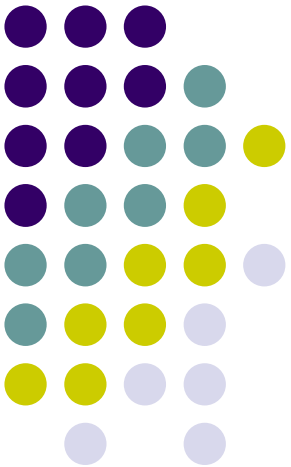
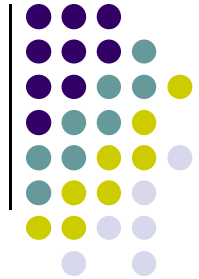
Chapter # 9
LEX and YACC

Dr. Shaukat Ali
Department of Computer Science
University of Peshawar

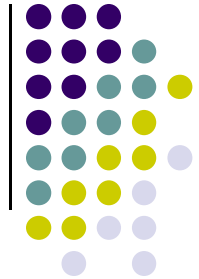


Lex and Yacc



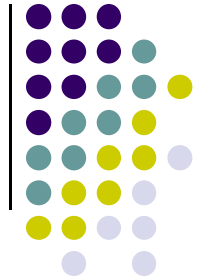
- Lex and yacc are a matched pair of tools.
- Lex breaks down files into sets of "tokens," roughly analogous to words.
- Yacc takes sets of tokens and assembles them into higher-level constructs, analogous to sentences.
- Lex's output is mostly designed to be fed into some kind of parser.
- Yacc is designed to work with the output of Lex.

Lex and Yacc



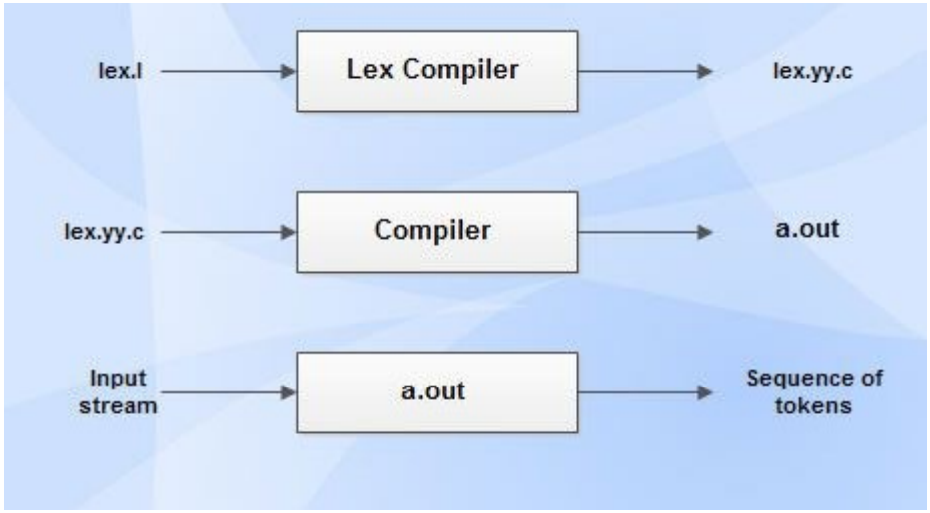
- Lex and yacc are tools for building programs.
  - Their output is itself code
    - Which needs to be fed into a compiler
    - May be additional user code is added to use the code generated by lex and yacc

Lex : A lexical analyzer generator



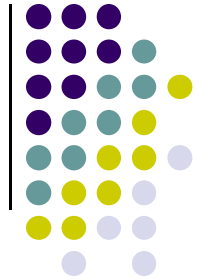
- Lex is a program designed to generate scanners, also known as tokenizers, which recognize lexical patterns in text
- Lex is an acronym that stands for "lexical analyzer generator.“
- The main purpose is to facilitate lexical analysis
  - The processing of character sequences in source code to produce tokens for use as input to other programs such as parsers
- Another tool for lexical analyzer generation is Flex

