THE NEWYACC USER'S MANUAL

Elizabeth L. White   John R. Callahan   James M. Purtilo

Computer Science Department
University of Maryland
College Park, Maryland 20742

This manual introduces NewYacc, a parser generator system built upon the original yacc system within Unix [3]. NewYacc's principle extension is to provide users with a way to associate rewrite rules with individual productions in the language grammar. These rules are used to describe how the parse tree (which is saved in NewYacc but not in original yacc) should be traversed, plus users can easily control what action is performed at each node in the tree during their traversals. This provides users with great leverage in the construction of a variety of source to source translation tools. This manual assumes a general familiarity with original yacc.

The research leading to construction of NewYacc was supported by Office of Naval Research under contract N0014-90-J4091.

1 INTRODUCTION

NewYacc, an extension of the UNIX tool Yacc[3], is a tool which allows users to do source-to-source transformations on input languages. A Yacc specification consists of grammar rules describing the input structure, C code to be invoked when rules are recognized, and a low-level lexical analysis function which divides the input into tokens. From this specification, Yacc generates a C function which calls the lexical analysis routine, parses the input and invokes actions when grammar rules are recognized. A specification for NewYacc contains all the elements of a Yacc specification with the addition of user-defined traversal specifications. The NewYacc generated function also parses the input and invokes actions when grammar rules are recognized. However, once the input has been accepted as a legal string in the language defined, the parse tree for the input can be traversed and additional actions performed using the user-defined traversal specifications. These traversals loosely correspond to attribute-dependencies in traditional attribute grammar systems. The idea is closely related to generalized syntax-directed translation schema (GSDTS) [2] from which attribute-based approaches were developed.

The functionality of many tools can be discussed in terms of the input language structure as defined by its grammar. This suggests an alternative to the low-level approach traditionally used for tool development. The idea is to adapt input at a high level so that existing compilers, code generators and other processing tools can be employed without alteration. Users need the ability to do high-level source-to-source translations quickly and easily by specifying what needs to be done in terms of the language constructs themselves. Traditionally, attribute grammar-based approaches have been used to perform such actions during a parse. NewYacc is less powerful than most attribute-based systems, but it is simple to employ and sufficiently powerful to deal with a wide class of translation tasks. It has proven to be an ideal "first-pass" tool in many applications, allowing reuse of language-independent tools. Some examples can be found in [5].

One type of tool which is easy to develop using NewYacc is one which outputs a transformation on its input. An example of such a tool is an instrumenter. If these desired transformations can be defined in terms of the input structure, the grammar defining the input language can be augmented to associate actions with different productions of the grammar such that the resulting tool builds a parse tree for a given input, traverses this tree, and outputs a filtered, augmented, or altered version.

A tool which extracts information from the input for the purposes of computing some type of statistics is also easily constructed. The input grammar can be augmented such that function calls and/or outputs are made at specific points in the parse tree during a selective traversal. These calls and outputs can be analyzed at the end of processing to compute the desired statistics.

If the tool's functionality requires multiple passes over a segment of the parse tree or that the parse tree be traversed in an unusual way, this can also be done easily using NewYacc. The subtrees of a node in the parse tree of an input to the tool can be traversed in any order and any number of times the tool developer specifies. Traversals can even be conditional, based on the information in another part of the parse tree.

NewYacc allows the use of dynamically scoped reference variables. Reference variables allow information from one section of the parse tree to be stored and used at other sections of the parse tree. This feature is very useful when developing tools which require the use of context-sensitive information. Reference variables are also useful when used as a “macro” packager for commonly used auxiliary stacks in Yacc. Opening a scope and declaring a new reference variable is analogous to pushing an item on a stack.

Numerous tools have already been developed using the NewYacc tool. These tools include a C cinema tool, a C profiler, an Ada static analyzer program and an Ada instrumenter.

Users who are unfamiliar with the tool Yacc should read the Yacc manual before continuing with this document. The next section gives a brief introduction to Yacc but is insufficient for those who have never used the tool. Those readers with a strong understanding of Yacc should skip to Section 3.

Use of the NewYacc tool is introduced in this document by way of example. Section 3 discusses what NewYacc specifications look like and how they are developed using two sample applications which the reader can implement if desired. Many (although not all) of the features of NewYacc are described in this section. All of the features of NewYacc are detailed along with examples in Section 4. These two sections may be read in either order.

2 YACC

This section contains a brief discussion of Yacc for those users who have some familiarity with the tool. Much of what follows is paraphrased from the Yacc documentation, but only aspects of Yacc which are important to understanding and using NewYacc are mentioned. In particular, only the syntax of Yacc specifications, lexical analysis, and the environment are discussed. Readers who are interested in Yacc actions, error handling, and an overall description of how the resulting Yacc and NewYacc parsers work are referred to the Yacc documentation.

2.1 BASIC SPECIFICATIONS

A Yacc specification file looks like:

```
declarations
%%
rules
%%
programs
```

The declaration and program sections may be empty.

The declarations section is used to declare the tokens (using %token) and the start symbol (%start) as well as give other pieces of information to the specification (%left, %right, %nonassoc, etc.). Declarations for C code can appear in the declarations section enclosed in the marks "%{" and "%}". These declarations have global scope, so they are known to the specification statements and the lexical analyzer. For example,

```
%{
   char * a ;
%}
```

can be placed in the declarations section making *a* accessible to all parts of the program. The Yacc parser only uses names beginning in "yy" (the parser produced by NewYacc also uses names beginning with "ny"); the user should avoid such names.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A : BODY ;

"A" represents a nonterminal name and "BODY" represents a sequence of zero or more names and literals. The colon and semicolon are Yacc punctuation. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

The program section contains code which will be included in the resulting parser. Any user-defined functions can be placed here if desired.

2.2 LEXICAL ANALYSIS

The user must supply a lexical analyzer to read the input stream and communicate tokens to the parser. The lexical analyzer is an integer-valued function called *yylex*. The function returns an integer token number representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable *yylval*.

A very useful tool for constructing lexical analyzers is the *Lex*[4] program, which is designed to work in harmony with Yacc parsers.

2.3 ENVIRONMENT

When the user inputs a specification to Yacc, the output is a file of C programs called *y.tab.c* (on most systems). The primary function produced by Yacc is called *yyparse*. When *yyparse* is called, it repeatedly invokes *yylex* to obtain an input token. The user must provide a certain amount of environment to this parser in order to develop a working program. A program called *main*, which calls *yyparse*, and a function called *yyerror*, which prints a message when a syntax error is detected, are necessary. These two routines may be supplied by the user or the default versions in the Yacc library may be used.

