Better error handling using Flex and Bison

Tips for building more user-friendly compilers and interpreters

Christian Hagen 28 July 2006

Although it is easy to generate programs using Flex and Bison, it is a bit harder to make those programs produce user-friendly syntax and semantic error messages. This article examines the error-handling features of Flex and Bison, shows how to use them, and details some pitfalls.

Introduction

As UNIX® developers know, Flex and Bison are powerful tools for developing lexical and grammar parsers, in particular language compilers and interpreters. If you're unfamiliar with these utilities or the tools they implement -- Lex and Yacc, respectively -- check the Resources section in this article for links to the Flex and Bison documentation and to additional articles that introduce both programs.

This article covers a somewhat more advanced topic: features and techniques for putting better error-handling capabilities into your compiler or interpreter. To illustrate these techniques, I use a sample program called ccalc, which implements an enhanced calculator based on the infix calculator from the Bison handbook. You can download ccalc and associated files from the Download section at the end of this article.

The enhancements include the use of variables. In ccalc, a variable is defined through its first use in an initialization such as `a = 3`. If a variable is used before it is initialized, a semantic error is generated, the variable is created with a value of zero, and a message is printed.

Sample source files

The sample source code consists of seven files:

- **ccalc.c:** The main program and some functions for input, output, and error processing
- **ccalc.h:**: Included definitions for all modules
- **cmath.c:** Mathematical functions
- **parse.y:** Input grammar for Bison
- **lex.l** Input for Flex

© Copyright IBM Corporation 2006

- **makefile:** A simple makefile
- **defs.txt:** Sample input file

The program accepts two parameters:

- `-debug`: Produces debugging output
- `filename`: The name of the input file; the default is `defs.txt`

Settings for Bison

To handle variable names and real values, the semantic type of Bison has to be enhanced:

Listing 1. Better Bison semantic types

```
/* generate include-file with symbols and types */
%defines
/* a more advanced semantic type */
%union {
  double      value;
  char        *string;
}
```

Some grammar rules produce specific semantic types, which have to be declared to Bison as in Listing 2. To get a more portable version of the Bison grammar, the symbols `+-*/()` are redefined. Instead of using a left parenthesis, `(`, the sample uses the terminal symbol, `LBRACE`, which is provided by the lexical analysis. In addition, the precedence for the operators has to be declared.

For Flex, the generated code depends in general on the codepage of the platform. You can use another codepage, but you have to transform the input read. Thus, unlike Bison code, Flex code is not readily portable.

Listing 2. Bison declarations

```
/* terminal symbols */

%token <string>   IDENTIFIER
%token <value>    VALUE
%type <value>     expression

/* operator-precedence
 * top-0: -
 *     1: * /
 *     2: + -
*/

%left ADD SUB
%left MULT DIV
%left NEG

%start program
```

The grammar is very much the same as in the Bison handbook except for the use of names as terminal symbols and the introduction of identifiers. An identifier is defined and initialized in an assignment and can be used anywhere a value is allowed. Listing 3 shows a sample grammar:

Listing 3. Sample Bison grammar

```
program
    : statement SEMICOLON program
    | statement SEMICOLON
    | statement error SEMICOLON program
    ;

statement
    : IDENTIFIER ASSIGN expression
    | expression
    ;

expression
    : LBRACE expression RBRACE
    | SUB expression %prec NEG
    | expression ADD expression
    | expression SUB expression
    | expression MULT expression
    | expression DIV expression
    | VALUE
    | IDENTIFIER
    ;
```

The third production of `program` lets the parser go past an error, search for a semicolon, and continue afterwards (normally an error would be fatal to the parser).

To make the example more interesting, the actual math in the body of a rule is implemented in a separate function. When doing advanced grammars, keep the rules short and use functions for processing that do not deal with parsing directly:

Listing 4. Using a separate function to implement a math rule

```
| expression DIV expression
  {
    $$ = ReduceDiv($1, $3);
  }
```

Finally, the function `yyerror()` has to be defined. This function is called when the generated parser detects a syntax error, invoking in turn the trivial function `PrintError()`, which prints enhanced error messages. See the source code for more details.

