CS301 W'99 Lecture Notes
Lecture 11

yacc Parser Generator

yacc = **y**et **a**nother **c**ompiler-**c**ompiler

```
grammar &        |--------|
action specs ->  |  yacc  |
                 |--------|
                     |
                     |
                     V
              |------------|
tokens -->    |  LALR(1)   |  --> parse trees,
              |  parser    |      intermediate
              |  yyparse() |      code, etc.
              |------------|
```

Grammar: BNF rules

Actions: C program fragments executed when reduction involving production is made.

General input format:
%{
C declarations
%}
yacc declarations (tokens, precedence declarations, etc.)
%%
rules & actions (BNF & C code)
%%
supporting C functions (e.g., `yylex()`, `yyerror()`*)*

yacc Example

```
%{                (begins C declarations)
#define YYSTYPE char*    (declares yylval type)
%}                (ends C declarations)
%start e          (declares start symbol)
%token ID         (declares token ID)
%%
e : e '+' t       { printf("reduce e:e+t\n"); }
  | t             { printf("reduce e:t\n"); }
  ;
t : t '*' f       { printf("reduce t:t*f\n"); }
  | f             { printf("reduce t:f\n"); }
  ;
f | '(' e ')'     { printf("reduce f:(e)\n"); }
  | ID            { printf("reduce f:ID %s\n",
                                yylval); }
  ;
%%
void main () { yyparse(); }

void yyerror(char *s) { printf("%s\n",s); }

int yylex() { .... }
```

– *Must use this name, whether generated from* `lex` *or hand-coded. Can return defined tokens or single character literals (e.g.,* `'+'`, `'*'`, `'('`, `')'`*). Must put any attribute value into* `yylval`.

yacc Mechanics

To generate the complete parser if lexer is included:

```
yacc foo.y ; cc -o foo y.tab.c
```

More complex lexical analyzers should go in a separate file, but `yylex` return values must correspond to `yacc`'s token code definitions.

Solution: processing `yacc` spec file with `-d` option generates `y.tab.h` file, which contains definitions for token codes (and for `YYSTYPE` and `yylval`).

Separately compiled lexical analyzer code can do

```
#include "y.tab.h"
```

and then use the token codes as return values, and write any attribute values into `yylval`. Overall sequence is:

```
yacc -d foo.y; cc -c y.tab.c;
cc -c lexer.c; cc -o foo y.tab.o lexer.o
```

More yacc flags:

To get detailed information on grammar errors, use the `-v` flag, which produces a file `y.output`.

To make `bison` generate its output files with the same name as `yacc` (`y.tab.c`), use the `-y` flag.

Expressing (E)BNF in yacc

BNF production $A \rightarrow \alpha |\ \beta |\ \ldots |\ \gamma$ is written:

```
A : α       { action for A → α }
  | β       { action for A → β }
  .
  .
  .
  | γ       { action for A → γ }
  ;
```

Constructing lists, e.g., `idlist` $\rightarrow$`ID`$\{$`,ID`$\}$:

- Left-recursion is most efficient:

```
idlist : ID
       | idlist ',' ID
       ;
```

- Right-recursion also works:

```
idlist : ID
       | ID ',' idlist
       ;
```

- Lists with 0 or more items are easy:

```
list :
     | list item
     ;
```


yacc Conflicts

Recall that ambiguous grammar can have shift-reduce and reduce-reduce conflicts, e.g., input `ID + ID + ID` with grammar

```
e : e '+' e
  | ID
  ;
```

When parser has seen `ID + ID`, it can either:

- **shift** next `+`, reaching `ID + ID + ID`, and then reduce rightmost `ID + ID`, producing final result `ID + (ID + ID)`; or

- **reduce** `ID + ID` to `ID` before reading next `+`, producing final result `(ID + ID) + ID`.

By default, `yacc` handle shift/reduce conflicts by **shifting**. This often gives the desired effect, so having shift/reduce conflicts in grammar is considered "ok."

`yacc` handles reduce-reduce conflicts by reducing with rule listed **first** in grammar. This is seldom what you want, so having reduce/reduce conflicts in grammar is considered "bad style."

To get non-default behavior you can give `yacc` explicit precedence and associativity info.

yacc Precedence & Associativity

Explicit precedence and associativity can be given for each token and/or each grammar rule.