Lex : A lexical analyzer generator

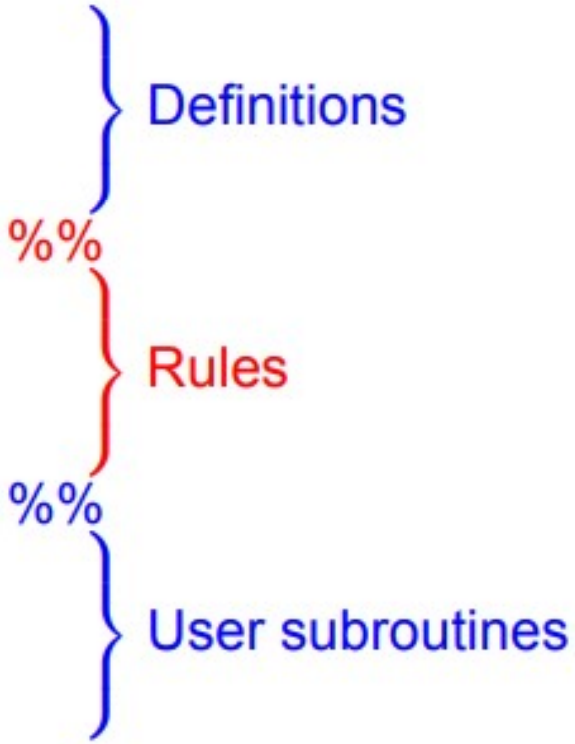


- ***lex.lex*** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms *lex.l* to a C program known as *lex.yy.c.*
- ***lex.yy.c*** is compiled by the C compiler to a file called *a.out.*
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.

Lex: A lexical analyzer generator

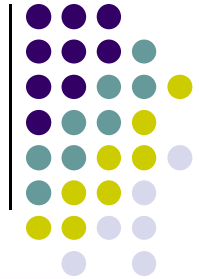


Structure of Lex Specification File



red : required
blue : optional

Lex: A lexical analyzer generator

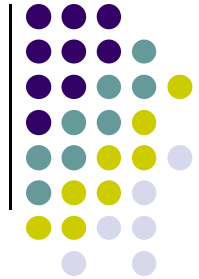


snazzle.lex:

```
%{
#include <iostream>
%}
%%
[ \t] ;
[0-9]+\.[0-9]+ { cout << "Found a floating-point number:" << yytext << endl; }
[0-9]+  { cout << "Found an integer:" << yytext << endl; }
[a-zA-Z0-9]+ { cout << "Found a string: " << yytext << endl; }
%%
main() {
        // lex through the input:
        yylex();
}
```

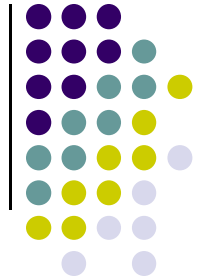
- Lex file has three sections.
  - The first is sort of "control" information,
  - The second is the actual token or grammar rule definitions,
  - The last is C code to be copied verbatim to the output.

Lex: A lexical analyzer generator



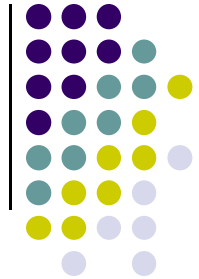
- Lines 1 through 3 are more C code to be copied.
  - In the control section, you can indicate C code to be copied to the output by enclosing it with "%{" and "%}“
    - This section includes include files, declaration of variables, and constants
  - We wouldn't need one at all if we didn't use **cout** in the middle section
- Line 4 is "%%", which means we're done with the control section and moving on to the token section.

Lex: A lexical analyzer generator



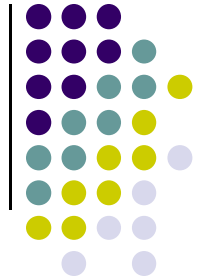
- Lines 5-8 are all the same (simple) format: they define a regular expression and an action (code segment).
  - Form : Pattern {Action}
    - Pattern is regular expression and action is code segment
  - When lex is reading through an input file and can match one of the *regular expressions*, it executes the *action*.
  - The *action* is just C++ code that is copied into the eventual lex output
    - You can have a single statement or you can have curly braces with a whole bunch of statements.

