Introduction to Compilers and Language Design

Hardcover ISBN: 978-0-359-13804-3
Paperback ISBN: 978-0-359-14283-5
First edition.

You can find the latest version of the PDF edition, and purchase inexpensive hardcover copies at `http://compilerbook.org`

Revision Date: February 5, 2020

Chapter 5 – Parsing in Practice

In this chapter, you will apply what you have learned about the theory of LL(1) and LR(1) grammars in order to build a working parser for a simple expression language. This will give you a basis to write a more complete parser for B-Minor in the following chapter.

While LL(1) parsers are commonly written by hand, LR(1) parsers are simply too cumbersome to do the same. Instead, we rely upon a **parser generator** to take a specification of a grammar and automatically produce the working code. In this chapter, we will give examples with Bison, a widely-used parser generator for C like languages.

Using Bison, we will define an LALR grammar for algebraic expressions, and then employ it to create three different varieties of programs.

- A **validator** reads the input program and then simply informs the user whether it is a valid sentence in the language specified by the grammar. Such a tool is often used to determine whether a given program conforms to one standard or another.
- An **interpreter** reads the input program and then actually executes the program to produce a result. One approach to interpretation is to compute the result of each operation as soon as it is parsed. The alternate approach is to parse the program into an abstract syntax tree, and then execute it.
- A **translator** reads the input program, parses it into an abstract syntax tree, and then traverses the abstract syntax tree to produce an equivalent program in a different format.

5.1 The Bison Parser Generator

It is not practical to implement an LALR parser by hand, and so we rely on tools to automatically generate tables and code from a grammar specification. YACC (Yet Another Compiler Compiler) was a widely used parser generator in the Unix environment, recently supplanted by the GNU Bison parser which is generally compatible. Bison is designed to automatically invoke Flex as needed, so it is easy to combine the two into a complete program.

Just as with the scanner, we must create a specification of the grammar to be parsed, where each rule is accompanied by an action to follow. The overall structure of a Bison file is similar to that of Flex:

```
%{
    (C preamble code)
%}
    (declarations)
%%
    (grammar rules)
%%
    (C postamble code)
```

The first section contains arbitrary C code, typically `#include` statements and global declarations. The second section can contain a variety of declarations specific to the Bison language. We will use the `%token` keyword to declare all of the terminals in our language. The main body of the file contains a series of rules of the form

```
expr : expr TOKEN_ADD expr
     | TOKEN_INT
     ;
```

indicating that non-terminal `expr` can produce the sentence `expr TOKEN_ADD expr` or the single terminal `TOKEN_INT`. White space is not significant, so it's ok to arrange the rules for clarity. Note that the usual naming convention is reversed: since upper case is customarily used for C constants, we use lower case to indicate non-terminals.

The resulting code creates a single function `yyparse()` that returns an integer: zero indicates a successful parse, one indicates a parse error, and two indicates an internal problem such as memory exhaustion. `yyparse` assumes that there exists a function `yylex` that returns integer token types. This can be written by hand or generated automatically by Flex. In the latter case, the input source can be changed by modifying the file pointer `yyin`.

Figure 5.1 gives a Bison specification for simple algebraic expressions on integers. Remember that Bison accepts an LR(1) grammar, so it is ok to have left recursion within the various rules.

```
%{
#include <stdio.h>
%}

%token TOKEN_INT
%token TOKEN_PLUS
%token TOKEN_MINUS
%token TOKEN_MUL
%token TOKEN_DIV
%token TOKEN_LPAREN
%token TOKEN_RPAREN
%token TOKEN_SEMI
%token TOKEN_ERROR

%%
program : expr TOKEN_SEMI;

expr : expr TOKEN_PLUS term
     | expr TOKEN_MINUS term
     | term
     ;

term : term TOKEN_MUL factor
     | term TOKEN_DIV factor
     | factor
     ;

factor: TOKEN_MINUS factor
      | TOKEN_LPAREN expr TOKEN_RPAREN
      | TOKEN_INT
     ;
%%

int yywrap() { return 0; }
```

