Parser builder

- Yacc is *Yet Another Compiler-Compiler*
- **Input** is **context free grammar**
  - Have C actions in grammar
    1. print
    2. the compiler changes something in a data structure
- **Output** is a parser in C
  - File is `y.tab.c` using Yacc
  - Different if you use bison (if you don't know, don't worry) — `filename.tab.c` where `filename.y` is the source name

Parser

- Accepts a grammar
- LALR(1)
  - **L**ook**A**head **1** token- "**L**eft-to-right scanning" **R**ightmost derivation in reverse
  - Works on most programming languages
  - Fast, $O(n)$ where n is the length of the input, and saves space (parsing table size)
  - For more info: "The dragon book" by A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, Principles, Techniques, and Tools", Addison-Wesley, 1988. (The compilers course textbook.)
- Perform some action upon reading in an expression
- Syntax checking

Yacc's Creations

Yacc used to create

1. C — "Introduction to Compiler Construction with UNIX" by A. T. Schreiner and H. G. Friedman, Jr., Prentice-Hall, 1985.
2. Phototypesetter language (troff?), APL, etc.
3. Examples presented here
   (a) Calculator (the basics of Yacc)
   (b) Roman to Arabic number converter (just given)
   (c) Toshi's toy language — directly related to the project
4. SQL parser — see the optional "lex & yacc" textbook
   - SQL = "Standard Query Language"
   - Used by practically every database program now

Let's jump right in! We write a calculator. Below is the Lex file **expr.l**

```
%{
extern int yylval;
%}

%%

[ \t]    ECHO;
"="      { ECHO;
           return('='); }
[0-9]+   { ECHO;
           yylval = atoi(yytext);
           return(NUMBER); }
"+"      { ECHO;
           return('+'); }
"-"      { ECHO;
           return('-'); }
"*"      { ECHO;
           return('*'); }
"/"      { ECHO;
           return('/'); }
\n       { return 0;  /* end of input */ }

%%
```

Below is file `expr.y`

```
%token NUMBER

// Place marker --- remove this line before compiling!!!

%%

line : /* empty */      { printf("\nempty line\n"); }
     | expr '='         { printf("Answer = %d\n", $1); }
     ;

expr : expr '+' expr    { $$ = $1 + $3; }
     | expr '-' expr    { $$ = $1 - $3; }
     | expr '*' expr    { $$ = $1 * $3; }
     | expr '/' expr    { $$ = $1 / $3; }
     | NUMBER           { $$ = $1; }
     ;

%%
```

- `atoi` converts a string into an integer
- `line` and `expr` are variables
- `line` is the start symbol
- NUMBER is a *token* — value placed in yylval (yy left value)
- A **token** signals to Yacc a terminal symbol
- $$ stores the value of the whole expression
- $n is the value of the n-th part

Below is file `expr.c`

```
#include <stdio.h>
#include "y.tab.c"
#include "lex.yy.c"

main()
{
   yyparse();
}
```

The inclusion of `y.tab.c` *must precede that of* `lex.yy.c`.

To compile do

```
% lex expr.l
% yacc expr.y
conflicts: 16 shift/reduce
% cc -o yacc_expr expr.c -ll -ly
```

where `-ly` is linking the yacc library. Now, if we run it, we get

```
% yacc_expr
9-4-2= <hit return --- below is printed by the program>
9-4-2=Answer = 7
% yacc_expr
9-4+2= <hit return --- below is printed by the program>
9-4+2=Answer = 3
```

Uh oh, what happened? What does this shift/reduce error mean? Why is it giving wrong results? It's doing $9 - (4 - 2)$ instead of the expected $(9 - 4) - 2$.

You will very likely get these errors!

Intro to parsing:

- Parser is a deterministic pushdown automata
- It
  - Calls `yylex` for the next token if needed
  - Based on the token, it either
    - ⋆ shifts — start of a grammar — pushes the current state onto the stack and moves to another state dependent on the token.
    - ⋆ reduce — parser sees the right-hand side of the grammar — pops off stack the state to go back to

What are the errors?

- shift/reduce — when can do either a shift or reduce
  - example: `expr - expr - expr` — can apply the rule `expr` → `expr - expr` on either the first two or last two terms. Both legal.
- reduce/reduce — when there's a choice of two legal reduce
  - example:
    S → Acd | Bce
    A → b
    B → b
    This is the language (bcd)|(bce). Unless you can look two letters ahead, you can't figure out which rule to reduce with.

What to do?

