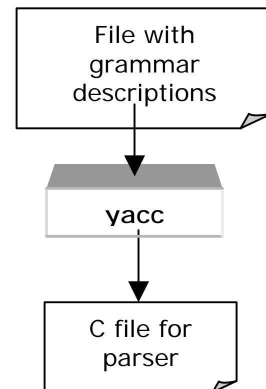
49 Writing YACC

"If you have lots of good advice, you will win."

YACC(Yet Another Compiler-Compiler) is a compiler writing tool. In this chapter, let's see how to write such a compiler writing tool.

49.1 Prelude

YACC was once available to Unix users only. Now we have DOS versions too. When we discussed about writing compilers, we have seen the uses of YACC. YACC gets the grammar for a given (new) language and generates a C file that can be compiled to work as a compiler for that new language. More specifically YACC don't directly generate compiler but generates parser.



YACC uses certain syntax or grammer to represent the grammar for new language. So one must be aware of the syntax used by YACC for its grammar file. As it has to output the compiler file, writing YACC is similar to writing a compiler.

49.2 BYACC

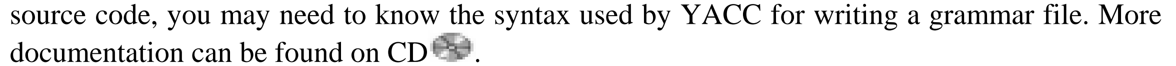
From the above discussion, it is clear that writing a YACC is really a tough job than writing a compiler! BYACC for DOS (**Berkeley YACC for MS-DOS**) is one of the good implementations.

49.2.1 Brief History

The original YACC was developed by AT&T. YACC interested many other people in the mean time. Later Berkeley University developed a open YACC and provided the source code to all. So the Berkeley's YACC was appreciated by all the people who are interested in writing compiler. Both AT&T and Berkeley's YACC was written for Unix environment. At that time, DOS doesn't have such utility. **Stephen C. Trier** used the source code provided by Berkeley and modified it for DOS and DOS version of YACC came into existence.

49.2.2 Source code

Source code of BYACC is more useful to understand the techniques and tactics used by real programmers. Many thanks to **Jeff Jenness** & **Stephen C. Trier** for providing such a good YACC. Following are the set of files used for BYACC. In order to understand the following

source code, you may need to know the syntax used by YACC for writing a grammar file. More documentation can be found on CD .

When you look at the source code, you may find that the function prototype declarations are in obsolete form. So you may get obsolete prototype declaration warning. That is because, the source code provided by Berkeley is quite older.

49.2.2.1 Def.h

```
#include <assert.h>
#include <ctype.h>
#include <stdio.h>
#ifdef MSDOS
#include <alloc.h>
#endif

/*  machine dependent definitions                          */
/*  the following definitions are for the VAX             */
/*  they might have to be changed for other machines      */

/*  MAXCHAR is the largest unsigned character value       */
/*  MAXSHORT is the largest value of a C short            */
/*  MINSHORT is the most negative value of a C short      */
/*  MAXTABLE is the maximum table size                    */
/*  BITS_PER_WORD is the number of bits in a C unsigned */
/*  WORDSIZE computes the number of words needed to       */
/*      store n bits                                       */
/*  BIT returns the value of the n-th bit starting        */
/*      from r (0-indexed)                                 */
/*  SETBIT sets the n-th bit starting from r              */

#define MAXCHAR         255
#define MAXSHORT        32767
#define MINSHORT        -32768
#define MAXTABLE        32500
#define BITS_PER_WORD   16
#define WORDSIZE(n)     (((n)+(BITS_PER_WORD-1))/BITS_PER_WORD)
#ifdef MSDOS
#define BIT(r, n)       ((((r)[(n) >> 4]) >> ((n) & 15)) & 1)
#define SETBIT(r, n)    ((r)[(n) >> 4] |= (1 << ((n) & 15)))
#else
#define BIT(r, n)       ((((r)[(n)>>5])>>((n)&31))&1)
#define SETBIT(r, n)    ((r)[(n)>>5]|=((unsigned)1<<((n)&31)))
#endif

/*  character names  */

#define NUL             '\0'    /*  the null character  */
```

```
#define NEWLINE          '\n'    /*  line feed  */
#define SP               ' '     /*  space  */
#define BS               '\b'    /*  backspace  */
#define HT               '\t'    /*  horizontal tab  */
#define VT               '\013'  /*  vertical tab  */
#define CR               '\r'    /*  carriage return  */
#define FF               '\f'    /*  form feed  */
#define QUOTE            '\''    /*  single quote  */
#define DOUBLE_QUOTE     '\"'    /*  double quote  */
#define BACKSLASH        '\\'    /*  backslash  */

/* defines for constructing filenames */

#ifdef MSDOS
#define CODE_SUFFIX      "_code.c"
#define DEFINES_SUFFIX   "_tab.h"
#define OUTPUT_SUFFIX    "_tab.c"
#define VERBOSE_SUFFIX   ".out"
#else
#define CODE_SUFFIX      ".code.c"
#define DEFINES_SUFFIX   ".tab.h"
#define OUTPUT_SUFFIX    ".tab.c"
#define VERBOSE_SUFFIX   ".output"
#endif

/* keyword codes */

#define TOKEN 0
#define LEFT 1
#define RIGHT 2
#define NONASSOC 3
#define MARK 4
#define TEXT 5
#define TYPE 6
#define START 7
#define UNION 8
#define IDENT 9

/*  symbol classes  */

#define UNKNOWN 0
#define TERM 1
#define NONTERM 2

/*  the undefined value  */
#define UNDEFINED (-1)
```

```
/*  action codes  */
#define SHIFT 1
#define REDUCE 2
#define ERROR 3

/*  character macros  */
#define IS_IDENT(c) (isalnum(c)||(c) == '_' || (c) == '.' || (c) == '$')
#define IS_OCTAL(c)     ((c) >= '0' && (c) <= '7')
#define NUMERIC_VALUE(c)        ((c) - '0')
/*  symbol macros  */
#define ISTOKEN(s)      ((s) < start_symbol)
#define ISVAR(s)        ((s) >= start_symbol)

/*  storage allocation macros  */
#define CALLOC(k,n)     (calloc((unsigned)(k),(unsigned)(n)))
#define FREE(x)         (free((char*)(x)))
#define MALLOC(n)       (malloc((unsigned)(n)))
#define NEW(t)          ((t*)allocate(sizeof(t)))
#define NEW2(n,t)       ((t*)allocate((unsigned)((n)*sizeof(t))))
#define REALLOC(p,n)    (realloc((char*)(p),(unsigned)(n)))

/*  the structure of a symbol table entry  */
typedef struct bucket bucket;
struct bucket
{
    struct bucket *link;
    struct bucket *next;
    char *name;
    char *tag;
    short value;
    short index;
    short prec;
    char class;
    char assoc;
};

/*  the structure of the LR(0) state machine  */
typedef struct core core;
struct core
{
    struct core *next;
    struct core *link;
    short number;
    short accessing_symbol;
    short nitems;
    short items[1];
};
```

```
/*  the structure used to record shifts  */

typedef struct shifts shifts;
struct shifts
{
    struct shifts *next;
    short number;
    short nshifts;
    short shift[1];
};

/*  the structure used to store reductions  */

typedef struct reductions reductions;
struct reductions
{
    struct reductions *next;
    short number;
    short nreds;
    short rules[1];
};

/*  the structure used to represent parser actions  */

typedef struct action action;
struct action
{
    struct action *next;
    short symbol;
    short number;
    short prec;
    char action_code;
    char assoc;
    char suppressed;
};

/* global variables */

extern char dflag;
extern char lflag;
extern char rflag;
extern char tflag;
extern char vflag;

extern char *myname;
extern char *cptr;
extern char *line;
```

```
extern int lineno;
extern int outline;

extern char *banner[];
extern char *tables[];
extern char *header[];
extern char *body[];
extern char *trailer[];

extern char *action_file_name;
extern char *code_file_name;
extern char *defines_file_name;
extern char *input_file_name;
extern char *output_file_name;
extern char *text_file_name;
extern char *union_file_name;
extern char *verbose_file_name;

extern FILE *action_file;
extern FILE *code_file;
extern FILE *defines_file;
extern FILE *input_file;
extern FILE *output_file;
extern FILE *text_file;
extern FILE *union_file;
extern FILE *verbose_file;

extern int nitems;
extern int nrules;
extern int nsyms;
extern int ntokens;
extern int nvars;
extern int ntags;

extern char unionized;
extern char line_format[];

extern int   start_symbol;
extern char  **symbol_name;
extern short *symbol_value;
extern short *symbol_prec;
extern char  *symbol_assoc;

extern short *ritem;
extern short *rlhs;
extern short *rrhs;
extern short *rprec;
```

```
extern char  *rassoc;

extern short **derives;
extern char *nullable;

extern bucket *first_symbol;
extern bucket *last_symbol;

extern int nstates;
extern core *first_state;
extern shifts *first_shift;
extern reductions *first_reduction;
extern short *accessing_symbol;
extern core **state_table;
extern shifts **shift_table;
extern reductions **reduction_table;
extern unsigned *LA;
extern short *LAruleno;
extern short *lookaheads;
extern short *goto_map;
extern short *from_state;
extern short *to_state;

extern action **parser;
extern int SRtotal;
extern int RRtotal;
extern short *SRconflicts;
extern short *RRconflicts;
extern short *defred;
extern short *rules_used;
extern short nunused;
extern short final_state;

/* global functions */

extern char *allocate();
extern bucket *lookup();
extern bucket *make_bucket();

/* system variables */

extern int errno;

/* system functions */

#ifndef MSDOS
extern void free();
```

```
extern char *calloc();
extern char *malloc();
extern char *realloc();
extern char *strcpy();
#endif
```

49.2.2.2 Closure.c

```
#include "defs.h"

short *itemset;
short *itemsetend;
unsigned *ruleset;

static unsigned *first_derives;
static unsigned *EFF;

set_EFF()
{
    register unsigned *row;
    register int symbol;
    register short *sp;
    register int rowsize;
    register int i;
    register int rule;

    rowsize = WORDSIZE(nvars);
    EFF = NEW2(nvars * rowsize, unsigned);

    row = EFF;
    for (i = start_symbol; i < nsyms; i++)
    {
      sp = derives[i];
      for (rule = *sp; rule > 0; rule = *++sp)
      {
          symbol = ritem[rrhs[rule]];
          if (ISVAR(symbol))
          {
            symbol -= start_symbol;
            SETBIT(row, symbol);
          }
      }
      row += rowsize;
    }

    reflexive_transitive_closure(EFF, nvars);
```

```
#ifdef      DEBUG
    print_EFF();
#endif
}

set_first_derives()
{
  register unsigned *rrow;
  register unsigned *vrow;
  register int j;
  register unsigned mask;
  register unsigned cword;
  register short *rp;

  int rule;
  int i;
  int rulesetsize;
  int varsetsize;

  rulesetsize = WORDSIZE(nrules);
  varsetsize = WORDSIZE(nvars);
  first_derives = NEW2(nvars * rulesetsize, unsigned) - ntokens *
rulesetsize;

  set_EFF();

  rrow = first_derives + ntokens * rulesetsize;
  for (i = start_symbol; i < nsyms; i++)
    {
      vrow = EFF + ((i - ntokens) * varsetsize);
      cword = *vrow++;
      mask = 1;
      for (j = start_symbol; j < nsyms; j++)
      {
        if (cword & mask)
          {
            rp = derives[j];
            while ((rule = *rp++) >= 0)
            {
              SETBIT(rrow, rule);
            }
          }

        mask <<= 1;
```

```
            if (mask == 0)
              {
                cword = *vrow++;
                mask = 1;
              }
          }
          vrow += varsetsize;
          rrow += rulesetsize;
        }

#ifdef      DEBUG
  print_first_derives();
#endif

  FREE(EFF);
}

closure(nucleus, n)
short *nucleus;
int n;
{
    register int ruleno;
    register unsigned word;
    register unsigned mask;
    register short *csp;
    register unsigned *dsp;
    register unsigned *rsp;
    register int rulesetsize;

    short *csend;
    unsigned *rsend;
    int symbol;
    int itemno;

    rulesetsize = WORDSIZE(nrules);
    rsp = ruleset;
    rsend = ruleset + rulesetsize;
    for (rsp = ruleset; rsp < rsend; rsp++)
      *rsp = 0;

    csend = nucleus + n;
    for (csp = nucleus; csp < csend; ++csp)
    {
      symbol = ritem[*csp];
      if (ISVAR(symbol))
      {
          dsp = first_derives + symbol * rulesetsize;
```

```
            rsp = ruleset;
            while (rsp < rsend)
              *rsp++ |= *dsp++;
        }
      }

      ruleno = 0;
      itemsetend = itemset;
      csp = nucleus;
      for (rsp = ruleset; rsp < rsend; ++rsp)
      {
        word = *rsp;
        if (word == 0)
            ruleno += BITS_PER_WORD;
        else
        {
            mask = 1;
            while (mask)
            {
              if (word & mask)
              {
                  itemno = rrhs[ruleno];
                  while (csp < csend && *csp < itemno)
                    *itemsetend++ = *csp++;
                  *itemsetend++ = itemno;
                  while (csp < csend && *csp == itemno)
                    ++csp;
              }

                  mask <<= 1;
                  ++ruleno;
            }
        }
      }

      while (csp < csend)
        *itemsetend++ = *csp++;

  #ifdef      DEBUG
    print_closure(n);
  #endif
  }

  finalize_closure()
  {
    FREE(itemset);
    FREE(ruleset);
```

```
  FREE(first_derives + ntokens * WORDSIZE(nrules));
}

#ifdef      DEBUG

print_closure(n)
int n;
{
  register short *isp;

  printf("\n\nn = %d\n\n", n);
  for (isp = itemset; isp < itemsetend; isp++)
    printf("   %d\n", *isp);
}

print_EFF()
{
    register int i, j, k;
    register unsigned *rowp;
    register unsigned word;
    register unsigned mask;

    printf("\n\nEpsilon Free Firsts\n");

    for (i = start_symbol; i < nsyms; i++)
    {
      printf("\n%s", symbol_name[i]);
      rowp = EFF + ((i - start_symbol) * WORDSIZE(nvars));
      word = *rowp++;

      mask = 1;
      for (j = 0; j < nvars; j++)
      {
          if (word & mask)
            printf("  %s", symbol_name[start_symbol + j]);

          mask <<= 1;
          if (mask == 0)
          {
            word = *rowp++;
            mask = 1;
          }
      }
    }
}
```

```
print_first_derives()
{
  register int i;
  register int j;
  register unsigned *rp;
  register unsigned cword;
  register unsigned mask;

  printf("\n\n\nFirst Derives\n");

  for (i = start_symbol; i < nsyms; i++)
    {
      printf("\n%s derives\n", symbol_name[i]);
      rp = first_derives + i * WORDSIZE(nrules);
      cword = *rp++;
      mask = 1;
      for (j = 0; j <= nrules; j++)
        {
        if (cword & mask)
          printf("   %d\n", j);

        mask <<= 1;
        if (mask == 0)
          {
            cword = *rp++;
            mask = 1;
          }
      }
    }

  fflush(stdout);
}

#endif
```

