Special Topic

Lex and Yacc

資料結構與程式設計

Data Structure and Programming

11/30/2016

Lex and Yacc, why should I care/learn?

◆ To parse a text file written in certain formal language, e.g.:
- AIGER format (HW#6)
- C/C++, Verilog, HTML,… etc.

◆ To practice how to define a formal language, e.g.:
- Interface language for certain (web/tool) service

Language, what is a computer language?

When do we need to define and use it?

Scripting language vs. programming language

Examples of “defining languages”

- Data analytics
  - SQL: database query language
  - SAS: language for statistical analysis
  - R: statistical computing and graphics
- (IC) Design verification
  - Aiger: And-Inverter-Graph
  - e, cbv, sugar, SystemVerilog, etc.
- Graphics/Multimedia
  - OpenGL
  - Maya Embedded Language

Example: Command Line Calculator

- Features:
  - Formula in one line. “Enter” for the answer
  - Understand integers (positive and negative)
    - Let’s not worry about integer overfloat at this moment
  - Operators: (), +, -, *, /
  - Precedence: () then *, / then +, -
    - Same precedence: Left to right
  - Ignore “white space”
  - (optional) Support floating number
- How to write this calculator in C++?

A “Parser” Way…

- To handle all the possible operations of the calculator, first we need to know ---
  - What are all the possible expressions?
  - What are the syntax rules?
  - How to formally define these syntax rules?
  - By what “tokens” can we represent these rules?
  - What are the atomic “tokens” of the expressions?
  - How to identify the syntax tokens from an expression?

Terminologies

- Parsing
  - Read and analyze a text file (usually a program) and transform it into an internal representation (data structure)
- Lexical analysis
  - Take a stream of characters as its input, and break it up into meaningful units, or tokens
    - [e.g.] a = b + 3;

      ➔ VAR_NAME EQ_OP VAR_NAME PLUS_OP INT_CONST SEMICOL

      // What about white spaces?
- Syntactical analysis
  - Take a stream of tokens as its input, and check if it follows the predefined syntactical rules
    - [e.g.] BOOL_EXPR : BOOL_TERM

      | BOOL_EXPR BOOL_OP BOOL_TERM

Let’s draft on the “syntax” of the calculator

- Hierarchy of “productions”
  - The end production should be “expression”
  - An expression is formed by operations on expression(s) and term(s)
  - A term is an expression enclosed by (), or a number
- Recognized tokens
  - Operators
  - Braces ()
  - Number
  - White space
  - New line (enter)

Lex and Yacc

- Lex: a lexical analyzer
  - lex file (.l suffix): specify the token analysis rules and corresponding actions
  - lex/flex: a program to generate a C/C++ program (lex.yy.c) from a lex (*.l) file
- Yacc: Yet Another Compiler's Compiler
  - yacc file (.y suffix): based on predefined tokens, specify the syntactical rules and the corresponding actions
    - Some of the tokens can come from the lex output
  - yacc/bison: a program to generate a C/C++ program (y.tab.c) from an yacc (*.y) file

Let's download "cal.tgz" from Ceiba !

Skeleton of a lex specification (.l file)

%{

< C global variables, prototypes, comments > → This part will be embedded into *.c

%}

[DEFINITION SECTION] → define how the scanned characters are mapped to tokens by regular expression

%%

[RULES SECTION] → define how to scan and what action to take for each token

%%

< C auxiliary subroutines> → any user code. For example, a main function to call the scanning function yylex().

A Lex File Example

```
%{
/* a Lex program that adds line numbers to lines of text
   printing the new text to standard output */
#include <iostream>
#include <iomanip>
using namespace std;
static int lineno =1;
%}
LINE .*\n
%%
{LINE} { cout << setw(5) << lineno++ << " " << yytext; }
%%
int main() {
   cout << "Now processing from standard input \n";
   yylex();
   return 0;
}
```

A Lex File Example (Compile and Execution)

// Compile the lex file
// "-o" to specify output name

```
> lex -o lineNo.cpp
  lineNo.l
```

// Compile the generated C++ file
// Need the 'l' library

```
> g++ -o lineNo
  lineNo.cpp -ll

> lineNo < lineNo.l
```

```
Now processing from...
    1 %{
    2 /* a Lex program
    3    printing the
    4 */
    5
    6 #include <iostream>
    7 #include <iomanip>
    8 using namespace std;
    9 static int lineno =1;
   10 %}
   11
   ...
```