3 NEWYACC

Before going into the details of NewYacc and NewYacc traversals, it is easier to look at a small example of its use. This section introduces a simple programming language SAMPL and two different tools (both of which operate on SAMPL program text) which can easily be implemented using NewYacc. Section 2.1 contains a specification of the language itself and a description of two tools: a pretty-printer and a flow graph extracter. Development of an application tool using NewYacc involves several steps which are outlined in Section 2.2. The final subsection follows these steps to show how the pretty printer and the flow graph extracter might be implemented using NewYacc. The two NewYacc specifications given use features of NewYacc which are discussed in depth in Section 4. Complete code for these two tools can be found in Appendix B.

3.1 SAMPL

Let us suppose that someone has implemented a new programming language called SAMPL. You are asked to develop some SAMPL tools — a pretty-printer and a flow-graph tool. These tasks will require different approaches: (1) the pretty-printer will *augment* SAMPL programs with additional text and (2) the flow-graph tool will *extract* information from SAMPL programs. Figure 2 displays a small SAMPL program and Figure 3 shows both the pretty-printed version and the flow graph for this program.

You do not have access to the SAMPL compiler, but you do have the SAMPL grammar written in BNF notation shown in Figure 1. Parser generator tools like Yacc and Lex are necessities here, yet the tasks will be tedious even with the use of such tools. To implement the pretty printer, for example, each grammar rule will have to be augmented with actions that echo the input matching that rule. Furthermore, Yacc and Lex do not buffer input tokens. Your parser will have to save and echo input tokens explicitly.

The flow-graph tool will need to output a graph encoded as a list of lists. A list represents a "basic block" of code that may include nested blocks. For each block in the code there is a list with the following format:

$$(\text{blocknum } ((\text{varlist})(\text{branchlist})(\text{stmt}_1) \; . \; . \; . \; (\text{stmt}_n)))$$

Each **blocknum** is of the form "B" followed by a unique number for each block. The **varlist** is a list of variables referenced in the conditional part of the statement (either a `while` or `if`) which encloses the block. The **branchlist** is a list of blocks to branch to upon exit. If the block is contained in a loop, this **branchlist** will contain a self-reference. There is also a list entry for each statement or block contained within the current block. The output is used as input to a

```
<program>          ::=    <block>

<block>            ::=    BEGIN <statement_list> END

<statement_list>   ::=    /* empty */
                   |      <statement> <statement_list>

<statement>        ::=    IF <expression> THEN <block>
                   |      WHILE <expression> DO <block>
                   |      IDENTIFIER ASSIGNOP <expression>

<expression>       ::=    <expression> <binaryop> <expression>
                   |      LPAREN <expression> RPAREN
                   |      IDENTIFIER
                   |      LITERAL
                   |      NOT LPAREN <expression> RPAREN

<binaryop>         ::=    EQUALS     |    LESSTHAN
                   |      GRTRTHAN   |    PLUS
                   |      DASH       |    STAR
                   |      SLASH
```

Figure 1: The SAMPL grammar.

language-independent tool(written in Common Lisp) that performs various flow-graph analysis functions.

3.2 USING NEWYACC

Parsed text can be represented as a tree — a parse tree — consisting of a root node, internal nodes, and leaf nodes. Each node is an instance of a rule in the grammar used to parse the input. To augment input or extract information contained in a parse tree, one must follow a path through the parse tree and output the required information once found. NewYacc allows users to specify traversals through the parse tree and actions to be performed at different points in the traversal.

3.2.1 TRANSLATIONS NewYacc is a superset of Yacc. NewYacc specification files are similar to the Yacc specification files as shown in Section 2, but have two differences:

1. The "%tag" specifiers in the declarations section.
2. Rule translations appended to some grammar rules (after the last Yacc semantic action).

The "%tag" specifier is placed in the declarations section along with other Yacc specifiers (e.g., %token, %left, %right, %start). It simply lists all of the display masks (see below) used in all

```
begin x := input
while x > 0 do begin
y := x*x+y output := y
x := input end end
```

Figure 2: A SAMPL program .

```
begin                                   (setq fg (quote
     x := input                              (B0 ( () ()
     while x > 0 do begin                         (B7 (x (input)))
          y := x*x+y                              (B26 ( (x) (B26)
          output := y                                  (B51 (y (x x y)))
          x := input                                   (B62 (output (y)))
     end                                               (B74 (x (input)))
end                                               ))
                                             ))
(a)                                     ))
                                        (b)
```

Figure 3: Pretty-printed (a) and flow-graph (b) versions of the SAMPL program in Figure. 2.

NewYacc translations. Masks are used to label and direct traversals through a parse tree. They roughly correspond to attributes used in more traditional approaches.

Translations are attached to the end of a grammar rule, after the last Yacc semantic action but before the semicolon that ends the rule. For example, the pretty-printer should separate each token in a SAMPL statement of the form "if x then y" by a single space character. To do this, the grammar rule

statement : IFR exp THENR block

would be appended with the translation

[(PRETTYP) #1() " " #2 " " #3() " " #4]

in order to specify a separation between terminals and non-terminals. Translations may also be attached to groups of rules. The group of statement rules in Figure 1 would be annotated for the pretty-printer as follows:

```
statement       :    IFR exp THENR block
                     [ (PRETTYP) #1() " " #2 " " #3() " " #4 ]
                |    WHILER exp DOR block
                     [ (PRETTYP) #1() " " #2 " " #3() " " #4 ]
                |    IDENTIFIER ASSIGNOP exp
                     [ (PRETTYP) #1() " " #2 " " #3() ]
                ;
```

The translations above are labelled with the PRETTYP mask. Each #n in the body of the translation refers to the n^{th} symbol on the right-hand side of the grammar rule. A *symbol* represents a non-terminal or terminal corresponding to a subtree or token (respectively) in the parse. Yacc semantic actions are not included in the enumeration of symbols. Symbols can be used repeatedly in a translation and in any order. The "()" following a #n in a translation indicates that the subtree or token is to be output literally. NewYacc handles the details of token storage. Translation may also include other objects: strings, references, conditionals, blocks, and assignments.

Each translation is supposed to reflect an "unparsing" of tokens that match a grammar rule [6]. A traversal starts at the root of a constructed parse tree and applies each matched translation as it is encountered along the traversed path. Traversals are *masked* to match translations with a particular mask. For example, a PRETTYP traversal from the root of a SAMPL parse tree would eventually encounter the PRETTYP translations on the SAMPL statement rules.

Translations also control the paths of traversals. Some of the #n symbol references in the statement translations above, for instance, do not represent tokens, but subtrees (non-terminals). In these cases, the subtrees are traversed recursively with the PRETTYP mask. If a rule does not have a translation matching a traversal, all of the subtrees are traversed in the order they appear on the right-hand side of the grammar rule.