49.2.2.3 Error.c

```
/* routines for printing error messages  */

#include "defs.h"

fatal(msg)
char *msg;
{
    fprintf(stderr, "%s: f - %s\n", myname, msg);
    done(2);
}
```

```
no_space()
{
    fprintf(stderr, "%s: f - out of space\n", myname);
    done(2);
}

open_error(filename)
char *filename;
{
    fprintf(stderr, "%s: f - cannot open \"%s\"\n", myname, filename);
    done(2);
}

unexpected_EOF()
{
    fprintf(stderr, "%s: e - line %d of \"%s\", unexpected end-of-
file\n",
          myname, lineno, input_file_name);
    done(1);
}

print_pos(st_line, st_cptr)
char *st_line;
char *st_cptr;
{
    register char *s;

    if (st_line == 0) return;
    for (s = st_line; *s != '\n'; ++s)
    {
      if (isprint(*s) || *s == '\t')
          putc(*s, stderr);
      else
          putc('?', stderr);
    }
    putc('\n', stderr);
    for (s = st_line; s < st_cptr; ++s)
    {
      if (*s == '\t')
          putc('\t', stderr);
      else
          putc(' ', stderr);
    }
    putc('^', stderr);
    putc('\n', stderr);
}
```

```
syntax_error(st_lineno, st_line, st_cptr)
int st_lineno;
char *st_line;
char *st_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", syntax error\n",
          myname, st_lineno, input_file_name);
    print_pos(st_line, st_cptr);
    done(1);
}

unterminated_comment(c_lineno, c_line, c_cptr)
int c_lineno;
char *c_line;
char *c_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", unmatched /*\n",
          myname, c_lineno, input_file_name);
    print_pos(c_line, c_cptr);
    done(1);
}

unterminated_string(s_lineno, s_line, s_cptr)
int s_lineno;
char *s_line;
char *s_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", unterminated string\n",
          myname, s_lineno, input_file_name);
    print_pos(s_line, s_cptr);
    done(1);
}

unterminated_text(t_lineno, t_line, t_cptr)
int t_lineno;
char *t_line;
char *t_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", unmatched %%{\n",
          myname, t_lineno, input_file_name);
    print_pos(t_line, t_cptr);
    done(1);
}

unterminated_union(u_lineno, u_line, u_cptr)
int u_lineno;
char *u_line;
```

```
char *u_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", unterminated %%union \
declaration\n", myname, u_lineno, input_file_name);
    print_pos(u_line, u_cptr);
    done(1);
}

over_unionized(u_cptr)
char *u_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", too many %%union \
declarations\n", myname, lineno, input_file_name);
    print_pos(line, u_cptr);
    done(1);
}

illegal_tag(t_lineno, t_line, t_cptr)
int t_lineno;
char *t_line;
char *t_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", illegal tag\n",
          myname, t_lineno, input_file_name);
    print_pos(t_line, t_cptr);
    done(1);
}

illegal_character(c_cptr)
char *c_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", illegal character\n",
          myname, lineno, input_file_name);
    print_pos(line, c_cptr);
    done(1);
}

used_reserved(s)
char *s;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", illegal use of reserved
symbol \
%s\n", myname, lineno, input_file_name, s);
    done(1);
}

tokenized_start(s)
```

```
char *s;
{
     fprintf(stderr, "%s: e - line %d of \"%s\", the start symbol %s
cannot be \
declared to be a token\n", myname, lineno, input_file_name, s);
     done(1);
}

retyped_warning(s)
char *s;
{
    fprintf(stderr, "%s: w - line %d of \"%s\", the type of %s has been
\
redeclared\n", myname, lineno, input_file_name, s);
}

reprec_warning(s)
char *s;
{
    fprintf(stderr, "%s: w - line %d of \"%s\", the precedence of %s has
been \
redeclared\n", myname, lineno, input_file_name, s);
}

revalued_warning(s)
char *s;
{
    fprintf(stderr, "%s: w - line %d of \"%s\", the value of %s has been
\
redeclared\n", myname, lineno, input_file_name, s);
}

terminal_start(s)
char *s;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", the start symbol %s is a
\
token\n", myname, lineno, input_file_name, s);
    done(1);
}

restarted_warning()
{
    fprintf(stderr, "%s: w - line %d of \"%s\", the start symbol has
been \
redeclared\n", myname, lineno, input_file_name);
}
```

```
no_grammar()
{
    fprintf(stderr, "%s: e - line %d of \"%s\", no grammar has been \
specified\n", myname, lineno, input_file_name);
    done(1);
}

terminal_lhs(s_lineno)
int s_lineno;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", a token appears on the
lhs \
of a production\n", myname, s_lineno, input_file_name);
    done(1);
}

prec_redeclared()
{
    fprintf(stderr, "%s: w - line %d of  \"%s\", conflicting %%prec \
specifiers\n", myname, lineno, input_file_name);
}

unterminated_action(a_lineno, a_line, a_cptr)
int a_lineno;
char *a_line;
char *a_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", unterminated action\n",
          myname, a_lineno, input_file_name);
    print_pos(a_line, a_cptr);
    done(1);
}

dollar_warning(a_lineno, i)
int a_lineno;
int i;
{
    fprintf(stderr, "%s: w - line %d of \"%s\", $%d references beyond
the \
end of the current rule\n", myname, a_lineno, input_file_name, i);
}

dollar_error(a_lineno, a_line, a_cptr)
int a_lineno;
char *a_line;
```

```
char *a_cptr;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", illegal $-name\n",
          myname, a_lineno, input_file_name);
    print_pos(a_line, a_cptr);
    done(1);
}

untyped_lhs()
{
    fprintf(stderr, "%s: e - line %d of \"%s\", $$ is untyped\n",
          myname, lineno, input_file_name);
    done(1);
}

untyped_rhs(i, s)
int i;
char *s;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", $%d (%s) is untyped\n",
          myname, lineno, input_file_name, i, s);
    done(1);
}

unknown_rhs(i)
int i;
{
    fprintf(stderr, "%s: e - line %d of \"%s\", $%d is untyped\n",
          myname, lineno, input_file_name, i);
    done(1);
}

default_action_warning()
{
    fprintf(stderr, "%s: w - line %d of \"%s\", the default action
assigns an \
undefined value to $$\n", myname, lineno, input_file_name);
}

undefined_goal(s)
char *s;
{
    fprintf(stderr, "%s: e - the start symbol %s is undefined\n",
myname, s);
    done(1);
}
```

```
undefined_symbol_warning(s)
char *s;
{
    fprintf(stderr, "%s: w - the symbol %s is undefined\n", myname, s);
}
```

