Stackbasierte Sprachen

Gruppe 101

Gerald Scharitzer
Christian Schwind
Bernhard Wachter

Graph des NFA

- Die Knoten und Kanten werden auf dem Heap gespeichert.
- Jede Kante besteht aus 2 Zellen.
- Die 1. Zelle enthält den Execution Token.
- Die 2. Zelle enthält den Folgeknoten.

Execution Token 1

- Jede Epsilon-Kante wird in den XT -1 kompiliert, welch nicht ausgeführt wird.
- Jede andere Kante wird in einen XT kompiliert, welcher 1 Zeichen vom Stack konsumiert und TRUE liefert, wenn die Kante passt.

```
: is-a ( c "name" -- )
  create here 1 chars allot c!
 DOES> ( c c-addr -- f ) c@ = ;
: match. ( c -- f ) drop TRUE ;
```

Execution Token 2

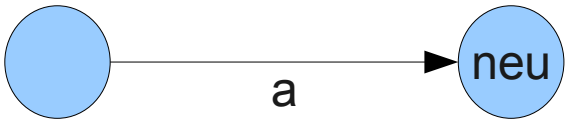
- Bereiche ([a-z][0-9]...) werden als Arrays gespeichert, welche durch das aktuelle Zeichen indiziert werden.

```
: Charclass ( -- )
  create here 256 chars allot 256 chars erase
  DOES> ( c c-addr -- f ) + c@ 1 =
;
```

Aufbau des NFAs

- Beginngraph:

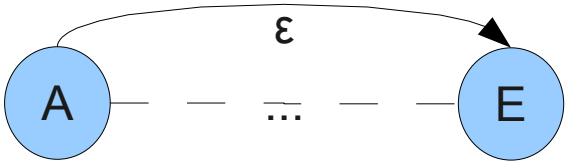
- für Buchstaben:



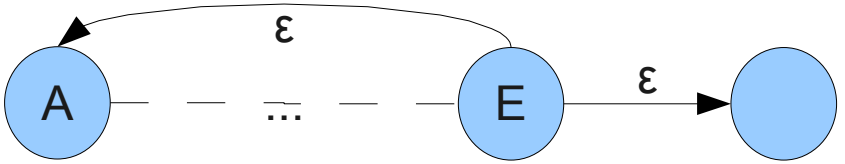
- für (oder):

Implementierung von +,?,*

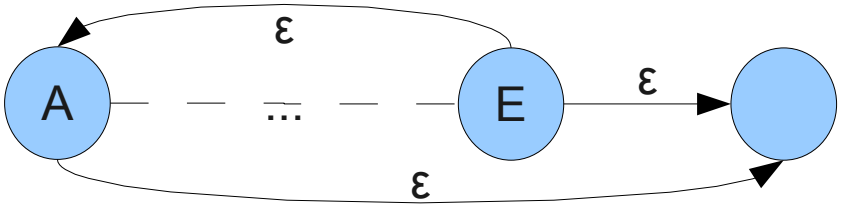
- „(…)?“



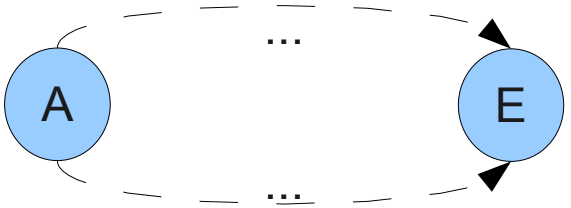
- „(…)+“



- „(…)*“



- „(…|…)“



*,+,? und | ohne Gruppierung

- Problem
  - „(abc)*“ möglich, „a*b*c*“ aber nicht
  - „(a|b)“ möglich, „a|b“ nicht
- Lösung:
  - jedes Zeichen wird als Gruppe aufgefasst
  - gesamte regex wird als Gruppe aufgefasst
  - Bsp: „a+b|c(de)*f“ --> „((a)+(b)|(c)((d)(e))*(f))“

