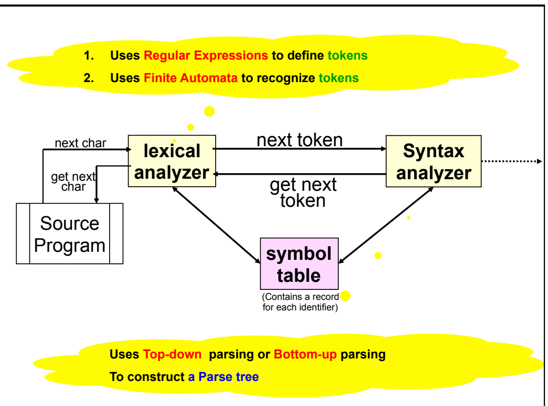
# Language Processing Systems

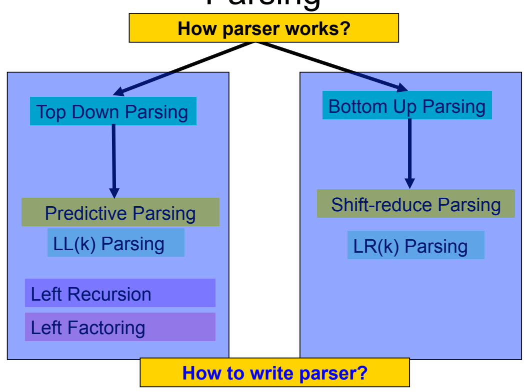
**Prof. Mohamed Hamada**

Software Engineering Lab.
The University of Aizu
Japan

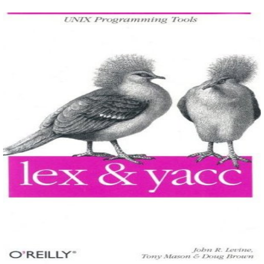
## Syntax Analysis (Parsing)



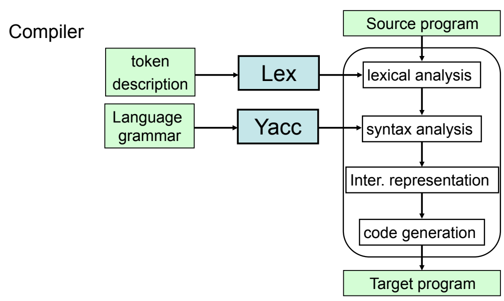

## Parsing




## Yacc




## Yacc



## How to write an LR parser?

General approach:
The construction is done automatically by a tool such as the *Unix* program ***yacc***.

Using the source program language grammar to write a simple yacc program and save it in a file named name.y

Using the unix program yacc to compile name.y resulting in a C (parser) program named y.tab.c

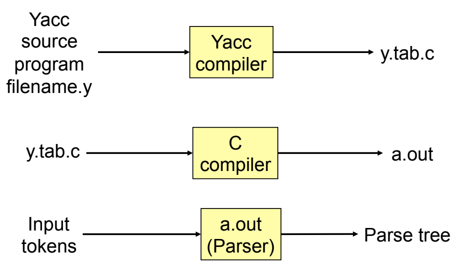
Compiling and linking the C program y.tab.c in a normal way resulting the required parser.