Lex: A lexical analyzer generator



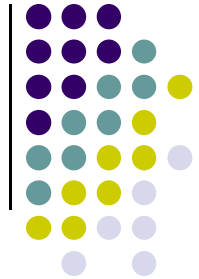
- Line 9 is another "%%" delimiter, meaning we're done with the second section and we can go onto the third.
- Lines 10-13 are the third section, which is exclusively for copied C code.
  - main() function – containing important call to yylex() function
  - Additional functions which are used in actions
  - These functions are compiled separately and loaded with lexical analyzer in the Lex output file

Lex: A lexical analyzer generator



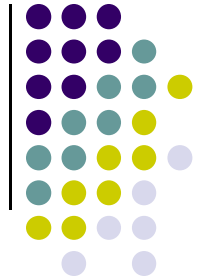
- Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found
- Once a match is found, the associated action takes place to produce token
- The token is then given to parser for further processing

Lex: A lexical analyzer generator



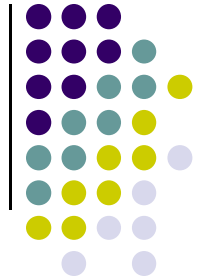
- Operators : " \ [ ] ˆ - ? . * | ( ) $ / { } % < >
- Letters and digits match themselves
- Period '.' matches any character (except newline)
- Brackets [ ] enclose a sequence of characters, termed a character class. This matches:
  - Any character in the sequence
  - A '-' in a character class denotes an inclusive range,
    - e.g.: [0-9] matches any digit.
  - A ˆ at the beginning denotes negation: [ˆ0-9] matches any character that is not a digit.

Lex: A lexical analyzer generator



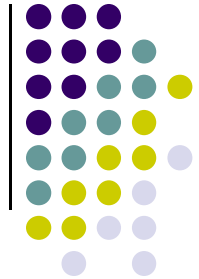
- A quoted character " " matches that character.
- \n, \t match newline, tab.
- parentheses () grouping
- Bar | alternatives
- Star * zero or more occurrences
- + one or more occurrence
- ? zero or one occurrence

Operators



Metacharacter	Matches
`.`	any character except newline
`\n`	newline
`*`	zero or more copies of the preceding expression
`+`	one or more copies of the preceding expression
`?`	zero or one copy of the preceding expression
`^`	beginning of line
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`"a+b"`	literal `"a+b"` (C escapes still work)
`[]`	character class

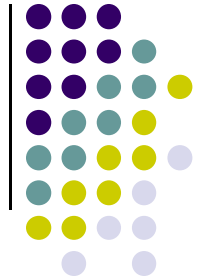
Lex: A lexical analyzer generator



Examples of Lex Rules

- `int  printf("keyword: INTEGER\n");`
- `[0-9]+  printf("number\n");`
- `"-"?[0-9]+("."[0-9]+)?  printf("number\n");`

Lex: A lexical analyzer generator



Choosing between different possible matches:

When more than one pattern can match the input, lex chooses as follows:

1. The *longest* match is preferred.

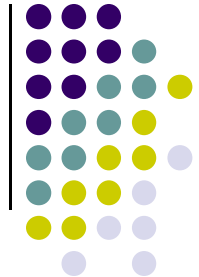
2. Among rules that match the same number of characters, the rule that occurs earliest in the list is preferred.

Example : the pattern

```
"/""*"(.|\n)*"*""/"
```

(intended to match multi-line comments) may consume all the input!

Lex: A lexical analyzer generator



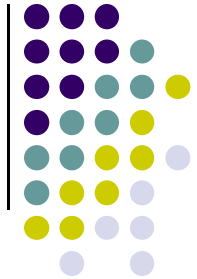
Lex source definitions

- Any source not intercepted by lex is copied into the generated program:
  - a line that is not part of a lex rule or action, which begins with a blank or tab, is copied out as above (useful for, e.g., global declarations)
  - anything included between lines containing only `%{` and `%}` is copied out as above (useful, e.g., for preprocessor statements that must start in col.1)
  - anything after the second `%%` delimiter is copied out after the lex output (useful for local function definitions).
- Definitions intended for lex are given before the first `%%`. Any line in this section that does not begin with a blank or tab, or is not enclosed by `%{ . . . %}`, is assumed to be defining a lex substitution string of the form