Figure 5.1: Bison Specification for Expression Validator

```
#include <stdio.h>

extern int yyparse();

int main()
{
    if(yyparse()==0) {
        printf("Parse successful!\n");
    } else {
        printf("Parse failed.\n");
    }
}
```

Figure 5.2: Main Program for Expression Validator

Figure 5.3 shows the general build procedure for a combined program that uses Bison and Flex together. The parser specification goes in `parser.bison`. We assume that you have written a suitable scanner and placed it in `scanner.flex`. Previously, we wrote `token.h` by hand. Here, we will rely on Bison to generate `token.h` automatically from the `%token` declarations, so that the parser and the scanner are working from the same information. Invoke Bison like this:

```
bison --defines=token.h --output=parser.c parser.bison
```

The `--output=parser.c` option directs Bison to write its code into the file `parser.c` instead of the cryptic `yy.tab.c`. Then, we compile `parser.c`, the `scanner.c` generated by Flex, and `main.c` independently, and link them together into a complete executable.

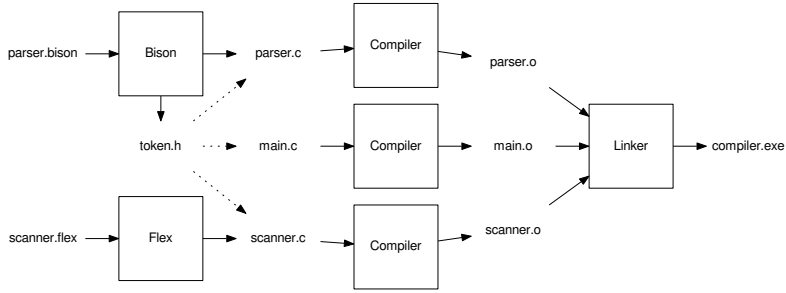


Figure 5.3: Build Procedure for Bison and Flex Together

If you give Bison the `-v` option, it will output a text representation of the LALR automaton to the file `parser.output`. For each state, it gives the items, using dot to indicate the parser position. Then, it lists the actions applied to that state. For example, suppose that we modify the grammar above so that it becomes ambiguous:

```
expr : expr TOKEN_PLUS expr
```

Bison will report that the grammar has one shift-reduce conflict, and `parser.output` will describe each state. In the event of a conflict, Bison will suppress one or more actions, and this is indicated by square brackets in the following report:

```
state 9
    2 expr: expr . TOKEN_PLUS expr
    2     | expr TOKEN_PLUS expr .

    TOKEN_PLUS  shift, and go to state 7
    TOKEN_PLUS  [reduce using rule 2 (expr)]
    $default    reduce using rule 2 (expr)
```

Be careful! If your grammar has shift-reduce or reduce-reduce conflicts, Bison will happily output working code with some of the conflicting actions suppressed. The code may appear to work on simple inputs, but is likely to have unexpected effects on complete programs. Always check for conflicts before proceeding.

5.2 Expression Validator

As written, the Bison specification in Figure 5.1 will simply evaluate whether the input program matches the desired grammar. `yyparse()` will return zero on success, and non-zero otherwise. Such a program is known as a **validator** and is often used to determine whether a given program is standards compliant.

There are a variety of online validators for web-related languages like HTML[1], CSS[2], and JSON[3]. By having a strict language definition separate from actual implementations (which may contain non-standard features) it is much easier for a programmer to determine whether their code is standards compliant, and therefore (presumably) portable.

[1] http://validator.w3.org
[2] http://css-validator.org
[3] http://jsonlint.com

5.3 Expression Interpreter

To do more than simply validate the program, we must make use of **semantic actions** embedded within the grammar itself. Following the right side of any production rule, you may place arbitrary C code inside of curly braces. This code may refer to **semantic values** which represent the values already computed for other non-terminals. Semantic values are given by dollar signs indicating the position of a non-terminal in a production rule. Two dollar signs indicates the semantic value of the current rule.

