

# Lecture 6: yacc and bison

David Hovemeyer

September 21, 2020

601.428/628 Compilers and Interpreters



# Today

- ▶ yacc and bison
- ▶ Using flex and bison together

# yacc/bison: background

# Approaches to parsing

We've discussed:

- ▶ Hand-coded recursive descent
- ▶ Precedence climbing (for infix expressions)
- ▶ LL(1) (sort of like automated recursive descent)

Today: yacc and bison

- ▶ Takes parser specification as input: grammar rules + actions
- ▶ Generates a *bottom-up* parser using LALR(1) table construction algorithm
  - ▶ Will discuss in detail next class

# yacc and bison

- ▶ yacc: “Yet Another Compiler Compiler”
- ▶ Invented by Stephen C. Johnson at AT&T Bell Labs in the early 1970s
- ▶ bison: open-source reimplementation of yacc

# Advantages of yacc/bison

- ▶ LALR(1) is a fairly powerful parsing algorithm
  - ▶ Can handle left recursion
- ▶ Much flexibility in semantic actions for parsing rules
  - ▶ Data types can be specified for grammar symbols
- ▶ Using yacc/bison is often the quickest approach to creating a front end for an interpreter or compiler

# Disadvantages of yacc/bison

- ▶ Grammar must be written with limitations of LALR(1) in mind
  - ▶ Of course, most practical parsing algorithms have limitations
- ▶ Error handling can be difficult

# yacc/bison basics



# yacc/bison parser specification

%{

*C preamble (includes, definitions, global vars)*

%}

*options*

%%

*grammar rules and actions*

%%

*C functions*

# Grammar symbols

- ▶ Terminal symbols: defined with `%token` directives in the options section
  - ▶ Each is assigned a unique integer value, defined in a generated header file
- ▶ Nonterminal symbols: defined by grammar rules

# Interaction with lexical analyzer

- ▶ Generated parser will call `yylex()` when it wants to read a token
- ▶ Token kinds are integer values
  - ▶ Can also use ASCII characters as token kinds as a convenient representation of single-character tokens
- ▶ Can use a flex-generated lexer to provide `yylex()`, or could hand-code

# Types, YYSTYPE, yylval

- ▶ All grammar symbols (terminal and nonterminal) can have a data type
- ▶ YYSTYPE is a C union data type, specified with the %union directive
- ▶ yylval is a global variable which is an instance of YYSTYPE
- ▶ Token (terminal symbol) types specified using %token directives
- ▶ Nonterminal types specified using %type directives

Example: if we want the parser to build a parse tree, we can make the type of every grammar symbol a pointer to a parse tree node:

```
%union {  
    struct Node *node;  
}  
%token<node> TOK_A, TOK_B, etc...  
%type<node> nonterm1, nonterm2, etc...
```

# Grammar rules

Say that your grammar has the following productions (nonterminals in *italics*, terminals in **bold**):

*sexp*  $\rightarrow$  *atom*

*sexp*  $\rightarrow$  ( *opt\_items* )

*opt\_items*  $\rightarrow$  *items*

*opt\_items*  $\rightarrow \epsilon$

*items*  $\rightarrow$  *sexp*

*items*  $\rightarrow$  *sexp items*

*atom*  $\rightarrow$  **number**

*atom*  $\rightarrow$  **symbol**

# Grammar rules in yacc/bison

Grammar rules from previous slide written in yacc/bison format (starting on left, continuing on right):

```
sexp
: atom
| TOK_LPAREN opt_items TOK_RPAREN
;
```

```
opt_items
: items
| /* epsilon */
;
```

```
items
: sexp
| sexp items
;
```

```
atom
: TOK_NUMBER
| TOK_SYMBOL
;
```

Productions are grouped by left-hand-side nonterminal; first grammar rule defines productions for the start symbol

# Actions

Each grammar rule can have an *action*: code executed when the grammar rule is *reduced* (more about this terminology next time)

- ▶ Values of right-hand symbols can be accessed as \$1, \$2, \$3, etc.
- ▶ Value of left-hand symbol can be defined by assigning to \$\$
- ▶ Types correspond to fields of YYSTYPE, and are specified using %token and %type directives (as seen earlier)

Example, building parse trees for *sexp* nonterminals:

```
sexp
: atom
  { $$ = node_build1(NODE_sexp, $1); }
| TOK_LPAREN opt_items TOK_RPAREN
  { $$ = node_build3(NODE_sexp, $1, $2, $3); }
;
```