The Definition Section

```
%{
< C global variables,
prototypes, comments >
%}

[DEFINITION SECTION]

%%
[RULES SECTION]
%%
< C auxiliary subroutines>
```

```
%}
<TOKEN>    <pattern>
%%
```

- TOKEN:
  - A named constant for the C/C++ program (cf. #define, enum)
- pattern
  - Follow regular expression

Regular Expression Basics

. : matches any single character except \n

* : matches 0 or more instances of the preceding regular expression

+ : matches 1 or more instances of the preceding regular expression

? : matches 0 or 1 of the preceding regular expression

| : matches the preceding or following regular expression

[ ] : defines a character class (e.g. [a-zA-Z])

( ) : groups enclosed regular expression into a new regular expression

"…": matches everything within the " " literally

Regular Expression Examples

- a natural number: e.g. 12345

  [1-9][0-9]*
- a word: e.g. cat

  [a-zA-Z]+
- a C/C++ variable: e.g. _name38

  [_a-zA-Z][_0-9a-zA-Z]*
- a (possibly) signed integer: 12345 or -12345

  [-+]?[1-9][0-9]*
- a floating point number: 1.2345

  [0-9]*"."[0-9]+

More on Lex Reg Exp

x|**y**	**x** or **y**
{**TOK**}	definition of **TOK**
x/**y**	**x**, only if followed by **y** (**y** not removed from input)
x{*m*,*n*}	*m* to *n* occurrences of **x**
^x	**x**, but only at beginning of line
x$	**x**, but only at end of line
"**s**"	exactly what is in the quotes (except for "\" and following character)

A regular expression finishes with a space, tab or newline

Meta-characters

- meta-characters (do not match themselves, because they are used in the preceding reg exps):
  - ( ) [ ] { } < > + / , ^ * | . \ " $? - %
- to match a meta-character, prefix with "\"
- to match a backslash, tab or newline, use \\, \t, or \n

Into definition section…

```
%}
DIGIT       [0-9]
NUM         [1-9]
SIGNEDINT   [-+]?{NUM}{DIGIT}*
FLOAT       {DIGIT}*"."{DIGIT}+
ALPHABET    [_a-zA-Z]
WORD        {ALPHABET}+
VARIABLE    {ALPHABET}({DIGIT}|{ALPHABET})*
%%
```

The rules section

```
%%
<pattern>      { <action to take when matched> }      or
{<TOKEN>}      { <action to take when matched> }
...
%%
```

```
%{
< C global variables,
prototypes, comments >
%}

[DEFINITION SECTION]

%%
[RULES SECTION]
%%
< C auxiliary subroutines>
```

- Patterns are specified by *regular expressions*.

For example:

```
%%
[A-Za-z]*     { cout << "this is a word"; }
{VARIABLE}    { cout << "this is a variable"; }
%%
```

State in Lexical Matching

- Sometimes pattern rules may depend on context.
  - Similar tokens may have different meanings in different contexts
- Left state
  - %s STATE   // to define states in "definition section"
  - <STATE> pattern_rule // to define pattern rules in "rules section" { some action; BEGIN OTHERSTATE; }
  - Pattern rule is applicable only under STATE
  - The (implicit) initial state of lex is INITIAL
- Right state
  - pattern1/pattern2 { some action; }
  - Pattern1 is applicable only when followed by pattern2
  - Pattern2 is NOT popped out from input string

  ➔ cf: pattern1 pattern2 { yyless(n); … }

Note: “ECHO” is a macro that writes text matched by the pattern.