For example, in the rule for addition, the appropriate semantic action is to add the left value (the first symbol) to the right value (the third symbol):

```
expr : expr TOKEN_PLUS term { $$ = $1 + $3; }
```

Where do the semantic values `$1` and `$3` come from? They simply come from the other rules that define those non-terminals. Eventually, we reach a rule that gives the value for a leaf node. For example, this rule indicates that the semantic value of an integer token is the integer value of the token text:

```
factor : TOKEN_INT { $$ = atoi(yytext); }
```

(Careful: the value of the token comes from the `yytext` array in the scanner, so you can only do this when the rule has a single terminal on the right hand side of the rule.)

In the cases where a non-terminal expands to a single non-terminal, we simply assign one semantic value to the other:

```
term : factor { $$ = $1; }
```

Because Bison is a bottom-up parser, it determines the semantic values of the leaf nodes in the parse tree first, then passes those up to the interior nodes, and so on until the result reaches the start symbol.

Figure 5.4 shows a Bison grammar that implements a complete interpreter. The main program simply invokes `yyparse()`. If successful, the result is stored in the global variable `parser_result` for extraction and use from the main program.

```
prog : expr TOKEN_SEMI       { parser_result = $1; }
     ;

expr : expr TOKEN_PLUS term        { $$ = $1 + $3; }
     | expr TOKEN_MINUS term       { $$ = $1 - $3; }
     | term                             { $$ = $1; }
     ;

term : term TOKEN_MUL factor       { $$ = $1 * $3; }
     | term TOKEN_DIV factor       { $$ = $1 / $3; }
     | factor                           { $$ = $1; }
     ;

factor
     : TOKEN_MINUS factor              { $$ = -$2; }
     | TOKEN_LPAREN expr TOKEN_RPAREN   { $$ = $2; }
     | TOKEN_INT              { $$ = atoi(yytext); }
     ;
```

Figure 5.4: Bison Specification for an Interpreter

5.4 Expression Trees

So far, our expression interpreter is computing results in the middle of parsing the input. While this works for simple expressions, it has several general drawbacks: One is that the program may perform a large amount of computation, only to discover a parse error late in the program. It is generally more desirable to find all parse errors *before* execution.

To fix this, we will add a new stage to the interpreter. Instead of computing values outright, we will construct a data structure known as the **abstract syntax tree** to represent the expression. Once the AST is created, we can traverse the tree to check, execute, and translate the program as needed.

Figure 5.5 shows the C code for a simple AST representing expressions. `expr_t` enumerates the five kinds of expression nodes. `struct expr` describes a node in the tree, which is described by a kind, a left and right pointer, and an integer `value` for a leaf. The function `expr_create` creates a new tree node of any kind, while `expr_create_value` creates one specifically of kind `EXPR_VALUE`. [4]

[4] Although it is verbally awkward, we are using the term "kind" rather than "type", which will have a very specific meaning later on.

Contents of File: expr.h

```
typedef enum {
    EXPR_ADD,
    EXPR_SUBTRACT,
    EXPR_DIVIDE,
    EXPR_MULTIPLY,
    EXPR_VALUE
} expr_t;

struct expr {
       expr_t kind;
       struct expr *left;
       struct expr *right;
       int value;
};
```

Contents of File: expr.c

```
struct expr * expr_create( expr_t kind,
                           struct expr *left,
                           struct expr *right )
{
    struct expr *e = malloc(sizeof(*e));
    e->kind = kind;
    e->value = 0;
    e->left = left;
    e->right = right;
    return e;
}

struct expr * expr_create_value( int value )
{
    struct expr *e = expr_create(EXPR_VALUE,0,0);
    e->value = value;
    return e;
}
```

Figure 5.5: AST for Expression Interpreter

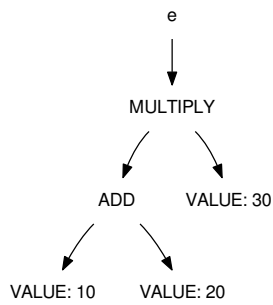
Using the expression structure, we can create some simple ASTs by hand. For example, if we wanted to create an AST corresponding to the expression `(10+20)*30`, we could issue this sequence of operations:

```
struct expr *a = expr_create_value(10);
struct expr *b = expr_create_value(20);
struct expr *c = expr_create(EXPR_ADD,a,b);
struct expr *d = expr_create_value(30);
struct expr *e = expr_create(EXPR_MULTIPLY,c,d);
```