  name translation

  E.g.:

  ```
  letter    [a-zA-Z]
  ```

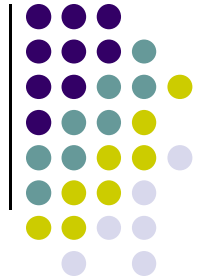


```
%{
#include "tokdefs.h"
#include <strings.h>
static int id_or_keywd(char *s);
%}

letter          [a-zA-Z]
digit           [0-9]
alfa            [a-zA-Z0-9_]
whitesp         [ \t\n]
%%
{whitesp}*              ;
{comment}               ;
{letter}{alfa}          REPORT(id_or_keywd(yytext), yytext);
   ...
%%
static struct {
  char *name;
  int val;
} keywd_entry,
  keywd_table[] = {
    "char",             CHAR,
    "int",              INT,
    "while",            WHILE,
        ...
   };

static int id_or_keywd(s)
```

Lex: A lexical analyzer generator



- This example can be compiled by running this:

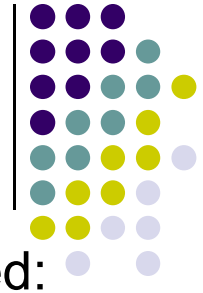
  % lex snazzle.lex

- This will produce the file "lex.yy.c", which we can then compile with g++:

  % g++ lex.yy.c -lfl -o snazzle

- Notice the "-lfl", which links in the dynamic lex libraries

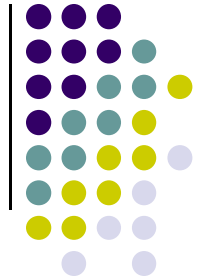
Lex: A lexical analyzer generator



- You should be able to run it and enter stuff on STDIN to be lexed:

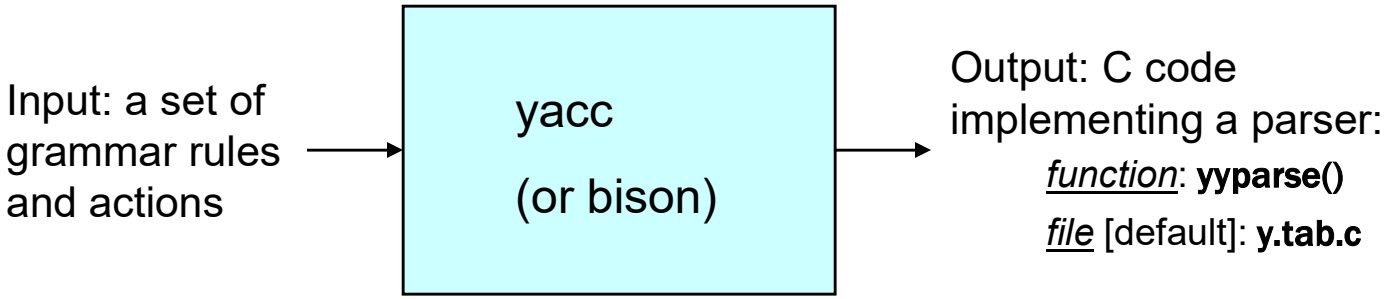
```
% ./snazzle
90
Found an integer:90
23.4
Found a floating-point number:23.4
4 5 6
Found an integer:4
Found an integer:5
Found an integer:6
this is text!
Found a string: this
Found a string: is
Found a string: text
!
```