## LR parser generators

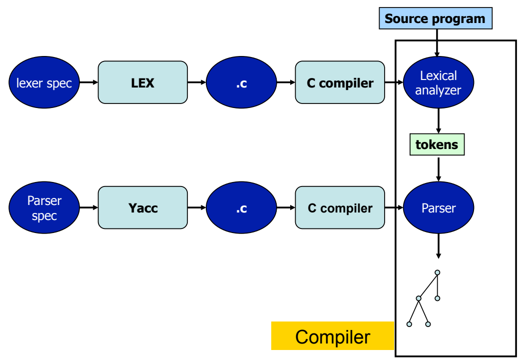
**Yacc: Yet another compiler compiler**

- Automatically generate LALR parsers
- Created by S.C. Johnson in 1970's

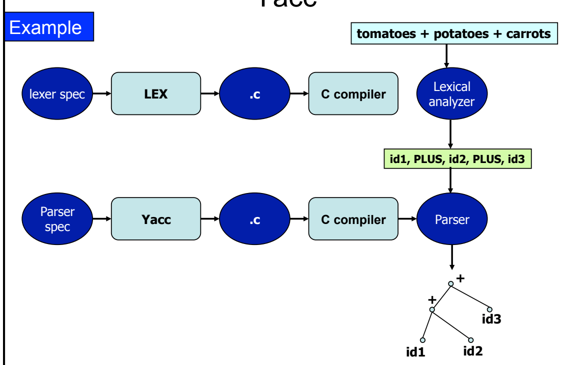
## Using Yacc



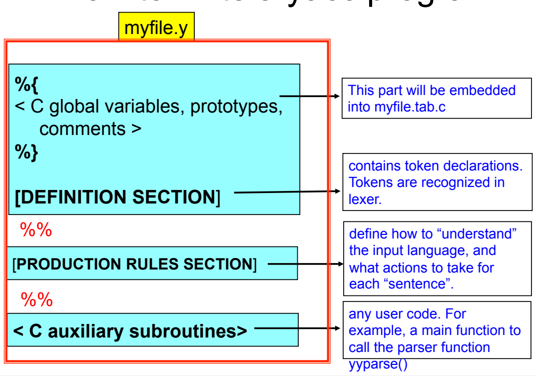

## Yacc




## Yacc




## How to write a yacc program



## Running Yacc programs

```
% yacc -d -v my_prog.y
% gcc -o y.tab.c -ly
```

**The `-d` option creates a file "`y.tab.h`", which contains a `#define` statement for each terminal declared. Place `#include "y.tab.h"` in between the `%{` and `%}` to use the tokens in the functions section.**

**The `-v` option creates a file "`y.output`", which contains useful information on debugging.**

**We can use Lex to create the lexical analyser. If so, we should also place `#include "y.tab.h"` in Lex's definitions section, and we must link the parser and lexer together with both libraries (`-ly` and `-ll`).**

## Running Yacc programs

- Yacc:
  - produce C file y.tab.c contains the C code to apply the grammar
  - y.tab.h contains the data structures to be used by lex to pass data to yacc

### DEFINITION SECTION

Any terminal symbols which will be used in the grammar must be declared in this section as a token. For example

```
%token VERB
%token NOUN
```

Non-terminals do not need to be pre-declared.

Anything enclosed between %{ ... %} in this section will be copied straight into y.tab.c (the C source for the parser).

All `#include` and `#define` statements, all variable declarations, all function declarations and any comments should be placed here.

### PRODUCTION RULES SECTION

**Grammar**

**A production rule:** nontermsym → symbol1 symbol2 … | symbol3 symbol4 … | ….

**Yacc**

```
nontermsym : symbol1 symbol2 ... { actions }
           | symbol3 symbol4 ... { actions }
           | ...
           ;
```

Alternatives

**Example:** a productionrule: expr → expr + expr

```
expr : expr '+' expr    { $$ = $1 + $3 }
```

Value of non-terminal on lhs

Value of n-th symbol on rhs

### PRODUCTION RULES SECTION

input file

```
%token DIGIT

%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
```

grammar — Semantics action

Yacc maintains a stack of "values" that may be referenced ($i) in the semantic actions

### PRODUCTION RULES SECTION

## Semantic Actions in Yacc

- Semantic actions are embedded in RHS of rules.

  An action consists of one or more C statements, enclosed in braces { … }.

- *Examples*:

  ident_decl : ID { symtbl_install( id_name ); }

  type_decl : type { tval = … } id_list;

PRODUCTION RULES SECTION

## Semantic Actions in Yacc

Each nonterminal can return a value.

- The value returned by the $i^{th}$ symbol on the RHS is denoted by **$i**.
- An action that occurs in the middle of a rule counts as a "symbol" for this.
- To set the value to be returned by a rule, assign to **$$**.

*By default, the value returned by a rule is the value of the first RHS symbol, i.e., **$1**.*

PRODUCTION RULES SECTION

**Example:**

```
statement → expression
expression → expression + expression |  expression - expression
           | expression * expression |  expression / expression
           | NUMBER
```

```
statement : expression  { printf (" = %g\n", $1); }
expression : expression '+' expression { $$ = $1 + $3; }
        | expression '-' expression { $$ = $1 - $3; }
        | expression '*' expression { $$ = $1 * $3; }
        | expression '/' expression  { $$ = $1 / $3 ; }
        | NUMBER  { $$ = $1; }
        ;
```

C auxiliary subroutines

**This section contains the user-defined main() routine, plus any other required functions. It is usual to include:**

**`lexerr()` - to be called if the lexical analyser finds an undefined token. The default case in the lexical analyser must therefore call this function.**

**`yyerror(char*)` - to be called if the parser cannot recognise the syntax of part of the input. The parser will pass a string describing the type of error.**