Of course, we could have accomplished the same thing by writing a single expression with nested values:

```
struct expr *e =
    expr_create(EXPR_MULTIPLY,
        expr_create(EXPR_ADD,
             expr_create_value(10),
             expr_create_value(20)
        ),
        expr_create_value(30)
    );
```

Either way, the result is a data structure like this:



Instead of building each node of the AST by hand, we want Bison to do the same work automatically. As each element of an expression is recognized, a new node in the tree should be created, and passed up so that it can be linked into the appropriate place. By doing a bottom-up parse, Bison will create the leaves of the tree first, and then link them into the parent nodes.

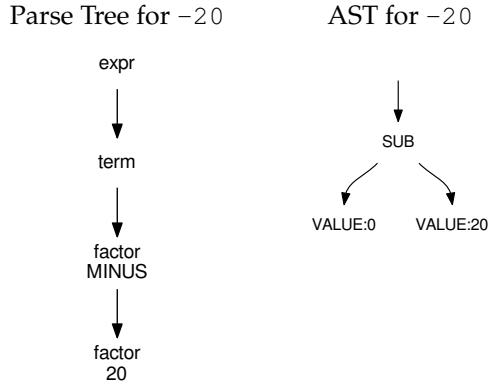
To accomplish this, we must write the semantic actions for each rule to either create a node in the tree, or pass up the pointer from the node below. Figure 5.6 shows how this is done:

```
%{
#include "expr.h"
#define YYSTYPE struct expr *
struct expr * parser_result = 0;
%}

/* token definitions omitted for brevity */

prog : expr TOKEN_SEMI
          { parser_result = $1; }
     ;

expr : expr TOKEN_PLUS term
          { $$ = expr_create(EXPR_ADD,$1,$3); }
     | expr TOKEN_MINUS term
          { $$ = expr_create(EXPR_SUBTRACT,$1,$3); }
     | term
          { $$ = $1; }
     ;

term : term TOKEN_MUL factor
          { $$ = expr_create(EXPR_MULTIPLY,$1,$3); }
     | term TOKEN_DIV factor
          { $$ = expr_create(EXPR_DIVIDE,$1,$3); }
     | factor
          { $$ = $1; }
     ;

factor
     : TOKEN_MINUS factor
          { $$ = expr_create(EXPR_SUBTRACT,
                             expr_create_value(0),$2); }
     | TOKEN_LPAREN expr TOKEN_RPAREN
          { $$ = $2; }
     | TOKEN_INT
          { $$ = expr_create_value(atoi(yytext));
     ;
```

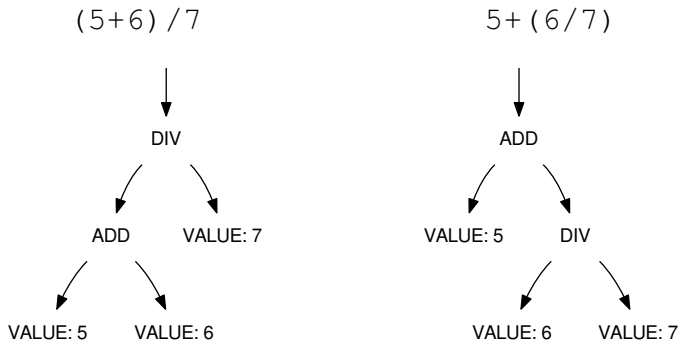
Figure 5.6: Building an AST for the Expression Grammar

Examine Figure 5.6 carefully and note several things:

- In the preamble, we must explicitly define the **semantic type** by setting the macro **YYSTYPE** to `struct expr *`. This causes Bison to use `struct expr *` as the internal type everywhere a semantic value such as `$$` or `$1` is used. The final parser result must have the same semantic type, of course.
- The AST does not always correspond directly to the parse tree. For example, where an `expr` produces a `factor`, we simply pass up the pointer to the underlying node with {`$$ = $1;`} On the other hand, when we encounter a unary minus in `term`, we return a subtree that actually implements subtraction between the value zero on the left and the expression on the right.



