

## Der Parser-Generator Accent

### 1 Vorgeschichte

Kurz vor 1960 kamen (mit der Sprache Algol60) die kontextfreien Grammatiken in die Informatik. Mitte der 1975-er Jahre wurden zahlreiche Parser-Generatoren entwickelt. Das sind Programme, die als Eingabe eine (kontextfreie) Grammatik erwarten und daraus einen (z.B. in C oder einer anderen Programmiersprache formulierten) Parser erzeugen. Aus heute schwer nachvollziehbaren Gründen (großer Stolz auf die entwickelten Werkzeuge?) war es üblich, diese Parser-Generatoren als "Compiler Compiler" zu bezeichnen, obwohl sie keine Compiler compilieren, sondern nur ("abstrakte") Grammatiken in ("konkrete") Parserprogramme übersetzen.

Der heute mit Abstand bekannteste und am weitesten verbreitete solche Parser-Generator ist der `yacc` (yet another compiler compiler, auf Deutsch etwa "noch so ein Compiler Compiler", eine scherzhafte Anspielung auf seine zahlreichen Vorgänger und Konkurrenten). Auch heute noch steht einem der `yacc` auf praktisch jedem Unix-System zur Verfügung, und eine Weiterentwicklung namens `bison` gibt es unter Unix und unter Windows. Das ähnlich wie `yacc` klingende englische Wort `yak` bezeichnet ein tibetisches Hochlandrind (*Bos grunniens*). Vermutlich deshalb wurde die Weiterentwicklung des `yacc` nach einem anderen Rindvieh (*Bison bison*) benannt.

In den 1975-Jahren waren verbreitete Computer um einen ziemlich grossen Faktor langsamer als PCs aus den 2005-er Jahren (etwa um den Faktor eine Million). Deshalb wurde der `yacc` 1975 auf sogenannte LR-Grammatiken eingeschränkt. Dabei ist LR eine ziemlich komplizierte und (für viele Menschen) nicht ganz leicht zu erkennende oder nachzuprüfende Eigenschaft. Wenn eine Grammatik diese Eigenschaft hat, kann man daraus einen besonders schnellen (tabellengesteuerten, LR-) Parser erzeugen, und genau das tut der `yacc`. Man sagt auch: Der `yacc` erzeugt aus einer LR-Grammatik einen LR-Parser. Das Programm `yacc` hat keine Probleme festzustellen, ob eine Grammatik LR ist oder nicht.

Wendet man den `yacc` auf eine Grammatik an, die nicht LR ist, bekommt man Fehlermeldungen wie "Shift reduce conflict ..." oder "Reduce reduce conflict ...". Auch geübte Compilerbauer fürchten diese Fehlermeldungen, weil es häufig unklar und schwierig ist zu erkennen, ob und wie man sie (durch einen "Umbau der Grammatik in eine LR-Grammatik") beseitigen kann.

**Anmerkung:** Bei der Definition der Sprache HTML unterlief den Entwicklern ein historisch bedeutsamer Fehler. Um das Schreiben von HTML-Seiten zu erleichtern, führten sie eine harmlos klingende Regel ein die es erlaubt, bestimmte geöffnete Klammern ("Tags niedriger Stufe") implizit ("ohne Schreibarbeit") dadurch zu schliessen, dass man eine andere Klammer ("einen Tag höherer Stufe") explizit schloss. Wegen dieser Regel ist die Grammatik für HTML nicht LR und man kann mit `yacc` keinen Parser daraus erzeugen. Statt von Hand Parser zu schreiben, die genau dem HTML-Standard entsprechen, begannen wichtige Browser-Hersteller einen Wettbewerb, wer den "am meisten akzeptierenden" (und damit am wenigsten standardkonformen) Parser und Browser schreiben kann. Einer verbreiteten Ansicht entsprechend hat Microsoft mit dem InternetExplorer diesen Wettbewerb gewonnen, aber dadurch den (auch vorher schon "kränkelnden") HTML-Standard geschwächt. XHTML ist eine Variante von HTML ohne die fragwürdige Klammer-Spar-Regel. Die Grammatik von XHTML ist LR.

**Accent** (a compiler compiler for the entire class of contextfree grammars) ist ein Parser-Generator, der für jede kontextfreie Grammatik einen Parser erzeugen kann (nicht nur für LR-Grammatiken oder andere Teilmengen). Die konkrete Syntax einer Accent-Spezifikation stimmt weitgehend mit der einer `yacc`-Spezifikation überein. Auch der Accent wird (um `yacc`-Fans nicht zum Lernen ungewohnter Bezeichnungen zu zwingen) als Compiler Compiler bezeichnet. Es folgt ein Beispiel.