For tokens, associativity is specified by `%left` or `%right` declarations (which replace `%token` declarations), and precedence is specified by the order of the declarations (highest precedence last). E.g.:

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

Precedence/associativity of **rules** is normally given by that of rightmost terminal:

```
e : e '+' e     rule has prec/assoc of '+'
  | e '*' e     rule has prec/assoc of '*'
  ;
```

On shift/reduce conflicts, `yacc` **shifts** if the input symbol has higher precedence than the reduction rule, **reduces** if symbol has lower precedence, and uses rule associativity to choose if precedences are equal.

So, with above declarations, can use ambiguous expression grammar directly:

```
e : e '+' e | e '-' e | e '*' e | e '/' e
  | e '^' e | '(' e ') | ID ;
```


yacc Unary operators

Sometimes want a single operator to have different associativity/precedence in different rules. E.g., want minus symbol (`'-'`) to have higher precedence when used as a unary operator than when used as a binary operator.

`yacc` allows you to set the precedence of a rule directly by adding a %prec qualifier to it. Unary minus is then handled by defining a "pseudo-terminal" for it, with appropriate precedence.

```
%token ID
%right '='
%left '+' '-'
%left '*' '/'
%left UNARYMINUS     (pseudo-token declaration)
%right '^'
%%
e : ID '=' e
  | e '+' e | e '-' e
  | '-' e  %prec UNARYMINUS
        (give rule prec/assoc of  UNARYMINUS
          rather than of '-')
  | e '*' e | e '/' e
  | e '^' e
  | '(' e ')'
  | ID
  ;
```

Syntax-directed Translation

Use **grammatical structure** of language to guide translation into lower-level form.

Traverse **parse tree** (constructed or virtual) evaluating **semantic rules**.

Semantic rules ("attribute equations"):

• Assign **values** to **attributes** attached to nodes of parser tree.

Examples: type or value of expression; code for statement block.

• Perform side-effects on global state.

Examples: make entries in symbol table; issue errors; generate code to output file.

Attributes are pieces of information (any kind!) attached to **nodes** of a grammar-induced tree.

Semantic rules are associated with grammar **productions**, because each tree node is "built" by a production. (Terminal nodes are assumed to have their attributes "at the beginning.")

Collectively, semantic rules make up an **attribute grammar**.

Attribute Evaluation

Attribute grammars can be used with parse tree (real or virtual) or abstract syntax tree.

Evaluation order of semantic rules may or may not follow **reduction** order during parser: depends on form of rules.

Computing attribute values is called **annotating** or **decorating** the tree.

If used with parse tree, often try to compute attribute values **while parsing**.

Sometimes, attributes are more important than parse tree itself.

Example: can use attribute grammar on **parse** trees to compute **AST** as an attribute!

More complicated attribute equations may require whole tree to exist first, before attribute evaluation begins.

An attribute is:

• **synthesized** if its value at a node depends only on values of attributes of **descendents** of that node; or

• **inherited** if its value at a node depends only on the values of attributes of **ancestors** and/or **siblings** of that node.

Synthesized attributes on Parse Trees

Attribute values at non-terminal node depend only on values at node's **children**. Values at terminal nodes are provided by lexical analyzer.

Example: desk calculator ("run-time" actions)

$S \rightarrow E$	`print (`E`.val)`
$E \rightarrow E_1$ `+` E_2	E`.val := `E_1`.val + `E_2`.val`
$E \rightarrow E_1$ `*` E_2	E`.val := `E_1`.val * `E_2`.val`
$E \rightarrow$ `(`E_1`)`	E`.val := `E_1`.val`
$E \rightarrow I$	E`.val := `I`.val`
$I \rightarrow I_1$ `digit`	I`.val := 10 * `I_1`.val + digit.val - '0'`
$I \rightarrow$ `digit`	I`.val := digit.val - '0'`

Attributes can be evaluated **bottom-up**.

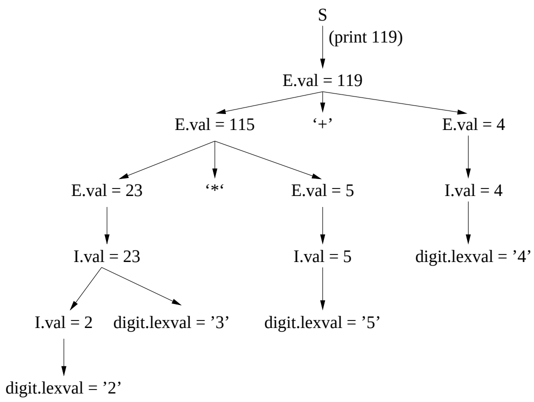
Evaluation can be done while parsing (either top-down or bottom-up).

