Prof. Dr. M. Jäger
FB MNI

Compilerbau - Übungsblatt 2 „Reguläre Ausdrücke und Erkennung von Tokens “

Aufgabe 1 Reguläre Ausdrücke

Überlegen Sie sich für die nachfolgend informal definierten Tokenkategorien reguläre Ausdrücke

a) Die Zeichenkette *hallo*

b) Zeichenketten, die mit *ha* beginnen und mit *o* enden und bei denen dazwischen 2,3 oder 4 Mal ein *l* vorkommt

c) Zeichenketten, die mit *ha* beginnen und mit *o* enden und bei denen dazwischen mindestens 2 Mal ein *l* vorkommt

d) Eine vorzeichenlose hexadezimale Konstante, wobei jede der Ziffern a-f groß oder klein geschrieben werden darf

e) Eine vorzeichenlose hexadezimale Konstante, wobei die Ziffern a-f entweder alle groß oder alle klein geschrieben werden müssen

f) Eine Folge von Wörtern, in der in beliebiger Anordnung *hello* und *hallo* vorkommen, jeweils getrennt durch genau ein Leerzeichen

g) Eine Folge von Wörtern, in der abwechselnd jeweils zuerst das Wort *hallo* und dann eines der beiden Wörter *hello* oder *hola* vorkommt, jeweils getrennt durch ein oder mehrere Leerzeichen

h) Kommentare, die mit `/*` beginnen und mit `*/` enden. Im Inhalt darf `*/` nicht vorkommen.

i) (Schwierig!) Kommentare, die mit `/*` beginnen und mit `*/` enden. Im Inhalt darf `*/` nicht vorkommen, es sei denn in einem durch Apostrophe begrenztem String-Literal. Beispiel:

```
/* Dies ist ein korrekter Kommentar mit "*/" */
```

Aufgabe 2 Tests mit *egrep*

Das „Pattern Matching“-Programm *egrep* prüft zeilenorientiert, ob die Eingabedaten zu einem regulären Ausdruck passen, der als Parameter übergeben wird. Es wird üblicherweise dazu verwendet in einer oder mehreren Textdateien eine musterbasierte Volltextsuche durchzuführen:

```
egrep -n <MUSTER> <DATEI 1> <DATEI 2> ...
```

a) Probieren Sie aus, wie sich *egrep* verhält, wenn Sie es wie folgt aufrufen:

```
egrep "^[-+]?[0-9]+$"
```

und dann jeweils in einer eigenen Zeile Zahlen mit und ohne Vorzeichen eingeben.

b) Was ändert sich, wenn Sie

```
"^[-+]?[0-9]+$"
```

ersetzen durch

```
"[-+]?[0-9]+"
```

c) Denken Sie sich geeignete Testeingaben zu den in Aufgabe 2 gefragten regulären Ausdrücken aus und überprüfen Sie ihre Lösung mit *egrep* am Rechner.

Aufgabe 3 Erkennung von Tokens

Entwickeln Sie ein Filterprogramm, das aus der Standardeingabe alle Ganzzahl- und Fließkommakonstanten „herausfiltert“. Zur Eingabe:

```
In diesem Text sind weniger als 40 Zahlen enthalten, davon
1-2 Fließkommazahlen, nämlich -34,4E-12 und 2,333.
```

könnte die Ausgabe wie folgt aussehen:

```
GANZZAHL               40
GANZZAHL                1
GANZZAHL               -2
FLIESSKOMMAZAHL  -34,4E-12
FLIESSKOMMAZAHL      2,333
```

Anhang: Auszug aus *flex*-Manual

```
PATTERNS
       The patterns in the input are written  using  an  extended
       set of regular expressions.  These are:

           x          match the character 'x'
           .          any character (byte) except newline
           [xyz]      a "character class"; in this case, the pattern
                        matches either an 'x', a 'y', or a 'z'
           [abj-oZ]   a "character class" with a range in it; matches
                        or a 'Z'
           [^A-Z]     a "negated character class", i.e., any character
                        but those in the class.  In this case, any
                        character EXCEPT an uppercase letter.
           [^A-Z\n]   any character EXCEPT an uppercase letter or
                        a newline
           r*         zero or more r's, where r is any regular expression
           r+         one or more r's
           r?         zero or one r's (that is, "an optional r")
           r{2,5}     anywhere from two to five r's
           r{2,}      two or more r's
           r{4}       exactly 4 r's
           {name}     the expansion of the "name" definition
                      (see above)
           "[xyz]\"foo"
                      the literal string: [xyz]"foo
           \X         if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v',
                        then the ANSI-C interpretation of \x.
                        Otherwise, a literal 'X' (used to escape
                        operators such as '*')
           \0         a NUL character (ASCII code 0)
           \123       the character with octal value 123
           \x2a       the character with hexadecimal value 2a
           (r)        match an r; parentheses are used to override
                        precedence (see below)


           rs         the regular expression r followed by the
                        regular expression s; called "concatenation"
                        text matched by s is included when determining
                        whether this rule is the "longest match",
                        but is then returned to the input before
                        the action is executed.  So the action only
                        sees the text matched by r.  This type
                        of pattern is called trailing context".
                        (There are some combinations of r/s that flex
                        cannot match correctly; see notes in the
                        Deficiencies / Bugs section below regarding
                        "dangerous trailing context".)
           ^r         an r, but only at the beginning of a line (i.e.,
                        which just starting to scan, or right after a
```

```
              newline has been scanned).
r$            an r, but only at the end of a line (i.e., just
              before a newline).  Equivalent to "r/\n".

              Note that flex's notion of "newline" is exactly
              whatever the C compiler used to compile flex
              interprets '\n' as; in particular, on some DOS
              systems you must either filter out \r's in the
              input yourself, or explicitly use r/\r\n for "r$".
```