# Using YACC and Lex with C++

Bruce Hahne and Hiroyuki Satō
Mitsubishi Electric Corporation, 5-1-1 Ōfuna. Kanagawa 247, Japan
{bruce,hiroyuki}@isl.melco.co.jp

**Abstract**

We consider one approach to using C++ to write a compiler in combination with the lexical analysis tool Lex and the parser generator tool YACC. This approach uses C++ classes and constructors to build a syntax tree during the parse. Synthesized and inherited attributes are easy to declare for individual terminals and nonterminals, and synthesized attributes can be computed during the parse as long as no inherited attributes are involved in the computation. We also consider the use of virtual functions for tree traversal. A complete example YACC specification is included which demonstrates these techniques. Results show that C++ offers several advantages over C for compiler design with YACC and Lex.

## 1 Introduction

Lex [9] and YACC [8] are two well-known compiler design tools which have been in use since the 1970's. These tools were originally designed for use with C, and examples of how to use the tools almost universally assume C as the base language [1, 5]. More recently, the object-oriented language C++ [11], a superset of ANSI C, has seen an enormous growth in popularity and is now being used in a wide variety of settings. To date, however, minimal attention has been directed towards considering what benefits the features of C++ can offer to a compiler designer or investigating the practical details of using C++ with Lex and YACC. We will address both of these issues, noting some of the benefits of our approach over a C-based implementation and providing a sample YACC specification which applies this approach to build a simple desk calculator.

In the discussion that follows, we assume familiarity with YACC, Lex. and C++.

## 2 Versions of Lex and YACC

Most Unix systems, and all POSIX-compliant systems, include a version of Lex similar to that described in [9]. The Free Software Foundation distributes an improved version, called Flex [10], which is more powerful than standard Lex and which outputs C++-compatible code. Flex is free and its output code carries no restrictions. If your C++ compiler is unable to parse the output of your system's version of Lex, you will need to substitute Flex or some other C++-compatible Lex package if you want to use C++ with a Lex-based tool.

Almost all Unix systems provide a version of YACC similar to that described in [8]. The Free Software Foundation also distributes an improved version of YACC. called Bison, which provides numerous improvements over YACC and which, like Flex, is capable of generating C++-compatible code [4]. However, all Bison-created parser generators carry within them a significant extract of the original Bison code, the result being that they fall under the Free Software Foundation's "copyleft" agreement. which places restrictions on how Bison users can distribute the source and executable code.

A version of YACC with no such restrictions on generated code is Berkeley YACC, or BYACC [2]. As the source code is available, it is possible to modify BYACC to generate C++-compatible code. Our present C++ compiler (**g++** version 2.5.7) will compile BYACC's output code with no modifications, but earlier versions of **g++**, and other C++ compilers, may require a one-line modification to the BYACC source. To make this modification, add the argument **char* name** to the **getenv()** function prototype on line 89 of the **skeleton.c** source file.

Another option for a license-free parser generator is Bison-A2.2 [6] from the Andrew Consortium at Carnegie-Mellon University. This package is a modified version of the Free Software Foundation's Bison 1.22 and includes a command-line switch to produce parser tables without including any of the original Bison code.

## 3 Using YACC with C++

In this section we outline our approach to integrating YACC and C++, with sample code taken from the desk calculator example of section 4. Subsection 3.1 shows how to merge C++ classes with YACC and Lex use, 3.2 demonstrates how to incorporate parse tree node attributes, and 3.3 considers how to use virtual functions to traverse the tree. All of these techniques focus on the use of YACC, and in fact the desk calculator includes its own lexer rather than using Lex. Little additional effort is required for Lex users; see section 3.1 for the details.

The grammar for the desk calculator has four nonterminals, two named terminals (tokens) produced by the lexer, and eleven unnamed terminals. The nonterminals are `list`, `stat`, `expr`, and `number`, the named terminals are `DIGIT` and `LETTER`, and the unnamed terminals include ten mathematical symbols and the newline (`\n`) character.