- For ambiguous grammar such as for `expr - expr - expr` can be read as either `( expr - expr ) - expr` or `expr - ( expr - expr )`, add something like

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

  in place of the place marker in `expr.y`. The lower two take higher precedence over the upper two, and they are all left associative. No errors will occur and the desk calculator will give the expected result.
- Do nothing — Yacc has two disambiguating rules:
  1. For shift/reduce — do shift
  2. For reduce/reduce — do earlier grammar reduce
- Try to fix it — the best approach :-)

Shift/Reduce and Reduce/Reduce Errors Part IV

Often, doing nothing works for shift/reduce errors. For example if you have the rule

```
stmt -> IF string stmt |
        IF string stmt ELSE stmt |
        string
```

Can either parse

```
if (cond1) if (cond2) stmt1 else stmt2
```

as either

```
if (cond1)
   if (cond2)
      stmt1
else
   stmt2
```

or

```
if (cond1)
   if (cond2)
      stmt1
   else
      stmt2
```

The latter is what we expect and doing a shift rather than a reduce gives it to us. *Try to fix all errors.*

Can *try* adding

```
%right ELSE string
```

but this means `string` has right precedence for the whole code.

Best to have Yacc take care of this *particular* error.

This is given with little comment (I want you to think about it). Below is `rom.l`:

```
%%
I           { yylval = 1; return(GLYPH); }
V           { yylval = 5; return(GLYPH); }
X           { yylval = 10; return(GLYPH); }
L           { yylval = 50; return(GLYPH); }
C           { yylval = 100; return(GLYPH); }
D           { yylval = 500; return(GLYPH); }
M           { yylval = 1000; return(GLYPH); }

[ \t\n]     { return(WHITE); }

.           ;

%%
```

Below is `rom.y`:

```
%{
int last=0;
%}

%token GLYPH WHITE

%%

file : number                   { printf("Got <%d>\n", $1);
                                  $$ = $1;
                                  last = 0; }
     | file WHITE number        { printf("Got <%d>\n", $3);
                                  $$ = $3;
                                  last = 0; }
     ;

number : GLYPH                  { last = $$ = $1; }
       | GLYPH number           { if ($1 >= last)
                                     $$ = $2 + (last=$1);
                                  else
                                     $$ = $2 - (last=$1); }
       ;

%%
```

Below is `rom.c`:

```
#include <stdio.h>
#include "y.tab.c"
#include "lex.yy.c"

main()
{
   yyparse();
}
```

After compiling (similar to expr) and naming it `rom` (instead of `yacc_expr`), an example run looks like:

```
% rom
XXVII <return>
Got <27>                    <- what the program printed out
MCMLXXXVIII <return>
Got <1988>                  <- what the program printed out
^C                          <- to quit out of the program
```

Toshi's Toy Language — Grammar

START	→	`---begin---` PROG `---end---`
PROG	→	PROG PROG |
		COMMAND |
		REVERSI |
		STUFF |
		PROGSTR |
		λ
PROGSTR	→	TEXT
COMMAND	→	`---combegin---` COMBODY `---comend---`
COMBODY	→	COMBODY COMBODY |
		COMMAND |
		TEXT |
		λ
REVERSI	→	`---rbegin---` TEXT `---rmid---` TEXT `---rend---`

- STUFF is `---stuff`$[a-z0-9\ \ ]*$`---`
  - This is very much like `---stuff---` but with options
  - example: `---stuff optiona optionb---`
- TEXT is $[a-z0-9\ \ ]+$
- Will see sometimes grammar given doesn't work well in Yacc — must fix

Output:

1. If TEXT is encountered outside of PROG, print `Ignoring: TEXT`
2. If STUFF is encountered, print `---my command here---`
3. If TEXT is encountered inside of PROG, print `progstr: TEXT`
4. In place of `---combegin---` print `*cb*`
5. In place of `---comend---` print `*ce*`
6. If TEXT is encountered inside of COMBODY, print `Inside COM: TEXT`
7. If REVERSI is encountered, then e.g. if

   ```
   ---rbegin--- abc ---rmid--- def ---rend---
   ```

   then print out

   ```
   Second: def     First: abc
   ```

8. If there's a syntax error, print the line number it occured on and stop

Below is file `toy.l`

```
%{
extern char *yylval;
extern int  lineno;
extern FILE *ofp;
%}

%START InBody

lowwords        [a-z0-9 ]+
whitespaces     [ \t]+

%%
```

```
---begin---                                { BEGIN InBody;
                                             return(BIGSTART); }
<InBody>---combegin---                     { return(COMSTART); }
<InBody>---comend---                       { return(COMEND); }
<InBody>---rbegin---                       { return(REVSTART); }
<InBody>---rmid---                         { return(REVMID); }
<InBody>---rend---                         { return(REVEND); }
<InBody>---stuff{lowwords}---              { return(STUFF); }
<InBody>---end---                          { BEGIN 0;
                                             return(BIGEND); }
<InBody>{whitespaces}                      ;
<InBody>{lowwords}                         { yylval = strdup(yytext);
                                             return(TEXT); }
<InBody>.                                  ;
{lowwords}                                 { fprintf(ofp,
                                                     "Ignoring: %s\n",
                                                     yytext); }
"\n"                                       { lineno++;
                                             fprintf(ofp, " \n"); }

%%
```

