Straight - Line Simple Language SL^2

- ambiguous => operator precedence parser
- unambiguous => SLR parser

1. LR(0) Automat

1 Konfiguration + 2 Absatz

z.B. $accept : . S $end

z.B. S : . A | . AS

A : . V = e ;

d.h. für alle potentiell im nächsten Schritt auf TOS aufsetzenden Nichtterminale werden Produktionen eingefügt. Rekursion stellt sicher, dass alle zulässigen Terminale erfasst werden, z.B. muß jedes SL^2 Programm mit einem zulässigen Nichtterminalen (der linken Seite der ersten Zuweisung) beginnen

3 Transitionen für alle potentiell als nächstes auf TOS aufsetzenden (Terminal- u. Nichtterminal-) Symbole

z.B. S, A, V

4. Zielzustände durch Shift des entsprechenden Symbols auf TOS in allen Produktionen . goto 1

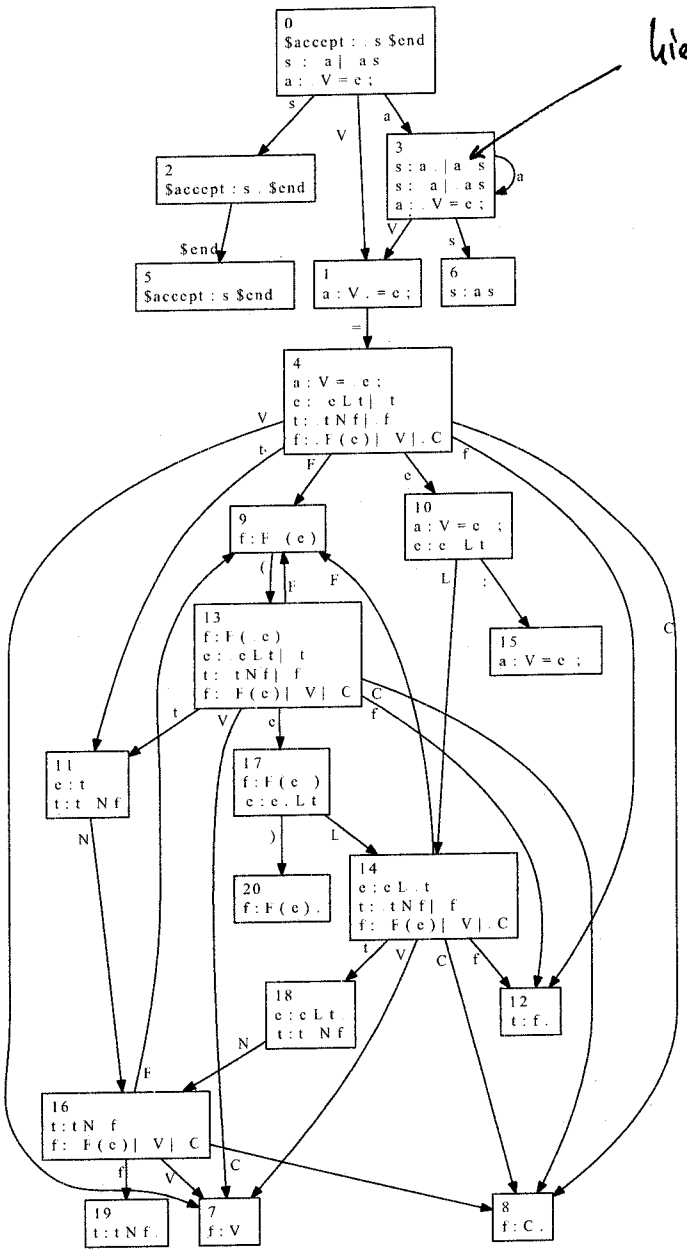


Figure 4.13. *Characteristic automaton for SL^2 programs.*

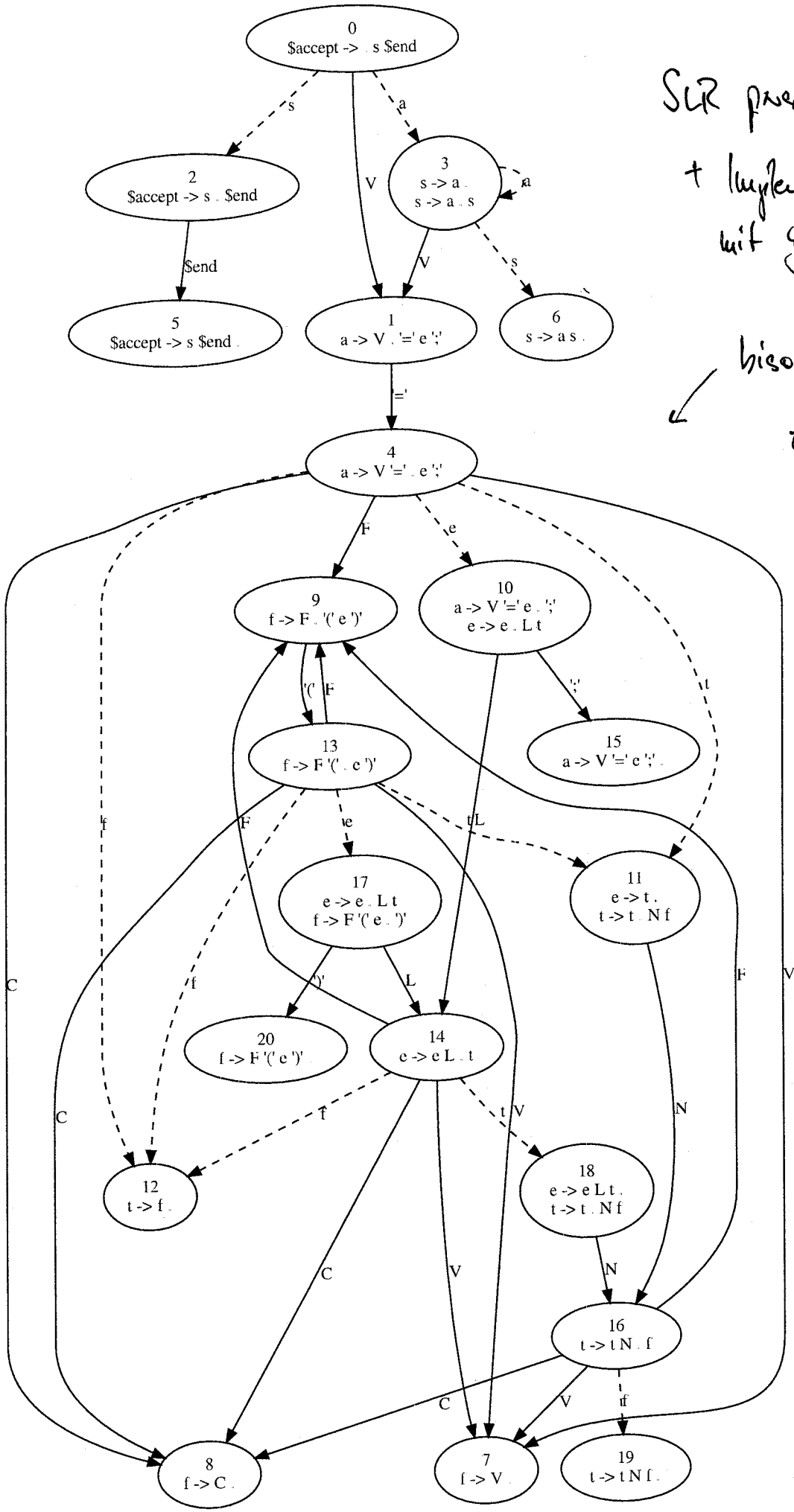
Table 4.2. *SLR Parsing of "$V = F(VNC);$" based on Definition 4.15; we show the contents of the* STACK*, the current* STATE *in the characteristic automaton, the string* PARSED *so far, the remaining* INPUT*, and the* ACTION *to be taken in each state. The parse tree can be derived by bottom-up interpretation of the reductions in the last column.*