Yacc: Overview

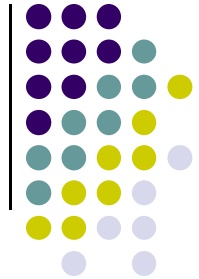


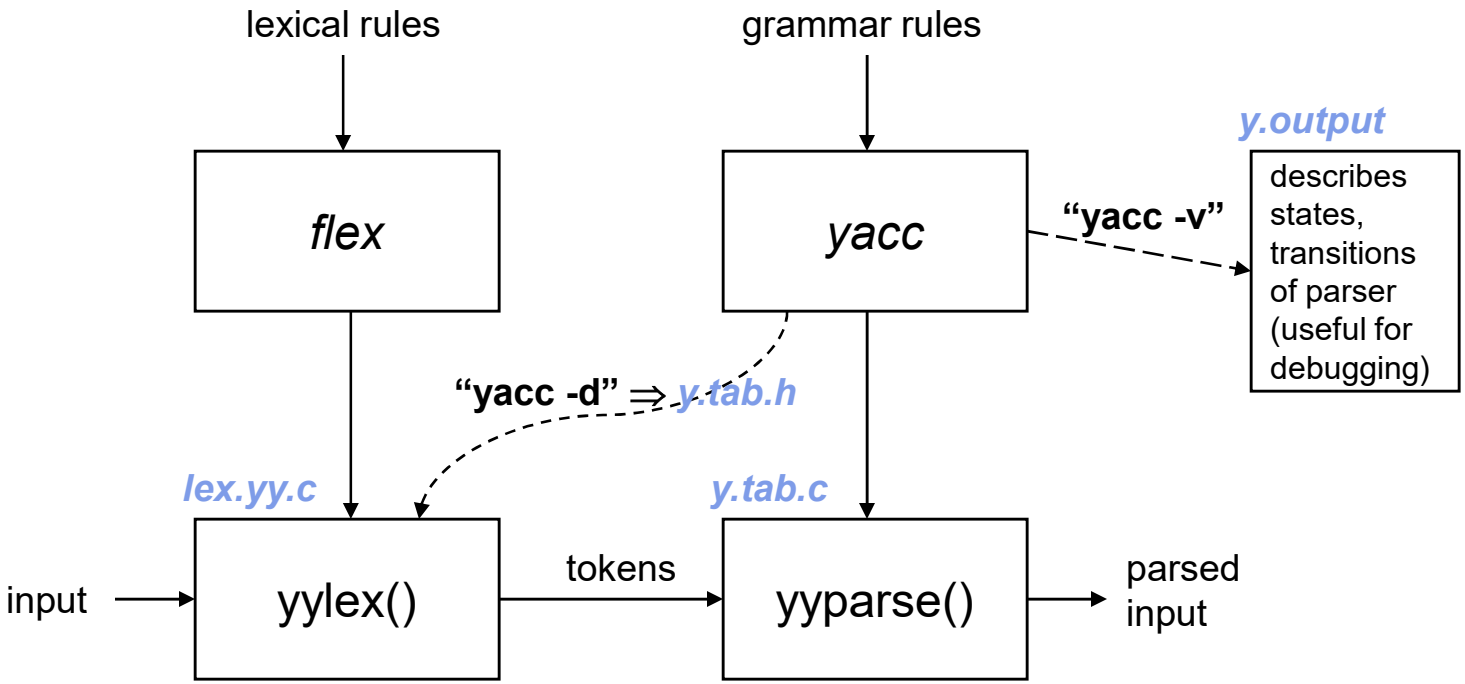
Parser generator:

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

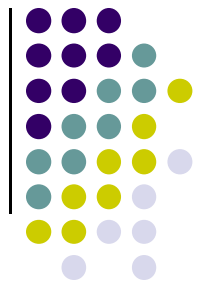


Using Yacc



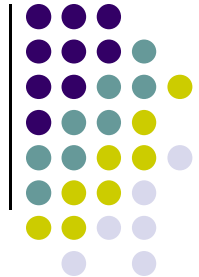

lexical rules
grammar rules
y.output
flex
yacc
"yacc -v"
describes states, transitions of parser (useful for debugging)
"yacc -d" ⇒ y.tab.h
lex.yy.c
y.tab.c
input
yylex()
tokens
yyparse()
parsed input

