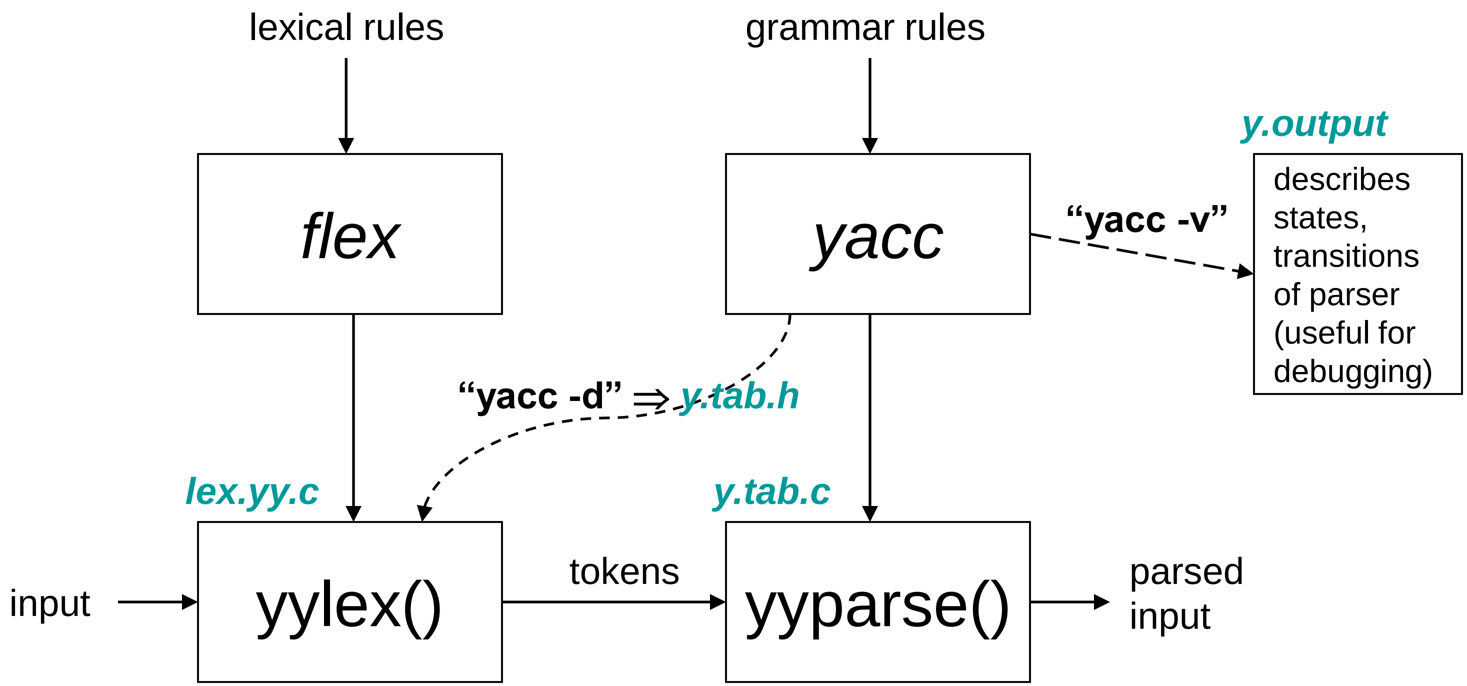
Introduction to Yacc

Introduction to Yacc

- Yacc: yet another compiler compiler.
- An LALR parser generator.
- Yacc generates
  - Tables – according to the grammar rules.
  - Driver routines – in C programming language.
  - y.output – a report file.

Using Yacc Together with Lex

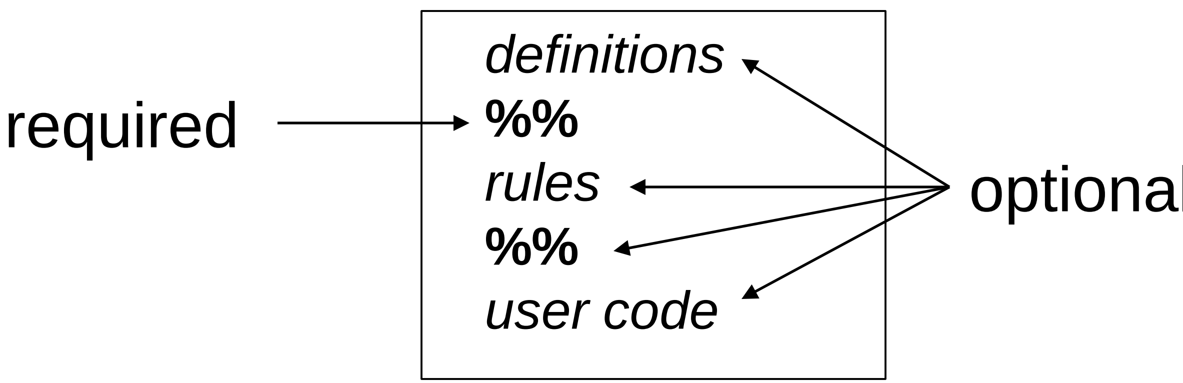


Cooperate with lex

- Invokes yylex() automatically.
- Generate *y.tab.h* file through the -d option.
- The lex input file must contains y.tab.h
- For each token that lex recognized, a number is returned (from *yylex()* function.)

