Tabellengesteuerte Verfahren

- Es gibt tabellengesteuerte Verfahren nach den beiden Strategien, top down und bottom up.
- Die Syntaxregeln der Sprache werden in Tabellen implementiert, die den Stackautomaten dann steuern.
- Für einen minimalen Parser wird eine Automatentabelle, ein Kellerspeicher, ein Analysealgorithmus und ein einfacher Lexer benötigt.
- Tabellengesteuerte bottom up Parser sind komplizierter.

Schrittweiser Bau eines top-down Parsers für Ausdrücke

- Umformung der Grammatik in einfache BNF
- Umformung der Regeln, so dass keine Linksrekursionen enthalten sind
- Bestimmung der terminalen Anfänge der Regeln
- Ursprüngliche Grammatik für Ausdrücke:

```
<expr> ::= <expr> '+' <term> | <expr> '-' <term> | <term>
<term> ::= <term> '*' <fact> | <term> '/' <fact> | <fact>
<fact> ::= '-' num | num | '(' <expr> ')'
```

Umgeformte Regeln

```
<expr>  :: <term> <rexpr>

<rexpr> :: '+' <term> <rexpr>
         | '-' <term> <rexpr>
         | nil

<term>  :: <fact> <rterm>

<rterm> :: '*' <fact> <rterm>
         | '/' <fact> <rterm>
         | nil

<fact>  :: '-' <num>
         | <num>
         | '(' <expr> ')'
```

Bestimmung der terminalen Anfänge

```
<expr>  -> <term> -> <fact> -> ['-' | '(' | num]
<rexpr> -> ['+' | '-' | epsilon]
<term>  -> <fact> -> ['-' | '(' | num]
<rterm> -> ['*' | '/' | epsilon]
<fact>  -> ['-' | '(' | num]
```

Aufbau der Tabelle

- Die Spalten der Tabelle werden durch die Eingabezeichen gebildet.
- Zeilen werden durch die linken Seiten der Regeln, durch die nichtterminalen Symbole gebildet.
- In der Zelle steht
  - die rechte Seite der Regel zu einem nichtterminalen Symbol und dem jeweiligen terminalen Anfangssymbol.
  - Error: bei korrekter Syntax der Eingabe gelangt man hierher nicht
  - Pop: Es bleibt nichts zu tun

Der Algorithmus

- Stackinitialisierung mit push <expr>
- Auskellern des obersten Stackelements
  - Nichtterminal :
    - Ermitteln der Zelle aus dem Nichtterminal (Zeile) und dem Eingabesymbol (Spalte)
    - Error: zu dieser Zelle gelangt man im Normalfall nicht.
    - Pop: an dem Stack ändert sich nun nichts mehr.
    - Regel: Einkellern aller Symbole der rechten Seite, so dass das am weitesten links stehende Symbol oben auf dem Stack liegt.
  - Terminal:
    - Übereinstimmung mit Eingabezeichen? (accept)-> ok/Error
- Nächster Analyseschritt

Aufbau der Tabelle

Regel	First								
	nix(ε)	+	-	*	/	(	)	num	leer
expr									
rexpr									
term									
rterm									
fac									

Die fertige Automatentabelle

Regel \ First	nix	+	-	*	/	(	)	num	leer
expr	error	error	rexpr term	error	error	rexpr term	error	rexpr term	error
rexpr	error	rexpr term '+'	rexpr term '-'	error	error	error	pop	error	pop
term	error	error	rterm fac	error	error	rterm fac	error	rterm fac	error
rterm	error	pop	pop	rterm fac '*'	rterm fac '/'	error	pop	error	pop
fac	error	error	num '-'	error	error	')' expr '('	error	num	error

Beispiel 2+3*4

Stack				Operation	Current Token
<expr>					Token=num(2)

Beispiel 2+3*4

Stack				Operation	Current Token
<expr>					Token=num(2)
<term>	<rexpr>				
<fact>	<rterm>	<rexpr>			
num	<rterm>	<rexpr>			
<rterm>	<rexpr>			accepted num(2)	Token='+'
<rexpr>				pop	
+'	<term>	<rexpr>			
<term>	<rexpr>			accepted '+'	Token=num(3)
<fact>	<rterm>	<rexpr>			
<num>	<rterm>	<rexpr>			
<rterm>	<rexpr>			accepted num(3)	Token='*'
*'	<fact>	<rterm>	<rexpr>		
<fact>	<rterm>	<rexpr>		Accepted '*'	Token=num(4)
<num>	<rterm>	<rexpr>			
<rterm>	<rexpr>			accepted num(3)	empty
<rexpr>				pop	
empty				accepted all ok	

bottom up Analyse

- Ist geeignet, linksrekursive Grammatiken zu verarbeiten
- LR-Parser basieren auf bottom-up Verfahren

- Basiert auf
- zwei Tabellen
  - Aktionstabelle
  - Sprungtabelle
- zwei Kellerspeichern
  - Symbolstack
  - Zustandsstack
- zwei Operationen
  - shift
  - reduce

Die Operationen shift/reduce:

Shift:

Die Shiftoperation wird immer dann ausgeführt, wenn in dem adressierten Feld der Automatentabelle ein neuer Zustand Qx aufgeführt ist.