Example: to clean up messily spacing text

```
punct [,.;:!?]
text [a-zA-Z]
%s OPEN
%s CLOSE
%s TEXT
%s PUNCT
%%
" "+ ;
<INITIAL>"("
{ ECHO; BEGIN OPEN; }
<TEXT>"(" { printf(" ");
  ECHO; BEGIN OPEN; }
<PUNCT>"(" { printf(" ");
  ECHO; BEGIN OPEN; }
")" { ECHO ; BEGIN CLOSE;}
<INITIAL>{text}+
{ ECHO; BEGIN TEXT; }
<OPEN>{text}+
{ ECHO; BEGIN TEXT; }
<CLOSE>{text}+ { printf(" ");
  ECHO; BEGIN TEXT;}
<TEXT>{text}+ { printf(" ");
  ECHO; BEGIN TEXT;}
<PUNCT>{text}+ { printf(" ");
  ECHO; BEGIN TEXT;}
{punct}+ { ECHO; BEGIN PUNCT;}
\n { ECHO; BEGIN INITIAL; }
%%
```

Remember...

```
%{
< C global variables,
  prototypes, comments >
%}

[DEFINITION SECTION] ←———      Define —
                               <TOKEN>  <pattern>

%%
[RULES SECTION] ←———           <pattern>  {<actions>}
%%                             can also be —
< C auxiliary subroutines >    {<TOKEN>}  {<actions>}
```

Example: A circuit parser

```
%{
...
int cirlineno=1;
...
}%
WS              [ \t]+
DIGIT           [0-9]
ALPHABET        [a-zA-Z]
SYMBOL          [_.$]
BRLEFT          [(]
BRRIGHT         [)]
SQLEFT          [\[]
SQRIGHT         [\]]
```

```
RANGE
   {BRLEFT}{DIGIT}+{BRRIGHT}|
   {SQLEFT}{DIGIT}+{SQRIGHT}
IDCHAR
   {DIGIT}|{ALPHABET}|{SYMBOL
   }|{RANGE}|[\\]{WS}
IDENTIFIER  {IDCHAR}+
%%
\.cir    return CIR;
...
{WS}     /* ignore
            whitespace */;
\n       cirlineno++;
.        { cerr << "Error:...
%%
```

Revisited: The Lex File Example

```
%{
/* a Lex program that adds line numbers to lines of text
   printing the new text to standard output */
#include <iostream>
#include <iomanip>
using namespace std;
static int lineno =1;
%}
LINE .*\n
%%
{LINE} { cout << setw(5) << lineno++ << " " << yytext ;
%%
int main() {
   cout << "Now processing from standard input \n";
   yylex();
   return 0;
}
```

What are they?

Lex predefined variables and functions

Name	Function
int yylex(void)	call to invoke lexer, returns token
FILE *yyin	input file
FILE *yyout	output file
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
INITIAL	initial start condition
BEGIN	to start a state
ECHO	write matched string

Lex predefined variables and functions

Name	Function
int yywrap(void)	May be replaced by user Called by the lexical analyzer whenever it inputs an EOF as the first character when trying to match a regular expression Return 1 if done, 0 if not done (to parse another file)
yymore()	append next string matched to current contents of yytext
yyless(n)	remove from yytext all but the first n characters
unput(c)	return character c to input stream

These predefined variables and functions play very important roles in linking lex and yacc.

Before we go onto yacc,
let's write the lex part of the calculator

What tokens are needed in calculator

- Sample inputs
  - 2+3-4*5
  - (2+3)*4
  - 2 +3   -4
  - 02+03
  - -2*-3
- Tokens to define
  - DIGIT
  - NUMBER
  - Operators & parenthesis
  - White space

We have talked about lex.

Let's learn yacc now.

Skeleton of a yacc specification (.y file)

%{

< C global variables, prototypes, comments > $\longrightarrow$ This part will be embedded into *.c

%}

[DEFINITION SECTION] $\longrightarrow$ Contains token declarations. Tokens are recognized in lexer.

%%

[PRODUCTION RULES SECTION] $\longrightarrow$ Define how to "understand" the input language, and what actions to take for each "sentence".

%%

< C auxiliary subroutines> $\longrightarrow$ Any user code. For example, a main function to call the parser function yyparse()

The Production Rules Section

```
%%
productionX : symbol1 symbol2 …  { action }
            | productionY { action } symbol3 …
            |   …
%%
```

➔ “Production” rule is composed of previously-defined productions and/or lex-defined symbols (tokens)

➔ “actions” can be insert anywhere between productions/symbols

➔ The rules of productions form a “syntax tree, where the leaf nodes are symbols/tokens