When parsing bottom-up, at time of a reduction all attribute values on RHS are known, so LHS can be computed.

Syntax-directed definitions that use **only** synthesized attributes are called **S-attributed definitions**.

Example

"Decorated" parse tree for input `23*5+4`.



Note: Same parse tree and attribute evaluation pattern would hold for **static** attributes, such as expression type, code sequence, etc.

Semantic Stack Method

Implements **synthesized** attribute evaluation in **bottom-up** shift-reduce parser.

Semantic stack is manipulated in parallel with parser stack.

When a terminal is **shifted** onto parser stack, its attributes are pushed onto semantic stack.

Before a **reduction** $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_k$, the top k values on semantic stack are attributes for RHS.

After the reduction, top value is attribute for LHS non-terminal.

Example (Desk Calculator)

Parse Stack	Semantic Stack	Input	Production
		23*5+4$	
digit	'2'	3*5+4$	
I	2	3*5+4$	$I \rightarrow$ digit
I digit	2 '3'	*5+4$	
I	23	*5+4$	$I \rightarrow I$ digit
E	23	*5+4$	$E \rightarrow I$
E *	23 _	5+4$	$E \rightarrow I$
E * digit	23 _ '5'	+4$	
E * I	23 _ 5	+4$	$I \rightarrow$ digit
E * E	23 _ 5	+4$	$E \rightarrow I$
E	115	+4$	$E \rightarrow E*E$
E +	115 _	4$	
E + digit	115 _ '4'	$	
E + I	115 _ 4	$	$I \rightarrow$ digit
E + E	115 _ 4	$	$E \rightarrow I$
E	119	$	$E \rightarrow E+E$
S	_	$	$S \rightarrow E$

Note that parse stack and semantic stack always have equal depths.

yacc Calculator

```
%union {
  int val;
  char lexval;
}
%type <val> S E I
            (these non-terminals have val attributes)
%token <lexval> digit
            (this token has lexval attribute)
%%
S : E         { print $1; }
  ;
E : E '+' E   { $$ = $1 + $3; }
  : E '*' E   { $$ = $1 * $3; }
  : '(' E ')' { $$ = $2; }
  : I         { $$ = $1; }
  ;
I : I digit   { $$ = 10*$1 + ($2 - '0'); }
  | digit     { $$ = $1 - '0'; }
  ;
```


yacc Value (Semantic) Stack

yacc-generated parser automatically maintains a **value stack** in parallel with the parsing stack.

```
YYSTYPE yylval, yyval, yyv[150], *yyvsp;
```

On a **shift**: yylval is pushed onto yyv.

On a **reduce**: user **action** code is executed with the meta-variables $1,$2,... bound to the elements of the value stack in 1-to-1 correspondence with RHS of production.

Example from desk calculator:

```
e : e '+' e    { $$ = $1 + $3; }
```

Suppose this rule is reduced when the top of stack yyvsp = &yyv[2]:

```
            e           '+'       e
yyv      [0]=115      [1]=??    [2]=4
           $1           $2        $3
```

The **assignment** to $$ sets yyval, so above action is equivalent to yyval = *(yyvsp-2) + *yyvsp.

After reduction, the RHS attributes are popped and yyval is pushed:

```
             e
yyv      [0]=119
```

Multiple types of semantics values

By default, YYSTYPE is int, but this can be changed:

- to a different (single) type by by including a #define YYSTYPE in the C declarations section; or
- to a union of possible types by using a %union declaration.

For example, the declaration

```
%union {
  char *svalue;
  int ivalue;
}
```

induces the following definition in the generated parser:

```
typedef union {
  char *svalue;
  int ivalue;
} YYSTYPE;
```

Now the declarations of tokens and non-terminals that carry values must be marked to indicate which member of the union to use, e.g.

```
%token<svalue> ID
%token<ivalue> NUM
%type<ivalue> e
```

must declare value-carrying non-terminals

Multiple types of semantics values (cont.)

The lexical analyzer must take care to set the correct union member, e.g.,

```
  ...;
  yylval.svalue = strsave(yytext);
  return ID;
```

Generated parser code will automatically reference the correct union member, e.g.,

```
e : e '+' e    { $$ = $1 + $3; }
```

generates:

```
  yylval.ivalue =
    (yyvsp-2)->ivalue + yyvsp->ivalue;
```