

Kapitel 4: Lexikalische Analyse

Aufgabe

Zeichen bzw. Zeichengruppen werden in Symbole (Token) umgewandelt, um die Syntax-Analyse zu vereinfachen

Themen

- Reguläre Ausdrücke
- Endliche Automaten
- Longest-Input-Match
- Generatoren

Reguläre Ausdrücke

Regulärer Ausdruck für Zahlen

$[0-9]^+ \mid [0-9]^* ([0-9]"." \mid "."[0-9]) [0-9]^*$ $[0-9] = 0 \mid 1 \mid \dots \mid 9$

Reguläre Definition (Namen für Teilausdrücke)

```
digit      = [0-9]
digits     = digit+
pointgroup = digit "." | "." digit
integer    = digits
real       = digits? pointgroup digits?           digits? = digit*
number     = integer | real
```

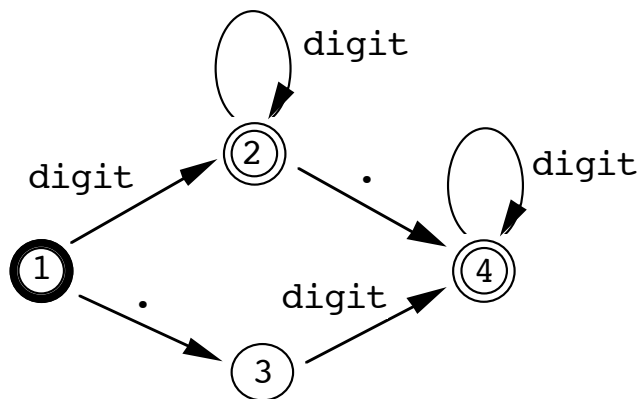
Verwendete Namen müssen vorher definiert sein → keine Rekursion

Endlicher Automat

Reguläre Definition

`number = digit+ | digit* (digit "." | "." digit) digit*`

Zustandsdiagramm



Zustandsmatrix

	digit	.
1	2	3
○ 2	2	4
3	4	-
○ 4	4	-

Endzustände werden mit Aktionen verbunden

- 2 → `inum` Token ausgeben, Zahlenwert als Attribut (oder Index)

Longest-Input-Match

Es gibt zwei Arten von lexikalischen Elementen

- a) feste Länge, z.B. + - (:=
- b) variable Länge, z.B. Zahlen, Namen

Bei Elementen mit variabler Länge: Longest-Input-Match
d.h. Erkennen der "längsten passenden Zeichenfolge"

Beispiel:

Zeichen eines Namens werden erkannt, bis ein anderes Zeichen kommt

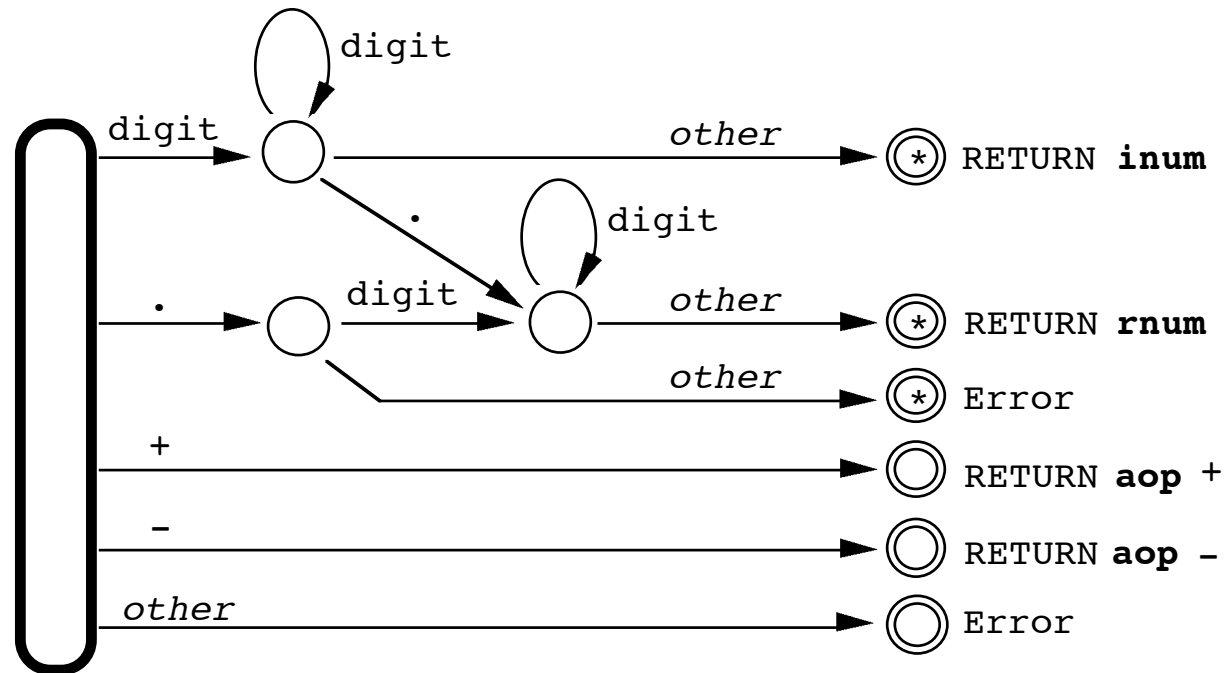
summe :=
↑

Das nicht mehr passende Zeichen hat zweifache Funktion

1. Ende des Namens
2. Anfang der Zeichenfolge :=

Analyse-Zyklus

Automat wird bei der Analyse wiederholt durchlaufen bzw. aufgerufen
z.B. Automat für Ausdrücke $17+3.14-0.32$



*) das nächste Eingabezeichen ist bereits gelesen

Generatoren

Lex, Flex

Reguläre Definition → NFA → DFA → Minimaler DFA

PCCTS, ANTLR

Lexikalische Analyse als Sonderfall der Syntax-Analyse (Reguläre Grammatik)

Programmiersprachen – Unterstützung

Prolog

Definite Clause Grammar (DCG)

(s. Anhang)

Java

`java.io.StreamTokenizer`

(s. Anhang)