	STACK	STATE	PARSED	INPUT	ACTION
1		0	V	$= F(VNC);$	S
2	0	1	$V =$	$F(VNC);$	S
3	0,1	4	$V = F$	$(VNC);$	S
4	0,1,4	9	$V = F($	$VNC);$	S
5	0,1,4,9	13	$V = F(V$	$NC);$	S
6	0,1,4,9,13	7		$NC);$	R($P9$)
7	0,1,4,9	13	$V = F(f$	$NC);$	S
8	0,1,4,9,13	12		$NC);$	R($P7$)
9	0,1,4,9	13	$V = F(t$	$NC);$	S
10	0,1,4,9,13	11	$V = F(tN$	$C);$	S
11	0,1,4,9,13,11	16	$V = F(tNC$	$);$	S
12	0,1,4,9,13,11,16	8		$);$	R($P10$)
13	0,1,4,9,13,11	16	$V = F(tNf$	$);$	S
14	0,1,4,9,13,11,16	19		$);$	R($P6$)
15	0,1,4,9	13	$V = F(t$	$);$	S
16	0,1,4,9,13	11		$);$	R($P5$)
17	0,1,4,9	13	$V = F(e$	$);$	S
18	0,1,4,9,13	17	$V = F(e)$	$;$	S
19	0,1,4,9,13,17	20		$;$	R($P8$)
20	0,1	4	$V = f$	$;$	S
21	0,1,4	12		$;$	R($P7$)
22	0,1	4	$V = t$	$;$	S
23	0,1,4	11		$;$	R($P5$)
24	0,1	4	$V = e$	$;$	S
25	0,1,4	10	$V = e;$		S
26	0,1,4,10	15			R($P3$)
27		0	a		S
28	0	3			R($P1$)
29		0	s		S
30	0	2	$s\$end$		S
31	0,2	5			R($P0$)
32		0	$\$accept$		ACCEPT

4.4.5 Parser for SL^2 Programs with `flex` and `bison`

We use `flex` and `bison` to implement a parser for SL^2 programs. For the sake of brevity, variables and constants are restricted to lowercase letters and single digits, respectively. The `flex` source file is shown in Listing 4.2. Whitespaces are ignored (line 5). Intrinsic functions (line 12), linear operators (line 13), and nonlinear operators (line 14) are represented by a single instance each. The named tokens (F, L, N, V, and C) to be returned to the parser are encoded as integers in the file

SLR pars



SLR parser

+ Implementierung mit g++

bison -g parser.y

Ausgabe

```
%{
#include "parser.tab.h"
int lineno=1;
%}

whitespace      [ \t]+
newline         \n
variable   [a-z]
constant   [0-9]

%%

{whitespace}     { }
{newline}  { lineno++; }
"sin"            { return F; }
"+"        { return L; }
"*"        { return N; }
{variable}       { return V; }
{constant}       { return C; }
.          { return yytext[0]; }

%%

void lexinit(FILE *source)
{
  yyin=source;
}
```

```
%{

#include<stdio.h> // for FILE
#include<iostream> // for cout

extern int lineno;
extern int yylex();
extern void lexinit(FILE*);
extern int yyerror(const char*);

%}

%token V C F L N

%%

s : a
  | a s
  ;
a : V '=' e ';'
e : e L t
  | t
  ;
t : t N f
  | f
  ;
f : F '(' e ')'
  | V
  | C
  ;

%%

int yyerror(const char *msg)
{
        std::cout << "Error: " << msg <<" in line "<< lineno
<<" of the input file."<< std::endl;
        exit(-1);
}


int main(int argc,char** argv)
{
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

prac. 4