Communication between Scanner and Parser



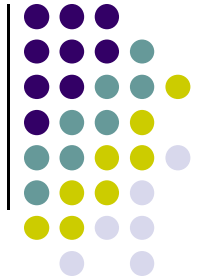
- Yacc determines integer representations for tokens:
  - Communicated to scanner in file **y.tab.h**
    - use "**yacc -d**" to produce **y.tab.h**
  - Token encodings:
    - "end of file" represented by '0';
    - A character literal: its ASCII value;
    - Other tokens: assigned numbers ≥ 257.
- Parser assumes the existence of a function '**int yylex()**' that implements the scanner.
- Scanner:
  - Return integer value indicates the type of token found
  - Values communicated to the parser using **yytext**, **yylval**
  - **yytext** determines lexeme of a token and **yylval** determines a integer assigned to a token
  - The token **error** is reserved for error handling

Communication between Scanner and Parser

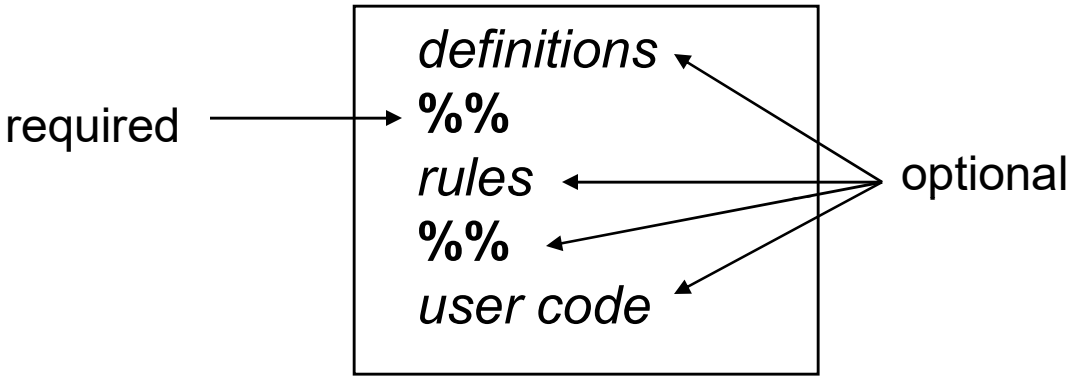


- Suppose the grammar spec is in a file `foo.y`. Then:
  - The command '`yacc foo.y`' yields a file `y.tab.c` containing the parser constructed by yacc.
  - The command '`yacc -d foo.y`' constructs a file `y.tab.h` that can be `#include`'d into the scanner generated by `lex`.
  - The command '`yacc -v foo.y`' additionally constructs a file `y.output` containing a description of the parser (useful for debugging).
- The user needs to supply a function `main()` to driver, and a function `yyerror()` that will be called by the parser if there is an error in the input.

yacc: input format

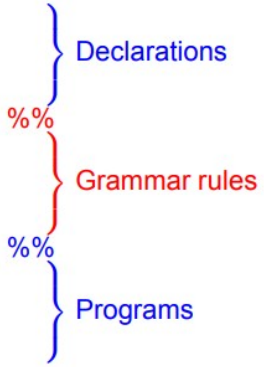


A yacc input file has the following structure:



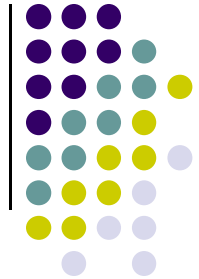
Shortest possible legal yacc input:

```
%%
```



red : required
blue : optional

int yyparse()

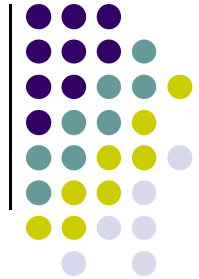


- Called once from main() [*user-supplied*]
- Repeatedly calls yylex() until done:
  - On syntax error, calls yyerror() [*user-supplied*]
  - Returns 0 if all of the input was processed;
  - Returns 1 if aborting due to syntax error.

