Einführung in yacc

Inhaltsverzeichnis

1 Einleitung, Motivation

- `yacc` ist ein universelles Werkzeug für die syntaktische Analyse (Parsergenerator) ("yet another compiler compiler")

- Entwicklung in den 70er Jahren, GNU Version heißt `bison`

- Unentbehrlich bei der Erstellung eines LALR(1)-Parsers, da Handkodierung sehr mühsam und fehleranfällig

Einsatzgebiet (meistens zusammen mit `lex`):

- Prinzipiell überall dort, wo Eingaben mit einer komplexen syntaktischen Struktur verarbeitet werden müssen
- Compilerbau
- Design und Implementierung von Kommandoprozessoren
- Menügeneratoren, ...

Mittels `lex` und `yacc` wurden hunderte von Compiler-Frontends erstellt

Vorgehensweise bei der Erstellung eines Parsers mit `lex` und `yacc` :

1 a) Spezifikation lexikalischer Einheiten (Token) als reguläre Ausdrücke in `lex`-Programm (z.B. `meinProg.lex`)

b) Übersetzung nach `lex.yy.c`

2 a) Spezifikation der syntaktischen Struktur als kontextfreie Grammatik in `yacc`-Programm (z.B. `meinProg.yacc`)

b) Einbinden des Scanners (`#include "lex.yy.c"`) in `yacc`-Programm

c) Übersetzung nach `y.tab.c`

3 a) ggf. Einbinden in eigenes Programm, Übersetzung in Maschinencode (Linken von Libraries `liby.a` und `libl.a`)

2 yacc-Programme

2.1 Struktur

Definitionsteil
%%
Regelteil
%%
Benutzerdefinierte Routinen

Spezifikation einer kontextfreien Grammatik $G = (N, T, S, P)$ in `yacc`-Programm:

Definitionsteil:
Definition von T und S

Regelteil:
Definition von N und P

Benutzerdef. Routinen:
u.a. Routine `int yylex()`, welche Token des Eingabetextes liefert (wird in der Regel durch `lex` realisiert)

2.2 Der Definitionsteil

Funktion des Definitionsteils:

- Definition von Token (T)
- Festlegung des Startsymbols (S)
- Einbinden von C-Code
- Prioritäts- und Assoziativitätsbedingungen (Kapitel 3)
- Definition beliebiger Rückgabetypen (Kapitel 4.1)

2.2.1 Definition von Token

`lex` liefert die Token als Integer-Werte (Genauer: `lex.yy.c` enthält die Funktion `int yylex()`, die von `yacc` benutzt wird)

Wie werden die Token mit Integer-Werten versehen?

Zum konsistenten Gebrauch der Tokenkodierung in `lex` und `yacc` wird wie folgt vorgegangen:

- Token ist ein einzelnes Zeichen (Literal) $\rightarrow$ ASCII-Wert wird verwendet $(1 - 256)$
- Token ist eine Zeichenkette, die von `lex` als regulärer Ausdruck erkannt wurde
  - Es wird ein Tokenname in `yacc` definiert und entsprechende symbolische Konstanten erzeugt (≥ 257)
  - Die symbolischen Konstanten werden dem `lex`-Programm zugänglich gemacht und dort als Rückgabewert verwendet
- Tokenwert ≤ 0 signalisiert Ende des Eingabetextes

Tokendefinition in `yacc`

```
%token NAME_1 NAME_2 ... NAME_n
```

oder

```
%token NAME_1
%token NAME_2
.
.
%token NAME_n
```

- Gültige Namen für Token sind C-Identifier
- *Konvention*: Tokennamen werden in Großbuchstaben geschrieben

Generierung der symbolischen Konstanten

Die Option `-d` beim Aufruf von `yacc`, z.B.

```
yacc -d meinProg.yacc
```

erzeugt zusätzlich zu `y.tab.c` die header-Datei `y.tab.h`, welche C-Konstantendefinitionen für die definierten Token enthält

Verwendung im `lex`-Programm

- Die header-Datei `y.tab.h` wird im Definitionsteil des `lex`-Programmes eingebunden[a]
- Die Aktionen zu den regulären Ausdrücken enden mit return-Anweisungen, die als Rückgabewert die symbolische Konstante für das erkannte Token verwenden

[a] Falls `lex.yy.c` nicht in `y.tab.c` inkludiert wird.

