5.5 Aufsteigende Analyse (2)

5.5 Aufsteigende Analyse

5.5.1 Prinzip der aufsteigenden Analyse
5.5.2 Algorithmus der LR-Syntaxanalyse
5.5.3 Konstruktion der Syntaxanalysetabellen
→ 5.5.4 Konflikte
5.5.5 Präzedenzen
5.5.6 Fehlerbehandlung

Konflikte

- oft durch mehrdeutige Grammatiken
  - d.h. Fehler des Benutzers
- selten durch Nicht-LALR(1)-Grammatiken
- jetzt: Typische Beispiele mit Auflösung
  - Ziel: in der Praxis diese Muster wiedererkennen

Typische Konflikte

- mehrdeutige Grammatik
  - Konflikt in Ausdrücken
  - Konflikt in if-then-else
  - Konflikt in verschachtelten Listen
  - Konflikt in überlappenden Alternativen
- Nicht-LALR(1)-Grammatik
  - Konflikt wegen begrenzter Vorschau
  - Konflikt wegen Nicht-LALR-Grammatik

Konflikt in Ausdrücken

- ein typischer Shift/Reduce-Konflikt

Demo

Konflikt in Ausdrücken (2)

- expr-minus-konfl.l

```
%{
#include "expr-minus-konfl.tab.h"
%}
%option noyywrap
%option nodefault
%%
"-"                     { return *yytext; }
[[:alpha:]][[:alnum:]]* { return ID; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```


Konflikt in Ausdrücken (3)

- expr-minus-konfl.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token '-' ID ILLEGAL_CHAR
%%
expr:             ID
                | expr '-' expr
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Konflikt in Ausdrücken (4)

- Diskussion: siehe Buch „lex&yacc“, S. 229-230
  - herausarbeiten, wo das Problem liegt
    - expr-minus-konfl.output ansehen
      - split-screen benutzen für Grammatik / Zustand 5
    - Punkt an zwei verschiedenen Stellen derselben Regel
    - wie kann „-“ auf 2. Version folgen? 1. Version.
    - Problem ist „expr - expr - expr“
    - mehrdeutig, da Assoziativität unklar
- Übung: Konflikt auflösen
  - erstmal ohne Präzedenzen
  - expr-minus-ok.y

Konflikt in if-then-else

- ein typischer Shift/Reduce-Konflikt durch optionalen Teil

Demo

Konflikt in if-then-else (2)

- if-then-else.l

```
%{
#include "if-then-else.tab.h"
%}
%option noyywrap
%option nodefault
%%
"if"                    { return IF; }
"then"                  { return THEN; }
"else"                  { return ELSE; }
[[:alpha:]][[:alnum:]]* { return EXPR; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```


Konflikt in if-then-else (3)

- if-then-else.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token IF THEN ELSE EXPR ILLEGAL_CHAR
%%
statement:        IF EXPR THEN statement
                | IF EXPR THEN statement ELSE statement
                | other
                ;
other:            EXPR
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Konflikt in if-then-else (4)

- Diskussion: siehe Buch „lex&yacc“, S. 231, 233-234
  - herausarbeiten, wo das Problem liegt
    - if-then-else.output ansehen
    - Wie kann auf *statement* ein ELSE folgen?
      Als Teil von Regel 2:
      IF EXPR THEN (IF EXPR THEN EXPR) ELSE EXPR
    - also bekanntes Dangling-Else-Problem
- Übung: Konflikt auflösen
  - erstmal ohne Präzedenzen
  - if-then-else-ok.y

Konflikt in verschachtelten Listen

- ein typischer Shift/Reduce-Konflikt

Demo

Konflikt in verschachtelten Listen (2)

- nested-lists.l

```
%{
#include "nested-lists.tab.h"
%}
%option noyywrap
%option nodefault
%%
";"                     { return *yytext; }
[[:alpha:]][[:alnum:]]* { return ID; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```


Konflikt in verschachtelten Listen (3)

- nested-lists.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token ';' ID ILLEGAL_CHAR
%%
list:             outerList ';'
                ;
outerList:        /* empty */
                | outerList outerListItem
                ;
outerListItem:    innerList
                ;
innerList:        /* empty */
                | innerList innerListItem
                ;
innerListItem:    ID
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Konflikt in verschachtelten Listen (4)

- Diskussion: siehe Buch „lex&yacc“, S. 232-233
  - herausarbeiten, wo das Problem liegt
    - nested-lists.output ansehen
    - Zustand 2: wie kann *outerListItem* mit „;“ anfangen? indem *innerList* leer ist
    - also gibt es zwei Wege, wie auf das erste *outerList* ein „;“ folgen kann: mehrdeutig!
    - entweder gar kein *outerListItem* oder ein leeres
- Übung: Konflikt auflösen
  - nested-lists-ok.y

Konflikt in überlappenden Alternativen

- ein typischer Reduce/Reduce-Konflikt

Demo

Konflikt in überlappenden Alternativen (2)

- jungen-maedchen.l