Settings for Flex

The lexical analyzer generated by Flex has to provide terminal symbols according to their semantic type. Listing 5 defines the rules for white space, real values, and identifiers and the symbols.

Listing 5. Sample Flex rules

```
[ \t\r\n]+  {
    /* eat up whitespace */
    }

{DIGIT}+  {
    yylval.value = atof(yytext);
    return VALUE;
    }
```

```
{DIGIT}+"."{DIGIT}*        {
    yylval.value = atof(yytext);
    return VALUE;
    }

{DIGIT}+[eE]["+""-"]?{DIGIT}*        {
    yylval.value = atof(yytext);
    return VALUE;
    }

{DIGIT}+"."{DIGIT}*[eE]["+""-"]?{DIGIT}*        {
    yylval.value = atof(yytext);
    return VALUE;
    }

{ID}        {
    yylval.string = malloc(strlen(yytext)+1);
    strcpy(yylval.string, yytext);
    return IDENTIFIER;
    }

"+"       { return ADD; }
"-"       { return SUB; }
"*"       { return MULT; }
"/"       { return DIV; }
"("       { return LBRACE; }
")"       { return RBRACE; }
";"       { return SEMICOLON; }
"="       { return ASSIGN; }
```

To help debugging, at the end of a program run, all the known variables and their current contents are printed.

Sample with plain error messages

Compile and run the sample parser program `ccalc` with the following input (which includes a slight typo):

Listing 6. Sample math parser input

```
a = 3;
3 aa = a * 4;
b = aa / ( a - 3 );
```

The output looks like this:

Listing 7. Sample math parser output

```
Error 'syntax error'
Error: reference to unknown variable 'aa'
division by zero!
final content of variables
   Name----------------- Value----------
   'a                  ' 3
   'b                  ' 3
   'aa                 ' 0
```

That output is not very useful, because it doesn't show where the problems are. That will be fixed in the next section.

Extending Bison for better error messages

The first Bison feature of interest, hidden deep in the Bison manuals, is that it is possible to generate more meaningful error messages in case of a syntax error by using the macro `YYERROR_VERBOSE`.

The plain `'syntax error'` message becomes:

```
Error 'syntax error, unexpected IDENTIFIER, expecting SEMICOLON'
```

This message is much better for debugging.

A better input function

With the old error messages, it is not easy to identify semantic errors. The example, of course, was pretty easy to fix, since you can spot the line with the error right away. In more complex grammars and the corresponding inputs, it might not be so easy. Let's write an input function to get the relevant lines from the file.

Flex has the useful macro `YY_INPUT`, which reads data for token interpretation. You can add a call in `YY_INPUT` to the function `GetNextChar()`, which reads the data from a file and keeps information about the position of the next character to read. `GetNextChar()` uses a buffer to hold one line of input. Two variables store the current line number and the next position in the line:

Listing 8. Better Flex YY_INPUT macro

```
#define YY_INPUT(buf,result,max_size)  {\
    result = GetNextChar(buf, max_size); \
    if (  result <= 0  ) \
      result = YY_NULL; \
    }
```

With the enhanced error-printing function, `PrintError()`, discussed earlier and a nice display of the problematic input line (see the sample source code for the full `PrintError()` source), you have a user-friendly message that displays the position of the next character:

Listing 9. Better Flex errors: Character position

```
       |....+....:....+....:....+....:....+....:....+....:....+
     1 |a = 3;
     2 |3 aa = a * 4;
...... !.....^
Error: syntax error, unexpected IDENTIFIER, expecting SEMICOLON
     3 |b = aa / ( a - 3 );
...... !.......^
Error: reference to unknown variable 'aa'
...... !..............^
Error: division by zero!
```

This same function can be called from other functions such as `ReduceDiv()` to print semantic errors such as *division by zero* or *unknown identifiers*.

