6 Syntaktische Analyse mit yacc

6.1 Einführung

6.1.1 Kontextfreie Grammatiken und Backus-Naur-Form

Beispiel: Der Kadonische Leuchtturm. Die Einwohner der Insel Kadonien lieben die Abwechslung, und sie streichen daher ihren Leuchtturm alle zwei Wochen neu an. Dies ist natürlich sehr verwirrend für die Seefahrer. Aber wir können den Seefahrern helfen: Jedes Mal geht eine zufällig sich ergebende Gruppe von kadonischen Frauen und Männern los, um den Turn neu zu streichen. Jeder Helfer streicht dabei drei Meter zusammenhängende Turmhöhe, und zwar in drei Ringen. Ein Mann streicht stets rot-weiß-rot, eine Frau stets weiß-rot-weiß. Der Turm ist zwölf Meter hoch. Damit folgen die Kadonier immer diesem Schema:

```
KADONISCHERTURM ::= HELFER HELFER HELFER HELFER
HELFER ::= MANN
         | FRAU
MANN ::= ROT WEISS ROT
FRAU ::= WEISS ROT WEISS
ROT ::= 'rot'
WEISS ::= 'weiss'
```

Die Leuchttürme auf den Nachbarinseln sind zwar auch immer in ein Meter breiten Ringen bemalt, und sie sind zum Teil auch zwölf Meter hoch. Aber die Nachbarn mögen die kadonische Leuchtturmmode nicht und haben streng darauf geachtet, niemals das Kadonische Schema zu verwenden. Wir schreiben daher ein Programm, das die Insel Kadonien sicher anhand seines Leuchtturms erkennt. Das Programm soll eine Zeile vom benutzenden Kapitän lesen und dann ausgeben, ob er vor Kadonien liegt oder nicht.

Das obige ist eine **Grammatik** in **Backus-Naur-Form (BNF)**.

Terminale Symbole, die wortwörtlich in der Eingabe vorkommen müssen, stehen in Anführungszeichen.

Der **senkrechte Strich** ist wie immer die Alternative.

Außer den verwendeten Möglichkeiten kann man auch Symbole oder Symbolfolgen in **eckige Klammern** einschließen, wenn sie optional sind. Teile in **geschweiften Klammern** dürfen beliebig oft wiederholt werden, einschließlich nullmal.

Die **Erkennung der Literale** „rot“ und „weiss“ überlassen wir einem **Lexer**.

Die Erkennung des **Turm-Musters** wäre **mit Lex-Startzuständen** sehr **mühsam**.

Das Programm **Yacc** dagegen **kann aus** einer BNF-artigen **Grammatik** einen **Syntax-**

checker generieren.

Das Yacc-Programm zur obigen Grammatik:

`leuchtturm.y`:

```
%{
#include <stdio.h>
%}
%token ROT WEISS SCHREIBFEHLER
%%
kadonischerturm:  helfer helfer helfer helfer
                ;
helfer:           mann
                | frau
                ;
mann:             ROT WEISS ROT
                ;
frau:             WEISS ROT WEISS
                ;
%%
int yyerror(char *msg) {
    return 0;
}
int main() {
    printf("Wie sieht der Leuchtturm aus, Sir? ");
    if(yyparse() == 0)
        printf("Wir liegen vor der Insel Kadonien, Sir!\n");
    else
        printf("Wir liegen *nicht* vor der Insel Kadonien, Sir!\n");
    return 0;
}
```

Die **Grundstruktur** eines Yacc-Programms ist ähnlich **wie** die eines **Lex-Programms**:

- **Deklarationen**
- **Regeln**
- **Unterprogramme** (optional)

Im **Regel-Teil** finden wir die **BNF-Grammatik** wieder, nur mit etwas **anderer Syntax**.

Die **Startregel** ist die **erste** Regel.

Das **Ende der Eingabe** muß immer am Ende der Abarbeitung der Startregel kommen.

Die `%token`**-Anweisung** sagt, daß der **Lexer** die **Token ROT und WEISS liefern** kann. Außerdem kann er auch das Token **SCHREIBFEHLER** liefern, falls ein anderes

Wort eingegeben wird.

