Yacc
a tool for Syntactic Analysis

CS 315 – Programming Languages

Pinar Duygulu

Bilkent University

Lex and Yacc

1) Lexical Analysis:
Lexical analyzer: scans the input stream and converts sequences of ***characters into tokens.***
Lex is a tool for writing lexical analyzers.

2) Syntactic Analysis (Parsing):
Parser: reads tokens and assembles them into language constructs using the grammar rules of the language.
Yacc is a tool for constructing parsers.

Yet Another Compiler to Compiler
Reads a specification file that codifies the grammar of a language and generates a parsing routine

Yacc

- Yacc specification describes a Context Free Grammar (CFG), that can be used to generate a parser.

Elements of a CFG:

1. Terminals: tokens and literal characters,
2. Variables (nonterminals): syntactical elements,
3. Production rules, and
4. Start rule.

Yacc

Format of a production rule:

```
symbol: definition
{action}
;
```

Example:

A → Bc is written in yacc as `a: b 'c';`

Yacc

Format of a yacc specification file:

```
declarations
%%
grammar rules and associated actions
%%
C programs
```

Declarations

To define tokens and their characteristics

`%token:` declare names of tokens
`%left:` define left-associative operators
`%right:` define right-associative operators
`%nonassoc:` define operators that may not associate with themselves
`%type:` declare the type of variables
`%union:` declare multiple data types for semantic values
`%start:` declare the start symbol (default is the first variable in rules)
`%prec:` assign precedence to a rule
`%{`
`C declarations` directly copied to the resulting C program
`%}` (E.g., variables, types, macros…)

A simple yacc specification to accept L={ *anbn* | *n>1*}.

```
/*anbn0.y */
%token A B
%%
start: anbn '\n' {return 0;}
anbn: A B
| A anbn B
;
%%
#include "lex.yy.c"
```

Lex – yacc pair

```
/* anbn0.l */
%%
a return (A);
b return (B);
. return (yytext[0]);
\n return ('\n');

/*anbn0.y */
%token A B
%%
start: anbn '\n' {return 0;}
anbn: A B
| A anbn B
;
%%
#include "lex.yy.c"
```

Running yacc on linux systems

In linux there is no liby.a library for yacc functions

You have to add the following lines to end of your yacc specification file

```
int yyerror(char *s)
{
 printf("%s\n", s);
}
int main(void)
{
 yyparse()
}
```

Then type

```
gcc -o exe_file y.tab.c -lfl
```

Printing messages

If the input stream does not match `start`, the default message of "`syntax error`" is printed and program terminates.
However, customized error messages can be generated.

```
/*anbn1.y */
%token A B
%%
start: anbn '\n' {printf(" is in anbn\n");
return 0;}
anbn: A B
| A anbn B
;
%%
#include "lex.yy.c"
yyerror(s)
char *s;
{ printf("%s, it is not in anbn\n", s);
}
```

Example output

```
$anbn
aabb
is in anbn
$anbn
acadbefbg
Syntax error, it is not in anbn
$
```

A grammar to accept L = {anbn | n >= 0}.

```
/*anbn_0.y */
%token A B
%%
start: anbn '\n' {printf(" is in
  anbn_0\n");
return 0;}
anbn: empty
| A anbn B
;
empty: ;
%%
#include "lex.yy.c"
yyerror(s)
char *s;
{ printf("%s, it is not in anbn_0\n", s);
```

Positional assignment of values for items

- $$: left-hand side
- $1: first item in the right-hand side
- $*n*: *n*th item in the right-hand side

Example : printing integers

```
/*print-int.l*/
%%
[0-9]+ {sscanf(yytext, "%d", &yylval);
return(INTEGER);
}
\n return(NEWLINE);
. return(yytext[0]);

/* print-int.y */
%token INTEGER NEWLINE
%%
lines: /* empty */
| lines NEWLINE
| lines line NEWLINE {printf("=%d\n", $2);}
| error NEWLINE {yyerror("Reenter:"); yyerrok;}
;
line: INTEGER {$$ = $1;}
;
%%
#include "lex.yy.c"
```

Execution

```
$print-int
7
=7
007
=7
zippy
syntax error
Reenter:

_
```

- NOTE: Although right-recursive rules can be used in yacc, left-recursive rules are preferred, and, in general, generate more efficient parsers.

yylval

- The type of `yylval` is `int` by default. To change the type of `yylval` use macro `YYSTYPE` in the declarations section of a yacc specifications file.

```
%{
#define YYSTYPE double
%}
```

If there are more than one data types for token values, `yylval` is declared as a `union`.

