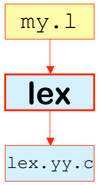
Lex and Yacc

A Quick Tour

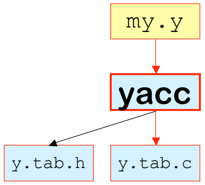
Lex (& Flex): A Lexical Analyzer Generator

- Input:
  - Regular exprs defining "tokens"
  - Fragments of C decls & code
- Output:
  - A C program "lex.yy.c"
- Use:
  - Compile & link with your main()
  - Calls to yylex() return successive tokens.



Yacc (& Bison & Byacc…): A Parser Generator

- Input:
  - A context-free grammar
  - Fragments of C declarations & code
- Output:
  - A C program & some header files
- Use:
  - Compile & link it with your main()
  - Call yyparse() to parse the entire input file
  - yyparse() calls yylex() to get successive tokens



Lex Input: "mylexer.l"

```
%{
   #include …
   int myglobal;
   …
%}
%%
[a-zA-Z]+    {handleit(); return 42; }
[ \t\n]      {; /* skip whitespace */}
…
%%
void handleit() {…}
…
```

Declarations: To front of C program

Token code

Rules and Actions

Subroutines: To end of C program

Yacc Input: “expr.y”

S → E
E → E+n | E-n | n

C Decls
```
%{
   #include …
%}
```
→ y.tab.c

Yacc Decls
```
%token NUM VAR
%%
```
→ y.tab.h

Rules and Actions
```
stmt: exp             { printf(”%d\n”,$1);}
 ;
exp : exp ’+’ NUM  { $$ = $1 + $3; }
 |    exp ’-’ NUM  { $$ = $1 - $3; }
 |    NUM          { $$ = $1; }
 ;
%%
```

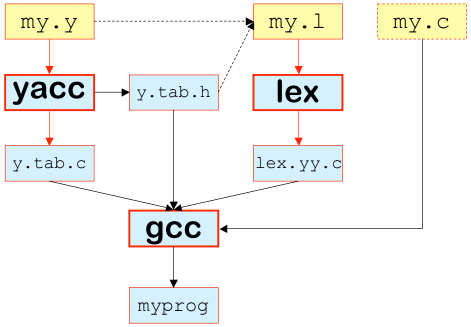
Subrs
```
…
```
→ y.tab.c

Expression lexer: “expr.l”

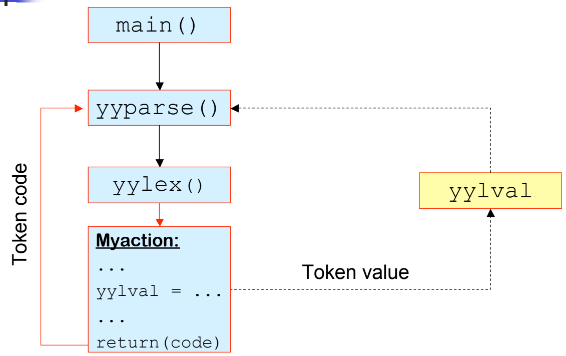
```
%{
#include "y.tab.h"


%}
%%
[0-9]+      { yylval = atoi(yytext); return NUM;}
[ \t]       {           /* ignore whitespace */ }
\n          { return 0;        /* logical EOF */ }
.           { return yytext[0]; /* +-*, etc. */ }
%%
yyerror(char *msg){printf("%s,%s\n",msg,yytext);}
int yywrap(){return 1;}
```

y.tab.h:
```
#define NUM     258
#define VAR     259
#define YYSTYPE int
extern   YYSTYPE yylval;
```

Lex/Yacc Interface: Compile Time



Lex/Yacc Interface: Run Time



Some C Tidbits

Enums

```
enum kind {
   title_kind,center_kind};
typedef struct node_s{
    enum kind k;
    struct node_s
      *lchild,*rchild;
    char *text;
  } node_t;
node_t root;
root.k = title_kind;
if(root.k==title_kind){…}
```

Malloc

```
root.rchild = (node_t*)
 malloc(sizeof(node_t));
```

Unions

```
typedef union {
      double d;
      int i;
 } YYSTYPE;
extern YYSTYPE yylval;
yylval.d = 3.14;
yylval.i = 3;
```

More Yacc Declarations

```
%union {
    node_t *node;
    char   *str; }

%token <str> BHTML BHEAD BTITLE BBODY BCENTER
%token <str> EHTML EHEAD ETITLE EBODY ECENTER
%token <str> P BR LI TEXT

%type <node> page head title words body
%type <node> heading list center item items

%start page
```

Type of yylval

Token names & types

Nonterm names & types

Start sym

Yacc In Action

PDA stack: alternates between "states" and symbols from (V ∪ Σ).

```
initially, push state 0
while not done {
  let S be the state on top of the stack;
  let i be the next input symbol (i in Σ);
  look at the the action defined in S for i:
    if "accept", halt and accept;
    if "error", halt and signal a syntax error;
    if "shift to state T", push i then T onto the stack;
    if "reduce via rule r (A → α )", then:
       pop exactly 2*|α| symbols
         (the 1st, 3rd, ... will be states, and
          the 2nd, 4th, ... will be the letters of α);
       let T = the state now exposed on top of the stack;
       T's action for A is "goto state U" for some U;
       push A, then U onto the stack.
}
```

Implementation note: given the tables, it's deterministic, and fast -- just table lookups, push/pop.