49.2.2.4 Lalr.c

```
#include "defs.h"

typedef
  struct shorts
    {
      struct shorts *next;
      short value;
    }
  shorts;

int tokensetsize;
short *lookaheads;
short *LAruleno;
unsigned *LA;
short *accessing_symbol;
core **state_table;
shifts **shift_table;
reductions **reduction_table;
short *goto_map;
short *from_state;
short *to_state;

short **transpose();

static int infinity;
static int maxrhs;
static int ngotos;
static unsigned *F;
static short **includes;
static shorts **lookback;
static short **R;
static short *INDEX;
static short *VERTICES;
static int top;

lalr()
{
    tokensetsize = WORDSIZE(ntokens);
```

```
    set_state_table();
    set_accessing_symbol();
    set_shift_table();
    set_reduction_table();
    set_maxrhs();
    initialize_LA();
    set_goto_map();
    initialize_F();
    build_relations();
    compute_FOLLOWS();
    compute_lookaheads();
}

set_state_table()
{
    register core *sp;

    state_table = NEW2(nstates, core *);
    for (sp = first_state; sp; sp = sp->next)
      state_table[sp->number] = sp;
}

set_accessing_symbol()
{
    register core *sp;

    accessing_symbol = NEW2(nstates, short);
    for (sp = first_state; sp; sp = sp->next)
      accessing_symbol[sp->number] = sp->accessing_symbol;
}

set_shift_table()
{
    register shifts *sp;

    shift_table = NEW2(nstates, shifts *);
    for (sp = first_shift; sp; sp = sp->next)
      shift_table[sp->number] = sp;
}

set_reduction_table()
{
    register reductions *rp;
    reduction_table = NEW2(nstates, reductions *);
    for (rp = first_reduction; rp; rp = rp->next)
      reduction_table[rp->number] = rp;
}
```

```
set_maxrhs()
{
  register short *itemp;
  register short *item_end;
  register int length;
  register int max;

  length = 0;
  max = 0;
  item_end = ritem + nitems;
  for (itemp = ritem; itemp < item_end; itemp++)
    {
      if (*itemp >= 0)
      {
        length++;
      }
      else
      {
        if (length > max) max = length;
        length = 0;
      }
    }

  maxrhs = max;
}

initialize_LA()
{
  register int i, j, k;
  register reductions *rp;

  lookaheads = NEW2(nstates + 1, short);

  k = 0;
  for (i = 0; i < nstates; i++)
    {
      lookaheads[i] = k;
      rp = reduction_table[i];
      if (rp)
      k += rp->nreds;
    }
  lookaheads[nstates] = k;

  LA = NEW2(k * tokensetsize, unsigned);
  LAruleno = NEW2(k, short);
  lookback = NEW2(k, shorts *);
```

```
    k = 0;
    for (i = 0; i < nstates; i++)
      {
        rp = reduction_table[i];
        if (rp)
        {
          for (j = 0; j < rp->nreds; j++)
            {
              LAruleno[k] = rp->rules[j];
              k++;
            }
        }
      }
  }

  set_goto_map()
  {
    register shifts *sp;
    register int i;
    register int symbol;
    register int k;
    register short *temp_map;
    register int state2;
    register int state1;

    goto_map = NEW2(nvars + 1, short) - ntokens;
    temp_map = NEW2(nvars + 1, short) - ntokens;

    ngotos = 0;
    for (sp = first_shift; sp; sp = sp->next)
      {
        for (i = sp->nshifts - 1; i >= 0; i--)
        {
          symbol = accessing_symbol[sp->shift[i]];

          if (ISTOKEN(symbol)) break;

          if (ngotos == MAXSHORT)
            fatal("too many gotos");

          ngotos++;
          goto_map[symbol]++;
          }
      }

    k = 0;
```

```
  for (i = ntokens; i < nsyms; i++)
    {
      temp_map[i] = k;
      k += goto_map[i];
    }

  for (i = ntokens; i < nsyms; i++)
    goto_map[i] = temp_map[i];

  goto_map[nsyms] = ngotos;
  temp_map[nsyms] = ngotos;

  from_state = NEW2(ngotos, short);
  to_state = NEW2(ngotos, short);

  for (sp = first_shift; sp; sp = sp->next)
    {
      state1 = sp->number;
      for (i = sp->nshifts - 1; i >= 0; i--)
      {
        state2 = sp->shift[i];
        symbol = accessing_symbol[state2];

        if (ISTOKEN(symbol)) break;

        k = temp_map[symbol]++;
        from_state[k] = state1;
        to_state[k] = state2;
      }
    }

  FREE(temp_map + ntokens);
}

/*  Map_goto maps a state/symbol pair into its numeric representation.
      */
int
map_goto(state, symbol)
int state;
int symbol;
{
    register int high;
    register int low;
    register int middle;
    register int s;

    low = goto_map[symbol];
```

```
    high = goto_map[symbol + 1];

    for (;;)
    {
      assert(low <= high);
      middle = (low + high) >> 1;
      s = from_state[middle];
      if (s == state)
          return (middle);
      else if (s < state)
          low = middle + 1;
      else
          high = middle - 1;
    }
}

initialize_F()
{
  register int i;
  register int j;
  register int k;
  register shifts *sp;
  register short *edge;
  register unsigned *rowp;
  register short *rp;
  register short **reads;
  register int nedges;
  register int stateno;
  register int symbol;
  register int nwords;

  nwords = ngotos * tokensetsize;
  F = NEW2(nwords, unsigned);

  reads = NEW2(ngotos, short *);
  edge = NEW2(ngotos + 1, short);
  nedges = 0;

  rowp = F;
  for (i = 0; i < ngotos; i++)
    {
      stateno = to_state[i];
      sp = shift_table[stateno];

      if (sp)
      {
        k = sp->nshifts;
```

```
        for (j = 0; j < k; j++)
          {
            symbol = accessing_symbol[sp->shift[j]];
            if (ISVAR(symbol))
            break;
            SETBIT(rowp, symbol);
          }

        for (; j < k; j++)
          {
            symbol = accessing_symbol[sp->shift[j]];
            if (nullable[symbol])
            edge[nedges++] = map_goto(stateno, symbol);
          }

        if (nedges)
          {
            reads[i] = rp = NEW2(nedges + 1, short);

            for (j = 0; j < nedges; j++)
            rp[j] = edge[j];

            rp[nedges] = -1;
            nedges = 0;
          }
      }

      rowp += tokensetsize;
    }

  SETBIT(F, 0);
  digraph(reads);

  for (i = 0; i < ngotos; i++)
    {
      if (reads[i])
      FREE(reads[i]);
    }

  FREE(reads);
  FREE(edge);
}

build_relations()
{
  register int i;
  register int j;
```

```
    register int k;
    register short *rulep;
    register short *rp;
    register shifts *sp;
    register int length;
    register int nedges;
    register int done;
    register int state1;
    register int stateno;
    register int symbol1;
    register int symbol2;
    register short *shortp;
    register short *edge;
    register short *states;
    register short **new_includes;

    includes = NEW2(ngotos, short *);
    edge = NEW2(ngotos + 1, short);
    states = NEW2(maxrhs + 1, short);

    for (i = 0; i < ngotos; i++)
      {
        nedges = 0;
        state1 = from_state[i];
        symbol1 = accessing_symbol[to_state[i]];

        for (rulep = derives[symbol1]; *rulep >= 0; rulep++)
        {
          length = 1;
          states[0] = state1;
          stateno = state1;

          for (rp = ritem + rrhs[*rulep]; *rp >= 0; rp++)
            {
              symbol2 = *rp;
              sp = shift_table[stateno];
              k = sp->nshifts;

              for (j = 0; j < k; j++)
              {
                stateno = sp->shift[j];
                if (accessing_symbol[stateno] == symbol2) break;
              }

              states[length++] = stateno;
            }
          add_lookback_edge(stateno, *rulep, i);
```

```
        length--;
        done = 0;
        while (!done)
          {
            done = 1;
            rp--;
            if (ISVAR(*rp))
            {
              stateno = states[--length];
              edge[nedges++] = map_goto(stateno, *rp);
              if (nullable[*rp] && length > 0) done = 0;
            }
          }
      }

      if (nedges)
      {
        includes[i] = shortp = NEW2(nedges + 1, short);
        for (j = 0; j < nedges; j++)
          shortp[j] = edge[j];
        shortp[nedges] = -1;
      }
    }

  new_includes = transpose(includes, ngotos);

  for (i = 0; i < ngotos; i++)
    if (includes[i])
      FREE(includes[i]);

  FREE(includes);

  includes = new_includes;

  FREE(edge);
  FREE(states);
}

add_lookback_edge(stateno, ruleno, gotono)
int stateno, ruleno, gotono;
{
    register int i, k;
    register int found;
    register shorts *sp;

    i = lookaheads[stateno];
    k = lookaheads[stateno + 1];
```

```
        found = 0;
        while (!found && i < k)
        {
          if (LAruleno[i] == ruleno)
              found = 1;
          else
              ++i;
        }
        assert(found);

        sp = NEW(shorts);
        sp->next = lookback[i];
        sp->value = gotono;
        lookback[i] = sp;
    }

    short **
    transpose(R, n)
    short **R;
    int n;
    {
      register short **new_R;
      register short **temp_R;
      register short *nedges;
      register short *sp;
      register int i;
      register int k;

      nedges = NEW2(n, short);

      for (i = 0; i < n; i++)
        {
          sp = R[i];
          if (sp)
          {
            while (*sp >= 0)
              nedges[*sp++]++;
          }
        }
      new_R = NEW2(n, short *);
      temp_R = NEW2(n, short *);

      for (i = 0; i < n; i++)
        {
          k = nedges[i];
          if (k > 0)
          {
```

```
            sp = NEW2(k + 1, short);
            new_R[i] = sp;
            temp_R[i] = sp;
            sp[k] = -1;
          }
        }

    FREE(nedges);

    for (i = 0; i < n; i++)
      {
        sp = R[i];
        if (sp)
        {
          while (*sp >= 0)
            *temp_R[*sp++]++ = i;
        }
      }
    FREE(temp_R);

    return (new_R);
  }

compute_FOLLOWS()
{
  digraph(includes);
}

compute_lookaheads()
{
  register int i, n;
  register unsigned *fp1, *fp2, *fp3;
  register shorts *sp, *next;
  register unsigned *rowp;

  rowp = LA;
  n = lookaheads[nstates];
  for (i = 0; i < n; i++)
    {
      fp3 = rowp + tokensetsize;
      for (sp = lookback[i]; sp; sp = sp->next)
      {
        fp1 = rowp;
        fp2 = F + tokensetsize * sp->value;
        while (fp1 < fp3)
          *fp1++ |= *fp2++;
      }
```

```
        rowp = fp3;
      }

  for (i = 0; i < n; i++)
    for (sp = lookback[i]; sp; sp = next)
      {
        next = sp->next;
        FREE(sp);
      }

  FREE(lookback);
  FREE(F);
}

digraph(relation)
short **relation;
{
  register int i;

  infinity = ngotos + 2;
  INDEX = NEW2(ngotos + 1, short);
  VERTICES = NEW2(ngotos + 1, short);
  top = 0;

  R = relation;

  for (i = 0; i < ngotos; i++)
    INDEX[i] = 0;

  for (i = 0; i < ngotos; i++)
    {
      if (INDEX[i] == 0 && R[i])
      traverse(i);
    }

  FREE(INDEX);
  FREE(VERTICES);
}

traverse(i)
register int i;
{
  register unsigned *fp1;
  register unsigned *fp2;
  register unsigned *fp3;
  register int j;
  register short *rp;
```

```
  int height;
  unsigned *base;

  VERTICES[++top] = i;
  INDEX[i] = height = top;

  base = F + i * tokensetsize;
  fp3 = base + tokensetsize;
  rp = R[i];
  if (rp)
    {
      while ((j = *rp++) >= 0)
      {
        if (INDEX[j] == 0)
          traverse(j);
        if (INDEX[i] > INDEX[j])
          INDEX[i] = INDEX[j];

        fp1 = base;
        fp2 = F + j * tokensetsize;

        while (fp1 < fp3)
          *fp1++ |= *fp2++;
      }
    }

  if (INDEX[i] == height)
    {
      for (;;)
      {
        j = VERTICES[top--];
        INDEX[j] = infinity;
        if (i == j)
          break;

        fp1 = base;
        fp2 = F + j * tokensetsize;

        while (fp1 < fp3)
          *fp2++ = *fp1++;
      }
    }
}
```

