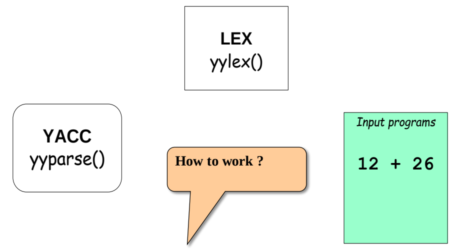
Yacc

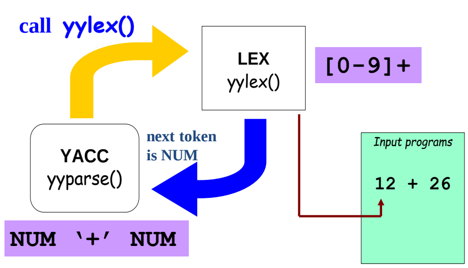
Yet Another Compiler Compiler

Some material adapted from slides by Andy D. Pimentel

LEX and YACC work as a team



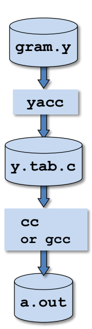
LEX and YACC work as a team



Availability

- lex, yacc on most UNIX systems
- bison: a yacc replacement from GNU
- flex: *f*ast *lex*ical analyzer
- BSD yacc
- Windows/MS-DOS versions exist

YACC's Basic Operational Sequence



File containing desired grammar in YACC format

YACC program

C source program created by YACC

C compiler

Executable program that will parse grammar given in gram.y

YACC File Format

```
Definitions

%%

Rules

%%

Supplementary Code
```

The identical LEX format was taken from this...

Rules Section

A context free grammar, e.g.:

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

Definitions section example

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

This is called a terminal

The start symbol (non-terminal)

Some details

- LEX produces a function called yylex()
- YACC produces a function called yyparse()
- yyparse() expects to be able to call yylex()
- How to get yylex()?
- Write your own!
- If you don't want to write your own: use lex!

If you wanted to write your own...

```
int yylex()
{
  if(it's a num)
    return NUM;
  else if(it's an id)
    return ID;
  else if(parsing is done)
    return 0;
  else if(it's an error)
    return -1;
}
```

Semantic actions

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

Semantic actions

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

Semantic actions (cont'd)

$1

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

Semantic actions (cont'd)

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

$2

Semantic actions (cont'd)

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

$3

Default: $$ = $1;

Precedence / Association

```
expr: expr '-' expr
    | expr '*' expr
    | expr '<' expr
    | '(' expr ')'
    ...
    ;
```

(1) 1 – 2 - 3

(2) 1 – 2 * 3

1. 1-2-3 = **(1-2)-3**? **or** 1-(2-3)?
   Define '-' operator is left-association.
2. 1-2*3 = 1-(2*3)
   Define "*" operator is precedent to "-" operator

Precedence / Association

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

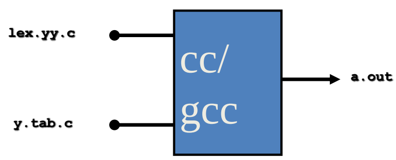
```
expr : expr '+' expr { $$ = $1 + $3; }
   | expr '-' expr { $$ = $1 - $3; }
   | expr '*' expr { $$ = $1 * $3; }
   | expr '/' expr     { if($3==0)
            yyerror("divide 0");
            else
            $$ = $1 / $3;
          }
   | '-' expr %prec UMINUS {$$ = -$2; }
```

Precedence / Association

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

highest precedence

Getting YACC & LEX to work together



Building Example

- Suppose you have a lex file called **`scanner.l`** and a yacc file called **`decl.y`** and want **`parser`**
- Steps to build...

```
yacc -d decl.y
lex scanner.l
gcc -c lex.yy.c y.tab.c
gcc -o parser lex.yy.o y.tab.o -ll
```

Note: scanner should include in the definitions section: **`#include "y.tab.h"`**

YACC

- Rules may be recursive
- Rules may be ambiguous
- Uses bottom-up Shift/Reduce parsing
  - Get a token
  - Push onto stack
  - Can it be reduced (How do we know?)
    - If yes: Reduce using a rule
    - If no: Get another token
- YACC can't look ahead > 1 token