Man kann wie bei **Lex** am **Anfang literalen C-Code** angeben, hier eine Include-Anweisung.

Am Ende können **Unterprogramme** angegeben werden. Hier müssen wir unser `main()` hinschreiben. Die Funktion `main()` ruft die **Funktion** `yyparse()` auf. `yyparse()` **gibt 0 zurück**, wenn alles **in Ordnung** war, und einen Wert **ungleich 0**, falls die **Eingabe nicht zur Grammatik paßt**.

Außerdem müssen wir eine **Fehlermeldefunktion** `yyerror()` schreiben. Diese **hier tut gar nichts**, weil wir die Auswertung später in `main()` machen.

Ein **Lex-Scanner** kann uns die Eingabe **in Token zerlegen** und die Worte „rot“ und „weiss“ erkennen:
`leuchtturm.l`:

```
%{
#include "leuchtturm.tab.h"
%}
%option noyywrap
%option nodefault
%%
"rot"           { return(ROT); }
"weiss"         { return(WEISS); }
\n              { return 0; /* Nur eine Zeile lesen. */ }
[ \t]           /* Ignoriere sonstigen Whitespace. */
.               { return(SCHREIBFEHLER); }
```

Am **ersten Zeilenende** gibt der Scanner 0 zurück als Zeichen, daß wir am Ende sind.

Die **Option** `noyywrap` gibt an, daß es keine weitere Eingabedateien gibt.

Die **Option** `nodefault` unterdrückt die Default-Regel für Zeichen, die nirgends passen. Stattdessen haben wir eine eigene Default-Regel.

Die **Rückgabewert-Konstanten** sind in der Datei `leuchtturm.tab.h` definiert. Sie wird **von Bison** aus der `%token`**-Zeile generiert**.

Bison ist die **Gnu-Version** von Yacc.

Eine solche Kombination von Bison-Parser und Flex-Scanner läßt sich mit dem folgenden **allgemeinen Makefile** übersetzen:
`Makefile`:

```
%.c: %.y # loesche alte implizite Regel
%.c: %.l # loesche alte implizite Regel
%.tab.c %.tab.h: %.y
        bison --defines $<
%.c: %.l %.tab.h
        flex -t $< > $@
%: %.tab.o %.o
        cc -o $@ $^
```

Zuerst müssen wir **zwei eingebaute implizite Regeln löschen**, weil sie im Weg sind.

Dann sagen wir, wie ein **Bison-Aufruf** aus einer ∗.y-Datei die beiden Dateien ∗.tab.c mit dem Parser und ∗.tab.h mit den Token-Definitionen generiert.

Die **Flex-Regel** ist fast die Default-Regel, nur kommt die Abhängigkeit von der Include-Datei mit den Token-Definitionen dazu.

Schließlich muß das **ausführbare Programm** aus dem Parser und dem Scanner zusammengebunden werden.

▶▶ `make leuchtturm`

▶▶ `./leuchtturm`

6.2 Online-Aufgabe: Klammerausdrücke

Die folgende Grammatik beschreibt korrekte Klammerausdrücke:

```
ka ::=
     | ka ka_einfach
ka_einfach ::= buchstabe
             | '(' ka ')'
             | '[' ka ']'
             | '{' ka '}'
             | '<' ka '>'
```

Schreibe einen Parser, der korrekte Klammerausdrücke von der Standard-Eingabe liest und erkennt.

Beachte, daß die erste Alternative von `ka` die ganz **leere Eingabe** ist.

Interessant ist hier die **starke Verwendung von Rekursion**, die **typisch** für fast alle Grammatiken ist.

Die **genaue Definition von** `buchstabe` überlassen wir dem Scanner.

Der folgende **Flex-Lexer** kann das Scannen übernehmen:
`ka.l`:

```
%{
#include "ka.tab.h"
%}
%option noyywrap
%option nodefault
%%
"("             |
")"             |
"["             |
"]"             |
"{"             |
"}"             |
"<"             |
">"             { return *yytext; }
.|\n            { return BUCHSTABE; }
```

Beachte, daß **alle** 255 **ASCII-Zeichen** größer als 0 **bereits implizit als Token** für Yacc **definiert** sind. Daher kann Flex die **Klammern direkt als Buchstabe zurückgeben**.