3.2.2 MAIN PROGRAMS The main program for a NewYacc application looks similar to that used in a Yacc application. The function *yyparse* is called first to perform the parse, executing the semantic actions attached to grammar rules while constructing the parse tree. If this is done successfully, the parse tree may be traversed by calling the function *nyprint* which has the form:

nyprint(outputf,mask,traversaltype,filterstrings)

where

outputf	is the name of the output function.
mask	is a mask (one of the %tag symbols).
traversaltype	is either NY_OPEN or NY_SELECT.
filterstrings	is either 0 (TRUE) or 1 (FALSE).

The simplest possible main program is one which first parses the input stream into a parse tree using *yyparse* and then uses *nyprint* to perform a traversal using a mask. It also contains an include file, *nytags.h*, for the NewYacc tags used. This program looks like:

```
#include "nytags.h"
main()
{
    if (yyparse() == 1) error(-1);
    nyprint(outputf, mask, traversaltype,filterstring);
}
```

Traversals are initiated by this function at the root of a parse tree. The function *nyprint* may be invoked multiple times after the parse because the tree is not discarded until the program exits. The mask argument is simply a %tag symbol used to match translations with the same label in the parse tree. The traversal type argument is either *open* or *selective*. These concepts will be discussed in the Features section. The string filter argument is used to specify whether or not a "\" character should be placed in front of other "\"'s produced by the traversal. This argument is necessary because the "\" is a special character in the C programming language.

An output function is called each time a string is produced during a traversal. This function is user-specified so that the programmer might collect tokens in an array, direct output to some file under user control, or send output to standard output. The output function written by the user should have one parameter of type pointer to character. One of the simpliest possible output functions is one which simply uses *printf* to print the string to standard output.

```
outputf(s)
char *s;
{
    if (s)
        printf("%s",s);
}
```

There is no default output function.

3.2.3 INVOKING NEWYACC TOOL

There are several option available when invoking the NewYacc tool:

-f infile	specifies the input grammar file
-l	leave the NewYacc and Yacc intermediate files used in the current directory
-s	use stdin for grammar input instead of naming a file
-d	turn on Yacc debugging
-Y yaccfile	the yacc to use after NewYacc preprocessing (the default is /usr/bin/yacc)

3.3 USING NEWYACC TO DEVELOP SAMPL APPLICATIONS

We can view the tasks of pretty-printing and flow graph extraction as traversals of a tree representing the parsed input text with the input tokens at the leaves of the tree. A traversal starts at the root of the tree and proceeds down subtrees to the leaves. NewYacc saves the entire parse tree (with the tokens as leaves) in memory so that application programs can traverse the tree in order to augment or extract context-sensitive information. If some subtree of the parse tree should be echoed verbatim, one can specify this at the root of the subtree in the grammar instead of at the leaves. Likewise, one can select information from subtrees at their root without the need for explicitly managing global variables or paths of inherited and synthesized attributes.

Pretty-printing a SAMPL program involves outputting all of the leaves of the parse tree for the program with spaces, tab characters, and carriage returns placed in certain spots. Figure 4 shows a NewYacc specification which will produce a parser to perform this operation. This specification only uses one display mask PRETTYP which is included in the declarations section using a "%tag". Three built-in functions, *tab*, *tabincr*, and *tabdecr*, are declared and used. Rule translations were added to the input grammar:

- In the production for "program", a translation was added to put a carriage return at the end of the input stream.
- In the production for "block", a translation was added which will put a carriage return after the BEGIN token, increment the current tabbing level (to supply the correct indentation), pretty-print out the statement list inside the block, then decrement the tabbing level before printing the END token.
- In the production for "statement_list", a translation was added to insert a carriage return between statements.

- In the productions for "statement", translations were added to insert blanks between the elements of a statement.
- In the productions for "expression", translation were added to insert blanks between the elements of an expression.

It should be fairly easy for the reader to see that these rule translations will produce the desired result if the tree is traversed in such a way that the leaf nodes are visited in the same order as the tokens they represent occur in the original program. This can be achieved using an *open* traversal, which is discussed in the Features section.

The flow graph producer for SAMPL programs is a more complex tool. Here, the output is a Lisp-like structure rather than the input program. The parse tree needs to be traversed as in the pretty-printer, but the purpose of the traversal is the extraction of information without outputting the input. A different kind of traversal, a *selective* traversal, is needed here. This traversal type is also discussed in the Features section.

A specification for the flow graph tool is shown in Figure 5. The grammar has been modified slightly from the original specification of SAMPL to simplify the computation of the location of the first character of an expression. This modification is not necessary, but serves to increase readability of the specification. The specification given uses three display masks, BLOCKS, BIN, and POS. It also uses three reference variables (which are discussed in the Features section), b, c and e and a built-in function *bcharno*. The "&" character indicates selective traversal of the indicated subtree or token.

As can be seen in the specification, each block is numbered by the character position within the source file of the first lexical token of that block. This ensures each block has a unique number. In order to keep track of the current block number, the parser will have to manage a global stack of block contexts as the input is parsed. This is done through the use of the reference variables b, c and e. The variable b always contains the current block number, while e holds the branch list for the block and c holds the variable list. Several rule translations are necessary to specify the information extraction:

- In the production for "program", a translation was added to declare an occurrence of the three reference variables and set them to hold the information about the outermost block of the program.
- In the production for "block", a translation was added which outputs the block information computed previously, followed by a traversal of the statements contained within the block.
- In the production "statement ::= IFR exp THENR block", a translation was added which computes the block number using a traversal on the second subtree with mask POS, computes the variable list by traversing the same list with mask BIN, and then traverses the block.

```
%{
                char *tab(),*tabincr(),*tabdecr();
%}
%tag            PRETTYP
%token          BEGINR ENDR IFR THENR WHILER DOR IDENTIFIER ASSIGNOP
%token          EQUALS LESSTHAN GRTRTHAN NOTR PLUS DASH STAR SLASH
%token          LPAREN RPAREN LITERAL
%left           EQUALS LESSTHAN GRTRTHAN PLUS DASH STAR SLASH
%start          program
%%
program         :       block
                        [ (PRETTYP) #1 "\n" ]
                ;
block           :       BEGINR statement_list ENDR
                        [ (PRETTYP) #1() "\n" tabincr()
                                    #2 tabdecr() tab() #3() ]
                ;
statement_list  :  /* empty */
                |       statement statement_list
                        [ (PRETTYP) tab() #1 "\n" #2 ]
                ;
statement       :       IFR expression THENR block
                        [ (PRETTYP) #1() " " #2 " " #3() " " #4 ]
                |       WHILER expression DOR block
                        [ (PRETTYP) #1() " " #2 " " #3() " " #4 ]
                |       IDENTIFIER ASSIGNOP expression
                        [ (PRETTYP) #1() " " #2 " " #3() ]
                ;
expression      :       expression binaryop expression
                        [ (PRETTYP) #1() " " #2() " " #3() ]
                |       LPAREN expression RPAREN
                        [ (PRETTYP) #1 " " #2() " " #3]
                |       IDENTIFIER
                |       LITERAL
                |       NOTR LPAREN expression RPAREN
                        [ (PRETTYP) #1 " " #2 " " #3() " " #4]
                ;
binaryop        :       EQUALS      |       LESSTHAN
                |       GRTRTHAN    |       PLUS
                |       DASH        |       STAR
                |       SLASH
                ;
%%
#include "lex.yy.c"
```

Figure 4: A solution to pretty-print SAMPL programs.