Example with three possible types for yylval:

```
%union{
double real; /* real value */
int integer; /* integer value */
char str[30]; /* string value */
}
```

yylval

Example:

`yytext = "0012",` type of `yylval: int`, value of `yylval: 12`

`yytext = "+1.70",` type of `yylval: float`, value of `yylval: 1.7`

The type of associated values of tokens can be specified by `%token` as

```
%token <real> REAL
%token <integer> INTEGER
%token <str> IDENTIFIER STRING
```

Type of variables can be defined by `%type` as

```
%type <real> real-expr
%type <integer> integer-expr
```

To return values for tokens from a lexical analyzer:

```
/* lexical-analyzer.l */
alphabetic [A-Za-z]
digit [0-9]
alphanumeric ({alphabetic}|{digit})
[+-]?{digit}*(\.)?{digit}+ {sscanf(yytext,
  %lf", &yylval.real);
return REAL;
}
{alphabetic}{alphanumeric}*
  {strcpy(yylval.str, yytext);
return IDENTIFIER;
}
```

Example: simple calculator

```
/* calculator.l */
integer       [0-9]+
dreal         ([0-9]*\.[0-9]+)
ereal         ([0-9]*\.[0-9]+[Ee][+-]?[0-9]+)
real          {dreal}|{ereal}
nl            \n
%%
[ \t]         ;
{integer}     { sscanf(yytext, "%d", &yylval.integer);
                return INTEGER;
              }
{real}        { sscanf(yytext, "%lf", &yylval.real);
                return REAL;
              }
\+            { return PLUS;}
\-            { return MINUS;}
\*            { return TIMES;}
\/            { return DIVIDE;}
\(            { return LP;}
\)            { return RP;}
{nl}          { extern int lineno; lineno++;
                return NL;
              }
.             { return yytext[0]; }
%%
int yywrap() { return 1; }
```

Example: simple calculator

```
/* calculator.y */

%{
#include <stdio.h>
%}

%union{ double   real; /* real value */
        int   integer; /* integer value */
      }

%token <real> REAL
%token <integer> INTEGER
%token PLUS MINUS TIMES DIVIDE LP RP NL

%type <real> rexpr
%type <integer> iexpr

%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS
```

Example: simple calculator

```
%%
lines: /* nothing */
     | lines line
     ;
line:  NL
     | iexpr NL
       { printf("%d) %d\n", lineno, $1);}
     | rexpr NL
       { printf("%d) %15.8lf\n", lineno, $1);}
     ;
iexpr: INTEGER
     | iexpr PLUS iexpr
       { $$ = $1 + $3;}
     | iexpr MINUS iexpr
       { $$ = $1 - $3;}
     | iexpr TIMES iexpr
       { $$ = $1 * $3;}
     | iexpr DIVIDE iexpr
       { if($3) $$ = $1 / $3;
         else { yyerror("divide by zero");      }
       }
     | MINUS iexpr %prec UMINUS
       { $$ = - $2;}

     | LP iexpr RP
       { $$ = $2;}
     ;
```

Example: simple calculator

```
rexpr: REAL
     | rexpr PLUS rexpr
       { $$ = $1 + $3;}
     | rexpr MINUS rexpr
       { $$ = $1 - $3;}
     | rexpr TIMES rexpr
       { $$ = $1 * $3;}
     | rexpr DIVIDE rexpr
       { if($3) $$ = $1 / $3;         else { yyerror( "divide by zero" );    }       }
     | MINUS rexpr %prec UMINUS
       { $$ = - $2;}
     | LP rexpr RP
       { $$ = $2;}
     | iexpr PLUS rexpr
       { $$ = (double)$1 + $3;}
     | iexpr MINUS rexpr
       { $$ = (double)$1 - $3;}
     | iexpr TIMES rexpr
       { $$ = (double)$1 * $3;}
     | iexpr DIVIDE rexpr
       { if($3) $$ = (double)$1 / $3;         else { yyerror( "divide by zero" );      }       }
     | rexpr PLUS iexpr
       { $$ = $1 + (double)$3;}
     | rexpr MINUS iexpr
       { $$ = $1 - (double)$3;}
     | rexpr TIMES iexpr
       { $$ = $1 * (double)$3;}
     | rexpr DIVIDE iexpr
       { if($3) $$ = $1 / (double)$3;         else { yyerror( "divide by zero" );      }       }
     ;
```

Example: simple calculator

```
%%
#include "lex.yy.c"
int lineno;

main() {
  return yyparse();
}

yyerror( char *s ) { fprintf( stderr, "%s\n", s); };
```

Simple programming language

Our example language provides arithmetic and relational expressions as well as assignment and print statements. To structure programs it features conditional and repetitive statements and the possibility to group statements to sequences.