### 3.1 Parse tree construction using C++

Since C++ programming centers on class design, it is natural that using C++ with YACC depends heavily on the use of classes. The basic steps for incorporating C++ classes into a YACC-based compiler are as follows:

1. Create an abstract base class which will serve as a syntax tree node.
2. Create a "nonleaf" and at least one "leaf" class inherited from the base class. The former will have a data structure to hold child nodes, the latter will not.
3. For each nonterminal in the grammar, create a class which inherits from the nonleaf class. Creating separate classes for each terminal, inheriting from the leaf class, may be necessary for some source languages but not for others.
4. Define the types of the terminals and nonterminals using YACC's `%union`, `%type`, and `%token` directives.
5. Add constructor calls to the Lex and YACC rules to build the syntax tree during the parse.

We should note that these steps are not the only possible approach to using C++ classes within YACC. For example, the free C++ parser CPPP from Brown University [3] does define a large number of classes to correspond to nonterminals, but incorporates these classes into YACC using only three different types within the `%union` directive.

*Step 1:* Base class

The base class should contain only those data items and functions which will be common to all nodes of the syntax tree, both internal nodes and leaves. For the example we will build here, the base class is very small:

```
class treenode
{
        public:
        virtual void print_tree(void) = 0;  // Descend tree, print contents.
};
```

We will discuss the `print_tree` function in more detail in section 3.3. A more complex compiler implementation might include data members such as a pointer to parent `treenode* parent` or a node type indicator such as `short type`. For many implementations, however, such a type indicator will be unnecessary; see the discussion of virtual function use in section 3.3.

*Step 2:* Nonleaf and leaf classes.

The nonleaf class must, at minimum, contain data members to hold the node's children and one or more constructors to build the node. It should inherit from the base syntax tree class. The nonleaf class from the desk calculator example of the next section looks like this:

```
class nonleaf : public treenode
{
        protected:
        treenode** children;
        int childcount;
        public:
        ~nonleaf() { delete [] children; };
        nonleaf();
        nonleaf(treenode*);
        nonleaf(treenode*, treenode*);
        nonleaf(treenode*, treenode*, treenode*);
        void print_tree(void);
        int value;
};
```

There are several points to note here. The **children** data member is a pointer to an array of syntax tree nodes, initialized with a **new** statement when the **nonleaf** constructor is called. The **children** member has **protected** access, instead of **private**, to allow objects which inherit from **nonleaf** to be able to access **children** directly. Note that there are multiple constructors for **nonleaf**, corresponding to the creation of a new **nonleaf** node with zero, one, two, or three children. This ability to use function overloading is an advantage over C, where we would have to use four different function names (**makenode0()** through **makenode3()**, perhaps) to produce a similar effect. The code for the **nonleaf** constructors will allocate space for the children in the **children** array and store the incoming `treenode` pointers into this array. The `print_tree()` function is the same one declared in the `treenode` class. The `value` data item is a node attribute, discussed in more detail in section 3.2.

For our desk calculator example we will not use a general leaf class, but will instead inherit classes for each leaf type directly from the `treenode` class. These classes are less complex than the nonleaf class. A sample is:

```
class digit_leaf : public treenode
{
        public:
        int digit;
        digit_leaf(int digit_in) { digit = digit_in;};
        void print_tree(void) { cout << digit; };
};
```

We see that this class has an attribute **digit** which holds an integer value for this leaf, a single constructor which assigns its **int** argument to the **digit** attribute, and the **print_tree** virtual function.

*Step 3:* Classes for non-terminals

The final step in class creation is to create the individual classes for each nonterminal. Each class will contain any needed attributes, possibly one or more virtual function declarations, and one constructor for each possible number of children of this node type. The class should of course inherit from the **nonleaf** class. As an example, the class for the desk calculator's **list** nonterminal is:

```
class list : public nonleaf
{
        public:
        list();
        list(treenode*);
        list(treenode*, treenode*);
        void print_tree(void);
};
```

The class contains nothing but constructors and the `print_tree` virtual function.

*Step 4:* Defining terminal and nonterminal types in YACC

The next step is to integrate the constructors into YACC's type system by configuring YACC to support C++ classes as pseudo-variables. This requires two steps, as follows.

First, we declare the new types to YACC using the `%union` directive. For our desk calculator, this declaration looks like this:

```
%union   {
        digit_leaf* digit_leaf;
        other_leaf* other_leaf;
        letter_leaf* letter_leaf;
        list* list;
        stat* stat;
        expr* expr;
        number* number;
}
```

The variable names on the left side of each line are C++ class names. The variable names on the right side are the names that YACC is to use when referring to the type. Thus for example, the YACC type named `digit_leaf` corresponds to the type "pointer to C++ class `digit_leaf`". It is not necessary to use the same names as the actual C++ class names, but doing so is a useful notational convenience.