- In the production "statement ::= WHILER exp DOR block", a translation was added which computes the block number using a traversal on the second subtree with mask POS, computes the branch list the same way, computes the variable list by traversing the same list with mask BIN, and then traverses the block.
- In the production "statement ::= IDENTIFIER ASSIGNOP exp", a translation was added to print out the unique number for the statement, the identifier being modified, and the identifiers which are being used in this modification.
- In the production for "exp", a translation was added to return the position within the input files of the character at the beginning of the expression.
- In the production "expression ::= IDENTIFIER", a translation was added which prints out the identifier followed with a space.

Figure 6 shows the commands necessary to create both of the SAMPL tools in a Unix environment. The necessary commands are shown following the "%" symbol with the parts for which complete pathnames must be added in **boldface**. Output from the tools is shown in *italics*. A lex file for processing tokens (*sampl.l*), a NewYacc specification file for both the pretty-printer and flow graph program (*sampl.ny*), a main program (*main.c*) and a sample makefile are all given in Appendix B. There are two tools used here which have not been previously mentioned. *nylexfix.lex* and *nytabcfix.yacc* are used to do some final editing of the lex and yacc files before compilation.

4 FEATURES

This section discusses the features of the NewYacc tool. The two different methods of tree traversal, open and selective, are explained in Section 4.1. NewYacc specifications may also contain built-in functions, user-defined functions, reference variables and statements. Use of built-in and user-defined functions are described in Sections 4.2 and 4.3. Section 4.4 discusses reference variables, and Sections 4.5 and 4.6 describe the block statement and conditional statement types.

4.1 TRAVERSALS

There are two types of nodes in a NewYacc produced parse tree of an input source file. Interior nodes represent some production rule of the input grammar for the language being parsed. Leaf nodes are produced for some string which is a terminal symbol of the language and has no associated subtrees.

A *traversal* in NewYacc is a user-specified dynamic walk through the parse tree and corresponds to a single translation of the input. The display masks and grammar symbols control the path of the traversal. Traversals always begin at the root with an associated display mask and proceed

```
%{
                char *b,*c,*e,*bcharno();
%}
%tag            BLOCKS BIN POS
%token          BEGINR ENDR IFR THENR WHILER DOR IDENTIFIER ASSIGNOP
%token          EQUALS LESSTHAN GRTRTHAN NOTR PLUS DASH STAR SLASH
%token          LPAREN RPAREN LITERAL
%left           EQUALS LESSTHAN GRTRTHAN PLUS DASH STAR SLASH
%start          program
%%
program         :       block
                        [ (BLOCKS) !b !c !e
                              @b="B0" @e="()" "(setq fg (quote "
                              @c="()" #1 "))\n"]
                ;
block           :       BEGINR statement_list ENDR
                        [ (BLOCKS) "(" @b " (" @c "("
                              @e ") " #2 "))"]
                ;
statement_list  :
                |       statement statement_list
                ;
statement       :       IFR exp THENR block
                        [ (BLOCKS) !b !c !e
                              @b="B"+&2(POS)
                              @c="("+&2(BIN)+")" #4 ]
                |       WHILER exp DOR block
                        [ (BLOCKS) !b !c !e
                              @b="B"+&2(POS)
                              @e="B"+&2(POS)
                              @c="("+&2(BIN)+")" #4 ]
                |       IDENTIFIER ASSIGNOP exp
                        [ (BLOCKS) "(B" bcharno()
                              " (" #1() " (" &3(BIN) ")))" ]
                ;
exp             :       expression
                        [ (POS) bcharno() ]
                ;
expression      :       expression EQUALS expression
                |       expression LESSTHAN expression
                |       expression GRTRTHAN expression
                |       expression PLUS expression
                |       expression DASH expression
                |       expression STAR expression
                |       expression SLASH expression
                |       LPAREN expression RPAREN
                |       IDENTIFIER
                        [ (BIN) #1() " " ]
                |       LITERAL
                |       NOTR LPAREN expression RPAREN
                ;
%%
#include "lex.yy.c"
```

Figure 5: A solution to extract SAMPL flow-graphs.

% lex sampl.l
% **nylexfix.lex** lex.yy.c
% **newyacc** -f sampl.ny -d
Adding display tag BLOCKS
Adding display tag BIN
Adding display tag PRETTYP
Adding display tag POS
Executing /usr/bin/yacc -vd ./ny.temp.y
% **nytabcfix.yacc** y.tab.c
% cc -c main.c
% cc -c y.tab.c
% cc -o sampl main.o y.tab.o **/jteam/callahan/lib/libny.a** -ll

Figure 6: Creating the SAMPL tools on a Unix system .

downward according to the NewYacc rule translations encountered. The grammar symbols and display masks control the path of the parse tree traversal. When a node is encountered during a traversal, there is always a current display mask. If this node has a rule translation with a mask that matches the current mask, then the elements of this translation are evaluated from left to right. During evaluation, literal strings in the rule translation are output and the subtrees indicated by grammar symbols are traversed. A subtree traversal by default uses the same display mask and traversal type as that of the current node, but this can be changed. These changes will be discussed later in this section. If there is no matching display mask, the default action depends upon whether the current traversal type is open or selective. After the entire rule translation has been evaluated, control returns to the parent of the node.