Example:

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

Yacc: Grammar Rules

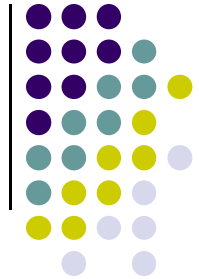


- Information about tokens:
  - Token names:
    - Declared using '**%token**'

      %token name1 name2 ...
    - Any name not declared as a token is assumed to be a nonterminal.
  - Start symbol of grammar, using '**%start**' [optional]

    %start name
    - If not declared explicitly, defaults to the nonterminal on the LHS of the first grammar rule listed
  - Stuff to be copied verbatim into the output (e.g., declarations, **#include**s): enclosed in **%{** … **}%**

Yacc: Grammar Rules



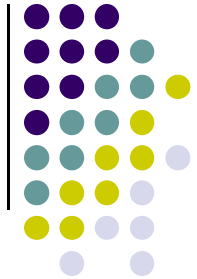
Grammar production		yacc rule
$A \rightarrow B_1\ B_2\ \ldots\ B_m$		$A\text{: } B_1\ B_2\ \ldots\ B_m$;
$B \rightarrow C_1\ C_2\ \ldots\ C_n$	➡	$B\text{: } C_1\ C_2\ \ldots\ C_n$;
$C \rightarrow D_1\ D_2\ \ldots\ D_k$		$C\text{: } D_1\ D_2\ \ldots\ D_k$;
		; /* ';' optional, but advised */

- Rule RHS can have arbitrary C code embedded, within { … }. E.g.:

  A : B1 { printf("after B1\n"); x = 0; } B2 { x++; } B3;

- Left-recursion more efficient than right-recursion:
  - A : A x | … rather than A : x A | …

Specifying Operator Properties



- Binary operators: **%left**, **%right**, **%nonassoc**:
  - Associativity of operator

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

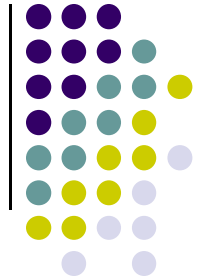
Operators in the same group have the same precedence

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**:

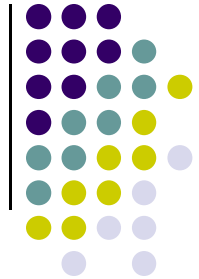
```
%left  '+'  '-'
%left  '*'  '/'
%right '^‘
```

Operators in the same group have the same precedence

- 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

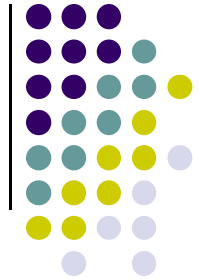
Across groups, precedence increases going down.

- 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: Error Handling



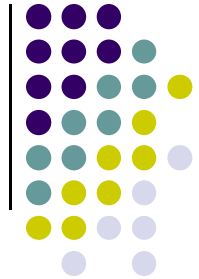
- The “token” ‘**error**’ is reserved for error handling:
  - Can be used in rules;
  - Suggests places where errors might be detected and recovery can occur.

Example:

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

Intended to recover from errors in ‘expr’

Error Messages



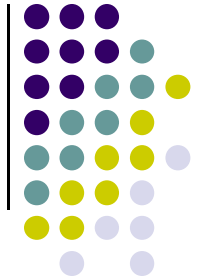
- On finding an error, the parser calls a function

  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.

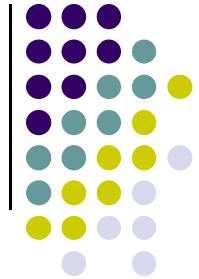
Error Messages: example



```
#include "y.tab.h"
extern int yychar, curr_line;
static void print_tok()
{
   if (yychar < 255) {
     fprintf(stderr, "%c", yychar);
   }
   else {
      switch (yychar) {
      case ID: …
      case INTCON: …
      …
      }
   }
}
```

```
void yyerror(char *s)
{
   fprintf(stderr,
           "[line %d]: %s",
            curr_line,
            s);
   print_tok();
}
```

Adding Semantic Actions

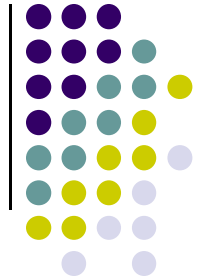


- Semantic actions for a rule are placed in its body:
  - an action consists of C code enclosed in { … }
  - may be placed anywhere in rule RHS

Example:

expr : ID { symTbl_lookup(idname); }

decl : type_name { tval = … } id_list;



- End of Chapter # 9