Writing Yacc Input File

- A Yacc input file consists of three sections:
  - Definition
  - Rules
  - User code
- Separate by %%



Definition Section

- C source code, include files, etc.
- `%token`
- Yacc invokes *yylex()* when a token is required.
- All terminal symbols should be declared through `%token`.
- Yacc produces *y.tab.h* by `%token` definitions.

Definitions

- Information about tokens:
  - token names:
    - declared using ‘**%token**’
    - single-character tokens don’t have to be declared
    - any name not declared as a token is assumed to be a nonterminal.
  - start symbol of grammar, using ‘**%start**’   [optional]
  - operator info:
    - precedence, associativity
  - stuff to be copied verbatim into the output (e.g., declarations, **#include**s): enclosed in **%{ … }%**

Rules

Grammar production		yacc rule
$A \to B_1\ B_2\ \ldots\ B_m$ $A \to C_1\ C_2\ \ldots\ C_n$ $A \to D_1\ D_2\ \ldots\ D_k$	➡	$A \to B_1\ B_2\ \ldots\ B_m$ $\mid\ C_1\ C_2\ \ldots\ C_n$ $\mid\ D_1\ D_2\ \ldots\ D_k$; /* ';' optional, but advised */

- Rule RHS can have arbitrary C code embedded, within { … }. E.g.:

  A : B1 { printf("after B1\n"); x = 0; } B2 { x++; } B3

- Left-recursion more efficient than right-recursion:
  - A : A x | …   rather than   A : x A | …

Conflicts

- Conflicts arise when there is more than one way to proceed with parsing.
- Two types:
  - shift-reduce [default action: *shift*]
  - reduce-reduce [default: *reduce with the first rule listed*]
- Removing conflicts:
  - specify operator precedence, associativity;
  - restructure the grammar
    - use **y.output** to identify reasons for the conflict.

Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

  %left '+' '-'

  %left '*' '/'

  %right '^'

  Operators in the same group have the same precedence

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
      | '–' expr    %prec '*'
      | …
```

Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

    %left '+' '-'

    %left '*' '/'

    %right '^'

    Operators in the same group have the same precedence

    Across groups, precedence increases going down.

- Unary operators: **%prec**
    - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
        | '–' expr    %prec '*'
        | …
```

Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

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

Operators in the same group have the same precedence

Across groups, precedence increases going down.

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
      | '–' expr    %prec '*'
      | …
```

The rule for unary '–' has the same (high) precedence as '*'

Error Handling

- The "token" '**<u>error</u>**' is reserved for error handling:
  - can be used in rules;
  - suggests places where errors might be detected and recovery can occur.

<u>Example</u>:

```
stmt  : IF '(' expr ')' stmt
      | IF '(' error ')' stmt
      | FOR …
      | …
```

Intended to recover from errors in 'expr'

Parser Behavior on Errors

When an error occurs, the parser:

- pops its stack until it enters a state where the token ‘error’ is legal;
- then behaves as if it saw the token ‘error’
  - performs the action encountered;
  - resets the lookahead token to the token that caused the error.

- If no ‘error’ rules specified, processing halts.

Controlling Error Behavior

- Parser remains in error state until three tokens successfully read in and shifted
  - prevents cascaded error messages;
  - if an error is detected while parser in error state:
    - no error message given;
    - input token causing the error is deleted.
- To force the parser to believe that an error has been fully recovered from:

```
yyerrok;
```

- To clear the token that caused the error:

```
yyclearin;
```

Placing ‘error’ tokens

Some guidelines:

- Close to the start symbol of the grammar:
  - To allow recovery without discarding all input.
- Near terminal symbols:
  - To allow only a small amount of input to be discarded on an error.
  - Consider tokens like ‘**)**’, ‘**;**’, that follow nonterminals.
- Without introducing conflicts.

Error Messages

- On finding an error, the parser calls a function

  void yyerror(char *s)   /* s points to an error msg */

  – user-supplied, prints out error message.

- More informative error messages:
  – int yychar: token no. of token causing the error.
  – user program keeps track of line numbers, as well as any additional info desired.

Error Messages: example

```
#include "y.tab.h"
extern int yychar, curr_line;
static void print_tok()
{
   if (yychar < 255) {
     fprintf(stderr, "%c", yychar);
   }
   else {
      switch (yychar) {
      case ID: …
      case INTCON: …
      …
      }
   }
}
```