```
%{
#include "jungen-maedchen.tab.h"
%}
%option noyywrap
%option nodefault
%%
"Jan"                   { return JAN; }
"Ulrich"                { return ULRICH; }
"Kirsten"               { return KIRSTEN; }
"Birgit"                { return BIRGIT; }
"Eike"                  { return EIKE; }
"Kai"                   { return KAI; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```


Konflikt in überlappenden Alternativen (3)

- jungen-maedchen.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token JAN ULRICH KIRSTEN BIRGIT EIKE KAI ILLEGAL_CHAR
%%
kinder:           maedchen
                | jungen
                ;
maedchen:         KIRSTEN
                | BIRGIT
                | EIKE
                | KAI
                ;
jungen:           JAN
                | ULRICH
                | EIKE
                | KAI
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Konflikt in überlappenden Alternativen (4)

- Diskussion: siehe Buch „lex&yacc“, S. 238-240
  - herausarbeiten, wo das Problem liegt
    - jungen-maedchen.output ansehen
    - Zustand 5: EIKE kann *maedchen* oder *jungen* sein: mehrdeutig!
    - Zustand 6: für KAI ebenso
- Übung: Konflikt auflösen
  - jungen-maedchen-ok.y

Konflikt wegen begrenzter Vorschau

- Grammatik ist nicht LALR(1), sondern LALR(2)
  - daher Shift/Reduce-Konflikt bei yacc
  - solche Grammatik nicht zu empfehlen
    - auch schwer für Menschen lesbar

Demo

Konflikt wegen begrenzter Vorschau (2)

- opt-keyword.l

```
%{
#include "opt-keyword.tab.h"
%}
%option noyywrap
%option nodefault
%%
"("                     { return '('; }
"command"               { return COMMAND; }
"keyword"               { return KEYWORD; }
[[:alpha:]][[:alnum:]]* { return ID; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```

- Beispiel für „command“: „print“
  Beispiel für „keyword“: „color“

Konflikt wegen begrenzter Vorschau (3)

- opt-keyword.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token COMMAND ID KEYWORD ILLEGAL_CHAR
%%
rule:             COMMAND optional_keyword '(' ID ')'
                  ;
optional_keyword: /* empty */
                  | '(' KEYWORD ')'
                  ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Konflikt wegen begrenzter Vorschau (4)

- Diskussion: siehe Buch „lex&yacc“, S. 237-238
  - herausarbeiten, wo das Problem liegt
    - opt-keyword.output ansehen
    - wie kann *optional_keyword* mit „(“ anfangen?
      - mit Regel 3
      - indem es leer ist (Regel 2)
    - *nicht* mehrdeutig: Lookahead von 2 löst Problem!
      - mit yacc nicht möglich
  - Lösung: Regel flachklopfen (s.u.)
    - dann stellt sich Frage nicht, ob Reduzieren oder nicht
    - Nachteil: wenn viele Regeln beteiligt, dann Ergebnis groß und unübersichtlich

Konflikt wegen begrenzter Vorschau (5)

- opt-keyword-ok.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token COMMAND ID KEYWORD ILLEGAL_CHAR
%%
rule:             COMMAND '(' KEYWORD ')' '(' ID ')'
                | COMMAND '(' ID ')'
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Konflikt wegen Nicht-LALR-Grammatik

- Grammatik ist nicht LALR(1), aber LR(1)
  - daher Reduce/Reduce-Konflikt bei yacc
    - zwei Zustände werden fälschlich identifiziert, weil die Kerne gleich sind
  - solche Grammatik ebenfalls nicht zu empfehlen

Demo

Konflikt wegen Nicht-LALR-Grammatik (2)

- param-return.l

```
%{
#include "param-return.tab.h"
%}
%option noyywrap
%option nodefault
%%
','                     |
':'                     { return *yytext; }
[[:alpha:]][[:alnum:]]* { return ID; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```


Konflikt wegen Nicht-LALR-Grammatik (3)

- param-return.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token ID ILLEGAL_CHAR
%%
def:              param_spec return_spec ','
                ;
param_spec:       type
                | name_list ':' type
                ;
return_spec:      type
                | name ':' type
                ;
type:             ID
                ;
name:             ID
                ;
name_list:        name
                | name ',' name_list
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```

Konflikt wegen Nicht-LALR-Grammatik (4)

- Diskussion: siehe bison-Manual, S. 69-71 („Mysterious reduce/reduce conflicts“)
  - herausarbeiten, wo das Problem liegt
    - param-return.output ansehen
    - Wie kann auf *type* oder *name* ein „,“ folgen?
      - *type*: am Ende von *def* (Regeln 1, 4/5)
      - *name*: Regel 9: in *name_list*
    - das sollten verschiedene Zustände sein!
    - Wo können *type* und *name* beide zusammen vorkommen?
      - in *return_spec* und *param_spec* (via *name_list*)
      - nach Lesen von ID haben zugehörige Zustände gleichen Kern:
        - 6 *type* → ID .
        - 7 *name* → ID .
  - Lösung: gleiche Kerne künstlich verschieden machen durch Regel, die nie passen kann (s.u.)