6.3 Semantik zur Syntax

Bei vielen Problemen wollen wir **nicht nur** prüfen, **ob** eine **Eingabe** zur Grammatik **paßt**.

Der **Lexer** liefert nicht nur zu jedem Token den **Typ** per Return-Wert mit, sondern er kann in der globalen Variablen `yylval` **auch** zusätzlich einen **Wert** liefern.

Entsprechend kann der **Parser zu** jedem **nicht-terminalen Symbol** auf der **linken** Seite einer Regel einen **neuen Wert** aus den Werten der Symbole auf der **rechten** Seite liefern.

Hierfür kann man in die Regeln **Aktionen** einfügen, die diesen Wert berechnen.

Weiterhin darf man **auch beliebige andere Anweisungen** in diesen Aktionen ausführen lassen.

Damit wird es möglich, einen **Parser für eine Programmiersprache** zu schreiben, der als Ergebnis einen **Syntaxbaum** des Eingabeprogramms liefert, der dann **an** einen **Code-Generator übergeben** werden kann.

Oder man kann einen **Taschenrechner** schreiben, der **Punktrechnung-vor-Strichrechnung** beherrscht und nebenbei auch die **Ausdrücke berechnet**:

Zunächst brauchen wir einen **Lexer**, für den wir unseren alten einfachen Taschenrechner aus Abschnitt 5.3 abwandeln:
`calc.l`:

```
%{
#include <math.h>
#include "calc.tab.h"
%}
%option noyywrap
DIGIT   [0-9]
%%
{DIGIT}+("."{DIGIT}*)?  {
                  yylval.zahl = atof(yytext);
                  return NUMBER;
                }
"="             |
"("             |
")"             |
"+"             |
"-"             |
"*"             |
"/"             { return yytext[0]; }
[[:space:]]     { /* ignoriere Whitespace */ }
.               { return ILLEGAL_CHAR; }
```

Dieser Lexer **kehrt** jetzt wiederum mit **`return` zurück**, sobald er eine Token erkannt hat.

Neu sind die Klammern und „`=`“, dafür haben wir `CLEAR` zur Vereinfachung **weggelassen**.

Den Code, der die **Aktionen** durchgeführt hat, haben wir **gelöscht**.

Stattdessen wird der **Typ der Eingabe** als **Returncode** übergeben. Außerdem wird eine **eingegebene Zahl** in der Variablen `yylval.zahl` an den den Parser übergeben.

Die **Variable** `yylval` wird **von Yacc** zur Verfügung gestellt, um Werte vom Lexer an den Parser zu übergeben. Da ein Wert je nach Token einen **verschiedenen Datentyp** haben kann, ist `yylval` eine **C-Union** aller dieser Datentypen. Daher müssen wir immer **auf** die **passende Komponente zugreifen**, hier `zahl`. Die Komponenten sind im **`%union`-Konstrukt** am Anfang der Yacc-Datei deklariert:
`calc.y`:

```
%{
#include <stdio.h>
%}
%union {
    double zahl;
}
%token ILLEGAL_CHAR
%token <zahl> NUMBER
%left '-' '+'
%left '*' '/'
%type <zahl> term
%%
eingabe:          /* empty */
                | eingabe berechnung
                ;
berechnung:       term '=' { printf("Ergebnis: %g\n", $1); }
                ;
term:             NUMBER
                | term '+' term { $$ = $1 + $3; }
                | term '-' term { $$ = $1 - $3; }
                | term '*' term { $$ = $1 * $3; }
                | term '/' term { $$ = $1 / $3; }
                | '(' term ')' { $$ = $2; }
                ;
%%
int yyerror(char *msg) {
    printf("\nEingabefehler: %s\n", msg);
    return 0;
}
int main() {
    return yyparse();
}
```

Hier haben wir **nur einen Datentyp** und daher nur eine Union-Komponente.

Wir haben wieder eine `main()`**-Funktion**. Und unsere `yyerror()`**-Funktion** macht nun eine **Ausgabe**, um den Benutzer über die Art des Fehlers zu informieren.

