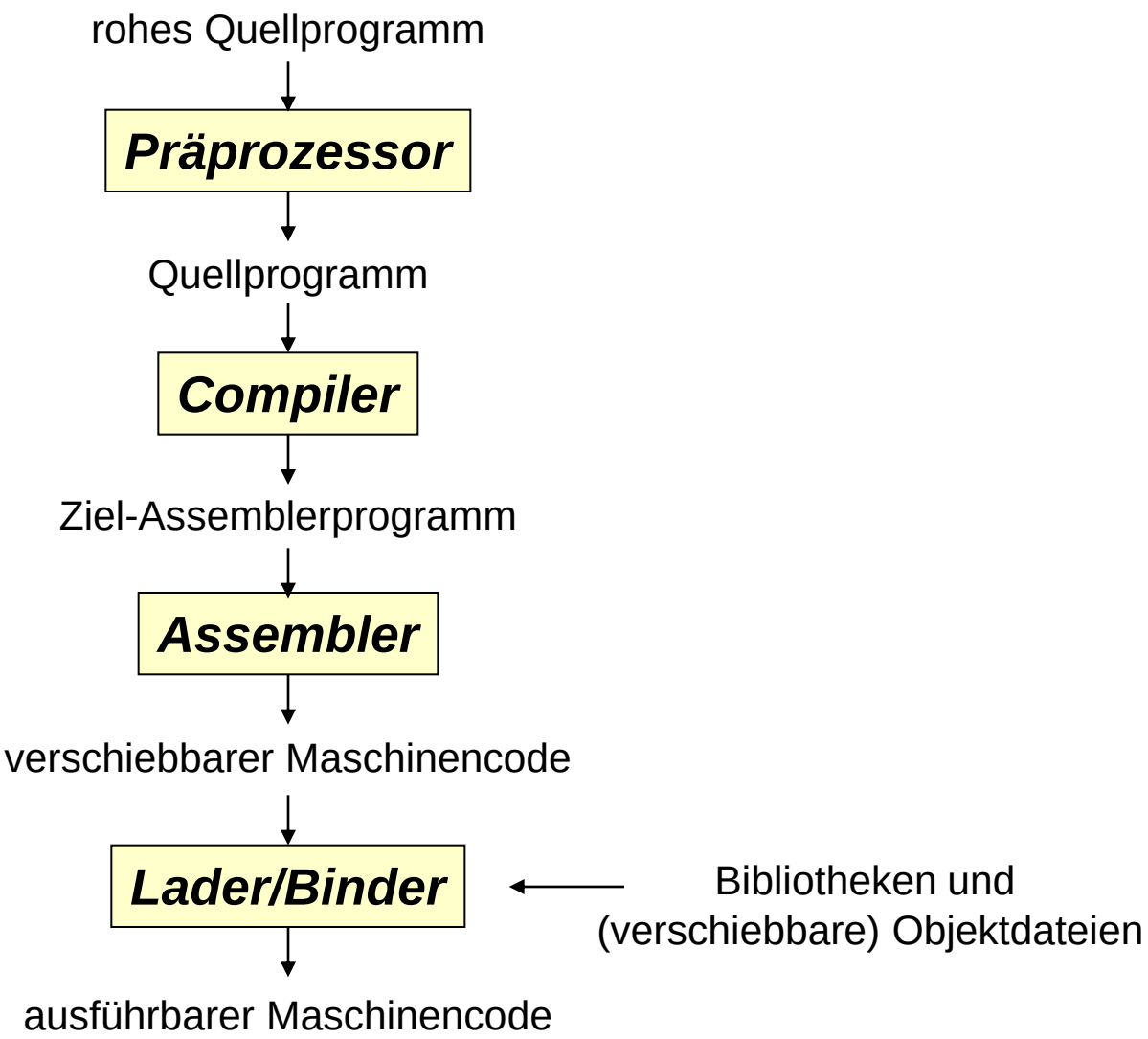
Einführung: Was ist ein Compiler

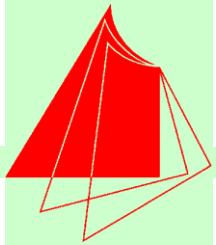
Ein Compiler ist ein Programm, das ein Programm einer bestimmten Sprache (Quellsprache) in ein äquivalentes Programm einer anderen Sprache (Zielsprache) übersetzt.



- 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:

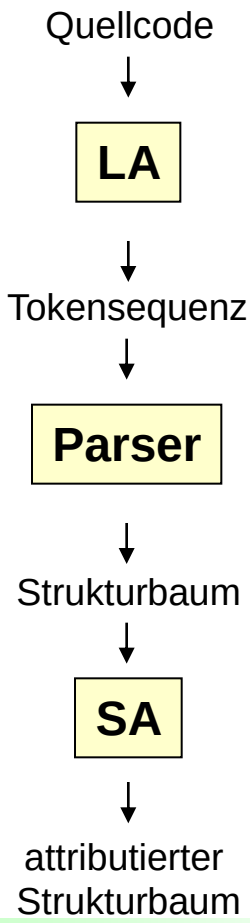
- Analyse: (Frontend)
  Der Analyse-Teil zerlegt das Quellprogramm in seine Bestandteile und erzeugt eine Zwischendarstellung

- Synthese: (Backend)
  Der Synthese-Teil konstruiert das gewünschte Zielprogramm aus der Zwischendarstellung
  - Code-Erzeugung
  - Optimierung

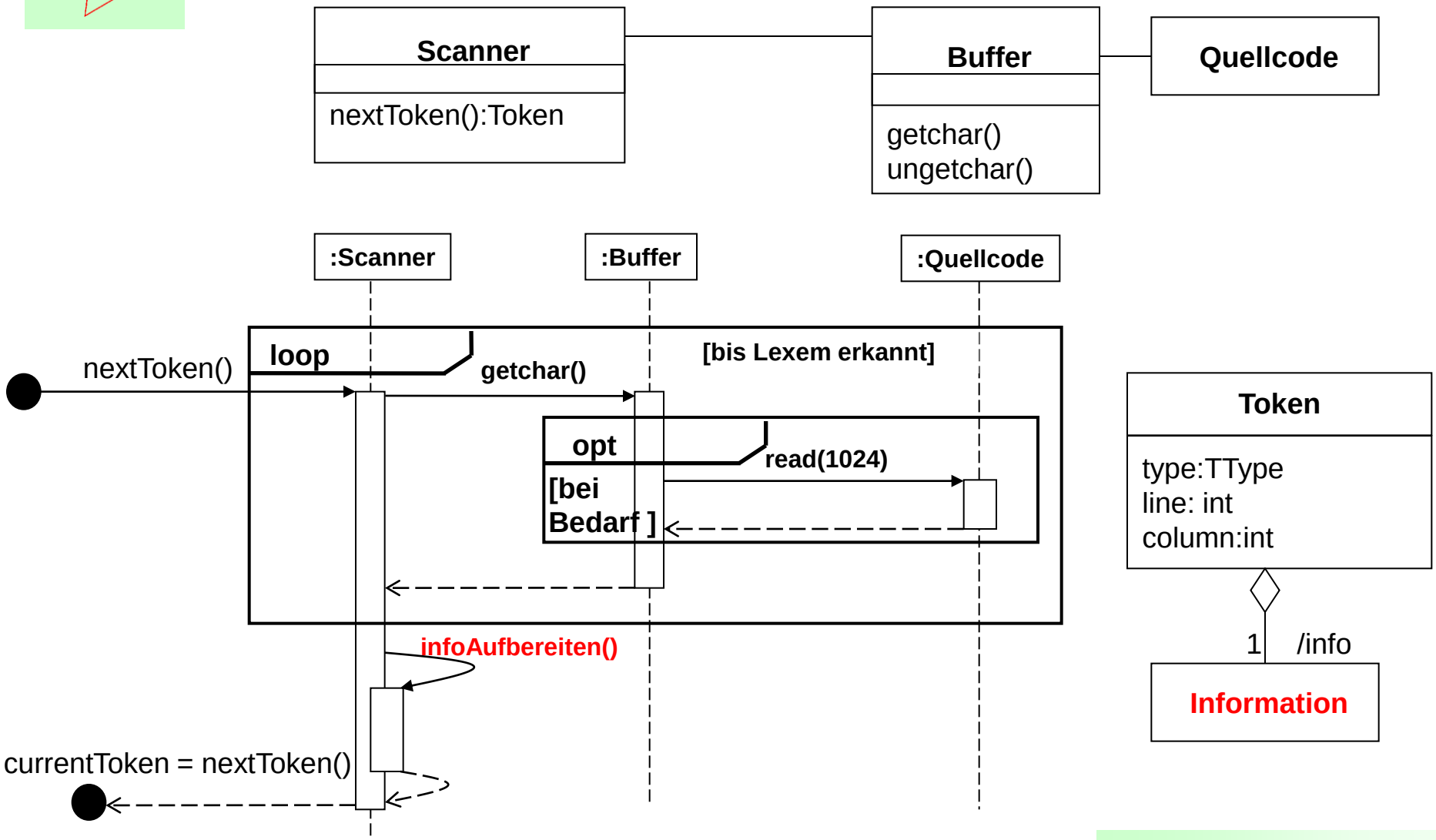
Die Analyse

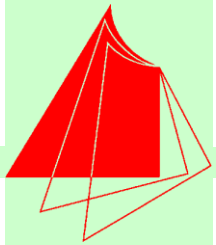
Die Analyse besteht ihrerseits aus mehreren Teilaufgaben

- **lexikalische Analyse**
  Zerlegung des Quellcodes in die Grundsymbole (Tokens) 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



Arbeitsweise eines Scanners





Lexikalische Analyse (Scanning)

Zerlegung des Quellcodes in seine Grundsymbole

Programmiersprache

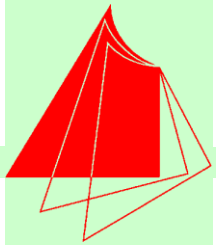
position := initial + rate * PI;

	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	–	–

Textformatierer

die lexikalische Analyse:

	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, die im Quellcode benutzten Bezeichner zu speichern, und Informationen über die verschiedenen Attribute zu sammeln wie Typ, Gültigkeitsbereich, Namen, Anzahl der Argumente usw.