Here is a typical program in our example language:

```
// Greatest Common Divisor

x := 8;
y := 12;
WHILE x != y DO
    IF x > y THEN
        x := x-y
    ELSE
        y := y-x
    FI
OD;
PRINT x
```

lex

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
"=" { return EQ; }
"!=" { return NE; }
"<" { return LT; }
"<=" { return LE; }
">" { return GT; }
">=" { return GE; }
"+" { return PLUS; }
"-" { return MINUS; }
"*" { return MULT; }
"/" { return DIVIDE; }
")" { return RPAREN; }
"(" { return LPAREN; }
":=" { return ASSIGN; }
";" { return SEMICOLON; }
```

lex

```
"IF" { return IF; }
"THEN" { return THEN; }
"ELSE" { return ELSE; }
"FI" { return FI; }
"WHILE" { return WHILE; }
"DO" { return DO; }
"OD" { return OD; }
"PRINT" { return PRINT; }
[0-9]+ { yylval = atoi(yytext); return
  NUMBER; }
[a-z] { yylval = yytext[0] - 'a'; return
  NAME; }
\ { ; }
\n { nextline(); }
\t { ; }
"//".*\n { nextline(); }
. { yyerror("illegal token"); }
%% #ifndef yywrap yywrap() { return 1; }
  #endif
```

BNF

```
statement: designator ASSIGN
  expression
| PRINT expression
| IF expression THEN stmtseq ELSE
  stmtseq FI
| IF expression THEN stmtseq FI
| WHILE expression DO stmtseq OD ;
```

yacc

```
statement:
designator ASSIGN expression {$$ = assignment($1, $3);}
| PRINT expression
   {$$ = print($2);}
| IF expression THEN stmtseq ELSE stmtseq FI
   {$$ = ifstmt($2, $4, $6);}
| IF expression THEN stmtseq FI
    {$$ = ifstmt($2, $4, empty());}
| WHILE expression DO stmtseq OD
    {$$ = whilestmt($2, $4);} ;
```

Complete yacc

```
%start ROOT
%token EQ
%token NE
%token LT
%token LE
%token GT
%token GE
%token PLUS
%token MINUS
%token MULT
%token DIVIDE
%token RPAREN
%token LPAREN
%token ASSIGN
%token SEMICOLON
%token IF
%token THEN
%token ELSE
%token FI
%token WHILE
%token DO
%token OD
%token PRINT
%token NUMBER
%token NAME
```

Complete yacc

```
%%
ROOT: stmtseq { execute($1); } ;
statement: designator ASSIGN expression
        { $$ = assignment($1, $3); }
 | PRINT expression
        { $$ = print($2); }
| IF expression THEN stmtseq ELSE stmtseq FI
        { $$ = ifstmt($2, $4, $6); }
| IF expression THEN stmtseq FI
        { $$ = ifstmt($2, $4, empty()); }
| WHILE expression DO stmtseq OD
        { $$ = whilestmt($2, $4); } ;
```

Complete yacc

```
stmtseq: stmtseq SEMICOLON statement
        { $$ = seq($1, $3); }
| statement
        { $$ = $1; } ;
expression: expr2
        { $$ = $1; }
| expr2 EQ expr2
      { $$ = eq($1, $3); }
| expr2 NE expr2
      { $$ = ne($1, $3); }
| expr2 LT expr2
      { $$ = le($1, $3); }
| expr2 LE expr2
      { $$ = le($1, $3); }
| expr2 GT expr2
      { $$ = gt($1, $3); }
| expr2 GE expr2
      { $$ = gt($1, $3); } ;
```

Complete yacc

```
expr2: expr3
            { $$ == $1; }
     | expr2 PLUS expr3
            { $$ = plus($1, $3); }
     | expr2 MINUS expr3
            { $$ = minus($1, $3); } ;
     | expr3: expr4
            { $$ = $1; }
     | expr3 MULT expr4
            { $$ = mult($1, $3); }
     | expr3 DIVIDE expr4
            { $$ = divide ($1, $3); } ;
```

Complete yacc

```
expr4: PLUS expr4 { $$ = $2; } | MINUS expr4 { $$ = neg($2); } | LPAREN
    expression RPAREN { $$ = $2; } | NUMBER { $$ = number($1); } | designator
    { $$ = $1; } ; designator: NAME { $$ = name($1); } ;
```

Complete yacc

Actions between rule elements

```
/* lex specification */
%%
a   return A;
b   return B;
\n return NL;
.   ;
```

input: ab
output: 1452673
input: aa
output: 14 syntax error
526
input: ba
output: 14 syntax error

```
/* yacc specification */
%token A B NL
%%
s: {printf("1");}
   a
   {printf("2");}
   b
   {printf("3");}
   NL
   {return 0;}
   ;
a: {printf("4");}
   A
   {printf("5");}
   ;
b: {printf("6");}
   B
   {printf("7");}
   ;
```

References

- http://memphis.compilertools.net/interpreter.html
- http://www.opengroup.org/onlinepubs/007908799/xcu/yacc.html
- http://dinosaur.compilertools.net/yacc/index.html