The second step in integrating the C++ classes with YACC is to assign a YACC type to each terminal and nonterminal. For nonterminals this assignment is done with the `%type` directive, enclosing the nonterminal name inside angle brackets and following it with the YACC type name:

```
%type <list> list
%type <stat> stat
%type <expr> expr
%type <number> number
```

The types of the values associated with terminals can be specified either within the `%token` statements which declare tokens returned by Lex, or within any `%left`, `%right`, or `%nonassoc` directive. Examples from the desk calculator code are:

```
%token <digit_leaf> DIGIT
%token <letter_leaf> LETTER
%token <other_leaf> '\n' '=' '(' ')'
%left <other_leaf> '|'
```

These lines assign the YACC type `digit_leaf`, which corresponds to a pointer to the C++ class of the same name, to the token `DIGIT`. Similarly, the other listed tokens are associated with the YACC types specified to their left in angle brackets.

*Step 5:* Constructor calls

The final step in the integration of C++ classes into YACC and Lex is to add constructor calls to the Lex and YACC source code to build the syntax tree. The desk calculator example in section 4 does not use Lex, but the code to use within a Lex specification is similar to that used in the `yylex()` function of the example. The lexer should allocate storage for leaf nodes and initialize them using a `new` call, as in this example, which allocates space for a new `letter_leaf`, calls `letter_leaf`'s constructor to set its `letter` attribute to the character `c` (the constructor is not shown here), and stores a pointer to the resulting object in `yylval.letter_leaf`.

```
yylval.letter_leaf = new letter_leaf(c);
```

The `yylval` is the communication mechanism which allows the lexer to pass values to YACC. When the `%union` scheme of YACC is used, `yylval` is defined as a union of all the types declared in the `%union` directive, so the above line of code simply stores the new `letter_leaf` object into the `yylval` union.

## 3.2 Constructors and Attributes

Once the programmer has built the constructors and incorporated them into YACC and Lex as described above, it is very simple to create synthesized attributes of any type, associated with any parse tree node, and to have YACC assign values to these attributes during the parse. The constructors can also hold inherited attributes, but assigning these during the YACC parse process is difficult since YACC is a bottom-up parser. We have already seen an example of attributes in the parse tree classes: the `nonleaf` class contains one called `value`.

The syntax for instructing YACC to set an attribute is quite simple and can been seen throughout the sample code of section 4. For example, in the production `expr : expr '-' expr` we can see the C++ pseudo-code

```
$$->value = $1->value - $3->value;
```

which performs the subtraction of the `value` attributes belonging to the second and third `expr` nonterminals and assigns the result to the `value` attribute of the first `expr` nonterminal. Here we see one of the advantages of using C++ with YACC: thanks to inheritance, it is possible to have one object which is simultaneously a syntax tree node and a specialized node which contains directly-accessible attributes specific to that terminal or nonterminal.

It is also possible, if all terminals and nonterminals are typed, to move all of the attribute-assignment code out of the YACC specification and into the body of the constructors. For example, for the production "`expr:  '-' expr`" in the desk calculator, the attribute assignment line "`$$->value = -$2->value;`" could be moved into the two-node `expr` constructor, but only if the type of the `node2` formal parameter is changed from `treenode*` to `expr*` to conform to C++ type-safety rules. In other words, the new YACC production would look like this:

```
  | '-' expr %prec UMINUS { $$ = new expr($1, $2); }
```

and the new two-node `expr` constructor would look like this:

```
  expr::expr(treenode* node1, expr* node2) : nonleaf(node1, node2)
    { value = -node2->value; }
```

Moving attribute assignments into constructors in this fashion typically requires many or all terminals to be separately typed to allow C++ to determine at compile-time which constructor to call. In the desk calculator code, moving all of `expr`'s `value` assignments into constructors would require different types for the terminals `+`, `-`, `*`, `/`, `%`, `&`, and `|`, and would require a different `expr` constructor for each of these mathematical operations.

