yacc - eine Einführung

Axel Kohnert

18th May 2005

Abstract

`yacc` ist ein Unixprogramm, welches die Erstellung einer Programmiersprache unterstützt. Der Benutzer gibt die definierende Grammatik an, und das Programm erstellt ein C-Programm, welches die spezifizierte Programmiersprache mittels einer Parsertafel implementiert. `yacc` wurde so entworfen, daß es gut mit `lex` zusammenarbeitet.

1 Syntax eines yacc-programms

1.1 Aufruf von yacc

Ein yacc-Programm, steht üblicherweise in einem file mit der Endung `.y`, dieser file, z.B. bsp.y, wird mit dem Kommando
`yacc bsp.y`
in den file `y.tab.c` kompiliert, und diesen kann man mit dem Kommando
`cc y.taby.c -ly`
in den ausführbaren file `a.out` übersetzen. Mit dem Aufruf
`make bsp`
wird automatische aus dem file `bsp.y` ein ausführbarer file `bsp` erzeugt. Dies funktioniert nicht auf allen UNIX systemen, manchmal weiß make nicht wie das geht, man muß ihm dann nachhelfen.

1.2 Syntax

Ein yacc Programm ist aus drei Teilen aufgebaut, es hat die grobe Form:

```
deklartionen
%%
regeln
%%
programme
```

Die genaue Form der drei Teile wird noch besprochen. Die Syntax ist also ähnlich zu einem lex-Programm. Auch bei einem yacc-programm, sind der der erste und dritte Teil optional. Im Unterschied zu lex mußjedoch mindestens eine Regel angegeben werden. Innerhalb von `/*` und `*/` kann in allen drei Teilen ein Kommentar eingefügt werden.

1.3 Regeln

Regeln haben die Form:

```
A : BODY;
```

Dabei ist `A` ein Nichtterminalsymbol, und `BODY` besteht aus null oder mehr Namen und Literalen. Der Doppelpunkt und der Strichpunkt sind yacc-Syntax. Ein Literal ist ein einzelnes Zeichen innerhalb von zwei '. Man kann dabei auch die übliche C-Syntax verwenden, z.B. '\n' ist ein newline. Man kann dabei mehrere Regeln mit gleicher linker Seite zusammenfassen.

```
A : B;
A : E F;
A : G H I;
```

ist identisch zu

```
A : B |  E F |  G H I;
```

Namen von Tokens, müssen mit

```
%token name1 name2 ...
```

im Deklarationsteil vereinbart werden. Das Startsymbol der Grammatik wird mit

```
%start symbol
```

ebenfalls im Deklarationsteil vereinbart, falls keines vereinbart wird, ist die linke Seite der ersten Regel das Startsymbol.

2 Die lexikalische Analyse in yacc

`yacc` benötigt eine Routine `yylex()` welche das nächste Zeichen, bzw. das nächste token liefert, diese Routine muß man selber schreiben, oder wie wir später sehen werden, von `lex` schreiben lassen, zunächst ein Beispiel wie man es selber macht.

```
%{
#include <stdio.h>
%}
%%
w:  'a' w 'a' | 'b' | w '\n'
{ printf("\nok\n"); };
%%
yylex()
{
return getchar();
}
```

Dieses Programm erkennt Worte der Form $a^n b a^n$. Dabei wurde schon eine sogenannte action verwandt.

2.1 Aktionen

Zu jeder Regel kann man vor dem Strichpunkt Aktionen in C-Syntax angeben. In diesen Aktionen darf jedoch keine return-Anweisung zum Beenden verwendet werden, sondern man muß statt dessen die break-Anweisung verwenden. return würde die gesamte Syntax Analyse beenden. Aktionen stehen innerhalb der geschweiften Klammern wie in obigen Beispiel. Der Programmteil wird ausgeführt wenn die Regel angewandt wurde. Die Action bezieht sich immer nur auf den Teil der Regel der direkt dabei steht, nicht auf mehrere Teile der rechten Seite, die durch oder veknüpft sind. Dazu nochmal ein Beispiel:

```
%{
#include <stdio.h>
%}
%%
w:   wa '\n'
{ printf("\nw\n"); };
wa: 'b' { printf("\nwa ohne a\n"); } | 'a' wa 'a'
{ printf("\nwa in a\n"); };
%%
yylex()
{
return getchar();
}
```