```
parse : parser.tab.c lex.yy.c
	g++ -g $^ -lfl -lm -o $@

parser.tab.c : parser.y
	bison -g -v --defines=parser.tab.h -o $@ $<

lex.yy.c : scanner.l
	flex $<

clean :
	rm -f parser.tab.* lex.yy.* parse parser.output
parser.dot parser.pdf

.PHONY: clean
```

makefile

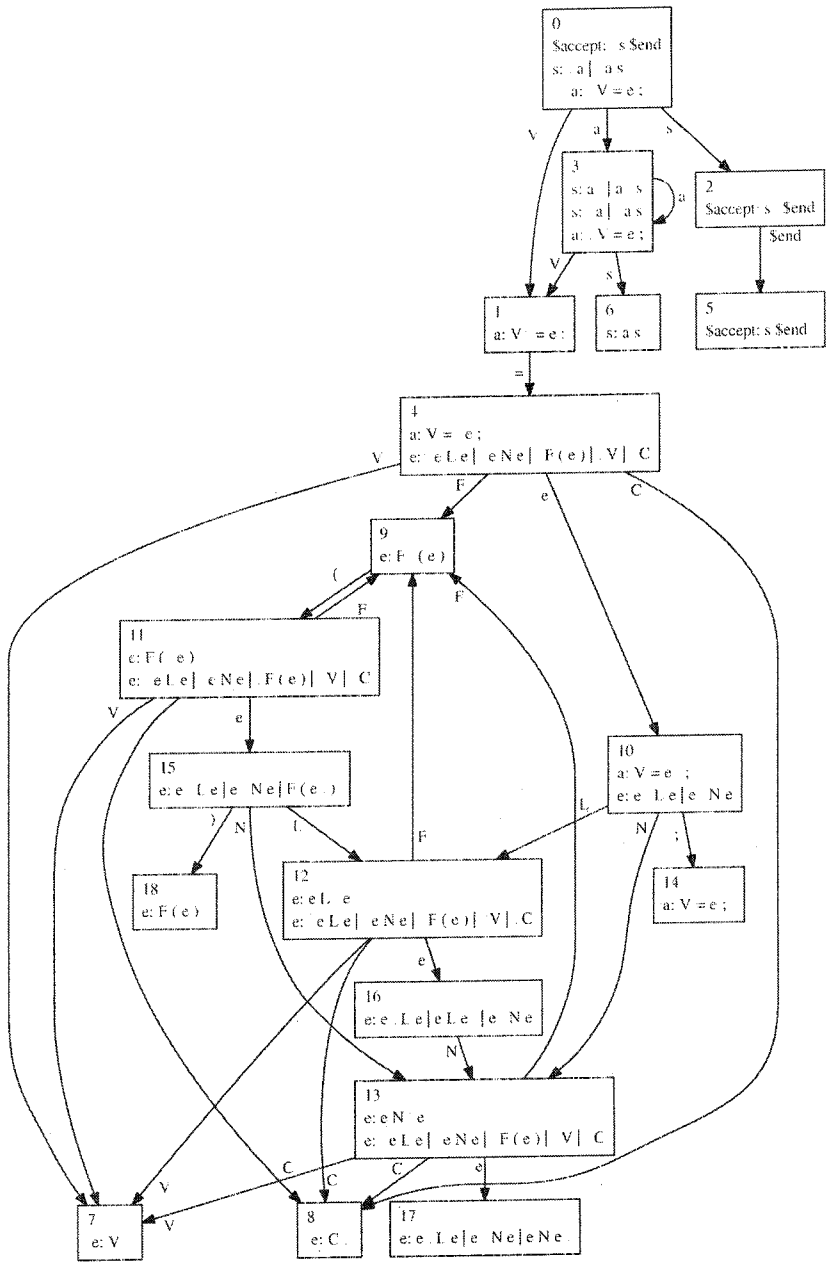


Figure 4.14. *Characteristic automaton for SL^2 programs.*

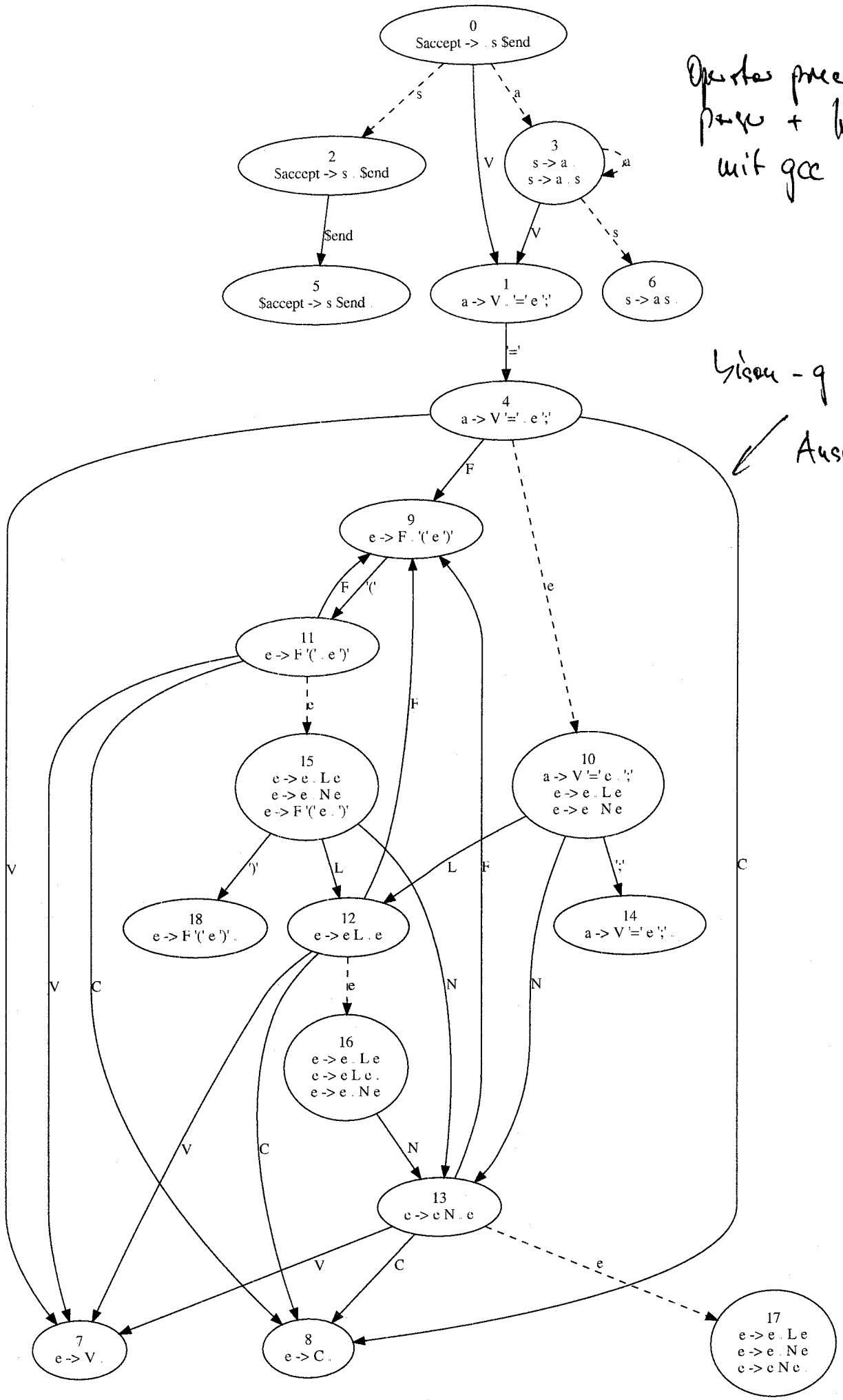
LR(0) Automat mit Operatorpräzedenz
für mehrdeutige Grammatik

Table 4.3. *Parsing "$V = F(VNC);$" based on Definition 4.16. We show the contents of the* STACK, *the current* STATE *in the characteristic automaton, the string* PARSED *so far, the remaining* INPUT, *and the* ACTION *to be taken in each state. The parse tree can be derived by bottom-up interpretation of the reductions in the last column.*

	STACK	STATE	PARSED	INPUT	ACTION
1		0	V	$= F(VNC);$	S
2	0	1	$V =$	$F(VNC);$	S
3	0,1	4	$V = F$	$(VNC);$	S
4	0,1,4	9	$V = F($	$VNC);$	S
5	0,1,4,9	11	$V = F(V$	$NC);$	S
6	0,1,4,9,11	7		$NC);$	R($P6$)
