# 5. Die syntaktische Analyse

Parsergenerator yacc / bison:

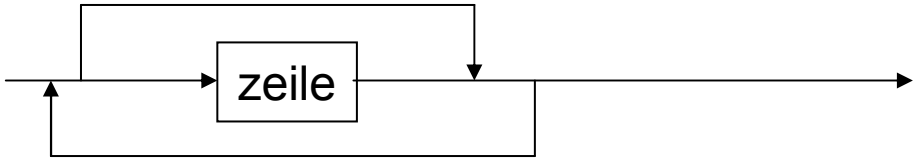
- ab 1970 von S.C.Johnson, Bell Labs, entwickelt
- yet another compiler-compiler
- benötigt Eingabestrukturbeschreibung (einer PS) ~ in BNF
- Output: C-Programm, analysiert/bearbeitet bel. Eingabe entsprechend der gegebenen Struktur
- analog zu lex:
  - eigener C-Code zur Festlegung von Aktionen, wenn Strukturelement in Eingabetext erkannt wurde
  - yacc erzeugt C-Programm *y.tab.c* als Output
  - dieses zu Parser kompileren/linken

# 5. Die syntaktische Analyse

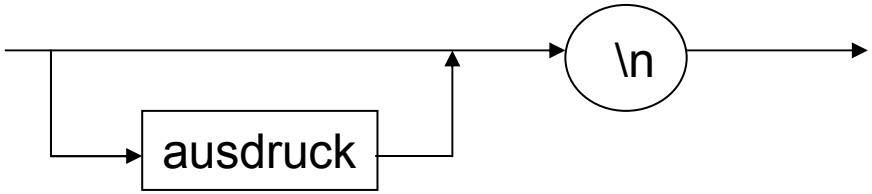
Beispiel:

gegeben: Syntaxdiagramme für Addierer

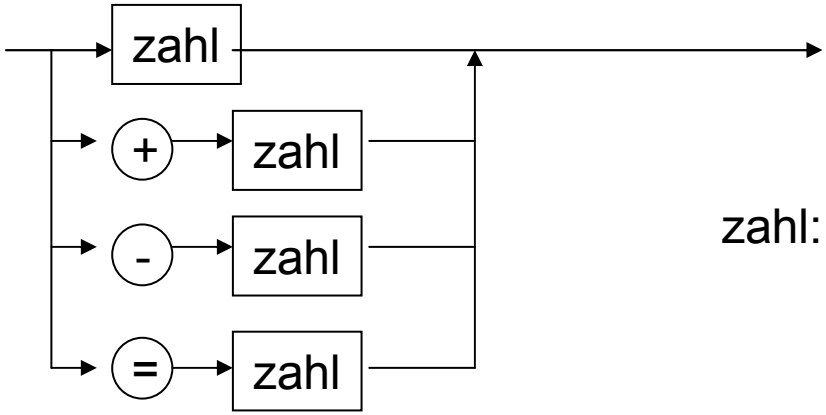
zeilen:



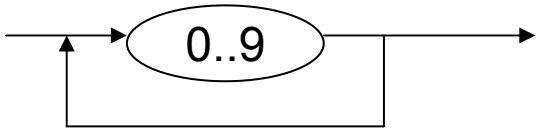

zeile:




ausdruck:




zahl:



# 5. Die syntaktische Analyse

Beispiel:  BNF zum Addierer

<zeilen> ::–  | <zeile> <zeilen>
<zeile>  ::–  \n | <ausdruck> \n
<ausdruck>  ::–  <zahl> | + <zahl> | - <zahl> | = <zahl>
<zahl>  ::– <ziffer> | <ziffer> <zahl>
<ziffer> ::– 0|1|2|3|4|5|6|7|8|9

- In Regelteil des yacc-Programms übernehmen (Syntax ähnlich)
- evtl. Teile an lex ausgliedern
- addierer.l, addierer.y

Aufgabe: Erstellen Sie Syntaxdiagramme für die Java-switch-Anweisung. Wandeln Sie diese dann in BNF um. Sie können davon ausgehen, dass ein Syntaxdiagramm für <statements> exisitert.

