Compiler Construction

Lecture 13: Syntactic Analysis IX(Wrap-Up)/ Semantic Analysis I (Attribute Grammars)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc09/

Winter semester 2009/10

Outline

Usage of yacc ("yet another compiler compiler"):

`spec.y` yacc specification	$\xrightarrow{\texttt{yacc}}$	`y.tab.c` Parser source	`lex.yy.c` Scanner source	$\xleftarrow{\texttt{[f]lex}}$	`spec.l` [f]lex specification

↓ cc ↓

`a.out`
Executable $LALR(1)$ parser

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

Declarations (optional)
%%
Rules
%%
Auxiliary procedures (optional)

yacc Specifications

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

- $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ represented as

$$\begin{array}{llll} A & : & \alpha_1 & \{Action_1\} \\ & | & \alpha_2 & \{Action_2\} \\ & \vdots & & \\ & | & \alpha_n & \{Action_n\}; \end{array}$$

- Semantic actions = C statements for computing attribute values
- `$$` = attribute value of A
- `$i` = attribute value of ith symbol on right-hand side
- Default action: `$$ = $1`

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

Example: Simple Desk Calculator I

```
%{/* SLR(1) grammar for arithmetic expressions (Example 11.1) */
  #include <stdio.h>
  #include <ctype.h>
%}
%token DIGIT
%%
line    : expr '\n'          { printf("%d\n", $1); };
expr    : expr '+' term      { $$ = $1 + $3; }
        | term               { $$ = $1; };
term    : term '*' factor    { $$ = $1 * $3; }
        | factor             { $$ = $1; };
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. 12.13) */
  #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 12.14) correctly
- also adequate for strong following weak operator (`*` after `+`; Example 12.13) and for right-associative operators
- not appropriate for weak following strong operator and for left-associative binary operators
  ($\Longrightarrow$ reduce; see Example 12.13)

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:

```
%[left|right] Operators_1
        ⋮
%[left|right] Operators_n
```

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

Example 13.1

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

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

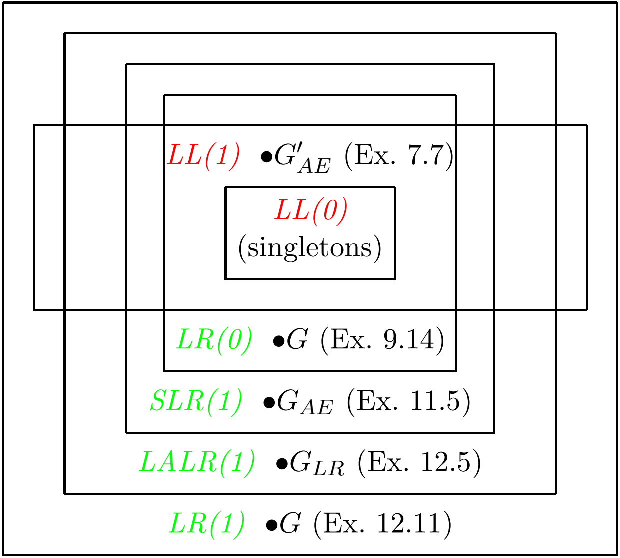
Precedences and Associativities in yacc II