Beispiel 1 (Number/Identifier)
Bei der Verarbeitung eines Eingabetextes durch das `yacc`*-Programm* `numId.yacc` *sollen für ganze Zahlen und C-Bezeichner die Token* `NUMBER` *und* `IDENTIFIER` *verwendet werden.*

1. *Definition der Token in* `numId.yacc`

```
%token NUMBER IDENTIFIER
%%
...
```

2. *Generierung symbolischer Konstanten*

```
yacc -d numId.yacc
```

3. *Inhalt der Headerdatei* `y.tab.h`

```
#define NUMBER 257
#define IDENTIFIER 258
```

4. *Verwendung der symbolischen Konstanten im* `lex`*-Programm* `numId.lex`

```
%{
#include "y.tab.h"
%}
%%
[+-]?(0|[1-9][0-9]*)    return(NUMBER);
[a-zA-Z_][a-zA-Z0-9_]*  return(IDENTIFIER);
.                       return(yytext[0]);
\n                      return('\n');
```


2.2.2 Festlegen des Startsymbols

Das Startsymbol wird folgendermaßen festgelegt:

```
%start nonterminal
```

- Definition nicht zwingend, aber empfohlen für bessere Lesbarkeit
- Falls kein Startsymbol definiert wird, wird defaultmäßig die linke Seite der ersten Regel als Startsymbol interpretiert.

2.2.3 Einbinden von C-Code

Analog zu `lex`-Programm mit

```
%{

  C-Anweisungen

%}
```

2.3 Der Regelteil

Funktion des Regelteils:

- Festlegen der Nonterminals (N)
  (geschieht implizit)
- Spezifikation der Produktionen (P) und dazugehöriger Aktionen

2.3.1 Aufbau des Regelteils

Eine Produktion

$$\begin{array}{lcl} A & \rightarrow & X_{11}X_{12}...X_{1n_1} \\ & | & X_{21}X_{22}...X_{2n_2} \\ & | & ... \\ & | & X_{m1}X_{m2}...X_{mn_m} \end{array}$$

mit $A \in N$, $X_{ij} \in N \cup T$

wird als `yacc`-Regel in der folgenden Form angegeben:

$$\begin{array}{lcl} \mathit{lsymbol} & : & \mathit{rsymbol}_{11} \ \{ \ \mathit{aktion}_{11} \ \} \ \ldots \ \mathit{rsymbol}_{1n_1} \ \{ \ \mathit{aktion}_{1n_1} \} \\ & | & \mathit{rsymbol}_{21} \ \{ \ \mathit{aktion}_{21} \ \} \ \ldots \ \mathit{rsymbol}_{2n_2} \ \{ \ \mathit{aktion}_{2n_2} \} \\ & | & \ldots \\ & | & \mathit{rsymbol}_{m1} \ \{ \ \mathit{aktion}_{m1} \ \} \ \ldots \ \mathit{rsymbol}_{mn_m} \ \{ \ \mathit{aktion}_{mn_m} \} \\ & ; & \end{array}$$

2.3.2 Terminal- und Nichtterminalsymbole

Terminal- und Nichtterminalsymbole müssen gültige C-Bezeichner als Namen haben

Terminalsymbole

- Token aus Definitionsteil
- Literale, d.h. einzelne Zeichen in Hochkomma, z.B. `'\n'`

Nichterminalsymbole

implizit definiert durch

- Linke Seite einer Regel
- Alle Symbole der rechten Seite, welche nicht Terminalsymbol sind

Jedes vorkommende Nichtterminalsymbol muß auf der linken Seite von mindestens einer Regel stehen.

Konvention: Nichtterminalsymbole werden klein geschrieben

2.3.3 ε-Regeln

ε-Regeln werden durch eine leere rechte Seite realisiert

$$\begin{array}{rcl} lsymbol & : & \texttt{/* leer */} \\ & | & rsymbol_1\ rsymbol_2 \\ & ; & \end{array}$$

oder explizit

$$\begin{array}{rcl} lsymbol & : & epsilon \\ & | & rsymbol_1\ rsymbol_2 \\ & ; & \\ & & \\ epsilon & : & \texttt{/* leer */} \\ & ; & \end{array}$$

2.3.4 Aktionen

Zu jedem Symbol der rechten Seite kann eine zugehörige Aktion angegeben werden.

