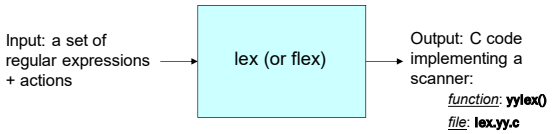
A brief lex & Yacc tutorial

Slides originally created by:
Saumya Debray (The Univ. of Arizona)
Modifications by:
Amey Karkare (IIT Kanpur)

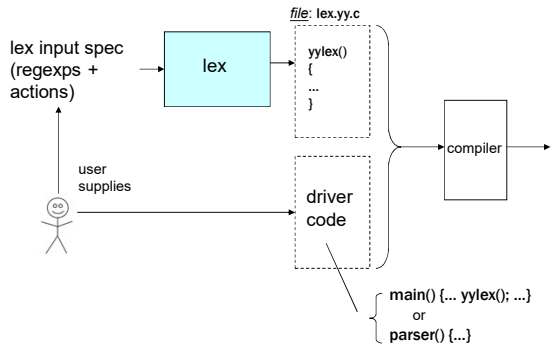
flex (and lex): Overview

Scanner generators:

- Helps write programs whose control flow is directed by instances of regular expressions in the input stream.

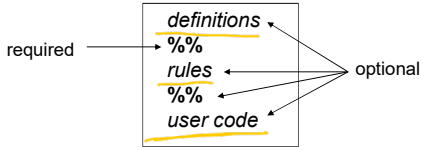


Using flex



flex: input format

An input file has the following structure:



Shortest possible legal flex input:

```
%%
```

It works like a cat program

Definitions

- A series of:
  - *name definitions*, each of the form
    name *definition*
    e.g.:
    DIGIT [0-9]
    CommentStart "/*"
    ID [a-zA-Z][a-zA-Z0-9]*
  - *start conditions*
  - stuff to be copied verbatim into the flex output (e.g., declarations, **#include**s):
    - enclosed in %{ … %}

Rules

- The *rules* portion of the input contains a sequence of rules.
- Each rule has the form
  pattern *action*
  where:
  - *pattern* describes a pattern to be matched on the input
  - *pattern* must be un-indented
  - *action* must begin on the same line.(version dependent), for multi lined action : use {}

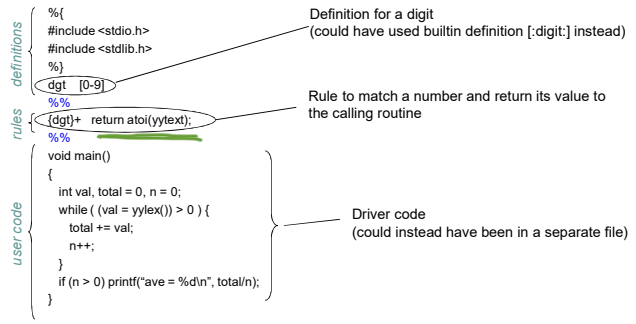
Example

A flex program to read a file of (positive) integers and compute the average:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt   [0-9]
%%
{dgt}+   return atoi(yytext);
%%
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n", total/n);
}
```


Example

A flex program to read a file of (positive) integers and compute the average:



Example

A flex program to read a file of (positive) integers and compute the average:

```
definitions
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt [0-9]
%%
rules
{dgt}+ return atoi(yytext);
%%
user code
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n", total/n);
}
```

defining and using a name

Example

A flex program to read a file of (positive) integers and compute the average:

```
definitions
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt [0-9]
%%
rules
{dgt}+ return atoi(yytext);
%%
user code
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n", total/n);
}
```

char * yytext;
a buffer that holds the input characters that actually match the pattern

LHS: Non-terminal
RHS: – " – + Terminals

For input:
⇒ 12x14y16z
xyz
0
ave = 14

* xyz is printed back: not really ignored.

Example

A flex program to read a file of (positive) integers and compute the average:

```
definitions
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt [0-9]
%%
rules
{dgt}+ return atoi(yytext);
%%
user code
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n", total/n);
}
```

defining and using a name

char * yytext;
a buffer that holds the input characters that actually match the pattern

Invoking the scanner: **yylex()**
Each time yylex() is called, the scanner continues processing the input from where it last left off.
Returns 0 on end-of-file.

Returns token no.
* Here, token no. = value of no.
we get return from the rules ⇒ value of no.

need to define in lexical analyzer

* Here (≤0) can't be handled by this simple calculator.
* It can't handle "\n", thus it's printed again.

Hands On Example

BNF format
yacc syntax

```
program : slist
    | /* empty */
    ;
slist : assign ';'
    | exit_command ';'
    | print exp ';'
    | slist assign ';'
    | slist print exp ';'
    | slist exit_command ';'
    ;
assign : identifier '=' exp
```

```
exp : term
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '(' exp ')'
    ;

term : number
    | identifier
    ;
```

yacc ; → source language

tokens or terminals

In YACC, we define our tokens separately

Lex script

```
%{
    #include <iostream>
    #include "y.tab.h"
    #include <cstring>
    void yyerror (char *s);
    int yylex();
%}

digit    [0-9]
alpha    [a-zA-Z_]
alphanum ({alpha}|{digit})
ws       [ \t\n]

%%
```


Lex script (contd…)

```
"print"     { return print; }
"calculate" { return print; }
"exit"      { return exit_command; }

{digit}+    { yylval.num = atoi(yytext);
              return number; }

{alpha}{alphanum}*    {
              yylval.id = strdup(yytext);
              return identifier; }

{ws}              ; /* nothing */

[-+\(\)=/*\n;]    { return yytext[0]; }

.                 { ECHO; yyerror("..."); }
```


Matching the Input

- When more than one pattern can match the input, the scanner behaves as follows:
  - the longest match is chosen;
  - if multiple rules match, the rule listed first in the flex input file is chosen;
  - if no rule matches, the default is to copy the next character to **stdout**.

```
(cs)  {printf("Department");}
(cs)[0-9]{3} {printf("Course");}
[a-zA-Z]+[0-9]+ {printf("AnythingElse");}
Input: cs335
```


Control flow of lexer

```
yylex() {
    /*scan the file pointed to by yyin (default stdin)*/
        1.Repeated call of input() to get the next character from the input .....
        2. Occasional calls of unput() .....
        3. Try matching with the regular expression and when matched do the action part.....

    /*      got  EOF */
    int status = yywrap(); /*default behaviour – return 1 */
    if(1 == status)
        exit() ;
    else
        yylex() ;
}
/*Redefine yywrap to handle multiple files*/
int yywrap() {
 .....
if(exists other files to process ) { yyin = nextFilePtr; return 1; }
else { return 0; }
}
```

Command Line Parsing

Lookahead

Start Conditions

- Used to activate rules conditionally.
  - Any rule prefixed with <*S*> will be activated only when the scanner is in start condition *S*.

```
%s MAGIC      <-------Inclusive start condition
%%
<MAGIC>.+     {BEGIN INITIAL; printf("Magic: "); ECHO; }
magic         {BEGIN MAGIC}
```

Input: magic two three

Warning: A rule without an explicit start state will match regardless of what start state is active.

```
%s MAGIC      <-------Inclusive start condition
%%
magic         {BEGIN MAGIC}
.+            ECHO;
<MAGIC>.+     {BEGIN INITIAL; printf("Magic: "); ECHO; }
```


Start Conditions (cont'd)

WayOut:

- Use of explicit start state using %x MAGIC
- For versions that lacks %x

