Lecture 9: yacc and bison

David Hovemeyer

September 25, 2024

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**):

$$
\begin{aligned}
&\textit{sexp} \longrightarrow \textit{atom} \\
&\textit{sexp} \longrightarrow \textbf{(}\ \textit{opt\_items}\ \textbf{)} \\
&\textit{opt\_items} \longrightarrow \textit{items} \\
&\textit{opt\_items} \longrightarrow \epsilon \\
&\textit{items} \longrightarrow \textit{sexp} \\
&\textit{items} \longrightarrow \textit{sexp}\ \textit{items} \\
&\textit{atom} \longrightarrow \textbf{number} \\
&\textit{atom} \longrightarrow \textbf{symbol}
\end{aligned}
$$

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 → **number**
value → **string**
value → *object*
value → *array*
object → **{** *opt_field_list* **}**
opt_field_list → *field_list*
opt_field_list → ϵ
field_list → *field*
field_list → *field* **,** *field_list*
field → **string** **:** *value*

array → **[** *opt_value_list* **]**
opt_value_list → *value_list*
opt_value_list → ϵ
value_list → *value*
value_list → *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 enumation 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
```

Semantic Actions

We don't have to build a tree

- Semantic actions are arbitrary code
- `YYSTYPE` (defined by the parser's `%union` directive) along with `%token` and `%type` directives allow us to specify an arbitrary type for each grammar symbol
- So, having the parser build a tree is not the only possibility
- E.g., if the input represents a computation, the semantic actions could perform the computation

Calculator example

- Simple calculator example, `calc.zip` on course website (in schedule entry for today's lecture)
- Integer values (represented using `long` data type)
- Named variables
- Operators: addition, subtraction, multiplication, division, exponentiation, assignment
- Parentheses for grouping
- Input is series of expressions (one per line), calculator computes value of each and outputs the result

Example run

```
$ ./calc
a = 4
4
b = 5
5
a + b * 6
34
(a + b) * 6
54
2 ^ 3 ^ 2
512
(2 ^ 3) ^ 2
64
```

YYSTYPE, token and nonterminal types

```
%union {
  long ival;          // computed value of expression
  const char *ident; // variable name
}

%token<ival> INT_LITERAL
%token<ident> IDENTIFIER
%token OP_PLUS OP_MINUS OP_TIMES OP_DIVIDE OP_EXP OP_ASSIGN
%token LPAREN RPAREN EOL

%type<ival> assign_expr additive_expr multiplicative_expr
%type<ival> exp_expr primary_expr
```

Statement list, statement rules

```
stmt_list
  : stmt_list stmt
  | stmt
  ;

stmt
  : assign_expr EOL
    { std::cout << $1 << "\n"; }
  ;
```

When a semantic action has been executed, the values of all symbols on the right hand side of the production are already known.

Also note the left recursion in `stmt_list`!

Assignment and additive expressions

```
assign_expr
  : IDENTIFIER OP_ASSIGN assign_expr
    { env[$1] = $3; $$ = $3; }
  | additive_expr
    { $$ = $1; }
  ;

additive_expr
  : additive_expr OP_PLUS multiplicative_expr  // <-- left recursion!
    { $$ = $1 + $3; }
  | additive_expr OP_MINUS multiplicative_expr // <-- left recursion!
    { $$ = $1 - $3; }
  | multiplicative_expr
    { $$ = $1; }
  ;
```

Note that `env` is a map of strings to `long` values.

Multiplicative and exponentiation expressions

```
multiplicative_expr
  : multiplicative_expr OP_TIMES exp_expr
    { $$ = $1 * $3; }
  | multiplicative_expr OP_DIVIDE exp_expr
    { $$ = $1 / $3; }
  | exp_expr
    { $$ = $1; }
  ;

exp_expr
  : primary_expr OP_EXP exp_expr
    { $$ = raise_to_power($1, $3); }
  | primary_expr
    { $$ = $1; }
  ;
```

Primary expressions

```
primary_expr
  : INT_LITERAL
    { $$ = $1; }
  | IDENTIFIER
    { auto it = env.find($1);
      if (it == env.end())
        throw std::runtime_error(std::string("undefined variable ") + $1);
      $$ = it->second; }
  | LPAREN assign_expr RPAREN
    { $$ = $2; }
  ;
```

Lexer patterns

```
[0-9]+              { yylval.ival = std::stol(yytext); return INT_LITERAL; }
[a-zA-Z][a-zA-Z]*   { yylval.ident = intern(yytext); return IDENTIFIER; }
"+"                 { return OP_PLUS; }
"-"                 { return OP_MINUS; }
"*"                 { return OP_TIMES; }
"/"                 { return OP_DIVIDE; }
"^"                 { return OP_EXP; }
"="                 { return OP_ASSIGN; }
"("                 { return LPAREN; }
")"                 { return RPAREN; }
"\n"                { return EOL; }
[ \t\r\f\v]+        { /* ignore whitespace */ }
.                   { std::string msg = "Unexpected character: '";
                      msg += yytext[0]; msg += "'";
                      throw std::runtime_error(msg); }
```

Avoiding use of global variables

Global variables

- The original `yacc` and `lex` tools were developed in the 1970s, when use of global variables was common
- Today: global variables prevent use of concurrency and also prevent program from using multiple lexer and/or parser instances
- Flex and bison both allow reentrant lexers and parsers (respectively) to be created

Pure parser

- Use `%option api.pure` to enable
- Use the `%parse-param` directive to specify a parameter to be passed to `yyparse()`: this is a reference to the *parser state* object, which should contain the lexer and anything else needed by the parser (e.g., pointer to root of tree being built)
  - Semantic actions can refer to this parameter
- Use the `%lex-param` directive to specify an argument to pass to `yylex()` (since we also want the lexer to be reentrant)

Reentrant lexer

- Use `%option reentrant`: the `yyscan_t` opaque pointer data type encapsulates lexer state (including which input source to read from), and value of this type is passed to `yylex()`
- Use `%option bison-bridge`: causes `yylval` to be passed to `yylex()` as a pointer (avoiding the need for it to be a global variable)