Aufbau NFA - Implementierung

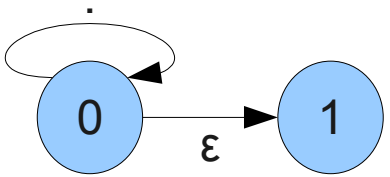
- Wort „build-nfa“:
  ```
  : build-nfa ( u-addr len – reg-addr ) ... ;
  ```
- übernimmt regex-string und baut Automaten
- während der Ausführung am Stack:
  - momentane Position und aktueller Knoten
  - Informationen über bestehende Gruppen
    - pro Gruppe ein 3-Tupel
      (Anfangskn. , Endkn. , Gr. geschlossen?)

Beispielablauf 1

- **Regex**: „`a(bc)*`“

```
Position: Beg. aktueller Knoten: 2

Gruppen: A  E  G
         1  -1 F
```

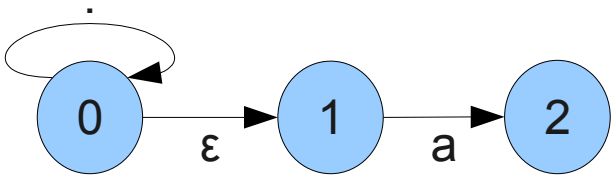


Beispielablauf 2

- Regex: „**a**`(bc)*`“

```
Position: 0  aktueller Knoten: 2

Gruppen: A  E  G
         1  -1 F
         1  2  T
```

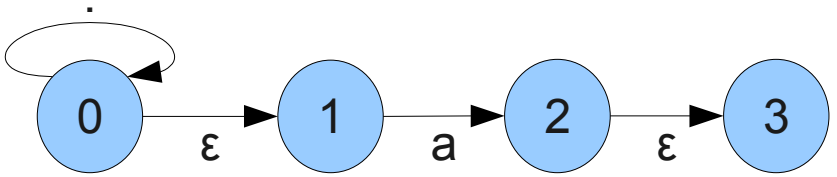


Beispielablauf 3

- Regex: „a(bc)*“

```
Position: 1  aktueller Knoten: 3

Gruppen: A  E  G
         1  -1 F
         3  -1 F
```

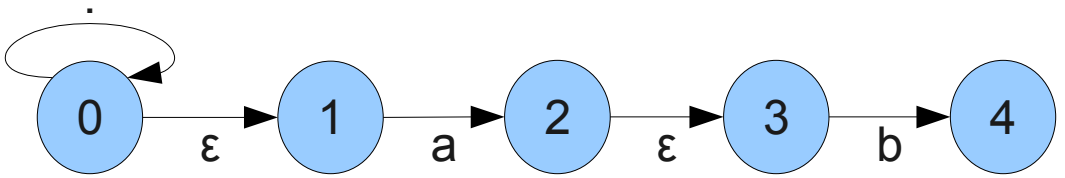


Beispielablauf 4

- Regex: „a(**b**c)*“

```
Position: 2  aktueller Knoten: 4

Gruppen: A  E  G
         1  -1 F
         3  -1 F
         3  4  T
```

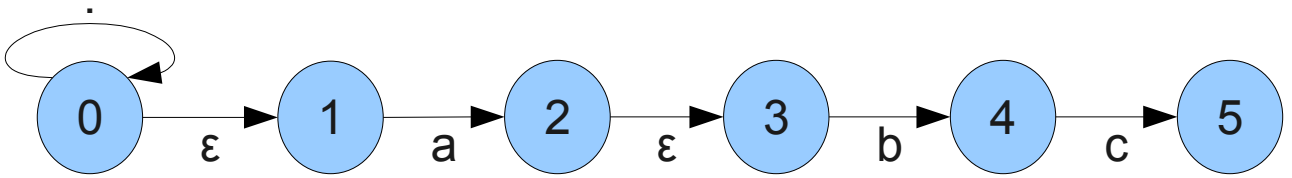


Beispielablauf 5

- Regex: „a(bc)*“

```
Position: 3  aktueller Knoten: 5

Gruppen: A  E  G
         1  -1 F
         3  -1 F
         4  5  T
```