## 3.3 Using virtual functions

Virtual functions are another feature of C++ which can be put to good use when designing a compiler according to the steps outlined in section 3.1. We can use virtual functions to traverse a section of the syntax tree, performing different operations at each node depending on the node's type. One simple example is a function to walk the tree and print out a representation of the contents of each leaf. For such a function, non-leaf nodes need to loop through all of their children and call the printing function recursively, while leaf nodes should produce output which corresponds to their contents. In C, such a function might be implemented by adding an integer type field called `type` to the syntax tree node data structure and writing a switch statement which looks something like:

```
switch(mynode.type)
{
        case EXPR:  /* Call the routine that prints an EXPR */
          break;
        case DIGIT:  /* Call the routine that prints a DIGIT */
          break;
/* etc. */
}
```

For grammars with many symbols, such a switch statement may be quite lengthy. Another C-based approach is to use an array of pointers to function, which requires careful pointer initialization to avoid function miscalls at runtime. In C++, virtual functions replace the need for both the type field and the switch statement. The `print_tree()` virtual function of the C++ desk calculator performs exactly the same task as the C code shown above, but the code generated by the C++ compiler handles the work of choosing which function to call at runtime. The different forms of the `print_tree()` function are all shown in the next section; usage should be self-evident from the code.

## 4 Example

This section contains the complete source code to a simple desk calculator program strongly modeled after the "simple example" which appears in section 10.11 of [8]. The similarity is intentional: since most Unix documentation sets include a copy of [8], it should be easy for most readers to compare the C and C++ versions of the desk calculator. We have converted the code in [8] to C++ and have added constructors, the `print_tree` function, and code to build the syntax tree. The resulting program is longer than the original C version, but it also does more.

To build the desk calculator program, process it with BYACC modified as discussed in section 2. then compile the resulting C++ program `y.tab.c`.

Here is the original description of the features of the desk calculator, taken from [8]:

> ...the desk calculator has 26 registers, labelled 'a' through 'z', and accepts arithmetic expressions made up of the operators +, -, *, /, % (mod operator), & (bitwise and), | (bitwise or), and assignment. If an expression at the top level is an assignment, the value is not printed; otherwise it is. As in C, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As written, the desk calculator also calls `print_tree()` once on the root node after parsing has completed, reproducing all user input by traversing the syntax tree and printing the contents of each leaf node.