4.1.1 OPEN TRAVERSALS A grammar symbol reference containing the "#" character in a NewYacc rule translation or the use of the `NY_OPEN` option in an invocation of *nyprint*() at the root specifies an open traversal. In the default case for open traversals, the subtrees for both the terminals and non-terminals are traversed in the same order they occur on the right-hand side of the rule. For a production of the form:

A : B c D e

(where uppercase letters represent non-terminals and lowercase letter represent terminals) the default action during an open traversal would be:

[#1 #2 #3 #4]

As a simple example of open traversals, consider a grammar which recognizes lists of ITEMs. If the necessary task was the reversal of this input list, a selective traversal could do this quite well:

```
tlist       :   list
                [ (REVERSE) #1 "\n" ]
            ;
list        :   list ITEM
(**)            [ (REVERSE) #2 " " #1 ]
            |   ITEM
                [ (REVERSE) #1 " " ]
            ;
```

The # character followed by a number n is a grammar symbol which corresponds to the n-th element on the right hand side of the production. These symbols indicate a traversal of the indicated subtree. In the line marked with "(**)", the #2 indicates traversal of the second subtree (which literally outputs the ITEM), followed by the output of the literal string " ", and finally the traversal of the first subtree, the rest of the list. It is easy to see that this traversal will perform the necessary reversal. A NewYacc specification which makes the input into a palindrome (i.e. transforms the list "a b c" to the list "a b c c b a") would be a simple extension which the reader may want to try.

As another example of open traversals, consider a set of productions, Yacc specifications and NewYacc actions for specifying two traversals, one to produce prefix output and the other to produce postfix output.

```
Expression  :   '(' Expression ')'                  { $$ = $2 }
                [(PREFIX,POSTFIX) #2]
            |   Expression '+' Expression           { $$ = $1 + $3 }
                [(PREFIX) #2 " " #1 " " #3]
(**)            [(POSTFIX) #1 " " #3 " " #2 " "]
            |   Expression '−' Expression           { $$ = $1 - $3 }
                [(PREFIX) #2 " " #1 " " #3]
                [(POSTFIX) #1 " " #3 " " #2]
            |   digit
            ;
```

The Yacc actions are contained in the braces and the NewYacc specifications are in the square brackets.

A parse tree for the input sentence "$(3 + (2 - 1)) - (4 + 5)$" is shown in Figure 7. If the task to be performed is the translation of infix expressions to postfix expressions, the above specification could be used and a POSTFIX traversal specified. In this case the nodes of the parse tree would be visited in the order indicated by the numbering in the figure and the output would be "3 2 1 − +4 5 + −".

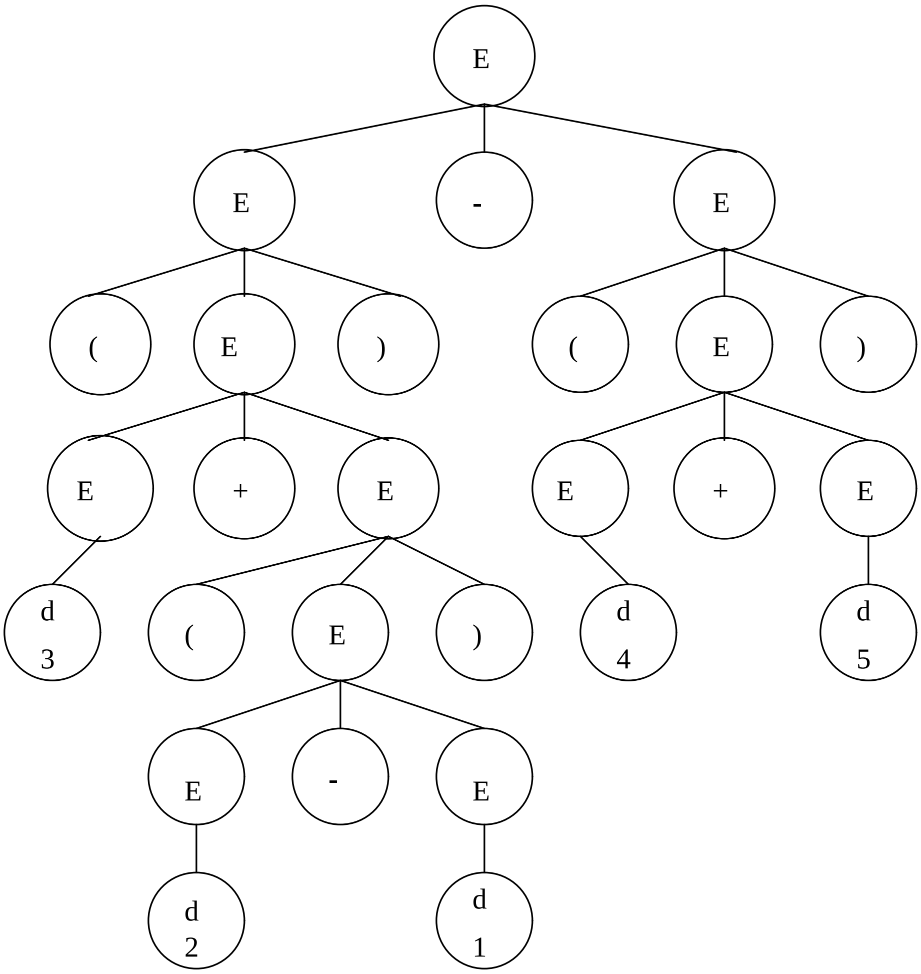

Figure 7: Parse Tree for $(3 + (2 - 1)) - (4 + 5)$.

Initiating a traversal with the display mask PREFIX would result in the output of "$- + 3 - 2\ 1 +$ $4\ 5$". A traversal can also be initiated at the root with a null display mask which will result in a parse tree traversal that visits all nodes and outputs the input sentence unchanged.

4.1.2 SELECTIVE TRAVERSALS

The second type of traversal is a selective traversal. This is indicated by a grammar symbol in a rule translation prefixed with the character "&" or by using the `NY_SELECT` option in the invocation of *nyprint*(). Lacking a matching display mask, the default translation for a selective traversal is the traversal of the subtrees for the non-terminals on the right-hand side of the production. Using the example production of the previous subsection, the default translation during a selective traversal would be:

[#1 #3]

Since leaf nodes are not output during a selective traversal, initiating the traversal with a null display mask will result in no output.

The expression grammar previously used can be modified slightly to extract information from the input without echoing the input itself. The following specification produces a translation which counts and returns the number of operands in the input expression. Two functions are used here. One function, *bump*, increments an integer counter which is declared in the main program and returns a null string. The second function, *outputcount*, looks at this integer counter, converts it into a string and returns this string.

Exp	:	Expression
		[(COUNTER) #1 outputcount()]
	;	
Expression	:	‘(’ Expression ‘)’
	|	Expression ‘+’ Expression
	|	Expression ‘−’ Expression
	|	digit
		[(COUNTER) bump()]
	;	

Traversed selectively, this specification will output the number of digits found in the input expression.