# 5. Die syntaktische Analyse

addierer.l

```
%%
[0-9]+        {yylval=atoi(yytext);return(ZAHL);}
[\n]          return(ZEILENENDE);
[+]           return(PLUS);
“-“           return(MINUS);
“=“           return(GLEICH);
q|quit|e|exit|ende   return(0);
.             ;
%%
```

Token ZAHL,PLUS,… werden in addierer.y definiert

yylval  Variable von yacc, um Werte von lex an yacc zu übergeben

# 5. Die syntaktische Analyse

addierer.y

```
%{
#include<stdio.h>
int akku = 0;
%}
%token    ZAHL  PLUS  MINUS  GLEICH  ZEILENENDE
%%
zeilen       :    /*leer */ | zeile zeilen;
zeile        :   ZEILENENDE
             | ausdruck ZEILENENDE {printf(„   akku:%d\n“,akku);} ;
ausdruck  :   ZAHL  {akku += $1;}
             | PLUS  ZAHL  {akku += $2;}
             | MINUS  ZAHL  {akku  -= $2;}
             | GLEICH  ZAHL  {akku  = $2;}
             ;
%%
#include“lex.yy.c“
```

# 5. Die syntaktische Analyse

yacc:

- Parsergenerator
- Aufbau yacc-Programm:

```
Definitionen
%%
yacc-Regeln
%%
benutzerdefinierte Routinen
```

- Definitionen, benutzerdef. Routinen optional
- Definitionen: global gültige C-Definitionen zwischen %{ und %},
  %token tname1 tname2 …
  %start lname
  weitere…

# 5. Die syntaktische Analyse

**yacc-Regeln:**

- = Grammatik-Regel
- Aufbau: *lname* **:** *rechte Seite* **;**
- *lname* ist nichtterminales Symbol
- rechte Seite Folge von ein/mehreren Namen oder Literalen
- Namen von Terminal-/Nichtterminalsymbolen: beginnen mit Buchstaben o. Unterstrich, gefolgt von Buchstaben/ Ziffern/Unterstrich, Groß-/Kleinschreibung wird unterschieden
- „Standard“: terminale Symbole in Großschreibung, nichtterminale Symbole in Kleinschreibung
- Literale: wie C char-Konstanten: ´a´, ´+´, ´\n´, ´\t´, …

Aufgabe: Umschreiben des Addierers, so dass er Literale für +,-, = nutzt

# 5. Die syntaktische Analyse

yacc-Regeln:

- Leere rechte Seite:
  - nichts zwischen : und ;
  - oder nur Kommentar (C-Syntax)
  - oder nur | bei Zusammenfassung von Regeln
- Zusammenfassung von Regeln (gleicher linker Seite):
  - rechte Seiten mit | trennen
- Aktionen:
  - jedes Symbol auf rechter Seite kann nachfolgend {Aktion} haben
  - Aktion = (Folge von) C-Anweisung(en)
  - Symbol i kann Wert lieferm; Zugriff auf diesen mittels $i
  - terminale Symbole (Token, Literale) liefern Wert mittels yylval aus lex (mehr dazu: spätere Folie)
  - Regel j kann Wert liefern: Aktion auf rechter Seite muss Variable $$ belegen oder Wert =$1 (Wert des 1. Symbols auf rechter Seite)

# 5. Die syntaktische Analyse

Beispiel:

```
%%
links1:  ;
links2 : /* ebenfalls leer */ ;
links3 :  /*leer */   | re1;
links4 :
        | re2
        | re3 {printf(“re3!\n“);}   /* Aktion wird wirklich nur bei re3 ausgeführt!*/
        ;
links5 : re1 {printf(“re1 mit Wert %d\n“, $1);}
        | re2  re3 {printf(“ nach re2 re3 mit Werten %d und %d\n“,$1, $2);}
        ;
links6 : re4 {$$=100;} ;
links7 : links6 ´+´ re2;  /*Rückgabewert von links7 ist Wert von links6 */
%%
```