```
%s NORMAL MAGIC
%%
%{
    BEGIN NORMAL;
%}
<NORMAL>magic         {BEGIN MAGIC}
<NORMAL>.+            ECHO;
<MAGIC>.+     {BEGIN NORMAL; printf("Magic: "); ECHO; }
```


Putting it all together

- Scanner implemented as a function
  `int yylex();`
  - return value indicates type of token found (encoded as a +ve integer);
  - the actual string matched is available in yytext.
- When compiling, link in the flex library using "-ll" (or equivalent flag)
- You need token type encodings
  - Without Yacc, manually define const/enum/#define names in a file y.tab.h
  - use "#include y.tab.h" in the definitions section of the flex input file.
- Later, it can be replaced by Yacc generated y.tab.h

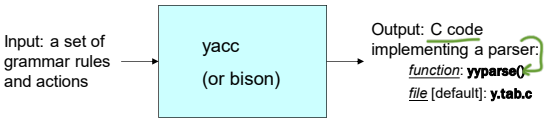
Syntax Analyzer Generator (YACC)
"Yet Another Compiler Compiler"

bison

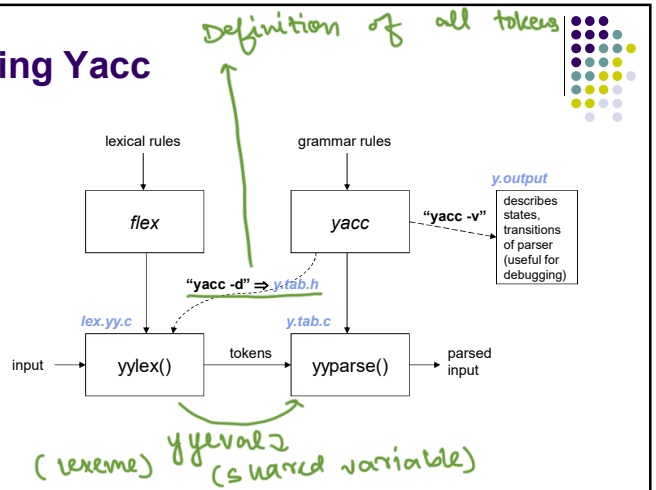
Yacc: Overview

Parser generator:

- Takes a specification for a context-free grammar.
- Produces code for a parser.



Using Yacc



int yyparse()

supposed to take whole file

- Called once from main() [*user-supplied*]
- Repeatedly calls yylex() until done:
  - On syntax error, calls yyerror() [*user-supplied*]
  - Returns 0 if all of the input was processed;
  - Returns 1 if aborting due to syntax error.

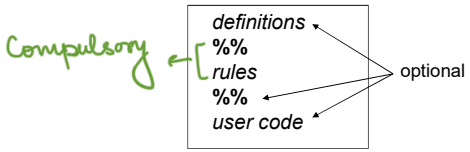
Example:

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


yacc: input format

A yacc input file has the following structure:

%% → error or empty language



Lex – Regular expressions

Yacc – Grammar

- Parser

```
%{
    #include<...>
    Declarations .... Copied verbatim into the output yyparse()
%}
%token NAME NUMBER
%%
statement  : NAME '=' expression
           | expression {printf(" %d\n", $1);}
           ;
expression : expression '+' NUMBER    {$$ = $1 + $3;}
           | expression '*' NUMBER    {$$ = $1 * $3;}
           | NUMBER                   {$$ = $1 ;}
%%
main()  { yyparse(); }
```

- Lexer

```
%{
    #include"y.tab.h"
%}
%%
[0-9]+      {yylval = atoi(yytext); return NUMBER; }
[ \t\n]     ; /*Ignore whitespace */
.           {return yytext[0] ;}
```


Yacc script (demo) – v1

- Calculator. Try with Inputs:
  - 3 + 5
  - 3 + 5 + 6
  - 2 * 7
  - 3 + 5 * 2
  - val = 3 + 3

Declaring Return Value Types

- Default type for nonterminal return values is **int**.
- Need to declare return value types if nonterminal return values can be of other types:
  - Declare the union of the different types that may be returned:

```
%union {
  struct symtab *st_ptr;
  double        dvalue;
}
```

  - Specify which union member a particular grammar symbol will return:

```
%token <st_ptr> NAME;
%token <dvalue> NUMBER;
%type <dvalue> expression;
```

(terminals / nonterminals)

Yacc script (demo) – v2

- Calculator. Try with Inputs:
  - 3 + 5
  - 3 + 5 + 6
  - 2 * 7
  - 3 + 5 * 2
  - val = 3 + 3

shif / reduce conflicts - associativity & precedence

Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

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

Operators in the same group have the same precedence

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
      | '–' expr    %prec '*'
      | …
```


Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

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

Operators in the same group have the same precedence

Across groups, precedence increases going down

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
      | '–' expr    %prec '*'
      | …
```


Specifying Operator Properties

- Binary operators: **%left**, **%right**, **%nonassoc**:

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

Operators in the same group have the same precedence

Across groups, precedence increases going down.

- Unary operators: **%prec**
  - Changes the precedence of a rule to be that of the token specified. E.g.:

```
%left '+'  '-'
%left  '*'  '/'
Expr: expr '+' expr
      | '–' expr    %prec '*'
      | …
```

The rule for unary '–' has the same (high) precedence as '*'

Yacc script (demo) – v3

* when yacc encounters an error, it skips it and looks for other errors. This can give us more than one error in one time.

* Better turn-around time.

Error Handling

- The "token" '**error**' is reserved for error handling:
  - can be used in rules;
  - suggests places where errors might be detected and recovery can occur.

Example:

```
stmt  : IF '(' expr ')' stmt
      | IF '(' error ')' stmt
      | FOR …
      | …
```

Alignment token — Intended to recover from errors in 'expr'

keyword

; , \n → alignment tokens.

Parser Behavior on Errors

When an error occurs, the parser:

- pops its stack until it enters a state where the token 'error' is legal;
- then behaves as if it saw the token 'error'
  - performs the action encountered;
  - resets the lookahead token to the token that caused the error.
- If no 'error' rules specified, processing halts.

program exits.

Error Messages

- On finding an error, the parser calls a function
  void yyerror(char *s) /* s points to an error msg */
  - user-supplied, prints out error message.
- More informative error messages:
  - int yychar: token no. of token causing the error.
  - user program keeps track of line numbers, as well as any additional info desired.

Conflicts

- A conflict occurs when the parser has multiple possible actions in some state for a given next token.
- Two kinds of conflicts:
  - *reduce-reduce conflict*:
    - Start: x B C
    - | y B D ;
    - x: A;
    - y: A;
  - **y.output** generated using : yacc -v
    - 1: Reduce/reduce conflict (reduce 3, reduce 4) on B.
    - state 1
    - x: A__ (3)
    - y: A__ (4)
    - reduce 3

S → X | Y
X → a
Y → a

S → X → a
S → Y → a
which one??

default : one listed first

Conflicts

- *shift-reduce conflict*:
  - Start: x
  - | y R ;
  - x: A R;
  - y: A;
- **y.output** generated using : yacc -v
  - 4: shift/reduce conflict (shift 6, reduce 4) on R.
  - state 4
  - x: A__R
  - y: A__ (4)
  - R shift 6

S → x → AR
S → yR → AR
* Should we take entire matching part (yR) or only a (x)

Example of a conflict

Grammar rules:

```
expr : TERMINAL
     | expr '-' expr

y.output:
    4: Shift/reduce conflict (shift 3, reduce 2) on –
    State 4:
        expr : expr__ -  expr
        expr : expr  -  expr_
```

Way Out : define precedence and associativity of the operators.

Example of a conflict

Grammar rules:

```
rule : command optional_keyword '(' id_list ')'
     ;
Optional_keyword:
                | '(' keyword ')'
                ;
```

Shift / reduce conflict on first parenthesis (in input stream.

Way Out : flattening

```
rule : command '(' id_list ')'
     | command '(' keyword ')'  '(' id_list ')';
     ;
```


Example of a conflict

Grammar rules:

```
rule :  grp_A
     | grp_B
     ;
grp_A: A_1
     |  COMM
;
 grp_B: B_1
     |  COMM
;
```

Way Out : flattening

Add COMM in a separate rule.

```
grp_comm : COMM;
```

Putting it all together

- Scanner implemented as a function

  int yylex();

  - return value indicates type of token found (encoded as a +ve integer);
  - the actual string matched is available in yytext.
- When compiling, link in the flex library using "-ll" (or equivalent flag)
- Scanner and parser need to agree on token type encodings
  - let yacc generate the token type encodings
    - yacc places these in a file y.tab.h
  - use "#include y.tab.h" in the definitions section of the flex input file.