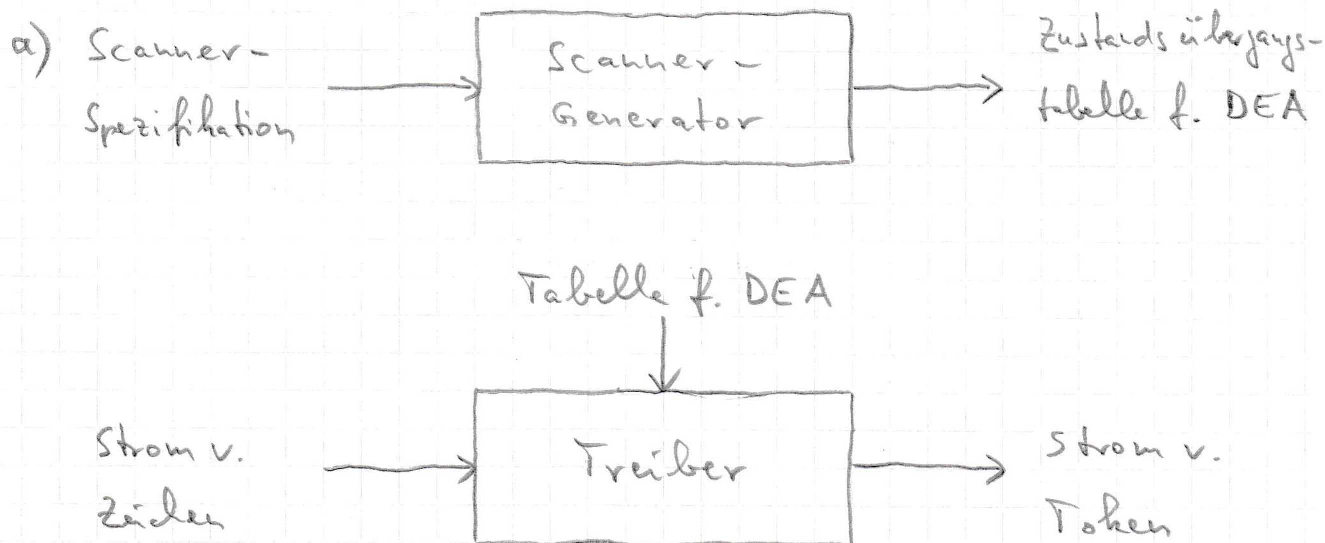
3. Scannergenerator

3.1. Funktionsweise

Es gibt zwei Alternativen:

a)



b) Der "Treiber" in a) ist ein tabellengesteuerter DEA. Die Zustandsinformation wird explizit in einer Variablen des Treibers gespeichert und für die Indizierung der Übergangstabelle benutzt. Statt dessen kann man aber auch den Zustand implizit festhalten: als Ort, an dem sich die Ausführung im Scanner gerade befindet. Statt eines von der Spezifikation unabhängigen Treibers und einer von der Spezifikation abhängigen Tabelle gibt es in dieser Variante nur ein von der Spezifikation abhängiges Programm, den Scanner. Das entspricht der Erstellung eines Scanners von Hand. Vorteil: effizientere Ausführung.

3.2. Scanner-Spezifikation

3.2.1. Elemente d. Spezifikationssprache

Zweck der Spez.-Sprache ist die Zuordnung von Aktionen (z.B. "return IDENTIFIER;") zu erkannten Tokens der zu übersetzenden Sprache. Die Tokens selbst werden durch reguläre Ausdrücke über einzelnen Zeichen des Zeichensatzes beschrieben. Also müssen folgende Elemente in der Spezifikationssprache vorhanden sein:

- Einzelne Zeichen des Zeichensatzes (auch nicht-druckbare!)
- Konkatenation
- Alternative
- Kleene'scher Abschluß
- ε
- Klammern zum Gruppieren von Teilausdrücken
- Aktionen in Form von Programmfragmenten

Sehr nützlich sind weiterhin:

- Kommentare
- Zeichenklassen (z.B. "alle Kleinbuchstaben")
- Namen als Abkürzungen für reguläre Teilausdrücke
- positiver Abschluß (spart Wiederholungen)
- optionale Teilausdrücke (als "Ersatz" für ε)
- Vorrangregeln für Operatoren (spart Klammern)

Zwingend vorhanden sein muß ein Escape-Mechanismus, um die Verwendung eines Zeichens von der Verwendung als Metazeichen (z.B. bei Klammern) zu unterscheiden.