Zur Erinnerung:

$$\begin{array}{rcl} lsymbol & : & rsymbol_{11}\ \{\ aktion_{11}\ \}\ \ldots\ rsymbol_{1n_1}\ \{\ aktion_{1n_1}\ \} \\ & | & rsymbol_{21}\ \{\ aktion_{21}\ \}\ \ldots\ rsymbol_{2n_2}\ \{\ aktion_{2n_2}\ \} \\ & | & \ldots \\ & | & rsymbol_{m1}\ \{\ aktion_{m1}\ \}\ \ldots\ rsymbol_{mn_m}\ \{\ aktion_{mn_m}\ \} \\ & ; & \end{array}$$

- Eine Aktion besteht aus einer oder mehreren C-Anweisungen in geschweiften Klammern.
- Eine Aktion $aktion_{ij}$ wird dann ausgeführt, wenn das Symbol $rsymbol_{ij}$ abgedeckt worden ist.

Aktionen, die nicht am Ende einer Regel angegeben sind, werden wie ein nichtterminales Symbol behandelt.

Es wird intern eine zusätzliche Regel eingefügt, welche als linke Seite dieses Nichtterminalsymbol hat und als rechte Seite nur die Aktion selbst.

Beispiel: Die Regel

$$
\begin{array}{rcl}
\mathit{lsymbol} & : & \mathit{rsymbol}_1 \ \{\mathit{aktion}_1\} \ \mathit{rsymbol}_2 \\
 & ; &
\end{array}
$$

wird intern umgewandelt zu

$$
\begin{array}{rcl}
\mathit{lsymbol} & : & \mathit{rsymbol}_1 \ \mathit{akt}_1 \ \mathit{rsymbol}_2 \\
 & ; & \\
\mathit{akt}_1 & : & \{\mathit{aktion}_1\} \\
 & ; &
\end{array}
$$

2.3.5 Bindung von Werten an Symbole

- Es ist möglich, Werte an Terminal- und Nichtterminalsymbole zu binden.
- Defaultmäßig sind dies `int`-Werte (es können beliebige Datentypen verwendet werden → Kapitel 4.1)

Verwendung und Modifikation von Werten der rechten Seite

Werte der Symbole einer rechten Seite

$\mathit{rsymbol}_1 \ \ldots \ \mathit{rsymbol}_n$

sind über die Variablen

`$1 ... $n`

zugreifbar.
(Beachte: Aktionen, die nicht am Ende einer Regel stehen, werden ebenfalls als Symbol behandelt.)

Verwendung und Modifikation des Wertes der linken Seite

- Wert der linken Seite ist mittels des Platzhalters `$$` zugreifbar.
- Werte der linken Seiten der darüberstehenden Regeln sind über die Variablen `$0`, `$-1`, `$-2`, ... zugreifbar.

Beispiel: Zuweisung an Nichtterminal *a*:

```
a   :   b '+' c {$$ = $1 + $3;}
    ;
```

- Falls keine ganz rechts stehende Aktion angegeben wird, wird defaultmäßig der Wert des ersten Symbols der rechten Seite der Variablen `$$` zugewiesen.

Beispiel:

```
a   :   b '+' c {$$ = $1;} /* default */
    ;
```


- Achtung: Für "innere Aktionen" referenziert der Platzhalter `$$` das künstlich erzeugte Nichtterminal der Aktion und nicht den Wert der linken Seite.

Beispiel: Die Regel

```
a   :   b {$$ += $1;} '+' c {$$ += $4;}
    ;
```

weist *a* nur den Wert von *c* zu.

Zuweisung von Werten an Terminalsymbole

- Verwendung der gemeinsamen Variablen `yylval` in `lex` und `yacc`
- Zuweisung des Wertes eines Token an `yylval` während des Scannens

Beispiel 2 (Dual Nach Dezimal)

Das `yacc`*-Programm* `dualNachDezimal.yacc` *liest Dualzahlen und wandelt diese in Dezimalzahlen um.*

2.3.6 Fehlerbehandlung

Default Fehlerbehandlung

Bei Lesen eines Tokens, welches nicht in die syntaktische Struktur paßt:

- Ausgabe der Meldung `"syntax error"`
- Abbruch

Wünschenswert ist eine Fehlerbehandlung, die

- präzisere Information über einen aufgetretenen Fehler liefert
  (Grund, Zeilennummer, ...)
- mit der Syntaxanalyse an einem geeigeneten Punkt fortfährt