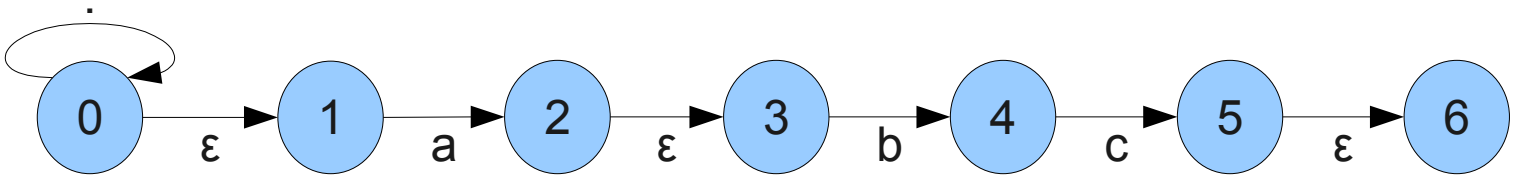


Beispielablauf 6

- Regex: „a(bc)*“

```
Position: 4  aktueller Knoten: 6

Gruppen: A  E  G
         1  -1 F
         3  6  T
```

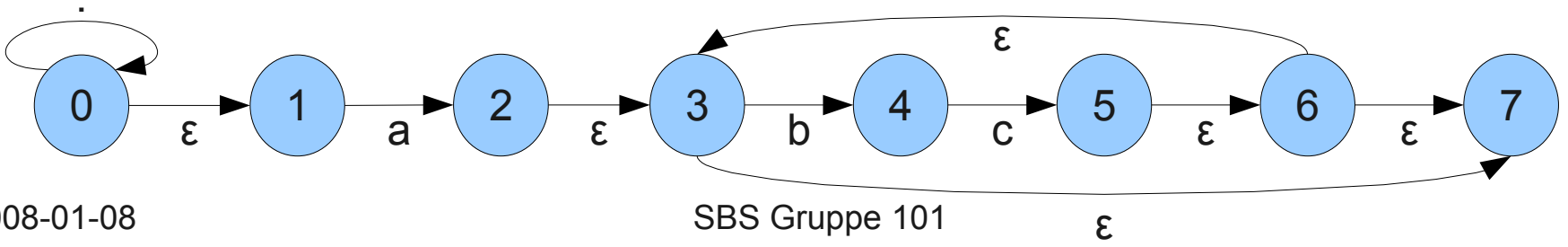


Beispielablauf 7

- **Regex**: „`a(bc)*`“

```
Position: 5  aktueller Knoten: 7

Gruppen: A  E  G
         1  -1 F
         3  6  T
```



Ausführung des NFA 1

- Wenn ein Knoten ohne Ausgangskanten erreicht wird, dann wird der String akzeptiert.

```
n GetNumOfEdges 0= if
   cleanmatchstack true EXIT
endif
```

Ausführung des NFA 2

- Wenn keine passende Ausgangskante existiert, dann wird die nächste Kante des vorhergenden Knotens betrachtet.
- Wenn dabei der Anfangsknoten erreicht wird, dann wird der String nicht akzeptiert.

```
e n GetNumOfEdges >= if
  n 0= if cleanmatchstack 0 EXIT
  else swap 1+ swap \ next edge of prev. node
endif else ...
```

Ausführung des NFA 3

- Der Weg durch den Graphen wird auf dem Stack gespeichert.
- Das Verfolgen von Epsilon-Kanten bewirkt keinen Fortschritt im Input-String.

```
: followeps { n e pos }

  n e pos \ log current edge

  n e GetGoalOfEdge 0 pos

;
```

Ausführung des NFA 4

- Der Execution Token der aktuellen Kante wird auf das aktuelle Zeichen angwandt.

```
s pos getCharn e getXtOfEdge execute if
  n e pos \ log current edge
  n e GetGoalOfEdge 0 pos 1+ \ next node
else
  n e 1+ pos \ next edge
endif
```

Vielen Dank für Ihre Aufmerksamkeit

Fragen?