Dabei kann auch dem Ergebnis der Regel, d.h. der linken Seite ein Wert zugewiesen werden. Dazu gibt es die Varibalen `$$` und `$i`. Mit der Regel

```
{ $$ = 1; }
```

Erhält der Ausdruck den Wert 1. Um auf den Wert der bisher ausgewerteten Regeln zuzugreifen hat man die Variablen `$1, $2, ..` sie entsprechen den einzelnen Worten auf der rechten Seite

```
A: B C D
```

dann steht in `$2` der Wert von `B`. So wäre bei der Implementation eines Taschenrechners folgendes sinnvoll:

```
expr: '( expr ')'
{ $$ = $2; }
      | expr '+' expr
{ $$ = $1 + $3; }
....
```

Die Werte von Regeln sind default mäßig vom Typ int. Wie man dies ändert später.

2.2 lex und yacc

Zuerst noch ein Beispiel wie man mit Tokens und einem eigenen `yylex()` arbeitet. Man muß im Deklarationsteil die Token benennen und schreibt dann im `yylex()` return token.

```
%{
#include <stdio.h>
%}
%token A B
%%
w:  A w A | B | w '\n';
%%
yylex()
{
char c;
c = getchar();
if (c == 'a')
return A;
if (c == 'A')
return A;
if (c == 'b')
return B;
if (c == 'B')
return B;
return c;
}
```

Will man nun die lexikalische Analyse von einem lex-programm vornehmen lassen ist folgendes zu tun. Bei den Aktionen des lex-programms, ist das Ergebnis mittels return (TOKEN) zurückzugeben. Dabei sind als Tokennamen die im yacc Programm definierten Namen zu verwenden. Das von lex erstellte Programm `lex.yy.c` ist im Programmteil mit der Anweisung

```
#include "lex.yy.c"
```

einzubinden. Ein Beispiel; Mit dem lexprogramm:

```
%%
a  {return (A);}
A  {return (A);}
b  {return (B);}
B  {return (B);}
'\n' { return yytext[0];}
.  {return yytext[0];}
```

und dem yacc programm:

```
%{
#include <stdio.h>
%}
%token A B
%%
w:  A w A | B | w '\n';
%%
#include "lex.yy.c"
```

kann man obiges Programm ersetzen. Beim Kompiliern muß man jedoch die lex option für den linker mit angeben:

```
cc y.tab.c -ly -ll
```

da sonst die lex-routine `yywrap` nicht gefunden wird.

2.3 nicht int-werte

Will man Zuweisungen machen, wobei der Typ verschieden von int ist, so muß man wie folgt vorgehen: Der Wert eines yacc-Ergebnisses wird als union umdefiniert, sodaß man verschiedene Typen behandeln kann. Dazu fügt man im Deklarationsteil folgende Zeilen ein:

```
typedef union {
int intarg;
....
} YYSTYPE;
```

und innerhalb dieser union kann man alle gewünschten typen definieren. Auch die token definitionen muß man ändern wenn man im Regelteil auf den Wert einzelner token zugreifen will. Dazu spezifiziert man den den Typ einzelner token.

```
%token <intarg> DIGIT;
%token <char> CHAR;
```

wir schauen uns dazu ein weiteres Beispiel an. Auch diesmal zusammen mit lex. Mit dem lex-programm:

```
%{
#include <math.h>
%}
%%
[1-9][0-9]*\.[0-9]+ {
yylval.doublearg = atof(yytext);
return DA;
}

\n return NEWLINE;

.{ return UNKNOWN; }
```

und dem yacc-programm:

```
%{
#include <stdio.h>
typedef union
{
double doublearg;
} YYSTYPE;
%}
%token <doublearg> DA
%token UNKNOWN
%token NEWLINE
%type <doublearg> iw
%%
w:  iw NEWLINE
{ printf ("Ergebnis %g \n", $1);} ;

iw:  { $$ = 0.0;
}
| iw u DA u
{
$$ = $1 + $3;
}
;
u : | u UNKNOWN ;
%%
#include "lex.yy.c"
```

erhält man die Summe der double-zahlen in der Eingabezeile.