49.2.2.5 Lr0.c

```
#include "defs.h"
```

```
extern short *itemset;
extern short *itemsetend;
extern unsigned *ruleset;

int nstates;
core *first_state;
shifts *first_shift;
reductions *first_reduction;

int get_state();
core *new_state();

static core **state_set;
static core *this_state;
static core *last_state;
static shifts *last_shift;
static reductions *last_reduction;

static int nshifts;
static short *shift_symbol;

static short *redset;
static short *shiftset;
static short **kernel_base;
static short **kernel_end;
static short *kernel_items;

allocate_itemsets()
{
  register short *itemp;
  register short *item_end;
  register int symbol;
  register int i;
  register int count;
  register int max;
  register short *symbol_count;

  count = 0;
  symbol_count = NEW2(nsyms, short);

  item_end = ritem + nitems;
  for (itemp = ritem; itemp < item_end; itemp++)
    {
      symbol = *itemp;
      if (symbol >= 0)
      {
        count++;
```

```
        symbol_count[symbol]++;
      }
    }

  kernel_base = NEW2(nsyms, short *);
  kernel_items = NEW2(count, short);

  count = 0;
  max = 0;
  for (i = 0; i < nsyms; i++)
    {
      kernel_base[i] = kernel_items + count;
      count += symbol_count[i];
      if (max < symbol_count[i])
      max = symbol_count[i];
    }

  shift_symbol = symbol_count;
  kernel_end = NEW2(nsyms, short *);
}

allocate_storage()
{
  allocate_itemsets();

  shiftset = NEW2(nsyms, short);
  redset = NEW2(nrules + 1, short);
  state_set = NEW2(nitems, core *);
}

append_states()
{
  register int i;
  register int j;
  register int symbol;

#ifdef      TRACE
  fprintf(stderr, "Entering append_states\n");
#endif

for (i = 1; i < nshifts; i++)
    {
      symbol = shift_symbol[i];
      j = i;
      while (j > 0 && shift_symbol[j - 1] > symbol)
      {
        shift_symbol[j] = shift_symbol[j - 1];
```

```
          j--;
        }
        shift_symbol[j] = symbol;
      }

    for (i = 0; i < nshifts; i++)
      {
        symbol = shift_symbol[i];
        shiftset[i] = get_state(symbol);
      }
}

free_storage()
{
  FREE(shift_symbol);
  FREE(redset);
  FREE(shiftset);
  FREE(kernel_base);
  FREE(kernel_end);
  FREE(kernel_items);
  FREE(state_set);
}

generate_states()
{
  allocate_storage();
  itemset = NEW2(nitems, short);
  ruleset = NEW2(WORDSIZE(nrules), unsigned);
  set_first_derives();
  initialize_states();

  while (this_state)
    {
      closure(this_state->items, this_state->nitems);
      save_reductions();
      new_itemsets();
      append_states();

      if (nshifts > 0)
        save_shifts();

      this_state = this_state->next;
    }

  finalize_closure();
  free_storage();
}
```

```
int
get_state(symbol)
int symbol;
{
  register int key;
  register short *isp1;
  register short *isp2;
  register short *iend;
  register core *sp;
  register int found;
  int n;

#ifdef      TRACE
  fprintf(stderr, "Entering get_state, symbol = %d\n", symbol);
#endif

  isp1 = kernel_base[symbol];
  iend = kernel_end[symbol];
  n = iend - isp1;

  key = *isp1;
  assert(0 <= key && key < nitems);
  sp = state_set[key];
  if (sp)
    {
      found = 0;
      while (!found)
      {
        if (sp->nitems == n)
          {
            found = 1;
            isp1 = kernel_base[symbol];
            isp2 = sp->items;

            while (found && isp1 < iend)
            {
              if (*isp1++ != *isp2++)
                found = 0;
            }
          }

        if (!found)
          {
            if (sp->link)
            {
              sp = sp->link;
            }
```

```
            else
            {
              sp = sp->link = new_state(symbol);
              found = 1;
            }
          }
      }
    }
  else
    {
      state_set[key] = sp = new_state(symbol);
    }

  return (sp->number);
}

initialize_states()
{
    register int i;
    register short *start_derives;
    register core *p;
    start_derives = derives[start_symbol];
    for (i = 0; start_derives[i] >= 0; ++i)
      continue;

    p = (core *) MALLOC(sizeof(core) + i*sizeof(short));
    if (p == 0) no_space();

    p->next = 0;
    p->link = 0;
    p->number = 0;
    p->accessing_symbol = 0;
    p->nitems = i;

    for (i = 0;  start_derives[i] >= 0; ++i)
      p->items[i] = rrhs[start_derives[i]];

    first_state = last_state = this_state = p;
    nstates = 1;
}
new_itemsets()
{
  register int i;
  register int shiftcount;
  register short *isp;
  register short *ksp;
  register int symbol;
```

```
  for (i = 0; i < nsyms; i++)
    kernel_end[i] = 0;

  shiftcount = 0;
  isp = itemset;
  while (isp < itemsetend)
    {
      i = *isp++;
      symbol = ritem[i];
      if (symbol > 0)
      {
          ksp = kernel_end[symbol];

          if (!ksp)
          {
            shift_symbol[shiftcount++] = symbol;
            ksp = kernel_base[symbol];
          }

          *ksp++ = i + 1;
          kernel_end[symbol] = ksp;
      }
    }

  nshifts = shiftcount;
}
core *
new_state(symbol)
int symbol;
{
  register int n;
  register core *p;
  register short *isp1;
  register short *isp2;
  register short *iend;

#ifdef      TRACE
  fprintf(stderr, "Entering new_state, symbol = %d\n", symbol);
#endif
  if (nstates >= MAXSHORT)
    fatal("too many states");

  isp1 = kernel_base[symbol];
  iend = kernel_end[symbol];
  n = iend - isp1;
```

```
  p = (core *) allocate((unsigned) (sizeof(core) + (n - 1) *
sizeof(short)));
  p->accessing_symbol = symbol;
  p->number = nstates;
  p->nitems = n;

  isp2 = p->items;
  while (isp1 < iend)
    *isp2++ = *isp1++;

  last_state->next = p;
  last_state = p;

  nstates++;

  return (p);
}

/* show_cores is used for debugging */

show_cores()
{
    core *p;
    int i, j, k, n;
    int itemno;

    k = 0;
    for (p = first_state; p; ++k, p = p->next)
    {
      if (k) printf("\n");
      printf("state %d, number = %d, accessing symbol = %s\n",
            k, p->number, symbol_name[p->accessing_symbol]);
      n = p->nitems;
      for (i = 0; i < n; ++i)
      {
          itemno = p->items[i];
          printf("%4d  ", itemno);
          j = itemno;
          while (ritem[j] >= 0) ++j;
          printf("%s :", symbol_name[rlhs[-ritem[j]]]);
          j = rrhs[-ritem[j]];
          while (j < itemno)
            printf(" %s", symbol_name[ritem[j++]]);
          printf(" .");
          while (ritem[j] >= 0)
            printf(" %s", symbol_name[ritem[j++]]);
          printf("\n");
```

```
            fflush(stdout);
        }
    }
}

/* show_ritems is used for debugging */

show_ritems()
{
    int i;

    for (i = 0; i < nitems; ++i)
      printf("ritem[%d] = %d\n", i, ritem[i]);
}

/* show_rrhs is used for debugging */
show_rrhs()
{
    int i;
    for (i = 0; i < nrules; ++i)
      printf("rrhs[%d] = %d\n", i, rrhs[i]);
}

/* show_shifts is used for debugging */

show_shifts()
{
    shifts *p;
    int i, j, k;
    k = 0;
    for (p = first_shift; p; ++k, p = p->next)
    {
      if (k) printf("\n");
      printf("shift %d, number = %d, nshifts = %d\n", k, p->number,
            p->nshifts);
      j = p->nshifts;
      for (i = 0; i < j; ++i)
          printf("\t%d\n", p->shift[i]);
    }
}
save_shifts()
{
  register shifts *p;
  register short *sp1;
  register short *sp2;
  register short *send;
```

```
    p = (shifts *) allocate((unsigned) (sizeof(shifts) +
                    (nshifts - 1) * sizeof(short)));

    p->number = this_state->number;
    p->nshifts = nshifts;

    sp1 = shiftset;
    sp2 = p->shift;
    send = shiftset + nshifts;

    while (sp1 < send)
      *sp2++ = *sp1++;

    if (last_shift)
      {
        last_shift->next = p;
        last_shift = p;
      }
    else
      {
        first_shift = p;
        last_shift = p;
      }
  }

  save_reductions()
  {
    register short *isp;
    register short *rp1;
    register short *rp2;
    register int item;
    register int count;
    register reductions *p;

    short *rend;

    count = 0;
    for (isp = itemset; isp < itemsetend; isp++)
      {
        item = ritem[*isp];
        if (item < 0)
        {
          redset[count++] = -item;
        }
      }
```

```
 if (count)
    {
      p = (reductions *) allocate((unsigned) (sizeof(reductions) +
                                (count - 1) * sizeof(short)));

      p->number = this_state->number;
      p->nreds = count;

      rp1 = redset;
      rp2 = p->rules;
      rend = rp1 + count;

      while (rp1 < rend)
      *rp2++ = *rp1++;

      if (last_reduction)
      {
        last_reduction->next = p;
        last_reduction = p;
      }
      else
      {
        first_reduction = p;
        last_reduction = p;
      }
    }
}

set_derives()
{
  register int i, k;
  register int lhs;
  register short *rules;

  derives = NEW2(nsyms, short *);
  rules = NEW2(nvars + nrules, short);

  k = 0;
  for (lhs = start_symbol; lhs < nsyms; lhs++)
    {
      derives[lhs] = rules + k;
      for (i = 0; i < nrules; i++)
      {
        if (rlhs[i] == lhs)
          {
            rules[k] = i;
```

```
                k++;
            }
        }
        rules[k] = -1;
        k++;
    }

#ifdef      DEBUG
  print_derives();
#endif
}

free_derives()
{
  FREE(derives[start_symbol]);
  FREE(derives);
}

#ifdef      DEBUG
print_derives()
{
  register int i;
  register short *sp;

  printf("\nDERIVES\n\n");

  for (i = start_symbol; i < nsyms; i++)
    {
      printf("%s derives ", symbol_name[i]);
      for (sp = derives[i]; *sp >= 0; sp++)
      {
        printf("  %d", *sp);
      }
      putchar('\n');
    }

  putchar('\n');
}
#endif

set_nullable()
{
    register int i, j;
    register int empty;
    int done;

    nullable = MALLOC(nsyms);
```

```
    if (nullable == 0) no_space();

    for (i = 0; i < nsyms; ++i)
      nullable[i] = 0;

    done = 0;
    while (!done)
    {
      done = 1;
      for (i = 1; i < nitems; i++)
      {
          empty = 1;
          while ((j = ritem[i]) >= 0)
          {
            if (!nullable[j])
                empty = 0;
            ++i;
          }
          if (empty)
          {
            j = rlhs[-j];
            if (!nullable[j])
            {
                nullable[j] = 1;
                done = 0;
            }
          }
      }
    }
#ifdef DEBUG
    for (i = 0; i < nsyms; i++)
    {
      if (nullable[i])
          printf("%s is nullable\n", symbol_name[i]);
      else
          printf("%s is not nullable\n", symbol_name[i]);
    }
#endif
}

free_nullable()
{
  FREE(nullable);
}
lr0()
{
    set_derives();
```

```
    set_nullable();
    generate_states();
}
```

49.2.2.6 Mkpar.c

```
#include "defs.h"

action **parser;
int SRtotal;
int RRtotal;
short *SRconflicts;
short *RRconflicts;
short *defred;
short *rules_used;
short nunused;
short final_state;
static int SRcount;
static int RRcount;

extern action *parse_actions();
extern action *get_shifts();
extern action *add_reductions();
extern action *add_reduce();

make_parser()
{
    register int i;

    parser = NEW2(nstates, action *);
    for (i = 0; i < nstates; i++)
      parser[i] = parse_actions(i);

    find_final_state();
    remove_conflicts();
    unused_rules();
    if (SRtotal + RRtotal > 0) total_conflicts();
    defreds();
}
action *
parse_actions(stateno)
register int stateno;
{
    register action *actions;
    actions = get_shifts(stateno);
    actions = add_reductions(stateno, actions);
```

```
    return (actions);
}

action *
get_shifts(stateno)
int stateno;
{
    register action *actions, *temp;
    register shifts *sp;
    register short *to_state;
    register int i, k;
    register int symbol;

    actions = 0;
    sp = shift_table[stateno];
    if (sp)
    {
      to_state = sp->shift;
      for (i = sp->nshifts - 1; i >= 0; i--)
      {
          k = to_state[i];
          symbol = accessing_symbol[k];
          if (ISTOKEN(symbol))
          {
            temp = NEW(action);
            temp->next = actions;
            temp->symbol = symbol;
            temp->number = k;
            temp->prec = symbol_prec[symbol];
            temp->action_code = SHIFT;
            temp->assoc = symbol_assoc[symbol];
            actions = temp;
          }
      }
    }
    return (actions);
}
action *
add_reductions(stateno, actions)
int stateno;
register action *actions;
{
    register int i, j, m, n;
    register int ruleno, tokensetsize;
    register unsigned *rowp;
    tokensetsize = WORDSIZE(ntokens);
    m = lookaheads[stateno];
```

```
    n = lookaheads[stateno + 1];
    for (i = m; i < n; i++)
    {
      ruleno = LAruleno[i];
      rowp = LA + i * tokensetsize;
      for (j = ntokens - 1; j >= 0; j--)
      {
          if (BIT(rowp, j))
            actions = add_reduce(actions, ruleno, j);
      }
    }
    return (actions);
}

action *
add_reduce(actions, ruleno, symbol)
register action *actions;
register int ruleno, symbol;
{
    register action *temp, *prev, *next;

    prev = 0;
    for (next = actions; next && next->symbol < symbol; next = next-
>next)
      prev = next;

    while (next && next->symbol == symbol && next->action_code == SHIFT)
    {
      prev = next;
      next = next->next;
    }

    while (next && next->symbol == symbol &&
          next->action_code == REDUCE && next->number < ruleno)
    {
      prev = next;
      next = next->next;
    }
    temp = NEW(action);
    temp->next = next;
    temp->symbol = symbol;
    temp->number = ruleno;
    temp->prec = rprec[ruleno];
    temp->action_code = REDUCE;
    temp->assoc = rassoc[ruleno];
    if (prev)
      prev->next = temp;
```

```
    else
      actions = temp;

    return (actions);
}

find_final_state()
{
    register int goal, i;
    register short *to_state;
    register shifts *p;

    p = shift_table[0];
    to_state = p->shift;
    goal = ritem[1];
    for (i = p->nshifts - 1; i >= 0; --i)
    {
      final_state = to_state[i];
      if (accessing_symbol[final_state] == goal) break;
    }
}

unused_rules()
{
    register int i;
    register action *p;

    rules_used = (short *) MALLOC(nrules*sizeof(short));
    if (rules_used == 0) no_space();

    for (i = 0; i < nrules; ++i)
      rules_used[i] = 0;

    for (i = 0; i < nstates; ++i)
    {
      for (p = parser[i]; p; p = p->next)
      {
          if (p->action_code == REDUCE && p->suppressed == 0)
            rules_used[p->number] = 1;
      }
    }

    nunused = 0;
    for (i = 3; i < nrules; ++i)
      if (!rules_used[i]) ++nunused;

    if (nunused)
```

```
        if (nunused == 1)
            fprintf(stderr, "%s: 1 rule never reduced\n", myname);
        else
            fprintf(stderr, "%s: %d rules never reduced\n", myname,
nunused);
}

remove_conflicts()
{
    register int i;
    register int symbol;
    register action *p, *q;

    SRtotal = 0;
    RRtotal = 0;
    SRconflicts = NEW2(nstates, short);
    RRconflicts = NEW2(nstates, short);
    for (i = 0; i < nstates; i++)
    {
      SRcount = 0;
      RRcount = 0;
      for (p = parser[i]; p; p = q->next)
      {
          symbol = p->symbol;
          q = p;
          while (q->next && q->next->symbol == symbol)
            q = q->next;
          if (i == final_state && symbol == 0)
            end_conflicts(p, q);
          else if (p != q)
            resolve_conflicts(p, q);
      }
      SRtotal += SRcount;
      RRtotal += RRcount;
      SRconflicts[i] = SRcount;
      RRconflicts[i] = RRcount;
    }
}

end_conflicts(p, q)
register action *p, *q;
{
    for (;;)
    {
      SRcount++;
      p->suppressed = 1;
      if (p == q) break;
```

```
      p = p->next;
    }
}

resolve_conflicts(first, last)
register action *first, *last;
{
    register action *p;
    register int count;

    count = 1;
    for (p = first; p != last; p = p->next)
      ++count;
    assert(count > 1);

    if (first->action_code == SHIFT && count == 2 &&
          first->prec > 0 && last->prec > 0)
    {
      if (first->prec == last->prec)
      {
          if (first->assoc == LEFT)
            first->suppressed = 2;
          else if (first->assoc == RIGHT)
            last->suppressed = 2;
          else
          {
            first->suppressed = 2;
            last->suppressed = 2;
            first->action_code = ERROR;
            last->action_code = ERROR;
          }
      }
      else if (first->prec < last->prec)
          first->suppressed = 2;
      else
          last->suppressed = 2;
    }
    else
    {
      if (first->action_code == SHIFT)
          SRcount += (count - 1);
        else
          RRcount += (count - 1);
      for (p = first; p != last; p = p->next, p->suppressed = 1)
          continue;
    }
}
```

```
total_conflicts()
{
    fprintf(stderr, "%s: ", myname);
    if (SRtotal == 1)
      fprintf(stderr, "1 shift/reduce conflict");
    else if (SRtotal > 1)
      fprintf(stderr, "%d shift/reduce conflicts", SRtotal);

    if (SRtotal && RRtotal)
      fprintf(stderr, ", ");

    if (RRtotal == 1)
      fprintf(stderr, "1 reduce/reduce conflict");
    else if (RRtotal > 1)
      fprintf(stderr, "%d reduce/reduce conflicts", RRtotal);

    fprintf(stderr, ".\n");
}

int
sole_reduction(stateno)
int stateno;
{
    register int count, ruleno;
    register action *p;

    count = 0;
    ruleno = 0;
    for (p = parser[stateno]; p; p = p->next)
    {
      if (p->action_code == SHIFT && p->suppressed == 0)
          return (0);
      else if (p->action_code == REDUCE && p->suppressed == 0)
      {
          if (ruleno > 0 && p->number != ruleno)
            return (0);
          if (p->symbol != 1)
            ++count;
          ruleno = p->number;
      }
    }

    if (count == 0)
      return (0);
    return (ruleno);
}
```

```
defreds()
{
    register int i;

    defred = NEW2(nstates, short);
    for (i = 0; i < nstates; i++)
      defred[i] = sole_reduction(i);
}

free_action_row(p)
register action *p;
{
  register action *q;

  while (p)
    {
      q = p->next;
      FREE(p);
      p = q;
    }
}

free_parser()
{
  register int i;

  for (i = 0; i < nstates; i++)
    free_action_row(parser[i]);

  FREE(parser);
}
```