```
void yyerror(char *s)
{
   fprintf(stderr,
             "[line %d]: %s",
              curr_line,
              s);
   print_tok();
}
```

Debugging the Parser

To trace the shift/reduce actions of the parser:

- when compiling:
  - #define YYDEBUG
- at runtime:
  - set yydebug = 1   /* extern int yydebug; */

Adding Semantic Actions

- Semantic actions for a rule are placed in its body:
  - an action consists of C code enclosed in { … }
  - may be placed anywhere in rule RHS

Example:

expr : ID { symTbl_lookup(idname); }

decl : type_name { tval = … } id_list;

Synthesized Attributes

Each nonterminal can "return" a value:

- The return value for a nonterminal X is "returned" to a rule that has X in its body, e.g.:

  A : … X …

  value "returned" by X

  X : …

- *This is different from the values returned by the scanner to the parser!*

Attribute return values

- To access the value returned by the i^{th} symbol in a rule body, use $i
  - an action occurring in a rule body counts as a symbol. E.g.:

$$\text{decl : } \underbrace{\text{type}}_{1}\ \underbrace{\text{\{ tval = \$1 \}}}_{2}\ \underbrace{\text{id\_list}}_{3}\ \underbrace{\text{\{ symtbl\_install(\$3, tval); \}}}_{4}$$

- To set the value to be returned by a rule, assign to $$
  - by default, the value of a rule is the value of its first symbol, i.e., $1.

Example

/* A variable declaration is an identifier followed by an optional subscript, e.g., **x** or **x[10]** */

```
var_decl : ident opt_subscript
        { if ( symtbl_lookup($1) != NULL )
              ErrMsg("multiple declarations", $1);
          else {
              st_entry = symtbl_install($1);
              if ( $2 == ARRAY ) {
                  st_entry->base_type = ARRAY;
                  st_entry->element_type = tval;
              }
              else {
                  st_entry->base_type = tval;
                  st_entry->element_type = UNDEF;
              }
          }
        }
```

```
opt_subscript : '[' INTCON ']' { $$ = ARRAY; }
              |  /* null */    { $$ = INTEGER; }
              ;
```

Declaring Return Value Types

- Default type for nonterminal return values is **int**.
- Need to declare return value types if nonterminal return values can be of other types:
  - Declare the union of the different types that may be returned:

```
%union {
    symtbl_ptr      st_ptr;
    id_list_ptr     list_of_ids;
    tree_node_ptr   syntax_tree_ptr;
    int             value;
}
```

  - Specify which union member a particular grammar symbol will return:

```
%token <value> INTCON, CHARCON;          terminals
%type <st_ptr> identifier;
%type <syntax_tree_ptr> expr, stmt;      nonterminals
```

Conflicts

- A conflict occurs when the parser has multiple possible actions in some state for a given next token.
- Two kinds of conflicts:
  - *shift-reduce conflict*:
    - The parser can either keep reading more of the input (“shift action”), or it can mimic a derivation step using the input it has read already (“reduce action”).
  - *reduce-reduce conflict*:
    - There is more than one production that can be used for mimicking a derivation step at that point.

Example of a conflict

Grammar rules:

$S \rightarrow$ **if** (e) S /* 1 */
| **if** (e) S **else** S /* 2 */

Input: **if** (e_1) **if** (e_2) S_2 **else** S_3

Parser state when input token = 'else':

- Input already seen: **if** (e_1) **if** (e_2) S_2
- Choices for continuing:

1. keep reading input ("shift"):
   - '**else**' part of innermost if
   - eventual parse structure:

   if (e_1) { **if** (e_2) S_2 **else** S_3 }

2. mimic derivation step using $S \rightarrow$ **if** (e) S ("reduce"):
   - '**else**' part of outermost if
   - eventual parse structure:

   if (e_1) { **if** (e_2) S_2 } **else** S_3

shift-reduce conflict

Handling Conflicts

General approach:

- Iterate as necessary:
  1. Use “yacc -v” to generate the file **y.output**.
  2. Examine **y.output** to find parser states with conflicts.
  3. For each such state, examine the items to figure why the conflict is occurring.
  4. Transform the grammar to eliminate the conflict:

Reason for conflict	**Possible grammar transformation**
Ambiguity with operators in expressions	Specify associativity, precedence
Error action	Move or eliminate offending error action
Semantic action	Move the offending semantic action
Insufficient lookahead	“expand out” the nonterminal involved
Other	…???…

Resources

- Yacc Manual: http://www.cs.uaf.edu/~cs631/yacc-docs.txt
  http://www.gnu.org/software/bison/manual/
- Doug Brown, John Levine, and Tony Mason, “lex & yacc”, second edition, O'Reilly.
- Thomas Niemann, “A Compact Guide to Lex & Yacc”.
- Lex/Yacc Win32 port: http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html
- Parser Generator: www.bumblebeesoftware.com