If you want to mark the last consumed token, you have to expand the Flex rules and modify the printing of errors. The functions `BeginToken()` and `PrintError()` (both found in the sample source code) are key: `BeginToken()` is called by every rule so it can remember the start and end of every token, and `PrintError()` is called every time an error is printed. That way, you can generate a useful message like this:

Listing 10. Better Flex errors: Indicating exact token position

```
     2 |3 aa = a * 4;
..... !..^^...........
Error: syntax error, unexpected IDENTIFIER, expecting SEMICOLON
```

Pitfall

The generated lexical parser may read multiple characters ahead before it detects a token. Therefore, this procedure might not show the correct position exactly. It depends ultimately on the rules you provide for Flex. The more complex they are, the less accurate the position. The rules in the sample can be processed by Flex by looking ahead just one character, which makes the position prediction accurate.

Bison's location mechanism

Let's look at a *division by zero* error. The last token read (closing parenthesis) is not the cause of the error. The expression `(a-3)` evaluates to zero. For a better error message, you need the expression's location. To do this, provide exact token locations in the global variable `yylloc` of type `YYLTYPE`. Together with the macro YYLLOC_DEFAULT (see the Bison documentation for the default definition), Bison calculates the location of an expression.

Remember that the type is defined only if you use a location in the grammar! This is a common mistake.

The default location type, `YYLTYPE`, is shown in Listing 11. You can redefine this type to include more information, such as the name of the file read by Flex.

Listing 11. The default location type YYLTYPE

```
typedef struct YYLTYPE
{
  int first_line;
  int first_column;
  int last_line;
  int last_column;
} YYLTYPE;
```

In the previous section, you saw the function `BeginToken()`, which is called at the beginning of a new token. This is the right time to store the location. In our example, a token cannot span multiple lines, therefore `first_line` and `last_line` are the same and hold the current line number. The other attributes are the start of the token (`first_column`) and the end (`last_column`), which are calculated by the start and the length of the token.

To use the location, you have to expand the rule-processing function as shown in Listing 12. The location of the token `$3` is referenced through `@3`. To avoid copying the whole structure in the rule, a pointer is generated, `&@3`. This may look a bit unusual, but it's alright.

Listing 12. Remember the location in a rule

```
| expression DIV expression
  {
    $$ = ReduceDiv($1, $3, &@3);
  }
```

In the processing function, you get a pointer to the `YYLTYPE` structure holding the location, and you can generate a nice error message.

Listing 13. Use the stored location in ReduceDiv

```
extern
double ReduceDiv(double a, double b, YYLTYPE *bloc) {
  if (  b == 0  ) {
    PrintError("division by zero! Line %d:c%d to %d:c%d",
                        bloc->first_line, bloc->first_column,
                        bloc->last_line, bloc->last_column);
    return MAXFLOAT;
  }
  return a / b;
}
```

Now the error messages help you find the problem. The division by zero error is located on line 3 between columns 10 and 18.

Listing 14. Better ReduceDiv() error messages

```
       |....+...:....+...:....+....:....+....:....+....:....+
     1 |a = 3;
     2 |3 aa = a * 4;
..... !..^^...........
Error: syntax error, unexpected IDENTIFIER, expecting SEMICOLON
     3 |b = aa / ( a - 3 );
..... !....^^.............
Error: reference to unknown variable 'aa'
..... !.................^..
 Error: division by zero! Line 3:10 to 3:18
final content of variables
   Name----------------- Value----------
   'a                  ' 3
   'b                  ' 3.40282e+38
   'aa                 ' 0
```

Conclusion

Flex and Bison are a powerful combination for parsing grammars. By using the tips and tricks in this article, you can build interpreters that also produce the kind of useful, easily understood error messages that you would find in your favorite compiler.

Downloadable resources

Description	Name	Size
Sample source code for this article	ccalc.zip	7KB

© Copyright IBM Corporation 2006
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)