

Compiler Construction

Lecture 14: Syntactic Analysis X

Thomas Noll

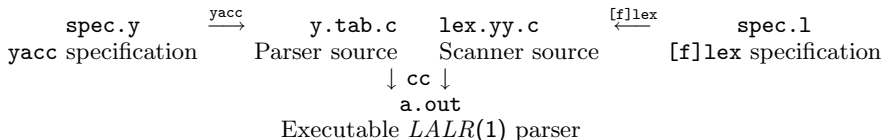
Lehrstuhl für Informatik 2
RWTH Aachen University
noll@cs.rwth-aachen.de

<http://www-i2.informatik.rwth-aachen.de/Teaching/Course/CB/2006/>

Winter semester 2006/07

- 1 Generating Parsers Using `yacc`
- 2 Expressiveness of LL and LR Grammars
- 3 LL and LR Parsing in Practice

Usage of **yacc** (“yet another compiler compiler”):



Like for `[f]lex`, a **yacc specification** is of the form

Declarations (optional)

%%

Rules

%%

Auxiliary procedures (optional)

- Declarations:
- Token definitions: `%token Tokens`
 - Not every token needs to be declared (`'+' , '=' , ...`)
 - Start symbol: `%start Symbol` (optional)
 - C code for declarations etc.: `%{ Code %}`

Rules: context-free productions and semantic actions

- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ represented as
$$\begin{array}{l} A \quad : \quad \alpha_1 \quad \{Action_1\} \\ \quad \quad | \quad \alpha_2 \quad \{Action_2\} \\ \quad \quad \vdots \\ \quad \quad | \quad \alpha_n \quad \{Action_n\}; \end{array}$$
- Semantic actions = C statements for computing attribute values
- `$$` = attribute value of A
- `$i` = attribute value of i th symbol on right-hand side
- Default action: `$$ = $1`

Auxiliary procedures: scanner (if not `[f]lex`), error routines, ...

Example: Simple Desk Calculator I

```
%{ /* SLR(1) grammar for arithmetic expressions (Example 12.5) */
#include <stdio.h>
#include <ctype.h>
%}
%token DIGIT
%%
line      : expr '\n'          { printf("%d\n", $1); };
expr     : expr '+' term      { $$ = $1 + $3; }
          | term
term      : term '*' factor    { $$ = $1 * $3; }
          | factor
factor    : '(' expr ')'      { $$ = $2; }
          | DIGIT             { $$ = $1; };
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) yylval = c - '0'; return DIGIT;
    return c;
}
```

Example: Simple Desk Calculator II

```
> yacc calc.y
> cc y.tab.c -ly
> a.out
2+3
5
> a.out
2+3*5
17
```

An Ambiguous Grammar I

```
%{ /* Ambiguous grammar for arithm. expressions (Ex. 13.13) */
#include <stdio.h>
#include <ctype.h>
%}
%token DIGIT
%%
line      : expr '\n'          { printf("%d\n", $1); };
expr      : expr '+' expr     { $$ = $1 + $3; }
          | expr '*' expr     { $$ = $1 * $3; }
          | DIGIT             { $$ = $1; };
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {yylval = c - '0'; return DIGIT;}
    return c;
}
```

An Ambiguous Grammar II

Invoking yacc with the option `-v` produces a report `y.output`:

```
...  
State 8  
  
2 expr: expr . '+' expr  
2   | expr '+' expr .  
3   | expr . '*' expr  
  
'+' shift and goto state 6  
'*' shift and goto state 7  
  
'+' [reduce with rule 2 (expr)]  
'*' [reduce with rule 2 (expr)]
```

```
State 9  
  
2 expr: expr . '+' expr  
3   | expr . '*' expr  
3   | expr '*' expr .  
  
'+' shift and goto state 6  
'*' shift and goto state 7  
  
'+' [reduce with rule 3 (expr)]  
'*' [reduce with rule 3 (expr)]
```

Conflict Handling in yacc

Default conflict resolving strategy in yacc:

reduce/reduce: choose **first conflicting production** in specification

shift/reduce: prefer **shift**

- resolves dangling–else ambiguity (Example 13.14) correctly
- also adequate for * after sum (Example 13.13) and for right–associative binary operators
- not appropriate for left–associative binary operators (\implies reduce; see Example 13.13)

For ambiguous grammar:

```
> yacc ambig.y
conflicts: 4 shift/reduce
> cc y.tab.c -ly
> a.out
2+3*5
17
> a.out
2*3+5
16
```

Precedences and Associativities in yacc I

General mechanism for resolving conflicts:

$$\begin{array}{l} \%[\text{left}|\text{right}] \textit{Operators}_1 \\ \vdots \\ \%[\text{left}|\text{right}] \textit{Operators}_n \end{array}$$