Das Symbol *error*

- `yacc` stellt “künstliches” Nichtterminal *error* zur Verfügung
- Verwendung bei “fehlergefährdeten” Produktionen als alternative rechte Seite
- Zusätzlich Angabe eines “Aufsetztoken” möglich

Beispiel:

expression-statement	:	*expression*
	|	*error* `';'` {`froutine();`}
	;	

- `';'` ist das Aufsetztoken
- `froutine()` könnte eigene Fehlerbehandlungsroutine sein

Funktionsweise:

- Fehler beim Lesen eines Tokens → Fehlerbehandlungsmodus wird eingeschaltet
- Solange Zustände vom Stack entfernen, bis Zustand erreicht ist, in dem der Punkt vor Nichtterminal *error* steht
- Falls solch ein Zustand nicht existiert → Ausgabe von “syntax error” und Abbruch
- Solange Eingabezeichen lesen, bis Aufsetztoken erkannt wird
- Fehlerbehandlungsmodus wird verlassen, sobald drei gültige Eingabesymbole erkannt werden
- Direktes Verlassen durch Aufruf der Funktion `yyerrok()` möglich

Die Funktion `yyerror`

Funktion

`void yyerror(char *s)`

ist zuständig für die Ausgabe der Fehlermeldung

Anpassung der Fehlerausgabe durch Überschreiben der Funktion möglich

Die char-Variable `yychar`

enthält die Tokennummer des aktuell gelesenen Tokens

Beispiel 3 (Taschenrechner)
Es soll ein einfacher Taschenrechner entwickelt werden, der folgende Operatoren realisiert

`+`	*Addition*
`-`	*Subtraktion*
`*`	*Multiplikation*
`/`	*Division*

Es werden nur ganze Zahlen verarbeitet.

2.4 Benutzerdefinierte Routinen

Funktion:

- Code wird unverändert nach `y.tab.c` kopiert
- Einbinden von benutzerdefinierten Funktionen, die in den Aktionen benutzt werden können
- Bereitstellung der Funktion `int yylex()`
- ggf. Überschreiben der Funktionen `yyerror()` und `main`

3 Fehlersuche und Auflösen von Konflikten

3.1 Fehlersuche

- Aktivieren des "Trace"-Modus durch Übersetzen von `y.tab.c` mit der Option `-DYYDEBUG`. Bei manchen `yacc` Realisierungen muss zusätzlich im Definitionsteil noch folgender C-Code eingefügt werden:

```
%{
#ifdef YYDEBUG
int yydebug = 1;
#endif
%}
```

- Generierung einer lesbaren Action- und Goto-Tabelle in Datei `y.output` (Option `-v` für `yacc` benutzen)

3.2 Auflösen von Konflikten

- `yacc` verarbeitet $LALR(1)$-Grammatiken
- Ist geg. Grammatik nicht $LALR(1)$, so meldet `yacc` Konflikt:
  - shift/reduce-Konflikt
  - reduce/reduce-Konflikt
- Konflikte lassen sich direkt in der (mit Option `-v`) generierten Action- und Goto-Tabelle ablesen

Beispiel 4 (shift/reduce-Konflikt)
Der Konflikt tritt in Zustand 4 bei Lookaheadzeichen 'Y' auf, weil unklar ist, ob geshiftet oder die Regel $b \rightarrow X$ reduziert werden soll.

Beispiel 5 (reduce/reduce-Konflikt)
Der Konflikt tritt in Zustand 4 bei Lookaheadzeichen 'B' auf, weil unklar ist, ob die Regel $x \rightarrow A$ oder $y \rightarrow A$ zur Reduktion benutzt werden soll.

Default Konfliktlösung

Konflikte werden von `yacc` nach folgenden Regeln aufgelöst

- shift/reduce Konflikt
  $\rightarrow$ es wird shift angewendet
- reduce/reduce-Konflikt
  $\rightarrow$ es wird erstangegebene Regel im Regelteil zur Reduktion verwendet

Besser: Konflikt selbst auflösen durch

- Transformation der Grammatik oder
- Festlegung von Prioritäts- und Assoziativitätsbedingungen

Beispiel 6 (Keine Assoziativität)
Es tritt ein shift/reduce-Konflikt in Zustand 4 bei Lookaheadzeichen '-' auf.
Dieser soll durch Zuweisung einer Assoziativität an das Minuszeichen beseitigt werden.