The entire YACC specification file for the C++ desk calculator follows.

```
%{
#include <stdio.h>
#include <ctype.h>
#include <fstream.h>
#include <string.h>

/* SYNTAX TREE CLASSES */
class treenode  {                // The abstract base class for all nodes.
        protected:
        virtual ~treenode(){};
        public:
        virtual void print_tree(void) = 0;  /* Descend tree, print contents.*/  };

class nonleaf : public treenode {  // All non-leaf classes inherit from this.
        protected:
        treenode** children;
        int childcount;
        public:
        ~nonleaf() { delete [] children; };
        nonleaf();
        nonleaf(treenode*);
        nonleaf(treenode*, treenode*);
        nonleaf(treenode*, treenode*, treenode*);
        void print_tree(void);
        int value;  };

class list : public nonleaf {  // The top-level nonterminal
        public:
        list();
        list(treenode*);
        list(treenode*, treenode*);
        void print_tree(void);  };
```

```
class stat : public nonleaf {  // An expression or an assignment to register.
        public:
        stat(treenode*);
        stat(treenode*, treenode*, treenode*);  };

class expr : public nonleaf {  // Any expression.
        public:
        expr(treenode*);
        expr(treenode*, treenode*);
        expr(treenode*, treenode*, treenode*);  };

class number : public nonleaf {  // An integer in base 8 or base 10.
        public:
        number(treenode*);
        number(treenode*, treenode*);  };

class letter_leaf : public treenode {  // Register access.
        public:
        char letter;
        letter_leaf(char char_in)  { letter = char_in; };
        void print_tree(void) { cout << letter; };  };

class digit_leaf : public treenode {  // A single digit.
        public:
        int digit;
        digit_leaf(int digit_in) { digit = digit_in;};
        void print_tree(void) { cout << digit; };  };

class other_leaf : public treenode {  // A mathematical symbol or a newline.
        public:
        char contents;
        other_leaf(char contents_in) { contents = contents_in; };
        void print_tree(void) { cout << contents; };  };
/* END SYNTAX TREE CLASSES */

int regs[26];  // Registers used for the letters a-z.
int base;  // We can work in base 8 or base 10.
treenode* root;  // Save the root for later use.
%}

%union   { digit_leaf* digit_leaf;
           other_leaf* other_leaf;
           letter_leaf* letter_leaf;
           list* list;
           stat* stat;
           expr* expr;
           number* number;  }

/* TERMINALS, NONTERMINALS, AND START SYMBOL */
%start list
%token <digit_leaf> DIGIT
%token <letter_leaf> LETTER
%token <other_leaf> '\n' '=' '(' ')'
%left <other_leaf> '|'
%left <other_leaf> '&'
%left <other_leaf> '+' '-'
%left <other_leaf> '*' '/' '%'
%left <other_leaf> UMINUS  /* Handle unary minus precedence. Not a real token. */
%type <list> list
%type <stat> stat
%type <expr> expr
%type <number> number

%%
/* YACC RULES */
list :  { $$ = new list();
          root = $$;  }
        | list stat '\n'  {
                $$ = new list($1, $2);
                root = $$;  }
        | list error '\n'  {
                $$ = new list($1);
                root = $$;
                yyerrok;  }         ;

stat : expr  {
                $$ = new stat($1);
                cout << $1->value << '\n'; }
        | LETTER '=' expr  {
                $$ = new stat($1, $2, $3);
                regs[$1->letter-'a'] = $3->value;  }        ;

expr : '(' expr ')'  {
                $$ = new expr($1, $2, $3);
                $$->value = $2->value;  }
        | expr '+' expr  {
                $$ = new expr($1, $2, $3);
                $$->value = $1->value + $3->value;  }
        | expr '-' expr  {
                $$ = new expr($1, $2, $3);
                $$->value = $1->value - $3->value;  }
        | expr '*' expr  {
                $$ = new expr($1, $2, $3);
                $$->value = $1->value * $3->value;  }
        | expr '/' expr  {
                $$ = new expr($1, $2, $3);
                $$->value = $1->value / $3->value;  }
        | expr '%' expr  {
                $$ = new expr($1, $2, $3);
                $$->value = $1->value % $3->value;  }
        | expr '&' expr  {
                $$ = new expr($1, $2, $3);
                $$->value = $1->value & $3->value;  }
        | expr '|' expr  {
                $$ = new expr($1, $2, $3);
                $$->value = $1->value | $3->value;  }
        | '-' expr  %prec UMINUS  {
                $$ = new expr($1, $2);
                $$->value = -$2->value;  }
        | LETTER  {
                $$ = new expr($1);
                $$->value = regs[$1->letter-'a'];  }
        | number  {
                $$ = new expr($1);
                $$->value = $1->value;  }        ;

number : DIGIT  {
                $$ = new number($1);
                $$->value = $1->digit;
                base = ($1->digit == 0) ? 8 : 10;  }
        | number DIGIT  {
                $$ = new number($1, $2);
                $$->value = base * $1->value + $2->digit;  }
%%
/* The lexer function which YACC calls to fetch tokens. */
yylex() {
        int c;
        while((c = getchar()) == ' ') {}
        if(islower(c))  {
                yylval.letter_leaf = new letter_leaf(c);
                return(LETTER);        }
        if(isdigit(c))  {
                yylval.digit_leaf = new digit_leaf(c-'0');
                return(DIGIT);        }
        yylval.other_leaf = new other_leaf(c);
        return(c);  }

/* Standard error function expected by YACC. */
yyerror(char* s)  { cerr << "Parse error\n"; }
```

```
int yyparse();  /* YACC will generate a yyparse() function for us.*/
main()  {
        yyparse();
        cout << "\nThe parse tree is:";
        root->print_tree();
        cout << '\n';  }

/* SYNTAX TREE CLASS CONSTRUCTORS, DESTRUCTORS, AND MEMBER FUNCTIONS */
nonleaf::nonleaf()  {
        childcount = 0;
        value = 0;  }
nonleaf::nonleaf(treenode* node1)  {
        childcount = 1;
        children = new treenode*[1];
        children[0] = node1;
        value = 0;  }
nonleaf::nonleaf(treenode* node1, treenode* node2)  {
        childcount = 2;
        children = new treenode*[2];
        children[0] = node1;
        children[1] = node2;
        value = 0;  }
nonleaf::nonleaf(treenode* node1, treenode* node2, treenode* node3)  {
        childcount = 3;
        children = new treenode*[3];
        children[0] = node1;
        children[1] = node2;
        children[2] = node3;
        value = 0;  }

void nonleaf::print_tree(void) { // Used for every nonleaf except class LIST
       int x;
       for(x=0; x<childcount; ++x)
         children[x]->print_tree();  }

list::list() : nonleaf(){}
list::list(treenode* node1) : nonleaf(node1){}
list::list(treenode* node1, treenode* node2) : nonleaf(node1, node2){}

void list::print_tree(void)  {
  int x;
  for(x=0; x<childcount; ++x)
        children[x]->print_tree();
  cout << '\n';  }

// Note that each constructor below invokes the nonleaf constructor
// to build the syntax tree node.
stat::stat(treenode* node1) : nonleaf(node1){}
stat::stat(treenode* node1, treenode* node2, treenode* node3)
        : nonleaf(node1, node2, node3){}
expr::expr(treenode* node1) : nonleaf(node1){}
expr::expr(treenode* node1, treenode* node2) : nonleaf(node1, node2){}
expr::expr(treenode* node1, treenode* node2, treenode* node3)
        : nonleaf(node1, node2, node3){}
number::number(treenode* node1) : nonleaf(node1){}
number::number(treenode* node1, treenode* node2) : nonleaf(node1, node2){}
```