```
%{/* 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;
}
```

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

Outline

Overview of Grammar Classes

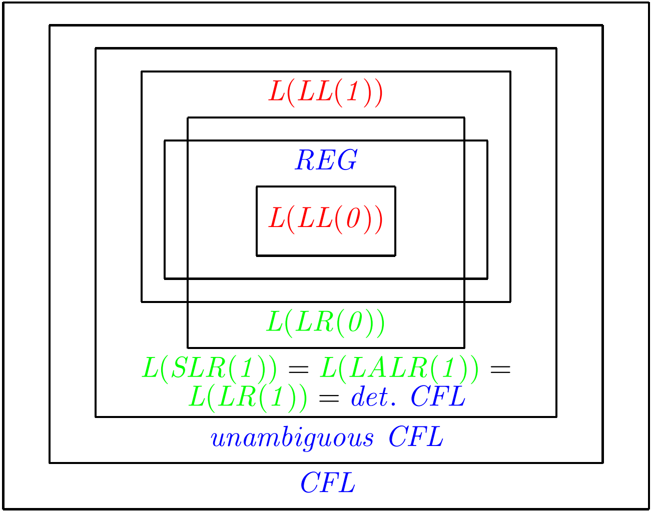


Moreover:

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

Outline

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

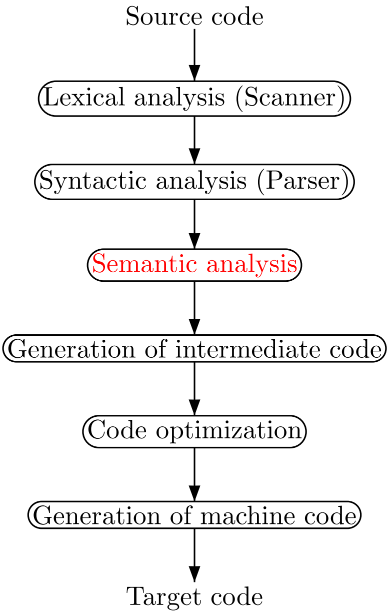
- "almost" $LL(1) \subseteq LALR(1)$ (only pathological counterexamples)
- LL requires elimination of left recursion and left factorization

Semantic actions : (see semantic analysis) LL wins

- actions can be placed anywhere in LL parsers without causing conflicts
- in LALR: implicit ε-productions

Outline

Conceptual Structure of a Compiler



Outline

Beyond Syntax

To generate (efficient) code, the compiler needs to answer many questions:

- Are there identifiers that are not declared? Declared but not used?
- Is `x` a scalar, an array, or a procedure? Of which type?
- Which declaration of `x` is used by each reference?
- Is `x` defined before it is used?
- Is the expression `3 * x + y` type consistent?
- Where should the value of `x` be stored (register/stack/heap)?
- Do `p` and `q` refer to the same memory location (aliasing)?
- ...

These cannot be expressed using context-free grammars!
(e.g., $\{ww \mid w \in \Sigma^*\} \notin CFL_\Sigma$)

Static Semantics

Static semantics refers to properties of program constructs

- which are true for every occurrence of this construct in every program execution (static) and
- can be decided at compile time
- but are context-sensitive and thus not expressible using context-free grammars (semantics).

Example properties:

Static: type or declaredness of an identifier, number of registers required to evaluate an expression, ...

Dynamic: value of an expression, size of runtime stack, ...

These properties are determined by

Scope rules: defines part of program where a declaration is valid

Visibility rules: defines part of scope where a declaration is visible (overlapping of global and local declarations)

Typing rules: defines type consistency of expressions, statements, ...

Outline

Attribute Grammars I

Goal: compute context-dependent but runtime-independent properties of a given program

Idea: enrich context-free grammar by semantic rules which annotate syntax tree with attribute values

$\Longrightarrow$ Semantic analysis = attribute evaluation

Result: attributed syntax tree

In greater detail:

- With every nonterminal a set of attributes is associated.
- Two types of attributes are distinguished:

  Synthesized: bottom-up computation (from the leafs to the root)

  Inherited: top-down computation (from the root to the leafs)
- With every production a set of semantic rules is associated.

Attribute Grammars II

Advantage: attribute grammars provide a very flexible and broadly applicable mechanism for transporting information throught the syntax tree ("syntax-directed translation")

- Attribute values: symbol tables, data types, code, error flags, ...
- Application in Compiler Construction:
  - static semantics
  - program analysis for optimization
  - code generation
  - error handling
- Automatic attribute evaluation by compiler generators (cf. `yacc`'s synthesized attributes)
- Originally designed by D. Knuth for defining the semantics of context-free languages (Math. Syst. Theory 2 (1968), pp. 127–145)

Example: Knuth's Binary Numbers I

Example 13.2 (only synthesized attributes)

Binary numbers (with fraction):

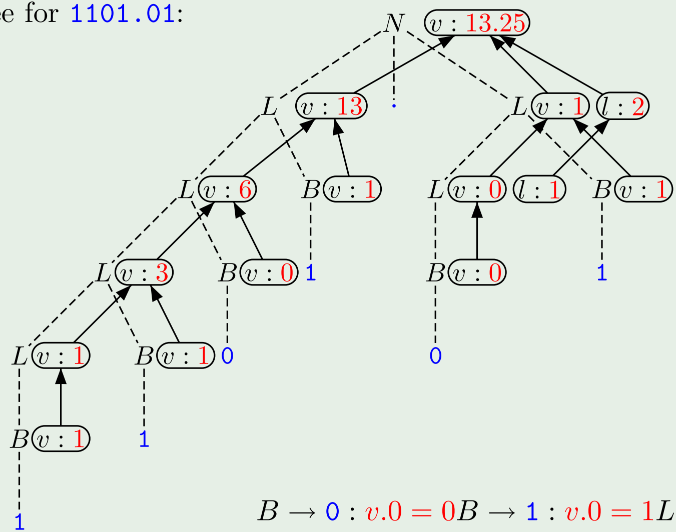
$$
\begin{array}{lllrcl}
G_B: & \text{Numbers} & N \to L & v.0 & = & v.1 \\
 & & N \to L\,.\,L & v.0 & = & v.1 + v.3/2^{l.3} \\
 & \text{Lists} & L \to B & v.0 & = & v.1 \\
 & & & l.0 & = & 1 \\
 & & L \to LB & v.0 & = & 2 * v.1 + v.2 \\
 & & & l.0 & = & l.1 + 1 \\
 & \text{Bits} & B \to \mathtt{0} & v.0 & = & 0 \\
 & \text{Bits} & B \to \mathtt{1} & v.0 & = & 1
\end{array}
$$

Synthesized attributes of N, L, B: v (value; domain: $V^v := \mathbb{Q}$)
of L: l (length; domain: $V^l := \mathbb{N}$)

Semantic rules: equations with attribute variables
(index = position of symbol; 0 = left-hand side)

Example 13.2 (continued)

Syntax tree for 1101.01:



$B \rightarrow 0 : v.0 = 0 B \rightarrow 1 : v.0 = 1 L \rightarrow B :$
$v.0 = v.1 L \rightarrow B : l.0 = 1 L \rightarrow LB : v.0 = 2 * v.1 + v.2 L \rightarrow LB : l.0 =$