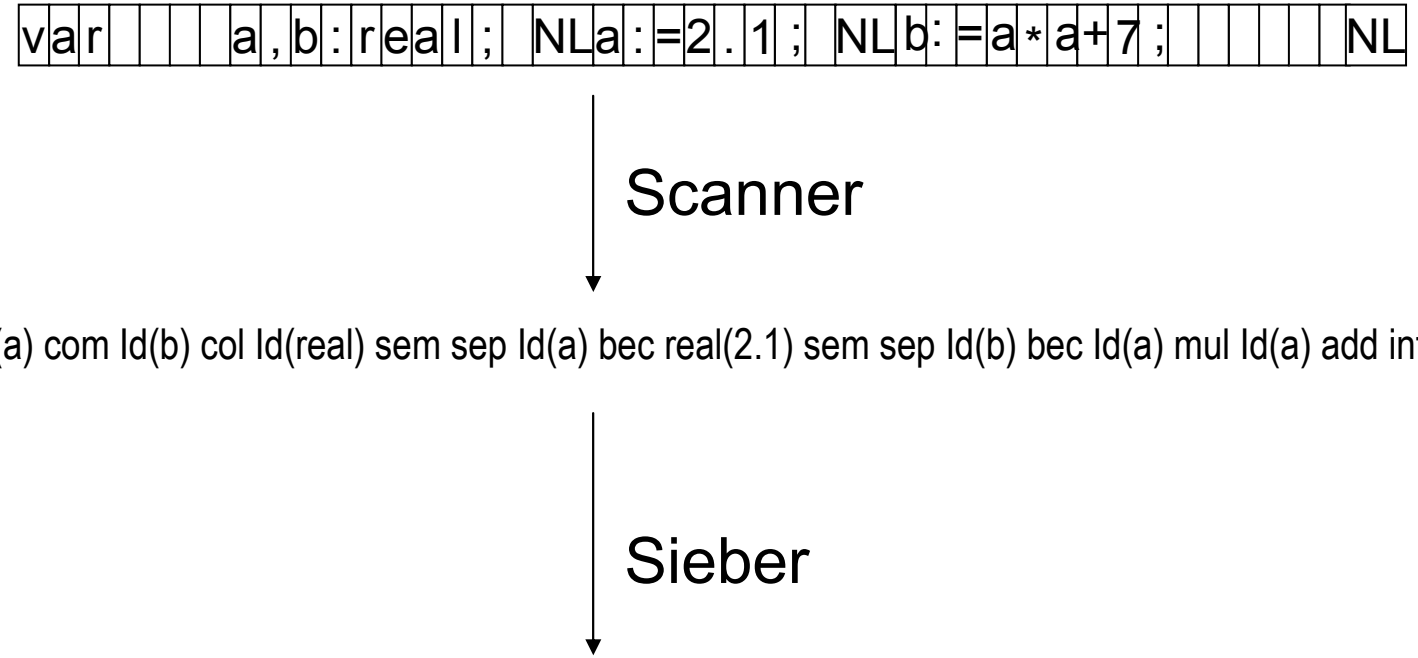
4. Die lexikalische Analyse

- zerlegt Folge von Zeichen in Eingabedatei in Folge von Symbolen (Token)
- Scanner-Sieber-Modul
- Token: Typ und Inhalt
- übliche Token-Typen:
  - reservierte Wörter (if, while, for,…)
  - Bezeichner (x, dauer, name, ..)
  - Literale für ganze Zahlen, Gleitkommazahlen, Strings, Zeichen
  - Trennzeichen (Komma, Semikolon)
  - Klammern
  - Operatoren (=, +, -, &&, …)

4. Die lexikalische Analyse



4. Die lexikalische Analyse

Tokenklassen:

- Menge aller möglichen Werte : reguläre Sprache
- beschrieben durch regulären Ausdruck

Wiederholung/Definition:

reguläre Sprachen über Alphabet ∑:

- leere Menge Ø, leere Sprache {ε}
- für alle a Є ∑ ist {a} reguläre Sprache
- R, S reguläre Sprachen über ∑, dann auch R*, RS, R U S

4. Die lexikalische Analyse

Wiederholung/Definition:

reguläre Ausdrücke (RA) über Alphabet ∑:

- Ø ist RA, beschreibt Sprache Ø
- ε ist RA, beschreibt Sprache {ε}
- a Є ∑ ist RA, beschreibt {a}
- r, s RA, die R bzw. S beschreiben, dann
  - (r | s) RA, der R U S beschreibt
  - (r s) RA, der RS beschreibt
  - (r)* RA, der R* beschreibt

Beispiel:

ab*a	beschreibt {a}{b}*{a}	aa, aba, abba, …
(ab)*	beschreibt {ab}*	ε, ab, abab, …

4. Die lexikalische Analyse

Aufgaben:

Erstellen eines regulären Ausdrucks für folgende Sprachen

Alphabet sei ∑ ={a, b}

1. Sprache aller Wörter, die mit a anfangen und mit b enden?
2. Sprache aller Wörter, die wenigstens 3 a enthalten?
3. Sprache aller Wörter, die mind. ein aa oder ein bb enthalten?