49.2.2.7 Output.c

```
#include "defs.h"

static int nvectors;
static int nentries;
static short **froms;
static short **tos;
static short *tally;
static short *width;
static short *state_count;
static short *order;
static short *base;
static short *pos;
static int maxtable;
```

```
static short *table;
static short *check;
static int lowzero;
static int high;

output()
{
    free_itemsets();
    free_shifts();
    free_reductions();
    output_stored_text();
    output_defines();
    output_rule_data();
    output_yydefred();
    output_actions();
    free_parser();
    output_debug();
    output_stype();
    if (rflag) write_section(tables);
    write_section(header);
    output_trailing_text();
    write_section(body);
    output_semantic_actions();
    write_section(trailer);
}

output_rule_data()
{
    register int i;
    register int j;


    fprintf(output_file, "short yylhs[] = {%42d,",
          symbol_value[start_symbol]);

    j = 10;
    for (i = 3; i < nrules; i++)
    {
      if (j >= 10)
      {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 1;
      }
        else
          ++j;
```

```
            fprintf(output_file, "%5d,", symbol_value[rlhs[i]]);
        }
        if (!rflag) outline += 2;
        fprintf(output_file, "\n};\n");

        fprintf(output_file, "short yylen[] = {%42d,", 2);

        j = 10;
        for (i = 3; i < nrules; i++)
        {
          if (j >= 10)
          {
              if (!rflag) ++outline;
              putc('\n', output_file);
              j = 1;
          }
          else
            j++;

            fprintf(output_file, "%5d,", rrhs[i + 1] - rrhs[i] - 1);
        }
        if (!rflag) outline += 2;
        fprintf(output_file, "\n};\n");
    }

    output_yydefred()
    {
        register int i, j;

        fprintf(output_file, "short yydefred[] = {%39d,",
              (defred[0] ? defred[0] - 2 : 0));

        j = 10;
        for (i = 1; i < nstates; i++)
        {
          if (j < 10)
              ++j;
          else
          {
              if (!rflag) ++outline;
              putc('\n', output_file);
              j = 1;
          }

          fprintf(output_file, "%5d,", (defred[i] ? defred[i] - 2 : 0));
        }
```

```
        if (!rflag) outline += 2;
        fprintf(output_file, "\n};\n");
    }

    output_actions()
    {
        nvectors = 2*nstates + nvars;

        froms = NEW2(nvectors, short *);
        tos = NEW2(nvectors, short *);
        tally = NEW2(nvectors, short);
        width = NEW2(nvectors, short);

        token_actions();
        FREE(lookaheads);
        FREE(LA);
        FREE(LAruleno);
        FREE(accessing_symbol);

        goto_actions();
        FREE(goto_map + ntokens);
        FREE(from_state);
        FREE(to_state);

        sort_actions();
        pack_table();
        output_base();
        output_table();
        output_check();
    }


    token_actions()
    {
        register int i, j;
        register int shiftcount, reducecount;
        register int max, min;
        register short *actionrow, *r, *s;
        register action *p;

        actionrow = NEW2(2*ntokens, short);
        for (i = 0; i < nstates; ++i)
        {
          if (parser[i])
          {
              for (j = 0; j < 2*ntokens; ++j)
              actionrow[j] = 0;
```

```
          shiftcount = 0;
          reducecount = 0;
          for (p = parser[i]; p; p = p->next)
          {
            if (p->suppressed == 0)
            {
                if (p->action_code == SHIFT)
                {
                  ++shiftcount;
                  actionrow[p->symbol] = p->number;
                }
                else if (p->action_code == REDUCE && p->number !=
defred[i])
                {
                  ++reducecount;
                  actionrow[p->symbol + ntokens] = p->number;
                }
            }
          }

          tally[i] = shiftcount;
          tally[nstates+i] = reducecount;
          width[i] = 0;
          width[nstates+i] = 0;
          if (shiftcount > 0)
          {
            froms[i] = r = NEW2(shiftcount, short);
            tos[i] = s = NEW2(shiftcount, short);
            min = MAXSHORT;
            max = 0;
            for (j = 0; j < ntokens; ++j)
            {
                if (actionrow[j])
                {
                  if (min > symbol_value[j])
                      min = symbol_value[j];
                  if (max < symbol_value[j])
                      max = symbol_value[j];
                  *r++ = symbol_value[j];
                  *s++ = actionrow[j];
                }
            }
            width[i] = max - min + 1;
          }
          if (reducecount > 0)
          {
            froms[nstates+i] = r = NEW2(reducecount, short);
```

```
            tos[nstates+i] = s = NEW2(reducecount, short);
            min = MAXSHORT;
            max = 0;
            for (j = 0; j < ntokens; ++j)
            {
                if (actionrow[ntokens+j])
                {
                  if (min > symbol_value[j])
                      min = symbol_value[j];
                  if (max < symbol_value[j])
                      max = symbol_value[j];
                  *r++ = symbol_value[j];
                  *s++ = actionrow[ntokens+j] - 2;
                }
            }
            width[nstates+i] = max - min + 1;
          }
      }
    }
    FREE(actionrow);
}

goto_actions()
{
    register int i, j, k;

    state_count = NEW2(nstates, short);

    k = default_goto(start_symbol + 1);
    fprintf(output_file, "short yydgoto[] = {%40d,", k);
    save_column(start_symbol + 1, k);

    j = 10;
    for (i = start_symbol + 2; i < nsyms; i++)
    {
      if (j >= 10)
      {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 1;
      }
      else
          ++j;

      k = default_goto(i);
      fprintf(output_file, "%5d,", k);
```

```
      save_column(i, k);
    }

    if (!rflag) outline += 2;
    fprintf(output_file, "\n};\n");
    FREE(state_count);
}

int
default_goto(symbol)
int symbol;
{
    register int i;
    register int m;
    register int n;
    register int default_state;
    register int max;

    m = goto_map[symbol];
    n = goto_map[symbol + 1];

    if (m == n) return (0);

    for (i = 0; i < nstates; i++)
      state_count[i] = 0;

    for (i = m; i < n; i++)
      state_count[to_state[i]]++;

    max = 0;
    default_state = 0;

    for (i = 0; i < nstates; i++)
    {
      if (state_count[i] > max)
      {
          max = state_count[i];
          default_state = i;
      }
    }
    return (default_state);
}
save_column(symbol, default_state)
int symbol;
int default_state;
{
    register int i;
```

```
    register int m;
    register int n;
    register short *sp;
    register short *sp1;
    register short *sp2;
    register int count;
    register int symno;

    m = goto_map[symbol];
    n = goto_map[symbol + 1];

    count = 0;
    for (i = m; i < n; i++)
    {
      if (to_state[i] != default_state)
          ++count;
    }
    if (count == 0) return;

    symno = symbol_value[symbol] + 2*nstates;

    froms[symno] = sp1 = sp = NEW2(count, short);
    tos[symno] = sp2 = NEW2(count, short);

    for (i = m; i < n; i++)
    {
      if (to_state[i] != default_state)
      {
          *sp1++ = from_state[i];
          *sp2++ = to_state[i];
      }
    }

    tally[symno] = count;
    width[symno] = sp1[-1] - sp[0] + 1;
}

sort_actions()
{
  register int i;
  register int j;
  register int k;
  register int t;
  register int w;

  order = NEW2(nvectors, short);
  nentries = 0;
```

```
  for (i = 0; i < nvectors; i++)
    {
      if (tally[i] > 0)
      {
        t = tally[i];
        w = width[i];
        j = nentries - 1;

        while (j >= 0 && (width[order[j]] < w))
          j--;

        while (j >= 0 && (width[order[j]] == w) && (tally[order[j]] <
t))
          j--;

        for (k = nentries - 1; k > j; k--)
          order[k + 1] = order[k];

        order[j + 1] = i;
        nentries++;
      }
    }
}

pack_table()
{
    register int i;
    register int place;
    register int state;

    base = NEW2(nvectors, short);
    pos = NEW2(nentries, short);

    maxtable = 1000;
    table = NEW2(maxtable, short);
    check = NEW2(maxtable, short);

    lowzero = 0;
    high = 0;

    for (i = 0; i < maxtable; i++)
      check[i] = -1;

    for (i = 0; i < nentries; i++)
    {
      state = matching_vector(i);
```

```
        if (state < 0)
            place = pack_vector(i);
        else
            place = base[state];

        pos[i] = place;
        base[order[i]] = place;
    }

    for (i = 0; i < nvectors; i++)
    {
        if (froms[i])
            FREE(froms[i]);
        if (tos[i])
            FREE(tos[i]);
    }

    FREE(froms);
    FREE(tos);
    FREE(pos);
}

/* The function matching_vector determines if the vector specified    */
/* by the input parameter matches a previously considered vector. The */
/* test at the start of the function checks if the vector represents  */
/* a row of shifts over terminal symbols or a row of reductions, or a */
/* column of shifts over a nonterminal symbol. Berkeley Yacc does not */
/* check if a column of shifts over a nonterminal symbols matches a   */
/* previously considered vector.  Because of the nature of LR parsing */
/* tables, no two columns can match.  Therefore, the only possible    */
/* match would be between a row and a column.  Such matches are       */
/* unlikely.  Therefore, to save time, no attempt is made to see if a */
/*  column matches a previously considered vector.                    */

/* Matching_vector is poorly designed.  The test could easily be made */
/* faster.  Also, it depends on the vectors being in a specific       */
/* order.                                                             */

int
matching_vector(vector)
int vector;
{
    register int i;
    register int j;
    register int k;
    register int t;
    register int w;
```

```
    register int match;
    register int prev;

    i = order[vector];
    if (i >= 2*nstates)
      return (-1);

    t = tally[i];
    w = width[i];

    for (prev = vector - 1; prev >= 0; prev--)
    {
      j = order[prev];
      if (width[j] != w || tally[j] != t)
          return (-1);

      match = 1;
      for (k = 0; match && k < t; k++)
      {
          if (tos[j][k] != tos[i][k] || froms[j][k] != froms[i][k])
            match = 0;
      }

      if (match)
          return (j);
    }

    return (-1);
}
int
pack_vector(vector)
int vector;
{
    register int i, j, k, l;
    register int t;
    register int loc;
    register int ok;
    register short *from;
    register short *to;
    int newmax;

    i = order[vector];
    t = tally[i];
    assert(t);

    from = froms[i];
    to = tos[i];
```

```
        j = lowzero - from[0];
        for (k = 1; k < t; ++k)
          if (lowzero - from[k] > j)
              j = lowzero - from[k];
        for (;; ++j)
        {
          if (j == 0)
              continue;
          ok = 1;
          for (k = 0; ok && k < t; k++)
          {
              loc = j + from[k];
              if (loc >= maxtable)
              {
                if (loc >= MAXTABLE)
                    fatal("maximum table size exceeded");

                newmax = maxtable;
                do { newmax += 200; } while (newmax <= loc);
                table = (short *) REALLOC(table, newmax*sizeof(short));
                if (table == 0) no_space();
                check = (short *) REALLOC(check, newmax*sizeof(short));
                if (check == 0) no_space();
                for (l  = maxtable; l < newmax; ++l)
                {
                    table[l] = 0;
                    check[l] = -1;
                }
                maxtable = newmax;
              }

              if (check[loc] != -1)
                ok = 0;
          }
          for (k = 0; ok && k < vector; k++)
          {
              if (pos[k] == j)
                ok = 0;
          }
          if (ok)
          {
              for (k = 0; k < t; k++)
              {
                loc = j + from[k];
                table[loc] = to[k];
                check[loc] = from[k];
```

```
            if (loc > high) high = loc;
          }