Konflikt wegen Nicht-LALR-Grammatik (5)

- param-return-ok.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token ID ILLEGAL_CHAR BOGUS
/* Das Token BOGUS wird niemals benutzt. */
%%
def:              param_spec return_spec ','
                  ;
param_spec:       type
                  | name_list ':' type
                  ;
return_spec:      type
                  | name ':' type
                  | ID BOGUS
                  ;
type:             ID
                  ;
name:             ID
                  ;
name_list:        name
                  | name ',' name_list
                  ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


5.5 Aufsteigende Analyse

5.5.1 Prinzip der aufsteigenden Analyse
5.5.2 Algorithmus der LR-Syntaxanalyse
5.5.3 Konstruktion der Syntaxanalysetabellen
5.5.4 Konflikte
→ 5.5.5 Präzedenzen
5.5.6 Fehlerbehandlung

Präzedenzen: Idee

- manchmal umständlich:
  Grammatik eindeutig bzw. LALR(1) machen
  - Beispiel: Ausdrücke
- Alternative:
  Konflikt bleibt in Grammatik,
  aber Parser nimmt immer „richtige“ Wahl
  - explizit definieren,
    welche Wahl „richtig“ ist:
    Präzedenz

Zwei Arten von Präzedenzen

- Assoziativität
- Priorität

Assoziativität in Ausdrücken

- Minus in Ausdrücken: links-assoziativ
  - 7 - 5 -1 = (7 - 5) - 1
    nicht: 7 - (5 - 1)

Demo

Assoziativität in Ausdrücken (2)

- `%left` statt `%token`
- expr-minus-assoc.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%left '-' ID ILLEGAL_CHAR
%%
expr:             ID
                | expr '-' expr
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Rechtsassoziativität in Ausdrücken

- Zuweisungen in C
  - `v1 = v2 = v3 = 0;`
  - interpretiert als:
    `v1 = (v2 = (v3 = 0));`

Demo

Rechtsassoziativität in Ausdrücken (2)

- `%right` statt `%token`
- expr-assign-assoc.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%right '=' ID ILLEGAL_CHAR
%%
expr:             ID
                | expr '=' expr
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Prioritäten in Ausdrücken

- „Punktrechnung vor Strichrechnung“
  - `7 + 5 * 2`
  - interpretiert als:
    `7 + (5 * 2)`
- eindeutige Grammatik möglich, aber umständlich

Demo

Prioritäten in Ausdrücken (2)