Welche Sprache wird durch folgenden RA beschrieben?

1. a|b
2. a(aa)*b*
3. (ab)*|(ba)*

Gegeben sei das Alphabet ∑ ={0,..,9,+,-,.}

Welcher RA beschreibt die Sprache aller Dezimalpunktzahlen?

4. Die lexikalische Analyse

Endliche Automaten:

- math. Maschinen zum Erkennen reg. Sprachen
- bestehen aus (∑,Q,S,F,T)
  - Eingabealphabet ∑
  - endliche Zustandmenge Q
  - Startzustand S Є Q
  - Menge an Endzuständen F aus Q
  - Übergangsrelation T aus (Q x ∑ x Q)
- Beispiel

  ∑={a,b}, Q={z0,z1,z2,z3},
  S={z0},F={z3}, T=

Q-alt	Zeichen	Q-neu
z0	a	z0
z0	b	z1
z1	a	z0
z1	b	z2
z2	a	z0
z2	b	Z3
z3	a	z3
z3	b	z3

Welche Sprache erkennt der Automat?

Aufgabe: Implementierung in Java, C

4. Die lexikalische Analyse

Endliche Automaten als Scanner (1):

- reguläre Ausdrücke für alle Token(klassen) erstellen
- zugehörigen NEA konstruieren
- minimalen DEA ableiten (Verfahren von Thompson)
- DEA in Programmiersprache implementieren (s. letzte Übungsaufgabe)

4. Die lexikalische Analyse

Endliche Automaten als Scanner (2):

- startet auf dem ersten noch nicht gelesenen Zeichen
- Folge von Schritten abhängig vom Zustand und vom gelesenen Zeichen
- meldet Finden eines Symbols, wenn in Endzustand und zu nächstem Zeichen kein Zustandsübergang existiert
- existiert kein Übergang ohne Endzustand: letzte Übergänge rückgängig machen bis zuletzt durchlaufenen Endzustand

4. Die lexikalische Analyse

Endliche Automaten als Scanner (3):

Tokenklassen für „richtige“ Programmiersprache z.B.

- Bezeichner: (a|…|z|A|…Z)(a|…|z|A|…Z|0|…|9)*
- Integer: (1|2|…|9)(0|1|2|…|9)*
- Float: (1|…|9)(0|…|9)*.(0|…|9)*
- Operatoren: (+|-|*|/|=|….)
- Anweisungen: if | then | else | for | while | …
- Kommentar: #(a|..|z|A|..|Z|_|-|:|…)#
- Separator: ;

Aufgabe: Schreiben Sie ein Java- o. C-Programm, welches eine Sprache mit diesen Token scannt und den Namen der jeweiligen Tokenklasse ausgibt.

4. Die lexikalische Analyse

lex, flex:

- Scanner-Generierungstool
- Input: Datei folgenden Aufbaus (bsp.l)

```
Definitionen
%%
lex-Regeln
%%
benutzerdefinierte
Routinen
```

- Generierung mittels flex bsp.l
- Output: Datei namens lex.yy.c mit Funktion yylex()
- yylex() in C-Dateien nutzen;
  gcc prog.c lex.yy.c –lfl –o prog

4. Die lexikalische Analyse

Definitionen-Teil:

1. Festlegung der Wirtsprache (C, Ratfor): %C bzw. %R
2. C- oder Ratfor-Programmteile
3. C- oder Ratfor-Programmteile in %{ ….%}
4. Reguläre Definitionen
5. Startbedingungen
6. Zeichensatz-Tabellen
7. Verstellen von lex-internen Tabellengrößen

4. Die lexikalische Analyse

C- bzw. Ratfor Programmteile:

entweder

- erstes Zeichen pro Zeile Leerstelle oder Tab
- dahinter C-/Ratfor-Deklarationen/Definitionen
- werden unverändert an globalster Stelle in generiertes Programm übernommen
- Beispiel:

```
        int operand[10];
        char op;
%%
....
```