Da der zu konstruierende Scanner i.A. mehrere Tokenklassen erkennen soll, muß es für die folgenden zwei Konfliktfälle "Diskriminierungsregeln" geben, die die Arbeitsweise des Scanners in diesen Fällen vorschreiben:

1) Ein String, der durch einen regulären Ausdruck beschrieben wird, besitzt ein Präfix (= Teilstring am "linken" Ende), das durch einen andren regulären Ausdruck in der Spezifikation beschrieben wird.
   Bsp.: "if8" ist a) ein Identifier
   oder b) ein "if" gefolgt von einer Zahl ?
   → 1. Diskriminierungsregel:
   "Der längste Match hat Vorrang!"
2) Ein String wird durch mehrere verschiedene reguläre Ausdrücke beschrieben.
   Bsp.: "if" ist a) ein Identifier
   oder b) das Schlüsselwort "if" ?
   → 2. Diskriminierungsregel:
   "Der zuerst gelistete reguläre Ausdruck hat Vorrang!"
   (Beachte: Die Reihenfolge der Regeln in der Spezifikation bekommt dadurch eine Bedeutung, die sie vorher nicht hatte.)

3.2.2. Scanner-Spezifikations-Sprache

<siehe folgende zwei Seiten>

2.4. Syntax of the Specification Language (LR Parsing)


```
specification    :  definitions SDEL rules E_O_F
                 |  definitions SDEL rules SDEL utilities E_O_F
                 ;

utilities        :  /* epsilon */
                 |  utilities PROG
                 ;

definitions      :  /* epsilon */
                 |  definitions definition
                 ;

definition       :  COLON IDENT regexpr
                 |  PROG
                 ;

rules            :  /* epsilon */
                 |  rules rule
                 ;

rule             :  regexpr PROG
                 ;

regexpr          :  regterm
                 |  regexpr BAR regterm
                 ;

regterm          :  regfactor
                 |  regterm regfactor
                 ;

regfactor        :  regprimary
                 |  regfactor STAR
                 |  regfactor PLUS
                 |  regfactor QMARK
                 ;

regprimary       :  LPAR regexpr RPAR
                 |  IDENT
                 |  CHAR
                 |  LBRA charclass RBRA
                 |  LBRA CARET charclass RBRA
                 |  DOT
                 ;

charclass        :  classcomponent
                 |  charclass classcomponent
                 ;

classcomponent   :  CHAR
                 |  CHAR DASH CHAR
                 ;
```

2.3. Lexical Analysis

Comments are enclosed in /* and */ and may span several lines. Comments do not nest. Blanks, tabs, returns, newlines, and comments are skipped during lexical analysis.

Identifiers are enclosed in { and }, must begin with a letter, and consist of letters and digits. The underscore counts as a letter.

Program fragments are enclosed in %{ and %}. They do not nest. Furthermore, they are immediately copied to the generated scanner during lexical analysis of the specification (so that there is no danger of overflowing the generator's matching buffer).

For the tokens and attributes returned by the scanner of the scanner generator see the following table:

<table>
<tr><th>Lexeme</th><th>Token</th><th>Attribute</th></tr>
<tr><td>blank</td><td>-</td><td>-</td></tr>
<tr><td>tab</td><td>-</td><td>-</td></tr>
<tr><td>return</td><td>-</td><td>-</td></tr>
<tr><td>newline</td><td>-</td><td>-</td></tr>
<tr><td>/* comment */</td><td>-</td><td>-</td></tr>
<tr><td>EOF</td><td>E_O_F</td><td>-</td></tr>
<tr><td>%%</td><td>SDEL</td><td>-</td></tr>
<tr><td>%{ program text %}</td><td>PROG</td><td>-</td></tr>
<tr><td>:</td><td>COLON</td><td>-</td></tr>
<tr><td>{ letter, followed by
letters or digits }</td><td>IDENT</td><td>pointer to name</td></tr>
<tr><td>|</td><td>BAR</td><td>-</td></tr>
<tr><td>*</td><td>STAR</td><td>-</td></tr>
<tr><td>+</td><td>PLUS</td><td>-</td></tr>
<tr><td>?</td><td>QMARK</td><td>-</td></tr>
<tr><td>(</td><td>LPAR</td><td>-</td></tr>
<tr><td>)</td><td>RPAR</td><td>-</td></tr>
<tr><td>[</td><td>LBRA</td><td>-</td></tr>
<tr><td>]</td><td>RBRA</td><td>-</td></tr>
<tr><td>-</td><td>DASH</td><td>-</td></tr>
<tr><td>^</td><td>CARET</td><td>-</td></tr>
<tr><td>.</td><td>DOT</td><td>-</td></tr>
<tr><td>\n \t \b \r \f \a</td><td>CHAR</td><td>ASCII code</td></tr>
<tr><td>\ octal digits</td><td>CHAR</td><td>ASCII code</td></tr>
<tr><td>\ other character</td><td>CHAR</td><td>ASCII code</td></tr>
<tr><td>any other character</td><td>CHAR</td><td>ASCII code</td></tr>
</table>