- expr-unambig.y
- expr-prio.l

```
%{
#include "expr-prio.tab.h"
%}
%option noyywrap
%option nodefault
%%
"+"                     |
"-"                     |
"*"                     |
"/"                     |
"<"                     |
">"                     |
"("                     |
")"                     { return *yytext; }
"=="                    { return EQ; }
"<="                    { return LE; }
">="                    { return GE; }
"!="                    { return NE; }
[[:alpha:]][[:alnum:]]* { return ID; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```


Prioritäten in Ausdrücken (3)

- expr-prio.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token '(' ')' ID ILLEGAL_CHAR
%nonassoc '<' '>' EQ LE GE NE
%left '+' '-'
%left '*' '/'
%%
e:        e '*' e
        | e '/' e
        | e '+' e
        | e '-' e
        | e '<' e
        | e '>' e
        | e EQ  e
        | e LE  e
        | e GE  e
        | e NE  e
        | '(' e ')'
        | ID
        ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```

Prioritäten in Ausdrücken (4)

- Operatoren auf gleicher Zeile `%left`, `%right`: gleiche Priorität
- spätere Zeilen: höhere Priorität
- ohne Prioritätsrelation: `%token`
- nichtassoziative Operatoren: `%nonassoc`
  - verboten: `2 < 3 < 4`

Präzedenzstufen

- Terminal
  - durch Reihenfolge der Zeilen mit `%left`, `%right`, `%nonassoc`
- Regel
  - gleich Präzedenzstufe des letzten Terminals in Regel

Algorithmus für Präzedenzen

- Auflösung Shift/Reduce-Konflikt:
  - vergleiche Präzedenzstufe der Regel mit Präzedenzstufe des Vorschausymbols
    - Vorschausymbol höher: Schieben
    - Regel höher: Reduzieren
    - gleich: falls Assoziativität dieser Stufe
      - links: Reduzieren
      - rechts: Schieben
      - nichtassoziativ: Fehler
    - ohne Stufe: Schieben + Warnung

Kontextabhängige Präzedenz

- falls Token in verschiedenen Regeln verschiedene Präzedenz
  - Beispiel: Minus als Subtraktion / als Vorzeichen

Demo

Kontextabhängige Präzedenz (2)

- explizites präzedenzgebendes Symbol für Regel
  - Symbol muß sonst nicht verwendet werden
- expr-prio-uminus.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token '(' ')' ID ILLEGAL_CHAR
%nonassoc '<' '>' EQ LE GE NE
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
e:          e '*' e
          | e '/' e
          | e '+' e
          | e '-' e
          | e '<' e
          | e '>' e
          | e EQ  e
          | e LE  e
          | e GE  e
          | e NE  e
          | '(' e ')'
          | '-' e     %prec UMINUS
          | ID
          ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Übung: Konflikt in if-then-else mit Präzedenz auflösen

- ein Shift/Reduce-Konflikt
  - überlegt Euch eine mögliche Lösung

Demo

Konflikt in if-then-else mit Präzedenz auflösen: Lösungen

- if-then-else.y
- if-then-else-prec1.y
- if-then-else-prec2.y

5.5 Aufsteigende Analyse

5.5.1 Prinzip der aufsteigenden Analyse
5.5.2 Algorithmus der LR-Syntaxanalyse
5.5.3 Konstruktion der Syntaxanalysetabellen
5.5.4 Konflikte
5.5.5 Präzedenzen
→ 5.5.6 Fehlerbehandlung

Behandlung von Syntaxfehlern

- schlecht:
  - Abbruch bei erstem Fehler
- besser:
  - fehlerhafte Stelle überspringen
  - Analyse wieder aufsetzen
  - nach weiteren Fehlern suchen
- Annahme:
  - es gibt Stellen, wo die Analyse wieder aufsetzen kann
    - Beispiel: Semikolon (Ende einer Anweisung)
      - nur Heuristik

Fehlerbehandlung mit yacc

- spezielle Fehler-Regeln
  - spezifizieren Stelle zum Wiederaufsetzen
  - benutzen irgendwo Terminalsymbol `error`
- Terminalsymbol `error`
  - eingebaut, reserviert
  - bei Fehler vom Parser als Eingabesymbol generiert
  - muß normal in einer Regel passen
    - Probleme:
      - noch zu viel Angefangenes auf Stack
      - noch etliche Symbole vor Aufsetzstelle in Eingabe

Algorithmus zur Fehlerbehandlung bei yacc

- melde Fehler
- lege Symbol `error` vorne in Eingabe
- baue Stack ab, bis Symbol `error` paßt
- Schiebe `error`
- solange Vorschausymbol nicht paßt, wirf es weg
- normale Bearbeitung
  - aber: melde keine Fehler, bis
    drei aufeinanderfolgende Symbole gepaßt haben

Demo

Algorithmus zur Fehlerbehandlung bei yacc (2)

- err-handling.l

```
%{
#include "err-handling.tab.h"
%}
%option noyywrap
%option nodefault
%%
"="                     |
"+"                     |
"*"                     |
"("                     |
")"                     |
";"                     { return *yytext; }
[[:alpha:]][[:alnum:]]* { return ID; }
[ \t]+                  /* ueberspringe White-Space */
\n                      { return 0; /* Nur eine Zeile lesen. */ }
.                       { return ILLEGAL_CHAR; }
```

Algorithmus zur Fehlerbehandlung bei yacc (3)

- err-handling.y

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
void yyerror(char *);
%}
%verbose
%token '(' ')' ID ';' ILLEGAL_CHAR
%right '='
%left '+'
%left '*'
%%
prog:             /* empty */
                | prog stmt
                ;
stmt:             expr ';'
                | error ';'
                ;
expr:             ID
                | expr '=' expr
                | expr '+' expr
                | expr '*' expr
                | '(' expr ')'
                | '(' error ')'
                ;
%%
void yyerror(char *msg) {
    printf("\n%s\n", msg);
}
int main() {
    return yyparse();
}
```


Algorithmus zur Fehlerbehandlung bei yacc (4)

- Vorsicht:
  - Fehler-Regeln sollten ein Synchronisationszeichen nach `error` haben
    - sonst Folgefehler möglich
    - deswegen haben moderne Sprachen Semikolon nach Anweisung
  - semantische Operationen können durch das Abräumen „außer Tritt“ geraten

Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex
5. Syntaxanalyse und der Parser-Generator yacc
6. → Syntaxgesteuerte Übersetzung
7. Übersetzungssteuerung mit make