4.1.3 CHANGING MASKS AND TRAVERSAL MODES DURING TRAVERSALS

A few more words are needed in regard to symbol references within translations. A grammar symbol reference is of the form[1]

$$[\backslash]\#\mathrm{n}[([\mathit{mask}])]$$

or

$$[\backslash]\&\mathrm{n}[([\mathit{mask}])]$$

where *mask* is a label with which the traversal searches the n^{th} grammar symbol (usually a non-terminal representing a subtree). A mask may be optional. When no mask is used, the mask of the current translation applies for the traversal searching the subtree. The mask of the current translation is called the *descending* mask because the current traversal mask is specified in an ancestor node or by the *nyprint* function if the current node is the root of the parse tree. If MSK is the descending traversal mask and the translation

$$[\ (\mathrm{MSK})\ \#1\ \#2\ \#3\ ]$$

[1] Square brackets denotes optional.

	descending type and mask	
symbol	open M	selective M
#n	open n(M)	selective n(M)
#n()	open n(NULL)	open n(NULL)
#n(X)	open n(X)	open n(X)
&n	selective n(M)	selective n(M)
&n()	selective n(NULL)	selective n(NULL)
&n(X)	selective n(X)	selective n(X)

Table 1: Symbol references and their contextual meanings

is matched, then this translation is equivalent to the translation

[(MSK) #1(MSK) #2(MSK) #3(MSK)]

if the descending traversal type is open and

[(MSK) &1(MSK) &2(MSK) &3(MSK)]

if the descending traversal type is selective. Twelve types of symbol references can be constructed from these forms. Table 1 displays six symbol forms and their meanings in the context of descending traversals. The other six forms are constructed from these by prefixing a "\" character to the symbol to enable string filtering on a subtree traversal. String filtering is briefly discussed in Section 3.2.2.

The NULL traversal mask is predefined and matches no translation. If NULL is the current display mask and a traversal is open, the input on the subtree is simply echoed. If the traversal is selective, the output for the subtree will be empty (e.g., see the "&n()" case in Table 1). Normally, the NULL mask is used only on terminal symbols during a selective traversal to cause a leaf node to be output. If the "()" is omitted and the descending traversal is selective, the terminals will not be echoed.

By combining traversal masks and types within translations, traversal may also be mixed. Almost all applications of NewYacc will require the use of mixed traversals. A selective traversal, for instance, may need to echo whole subtrees, or an open traversal might echo the entire input except for a few tokens. These mechanisms allow developers to view their annotations as operations on parse trees, rather than abstract attributes.

For example, a small modification can be made to the grammar to produce a translation that changes the current display mask during a traversal so that an expression can be printed out in a strange combination of prefix and postfix notation.

```
Expression    :    '(' Expression ')'
                   [(PREFIX) #1 #2(POSTFIX) #3]
                   [(POSTFIX) #1 #2(PREFIX) #3]
              |    Expression '+' Expression
                   [(PREFIX) "+" #1 " " #3]
                   [(POSTFIX) #1 " " #3 "+"]
              |    Expression '–' Expression
                   [(PREFIX) "–" #1 " " #3]
                   [(POSTFIX) #1 " " #3 "–"]
              |    digit
              ;
```

Assuming the traversal is initiated to be an open traversal initially, the occurrence of parentheses causes the current traversal type to change. For example, given the input "(3 + (2 - 1)) - (4 +5)" and an initial display mask of POSTFIX and a traversal type of open, the string "(+ 3 (2 1 -))(+ 4 5) -" would be output. Looking again at Figure 7, the display mask changes for the second subtree of every node which has parentheses as a subtree.

A specification which switches between open and selective modes can be used to output all addition operands in the input expression.

```
Expression    :    '(' Expression ')'
              |    Expression '+' Expression
                   [(ADDERS) #1() #2() #3() "\n"]
              |    Expression '–' Expression
              |    digit
              ;
```

If this traversal is initiated to be selective with display mask ADDERS, the null mask, indicated by "()", causes an open traversal (and literal output) of some subtrees while the rest of the parse tree is still traversed selectively. For the input "(3 + (2 - 1)) - (4 + 5)", the output would be:

3 + (2 - 1)
4 + 5

since these are the two addition expressions of the input.

4.1.4 SEMANTICS OF TRAVERSAL AT A NODE

At a node in the parse tree for a grammar rule of the form:

$$A : E_1E_2...E_n;$$

where the current display mask is M and the traversal type is either *open* or *selective*.

```
Algorithm Traversal
    if there is no rule translation with a mask = M then
        if traversal type is open then
            traverse the subtrees E1...En from left to right
        else
            traverse the non-leaf subtrees of E1...En from left to right
        end if;
    else   /* there is a matching rule translation */
        For each element of the rule translation which matches do
            case element type of
                literal: send it to output
                function: make the call and send the result to output
                reference var decl: allocate space for the variable; initialize to the null string
                reference var: send the value of the variable to output
                assignment to reference var: evaluate the right hand side and store the result
                conditional statement: evaluate the expression and use the result to decide the direction of
                        further evaluation
                block opening (closing) : open (close) block
                subtree symbol:
                    if the symbol has an associated display mask M1 then
                        if symbol prefixed with a "#" then
                            traverse the subtree with open traversal using M1
                        else
                            traverse the subtree with selective traversal using M1
                        end if;
                    else
                        if symbol prefixed with a "#" then
                            traverse with the same traversal as before using M
                        else
                            traverse with selective traversal using M
                        end if;
                    end if;
            end case;
        end for;
    end if;
end Algorithm;
```

4.2 USER-DEFINED FUNCTIONS

A user may define functions for use during the parse tree traversal process. The type of these functions must be pointer to character (string) and all functions are defined in the declarations section of the NewYacc specification. User-defined functions may have zero or more actual parameters (arguments of the function invocation) of type string. The formal parameters (the arguments in the function declaration) are accessed via two parameters. The first parameter is an integer which tells how many arguments were used and the second contains the value of these passed arguments stored in an array of strings. For example, a function *foobar* could be declared for use with NewYacc as follows:

```
char *foobar(argc, argv)
    int argc;
    char **argv;
{ body of foobar }
```

The actual parameters of the function invocation are contained in strings *argv[0], argv[1], . . . , argv[argc-1].* The value returned by a function is incorporated into the translation, assigned to a variable, or used as input to another function.

For example, a function *makestr* which takes 1 or more string and returns a single string can be written for use in a rule translation.

```
char *makestr(argc, argv)
    int argc;
    char **argv;
{   char * s;
    int len=0, i;
    for (i=0; i < argc; i++)
        len += strlen(argv[i]);
    s = (char *) malloc(len + 1);
    strcpy(s,argv[0]);
    for (i = 1; i < argc; i++)
        strcat(s,argv[i]);
    return (s);
}
```

This function takes a variable number of string arguments and returns a single string argument. If declared in the NewYacc specification, it can be used in a rule translation for a variety of purposes.