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

Dies ist die Aufgabe der Symboltabelle

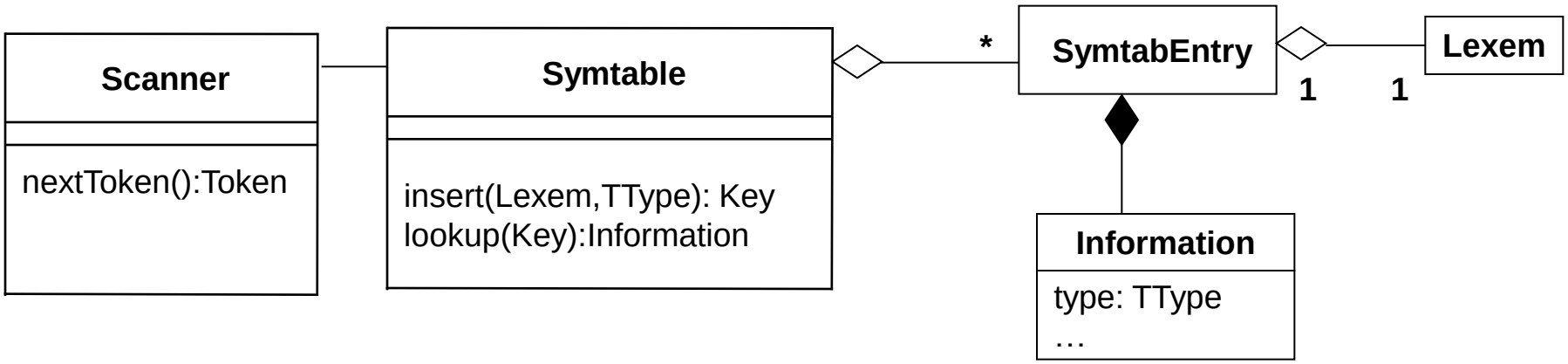
Die Symboltabelle

In der Symboltabelle wird immer dann gesucht, wenn im Quelltext ein Name gefunden wurde. Änderungen treten in der Tabelle auf, wenn ein neuer Name oder neue Informationen über einen existierenden Namen erkannt werden.

Hierzu stehen zwei Operationen zur Verfügung:

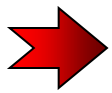
insert(Lexem,TokenType) : Key dieser **Key** **wird im Token gespeichert**

lookup(Key) : Information



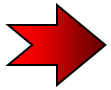
Alles ist kostbar

- Speicherplatz (nicht wirklich)
- Zeit (auf jeden Fall)



Man benötigt geeignete Strukturen zur Verwaltung der Objekte

- **Lexeme** sind beliebig lang, es gibt kurze „x“, „i“, aber auch sehr lange wie „ BufferedOutputStream“



- **keine festen Char-Vektoren, Arrays oder Strings**
- **keine dynamischen Strukturen (hoher Verwaltungsaufwand nicht notwendige Lexeme werden nicht gelöscht)**

Stringtabellen für Lexeme

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

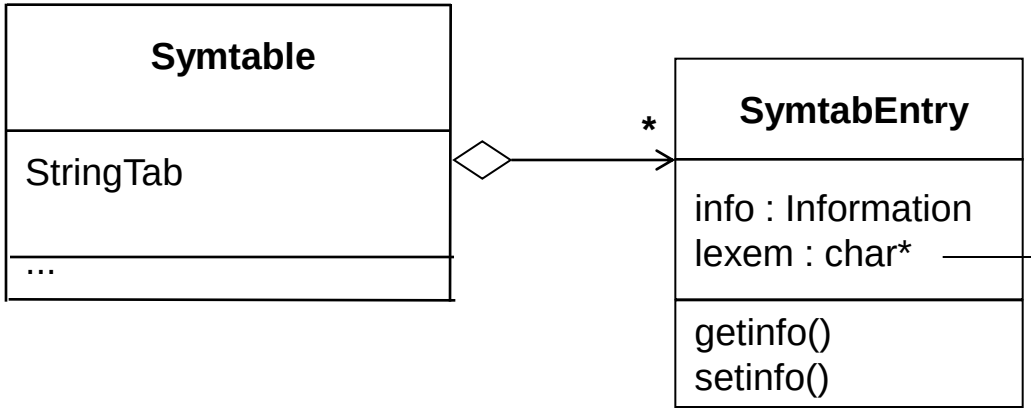
position := initial + rate * PI

mit Längenangabe

8	p	o	s	i	t	i	o	n	7	i	n	i	t	i	a	l	4	r	a	t	e	2	P	I	...

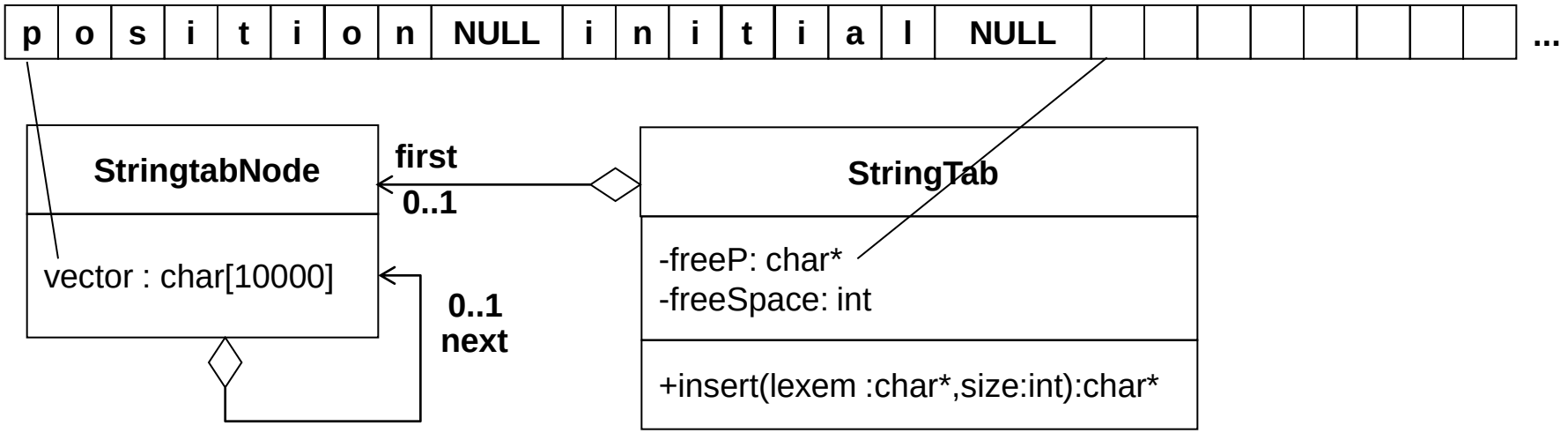
mit Endmarke

p	o	s	i	t	i	o	n	NULL	i	n	i	t	i	a	l	NULL	r	a	t	e	NULL	P	I	...