Es wird ein Zeichen aus dem Eingabestrom in den Symbolstack übernommen und der in der Automatentabelle angegebene Zustand in den Zustandsstack gespeichert.

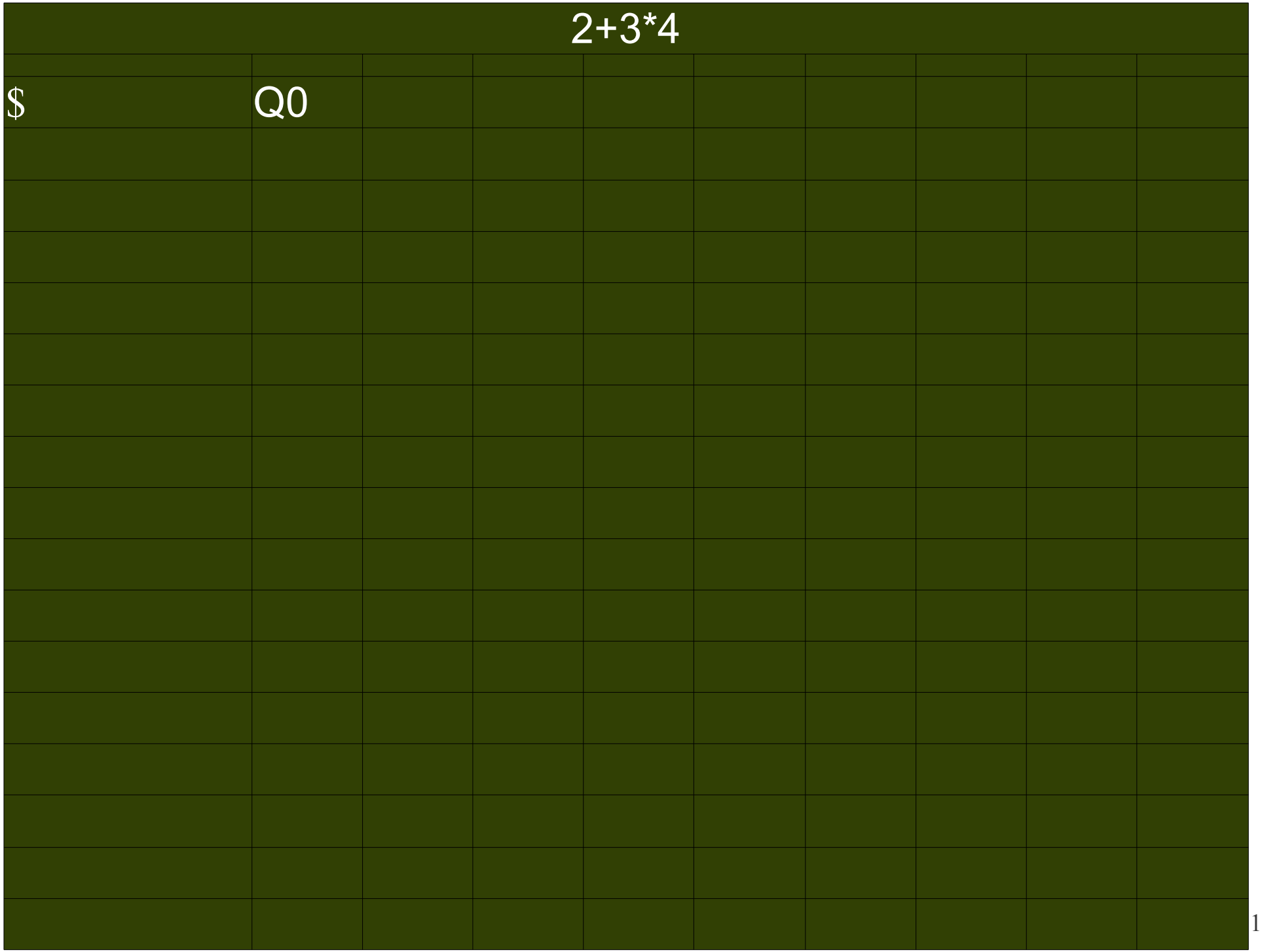
Reduce:

Die Reduceoperation wird ausgeführt, wenn in der Automatentabelle eine Regel gefunden wird. Dazu werden die Symbole auf der rechten Seite der Regel aus dem Symbolstack und ebenso viele Einträge aus dem Zustandsstack entfernt.