## 5 Conclusion

We have presented one method of substituting C++ for C in a YACC-based compiler. The example techniques shown here suggest, at minimum, that C++ makes designing a parse tree with attributes more elegant than doing the same within C and that using virtual functions can avoid the need for long and potentially error-prone `switch` statements. At Mitsubishi, we have successfully adopted the methods shown here in our work on a compiler for a new C++-based parallel language.

## 6 Acknowledgments

Several of the methods covered in this paper were inspired by a post to the Usenet newsgroup comp.compilers in January 1994 by Richard Hipp (`drh@world.std.com`), who kindly archived the feedback he received and sent a copy to all interested readers.

We are also thankful to Mark Hopkins (`markh@csd4.csd.uwm.edu`) for maintaining the *Catalog of Free Compilers and Interpreters* [7], without which we most likely never would have discovered BYACC or CPPP.

John Interrante (`interran@uluru.stanford.edu`) and Vern Paxson (`vern@ee.lbl.gov`) provided valuable feedback on a preliminary version of this paper.

## References

[1] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[2] Corbett, Robert. BYACC parser generator version 1.9, 1993, available via anonymous ftp at `ftp.cs.berkeley.edu:/ucb/4bsd/byacc.tar.Z`

[3] Davis, Tony and Steven P. Reiss. Documentation and source code for CPPP version 1.61, Brown University, 1994. Available via anonymous ftp at `wilma.cs.brown.edu:/pub/cppp.tar.Z`.

[4] Donnelly, Charles and Richard Stallman. *The Bison Manual*, Free Software Foundation, Inc., Cambridge, Massachusetts, 1990. Available via anonymous ftp as part of the Bison distribution at `prep.ai.mit.edu`.

[5] Fischer, Charles N. and Richard J. LeBlanc, Jr. *Crafting a Compiler with C*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[6] Hansen, Wilfred J. Bison version A2.2, Andrew Consortium, Carnegie Mellon School of Computer Science, April 1994. Available via anonymous ftp at `ftp.andrew.cmu.edu` in `/pub/AUIS/bison/`.

[7] Hopkins, Mark. "Catalog of Free Compilers and Interpreters", monthly posting to Usenet group comp.compilers, 1993. Available via anonymous ftp at `rtfm.mit.edu`.

[8] Johnson, S.C. "YACC - Yet Another Compiler Compiler," Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[9] Lesk, M. E. "Lex - a Lexical Analyzer Generator," Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[10] Paxson, Vern. On-line manual pages distributed with the Flex software package, 1990, available via anonymous ftp from `ftp.ee.lbl.gov` or `prep.ai.mit.edu`.

[11] Stroustrup, Bjarne. *The C++ Programming Language*, second edition, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.