3.2.1 Assoziativitätsbedingungen

Die Befehle `%left` und `%right`

Mit den Anweisungen

```
%left TOKEN_1 ... TOKEN_n
```

oder

```
%left TOKEN_1
...
%left TOKEN_n
```

lassen sich Token im Definitionsteil als *linksassoziativ* definieren.
Analog werden mit `%right` *rechtsassoziative* Token definiert.

Beispiel 7 (Linksassoziativität)
Der Konflikt wird zugunsten des reduce aufgelöst.
Linksassoziativität → semantische Auswertung von links nach rechts

Beispiel 8 (Rechtsassoziativität)
Der Konflikt wird zugunsten des shift aufgelöst.
Rechtsassoziativität → semantische Auswertung von rechts nach links

3.2.2 Prioritätsbedingungen

Die Reihenfolge der `%left` und `%right` Angaben bestimmen die Prioritäten der einzelnen Token (aufsteigende Priorität).

Zum Beispiel legen die Anweisungen

```
%left '+' '-'
%left '*' '/'
```

fest, dass

- `'+'`, `'-'`, `'*'`, `'/'` linksassoziativ sind
- `'*'` und `'/'` höhere Priorität haben als `'+'` und `'-'`
- `'*'` und `'/'` sowie `'+'` und `'-'` jeweils gleiche Priorität haben

Beispiel 9 (Erweiterter Taschenrechner)
Der Taschenrechner aus Beispiel 3 wird erweitert, so dass Register mit den Namen `a` *…* `z` *zur Verfügung stehen, sowie die Operatoren*

`+`	*Addition*
`-`	*Subtraktion*
`*`	*Multiplikation*
`/`	*Division*
`%`	*Modulo*
`&`	*Bitweises UND*
`|`	*Bitweises ODER*
`^`	*Potenz*
`=`	*Zuweisung*

Auf der linken Seite einer Zuweisung sollen dabei nur Registernamen erlaubt sein.

Weiterhin sind nur ganze Zahlen erlaubt.

Der Taschenrechner arbeitet noch nicht ganz korrekt:
Bei der Eingabe

```
-2^4
```

wird als Ergebnis `-16` *anstatt* `16` *ausgegeben.*

Grund: Festgelegte Priorität für das Minuszeichen bezieht sich auf den Subtraktionsoperator; diese ist niedriger als die des Potenzoperators.

Der Befehl `%prec`

Mit der Anweisung

`%prec TOKEN`

auf der rechten Seite einer Regel läßt sich dieser die Priorität des angegebenen Tokens zuweisen.

Beispiel 10 (Mod. erw. Taschenrechner)
Der Taschenrechner aus Beispiel 9 wird nun so modifiziert, dass das Minus-Vorzeichen die höchste Priorität hat.

4 Sonstiges

4.1 Beliebige Rückgabetypen

Der Standard Datentyp für den Rückgabewert von Aktionen (`$$`, `$1`, ...) oder des einem Token zugeordneten Wertes (`yylval`) ist `int`.

Sollen weitere Datentypen für die Rückgabe verwendet werden, so müssen diese mittels des `%union` Schlüsselwortes im Definitionsteil festgelegt werden:

```
%union{
    typ_1   typname_1
    typ_2   typname_2
    ...
}
```


Anwendung bei Token

Es ist zu kennzeichnen, welchen Rückgabedatentyp ein Token haben soll:

```
%token <typname_i> TOKEN
```

Anwendung bei Nichtterminalen

Für Nichtterminale wird mittels des `%type` Schlüsselwortes der Rückgabewert festgelegt:

```
%type  <typname_i> nichtterminal
```

Beispiel 11 (Double Erw. Taschenrechner)
Der Taschenrechner aus Beispiel 10 wird nun so modifiziert, dass er auch mit Gleitpunktzahlen rechnen kann.

5 Referenzen

- `lex/yacc` -Übersichtseite mit Download-Verweisen, Dokumentation und Literaturhinweisen
  `http://dinosaur.compilertools.net/`
- Umfangreiche Übesichtseite über Tools zum Compilerbau
  `http://catalog.compilertools.net/`