Dabei müssen die aus dem Stack entfernten Symbole mit denen der anzuwendenden Regel übereinstimmen. Danach wird das Symbol auf der linken Seite der Regel in den Symbolkeller geschrieben. Das jetzt oberste Symbol und der jetzt oberste Zustand bilden die Indizes für die Sprungtabelle. Der hier gefundene Zustand wird in den Zustandskeller geschrieben.

Automatentabelle bottom up

	Aktionstabelle						***Sprungtabelle***		
	Num	+	*	(	)	$	E	T	F
Q0	Q5			Q4			Q1	Q2	Q3
Q1		Q6				ACCEPT			
Q2		E::=T	Q7		E::=T	E::=T			
Q3		T::=F	T::=F		T::=F	T::=F			
Q4	Q5			Q4			Q8	Q2	Q3
Q5		F::=num	F::=num		F::=num	F::=num			
Q6	Q5			Q4				Q9	Q3
Q7	Q5			Q4					Q10
Q8		Q6			Q11				
Q9		E::=E+T	Q7		E::=E+T	E::=E+T			
Q10		T::=T*F	T::=T*F		T::=T*F	T::=T*F			
Q11		F::=(E)	F::=(E)		F::=(E)	F::=(E)			


2+3*4
$
Q0

2+3*4									
ε	Q0								
num	Q0	Q5							
F	Q0	Q3							
T	Q0	Q2							
E	Q0	Q1							
E+	Q0	Q1	Q6						
E+num	Q0	Q1	Q6	Q5					
E+F	Q0	Q1	Q6	Q3					
E+T	Q0	Q1	Q6	Q9					
E+T*	Q0	Q1	Q6	Q9	Q7				
E+T*num	Q0	Q1	Q6	Q9	Q7	Q5			
E+T*F	Q0	Q1	Q6	Q9	Q7	Q10			
E+T	Q0	Q1	Q6	Q9					
E	Q0	Q1							
Ɛ	Accept								

Implementierung

Zeichenklassenvektor für einen einfachen Lexer

```
/*           0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F */
int  vz[]={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*0x10*/   -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
/*0x20*/   -1,-1,-1,-1,-1,-1,-1,-1,bl,br,ti,pl,-1,-1,-1,-1,
/*0x30*/    n, n, n, n, n, n, n, n, n, n,-1,-1,-1,-1,-1,-1};
```

```
typedef enum Symbs
{
  Nix ,    n,   pl,   ti,   bl,  br,  // Terminale num, + * ( )
  empty,  e,    t,    f,    error     // NichtTerminale expr, term, fact
}Symbols;
```

Lexer erzeugt zusätzlich den numerischen Wert einer Zahl in einer Variablen val
Der Symbolcode steht in der Variablen col

Stackimplementa-tion durch Liste

```
// Stackimplementation mit linked list

typedef struct node
{
  struct node* nxt;
  char item;
}tnode;

// Symbolstack
tnode* pStackSymb =NULL;

// Statestack
tnode* pStackState=NULL;

// ValueStack
tnode* pStackVal=NULL;
```

```
void push(tnode** pStack,char i)
{
  tnode * ptmp=
                malloc(sizeof(tnode));
  ptmp->item=i;
  ptmp->nxt=*pStack;
  *pStack=ptmp;
}

char pop(tnode**pStack)
{
  char ret=nix;
  if(*pStack)
  {
     tnode*ptmp;
     ret=(*pStack)->item;
     ptmp=*pStack;
     *pStack=(*pStack)->nxt;
     free (ptmp);
  }
  return ret;
}
```

Aktionentabelle

```
typedef struct
{
  char Q;
  char R[5];
  void(*f)();
}tabEntry;
```