In Figure 8 *gensym*() is an example of the invocation of a user-defined function whose return value is assigned to a variable. The use of *bcharno*() in Figure 5 shows a user-defined function whose value is incorporated into the output of the translation.

4.3 BUILT-IN FUNCTIONS

There are several built-in functions which are useful when transforming and extracting information from a source file:

Function for tabbing:

char * tabdecr()	-	decrement the current tab level
char * tabincr()	-	increment the current tab level
char * tab()	-	place tab characters corresponding to the current tab level into the output

Functions that generate input line numbers:

char * blineno()	-	the beginning line number of the input matching the current rule
char * elineno()	-	the ending line number of the input matching the current rule

Functions that generate input character numbers:

char * bcharno()	-	the beginning character number within the text of the input matching the current rule
char * echarno()	-	the ending character number within the text of the input matching the current rule
char * bcwline()	-	the beginning character number within the line of the input matching the current rule
char * ecwline()	-	the ending character number within line of the input matching the current rule

Examples of the use of these functions can be found among the example specifications which come with the NewYacc tool itself.

4.4 REFERENCE VARIABLES

NewYacc allows the definition and use of reference variables within rule translations. These dynamically scoped variables are visible to nodes which are descendants of the node declaring the variables. Use of reference variables allows easy access to context sensitive information and can replace the use of an auxiliary stack to hold necessary information.

The symbol "!" is used to declare an instance of a reference variable and "@" is used to indicate the access of the variable. Variables must be declared at the beginning of the rule translation following the display mask. The following example shows the declaration and simple use of two reference variables.

```
A    :    B c D e
[(REFVAR) !a !b @a="stringtype" #1 @b=#3() ]
```

The value of variable "a" is assigned to the value of a literal string. As the first subtree is traversed, any reference to "a" (assuming another reference variable "a" is not declared) will return the value "stringtype". A reference to "b" (with the same assumptions) will return a null string, since "b" is only given a value after the traversal. When "b" is given a value, it is assigned to the string value that the open traversal with no display mask of the third subtree will return. This string will consist of the literal string value of all of the leaf nodes of this subtree appended together.

The next simple example uses the list grammar seen previously and creates a tool which rotates the first element of a list to the last spot in the list.

```
tlist     :   list
(*1)          [(ROTATE) !first #1 " " @first ]
          ;
list      :   list ITEM
              [(ROTATE) #1 " " #2 ]
          |   ITEM
(*2)          [(ROTATE) @first=#1() ]
          ;
```

A single reference variable "first" is declared at the start of the traversal (at location *1) and is used to hold the value of the first item in the list when found (at location *2). When the entire list is traversed, this first item is output (at *1). If the input list is "a b c", the output will be "b c a".

A more complex example of the use of reference variables can be seen in Figure 8. The purpose of this specification is to substitute simple variable names for digits in the input string. For example, the input string "$2+(7*3)+2$" would result in the output "$a = 2 \;\; b = 7 \;\; a+(b*3)+a$" during an open traversal with the display mask SUBSTITUTE. Two reference variables are used, one to hold the digit being substituted and the second to hold the simple variable name which is to be substituted. The first rule translation simply declares a new occurrence of the variables "var" and "val" whenever a node "stat" occurs in the parse tree. The second rule translation in the specification takes advantage of the fact that reference variables are given a null string value when they are declared to decide whether the digit has been assigned a variable name yet.

4.5 BLOCKS

Another useful feature of NewYacc is the ability to create blocks for scoping purposes. The blocks are delimited by the characters "{" and "}". Any variables declared within a block are not visible outside the block. For example, if the following rule translation was encountered during a traversal:

```
[(BLOCK) !a !b @a="outstring" @b="instring" {!a @a=@b @a " "} @a ]
```

the output would be the string "instring outstring" since the value of "a" inside the block is assigned to the value of b and then output followed by a blank. Finally, the value of "a" outside the block is output.

Each rule translation can also be thought of as opening a new block. Blocks are commonly used with conditional statements which are discussed in the next sections and examples of block usage may be found there.

4.6 CONDITIONALS

NewYacc provides conditional tests in a rule translation based on string comparisons between translation items. Examples of translations items which can be compared are literal strings, reference variables, strings returned from functions, and strings returned from subtree traversals. The syntax of conditional statements is:

```
if_stmt :     'IF' '(' expression1 condition expression2 ')' 'THEN'
                   '{' statements '}'
              'ELSE'
                   '{' statements '}'
```

where "expression1" and "expression2" are string types and the condition is one of { ==, ! =, <, >, >=, <= }.

Going back to the list grammar, this next example takes a list and removes the elements which are of value a.

```
tlist         :    list
              ;
list          :    list ITEM
                   [(NOA) IF (#2() !="a") THEN { #1 " " #2}
                   ELSE {#1 } ]
              |    ITEM
                   [(NOA) IF (#1() != "a") THEN {#1 } ELSE {} ]
              ;
```

Here each leaf value is checked and output if it is not equal to a.

A more interesting task using the same grammar is one which merges adjacent identical elements into a single element.

```
list    ::=   list stat ;
stat    ::=   expr
              [(SUBSTITUTE) !var !val #1]
        |     LETTER '=' expr ;
expr    ::=   '(' expr ')'
        |     expr '+' expr
        |     expr '-' expr
              . . .
        |     term
              [(SUBSTITUTE) IF (@var == "") THEN
              { @var = gensym() @val = #1 @var "=" #1 }
              ELSE { IF (@val == #1) THEN { @var }
                    ELSE {#1 }]
```

Figure 8: Use of Conditionals, References, and User-Defined Functions.

```
tlist        :   list
                 [(MERGE) !item #1 ]
             ;
list         :   list ITEM
                 [(MERGE) IF (#2() != @item) THEN {@item=#2() #1 " " #2}
                 ELSE {#1 } ]
             |   ITEM
                 [(MERGE) IF (#1() != @item) THEN {#1 } ELSE {} ]
             ;
```

The idea in this example is to always keep the most recently seen ITEM in a reference variable "item". For each new element of the list seen, compare it to the most recently seen ITEM. If they match, do not output the new elements. If they do not match, save the value of the new element in "item" and continue traversing (remembering to output this elements when the traversal is done).

Figure 8 shows a more complex example use of conditional tests and reference variables. The nested conditional decides whether to initialize the auxiliary variable, to output the contents of a reference variable, or to traverse a subtree based on the string values of the two reference variables in relation to the contents of the current node.

References

[1] Aho, A. and J. Ullman. **Principles of Compiler Design**, Addison-Wesley, (1979).

[2] Aho, A. and J. Ullman. **The Theory of Parsing, Translation, and Compiling, vol 2**. Prentice Hall, (1973), pp. 758-782.