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;}
```

Auffinden von Lexemen

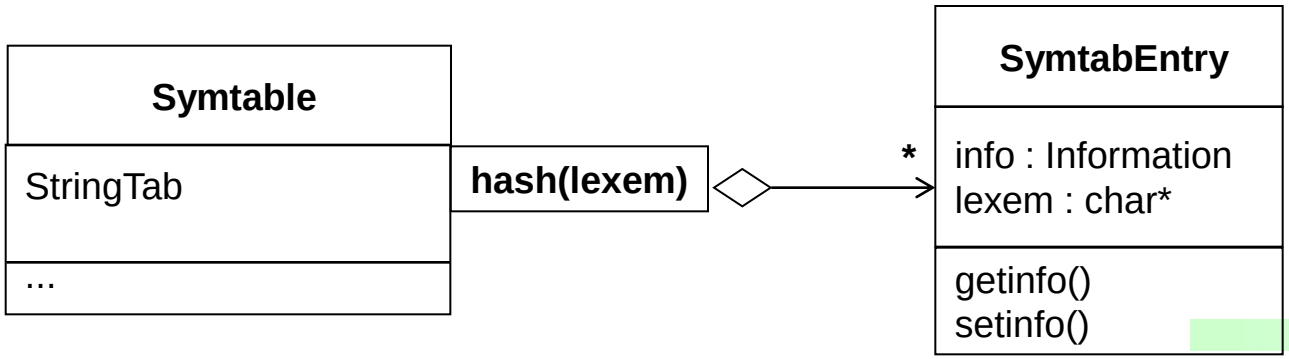
Wichtig ist auch, dass man schnell entscheiden kann, ob ein Lexem neu oder bereits in der Tabelle vorhanden ist.

Hierfür gibt es mehrere Möglichkeiten:

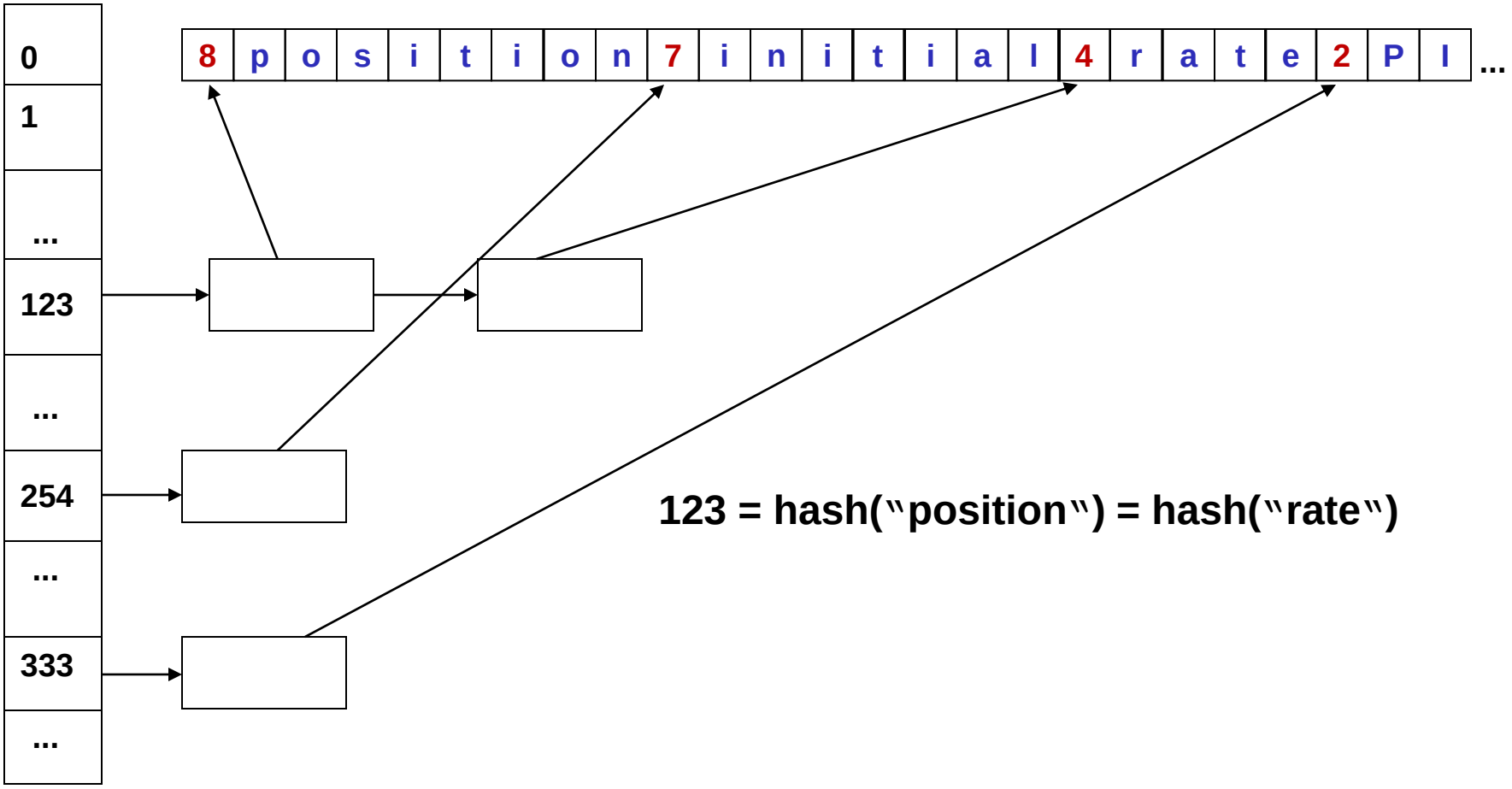
- **Alle SymtabEntries sind sortiert (nach dem Lexem) und man sucht den Eintrag mit dem passenden Lexem (Aufwand O(log n) wobei n die Anzahl der Lexeme ist).**

  Nachteil: Das Einfügen neuer Lexeme ist relativ aufwendig.

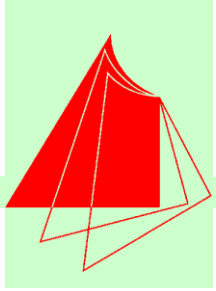
- **Symtable wird als Hashtabelle realisiert (z.B. mit Kollisionsauflösung durch Verkettung)**



Hashtabellen und Funktionen



Symboltabelle



Hashtabellen und Funktionen

Die ideale Hashfunktion hat keine Kollisionen, eine große Streuung und kaum Berechnungsaufwand

Einige Hashfunktionen:

a) $f(a_1 \ldots a_n) = c_1 - ord(`A`)$

b) $f(a_1 \ldots a_n) = (c_1 + c_n) \bmod m$

c) $f(a_1 \ldots a_n) = \sum_{i=1}^{n} c_i \bmod m$

d) $f(a_1 \ldots a_n) = (16\, c_1 + 8\, c_n + n) \bmod m$

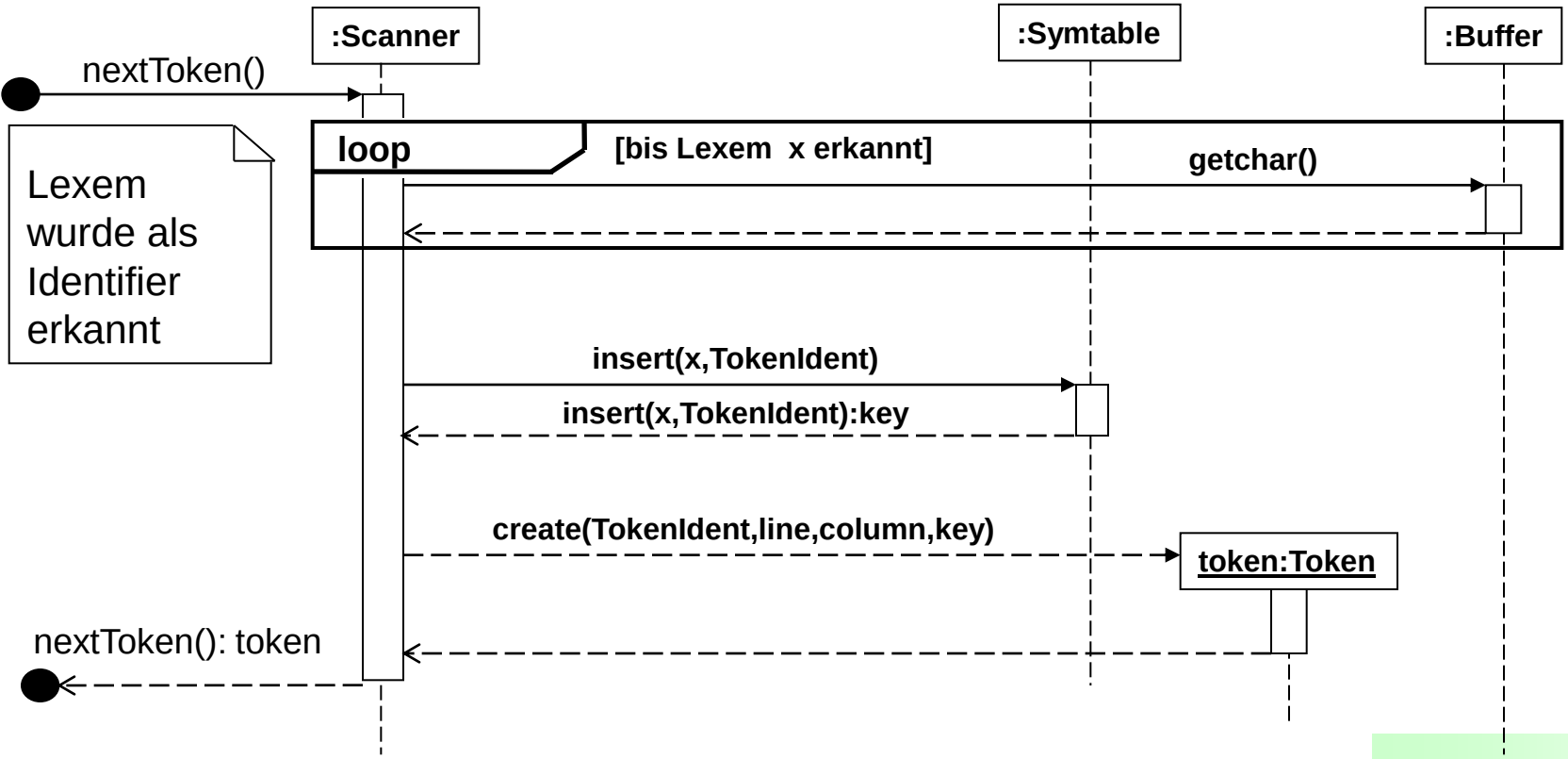
f) $f(a_1 \ldots a_n) = (c_1 + 11\, c_n + 26n) \bmod m$

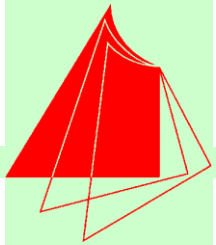
	Kollisionen	Streuung	Aufwand
ideal:	–	+	–
a)	+	–	–
b)	0	0	–
c)	–	+	+
d)	–	+	–
e)	–	+	–

ci = ASCII-Code für ai

Doch wie erkennt man ein Lexem?

Nochmals zur Aufgabe: Der Scanner liest nach der Aufforderung nextToken() solange mittels getchar() das nächste Zeichen, bis er ein Lexem gefunden hat. In der Symboltabelle wird dann die entsprechende Information zum Lexem angefordert. Falls das Lexem neu war, wird es eingetragen.





Etwas Theorie

Im wesentlichen besteht die Aufgabe darin, Wörter einer Sprache zu erkennen.

Was ist eine Sprache?

trivial!

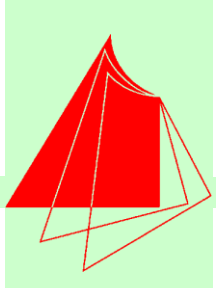
Eine Menge von Wörtern

Was ist ein Wort?

Eine Folge von Symbolen (String) $s=a_1...a_n$

Was ist eine Symbol a?

Eine Element einer Menge $a \in \Sigma$



Operationen auf Mengen

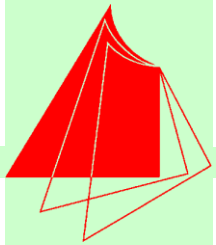
- $\cup$ Vereinigung (klar) $X \cup Y = \{s \mid s \in X \text{ oder } s \in Y\}$
- $\cap$ Schnitt (klar) $X \cap Y = \{s \mid s \in X \text{ und } s \in Y\}$

- Konkatenation $XY = \{st \mid s \in X \text{ und } t \in Y\}$

  Spezialformen:
  - $XX = \{st \mid s \in X \text{ und } t \in X\}$ sei X^2
  - $XXX = \{stu \mid s \in X \text{ und } t \in X \text{ und } u \in X\}$ sei X^3

  offensichtlich ist die aber $X^3 = XX^2$

  Definition $X^i = XX^{i-1}$ für $i > 1$



Operationen auf Mengen

Was gilt für 1?

Fortsetzung der Definition $X = XX^0$

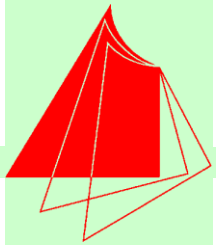
$$\{s \mid s \in X\} = \{st \mid s \in X \text{ und } t \in X^0\}$$