## 2 Ein Parser für die Sprache der geraden Binärzahlen

Eine Grammatik für die Sprache der geraden Binärzahlen sieht in (yacc-) Accent-Notation etwa so aus:

```

1  start :
2    gerade { printf("OK\n"); }
3  ;
4
5  irgend :
6    '0'
7    | '1'
8    | '0' irgend
9    | '1' irgend
10 ;
11
12 gerade :
13   '0'
14   | irgend '0'
15 ;

```

In dieser Grammatik sind `start`, `irgend` und `gerade` Zwischensymbole. `'0'` und `'1'` sind Endsymbole. Das zuerst ("zuoberst") definierte Zwischensymbol ist automatisch das Startsymbol (hier: `start`).

Alle Regeln mit dem selben Zwischensymbol auf der linken Seite müssen nacheinander notiert werden. Die linke Seite (das Zwischensymbol) schreibt man nur einmal hin, trennt die rechten Seiten der Regeln durch senkrechte Striche `|` und schließt das Ganze mit einem Semikolon `;` ab. Im Beispiel gibt es nur eine Regel mit dem Zwischensymbol `start` auf der linken Seite (Zeile 1 bis 3). Für das Zwischensymbol `irgend` gibt es vier Regeln (Zeile 5 bis 10) und für `gerade` zwei Regeln (Zeile 12 bis 15).

Nach jeder Regel darf man eine C-Blockanweisung `{ . . . }` notieren. Sie wird ausgeführt, wenn die entsprechende Regel zum Ableiten der Eingabe des Parsers benutzt wird. Das Beispiel enthält nur eine einzige solche Blockanweisung (in Zeile 2, am Ende der einzigen Regel für `start`).

Ein Parser liest seine Eingabe in aller Regel mit Hilfe eines sogenannten Lexers (oder: Scanners) ein. Für (yacc und) Accent kann man einen solchen Lexer mit dem Lexer-Generator `lex` (oder mit der `lex`-Varianten `flex`) erzeugen. Für die obige Accent-Spezifikation kann eine passende `flex`-Spezifikation etwa so aussehen:

```

16 %{
17 #include "yygrammar.h"
18 %}
19 %%
20 "0"      { incColNr(); return '0'; }
21 "1"      { incColNr(); return '1'; }
22 " "      { incColNr(); /* skip blank */ }
23 "--" .* \n { incLineNr(); /* skip comment and adjust lineNumber */ }
24 \n       { incLineNr(); /* skip newline and adjust lineNumber */ }
25 .       { incColNr(); yyerror("illegal character"); }

```

Diese `lex`- (`flex`-) Spezifikation legt fest:

Die Eingabe sollte im Wesentlichen nur aus den Zeichenketten `"0"` und `"1"` bestehen (Zeile 20, 21). Ausserdem sind Blanks `" "` erlaubt (Zeile 22), Kommentare (Zeile 23) und `newline`-Zeichen `\n` (Zeile 24). Blanks, Kommentare und `newline`-Zeichen werden vom Lexer einfach überlesen (in den C-Blockanweisung in den Zeilen 22, 23 bzw. 24 steht kein `return`-Befehl, der die Kontrolle vom Lexer zurück an den Parser geben würde). Kommentare müssen mit `"--"` beginnen und reichen bis zum nächsten `newline`-Zeichen `\n`. Alle anderen Zeichen werden vom Lexer als unerlaubt betrachtet (Zeile 25, der Punkt `.` bezeichnet ein beliebiges Zeichen).

Immer wenn der Parser ein "weiteres Stück Eingabe" benötigt, ruft er den Lexer auf. Wenn der Lexer in der Eingabe das Lexem "0" (bzw. "1") erkennt, liefert er dem Parser das Token '0' (bzw. '1'), mit der `return`-Anweisungen in Zeile 20 (bzw. 21). Blanks und Kommentare überliest der Lexer, ohne dass der Parser etwas davon erfährt.

Ausser einer Accent-Spezifikation (siehe oben Zeile 1 bis 15) und einer dazu passenden `lex`-Spezifikation (Zeile 16 bis 25) braucht man zu Erzeugung eines lauffähigen Parsers noch ein paar weitere C-Funktionen, etwa die folgenden:

```

26 #include <stdio.h>
27
28 extern FILE * yyin;
29 int main(int args, char** argv) {
30     // If the user has specified the input for the parser on the
31     // command line, this input is written into a file tmp.tmp
32     // and read from there. Otherwise the parser will read from
33     // the standard input (which may be redirected to a file):
34     if (args == 2) {
35         yyin = fopen("tmp.tmp", "w");
36         fputs(argv[1], yyin);
37         fclose(yyin);
38         printf("%s, input: %15s : ", argv[0], argv[1]);
39         yyin = fopen("tmp.tmp", "r");
40     }
41
42     yyparse();
43     return 0;
44 } // main
45
46 yyerror(char * msg) {
47     extern long yypos;
48
49     printf("line %d: %s\n", yypos, msg);
50     exit(1);
51 } // yyerror
52
53 yywrap() { return 1;}
```

Der Lexer erwartet in der globalen Variablen `yyin` Hinweise auf die Datei, aus der er "das Quellprogramm" einlesen soll. Ist `yyin` gleich `null` liest er von der Standardeingabe.