Bedeutung von DOT: irgendein Zeichen außer \n

3.3. Ad-hoc Scanner für den Bootstrap

Ein solcher Scanner kann sehr einfach programmiert werden.

Wie werden die beiden Diskriminierungsregeln implementiert?

1. Regel: "Der längste Match hat Vorrang!"

(Unterscheidet zwischen z.B. "if8": Identifier vs. "if" und Zahl)

Weiter Zeichen lesen, bis eines kommt, das zu keinem Token paßt!

2. Regel: "Der zuerst gelistete reguläre Ausdruck hat Vorrang!"

(Unterscheidet zwischen z.B. "if": Identifier vs. Schlüsselwort "if")

Tokenklassen für Schlüsselworte explizit zurückgeben, Tokenklasse für Identifier erst danach (implizit), wenn es kein Schlüsselwort war!

Details, die beachtet werden müssen:

- Wie wird die Funktion zum Scannen des nächsten Tokens aufgerufen? Was ist der Rückgabewert?
- Wie wird der "semantische Wert" des Tokens übermittelt? Wie wird der Typ dafür festgelegt?
- Behandlung von EOF
- Beim Programmieren wird die Erkennung <u>der</u> Tokens zusammengefaßt, die mit dem gleichen Zeichen beginnen.
- Wegen der 1. Regel oben ("längster Match") ist es an bestimmten Stellen nötig, das nächste Zeichen zu lesen. Wenn das nicht mehr zum momentanen Token gehört, muß es in die Eingabe zurückgestellt werden, damit es beim nächsten Token nicht überlesen wird.

```
/* scanner */

#define END_OF_INPUT    0
#define NUMBER          1
#define VAR             2
#define NEWLINE         3
#define PLUS            4
#define MINUS           5
#define STAR            6
#define SLASH           7
#define LPAREN          8
#define RPAREN          9
#define STORE           10

typedef union {
  double value;
  int index;
} SemanticValue;

SemanticValue tokenValue;

int nextToken(void) {
  int c;
  double value, weight;
  int s, e;

  c = getchar();
  while (c == ' ' || c == '\t') {
    c = getchar();
  }
  if (c == EOF) {
    return END_OF_INPUT;
  }
  if (c == '.' || isdigit(c)) {
    value = 0.0;
    while (isdigit(c)) {
      value *= 10.0;
      value += c - '0';
      c = getchar();
    }
    if (c == '.') {
      c = getchar();
      weight = 0.1;
      while (isdigit(c)) {
        value += (c - '0') * weight;
        weight /= 10.0;
        c = getchar();
      }
    }
    if (c == 'e') {
      c = getchar();
      if (c == '+') {
        s = 1;
        c = getchar();
      } else
      if (c == '-') {
        s = -1;
        c = getchar();
      } else {
        s = 1;
      }
      if (!isdigit(c)) {
        printf("no digit after 'e' in number\n");
        exit(99);
```

```
      }
      e = 0;
      while (isdigit(c)) {
        e *= 10;
        e += c - '0';
        c = getchar();
      }
      value *= pow(10, s * e);
    }
    ungetc(c, stdin);
    tokenValue.value = value;
    return NUMBER;
  }
  if (islower(c)) {
    tokenValue.index = c - 'a';
    return VAR;
  }
  if (c == '\n') {
    return NEWLINE;
  }
  if (c == '+') {
    return PLUS;
  }
  if (c == '-') {
    c = getchar();
    if (c == '>') {
      return STORE;
    }
    ungetc(c, stdin);
    return MINUS;
  }
  if (c == '*') {
    return STAR;
  }
  if (c == '/') {
    return SLASH;
  }
  if (c == '(') {
    return LPAREN;
  }
  if (c == ')') {
    return RPAREN;
  }
  printf("illegal character '%c' (0x%02X)\n", c, c);
  exit(99);
  /* not reached */
  return 0;
}
```