**The line number of the input when the error occurs is held in `yylineno`.**

**The last token read is held in `yytext`.**

C auxiliary subroutines

Yacc interface to lexical analyzer

- Yacc invokes yylex() to get the next token
- the "value" of a token must be stored in the global variable yylval
- the default value type is int, but can be changed

**Example**

```
%%
yylex()
{
        int c;

        c = getchar();

        if (isdigit(c)) {
                yylval = c - '0';
                return DIGIT;
        }
        return c;
}
```

C auxiliary subroutines

Yacc interface to back-end

- Yacc generates a function named yyparse()
- syntax errors are reported by invoking a callback function yyerror()

**Example**

```
%%
yylex()
{
    ...
}
main()
{
    yyparse();
}

yyerror()
{
    printf("syntax error\n");
    exit(1);
}
```

## Yacc Errors

**Yacc can not accept ambiguous grammars, nor can it accept grammars requiring two or more symbols of lookahead.**

**The two most common error messages are:**

```
shift-reduce conflict
reduce-reduce conflict
```

**The first case is where the parser would have a choice as to whether it shifts the next symbol from the input, or reduces the current symbols on the top of the stack.**

**The second case is where the parser has a choice of rules to reduce the stack.**

## Yacc Errors

Do not let errors go uncorrected. A parser will be generated, but it may produce unexpected results.

Study the file "y.output" to find out when the errors occur.

The SUN C compiler and the Berkeley PASCAL compiler are both written in Yacc.

You should be able to change your grammar rules to get an unambiguous grammar.

## Yacc Errors

Example 1

Yacc

```
Expr  :  INT_T
      |  Expr + Expr
      ;
```

Causes a shift-reduce error, because

`INT_T + INT_T + INT_T`

can be parsed in two ways.

Yacc

```
Animal  :  Dog
        |  Cat
        ;

Dog     :  FRED_T;

Cat     :  FRED_T;
```

Causes a reduce-reduce error, because

`FRED_T`

can be parsed in two ways.

## Yacc Errors

Example 2

1. input file (desk0.y)

Yacc

```
%token DIGIT
%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
yylex()
{
    int c;

    c = getchar();

    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

2. run yacc

```
> yacc -v desk0.y
```

Conflicts: 4 shift/reduce

## Conflict resolution in Yacc

Correcting errors

- **shift-reduce**: prefer shift
- **reduce-reduce**: prefer the rule that comes first

## Conflict resolution in Yacc

Correcting errors

- shift-reduce: prefer shift
- reduce-reduce: prefer the rule that comes first

```
>cat y.output
State 11 conflicts: 2 shift/reduce
State 12 conflicts: 2 shift/reduce .

Grammar

  0 $accept: line $end

  1 line: expr '\n'

  2 expr: expr '+' expr
  3     | expr '*' expr
  4     | '(' expr ')'
  5     | DIGIT
```

## Conflict resolution in Yacc

Correcting errors

- shift-reduce: prefer shift
- reduce-reduce: prefer the rule that comes first

```
state 11

  2 expr: expr . '+' expr
  2     | expr '+' expr .
  3     | expr . '*' expr

  '+'  shift, and go to state 8
  '*'  shift, and go to state 9

  '+'    [reduce using rule 2 (expr)]
  '*'    [reduce using rule 2 (expr)]
  $default  reduce using rule 2 (expr)
```

```
state 12

  2 expr: expr . '+' expr
  3     | expr . '*' expr
  3     | expr '*' expr .

  '+'  shift, and go to state 8
  '*'  shift, and go to state 9

  '+'    [reduce using rule 3 (expr)]
  '*'    [reduce using rule 3 (expr)]
  $default  reduce using rule 3 (expr)
```

## Conflict resolution in Yacc

**Define operator's precedence and associativity**
resolve shift/reduce conflict in Example 2

Definition section

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

Specify the associativity

Higher precedence operators are defined later

---

**Example 2** Correct

**Operator precedence in Yacc**

priority from top (low) to bottom (high)

```
> yacc -v desk0.y
```

```
> gcc -o desk0 y.tab.c
```

```
%token DIGIT
%left '+'
%left '*'
%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
yylex()
{
    int c;

    c = getchar();

    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    }
    return c;
}
```

---

## Exercise

multiple lines:

```
%%
lines: line
     | lines line
     ;
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
```

Extend the interpreter to a desk calculator with registers named a – z. Example input: v=3*(w+4)

---

## Answer