Wenn man mit obiger `main`-Funktion einen Parser namens `pars01` erzeugt, kann man ihn wie folgt auf vier verschiedene Weisen aufrufen:

```

54 > pars01 1010
55 > pars01 "1 01 0"
56 > pars01
57 > pars01 < input
```

Beim Aufruf in Zeile 54 wird `1010` eingelesen und geparkt. In Zeile 55 sind die doppelten Anführungsstriche nötig, weil die Eingabe Trennzeichen (Blanks oder Tabzeichen) enthält. In Zeile 56 liest der Parser von der Standardeingabe (Tastatur), der Benutzer muss seine Eingabe mit `Strg-D` bzw. `Strg-Z` abschliessen. In Zeile 57 liest der Parser aus der angegebenen Datei `input`.

Eine DOS-Stapelverarbeitungsdatei (eine `.bat`-Datei) zum Erzeugen eines Parsers kann etwa so aussehen:

```

58 rem build.bat fuer das Verzeichnis Gerade2erZahlen
59 set ACCENT=..\accent\accent.exe
60 set ENTIRE=..\entire\entire.c
61 set LEX=flex
62 set CC=gcc
```

```

63
64 %ACCENT% spec.acc
65 %LEX% spec.lex
66 %CC% -o gerade2erZahlen.exe yygrammar.c lex.yy.c errmsg.c main2.c %ENTIRE%
67 gerade2erZahlen 10010

```

In Zeile 64 werden mit dem Compiler `accent` aus der Accent-Spezifikation `spec.acc` (die z.B. die Zeilen 1 bis 15 enthält) zwei C-Quelldateien namens `yygrammar.h` und `yygrammar.c` erzeugt. Man beachte, dass die Kopfdatei (engl. the header file) `yygrammar.h` in der `lex`-Spezifikation inkludiert werden muss (siehe oben Zeile 17).

In Zeile 65 wird mit dem Lexer-Generator `flex` aus der `lex`-Spezifikation `spec.lex` (die z.B. die Zeilen 16 bis 25 enthält) eine C-Quelldatei namens `lex.yy.c` erzeugt.

In Zeile 66 übersetzt der C-Compiler `gcc` die Dateien `yygrammar.c`, `lex.yy.c`, `errmsg.c`, `main2.c` und `entire.c` in eine ausführbare Datei namens `gerade2erZahlen.exe`.

In Zeile 67 wird der Parser `gerade2erZahlen` aufgerufen und parst die Eingabe `10010`. Wenn alles gut geht sollte er als Ergebnis `OK` ausgeben (siehe den `printf`-Befehl oben in Zeile 2).

Eine `.bat`- oder `bash`-Datei zum Testen des Parser kann etwa so aussehen:

```

68 echo -----
69 echo "Korrekte Eingaben fuer gerade2erZahlen:"
70 echo -----
71 gerade2erZahlen 0
72 gerade2erZahlen 00
73 gerade2erZahlen 10
74 ...
75 echo -----
76 echo "Fehlerhafte Eingaben fuer gerade2erZahlen:"
77 echo -----
78 gerade2erZahlen 1
79 gerade2erZahlen "0 1"
80 gerade2erZahlen 11
81 ...
82 gerade2erZahlen 0011102010101

```

Die Ausgabe des Testskripts sollte etwa so aussehen:

```

83 -----
84 "Korrekte Eingaben fuer gerade2erZahlen:"
85 -----
86 gerade2erZahlen, input: 0 : OK
87 gerade2erZahlen, input: 00 : OK
88 gerade2erZahlen, input: 10 : OK
89 ...
90 -----
91 "Fehlerhafte Eingaben fuer gerade2erZahlen:"
92 -----
93 gerade2erZahlen, input: 1 : line 1, col 1: syntax error
94 gerade2erZahlen, input: 0 1 : line 1, col 3: syntax error
95 gerade2erZahlen, input: 11 : line 1, col 2: syntax error
96 ...
97 gerade2erZahlen, input: 0011102010101 : line 1, col 7: illegal character '2'

```

**Gentle-97** arbeitet mit dem LR-Parser-Generator `yacc` (bzw. `bison`). **Gentle 21** arbeitet wahlweise mit dem `yacc` (bzw. `bison`) oder mit dem universellen Parser-Generator `Accent` zusammen.