Below is `toy.y`:

```
%{
typedef char *CHAR_PTR;
#define YYSTYPE CHAR_PTR
extern int lineno;
extern FILE *ofp;
%}

%token BIGSTART BIGEND
%token COMSTART COMEND
%token REVSTART REVMID REVEND
%token STUFF
%token TEXT

%start start

%%
```

```
start : BIGSTART prog BIGEND
      ;
prog : prog prog
     | command
     | reversi
     | STUFF
           { fprintf(ofp, "\n---my command here---\n"); }
     | progstr
     | /* empty */
     ;
progstr : TEXT
           { fprintf(ofp, "progstr: %s\n", $1);
             free($1); }
        ;
command : COMSTART
           { fprintf(ofp, "*cb*"); }
          combody
          COMEND
           { fprintf(ofp, "*ce*"); }
        ;
```

```
combody : combody combody
        | command
        | TEXT
            { fprintf(ofp, "Inside COM: %s\n", $1);
              free($1); }
        | /* empty */
        ;
reversi : REVSTART TEXT REVMID TEXT REVEND
            { fprintf(ofp, "Second: %s   First: %s\n",
                                $4, $2);
              free($2);  free($4); }
        ;

%%
yyerror(char *s)
{
  fprintf(stderr, "line no. %d \n", lineno+1);
}
```

Below is `toy.c`:

```
#include <stdio.h>
#include <string.h>
#include "y.tab.c"
#include "lex.yy.c"

int lineno = 0;

FILE *ofp;

main(int argc, char *argv[])
{
   yyin = fopen(argv[1], "r");
   ofp = fopen(argv[2], "w");
   yyparse();
   fclose(yyin);
   fclose(ofp);
}
```

Compile the source like this

```
% lex toy.l
% yacc toy.y
conflicts:26 shift/reduce, 2 reduce/reduce
% cc -o toy toy.c -ll -ly
```

The number of errors depends on which machine you run Yacc on. Run the program like this

```
toy toy.input toy.output
```

Below is a sample input file `toy.input`

```
outside line a
outside line b
---begin---
inside line a
---rbegin---alpha zeta ---rmid---beta zeta---rend---
more stuff
more stuff revenge
---combegin--- text1a text1b ---combegin---
text2a text2b text2c ---comend--- test3a ---comend---
---stuff optiona optionb---
inside line b
---end---
outside line a
outside line b
```

```
Ignoring: outside line a
Ignoring: outside line b

progstr: inside line a

Second: beta zeta   First: alpha zeta
progstr: more stuff
progstr: more stuff revenge
*cb*Inside COM:  text1a text1b
*cb*
Inside COM: text2a text2b text2c
*ce*Inside COM:  test3a
*ce*
---my command here---
progstr: inside line b

Ignoring: outside line a
Ignoring: outside line b
```

- Lex
  1. Must copy the string into `yylval` via *strdup*
  2. Notice the STUFF token is the same regardless of options to stuff
  3. `lineno` incremented each time newline is found — keep track of what line we're on

- Yacc
  1. Define the type of yylval by defining YYSTYPE — default is int
  2. Can put more than one token on a line
  3. The `%start` denotes which rule to start with. By default, it's the left hand side of the first grammar
  4. `free` is used to free the space allocated by `strdup`
  5. Output spaced since I put lots of \n's in the printf's
  6. Note the function `yyerror` is called when there's a parsing error. Read Yacc manual for more details. Recovery is hard! If I add another `---combegin---` right before the first `---combegin---`, I get on console

     line no. 11

     This is since STUFF shouldn't appear inside a combody.
- C
  1. <string.h> needed for `strdup`
  2. In project, you should do better than what I did for getting file names
  3. Similar to `interesting.c` in Lex lecture

- We saw it works, but let's fix it
  - Good example of how to fix errors
  - Very relevant to the yacc project
- You can *try* to use the `-v` option for yacc, e.g.

```
% yacc -v toy.y
```

  and view `y.output`. Unfortunately, usually not very readable.

```
prog : prog command
     | prog reversi
     | prog STUFF
            { fprintf(ofp, "\n---my command here---\n"); }
     | prog progstr
     | /* empty */
     ;
combody : combody command comtext
        | combody command
        | comtext
        | /* empty */
        ;
comtext : TEXT
           { fprintf(ofp, "Inside COM: %s\n", $1);
             free($1); }
```

- Replacing the rules on slides 22 and 23 by the rules on the previous slide, and re-yacc'ing it, no shift/reduce or reduce/reduce errors will result
- No change to the `command` rule needed
- `comtext` was introduced just to make reading it easier — no real change
- Still does the same thing
- General idea: If you have something like

  $$A \rightarrow AA \mid B \mid C \mid q \mid \lambda$$

  then rewrite it as

  $$A \rightarrow AB \mid AC \mid Aq \mid \lambda$$

- Try to make left recursive — Yacc generates smaller parser than right recursive