          while (check[lowzero] != -1)
            ++lowzero;

          return (j);
      }
    }
}

output_base()
{
    register int i, j;

    fprintf(output_file, "short yysindex[] = {%39d,", base[0]);

    j = 10;
    for (i = 1; i < nstates; i++)
    {
      if (j >= 10)
      {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 1;
      }
      else
          ++j;

      fprintf(output_file, "%5d,", base[i]);
    }

    if (!rflag) outline += 2;
    fprintf(output_file, "\n};\nshort yyrindex[] = {%39d,",
          base[nstates]);

    j = 10;
    for (i = nstates + 1; i < 2*nstates; i++)
    {
      if (j >= 10)
      {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 1;
      }
      else
          ++j;
```

```
        fprintf(output_file, "%5d,", base[i]);
    }

    if (!rflag) outline += 2;
    fprintf(output_file, "\n};\nshort yygindex[] = {%39d,",
          base[2*nstates]);

    j = 10;
    for (i = 2*nstates + 1; i < nvectors - 1; i++)
    {
      if (j >= 10)
      {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 1;
      }
      else
          ++j;

      fprintf(output_file, "%5d,", base[i]);
    }

    if (!rflag) outline += 2;
    fprintf(output_file, "\n};\n");
    FREE(base);
}

output_table()
{
    register int i;
    register int j;

    ++outline;
    fprintf(code_file, "#define YYTABLESIZE %d\n", high);
    fprintf(output_file, "short yytable[] = {%40d,", table[0]);

    j = 10;
    for (i = 1; i <= high; i++)
    {
      if (j >= 10)
      {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 1;
      }
      else
          ++j;
```

```
      fprintf(output_file, "%5d,", table[i]);
    }

    if (!rflag) outline += 2;
    fprintf(output_file, "\n};\n");
    FREE(table);
}

output_check()
{
    register int i;
    register int j;

    fprintf(output_file, "short yycheck[] = {%40d,", check[0]);

    j = 10;
    for (i = 1; i <= high; i++)
    {
      if (j >= 10)
      {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 1;
      }
      else
          ++j;

      fprintf(output_file, "%5d,", check[i]);
    }

    if (!rflag) outline += 2;
    fprintf(output_file, "\n};\n");
    FREE(check);
}

int
is_C_identifier(name)
char *name;
{
    register char *s;
    register int c;

    s = name;
    c = *s;
    if (c == '"')
    {
      c = *++s;
```

```
        if (!isalpha(c) && c != '_' && c != '$')
            return (0);
        while ((c = *++s) != '"')
        {
            if (!isalnum(c) && c != '_' && c != '$')
              return (0);
        }
        return (1);
      }

      if (!isalpha(c) && c != '_' && c != '$')
        return (0);
      while (c = *++s)
      {
        if (!isalnum(c) && c != '_' && c != '$')
            return (0);
      }
      return (1);
  }

  output_defines()
  {
      register int c, i;
      register char *s;

      for (i = 2; i < ntokens; ++i)
      {
        s = symbol_name[i];

        if (is_C_identifier(s))
        {
            fprintf(code_file, "#define ");
            if (dflag) fprintf(defines_file, "#define ");
            c = *s;
            if (c == '"')
            {
              while ((c = *++s) != '"')
              {
                  putc(c, code_file);
                  if (dflag) putc(c, defines_file);
              }
            }
            else
            {
              do
              {
                  putc(c, code_file);
```

```
                if (dflag) putc(c, defines_file);
            }
            while (c = *++s);
          }
          ++outline;
          fprintf(code_file, " %d\n", symbol_value[i]);
          if (dflag) fprintf(defines_file, " %d\n", symbol_value[i]);
      }
    }

    ++outline;
    fprintf(code_file, "#define YYERRCODE %d\n", symbol_value[1]);

    if (dflag && unionized)
    {
      fclose(union_file);
      union_file = fopen(union_file_name, "r");
      if (union_file == NULL) open_error(union_file_name);
      while ((c = getc(union_file)) != EOF)
          putc(c, defines_file);
      fprintf(defines_file, " YYSTYPE;\nextern YYSTYPE yylval;\n");
    }
}

output_stored_text()
{
    register int c;
    register FILE *in, *out;

    fclose(text_file);
    text_file = fopen(text_file_name, "r");
    if (text_file == NULL)
      open_error(text_file_name);
    in = text_file;
    if ((c = getc(in)) == EOF)
      return;
    out = code_file;
    if (c ==  '\n')
      ++outline;
    putc(c, out);
    while ((c = getc(in)) != EOF)
    {
      if (c == '\n')
          ++outline;
      putc(c, out);
    }
    if (!lflag)
```

```
        fprintf(out, line_format, ++outline + 1, code_file_name);
}

output_debug()
{
    register int i, j, k, max;
    char **symnam, *s;

    ++outline;
    fprintf(code_file, "#define YYFINAL %d\n", final_state);
    outline += 3;
    fprintf(code_file, "#ifndef YYDEBUG\n#define YYDEBUG %d\n#endif\n",
          tflag);
    if (rflag)
      fprintf(output_file, "#ifndef YYDEBUG\n#define YYDEBUG
%d\n#endif\n",
            tflag);

    max = 0;
    for (i = 2; i < ntokens; ++i)
      if (symbol_value[i] > max)
          max = symbol_value[i];
    ++outline;
    fprintf(code_file, "#define YYMAXTOKEN %d\n", max);

    symnam = (char **) MALLOC((max+1)*sizeof(char *));
    if (symnam == 0) no_space();

    /* Note that it is  not necessary to initialize the element   */
    /* symnam[max].                                               */
    for (i = 0; i < max; ++i)
      symnam[i] = 0;
    for (i = ntokens - 1; i >= 2; --i)
      symnam[symbol_value[i]] = symbol_name[i];
    symnam[0] = "end-of-file";

    if (!rflag) ++outline;
    fprintf(output_file, "#if YYDEBUG\nchar *yyname[] = {");
    j = 80;
    for (i = 0; i <= max; ++i)
    {
      if (s = symnam[i])
      {
          if (s[0] == '"')
          {
            k = 7;
```

```
    while (*++s != '"')
    {
        ++k;
        if (*s == '\\')
        {
          k += 2;
          if (*++s == '\\')
              ++k;
        }
    }
    j += k;
    if (j > 80)
    {
        if (!rflag) ++outline;
        putc('\n', output_file);
        j = k;
    }
    fprintf(output_file, "\"\\\"");
    s = symnam[i];
    while (*++s != '"')
    {
        if (*s == '\\')
        {
          fprintf(output_file, "\\\\");
          if (*++s == '\\')
              fprintf(output_file, "\\\\");
          else
              putc(*s, output_file);
        }
        else
          putc(*s, output_file);
    }
    fprintf(output_file, "\\\"\",");
}
else if (s[0] == '\'')
{
    if (s[1] == '"')
    {
        j += 7;
        if (j > 80)
        {
          if (!rflag) ++outline;
          putc('\n', output_file);
          j = 7;
        }
        fprintf(output_file, "\"'\\\"'\",");
    }
```

```
        else
        {
            k = 5;
            while (*++s != '\'')
            {
              ++k;
              if (*s == '\\')
              {
                  k += 2;
                  if (*++s == '\\')
                    ++k;
              }
            }
            j += k;
            if (j > 80)
            {
              if (!rflag) ++outline;
              putc('\n', output_file);
              j = k;
            }
            fprintf(output_file, "\"'");
            s = symnam[i];
            while (*++s != '\'')
            {
              if (*s == '\\')
              {
                  fprintf(output_file, "\\\\");
                  if (*++s == '\\')
                    fprintf(output_file, "\\\\");
                  else
                    putc(*s, output_file);
              }
              else
                  putc(*s, output_file);
            }
            fprintf(output_file, "'\",");
        }
      }
      else
      {
        k = strlen(s) + 3;
        j += k;
        if (j > 80)
        {
            if (!rflag) ++outline;
            putc('\n', output_file);
```

```
                j = k;
            }
            putc('"', output_file);
            do { putc(*s, output_file); } while (*++s);
            fprintf(output_file, "\",");
        }
    }
    else
    {
        j += 2;
        if (j > 80)
        {
            if (!rflag) ++outline;
            putc('\n', output_file);
            j = 2;
        }
        fprintf(output_file, "0,");
    }
}
if (!rflag) outline += 2;
fprintf(output_file, "\n};\n");
FREE(symnam);

if (!rflag) ++outline;
fprintf(output_file, "char *yyrule[] = {\n");
for (i = 2; i < nrules; ++i)
{
    fprintf(output_file, "\"%s :", symbol_name[rlhs[i]]);
    for (j = rrhs[i]; ritem[j] > 0; ++j)
    {
        s = symbol_name[ritem[j]];
        if (s[0] == '"')
        {
            fprintf(output_file, " \\\"");
            while (*++s != '"')
            {
                if (*s == '\\')
                {
                    if (s[1] == '\\')
                        fprintf(output_file, "\\\\\\\\");
                    else
                        fprintf(output_file, "\\\\%c", s[1]);
                    ++s;
                }
                else
                    putc(*s, output_file);
            }
```

```
            fprintf(output_file, "\\\"");
          }
          else if (s[0] == '\'')
          {
            if (s[1] == '"')
                fprintf(output_file, " '\\\"'");
            else if (s[1] == '\\')
            {
                if (s[2] == '\\')
                  fprintf(output_file, " '\\\\\\\\");
                else
                  fprintf(output_file, " '\\\\%c", s[2]);
                s += 2;
                while (*++s != '\'')
                  putc(*s, output_file);
                putc('\'', output_file);
            }
            else
                fprintf(output_file, " '%c'", s[1]);
          }
          else
            fprintf(output_file, " %s", s);
      }
      if (!rflag) ++outline;
      fprintf(output_file, "\",\n");
    }

    if (!rflag) outline += 2;
    fprintf(output_file, "};\n#endif\n");
}

output_stype()
{
    if (!unionized && ntags == 0)
    {
      outline += 3;
      fprintf(code_file, "#ifndef YYSTYPE\ntypedef int
YYSTYPE;\n#endif\n");
    }
}

output_trailing_text()
{
    register int c, last;
    register FILE *in, *out;
    if (line == 0)
      return;
```

```
    in = input_file;
    out = code_file;
    c = *cptr;
    if (c == '\n')
    {
      ++lineno;
      if ((c = getc(in)) == EOF)
          return;
      if (!lflag)
      {
          ++outline;
          fprintf(out, line_format, lineno, input_file_name);
      }
      if (c == '\n')
          ++outline;
      putc(c, out);
      last = c;
    }
    else
    {
      if (!lflag)
      {
          ++outline;
          fprintf(out, line_format, lineno, input_file_name);
      }
      do { putc(c, out); } while ((c = *++cptr) != '\n');
      ++outline;
      putc('\n', out);
      last = '\n';
    }
    while ((c = getc(in)) != EOF)
    {
      if (c == '\n')
          ++outline;
      putc(c, out);
      last = c;
    }

    if (last != '\n')
    {
      ++outline;
      putc('\n', out);
    }
    if (!lflag)
      fprintf(out, line_format, ++outline + 1, code_file_name);
}
```

```
output_semantic_actions()
{
    register int c, last;
    register FILE *out;

    fclose(action_file);
    action_file = fopen(action_file_name, "r");
    if (action_file == NULL)
      open_error(action_file_name);

    if ((c = getc(action_file)) == EOF)
      return;

    out = code_file;
    last = c;
    if (c == '\n')
      ++outline;
    putc(c, out);
    while ((c = getc(action_file)) != EOF)
    {
      if (c == '\n')
          ++outline;
      putc(c, out);
      last = c;
    }
    if (last != '\n')
    {
      ++outline;
      putc('\n', out);
    }
    if (!lflag)
      fprintf(out, line_format, ++outline + 1, code_file_name);
}

free_itemsets()
{
    register core *cp, *next;

    FREE(state_table);
    for (cp = first_state; cp; cp = next)
    {
      next = cp->next;
      FREE(cp);
    }
}
```

```
free_shifts()
{
    register shifts *sp, *next;

    FREE(shift_table);
    for (sp = first_shift; sp; sp = next)
    {
      next = sp->next;
      FREE(sp);
    }
}

free_reductions()
{
    register reductions *rp, *next;

    FREE(reduction_table);
    for (rp = first_reduction; rp; rp = next)
    {
      next = rp->next;
      FREE(rp);
    }
}
```