An example of yacc rules

```
%%
statement : expression  { cout << “= ” << $1 << “\n”; }
expression : expression ‘+’ expression { $$ = $1 + $3; }
           | expression ‘-’ expression { $$ = $1 - $3; }
           | NUMBER  { $$ = $1; }
%%
```

left-hand-side (the production)

first token of the rule

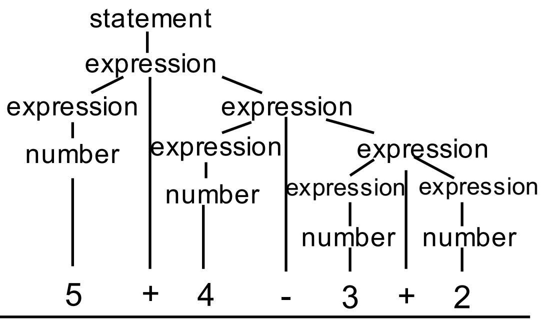
third token of the rule

An example of yacc rules

```
%%
statement : expression  { cout << "= " << $1 << "\n"; }
expression : expression '+' expression { $$ = $1 + $3; }
           | expression '-' expression { $$ = $1 - $3; }
           | NUMBER  { $$ = $1; }
%%
```

According to these two productions,

5 + 4 – 3 + 2
is parsed into:



yacc : a shift and reduce process

- Implemented by a "stack"
  - Tokens on the left of the pointer '.' have been shifted into the stack, while tokens on the right are remaining inputs
  - The top tokens in the stack can be reduced by the rules

-
  ```
  . 5 + 4 - 3 + 2
  5 . + 4 - 3 + 2      // shift
  E . + 4 - 3 + 2      // reduce
  E + 4 . - 3 + 2      // shift, shift
  E + E . - 3 + 2      // reduce
  E . - 3 + 2          // reduce
  E - E . + 2          // shift, shift, reduce
  E . + 2              // reduce
  E + E .              // shift, shift, reduce
  E .                         // reduce
  S .                         // reduce
  (E: expression; S: statement)
  ```

Another example of yacc rules

◆
```
statement  : expression
expression : expression '+' expression
           | expression '*' expression
           | IDENTIFIER
```

◆ Consider the following statement : x + y * z
- Will it performs ---
  (x + y) * z ?
  Or x + (y * z) ?

shift-reduce, reduce-reduce conflicts

◆ Note: yacc allows no ambiguity

◆ If the rules allow more than one possibilities in shift/reduce operations, there is a conflict.
- In the previous example, there is a shift-reduce conflict
- The following rules have a reduce-reduce conflict

```
expression : word
           | word IDENTIFIER
word       : IDENTIFIER
```

shift-reduce, reduce-reduce conflicts

- Since yacc allows no ambiguity, it takes a default action when there is a conflict:
  - For shift-reduce, yacc will shift.
  - For reduce-reduce, yacc will use the first rule in the listing.
- Although these conflicts may not affect the correctness of your parser, you should always try to remove them.

1. Make the grammar unambiguous
2. Supply with directives to indicate which operator has precedence

Precedences and Associativities

- Associativity: define the order on a series of operations
  - e.g. "a + b + c" should be "(a + b) + c" or "a + (b + c)"?
  - yacc keywords: left, right, nonasoc
    - Declared in the "definition section"
  - Common example:
    ```
    % right '='
    % left '+' '-'
    % left '*' '/'
    ```
  - Example of "nonasoc"
    `% nonasoc NEQ` ← That is, "a != b != c" is illegal
    (Why not `%nonasoc '!='` or `%nonasoc "!="` ?)
- Precedence
  - All of the tokens on the same line are assumed to have the same precedence level and associativity
  - The lines of associativity are listed in order of increasing precedence or binding strength

  → In the previous example, '+' and '-' have the same precedence and associativity, but '*' and '/' have the higher precedence than '+' and '-'

Remember... Skeleton of a Yacc File

```
%{
< C global variables, prototypes, comments >
%}
[DEFINITION SECTION]  ————→
%%
[PRODUCTION RULES SECTION]
production : symbol1 symbol2 …
             { action }
           |  symbol3 symbol4 …
             { action }
           |  …
%%
< C auxiliary subroutines>
```

Contains token declarations. Tokens are recognized in lexer.

How the tokens recognized in lex be used in yacc?