Mit den `%token`-Anweisungen definieren wir wieder die **erlaubten Token**. Die Token-Definitionen und die Deklaration von `yylval` gehen wieder in die **Datei** `calc.tab.h`.

In der **Grammatik** gibt es etwas neues: **Aktionen** in geschweiften Klammern. Am Ende einer `berechnung` wird das **Ergebnis ausgegeben**.

Die **Variable** `$1` enthält den **Wert des ersten Tokens**, hier von `term`.

In der Regel für `term` sehen wir **weitere solche Variablen**, die auf das **zweite** und **dritte** Token zugreifen.

Die Variable `$$` nimmt den **Wert des gesamten Ausdrucks** auf, so daß der Wert von `term` in weiteren Regeln verwendet werden kann.

Mit der `%type`-Anweisung oben haben wir festgelegt, daß `term` eine `zahl` ist, also vom Typ `double`.

Wenn der **Lexer** eine `NUMBER` zurückliefert, und wenn `yylval.zahl` die **Zahl** enthält, dann kann man also in Yacc über `$1` auf diese Zahl **zugreifen**. Der **Typ** von `NUMBER` ist oben in der **erweiterten Token-Anweisung** deklariert worden.

Wenn keine Aktion angegeben ist, dann ist die **Default-Aktion**, `$1` auf `$$` zuzuweisen. Dies ist hier bei `NUMBER` geschehen.

6.4 Operator-Assoziativität

▶▶ `make calc`

▶▶ `./calc`

Was passiert bei

▶▶ `9-2=`

▶▶ `9-2-1=`

Es gibt **zwei Möglichkeiten**, den Ausdruck **zu parsen**, entweder `(9-2)-1` oder `9-(2-1)`. Die **Grammatik ist mehrdeutig**. Durch die **`%left`-Anweisung** am Anfang sagen wir Yacc, daß wir nur die erste Möglichkeit haben wollen.

Eine `%left`-Anweisung ist **wie** eine **`%token`-Anweisung**, nur daß der Operator immer **linksassoziativ** ist.

Entsprechend gibt es auch `%right`.

Und es gibt `%nonassoc` für Operatoren, die **nicht mehrfach verwendet** werden dürfen, wie z.B. `if(a<x<b)` in vielen Sprachen **nicht erlaubt** ist.

6.5 Operator-Präzedenz

Was passiert bei

▶▶ `2+3*4=`

▶▶ `3*4+2=`

Woher „weiß“ der Rechner, daß **Punktrechnung vor Strichrechnung** geht? Die **Grammatik** ist wiederum **mehrdeutig**.

Die **Token** haben eine **Präzedenz**. **Um so später** sie deklariert werden, **um so höher** ist die Präzedenz. Hier werden `*` und `/` nach `+` und `-` deklariert. Token auf der **gleichen Zeile** haben **gleiche Präzedenz**.

Wenn die **Präzedenz gleich** ist, dann kommt wieder die **Assoziativität** zum Tragen.

6.6 Fehlerbehandlung

Was passiert bei Eingabefehlern?

▶▶ `2+=`

Diese Fehlermeldung ist **nicht** sehr **aussagekräftig**.

Man kann **mehr Informationen** ausgeben lassen, wenn man das **Makro** `YYERRROR_VERBOSE` im **C-Deklarationsteil** auf einen beliebigen Wert definiert.

▶▶ Machen.

▶▶ `make calc`

▶▶ `./calc`

▶▶ `2+=`

Wenn man die **Token einzenld** bearbeiten läßt, sieht man auch genau, wo es schiefgeht:

▶▶ `./calc`

▶▶ `2`

▶▶ `+`

▶▶ `=`

So sieht man, **welche Token erwartet** werden.

Man kann sich auch die **Zeilennummer** in der Eingabe ausgeben lassen, aber das macht bei **interaktiven Programmen wenig Sinn**.

Oft will man den **Fehler** auch **im Parser behandeln** und dann die **Bearbeitung fortsetzen**, zumindest vorläufig.