7	0,1,4,9	11	$V = F(e$	$NC);$	S
8	0,1,4,9,11	15	$V = F(eN$	$C);$	S
9	0,1,4,9,11,15	13	$V = F(eNC$	$);$	S
10	0,1,4,9,11,15,13	8		$);$	R($P7$)
11	0,1,4,9,11,15	13	$V = F(eNe$	$);$	S
12	0,1,4,9,11,15,13	17		$);$	R($P4$)
13	0,1,4,9	11	$V = F(e$	$);$	S
14	0,1,4,9,11	15	$V = F(e)$	$;$	S
15	0,1,4,9,11,15	18		$;$	R($P5$)
16	0,1	4	$V = e$	$;$	S
17	0,1,4	10	$V = e;$		S
18	0,1,4,10	14			R($P3$)
19		0	a		S
20	0	3			R($P1$)
21		0	s		S
22	0	2	$\$end$		S
23	0,2	5			R($P0$)
24		0	$\$accept$		ACCEPT

Operatorprizedenz pwse

parser.tab.h. Its inclusion into lex.yy.c prior to any other automatically generated code is triggered by the corresponding preprocessor directive at the beginning of the first section of the `flex` input file (lines 1–3). The remaining unnamed single character tokens are simply forwarded to the parser (line 17). A special lexinit routine is provided to set the pointer yyin to the given source file (line 21).



Operator precedence parser + Implementation mit gcc

Yacc - y parse.y

Ausgabe