1. In a rule of a lex file, we return a token. The matched string is stored in char* yytext. And we can store a value in yylval.
   - **// in the rule section of a lex file**

```
[0-9]+ {
    yylval = atoi(yytext);
    return INTEGER;
}
```

How can the tokens recognized in lex be used in yacc?

2. In the yacc file, the token needs to be defined in "definition section".
   - `%token INTEGER`

   ➔ which is translated into the header file y.tab.h as:

   -
     ```
     #ifndef YYSTYPE
     #define YYSTYPE int
     #endif
     #define INTEGER 258
     extern YYSTYPE yylval;
     ```

   ➔ where YYSTYPE defines the type of yylval (default is integer)

   ➔ The value of generated token is greater than 256, as the values 0 ~ 255 are reserved for characters.
   - `e.g. [-+] return *yytext;`

How can the tokens recognized in lex be used in yacc?

3. And tokens are then used in production rules
   - e.g.
     ```
     expression: expression '+' expression
                    { $$ = $1 + $3; }
                  | expression '-' expression
                    { $$ = $1 - $3; }
                  | INTEGER { $$ = $1; }
     ```

Putting token definitions and rules together

The lex file

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "simple.tab.h"
%}
%%
[0-9]+ {
  yylval = atoi(yytext);
  return INTEGER;
}
```

```
[-+\n] return *yytext;
[ \t]   ;/*skip whitespace*/
.       yyerror
       ("invalid character");
%%
int yywrap(void) {
  return 1;
}
```

Putting token definitions and rules together

The yacc file

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token INTEGER
%%
program: program expr
  '\n' { printf("%d\n",
         $2); }
|
;
```

```
expr:INTEGER { $$ = $1; }
   | expr '+' expr
     { $$ = $1 + $3; }
   | expr '-' expr
     { $$ = $1 - $3; }
;
%%
void yyerror(char *s) {
  fprintf(stderr, "%s\n",s);
}
int main(void) {
  yyparse();
  return 0;
}
```

What if we want to return something other than integers?

- In the lex file:
  - `static string tempStr;  // as global var`

  -
    ```
    {IDENTIFIER}  tempStr = yytext;
                  yylval.sv = &tempStr;
                  return IDENTIFIER;
    ```
- In the yacc file:

  -
    ```
    %token CIR INPUT OUTPUT INV AND
    %union
    {
        int          iv;
        std::string* sv;
    };
    %token <sv> IDENTIFIER
    ```

Why not “string”?

Summary about token (1/4)

- Recognized pattern in lex is return as a token, which is then understood in yacc
  - In lex file,
    <pattern> { ….; return SOME_TOK; }
  - In yacc file,
    // definition section
    %token SOME_TOK
    // rule section
    PROD: …. SOME_TOK

Summary about token (2/4)

- Single character tokens are predefined. You can pass characters as tokens.
  - In lex file,
    “+” { ….; return ‘+’; }
  - In yacc file,
    // no need to define token in definition section
    // rule section
    PROD: …. ‘+’ ....

Summary about token (3/4)

- The default type of token is “integer”. If there are other types of token, use “union” to define the type of tokens.
  - The value of the return token is stored in “yylval”.
  - In lex file,
    <pattern> { ….; yylval.sv = &str; return WORD; }
  - In yacc file,

```
// definition section
%union {
   std::string* sv;
};
%token <sv> WORD
// rule section
PROD: …. WORD { cout << *($3) << endl; }
```

Summary about token (4/4)

- Sometimes it is necessary to assign value to the production (of a rule). In that case, you need to use “%type” to define the type of the production.
  - This has nothing to do with lex file.
  - In yacc file,
    ```
    // definition section
    %union {
       std::string* sv;
    };
    %type <sv> COMBWORD
    // rule section
    COMBWORD: WORD ‘+’ WORD { $$ = $1 + $3; }
    ```
  - Type specifier (e.g. <sv>, <nv>) should match something in “union”

Concept: token or token <iv>

- What’s the difference?

