Lex and Yacc: A Brisk Tutorial

Saumya K. Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Lex: A Scanner Generator

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

Table of regular expressions
+ associated actions

↓

Lex

↓

yylex()
(in file **lex.yy.c**)

- `yylex()` :
  - matches the input stream against the table of regular expressions supplied
  - carries out the associated action when a match is found.

Structure of Lex Specification File

Definitions
%%
Rules
%%
User subroutines

red : required
blue : optional

Rules : line oriented:

⟨*reg. exp*⟩ ⟨*whitespace*⟩ ⟨*action*⟩

⟨*reg. exp*⟩ : starts at beginning of line, continues upto first unescaped whitespace

⟨*action*⟩ : a single C statement
(multiple statements: enclose in braces { }).

unmatched input characters : copied to `stdout`.

Lex Regular Expressions

Similar to `egrep` :

- *operators* : `" \ [ ] ^ - ? . * | ( ) $ / { } % < >`
- 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 `^` at the beginning denotes *negation*: `[^0-9]` matches any character that is not a digit.
- a quoted character `" "` matches that character.
  operators can be escaped via `\`.
- `\n`, `\t` match newline, tab.
-

parentheses	`()`	grouping
bar	`|`	alternatives
star	`*`	zero or more occurrences
	`+`	one or more occurrence
	`?`	zero or one occurrence

Examples of Lex Rules

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

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!

Communicating with the user program

`yytext` : a character array that contains the actual string that matched a pattern.

`yyleng` : the no. of characters matched.

Example :

- `[a-z][a-z0-9_]*  printf("ident: %s\n", yytext);`
- Counting the number of words in a file and their total size:

  ```
  [a-zA-Z]+   {nwords += 1; size += yyleng;}
  ```


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]
  ```


An Example

```
%{
#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)
char *s;
{
  ...
}
```

Left Context Sensitivity: Start Conditions

Start conditions are a mechanism for conditionally activating patterns. This is useful for handling

- conceptually different components of an input; or
- situations where the lex defaults (e.g., "longest possible match") don't work well, e.g., comments or quoted strings.

Basic Idea:

- Declare a set of *start condition names* using

  `%Start` $name_1$ $name_2$ ...

- If *scn* is a start condition name, then a pattern prefixed with <*scn*> will only be active when the scanner is in start condition *scn*.
- The scanner begins in start condition INITIAL, of which all non-<*scn*>-prefixed rules are members.
- Start conditions such as these are *inclusive*: i.e., being in that start condition adds appropriately prefixed rules to the active rule set.

  `flex` also allows *exclusive* start conditions (declared using `%x`), which are sometimes more convenient.

Example of use of start conditions

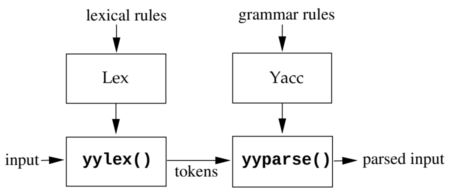
```
%Start comment0 comment1
%{
#include "tokens.h"
%}
whitespace      [ \t\n]
digit           [0-9]
intcon          {digit}+
floatcon        {digit}+"."{digit}+
start_comment   "/""*"

%%
<INITIAL>{start_comment}    BEGIN(comment0);
<comment0>"*"               BEGIN(comment1);
<comment0>[^*]              ;
<comment1>"*"               ;
<comment1>"/"               BEGIN(INITIAL);
<comment1>[^*/]             BEGIN(comment0);

{intcon}                    return(INTCON);
{floatcon}                  return(FLOATCON);
  ...
%%
  ...
```


Yacc: A Parser Generator

- Takes a specification for a CFG, produces an LALR parser.



- Form of a yacc specification file:

  Declarations

  %%

  Grammar rules

  %%

  Programs

  red : required
  blue : optional

Yacc: Grammar Rules

Terminals (**tokens**) : Names must be declared:

`%token` $name_1$ $name_2$ ...

Any name not declared as a token in the declarations section is assumed to be a nonterminal.

Start symbol :

- may be declared, via: `%start` *name*
- if not declared explicitly, defaults to the nonterminal on the LHS of the first grammar rule listed.

Productions : A grammar production $A \to B_1 B_2 \cdots B_n$ is written as

$A : B_1 B_2 \cdots B_n$;

Note: Left-recursion is preferred to right-recursion for efficiency reasons.

Example:

```
stmt : KEYWD_IF '(' expr ')' stmt ;
```

Communication between Scanner and Parser

- The user must supply an integer-valued function `yylex()` that implements the lexical analyzer (scanner).
- If there is a value associated with the token, it should be assigned to the external variable `yylval`.
- The token `error` is reserved for error handling.
- *Token numbers* : These may be chosen by the user if desired. The default is:
  - chosen by yacc [in a file `y.tab.h`]
  - the token no. for a literal is its ASCII value
  - other tokens are assigned numbers starting at 257
  - the endmarker must have a number zero or negative.
- Generate `y.tab.h` using '`yacc -d`'

Using Yacc

- 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.

Conflicts and Ambiguities

- Conflicts may be either *shift/reduce* or *reduce/reduce*:
  - In a shift/reduce conflict, the default is to shift.
  - In a reduce/reduce conflict, the default is to *reduce using the first applicable grammar rule*.
- *Arithmetic Operators* : associativity and precedence can be specified:

  Associativity: use `%left`, `%right`, `%nonassoc`

  Precedence (Binary Operators):
  - Specify associativity using `%left` etc.
  - Operators within a group have same precedence. Between groups, precedence increases going down.

  Precedence (Unary Operators): use `%prec` keyword. This changes the precedence of a rule to be that of the following token.

  Example:

```
%left '+' '-'
%left '*' '/'
    ...
expr : expr '+' expr
     | expr '*' expr
     | '-' expr       %prec '*'
     | ID
```


Yacc: Error Handling

- The token `error` is reserved for error handling. This can be used in grammar rules, to indicate where error might occur and recovery take place.
- *When an error is detected* :
  - If an `error` token is specified, the parser pops its stack until it finds a state where the `error` token is legal.

    It then behaves as if `error` is the current lookahead token, and performs the action encountered.
  - If there is no rule using the `error` token, processing halts when an error is encountered.
- To prevent cascading error messages, the parser remains in an *error state* after detecting an error until 3 tokens have been successfully read and shifted.

  If an error is encountered when in the error state, no error message is given, and the input token is discarded.

Yacc Error Handling: (cont'd)

- A rule of the form

  ```
  stmt :  error
  ```

  means that on syntax error, the parser would attempt to skip over the offending statement, looking for 3 tokens that can legally follow `stmt`.

- A rule of the form

  ```
  stmt :  error ';'
  ```

  causes the parser to skip to the next `';'` after `stmt`: all intervening tokens are deleted.

- Actions may be associated with these special error rules: these might attempt to (re)initialize tables, reclaim space, turn off code generation, etc.

Adding `error` symbols

Their placement is guided by the following (conflicting!) goals:

- as close as possible to the start symbol of the grammar

  (to allow recovery without discarding the entire program)

- as close as possible to each terminal symbol

  (to allow only a small amount of input to be discarded on an error)

- without introducing conflicts

  (this may be difficult;
  shift/reduce conflicts may be acceptable if they serve to lengthen strings, i.e., delay reporting of errors)

Error Messages

The use should provide a function `yyerror()` that is called when a syntax error is detected:

```
yyerror(s)
char *s;   /* s: a string containing an error msg */
{          /* usually "syntax error" */
  ...
}
```

More informative error messages :

– line no. in source program : `yylineno`
– token no. causing error : `yychar`

Example :

```
extern int yylineno, yychar;
yyerror(s)
char *s;
{
  fprintf(stderr,
          "%s: token %d on line %d\n",
             /*      ^^ Ugh: internal token no.?!? */
          s, yychar, yylineno);
}
```


Controlling error actions

Sometimes we may want to stop discarding tokens, if a certain (synchronizing) token is seen: for this, attach an action `{yyerrok;}`

Example :

```
id_list :  id_list ',' ID { yyerrok; }
        |  ID
        |  error
```

Special-purpose error handling:

– set a global flag to indicate the problem;

– use this flag in `yyerror()` to give better error messages.

Example :

```
compd_stmt : '{' stmt_list '}'
     | '{' stmt_list error   {errno = NO_RBRACE;}
     | '{' error '}'
   ...
yyerror(s)
{
   if (errno == NO_RBRACE) printf("missing }\n");
   else ...
}
```

Adding Semantic Actions to Yacc Specifications

Semantic actions may be associated with each rule.

An action is specified by one or more statements, enclosed in braces { }.

Examples :

```
ident_decl  :  ID    {install_symtab(id_name);}

type_decl  :  type { tval = ...} id_list
```

Note: Actions may occur inside RHS of a production.

(Synthesized) Attributes for Nonterminals

Each nonterminal can return a value:

- To access the value returned by the i^{th} symbol in the body of a rule, use $i.

  If an action occurs in the middle of a rule, it is counted as a "symbol" that can return a value.

- To set the value to be returned by a rule, assign to $$.

  By default, the value of a rule is the value of the first element in it ($1).

Example :

```
func_defn :  1 ID          2 {install_symtab(id_name);}
             3 '('         4 {scope = LOCAL;}
             5 formals
             6 ')'
             7 parm_types  8 {install_formals($5, $7);}
             9 '{'
            10 body
            11 '}'        12 {scope = GLOBAL; symtab_cleanup();}
```


Example

```
var_decl
   : identifier opt_dimension
       { if ( Seen($1) > 0 ) {
            errmsg(MULTI_DECL, Ident($1));
         }
         else {
            Seen($1) = DECL;
            if ($2 < 0) {
               BaseType($1) = tval;
            }
            else {
               BaseType($1) = ARRAY;
               ArrayEltType($1) = tval;
               ArrayDim($1) = $2;
            }
         }
       }
   ;

opt_dimension
   : '[' INTCON ']'    { $$ = Value(yylval); }
   |   /* epsilon */   { $$ = -1; }
   ;
```


Synthesized Attributes: Types

By default, values returned by actions and the lexical analyzer are integers.

In general, actions may need to return values of other types, e.g., pointers to the symbol table or syntax tree nodes. For this, we need to:

1. Declare the union of the various kinds of values that may be returned. E.g.:

```
%union {
   symtab_ptr    st_ptr;
   id_list_ptr   list_of_ids;
   tree_node     st_node;
   int           value;
}
```

2. Specify which union member a particular nonterminal will return:

```
%token <value> INTCON, CHARCON;      } terminals

%type <st_ptr> identifier;           } nonterminals
%type <list_of_ids> formals;
```

Example

```
%union {
   st_rec_ptr          st_ptr;
   id_list_ptr         list_of_ids;
   expr_node_ptr       expr;
   stmt_node_ptr       stmt;
   int                 value;
}

%token ID;                                 /* identifiers */
%token <value> INTCON CHARCON              /* constants */
%token CHAR INT VOID                       /* types */
...
%left                  AND OR
%nonassoc              LE GE EQ NEQ '<' '>'
%left                  '+' '-'
...
%type <value>          opt_dimension, type
%type <st_ptr>         identifier;
%type <list_of_ids>    formals
...

%start prog

%{
st_rec_ptr             id_ptr;         /* globals */
id_list_ptr            fparms;
...
%}
%%
...
```