```
%{
#include "parser.tab.h"
%}

whitespace      [ \t\n]+
variable   [a-z]
constant   [0-9]

%%

{whitespace}     { }
"sin"            { return F; }
"+"         { return L; }
"*"         { return N; }
{variable}       { return V; }
{constant}       { return C; }
.           { return yytext[0]; }

%%

void lexinit(FILE *source) { yyin=source; }
```

Scanner.l

```
%token V C F L N

%left L
%left N

%%

s : a
  | a s
  ;
a : V '=' e ';' ;
e : e L e
  | e N e
  | F '(' e ')'
  | V
  | C
  ;

%%

#include<stdio.h>

int yyerror(char *msg) { printf("ERROR: %s \n",msg); return -
1; }

int main(int argc,char** argv)
{
  FILE *source_file=fopen(argv[1],"r");
  lexinit(source_file);
  yyparse();
  fclose(source_file);
  return 0;
}
```

Page. 9

```
parse : parser.tab.c lex.yy.c
	gcc -g $^ -lfl -lm -o $@

parser.tab.c : parser.y
	bison -g -v --defines=parser.tab.h -o $@ $<

lex.yy.c : scanner.l
	flex $<

clean :
	rm -f parser.tab.* lex.yy.* parse parser.output
parser.dot parser.pdf

.PHONY: clean
```

Makefile