[3] Johnson, S. Yacc: Yet Another Compiler-Compiler. Bell Laboratories, (1979).

[4] Lesk, M. E. and E. Schmidt. Lex: A Lexical Analyzer Generator. Bell Laboratories, (1979).

[5] Purtilo, J. and J. Callahan. Parse Tree Annotations, *Communications of the ACM*, vol. 32, (December 1989), pp. 1467- 1477.

[6] Reps, T. and T. Teitelbaum. The synthesizer generator. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (April 1984), pp. 42-48.

ACKNOWLEDGEMENT: Thanks to Spencer Rugaber for his assistance in checking out the NewYacc distributions, and also for his helpful comments concerning this document.

A OPERATION

Tool development using the NewYacc system is a several step process. These steps are enumerated below along with the actual commands for producing the tool.

1. A grammar for the input language and a lexical analyzer for the tokens of that language must be produced. The lexical analyzer can be produced by the user or generated using Lex [4] and is invoked in the same way as in Yacc. The input grammar must be LALR(1) with disambiguating rules as required for Yacc. In some systems if Lex is used to produce the lexical analyzer, the produced *lex.yy.c* file must be further edited using a script *nylexfix.lex*.

   lex <filename>.l
   nylexfix.lex lex.yy.c

2. The grammar is augmented Yacc and NewYacc specifications as needed for the application. Be sure to either "#include" *lex.yy.c* in the program section or link the lex.yy.c code with the parser, so that the lexical analyzer will be part of the package.

3. This augmented grammar is input to the NewYacc tool which produces a parser in a file *y.tab.c* (on most system). On some systems, this file must be edited using the script *nytabcfix.yacc*.

   newyacc -f sampl.ny -v -d -Y
   nytabcfix.yacc y.tab.c

4. A main program written in C must be produced. This main program must contain a call to *yyparse()* to parse the input sentence and output the parse tree followed by a call to *nyprint()* (with the appropriate parameters) to perform a traversal. The *nyprint()* function may be called multiple times. This file must also "#include" a file called *nytags.h* which is produced by the NewYacc tool.

5. Compile and link together *y.tab.c*, the main program, and any other C code being used in the application.

   cc -c main.c
   cc -c y.tab.c
   cc -o <outfilename> main.o y.tab.o <newyacclib> -ll

B SPECIFICATION FILES FOR SAMPL TOOLS

B.1 LEXICAL ANALYZER SPECIFICATION FILE: sampl.l

```
%%
begin                       { return(BEGINR); }
end                         { return(ENDR); }
if                          { return(IFR); }
then                        { return(THENR); }
while                       { return(WHILER); }
do                          { return(DOR); }
\:\=                        { return(ASSIGNOP); }
\=                          { return(EQUALS); }
\<                          { return(LESSTHAN); }
\>                          { return(GRTRTHAN); }
not                         { return(NOTR); }
\+                          { return(PLUS); }
\-                          { return(DASH); }
\*                          { return(STAR); }
\/                          { return(SLASH); }
\(                          { return(LPAREN); }
\)                          { return(RPAREN); }
[0-9]+                      { return(LITERAL); }
[A-Za-z_][A-Za-z0-9_]*      { return(IDENTIFIER); }
[ \t\r\n]                   ;
```

B.2 NEWYACC SPECIFICATION FILE: sampl.ny

```
%{ char *b,*c,*e,*bcharno(), *tab(), *tabincr(), *tabdecr();
%}
%tag            BLOCKS BIN POS PRETTYP
%token          BEGINR ENDR IFR THENR WHILER DOR IDENTIFIER ASSIGNOP
%token          EQUALS LESSTHAN GRTRTHAN NOTR PLUS DASH STAR SLASH
%token          LPAREN RPAREN LITERAL
%left           EQUALS LESSTHAN GRTRTHAN PLUS DASH STAR SLASH
%start          program
%%
program         :       block
                        [ (BLOCKS) !b !c !e @b="B0" "(setq fg (quote " @c="()" #1 ")) \n"]
                        [ (PRETTYP) #1 "\n" ] ;
block           :       BEGINR statement_list ENDR
                        [ (BLOCKS) "(" @b " (" @c "(" @e ") " #2 "))"]
                        [ (PRETTYP) #1() "\n" tabincr() #2 tabdecr() tab() #3() ] ;
statement_list  :
                |       statement statement_list
                        [ (PRETTYP) tab() #1 "\n" #2];
statement       :       IFR exp THENR block
                        [ (BLOCKS) !b !c !e @b="B"+&2(POS) @c="("+&2(BIN)+")" #4 ]
                        [ (PRETTYP) #1() " " #2 " " #3() " " #4 ]
                |       WHILER exp DOR block
                        [ (BLOCKS) !b !c !e @b="B"+&2(POS) @e="B"+&2(POS)
                              @c="("+&2(BIN)+")" #4 ]
                        [ (PRETTYP) #1() " " #2 " " #3() " " #4 ]
                |       IDENTIFIER ASSIGNOP exp
                        [ (BLOCKS) "(B" bcharno() " (" #1() " (" &3(BIN) ")))" ]
                        [ (PRETTYP) #1() " " #2() " " #3 ‘] ;
exp             :       expression
                        [ (POS) bcharno() ] ;
```

```
expression      :       expression EQUALS expression
                |       expression LESSTHAN expression
                |       expression GRTRTHAN expression
                |       expression PLUS expression
                |       expression DASH expression
                |       expression STAR expression
                |       expression SLASH expression
                |       LPAREN expression RPAREN
                |       IDENTIFIER
                        [ (BIN) #1() " " ]
                |       LITERAL
                |       NOTR LPAREN expression RPAREN ;
%%
#include "lex.yy.c"
```

B.3 MAIN PROGRAM : main.c

```
#include <stdio.h>
#include "nytags.h"

myprint(s)
char *s;
{
  if (s ) fputs(s,stdout);
}

main()
{
  if (yyparse() == 1) exit(-1);
  nyprint(myprint,PRETTYP,NY_OPEN,0);   /* pretty-printer */
  nyprint(myprint,BLOCKS,NY_SELECT,0);  /* flow-graph extracter */
}
```

B.4 makefile

```
all:            sampl

sampl:          y.tab.o main.o
                cc main.o y.tab.o -o sampl $(NEWYACCLIB) -ll

y.tab.o:        y.tab.c
                cc -c y.tab.c

main.o:         main.c
                cc -c main.c

y.tab.c:        sampl.ny lex.yy.c
                $(NEWYACC) -f sampl.y
                nytabcfix.yacc y.tab.c

main.c:         nytags.h

lex.yy.c:       sampl.l
                lex sampl.l
                nylexfix.lex lex.yy.c
```