Regel	In der Tabelle
Expr :: expr '+' term	`{e,e,pl,t}`
Term :: term '*' fact	`{t,t,ti,f}`
Fact :: '(' expr ')'	`{f,bl,e,br}`

```
tabEntry ActionTable[][7]=
{
/*       leer       Num         +                   *                   (           )                   $ */
/* 0*/{ {0,{0}, 0},{5,{0},fp}, {0,{0}        , 0}, {0,{0}        , 0}, {4,{0}, 0}, {0,{0}        , 0}, {0,{0}        , 0}},
/* 1*/{ {0,{0}, 0},{0,{0}, 0}, {6,{0}        , 0}, {0,{0}        , 0}, {0,{0}, 0}, {0,{0}        , 0}, {0,{0}        ,fe}},
/* 2*/{ {0,{0}, 0},{0,{0}, 0}, {0,{e,t}      , 0}, {7,{0}        , 0}, {0,{0}, 0}, {0,{e,t}      , 0}, {0,{e,t}      , 0}},
/* 3*/{ {0,{0}, 0},{0,{0}, 0}, {0,{t,f}      , 0}, {0,{t,f}      , 0}, {0,{0}, 0}, {0,{t,f}      , 0}, {0,{t,f}      , 0}},
/* 4*/{ {0,{0}, 0},{5,{0},fp}, {0,{0}        , 0}, {0,{0}        , 0}, {4,{0}, 0}, {0,{0}        , 0}, {0,{0}        , 0}},
/* 5*/{ {0,{0}, 0},{0,{0}, 0}, {0,{f,n}      , 0}, {0,{f,n}      , 0}, {0,{0}, 0}, {0,{f,n}      , 0}, {0,{f,n}      , 0}},
/* 6*/{ {0,{0}, 0},{5,{0},fp}, {0,{0}        , 0}, {0,{0}        , 0}, {4,{0}, 0}, {0,{0}        , 0}, {0,{0}        , 0}},
/* 7*/{ {0,{0}, 0},{5,{0},fp}, {0,{0}        , 0}, {0,{0}        , 0}, {4,{0}, 0}, {0,{0}        , 0}, {0,{0}        , 0}},
/* 8*/{ {0,{0}, 0},{0,{0}, 0}, {6,{0}        , 0}, {0,{0}        , 0}, {0,{0}, 0}, {11,{0}       , 0}, {0,{0}        , 0}},
/* 9*/{ {0,{0}, 0},{0,{0}, 0}, {0,{e,e,pl,t} ,fa}, {7,{0}        , 0}, {0,{0}, 0}, {0,{e,e,pl,t} ,fa}, {0,{e,e,pl,t} ,fa}},
/*10*/{ {0,{0}, 0},{0,{0}, 0}, {0,{t,t,ti,f} ,fm}, {0,{t,t,ti,f} ,fm}, {0,{0}, 0}, {0,{t,t,ti,f} ,fm}, {0,{t,t,ti,f} ,fm}},
/*11*/{ {0,{0}, 0},{0,{0}, 0}, {0,{f,bl,e,br}, 0}, {0,{f,bl,e,br}, 0}, {0,{0}, 0}, {0,{f,bl,e,br}, 0}, {0,{f,bl,e,br}, 0}}
};
```

Sprungtabelle

```
char JumpTable[][3]=
{
/*      e  t  f*/
/* 0*/{1, 2, 3 },
/* 1*/{0, 0, 0 },
/* 2*/{0, 0, 0 },
/* 3*/{0, 0, 0 },
/* 4*/{8, 2, 3 },
/* 5*/{0, 0, 0 },
/* 6*/{0, 9, 3 },
/* 7*/{0, 0,10 },
/* 8*/{0, 0, 0 },
/* 9*/{0, 0, 0 },
/*10*/{0, 0, 0 },
/*11*/{0, 0, 0 }
};
```

Operation shift

```
void shift()
{
  push(&pStackSymb ,currTok);
  currState=ActionTable[currState][currTok].Q;
  push(&pStackState,currState);
  currTok=lex();
  printf("shift> ");
}
```

Operation reduce

```
void reduce()
{
  char left;   // left side of the rule
  char n;      // Len of rule (#items)
  char right; // current Symbol right
  char state;// current state
  char x;      // current ppoped Symbol from stack

  n=strlen(ActionTable[currState][currTok].R)-1; // len -1
  left=ActionTable[currState][currTok].R[0];
  while(n)
  {
    right=ActionTable[currState][currTok].R[n--];
    if((x=pop(&pStackSymb))!=right) {printf("Error at %d\n",pos);exit(-1);}
    pop(&pStackState);
  }
  push(&pStackSymb,left);
  //obersten Zustand lesen
  state=pop(&pStackState);
  push(&pStackState,state);
  currState=JumpTable[state][left-e];
  push(&pStackState,currState);
  printf("reduce> ");
}
```

Pop from symbolStack

Pop from stateStack

parse

```
int parse(char* p)
{
  pstr=p;
  currState=0;
  currTok=lex();
  push(&pStackState,0);
  push(&pStackSymb ,0);
  while(!(currState==1 && currTok==empty))
  {
    if (ActionTable[currState][currTok].f)ActionTable[currState][currTok].f();
    if (ActionTable[currState][currTok].Q!=0)shift();
    else                                                    reduce();
    printf("current Token: %s\n",StrSy[col]);
    printStackSymb(); puts("");
    printStackState();puts("");
    if(pause)getchar();
  }
  fe();
  printf("all ok\n");
}
```