Shift and reducing

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

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

stack:
<empty>

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

Shift and reducing

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

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

SHIFT!

stack:
NAME

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

Shift and reducing

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

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

SHIFT!

stack:
NAME '='

input:
7; b = 3 + a + 2

Shift and reducing

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

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

SHIFT!

stack:
NAME '=' 7

input:
; b = 3 + a + 2

Shift and reducing

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

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

REDUCE!

stack:
NAME '=' exp

input:
; b = 3 + a + 2

Shift and reducing

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

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

REDUCE!

stack:
stmt

input:
; b = 3 + a + 2

Shift and reducing

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

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

SHIFT!

stack:
stmt ';'

input:
b = 3 + a + 2

Shift and reducing

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

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

SHIFT!

stack:
stmt ';' NAME

input:
= 3 + a + 2

Shift and reducing

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

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

SHIFT!

stack:
stmt ';' NAME '='

input:
3 + a + 2

Shift and reducing

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

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

SHIFT!

stack:
stmt ';' NAME '=' NUMBER

input:
+ a + 2

Shift and reducing

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

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

REDUCE!

stack:
stmt ';' NAME '=' exp

input:
+ a + 2

Shift and reducing

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

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

SHIFT!

stack:
stmt ';' NAME '=' exp '+'

input:
a + 2

Shift and reducing

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

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

SHIFT!

stack:
stmt ';' NAME '=' exp '+' NAME

input:
+ 2

Shift and reducing

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

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

REDUCE!

stack:
stmt ';' NAME '=' exp '+' exp

input:
+ 2

Shift and reducing

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

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

REDUCE!

stack:
stmt ';' NAME '=' exp

input:
+ 2

Shift and reducing

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

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

SHIFT!

```
stack:
stmt ';' NAME '=' exp '+'
```

```
input:
2
```

Shift and reducing

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

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

SHIFT!

```
stack:
stmt ';' NAME '=' exp '+'
NUMBER
```

```
input:
<empty>
```

Shift and reducing

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

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

REDUCE!

```
stack:
stmt ';' NAME '=' exp '+'
exp
```

```
input:
<empty>
```

Shift and reducing

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

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

REDUCE!

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

```
input:
<empty>
```

Shift and reducing

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

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

REDUCE!

```
stack:
stmt ';' stmt
```

```
input:
<empty>
```

Shift and reducing

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

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

REDUCE!

```
stack:
stmt
```

```
input:
<empty>
```

Shift and reducing

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

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

DONE!

```
stack:
stmt
```

```
input:
<empty>
```

IF-ELSE Ambiguity

- Consider following rule:

```
stmt:
    IF expr stmt
  | IF expr stmt ELSE stmt
  ...
```

Following state : IF expr IF expr stmt . ELSE stmt

- Two possible derivations:

IF expr IF expr stmt . ELSE stmt IF expr IF expr stmt ELSE . stmt IF expr IF expr stmt ELSE stmt . IF expr stmt	IF expr IF expr stmt . ELSE stmt IF expr stmt . ELSE stmt IF expr stmt ELSE . stmt IF expr stmt ELSE stmt .

IF-ELSE Ambiguity

- It is a shift/reduce conflict
- YACC will always do shift first
- Solution 1 : re-write grammar

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

IF-ELSE Ambiguity

- Solution 2:

```
%nonassoc IFX
%nonassoc ELSE

stmt:
    IF expr stmt %prec IFX
  | IF expr stmt ELSE stmt
```

the rule has the same precedence as token IFX

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.
  - e.g.: IF-ELSE ambiguity
- To resolve conflict, YACC will choose to shift

Reduce/Reduce Conflicts

- Reduce/Reduce Conflicts:

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

- YACC (Bison) resolves conflict by reducing using rule that is earlier in grammar
- Not good practice to rely on this
- So, modify grammar to eliminate them

Use left recursion in yacc

- Left recursion

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

- Right recursion

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

- LR parser prefers left recursion
- LL parser prefers right recursion
- Yacc is a LR parser: use left recursion