Hierfür gibt es das besondere Token `error`:
`calc-err.y`:

```
%{
#include <stdio.h>
#define YYERROR_VERBOSE
%}
%union {
    double zahl;
}
%token ILLEGAL_CHAR
%token <zahl> NUMBER
%left '-' '+'
%left '*' '/'
%type <zahl> term
%%
eingabe:          /* empty */
                | eingabe berechnung
                ;
berechnung:       term '=' { printf("Ergebnis: %g\n", $1); }
                | error '='{ yyerrok; printf("Neue Eingabe:\n"); }
                ;
term:             NUMBER
                | term '+' term { $$ = $1 + $3; }
                | term '-' term { $$ = $1 - $3; }
                | term '*' term { $$ = $1 * $3; }
                | term '/' term { $$ = $1 / $3; }
                | '(' term ')' { $$ = $2; }
                ;
%%
int yyerror(char *msg) {
    printf("\nEingabefehler: %s\n", msg);
    return 0;
}
int main() {
    return yyparse();
}
```

Wenn ein **Token gefunden** wird, auf das **keine Regel paßt**, ist das ein Fehler. Dann wird **versucht,** das Token `error` **zu matchen**. Wenn das in der **aktuellen Regel nicht** geht, wird diese **abgebrochen,** und das gleiche in der **nächst höheren halbfertigen Regel** versucht.

Wenn eine `error`-Regel **gefunden** wurde, dann **müssen drei Token passen**, damit **wieder normal** bearbeitet wird.

In unserem Falle sind wir **bei** einem `=` **sicher**, daß wir **wieder richtig** sind. Daher wird `yyerrok` **aufgerufen**, das den Parser **sofort** wieder in den **Normalzustand** bringt.

▶▶ `make calc-err`

▶▶ `./calc-err`

6.7 Aufruf-Optionen von Bison

Aufruf-Optionen von Bison

Option	Kurzform	Bedeutung
`--defines`	`-d`	Schreibe die `.tab.h`-Datei für Lex.
`--verbose`	`-v`	Schreibe eine `.output`-Datei mit allen Parser-Zuständen und allen Regel-Konflikten
`--debug`	`-t`	Definiere das Makro `YYDEBUG`, so daß Debugging möglich wird.
`--name-prefix=`*prefix*	`-p` *prefix*	Ersetze den Präfix yy in allen Namen durch den angebenen Präfix. (Erlaubt mehrere Parser.)
`--help`	`-h`	Gibt alle Kommandozeilen-Optionen aus.
...	...	...

Kurs Unix-Tools, WS 01/02, Jan Bredereke 43

Wird im **C-Deklarationsteil** das Makro **YYDEBUG** definiert, und wird die **Variable yydebug** auf einen Wert ungleich 0 gesetzt, dann macht der Parser sehr viele **Debugger-Ausgaben**. Man kann genau sehen, **wann welche Regel** angewandt wird. Die **Ausgabe** ist aber *sehr* **umfangreich**.

6.8 Weitere Features

Man kann auch **Aktionen in der Mitte von Regeln** haben.

Die **Variablen** `$1` für die **semantischen Werte** usw. können auch **variable Typen** haben.

Man kann die **Präzedenz von Operatoren für einzelne Regeln ändern**.

Man kann auch explizit angeben, welches die **Start-Regel** ist.

Mit den **Makros YYABORT** und **YYACCEPT** kann man das **Parsen sofort beenden**, entweder als Fehler oder als Erfolg.

Mit dem **Makro YYERROR** kann man **aus** einer **Aktion** heraus die **Fehlerbehandlung** anstoßen, für kontextsensitive Fehler.

Den **Status der Fehlerbehandlung** kann man mit **YYRECOVERING** abfragen.

Wenn bei der **Fehlerbehandlung** das **Look-Ahead-Token** im Weg ist, kann man es mit **yyclearin löschen** lassen.

6.9 Bison und andere Versionen von Yacc

Bison **kann** praktisch **alles**, was AT&s Yacc auch kann.

Zusätzliche Features von Bison:

- **Bessere Zeilennummer-Ausgabe** bei Fehlern.
- Einen **anderen Dateinamen** für die Ausgabedatei. (Standard: *Immer* `y.tab.c`)
- Eine Option, das `yy` in **allen Namen umzubenennen**, um mehrere Parser in einem Programm haben zu können.
- Einige kleine **technische** Zusätze.