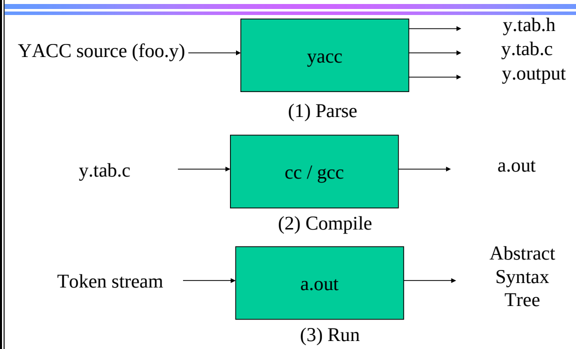
Introduction

- What is **YACC** ?
  - **Tool which will produce a parser for a given grammar**.
  - YACC (Yet Another Compiler Compiler) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntactic analyzer of a language produced by this grammar.

History

- Yacc original written by *Stephen C. Johnson*, 1975.
- Variants:
  - lex, yacc (AT&T)
  - bison: a yacc replacement (GNU)
  - flex: *f*ast *lex*ical analyzer (GNU)
  - BSD yacc
  - PCLEX, PCYACC (Abraxas Software)

How YACC Works



An YACC File Example

```
%{
#include <stdio.h>
%}

%token NAME NUMBER
%%

statement: NAME '=' expression
        | expression            { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
        |   expression '-' NUMBER { $$ = $1 - $3; }
        |   NUMBER                { $$ = $1; }
        ;
%%
int yyerror(char *s)
{
   fprintf(stderr, "%s\n", s);
   return 0;
}

int main(void)
{
   yyparse();
   return 0;
}
```

YACC File Format

%{
C declarations
%}
yacc declarations
%%
Grammar rules
%%
Additional C code

– **Comments in /* ... */ may appear in any of the sections.**

Definitions Section

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

It is a terminal

start from expr

Start Symbol

- The first non-terminal specified in the grammar specification section.
- To overwrite it with `%start` declaraction.

```
%start  non-terminal
```

Rules Section

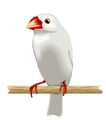
- Is a grammar
- Example

```
expr : expr '+' term | term;
term : term '*' factor | factor;
factor : '(' expr ')' | ID | NUM;
```

Rules Section

- Normally written like this
- Example:

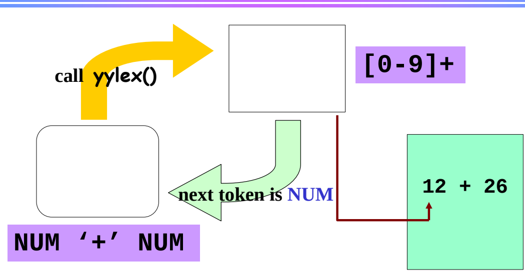
```
expr   : expr '+' term
       | term
       ;
term   : term '*' factor
       | factor
       ;
factor : '(' expr ')'
       | ID
       | NUM
       ;
```



The Position of Rules

```
expr : expr '+' term     { $$ = $1 + $3; }
     | term              { $$ = $1; }
     ;
term : term '*' factor   { $$ = $1 * $3; }
     | factor            { $$ = $1; }
     ;
factor : '(' expr ')'    { $$ = $2; }
       | ID
       | NUM
       ;
```

Works with LEX

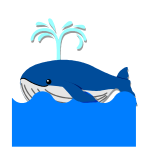


LEX and YACC need a way to identify tokens

Communication between LEX and YACC

- Use enumeration / define
  - YACC creates y.tab.h
  - LEX includes y.tab.h

yacc -d gram.y
Will produce:
y.tab.h



Communication between LEX and YACC

scanner.l

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
id        [_a-zA-Z][_a-zA-Z0-9]*
%%
int       { return INT; }
char      { return CHAR; }
float     { return FLOAT; }
{id}      { return ID;}
```

parser.y

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token  CHAR, FLOAT, ID, INT
%%
```

```
yacc -d xxx.y
   produces
y.tab.h

# define CHAR 258
# define FLOAT 259
# define ID 260
# define INT 261
```

YACC

- Rules may be recursive
- Rules may be ambiguous*
- Uses bottom up Shift/Reduce parsing
  - Get a token
  - Push onto stack
  - Can it reduced ?
    - yes: Reduce using a rule
    - no: Get another token
- Yacc **cannot** look ahead more than one token

Passing value of token

- Every terminal-token (symbol) may represent a value or data type
  - May be a numeric quantity in case of a number (42)
  - May be a pointer to a string ("Hello, World!")
- When using lex, we put the value into **yylval**
  - In complex situations **yylval** is a union
- Typical lex code:

```
  [0-9]+     {yylval = atoi(yytext); return
NUM}
```

Passing value of token

- Yacc allows symbols to have multiple types of value symbols

```
%union {
       double dval;
       int    vblno;
       char*  strval;
}
```

Passing value of token

```
%union {
       double  dval;
       int     vblno;
       char*   strval;
}
```

yacc -d

```
y.tab.h
…
extern YYSTYPE yylval;
```

```
[0-9]+     { yylval.vblno = atoi(yytext);
             return NUM;}
[A-z]+     { yylval.strval = strdup(yytext);
             return STRING;}
```

Lex file
include "y.tab.h"

Yacc Example

- Taken from Lex & Yacc
- Example: Simple calculator

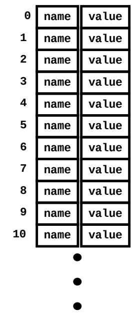
```
a = 4 + 6
a
a=10
b = 7
c = a + b
c
c = 17
$
```

Grammar

```
expression ::= expression '+' term |
               expression '-' term |
               term

term       ::= term '*' factor |
               term '/' factor |
               factor

factor     ::= '(' expression ')' |
               '-' factor  |
               NUMBER |
               NAME
```

Symbol Table

```
#define NSYMS 20  /* maximum number
                     of symbols */

struct symtab {
  char *name;
  double value;
} symtab[NSYMS];

struct symtab *symlook();
```



parser.h

Parser

```
%{
#include "parser.h"
#include <string.h>
%}

%union {
   double dval;
   struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER

%type <dval> expression
%type <dval> term
%type <dval> factor
%%
```

Terminal **NAME** and <**symp**>have the same data type.