- Parentheses are not directly represented in the AST. Instead, they have the effect of ordering the nodes in the tree to achieve the desired evaluation order. For example, consider the AST generated by these sentences:



```
int expr_evaluate( struct expr *e )
{
    if(!e) return 0;

    int l = expr_evaluate(e->left);
    int r = expr_evaluate(e->right);

    switch(e->kind) {
        case EXPR_VALUE:    return e->value;
        case EXPR_ADD:      return l+r;
        case EXPR_SUBTRACT: return l-r;
        case EXPR_MULTIPLY: return l*r;
        case EXPR_DIVIDE:
            if(r==0) {
                printf("runtime error: divide by zero\n");
                exit(1);
            }
            return l/r;
    }

    return 0;
}
```

Figure 5.7: Evaluating Expressions

Now that we have constructed the AST, we can use it as the basis for computation and many other operations.

The AST can be evaluated arithmetically by calling `expr_evaluate` shown in Figure 5.7. This function performs a post-order traversal of the tree by invoking itself recursively on the left and right pointers of the node. (Note the check for a null pointer at the beginning of the function.) Those calls return `l` and `r` which contain the integer result of the left and right subtrees. Then, the result of this node is computed by switching on the `kind` of the current node. (Note also that we must check for division-by-zero explicitly, otherwise `expr_evaluate` would crash when `r` is zero.)

```
void expr_print( struct expr *e )
{
    if(!e) return;

    printf("(");
    expr_print(e->left);

    switch(e->kind) {
        case EXPR_VALUE:    printf("%d",e->value); break;
        case EXPR_ADD:      printf("+"); break;
        case EXPR_SUBTRACT: printf("-"); break;
        case EXPR_MULTIPLY: printf("*"); break;
        case EXPR_DIVIDE:   printf("/"); break;
    }

    expr_print(e->right);
    printf(")");
}
```

Figure 5.8: Printing and Evaluating Expressions

In a similar way, the AST can be converted back to text by calling `expr_print`, shown in Figure 5.8. This function performs an in-order traversal of the expression tree by recursively calling `expr_print` on the left side of a node, displaying the current node, then calling `expr_print` on the right side. Again, note the test for null at the beginning of the function.

As noted earlier, parentheses are not directly reflected in the AST. To be conservative, this function displays a parenthesis around every value. While correct, it results in a lot of parentheses! A better solution would be to only print a parenthesis when a subtree contains an operator of lower precedence.

5.5 Exercises

1. Consult the Bison manual and determine how to automatically generate a graph of the LALR automaton from your grammar. Compare the output from Bison against your hand-drawn version.

2. Modify `expr_evaluate()` (and anything else needed) to handle floating point values instead of integers.

3. Modify `expr_print()` so that it displays the minimum number of parentheses necessary for correctness.

4. Extend the parser and interpreter to allow for invoking several built-in mathematical functions such as `sin(x)`, `sqrt(x)` and so forth.

   Before coding, think a little bit about where to put the names of the functions. Should they be keywords in the language? Or should any function names be simply treated as identifiers and checked in `expr_evaluate()`?

5. Extend the parser and interpreter to allow for variable assignment and use, so that you can write multiple assignment statements followed by a single expression to be evaluated, like this:

```
g = 9.8;
t = 5;
g*t*t - 7*t + 10;
```

5.6 Further Reading

As its name suggests, YACC was not the first compiler construction tool, but it remains widely used today and has led to a proliferation of similar tools written in various languages and addressing different classes of grammars. Here is just a small selection:

1. S. C. Johnson, "YACC: Yet Another Compiler-Compiler", Bell Laboratories Technical Journal, 1975.

2. D. Grune and C.J.H Jacobs, "A programmer-friendly LL(1) parser generator", Software: Practice and Experience, volume 18, number 1.

3. T.J. Parr and R.W. Quong, "ANTLR: A predicated LL(k) Parser Generator", Software: Practice and Experience, 1995.

4. S. McPeak, G.C. Necula, "Elkhound: A Fast, Practical GLR Parser Generator", International Conference on Compiler Construction, 2004.