  -
    ```
    %token NUMBER
    %type <iv> TERM
    …
    TERM: NUMBER { $$ = $1; }
    ```

  -
    ```
    %token <iv> NUMBER
    %type <iv> TERM
    …
    TERM: NUMBER { $$ = $1; }
    ```
- Which one is correct?

Multiple lex or yacc files

- Sometimes in our program we have multiple files to parse.
  - yylex, yyparse, yytext… which yy?
- Use different pre-fixes to distinguish
  - e.g. cirlex, liblex, etc
  - For lex: lex -P<prefix>...
  - For yacc: yacc -p <prefix>...

Can the production rules pass class objects (pointers)?

```
%union
{
   int               iv;
   std::string*      sv;
   class SynNode*    nv
};
%token <sv> IDENTIFIER
%type <nv> NUM_EXPR NUM_TERM NUM_OP
%%
NUM_EXPR : NUM_TERM { $$ = $1; }
       | NUM_EXPR NUM_OP NUM_TERM
       { $2->addChildren($1, $3); $$ = $2; }
NUM_TERM : IDENTIFIER { $$ = new TermNode(*($1)); }
       | INTEGER { $$ = new ConstNode($1); }
NUM_OP : '+' { $$ = new OpNode(ADD_OP); }
       | '-' …
```

How to call lex/yacc in C++ code?

1. Calling "yyparse()" in your circuit reader member funciton

   Example:

```
CirMgr::readCircuit(const string& fileName)
{
   extern FILE* yyin;
   extern int yyparse();
   extern int parseErrorCount;
   if (!(yyin = fopen(fileName.c_str(),
         "r")))
   ...
   if (yyparse() != 0 ||
       parseErrorCount != 0) return false;
   fclose(yyin);
}
```

   [Note] stdin is used if yyin is not specified

How to call lex/yacc in C++ code?

2. Modify your yacc (.y) file
   - Make the variables defined in lex "extern", if you want to use it in yacc
   - If there is any (global) variable that is defined outside the yacc file, add "extern" (e.g. extern CirMgr* cirMgr).
   - Declare external function prototypes
     - e.g. int yylex(void);
   - Implement your own "void yyerror(const char *str) { ... }"
   - (FYI) some of the errors can be detected by the production rules (and output the message by yyerror()), but some of them can only be detected by the lex/yacc internal predefined token "error"

How to use lex/yacc in C++?

3. Modify your lex (.l) file
   - Declare some global variables if necessary, e.g. yylineno.
   - Declare some external functions:
     - e.g. extern "C" { int cirwrap() { return 1; } }

     ➔ extern "C" to make C functions visible in C++ files
   - [Note] The matched string in lex (i.e. yytext) is temporary. Be sure to allocate memory and copy before sending it to yacc, or construct a string for it. However, the token type "union {}" in yacc cannot take class object (i.e memory size undefined), so we need to use "string *" (See example in page 38)

How to use lex/yacc in C++?

4. Modify makefile
   - For example, add these lines to the "Makefile" under the "src/cir" directory

```
LEX       = flex
YACC      = bison
LEX_FLAG  = -Pcir
YACC_FLAG = -d -p cir

cirLex.cpp: cirLex.l
        @echo "> lexing: $<"
        @$(LEX) $(LEX_FLAG) -o $@ $<

cirParse.cpp: cirParse.y
        @echo "> yaccing: $<"
        @$(YACC) $(YACC_FLAG) -o $@ $<
```

➔ The -P and -p flags in lex and yacc change the "yy" prefixes to your specified one.

➔ -d create a separate header file (with the same base file name as the generated cpp file)

Let’s work on the yacc file
of the calculator example.

What production rules to define

- Rules (top-down)
  - Answer
  - Expression // when entered, get the answer
  - Types of expression
    - Sum and Product
    - Who should be defined on top?
  - Term
    - What about parenthesis?
    - What about positive/negative sign?
  - NUMBER (as leaves)

References

1. “Lex and YACC primer/HOWTO”, http://ds9a.nl/lex-yacc/cvs/lex-yacc-howto.html
2. “Example Lex Files”, http://myweb.stedwards.edu/laurab/cosc4342/lex-examples.html
3. “Lex and Yacc Tutorial”, http://203.208.166.84/dtanvirahmed/cse309N/LexYacc.ppt
4. “Lex and Yacc Tutorial”, http://epaperpress.com/lexandyacc/download/lexyacc.pdf
5. “Parsing with Yacc”, http://uw714doc.sco.com/en/SDK_tools/Parsyacc.html