# Complete example



# JSON parser

- ▶ JSON: JavaScript Object Notation (<https://www.json.org/>)
- ▶ Commonly used in web applications for data exchange
  - ▶ Increasingly common for non-web applications as well
- ▶ Let's use flex and bison to make a parser for it
  - ▶ <https://github.com/daveho/jsonparser>

# JSON overview

- ▶ Values are numbers, strings, objects and arrays
- ▶ Objects: curly braces (`{` and `}`) surrounding a sequence of fields
- ▶ Arrays: square brackets (`[` and `]`) surrounding a sequence of values
- ▶ Sequences: items separated by commas (`,`)
- ▶ Object fields: use colon (`:`) to join field name and value

# Example JSON object

```
{  
  "name" : "Admin",  
  "age" : 36,  
  "rights" : [ "admin", "editor", "contributor" ]  
}
```

Source: <https://restfulapi.net/json-objects/>

# JSON grammar

Nonterminals in *italics*, terminals in **bold**

*value*  $\rightarrow$  **number**

*value*  $\rightarrow$  **string**

*value*  $\rightarrow$  *object*

*value*  $\rightarrow$  *array*

*object*  $\rightarrow$  { *opt\_field\_list* }

*opt\_field\_list*  $\rightarrow$  *field\_list*

*opt\_field\_list*  $\rightarrow$   $\epsilon$

*field\_list*  $\rightarrow$  *field*

*field\_list*  $\rightarrow$  *field* , *field\_list*

*field*  $\rightarrow$  **string** : *value*

*array*  $\rightarrow$  [ *opt\_value\_list* ]

*opt\_value\_list*  $\rightarrow$  *value\_list*

*opt\_value\_list*  $\rightarrow$   $\epsilon$

*value\_list*  $\rightarrow$  *value*

*value\_list*  $\rightarrow$  *value* , *value\_list*

# Lexer: create\_token function

The `create_token` function creates a struct `Node` to represent a token, and returns the integer value uniquely identifying the token kind

- ▶ Token is conveyed to parser using `yylval` union

```
int create_token(int kind, const char *lexeme) {  
    struct Node *n = node_alloc_str_copy(kind, lexeme);  
    // FIXME: set source info  
    yylval.node = n;  
    return kind;  
}
```

## Lexer: easy parts

```
"{"      { return create_token(TOK_LBRACE, yytext); }
"}"      { return create_token(TOK_RBRACE, yytext); }
"["      { return create_token(TOK_LBRACKET, yytext); }
"]"      { return create_token(TOK_RBRACKET, yytext); }
":"      { return create_token(TOK_COLON, yytext); }
","      { return create_token(TOK_COMMA, yytext); }
[ \t\r\n\v]+ { /* ignore whitespace */ }
```

# Lexer: numbers

```
"-"?(0|[1-9][0-9]*)("."[0-9]*)?((e|E)( "+" | "-" )?[0-9]+)? {  
    return create_token(TOK_NUMBER, yytext); }
```

- ▶ Regular expression is slightly complicated due to possibility of minus sign, decimal point, and/or exponent
- ▶ ? means “zero or one” (i.e., optional)

# Lexer: string literals

- ▶ String literals would be fairly complicated to write a regular expression for
- ▶ We can use *lexer states* to simplify handling them
- ▶ Idea: when the opening double quote (") character is seen, enter STRLIT lexer state
  - ▶ After terminating " is seen, return to default INITIAL state
- ▶ Lexer specification has the directive

`%x STRLIT`

in the options section to define the additional lexer state

- ▶ A global character buffer `g_strbuf` is used to accumulate the string literal's lexeme (not a great design, but expedient)



# Lexer: string literals

```
/* beginning of string literal */
\"      { g_strbuf[0] = '\\0'; add_to_string("\\"); BEGIN STRLIT; }
/* escape sequence */
<STRLIT>\\([\\\"/bfnrt]|u[0-9A-Fa-f]{4}) { add_to_string(yytext); }
/* string literal ends */
<STRLIT>\"      { add_to_string("\\");
                  BEGIN INITIAL;
                  return create_token(TOK_STRING_LITERAL, g_strbuf); }
<STRLIT><<EOF>> { err_fatal("Unterminated string literal"); }
/* "ordinary" character in string
 * (FIXME: should reject control chars) */
<STRLIT>.      { add_to_string(yytext); }
```

Definition of add\_to\_string function:

```
void add_to_string(const char *s) {
    strcat(g_strbuf, s);
}
```