- operators in one line have given associativity and same precedence
- precedence increases over lines

Example 14.1

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

^ (right associative) binds stronger than * and / (left associative), which in turn bind stronger than + and - (left associative)

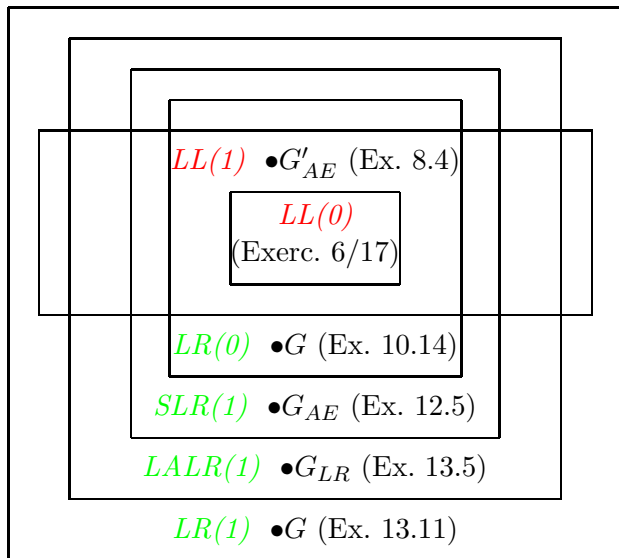
Precedences and Associativities in yacc II

```
%{ /* Ambiguous grammar for arithmetic expressions
    with precedences and associativities */
#include <stdio.h>
#include <ctype.h>
%}
%token DIGIT
%left '+'
%left '*'
%%
line   : expr '\n' { printf("%d\n", $1); };
expr   : expr '+' expr { $$ = $1 + $3; }
       | expr '*' expr { $$ = $1 * $3; }
       | DIGIT         { $$ = $1; };
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {yylval = c - '0'; return DIGIT;}
    return c;
}
```

Precedences and Associativities in yacc III

```
> yacc nonambig.y
> cc y.tab.c -ly
> a.out
2*3+5
11
> a.out
2+3*5
17
```

- 1 Generating Parsers Using yacc
- 2 Expressiveness of LL and LR Grammars
- 3 LL and LR Parsing in Practice

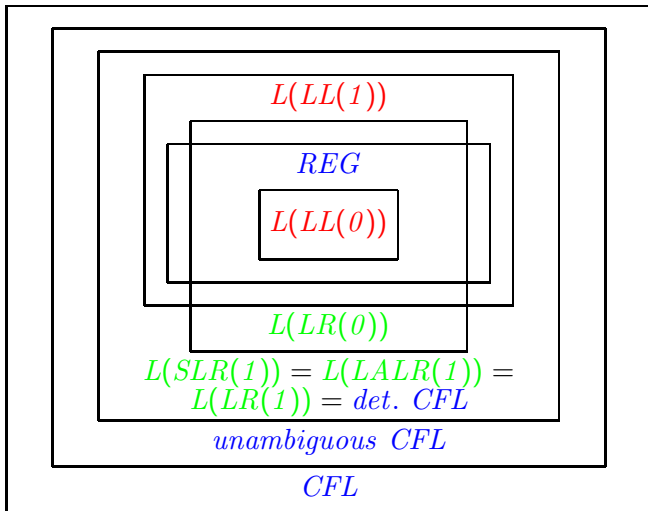


Moreover:

- $LL(k) \subsetneq LL(k+1)$
for every $k \in \mathbb{N}$
- $LR(k) \subsetneq LR(k+1)$
for every $k \in \mathbb{N}$
- $LL(k) \subsetneq LR(k)$
for every $k \in \mathbb{N}$

Overview of Language Classes

(cf. O. Mayer: *Syntaxanalyse*, BI-Verlag, 1978, p. 409ff)



Moreover:

- $L(LL(k)) \subsetneq L(LL(k+1)) \subsetneq L(LR(1))$
for every $k \in \mathbb{N}$
- $L(LR(k)) = L(LR(1))$
for every $k \geq 1$

- 1 Generating Parsers Using yacc
- 2 Expressiveness of LL and LR Grammars
- 3 LL and LR Parsing in Practice

In practice: use of *LL(1)* or *LALR(1)*

Detailed comparison (cf. Fischer/LeBlanc: *Crafting a Compiler*, Benjamin/Cummings, 1988):

Simplicity : LL wins

- LL parsing technique easier to understand
- recursive-descent parser easier to debug than LALR action tables

Generality : LALR wins

- “almost” $LL(1) \subseteq LALR(1)$ (only pathological counterexamples; cf. Assignment 8/Exercise 24)
- LL requires elimination of left recursion and left factorization

Semantic actions : LL wins

- (see semantic analysis)
- actions can be placed anywhere in LL parsers without causing conflicts