t muss ein Symbol sein, das s nicht verlängert. Man nennt dieses den „leeren String“ und schreibt statt t ε

$x\varepsilon = x$ für alle x und **$X^0 = \{\varepsilon\}$**

Definition

$$X^i = \begin{cases} X^0 & \text{für } i = 0 \\ XX^{i-1} & \text{sonst} \end{cases}$$



Operationen auf Mengen

- Abschluss $X^* = \bigcup_{i \in \mathbb{N}} X^i$ alle Strings endlicher Länge über dem Alphabet X

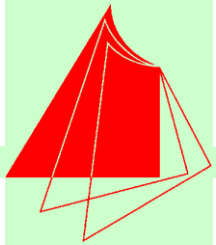
- positiver Abschluss $X^+ = X^* \setminus \{\varepsilon\} = XX^*$

Definition: (Reguläre Sprache)

a) Jede endliche Teilmenge von X^* ist eine reguläre Sprache.

b) Sind L und M reguläre Sprachen, dann sind es auch $L \cup M$, LM und L^*

c) Ist $Y \subseteq X^*$ eine reguläre Sprache, dann kann Y gemäß a) und b) gebildet werden.

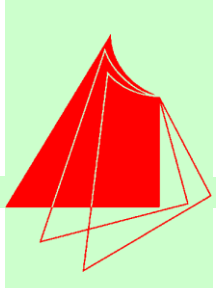


Reguläre Ausdrücke

Dies legt es nahe, reguläre Sprachen konstruktiv über ihrem Alphabet X zu beschreiben.

- ε ist ein regulärer Ausdruck. Es sei $L(\varepsilon) = \{\varepsilon\}$ – die Sprache die nur aus ε besteht
- Ist a ein Symbol aus X ($a \in X$), dann ist auch a ein regulärere Ausdruck. Es sei $L(a) = \{a\}$ – die Sprache, die nur aus a besteht.
- Sind s und t reguläre Ausdrücke, dann ist auch
  - (s) | (t) ein regulärer Ausdruck. Es sei $L((s) \mid (t)) = L(s) \cup L(t)$
  - (s)(t) ein regulärer Ausdruck. Es sei $L((s)(t)) = L(s)L(t)$
  - (s)* ein regulärer Ausdruck. Es sei $L((s)^*) = L(s)^*$
  - (s) ein regulärer Ausdruck. Es sei $L((s)) = L(s)$

Ist $M \subseteq X^*$ eine reguläre Sprache, dann existiert ein regulärer Ausdruck r mit L(r) = M.

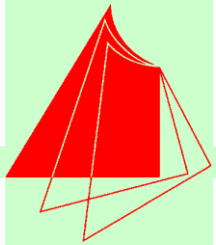


Wieder etwas Praxis

Die Menge aller gültigen Zeichen und Bezeichner einer Programmiersprache ist üblicherweise eine reguläre Sprache. Und kann somit über reguläre Ausdrücke beschrieben werden.

digit	**::=**	`0` **|** `1` **|** `2` **|** `3` **|** `4` **|** `5` **|** `6` **|** `7` **|** `8` **|** `9`
letter	**::=**	`A` **|** `B` **|** `C` **| ... |** `Z` **|** `a` **|** `b` **| ... |** `z` **|**
sign	**::=**	`+` **|** `-` **|** `/` **|** `*` **|** `<` **|** `>` **|** `=` **|** `!` **|** `&` **|** `;` **|** `:` **|** `(` **|** `)` **|** `{` **|** `}` **|** `[` **|** `]`
integer	**::= digit digit***	
real	**::= integer . integer (ε | (E (ε | (+ | -)) integer))**	
Identifier	**::= letter (letter | digit)***	
write	**::=** `write`	
read	**::=** `read`	

Die gültigen Symbole bilden die reguläre Sprache
L(sign | integer | real | identifier| write| read)



Wieder etwas Praxis

Oft schreibt man auch:

{a} statt a* und [a] statt (ε | a)

digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