```
%{
int reg[26];
%}
%token DIGIT
%token REG
%right '='
%left '+'
%left '*'
%%
expr : REG '=' expr       { $$ = reg[$1] = $3;}
     | expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | REG                { $$ = reg[$1];}
     | DIGIT
     ;
%%
```

---

## Answer

```
%%

yylex()
{   int c = getchar();

    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    } else if ('a' <= c && c <= 'z') {
        yylval = c - 'a';
        return REG;
    }
    return c;
}
```

---

## Example Yacc Script

A case study 1

**S → NP VP**
**NP → Det NP1 | PN**
**NP1 → Adj NP1| N**
**Det → a | the**
**PN → peter | paul | mary**
**Adj → large | grey**
**N → dog | cat | male | female**
**VP → V NP**
**V → is | likes | hates**

**We want to write a Yacc script which will handle files with multiple sentences from this grammar. Each sentence will be delimited by a ".".**

**Change the first production to**
**S → NP VP .**

**and add**

**D → S D | S**

## The Lex Script

```
%{
/* simple part of speech lexer */

#include "y.tab.h"
%}

L [a-zA-Z]

%%

[ \t\n]+              /* ignore space */
is|likes|hates        return VERB_T;
a|the                 return DET_T;
dog |
cat |
male |
female                return NOUN_T;
peter|paul|mary       return PROPER_T;
large|grey            return ADJ_T;
\.                    return PERIOD_T;
{L}+                  lexerr();
.                     lexerr();

%%
```

## Yacc Definitions

```
%{
/* simple natural language grammar */

#include <stdio.h>
#include "y.tab.h"

extern in yyleng;
extern char yytext[];
extern int yylineno;
extern int yyval;

extern int yyparse();
%}

%token DET_T
%token NOUN_T
%token PROPER_T
%token VERB_T
%token ADJ_T
%token PERIOD_T

%%
```

## Yacc rules

```
/* a document is a sentence followed
   by a document, or is empty */

Doc   :   Sent Doc
        |   /* empty */
        ;

Sent  :   NounPhrase VerbPhrase PERIOD_T
        ;

NounPhrase  :  DET_T NounPhraseUn
                 |  PROPER_T
                 ;

NounPhraseUn  :  ADJ_T NounPhraseUn
                     |  NOUN_T
                     ;

VerbPhrase  :  VERB_T NounPhrase
                ;

%%
```

## User-defined functions

```
void lexerr()
{
   printf("Invalid input '%s' at line%i\n",
             yytext,yylineno);
   exit(1);
}

void yyerror(s)
char *s;
{
   (void)fprintf(stderr,
      "%s at line %i, last token: %s\n",
             s, yylineno, yytext);
}

void main()
{
   if (yyparse() == 0)
      printf("Parse OK\n");
   else printf("Parse Failed\n");
}
```

## Running the example

```
% yacc -d -v parser.y
% cc -c y.tab.c
% lex parser.l
% cc -c lex.yy.c
% cc y.tab.o lex.yy.o -o parser -ly -ll
```

```
peter is a large grey cat.
the dog is a female.
paul is peter.
```

file1

```
the cat is mary.
a dogcat is a male.
```

file2

```
peter is male.
mary is a female.
```

file3

```
% parser < file1
Parse OK
% parser < file2
Invalid input 'dogcat' at line 2
% parser < file3
syntax error at line 1, last token: male
```

## A case study 2 – The Calculator

zcalc.l

```
%{
#include "zcalc.tab.h"
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
                 { yylval.dval = atof(yytext);
                   return NUMBER; }
[ \t]            ;
[a-zA-Z][a-zA-Z0-(]*
                 { struct symtab *sp = symlook(yytext);
                   yylval.symp = sp;
                   return NAME;
                 }

%%
```

zcalc.y

```
%{
#include "zcalc.h"
%}
%union { double dval; struct symtab *symp; }
%token <symp> NAME
%token <dval> NUMBER
%left '+' '-'
%type <dval> expression
%%
statement_list :  statement '\n' | statement_list statement '\n'
statement : NAME '=' expression  {$1->value = $3;}
          | expression { printf (" = %g\n", $1); }

expression : expression '+' expression { $$ = $1 + $3; }
           | expression '-' expression { $$ = $1 - $3; }
           | NUMBER  { $$ = $1; }
           | NAME { $$ = $1->value; }
%%
struct symtab * symlook( char *s )
{ /* this function looks up the symbol table and check whether
the symbol s is already there. If not, add s into symbol table. */
}
int main() {
   yyparse();
   return 0;
}
```