Kapitel 3: Lexikalische Analyse

Aufgabe

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

Themen

- **Reguläre Ausdrücke**
- **Longest–Input–Match**
- **Generatoren**
- **Lex Beispiel**

Reguläre Ausdrücke

Regulärer Ausdruck für Zahlen

```
[0-9]+ | [0-9]* ([0-9]"."|"."[0-9]) [0-9]*        [0-9] = 0|1|...|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

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 `:=`

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)

Lex Beispiel

```
comment         \/\/.*
number          [0-9]+
register        %r([abcd]x|[sb]p|[sd]i|[89]|1[0-5])
whitespace      [\n\t ]

%%

"+="            return (PLUSASSIGNOP);
"="             return (ASSIGNOP);
"+"             return (PLUSOP);
";"             return (';');
{register}      return (REGISTER);
{number}        return (NUMBER);
{whitespace}+   ;
{comment}       ;
.               printf("Lexical error.\n"); exit(1);
```