letter ::= `A` | `B` | `C` | ... | `Z` | `a` | `b` | ... | `z` |

sign ::= `+` | `-` | `/` | `*` | `<` | `>` | `=` | `!` | `&` | `;` | `:` | `(` | `)` | `{` | `}` | `[` | `]`

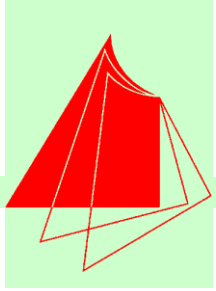
integer ::= digit {digit}

real ::= integer . integer [`E` [`+` | `-`] integer]

Identifier ::= letter {letter | digit}

write ::= `write`

read ::= `read`



Was hat dies mit einem Scanner zu tun?

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

Das heißt:

- nach Abarbeiten eines beliebige Wortes aus L(s) befindet sich der Automat in einem Final-Zustand.
- nach Abarbeiten eines beliebigen Wortes, das nicht aus L(s) ist, befindet sich der Automat nicht in einem Final-Zustand

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 Final-Zustände (F $\subseteq$Q)

Darstellung von Automaten

A:

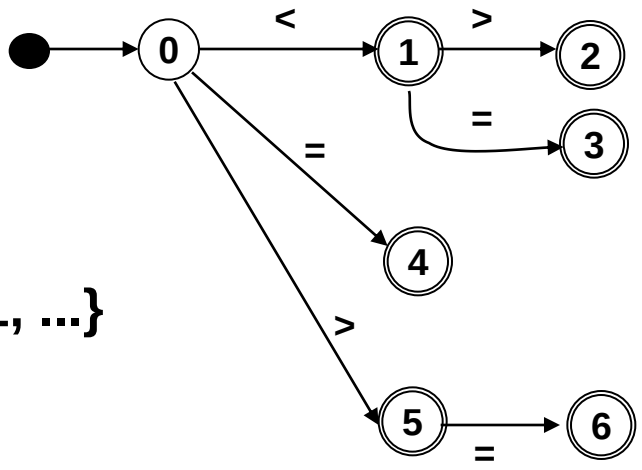
Q = {0,...,6}

S = {<,=,>}

P = {0< ::= 1, ...}

i = 0

F = {1,...,6}



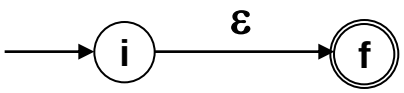
akzeptiert: L(A) = {<,>,=,<=,>=,<>} **dies entspricht der durch den regulären Ausdruck** zeichen **definierten Sprache** L(zeichen)

zeichen ::= < | > | = | <= | >= | <>

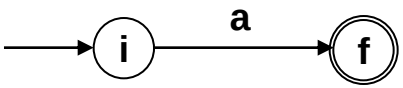
Konstruktion von Automaten

Eingabe: Ein regulärer Ausdruck r über dem Alphabet X,
Ausgabe: Ein Automat A(r) mit L(A(r)) = L(r)

- r ::= ε

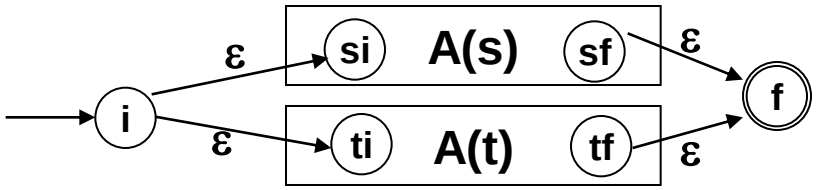


- r ::= a a ∈ X

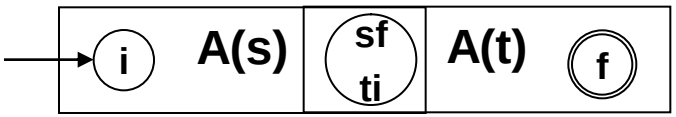


A((r)) = A(r)

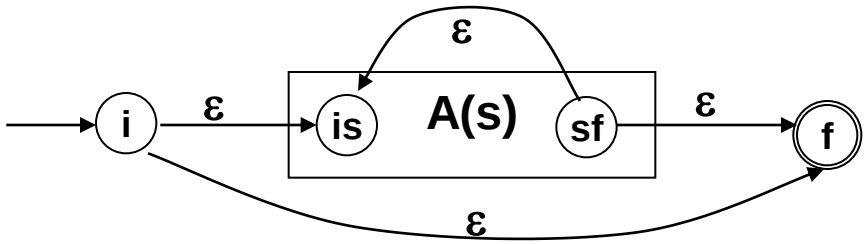
- r ::= s | t



- r ::= st



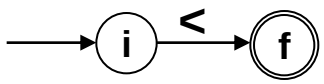
- r ::= s*



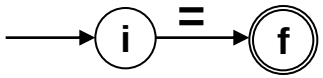
Bsp.:

zeichen ::= < | > | = | <= | >= | <>

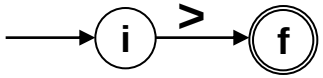
r::= <



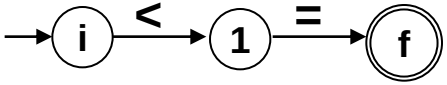
r::= =



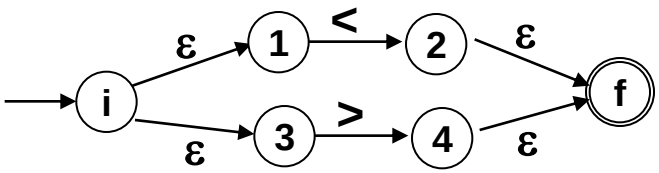
r::= >



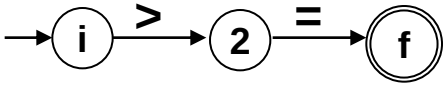
r::= <=



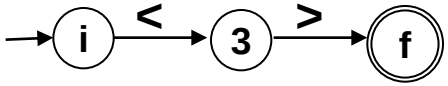
r::= <|>



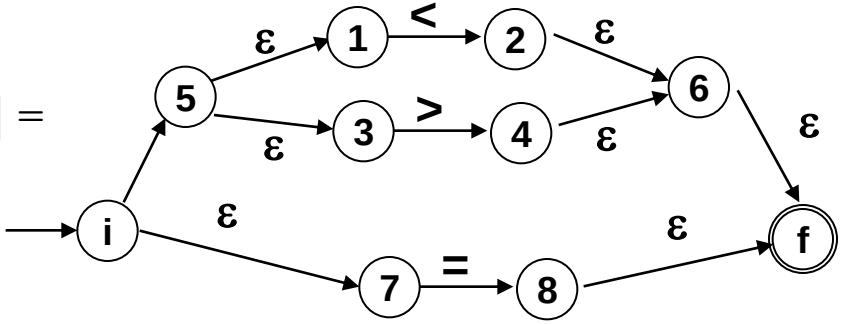
r::= >=



r::= <>



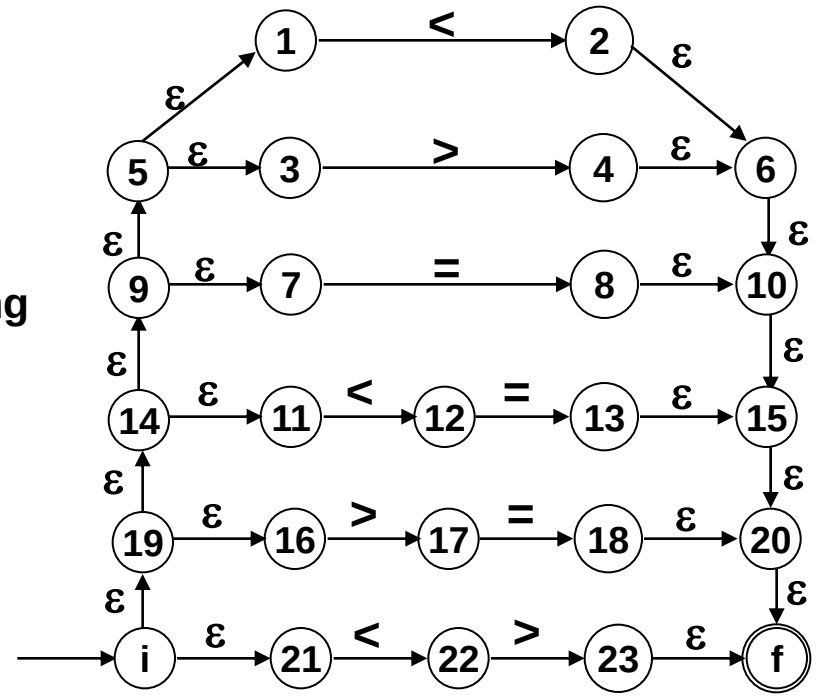
r::= (<|>) | =



Bsp.:

zeichen ::= < | > | = | <= | >= | <>

r::= ((((<|>)|=)|<=)|>=)|<>



Problem: Welchen Übergang soll man wählen?

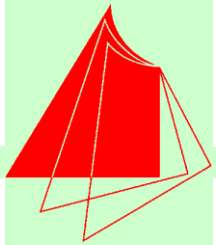
Bsp.: Folge <=

übergang 1 → 2

oder 11 → 12

oder 21 → 22

Wie wird aus diesem Automat der vorherige Automat?



Wiederholtes Zusammenfassen

Man konstruiert wiederholt verschiedene Mengen

Closure(t) = {s | s ist von t nur über e-Übergänge erreichbar oder s=t}

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

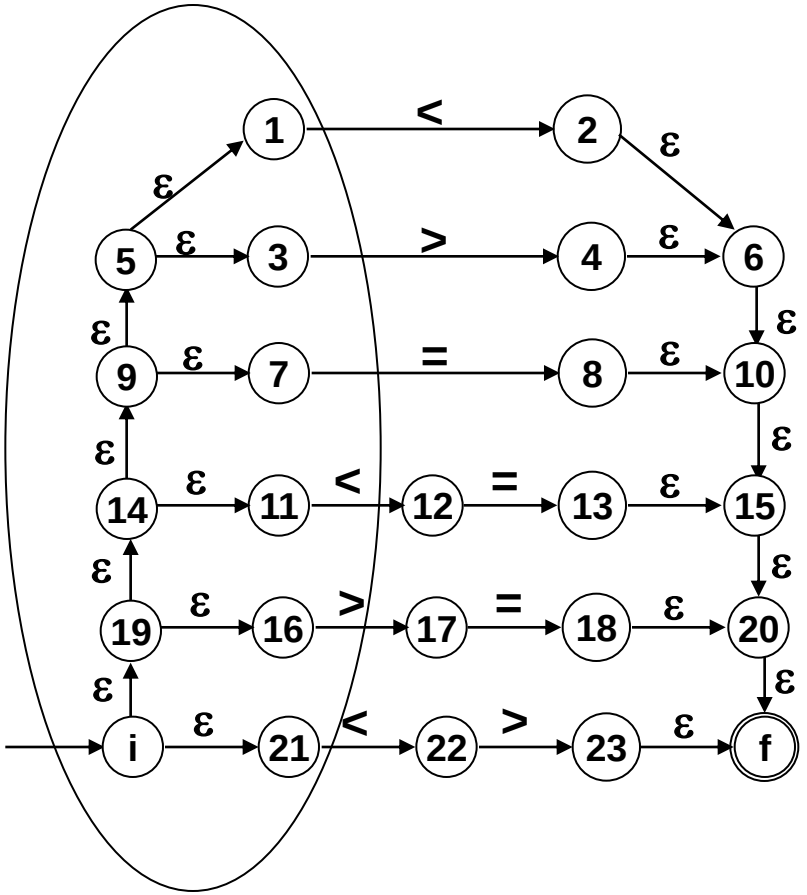
Move(S,a) = { t | es gibt eine Regel sa ::=t $\in$ P und s $\in$ S}

1. Neuer Startzustand O = Closure(i)
2. Bestimme für alle Symbole s Closure(Move(O,s))