Nonterminal **expression** and <**dval**> have the same data type.

parser.y

Parser (cont'd)

```
statement_list:     statement '\n'
               |    statement_list statement '\n'
               ;

statement:    NAME '=' expression { $1->value = $3; }
         |    expression          { printf("= %g\n", $1); }
         ;

expression: expression '+' term { $$ = $1 + $3; }
          | expression '-' term { $$ = $1 - $3; }
          | term
          ;
```

parser.y

Parser (cont'd)

```
term:       term '*' factor { $$ = $1 * $3; }
    |       term '/' factor { if ($3 == 0.0)
                                yyerror("divide by zero");
                              else
                                $$ = $1 / $3;
                            }
    | factor
    ;

factor: '(' expression ')' { $$ = $2; }
      | '-' factor         { $$ = -$2; }
      | NUMBER             { $$ = $1; }
      | NAME               { $$ = $1->value; }
      ;
%%
```

parser.y

Scanner

```
%{
#include "y.tab.h"
#include "parser.h"
#include <math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
    yylval.dval = atof(yytext);
  return NUMBER;
  }

[ \t] ;            /* ignore white space */
```

scanner.l

Scanner (cont'd)

```
[A-Za-z][A-Za-z0-9]*  { /* return symbol pointer */
                         yylval.symp = symlook(yytext);
                         return NAME;
                      }

"$"   { return 0; /* end of input */ }

\n|"="|"+"|"-"|"*"|"/"  return yytext[0];
%%
```

scanner.l

Precedence / Association

(1) 1 - 2 - 3

(2) 1 - 2 * 3

1. 1-2-3 = (1-2)-3? **or** 1-(2-3)?
2. 1-2*3 = 1-(2*3) **or** (1-2)*3?

Yacc: Shift/Reduce conflicts. Default is to **shift**.

Precedence / Association

```
%right '='
%left  '<' '>' NE LE GE
%left  '+' '-'
%left  '*' '/'
```

highest precedence

Precedence / Association

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

```
expr  : expr '+' expr   { $$ = $1 + $3; }
      | expr '-' expr   { $$ = $1 - $3; }
      | expr '*' expr   { $$ = $1 * $3; }
      | expr '/' expr
                        {
                              if($3==0)
                                    yyerror("divide 0");
                              else
                                    $$ = $1 / $3;
                        }
      | '-' expr %prec UMINUS {$$ = -$2; }
```

IF-ELSE Ambiguity

- Consider following rule:

```
stmt  : IF expr stmt
      | IF expr stmt ELSE stmt
      ……
```

Following state ?

```
IF expr IF expr stmt ELSE stmt
```

IF-ELSE Ambiguity

- It is a shift/reduce conflict.
- Yacc will always choose to shift.
- A solution:

```
stmt    : matched
        | unmatched
        ;
matched: other_stmt
        | IF expr THEN matched ELSE matched
        ;
unmatched: IF expr THEN stmt
          | IF expr THEN matched ELSE unmatched
          ;
```

Shift/Reduce Conflicts

- **shift/reduce conflict**
  - occurs when a grammar is written in such a way that a decision between shifting and reducing can not be made.
  - ex: IF-ELSE ambigious.
- To resolve this conflict, yacc will choose to shift.

Reduce/Reduce Conflicts

- ***Reduce/Reduce Conflicts:***

```
start : expr | stmt
      ;
expr : CONSTANT;
stmt : CONSTANT;
```

- Yacc resolves the conflict by reducing using the rule that occurs earlier in the grammar. **NOT GOOD!!**
- So, modify grammar to eliminate them.

Error Messages

- Bad error message:
  - Syntax error.
  - Compiler needs to give programmer a good advice.
- It is better to track the line number in lex:

```
void yyerror(char *s)
{
   fprintf(stderr, "line %d: %s\n:", yylineno, s);
}
```

Debug Your Parser

1. Use –t option **or** define YYDEBUG to 1.
2. Set variable *yydebug* to 1 when you want to trace parsing status.
3. If you want to trace the semantic values
   - Define your **YYPRINT** function

Shift and Reducing: Example

```
stmt: stmt ';' stmt
    | NAME '=' exp

exp: exp '+' exp
   | exp '-' exp
   | NAME
   | NUMBER
```

```
stack:
<empty>
```

```
input:
a = 7; b = 3 + a + 2
```

Recursive Grammar

- Left recursion

```
list:
     item
    | list ',' item
    ;
```

- Right recursion

```
list:
     item
    | item ',' list
    ;
```

- LR parser (e.g. yacc) prefers left recursion.
- LL parser prefers right recursion.

YACC Declaration Summary

`%start'
Specify the grammar's start symbol

`%union'
Declare the collection of data types that semantic values may have

`%token'
Declare a terminal symbol (token type name) with no precedence or associativity specified

`%type'
Declare the type of semantic values for a nonterminal symbol

YACC Declaration Summary

`%right'
Declare a terminal symbol (token type name) that is right-associative

`%left'
Declare a terminal symbol (token type name) that is left-associative

`%nonassoc'
Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error,
ex: x *op*. y *op*. z is syntax error)