# Lexer: handling unknown characters

Final lexer rule:

```
. { err_fatal("Unknown character"); }
```

# Parser: types

We'll have the parser build a parse tree, so the type of every symbol (terminal and nonterminal) will be a pointer to a parse node:

```
%union {  
    struct Node *node;  
}  
  
%token<node> TOK_LBRACE TOK_RBRACE TOK_LBRACKET TOK_RBRACKET  
%token<node> TOK_COLON TOK_COMMA  
%token<node> TOK_NUMBER TOK_STRING_LITERAL  
  
%type<node> value  
%type<node> object opt_field_list field_list field  
%type<node> array opt_value_list value_list
```

# Parser: integer values for nonterminal symbols

- ▶ All parse nodes in the tree should be tagged with an integer code identifying their grammar symbol
- ▶ For terminal symbols, use the token kind value
  - ▶ yacc/bison will emit these in a header file: e.g., for `parse.y`, header file is `parse.tab.h`
- ▶ What to do for nonterminal symbols?
- ▶ Observation: if we use formatting suggested earlier, left hand sides of productions are on a line by themselves: e.g.,

```
field_list
: field
| field TOK_COMMA field_list
;
```

- ▶ Idea: use a script to extract names of all terminal and nonterminal symbols from parser spec, generate header and source files

# scan\_grammar\_symbols.rb

Running the script (user input in **bold**):

```
$ ./scan_grammar_symbols.rb < parse.y
Generating grammar_symbols.h/grammar_symbols.c...Done!
$ ls grammar_symbols.*
grammar_symbols.c  grammar_symbols.h
```

Header file will have an enumeration called GrammarSymbol with members for all terminal and nonterminal symbols

- ▶ All symbols are prefixed with NODE\_

Also declares a function called get\_grammar\_symbol\_name to translate grammar symbols to strings, useful for printing textual representation of parse tree

# Parser: grammar rules, actions

Given the header file defining identifiers for grammar symbols, we can define an action for each grammar rule to create a parse node of the appropriate type

Examples:

```
opt_field_list
: field_list { $$ = node_build1(NODE_opt_field_list, $1); }
| /* epsilon */ { $$ = node_build0(NODE_opt_field_list); }
;
```

```
field_list
: field { $$ = node_build1(NODE_field_list, $1); }
| field TOK_COMMA field_list
  { $$ = node_build3(NODE_field_list, $1, $2, $3); }
;
```

# main function

```
int yyparse(void);

int main(void) {
    // yyparse() will set this to the root of the parse tree
    extern struct Node *g_parse_tree;

    yyparse();

    treeprint(g_parse_tree, get_grammar_symbol_name);
    return 0;
}
```

Lexer will implicitly read from standard input (can set yyin to read from a different input source)

# Running the program

```
$ ./jsonparser
[{"bananas" : 3}, {"apples" : 4}]
value
+--array
  +--TOK_LBRACKET[[]
  +--opt_value_list
    | +--value_list
    |   +--value
    |     | +--object
    |       | +--TOK_LBRACE[{]
    |       | +--opt_field_list
    |       |   +--field_list
    |       |     +--field
    |       |       +--TOK_STRING_LITERAL["bananas"]
    |       |       +--TOK_COLON[:]
    |       |       +--value
    |       |         +--TOK_NUMBER[3]
    |       |     +--TOK_RBRACE[}]
    ...additional output omitted...
```



# Using flex and bison

# Makefile issues

- ▶ Both flex and bison generate source code
- ▶ So, writing a Makefile can be interesting
- ▶ General idea: have explicit rules to generate `.c` files from `.l` and `.y` files
  - ▶ `.y` file will also generate a `.h` file
- ▶ Also need to run `generate_grammar_symbols.rb` to generate `grammar_symbols.h` and `grammar_symbols.c`

# Example Makefile

```
C_SRCS = main.c util.c parse.tab.c lex.yy.c grammar_symbols.c node.c treeprint.c
C_OBJS = $(C_SRCS:%.c=%.o)

CC = gcc
CFLAGS = -g -Wall

%.o : %.c
    $(CC) $(CFLAGS) -c $<

jsonparser : $(C_OBJS)
    $(CXX) -o $@ $(C_OBJS)

parse.tab.c parse.tab.h : parse.y
    bison -d parse.y

lex.yy.c : lex.l
    flex lex.l

grammar_symbols.h grammar_symbols.c : parse.y scan_grammar_symbols.rb
    ./scan_grammar_symbols.rb < parse.y

clean :
    rm -f *.o parse.tab.c lex.yy.c parse.tab.h grammar_symbols.h grammar_symbols.c
```