   daraus ergeben sich maximal #s neue Zustände Z1 ... Z#s

   Wiederhole 2 für alle neuen Zustände bis keine neuen Zustände mehr hinzu kommen
3. Bestimme die neuen Finalzustände (alle Mengen die einen alten Finalzustand enthalten.)

Beispiel

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



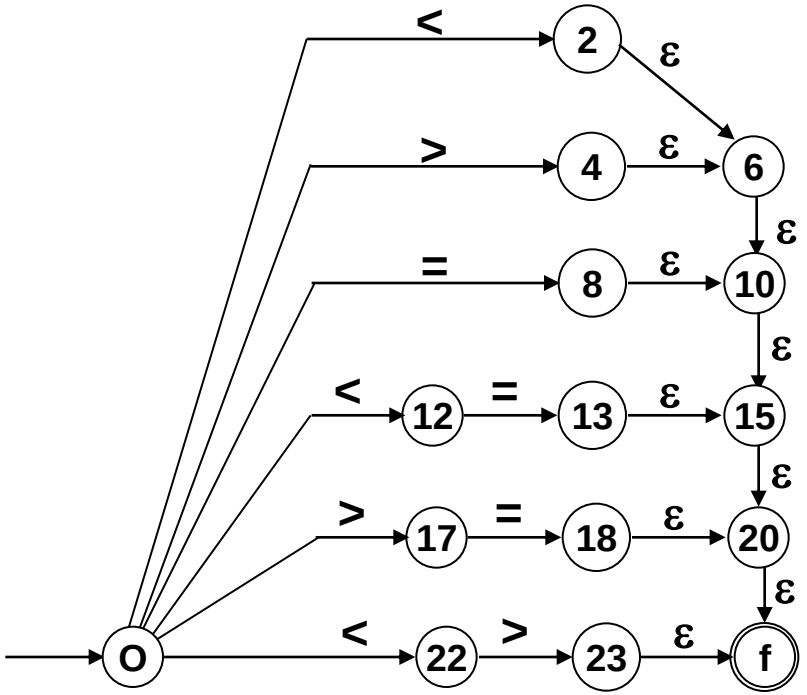
Beispiel

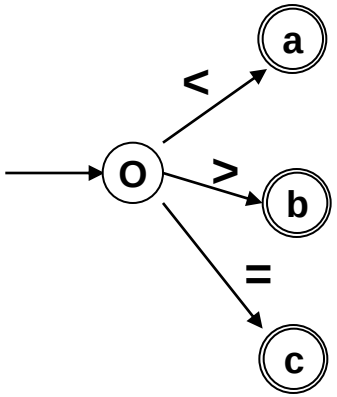
O = {i,1,3,5,7,9,11,14,16,19,21}

2. move(O,<) = {2,12,22}
   a= closure(move(O,<)) ={2,6,10,12,15,20,22,f}

   move(O,>) = {4,17}
   b = closure(move(O,>)) ={4,6,10,15,17,20,f}

   move(O,=) = {8}
   c = closure(move(O,=)) = {8,10,12,15,20,f}





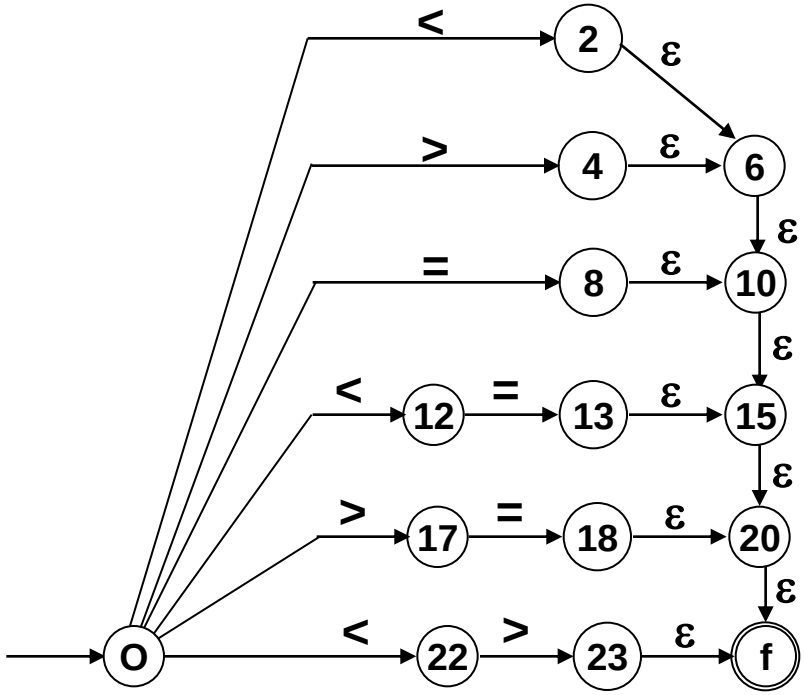
Beispiel

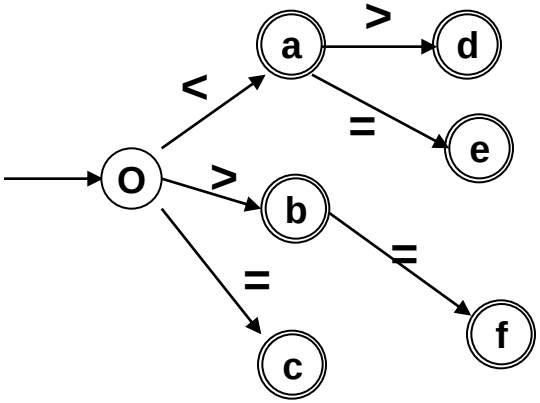
O = {i,1,3,5,7,9,11,14,16,19,21}

a={2,6,10,12,15,20,22,f}
closure(move(a,<)) = {}
d = closure(move(a,>)) = {23,f}
e = closure(move(a,=)) = {13,15,20,f}

b={4,6,10,15,17,20,f}
closure(move(b,<)) = {}
closure(move(b,>)) = {}
f = closure(move(b,=)) = {18,20,f}

c={8,10,15,20,f}
closure(move(b,<)) = {}
closure(move(b,>)) = {}
closure(move(b,=)) = {}





Beispiel

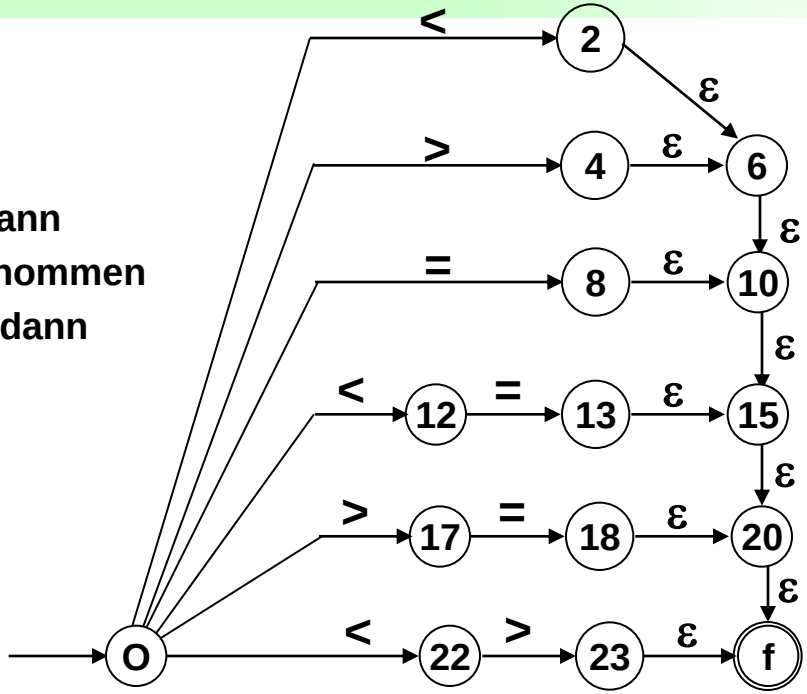
O= {i,1,3,5,7,9,11,14,16,19,21}
a={2,6,10,12,15,20,22,f}
b={4,6,10,15,17,20,f}
c={8,10,15,20,f}

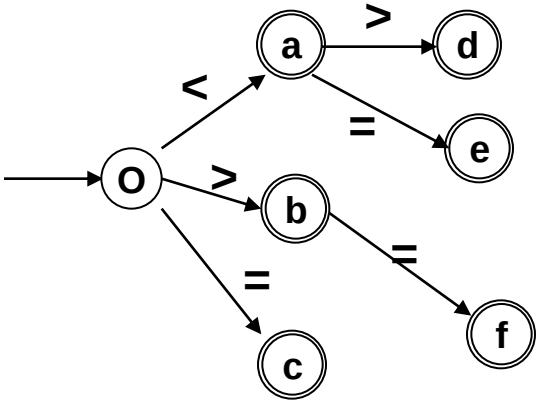
d={23,f}
closure(move(d,<)) = {}
closure(move(d,>)) = {}
closure(move(d,=)) = {}

e={13,15,20,f}
closure(move(e,<)) = {}
closure(move(e,>)) = {}
closure(move(e,=)) = {}

f={18,20,f}
closure(move(f,<)) = {}
closure(move(f,>)) = {}
closure(move(f,=)) = {}

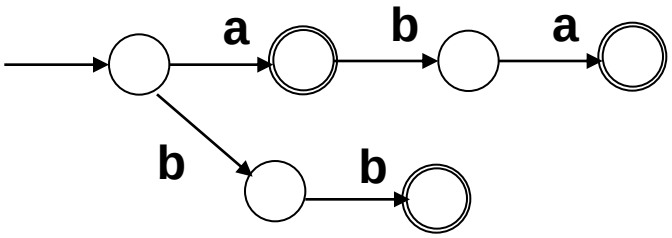
Bem.: Die leere Menge kann als Fehlerzustand aufgenommen werden, der Automat ist dann vollständig.





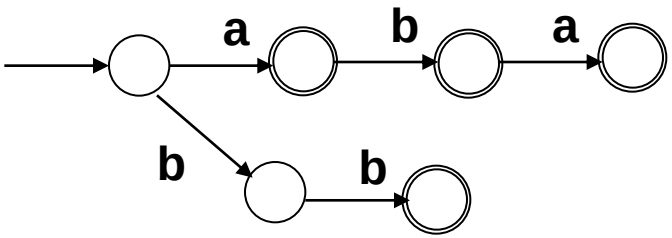
Aufbereiten der Eingabe

Falls ein Text aus mehreren Worten bestehen kann, 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`

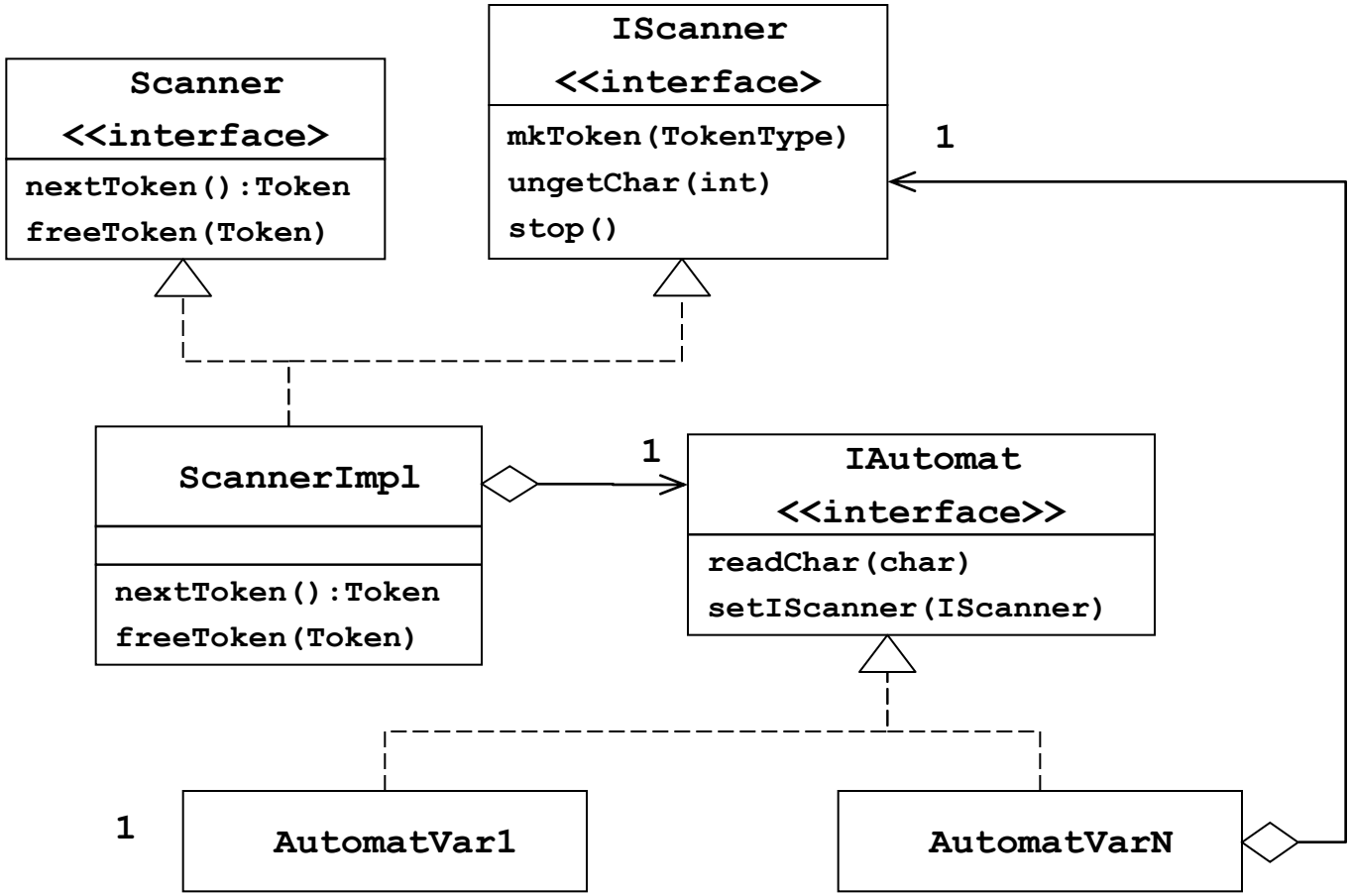
Das Lexem `ab` und ein Fehler.

Die Eingabe muss gepuffert werden.

Eine Pufferung ermöglicht darüber hinaus ein effizientes Einlesen von Daten.

Realisierung von Automaten

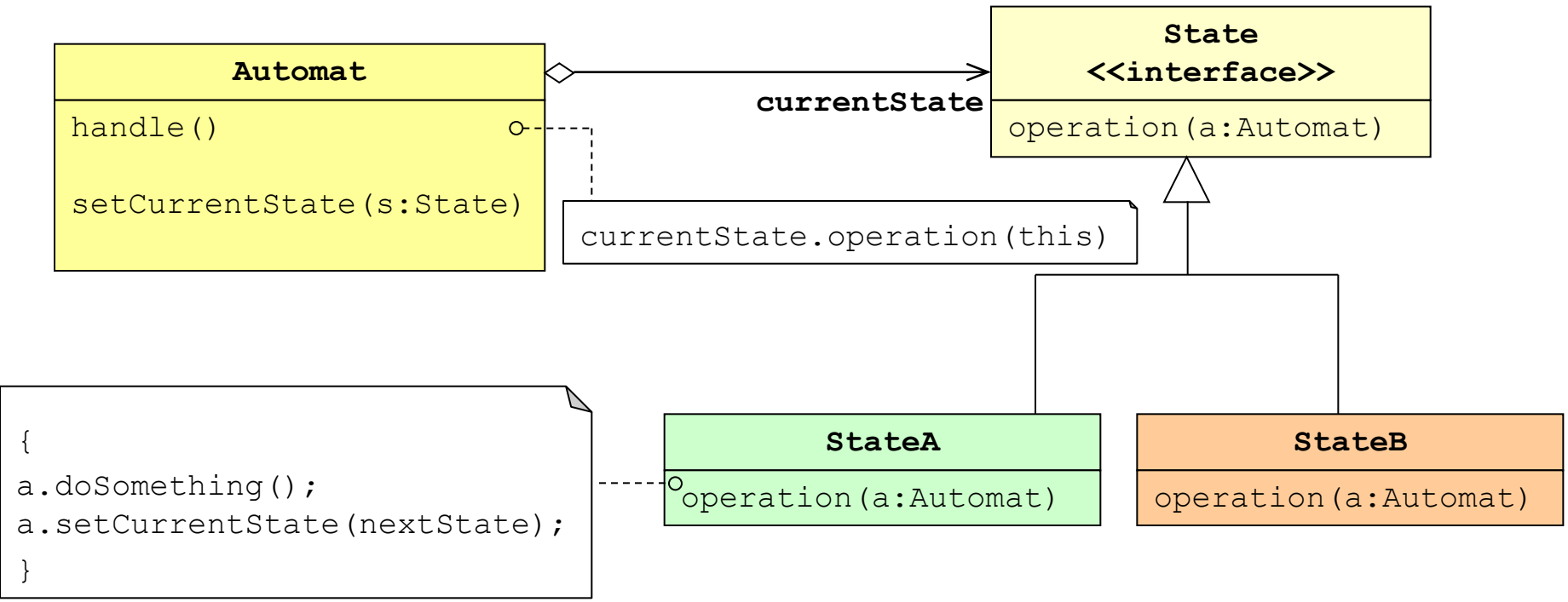
Eine allgemeine und flexible Struktur



Der objektorientierte Automat

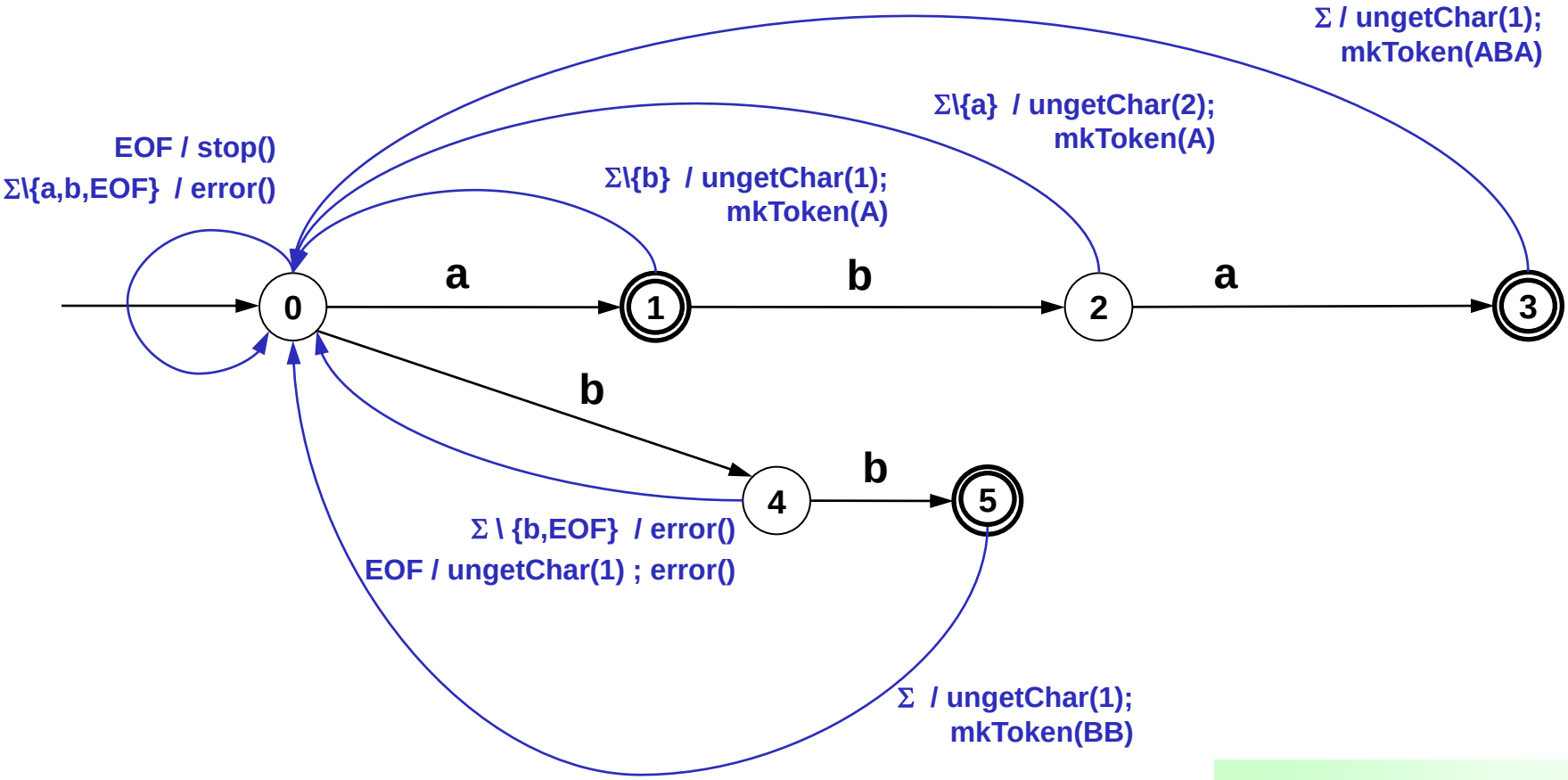
State-Pattern

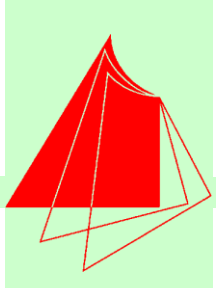
Ermöglicht es einem Objekt sein Verhalten zu ändern, wenn der interne Zustand sich ändert.



Operationen bestimmen

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



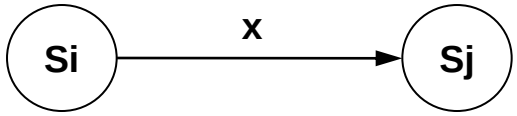


Implementieren

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


void State4::read(char c, Automat* 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();
  }
}
```