49.2.2.8 Reader.c

```
#include "defs.h"

/*  The line size must be a positive integer.  One hundred was chosen */
/*  because few lines in Yacc input grammars exceed 100 characters.   */
/*  Note that if a line exceeds LINESIZE characters, the line buffer  */
/*  will be expanded to accomodate it.                                */

#define LINESIZE 100

char *cache;
int cinc, cache_size;

int ntags, tagmax;
char **tag_table;

char saw_eof, unionized;
char *cptr, *line;
int linesize;

bucket *goal;
int prec;
```

```
int gensym;
char last_was_action;

int maxitems;
bucket **pitem;

int maxrules;
bucket **plhs;

int name_pool_size;
char *name_pool;

char line_format[] = "#line %d \"%s\"\n";

cachec(c)
int c;
{
    assert(cinc >= 0);
    if (cinc >= cache_size)
    {
        cache_size += 256;
        cache = REALLOC(cache, cache_size);
        if (cache == 0) no_space();
    }
    cache[cinc] = c;
    ++cinc;
}

get_line()
{
    register FILE *f = input_file;
    register int c;
    register int i;

    if (saw_eof || (c = getc(f)) == EOF)
    {
        if (line) { FREE(line); line = 0; }
        cptr = 0;
        saw_eof = 1;
        return;
    }

    if (line == 0 || linesize != (LINESIZE + 1))
    {
        if (line) FREE(line);
        linesize = LINESIZE + 1;
        line = MALLOC(linesize);
```

```
        if (line == 0) no_space();
    }

    i = 0;
    ++lineno;
    for (;;)
    {
        line[i]  =  c;
        if (c == '\n') { cptr = line; return; }
        if (++i >= linesize)
        {
            linesize += LINESIZE;
            line = REALLOC(line, linesize);
            if (line ==  0) no_space();
        }
        c = getc(f);
        if (c ==  EOF)
        {
            line[i] = '\n';
            saw_eof = 1;
            cptr = line;
            return;
        }
    }
}

char *
dup_line()
{
    register char *p, *s, *t;

    if (line == 0) return (0);
    s = line;
    while (*s != '\n') ++s;
    p = MALLOC(s - line + 1);
    if (p == 0) no_space();

    s = line;
    t = p;
    while ((*t++ = *s++) != '\n') continue;
    return (p);
}

skip_comment()
{
    register char *s;
```

```
    int st_lineno = lineno;
    char *st_line = dup_line();
    char *st_cptr = st_line + (cptr - line);

    s = cptr + 2;
    for (;;)
    {
        if (*s == '*' && s[1] == '/')
        {
            cptr = s + 2;
            FREE(st_line);
            return;
        }
        if (*s == '\n')
        {
            get_line();
            if (line == 0)
                unterminated_comment(st_lineno, st_line, st_cptr);
            s = cptr;
        }
        else
            ++s;
    }
}
int
nextc()
{
    register char *s;
    if (line == 0)
    {
        get_line();
        if (line == 0)
            return (EOF);
    }

    s = cptr;
    for (;;)
    {
        switch (*s)
        {
        case '\n':
            get_line();
            if (line == 0) return (EOF);
            s = cptr;
            break;

        case ' ':
```

```
        case '\t':
        case '\f':
        case '\r':
        case '\v':
        case ',':
        case ';':
            ++s;
            break;

        case '\\':
            cptr = s;
            return ('%');

        case '/':
            if (s[1] == '*')
            {
                cptr = s;
                skip_comment();
                s = cptr;
                break;
            }
            else if (s[1] == '/')
            {
                get_line();
                if (line == 0) return (EOF);
                s = cptr;
                break;
            }
            /* fall through */
        default:
            cptr = s;
            return (*s);
        }
    }
}

int
keyword()
{
    register int c;
    char *t_cptr = cptr;

    c = *++cptr;
    if (isalpha(c))
    {
        cinc = 0;
```

```
    for (;;)
        {
            if (isalpha(c))
            {
                if (isupper(c)) c = tolower(c);
                cachec(c);
            }
            else if (isdigit(c) || c == '_' || c == '.' || c == '$')
                cachec(c);
            else
                break;
            c = *++cptr;
        }
        cachec(NUL);

        if (strcmp(cache, "token") == 0 || strcmp(cache, "term") == 0)
            return (TOKEN);
        if (strcmp(cache, "type") == 0)
            return (TYPE);
        if (strcmp(cache, "left") == 0)
            return (LEFT);
        if (strcmp(cache, "right") == 0)
            return (RIGHT);
        if (strcmp(cache, "nonassoc") == 0 || strcmp(cache, "binary") ==
0)
            return (NONASSOC);
        if (strcmp(cache, "start") == 0)
            return (START);
        if (strcmp(cache, "union") == 0)
            return (UNION);


        if (strcmp(cache, "ident") == 0)
            return (IDENT);
    }
    else
    {
        ++cptr;
        if (c == '{')
            return (TEXT);
        if (c == '%' || c == '\\')
            return (MARK);
        if (c == '<')
            return (LEFT);
        if (c == '>')
            return (RIGHT);
        if (c == '0')
```

```
            return (TOKEN);
        if (c == '2')
            return (NONASSOC);
    }
    syntax_error(lineno, line, t_cptr);
    /*NOTREACHED*/
}

copy_ident()
{
    register int c;
    register FILE *f = output_file;

    c = nextc();
    if (c == EOF) unexpected_EOF();
    if (c != '"') syntax_error(lineno, line, cptr);
    ++outline;
    fprintf(f, "#ident \"");
    for (;;)
    {
        c = *++cptr;
        if (c == '\n')
        {
            fprintf(f, "\"\n");
            return;
        }
        putc(c, f);
        if (c == '"')
        {
            putc('\n', f);
            ++cptr;

            return;
        }
    }
}

copy_text()
{
    register int c;
    int quote;
    register FILE *f = text_file;
    int need_newline = 0;
    int t_lineno = lineno;
    char *t_line = dup_line();
    char *t_cptr = t_line + (cptr - line - 2);
```

```
        if (*cptr == '\n')
        {
            get_line();
            if (line == 0)
                unterminated_text(t_lineno, t_line, t_cptr);
        }
        if (!lflag) fprintf(f, line_format, lineno, input_file_name);

    loop:
        c = *cptr++;
        switch (c)
        {
        case '\n':
        next_line:
            putc('\n', f);
            need_newline = 0;
            get_line();
            if (line) goto loop;
            unterminated_text(t_lineno, t_line, t_cptr);

        case '\'':
        case '"':
            {
                int s_lineno = lineno;
                char *s_line = dup_line();
                char *s_cptr = s_line + (cptr - line - 1);

                quote = c;
                putc(c, f);
                for (;;)
                {
                    c = *cptr++;
                    putc(c, f);
                    if (c == quote)
                    {
                        need_newline = 1;
                        FREE(s_line);
                        goto loop;
                    }
                    if (c == '\n')
                        unterminated_string(s_lineno, s_line, s_cptr);
                    if (c == '\\')
                    {
                        c = *cptr++;
                        putc(c, f);
                        if (c == '\n')
                        {
```

```
                        get_line();
                        if (line == 0)
                            unterminated_string(s_lineno, s_line,
s_cptr);
                    }
                }
            }
        }

    case '/':
        putc(c, f);
        need_newline = 1;
        c = *cptr;
        if (c == '/')
        {
            putc('*', f);
            while ((c = *++cptr) != '\n')
            {
                if (c == '*' && cptr[1] == '/')
                    fprintf(f, "* ");
                else
                    putc(c, f);
            }
            fprintf(f, "*/");
            goto next_line;
        }
        if (c == '*')
        {
            int c_lineno = lineno;
            char *c_line = dup_line();
            char *c_cptr = c_line + (cptr - line - 1);

            putc('*', f);
            ++cptr;
            for (;;)
            {
                c = *cptr++;
                putc(c, f);
                if (c == '*' && *cptr == '/')
                {
                    putc('/', f);
                    ++cptr;
                    FREE(c_line);
                    goto loop;
                }
                if (c == '\n')
                {
```

```
                    get_line();
                    if (line == 0)
                        unterminated_comment(c_lineno, c_line, c_cptr);
                }
            }
        }
        need_newline = 1;
        goto loop;

    case '%':
    case '\\':
        if (*cptr == '}')
        {
            if (need_newline) putc('\n', f);
            ++cptr;
            FREE(t_line);
            return;
        }
        /* fall through */

    default:
        putc(c, f);
        need_newline = 1;
        goto loop;
    }
}

copy_union()
{
    register int c;
    int quote;
    int depth;
    int u_lineno = lineno;
    char *u_line = dup_line();
    char *u_cptr = u_line + (cptr - line - 6);

    if (unionized) over_unionized(cptr - 6);
    unionized = 1;

    if (!lflag)
        fprintf(text_file, line_format, lineno, input_file_name);

    fprintf(text_file, "typedef union");
    if (dflag) fprintf(union_file, "typedef union");

    depth = 0;
loop:
```

```
c = *cptr++;
putc(c, text_file);
if (dflag) putc(c, union_file);
switch (c)
{
case '\n':
next_line:
    get_line();
    if (line == 0) unterminated_union(u_lineno, u_line, u_cptr);
    goto loop;

case '{':
    ++depth;
    goto loop;

case '}':
    if (--depth == 0)
    {
        fprintf(text_file, " YYSTYPE;\n");
        FREE(u_line);
        return;
    }
    goto loop;

case '\'':
case '"':
    {
        int s_lineno = lineno;
        char *s_line = dup_line();
        char *s_cptr = s_line + (cptr - line - 1);

        quote = c;
        for (;;)
        {
            c = *cptr++;
            putc(c, text_file);
            if (dflag) putc(c, union_file);
            if (c == quote)
            {
                FREE(s_line);
                goto loop;
            }
            if (c == '\n')
                unterminated_string(s_lineno, s_line, s_cptr);
            if (c == '\\')
            {
                c = *cptr++;
```

```
                        putc(c, text_file);
                        if (dflag) putc(c, union_file);
                        if (c == '\n')
                        {
                            get_line();
                            if (line == 0)
                                unterminated_string(s_lineno, s_line,
s_cptr);
                        }
                    }
                }
            }

        case '/':
            c = *cptr;
            if (c == '/')
            {
                putc('*', text_file);
                if (dflag) putc('*', union_file);
                while ((c = *++cptr) != '\n')
                {
                    if (c == '*' && cptr[1] == '/')
                    {
                        fprintf(text_file, "* ");
                        if (dflag) fprintf(union_file, "* ");
                    }
                    else
                    {
                        putc(c, text_file);
                        if (dflag) putc(c, union_file);
                    }
                }
                fprintf(text_file, "*/\n");
                if (dflag) fprintf(union_file, "*/\n");
                goto next_line;
            }
            if (c == '*')
            {
                int c_lineno = lineno;
                char *c_line = dup_line();
                char *c_cptr = c_line + (cptr - line - 1);

                putc('*', text_file);
                if (dflag) putc('*', union_file);
                ++cptr;
                for (;;)
                {
```

```
                    c = *cptr++;
                    putc(c, text_file);
                    if (dflag) putc(c, union_file);
                    if (c == '*' && *cptr == '/')
                    {
                        putc('/', text_file);
                        if (dflag) putc('/', union_file);
                        ++cptr;
                        FREE(c_line);
                        goto loop;
                    }
                    if (c == '\n')
                    {
                        get_line();
                        if (line == 0)
                            unterminated_comment(c_lineno, c_line, c_cptr);
                    }
                }
            }
            goto loop;

        default:
            goto loop;
        }
}

int
hexval(c)
int c;
{
    if (c >= '0' && c <= '9')
        return (c - '0');
    if (c >= 'A' && c <= 'F')
        return (c - 'A' + 10);
    if (c >= 'a' && c <= 'f')
        return (c - 'a' + 10);
    return (-1);
}

bucket *
get_literal()
{
    register int c, quote;
    register int i;
    register int n;
    register char *s;
    register bucket *bp;
```

```
    int s_lineno = lineno;
    char *s_line = dup_line();
    char *s_cptr = s_line + (cptr - line);

    quote = *cptr++;
    cinc = 0;
    for (;;)
    {
        c = *cptr++;
        if (c == quote) break;
        if (c == '\n') unterminated_string(s_lineno, s_line, s_cptr);
        if (c == '\\')
        {
            char *c_cptr = cptr - 1;

            c = *cptr++;
            switch (c)
            {
            case '\n':
                get_line();
                if (line == 0) unterminated_string(s_lineno, s_line,
s_cptr);
                continue;

            case '0': case '1': case '2': case '3':
            case '4': case '5': case '6': case '7':
                n = c - '0';
                c = *cptr;
                if (IS_OCTAL(c))
                {
                    n = (n << 3) + (c - '0');
                    c = *++cptr;
                    if (IS_OCTAL(c))
                    {
                        n = (n << 3) + (c - '0');
                        ++cptr;
                    }
                }
                if (n > MAXCHAR) illegal_character(c_cptr);
                c = n;
                break;

            case 'x':
                c = *cptr++;
                n = hexval(c);
                if (n < 0 || n >= 16)
                    illegal_character(c_cptr);
```

```
            for (;;)
            {
                c = *cptr;
                i = hexval(c);
                if (i < 0 || i >= 16) break;
                ++cptr;
                n = (n << 4) + i;