Zustandsübergangstabelle



Zustand

Eingabe

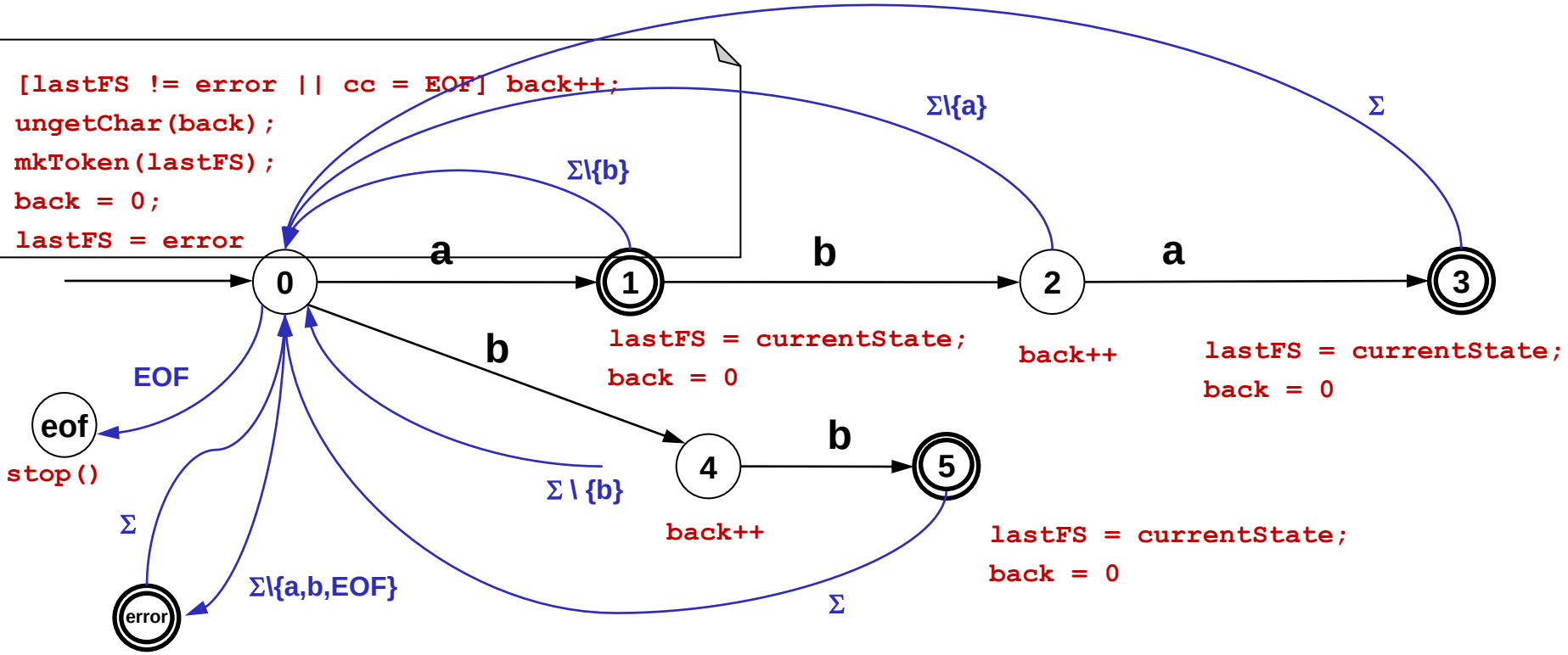
			Si	
x			Sj	

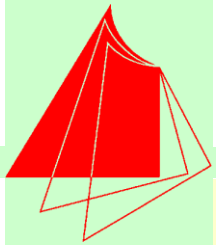
Folgezustand

```
nextState = stateMatrix[currentState][currentChar]
```

Zustände ergänzen

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





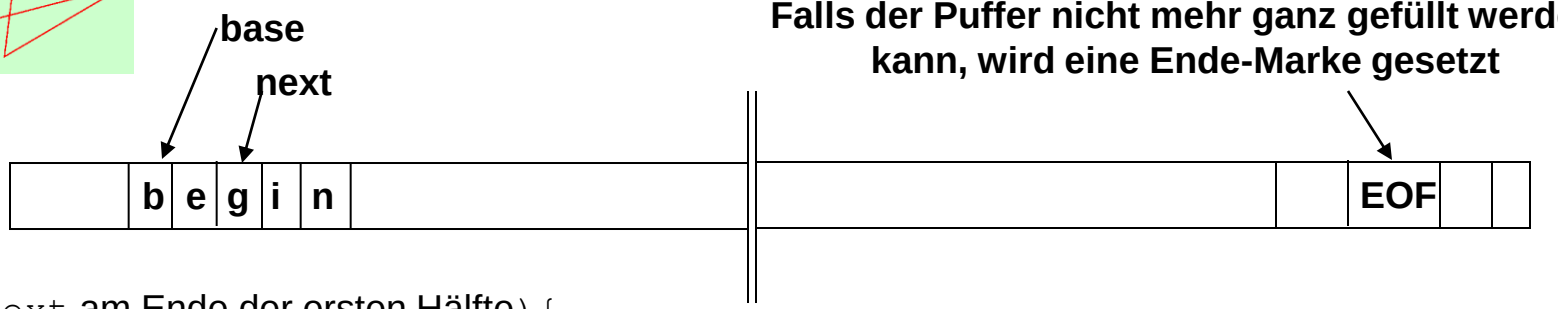
Implementieren

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

```
void Automat::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 != FINAL_STATE_ERROR
            ||currentChar == '\0') ungetChar(this->back);
          mkToken(this->lastFinalState);
          this->back = 0;
          this->lastFinalState = FINAL_STATE_ERROR;
        } else if(this->currentState == STATE_EOF) stop();
  }
}
```

Ein übliches Pufferschema

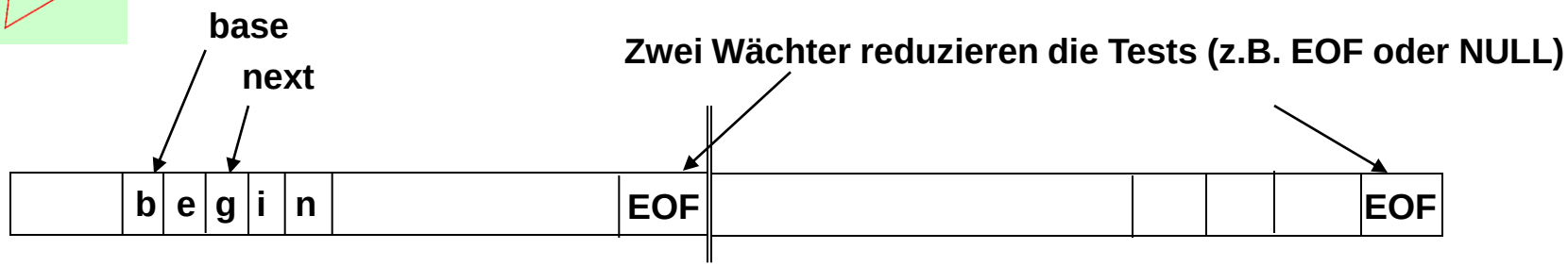



```
if (next am Ende der ersten Hälfte) {
    lade zweite Hälfte neu;
    next++;
}
else { if (next am Ende der zweiten Hälfte) {
            lade erste Hälfte neu;
            next = Anfang erste Hälfte;
        }
        else { if (*next == EOF) {
                lexikalische Analyse beenden;
                }
                else next++
        }
}
```

Problem: 3 Tests

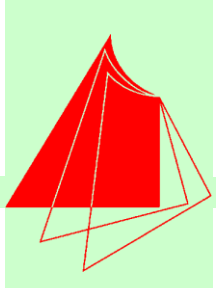
1. **next am Ende der ersten Hälfte**
2. **next am Ende der zweiten Hälfte**
3. ***next == EOF**

Variante mit Wächter




```
if (*next == EOF) {
  if (next am Ende der ersten Hälfte){
    lade zweite Hälfte neu;
    next++;
  }
  else if (next am Ende der zweiten Hälfte){
    lade erste Hälfte neu;
    next = Anfang erste Hälfte;
  }
  else lexikalische Analyse beenden
}
next++
```

Pufferende und Datenende werden durch das gleiche Zeichen dargestellt.



Aufgabe:

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

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

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

sign ::= + | - | / | * | < | > | = | ! | & | ; | : | (|) | { | } | [|]

integer ::= **digit digit***

real ::= **integer . integer (ε | (E (ε | (+ | -)) integer))**

Identifier ::= **letter (letter | digit)***

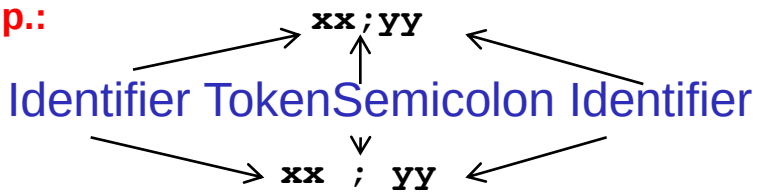
write ::= write

read ::= read

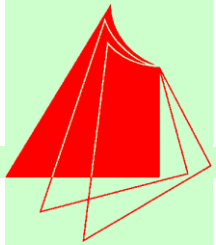
Achtung:
Auch für jedes „sign“ muss ein eigenes Token erstellt werden.

Hinweise:
Eine Trennung der einzelnen Worte mittels Leerzeichen ist nicht notwendig aber erlaubt.

Bsp.:



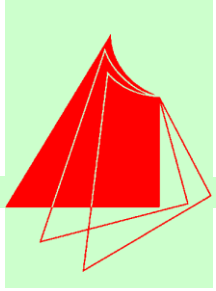
Kommentare sind zugelassen und sollen wie Leerzeichen behandelt werden. (* Dies ist eine Kommentar! Alles ist erlaubt! Kommentare können mehrzeilig sein! *)



Aufgabe: Details

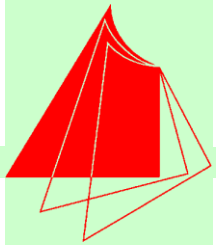
Machen Sie sich über die Lösung der Aufgabe Gedanken und zerlegen Sie die Aufgabe in mindestens drei Teile:

1. **Puffer (I/O)**
2. **Symboltabelle**
3. **Automat und Scanner**



Automat und Scanner

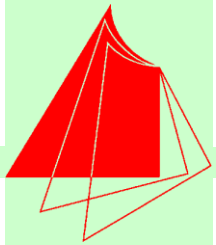
- Entwerfen Sie für den Scanner eine geeignete Schnittstelle, die mindestens die Operation **nextToken(): Token** enthält.
- Erstellen und implementieren Sie einen endlichen (deterministischen) Automaten, der auch die Kommentare verarbeitet und **behandeln Sie Schlüsselworte und Identifier gleich**. (**Kommentare sollen überlesen werden**.)
- Für erkannte Lexeme ist ein Token zu erstellen, das **Zeile**, **Spalte** und **Typ** enthält.
  - Erkannte **Identifier** sind in die Symboltabelle einzutragen und ein Verweis auf diesen Eintrag ist im Token zu speichern.
  - Für erkannte **Integer-** und **Real-Konstanten** (**int** bzw. **real**) ist deren **Wert** zu bestimmen und im Token zu speichern.
    Wandeln Sie Int-Zahlen hierzu nach `long` und Real-Zahlen nach `double`.
    Bereichsüberschreitungen stellen Fehler dar und müssen erkannt werden (nutzen Sie hierzu `strtol` bzw. `strtod` und `error.h`)
- Bei „**Zeichen**“, die nicht zur Sprache gehören, soll ein **Fehlertoken** erzeugt werden. Dieses beinhaltet neben Zeile und Spalte auch das fehlerhafte Zeichen.



Symboltabelle

- Entwerfen Sie für die Symboltabelle eine geeignete Schnittstelle.
- Realisieren Sie die **Kollisionsauflösung** in Ihrer Hash-Tabelle durch **Verkettung**.
- Allozieren Sie Speicherplatz stets en gros.
- Zur **Vorbelegung** der Symboltabelle implementieren Sie die Operation **`initSymbols()`**, die die Symboltabelle mit Schlüsselworten füllt.

```
void Symtable::initSymbols(){ insert("write",TokenWrite); ...}
```

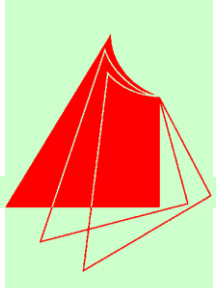


Puffer

- Zur besseren Verarbeitung puffern Sie Ein- und Ausgabedateien. Entwerfen Sie für ihre Puffer eine geeignete Schnittstelle.

- Verwenden Sie ausschließlich die C-Funktionen `CreateFile`, `ReadFile` und `WriteFile` aus dem **Microsoft Platform SDK** (`windows.h`).

- Verwenden Sie `FILE_FLAG_NO_BUFFERING`, um den System-Cache auszuschalten.

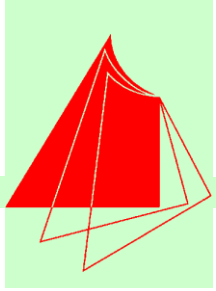


Test-Programm

Schreiben Sie ein Testprogramm, das zwei Dateien entgegennimmt, einen Scanner erzeugt, Tokens anfordert und diese in eine Datei ausgibt, bis die Eingabe vollständig abgearbeitet wurde.

```
int wmain(int argc, wchar_t* argv[], wchar_t *envp[]){
   . . .
   if (argc < 1) return 1;
   Scanner* s = new Scanner(argv[1]);
   Token* t;
   while (t = s->nextToken()){
      //Token ausgeben
   }
}
```

„Test.cpp“



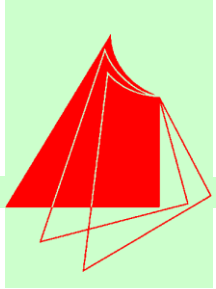
Aufgabe: Aufruf-Syntax (Beispiel)

```
C:\> scanner Scanner-test.txt out.txt
processing ...
unknown Token Line: 2 Column: 1  Symbol: Ä
stop
C:\>
```

```
X = 3 + 4;
Äy = X / (X – 4);
Z = ((3 + 4 – 6));
(* eine einfache Aufgabe !! *)
Resultat = X * y
```

„Scanner-test.txt “

- „**Scanner-test.txt**“ ist eine Eingabedatei, mit dem zu bearbeitenden Text.
- „**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;
Äy = X /(X – 4);
Z = ((3 + 4 – 6));
(* eine einfache Aufgabe !! *)
Resultat = X * y
```

out.txt

```
Token Identifier    Line: 1 Column: 1   Lexem: X
Token Assign        Line: 1 Column: 3
Token Integer       Line: 1 Column: 5   Value: 3
Token Plus          Line: 1 Column: 7
Token Integer       Line: 1 Column: 9   Value: 4
Token Semicolon     Line: 1 Column: 10
Token Identifier    Line: 2 Column: 2   Lexem: y
…
Token Identifier    Line: 5 Column: 16  Lexem: y
```