                if (n > MAXCHAR) illegal_character(c_cptr);
            }
            c = n;
            break;

        case 'a': c = 7; break;
        case 'b': c = '\b'; break;
        case 'f': c = '\f'; break;
        case 'n': c = '\n'; break;
        case 'r': c = '\r'; break;
        case 't': c = '\t'; break;
        case 'v': c = '\v'; break;
        }
    }
    cachec(c);
}
FREE(s_line);

n = cinc;
s = MALLOC(n);
if (s == 0) no_space();

for (i = 0; i < n; ++i)
    s[i] = cache[i];

cinc = 0;
if (n == 1)
    cachec('\'');
else
    cachec('"');

for (i = 0; i < n; ++i)
{
    c = ((unsigned char *)s)[i];
    if (c == '\\' || c == cache[0])
    {
        cachec('\\');
        cachec(c);
    }
    else if (isprint(c))
```

```
                cachec(c);
            else
            {
                cachec('\\');
                switch (c)
                {
                case 7: cachec('a'); break;
                case '\b': cachec('b'); break;
                case '\f': cachec('f'); break;
                case '\n': cachec('n'); break;
                case '\r': cachec('r'); break;
                case '\t': cachec('t'); break;
                case '\v': cachec('v'); break;
                default:
                    cachec(((c >> 6) & 7) + '0');
                    cachec(((c >> 3) & 7) + '0');
                    cachec((c & 7) + '0');
                    break;
                }
            }
        }

        if (n == 1)
            cachec('\'');
        else
            cachec('"');

        cachec(NUL);
        bp = lookup(cache);
        bp->class = TERM;
        if (n == 1 && bp->value == UNDEFINED)
            bp->value = *(unsigned char *)s;
        FREE(s);

        return (bp);
    }

    int
    is_reserved(name)
    char *name;
    {
        char *s;

        if (strcmp(name, ".") == 0 ||
                strcmp(name, "$accept") == 0 ||
                strcmp(name, "$end") == 0)
            return (1);
```

```
    if (name[0] == '$' && name[1] == '$' && isdigit(name[2]))
    {
        s = name + 3;
        while (isdigit(*s)) ++s;
        if (*s == NUL) return (1);
    }

    return (0);
}

bucket *
get_name()
{
    register int c;

    cinc = 0;
    for (c = *cptr; IS_IDENT(c); c = *++cptr)
        cachec(c);
    cachec(NUL);

    if (is_reserved(cache)) used_reserved(cache);

    return (lookup(cache));
}

int
get_number()
{
    register int c;
    register int n;

    n = 0;
    for (c = *cptr; isdigit(c); c = *++cptr)
        n = 10*n + (c - '0');

    return (n);
}

char *
get_tag()
{
    register int c;
    register int i;
    register char *s;
    int t_lineno = lineno;
    char *t_line = dup_line();
    char *t_cptr = t_line + (cptr - line);
```

```
    ++cptr;
    c = nextc();
    if (c == EOF) unexpected_EOF();
    if (!isalpha(c) && c != '_' && c != '$')
        illegal_tag(t_lineno, t_line, t_cptr);

    cinc = 0;
    do { cachec(c); c = *++cptr; } while (IS_IDENT(c));
    cachec(NUL);
    c = nextc();
    if (c == EOF) unexpected_EOF();
    if (c != '>')
        illegal_tag(t_lineno, t_line, t_cptr);
    ++cptr;

    for (i = 0; i < ntags; ++i)
    {
        if (strcmp(cache, tag_table[i]) == 0)
            return (tag_table[i]);
    }

    if (ntags >= tagmax)
    {
        tagmax += 16;
        tag_table = (char **)
                        (tag_table ? REALLOC(tag_table,
tagmax*sizeof(char *))
                                   : MALLOC(tagmax*sizeof(char *)));
        if (tag_table == 0) no_space();
    }

    s = MALLOC(cinc);
    if  (s == 0) no_space();
    strcpy(s, cache);
    tag_table[ntags] = s;
    ++ntags;
    FREE(t_line);
    return (s);
}

declare_tokens(assoc)
int assoc;
{
    register int c;
    register bucket *bp;
    int value;
    char *tag = 0;
```

```
if (assoc != TOKEN) ++prec;

c = nextc();
if (c == EOF) unexpected_EOF();
if (c == '<')
{
    tag = get_tag();
    c = nextc();
    if (c == EOF) unexpected_EOF();
}

for (;;)
{
    if (isalpha(c) || c == '_' || c == '.' || c == '$')
        bp = get_name();
    else if (c == '\'' || c == '"')
        bp = get_literal();
    else
        return;

    if (bp == goal) tokenized_start(bp->name);
    bp->class = TERM;

    if (tag)
    {
        if (bp->tag && tag != bp->tag)
            retyped_warning(bp->name);
        bp->tag = tag;
    }
    if (assoc != TOKEN)
    {
        if (bp->prec && prec != bp->prec)
            reprec_warning(bp->name);
        bp->assoc = assoc;
        bp->prec = prec;
    }

    c = nextc();
    if (c == EOF) unexpected_EOF();
    value = UNDEFINED;
    if (isdigit(c))
    {
        value = get_number();
        if (bp->value != UNDEFINED && value != bp->value)
            revalued_warning(bp->name);
        bp->value = value;
        c = nextc();
```

```
                if (c == EOF) unexpected_EOF();
            }
        }
    }

    declare_types()
    {
        register int c;
        register bucket *bp;
        char *tag;

        c = nextc();
        if (c == EOF) unexpected_EOF();
        if (c != '<') syntax_error(lineno, line, cptr);
        tag = get_tag();

        for (;;)
        {
            c = nextc();
            if (isalpha(c) || c == '_' || c == '.' || c == '$')
                bp = get_name();
            else if (c == '\'' || c == '"')
                bp = get_literal();
            else
                return;

            if (bp->tag && tag != bp->tag)
                retyped_warning(bp->name);

            bp->tag = tag;
        }
    }

    declare_start()
    {
        register int c;
        register bucket *bp;

        c = nextc();
        if (c == EOF) unexpected_EOF();
        if (!isalpha(c) && c != '_' && c != '.' && c != '$')
            syntax_error(lineno, line, cptr);
        bp = get_name();
        if (bp->class == TERM)
            terminal_start(bp->name);
        if (goal && goal != bp)
            restarted_warning();
```

```
    goal = bp;
}

read_declarations()
{
    register int c, k;

    cache_size = 256;
    cache = MALLOC(cache_size);
    if (cache == 0) no_space();

    for (;;)
    {
        c = nextc();
        if (c == EOF) unexpected_EOF();
        if (c != '%') syntax_error(lineno, line, cptr);
        switch (k = keyword())
        {
        case MARK:
            return;

        case IDENT:
            copy_ident();
            break;

        case TEXT:
            copy_text();
            break;

        case UNION:
            copy_union();
            break;

        case TOKEN:
        case LEFT:
        case RIGHT:
        case NONASSOC:
            declare_tokens(k);
            break;

        case TYPE:
            declare_types();
            break;

        case START:
            declare_start();
```

```
            break;
          }
      }
  }

  initialize_grammar()
  {
      nitems = 4;
      maxitems = 300;
      pitem = (bucket **) MALLOC(maxitems*sizeof(bucket *));
      if (pitem == 0) no_space();
      pitem[0] = 0;
      pitem[1] = 0;
      pitem[2] = 0;
      pitem[3] = 0;

      nrules = 3;
      maxrules = 100;
      plhs = (bucket **) MALLOC(maxrules*sizeof(bucket *));
      if (plhs == 0) no_space();
      plhs[0] = 0;
      plhs[1] = 0;
      plhs[2] = 0;
      rprec = (short *) MALLOC(maxrules*sizeof(short));
      if (rprec == 0) no_space();
      rprec[0] = 0;
      rprec[1] = 0;
      rprec[2] = 0;
      rassoc = (char *) MALLOC(maxrules*sizeof(char));
      if (rassoc == 0) no_space();
      rassoc[0] = TOKEN;
      rassoc[1] = TOKEN;
      rassoc[2] = TOKEN;
  }

  expand_items()
  {
      maxitems += 300;
      pitem = (bucket **) REALLOC(pitem, maxitems*sizeof(bucket *));
      if (pitem == 0) no_space();
  }

  expand_rules()
  {
      maxrules += 100;
      plhs = (bucket **) REALLOC(plhs, maxrules*sizeof(bucket *));
      if (plhs == 0) no_space();
```

```
    rprec = (short *) REALLOC(rprec, maxrules*sizeof(short));
    if (rprec == 0) no_space();
    rassoc = (char *) REALLOC(rassoc, maxrules*sizeof(char));
    if (rassoc == 0) no_space();
}

advance_to_start()
{
    register int c;
    register bucket *bp;
    char *s_cptr;
    int s_lineno;

    for (;;)
    {
        c = nextc();
        if (c != '%') break;
        s_cptr = cptr;
        switch (keyword())
        {
        case MARK:
            no_grammar();

        case TEXT:
            copy_text();
            break;

        case START:
            declare_start();
            break;
        default:
            syntax_error(lineno, line, s_cptr);
        }
    }

    c = nextc();
    if (!isalpha(c) && c != '_' && c != '.' && c != '_')
        syntax_error(lineno, line, cptr);
    bp = get_name();
    if (goal == 0)
    {
        if (bp->class == TERM)
            terminal_start(bp->name);
        goal = bp;
    }

    s_lineno = lineno;
```

```
        c = nextc();
        if (c == EOF) unexpected_EOF();
        if (c != ':') syntax_error(lineno, line, cptr);
        start_rule(bp, s_lineno);
        ++cptr;
    }

    start_rule(bp, s_lineno)
    register bucket *bp;
    int s_lineno;
    {
        if (bp->class == TERM)
            terminal_lhs(s_lineno);
        bp->class = NONTERM;
        if (nrules >= maxrules)
            expand_rules();
        plhs[nrules] = bp;
        rprec[nrules] = UNDEFINED;
        rassoc[nrules] = TOKEN;
    }

    end_rule()
    {
        register int i;
        if (!last_was_action && plhs[nrules]->tag)
        {
            for (i = nitems - 1; pitem[i]; --i) continue;
            if (pitem[i+1] == 0 || pitem[i+1]->tag != plhs[nrules]->tag)
                default_action_warning();
        }

        last_was_action = 0;
        if (nitems >= maxitems) expand_items();
        pitem[nitems] = 0;
        ++nitems;
        ++nrules;
    }

    insert_empty_rule()
    {
        register bucket *bp, **bpp;

        assert(cache);
        sprintf(cache, "$$%d", ++gensym);
        bp = make_bucket(cache);
        last_symbol->next = bp;
        last_symbol = bp;
```

```
    bp->tag = plhs[nrules]->tag;
    bp->class = NONTERM;

    if ((nitems += 2) > maxitems)
        expand_items();
    bpp = pitem + nitems - 1;
    *bpp-- = bp;
    while (bpp[0] = bpp[-1]) --bpp;

    if (++nrules >= maxrules)
        expand_rules();
    plhs[nrules] = plhs[nrules-1];
    plhs[nrules-1] = bp;
    rprec[nrules] = rprec[nrules-1];
    rprec[nrules-1] = 0;
    rassoc[nrules] = rassoc[nrules-1];
    rassoc[nrules-1] = TOKEN;
}

add_symbol()
{
    register int c;
    register bucket *bp;
    int s_lineno = lineno;

    c = *cptr;
    if (c == '\'' || c == '"')
        bp = get_literal();
    else
        bp = get_name();
    c = nextc();
    if (c == ':')
    {
        end_rule();
        start_rule(bp, s_lineno);
        ++cptr;
        return;
    }

    if (last_was_action)
        insert_empty_rule();
    last_was_action = 0;

    if (++nitems > maxitems)
        expand_items();
    pitem[nitems-1] = bp;
}
```

```
copy_action()
{
    register int c;
    register int i, n;
    int depth;
    int quote;
    char *tag;
    register FILE *f = action_file;
    int a_lineno = lineno;
    char *a_line = dup_line();
    char *a_cptr = a_line + (cptr - line);

    if (last_was_action)
        insert_empty_rule();
    last_was_action = 1;

    fprintf(f, "case %d:\n", nrules - 2);
    if (!lflag)
        fprintf(f, line_format, lineno, input_file_name);
    if (*cptr == '=') ++cptr;

    n = 0;
    for (i = nitems - 1; pitem[i]; --i) ++n;

    depth = 0;
loop:
    c = *cptr;

    if (c == '$')
    {
        if (cptr[1] == '<')
        {
            int d_lineno = lineno;
            char *d_line = dup_line();
            char *d_cptr = d_line + (cptr - line);

            ++cptr;
            tag = get_tag();
            c = *cptr;
            if (c == '$')
            {
                fprintf(f, "yyval.%s", tag);
                ++cptr;
                FREE(d_line);
                goto loop;
            }
```

```
            else if (isdigit(c))
            {
                i = get_number();
                if (i > n) dollar_warning(d_lineno, i);
                fprintf(f, "yyvsp[%d].%s", i - n, tag);
                FREE(d_line);
                goto loop;
            }
            else if (c == '-' && isdigit(cptr[1]))
            {
                ++cptr;
                i = -get_number() - n;
                fprintf(f, "yyvsp[%d].%s", i, tag);
                FREE(d_line);
                goto loop;
            }
            else
                dollar_error(d_lineno, d_line, d_cptr);
        }
        else if (cptr[1] == '$')
        {
            if (ntags)
            {
                tag = plhs[nrules]->tag;
                if (tag == 0) untyped_lhs();
                fprintf(f, "yyval.%s", tag);
            }
            else
                fprintf(f, "yyval");
            cptr += 2;
            goto loop;
        }
        else if (isdigit(cptr[1]))
        {
            ++cptr;
            i = get_number();
            if (ntags)
            {
                if (i <= 0 || i > n)
                    unknown_rhs(i);
                tag = pitem[nitems + i - n - 1]->tag;
                if (tag == 0) untyped_rhs(i, pitem[nitems + i - n - 1]-
>name);
                fprintf(f, "yyvsp[%d].%s", i - n, tag);
            }
```

```
         else
            {
                if (i > n)
                    dollar_warning(lineno, i);
                fprintf(f, "yyvsp[%d]", i - n);
            }
            goto loop;
        }
        else if (cptr[1] == '-')
        {
            cptr += 2;
            i = get_number();
            if (ntags)
                unknown_rhs(-i);
            fprintf(f, "yyvsp[%d]", -i - n);
            goto loop;
        }
    }
    if (isalpha(c) || c == '_' || c == '$')
    {
        do
        {
            putc(c, f);
            c = *++cptr;
        } while (isalnum(c) || c == '_' || c == '$');

        goto loop;
    }
    putc(c, f);
    ++cptr;

    switch (c)
    {
    case '\n':
    next_line:
        get_line();
        if (line) goto loop;
        unterminated_action(a_lineno, a_line, a_cptr);

    case ';':
        if (depth > 0) goto loop;
        fprintf(f, "\nbreak;\n");
        return;

    case '{':
        ++depth;
        goto loop;
```

```
    case '}':
        if (--depth > 0) goto loop;
        fprintf(f, "\nbreak;\n");
        return;

    case '\'':
    case '"':
        {
            int s_lineno = lineno;
            char *s_line = dup_line();
            char *s_cptr = s_line + (cptr - line - 1);
            quote = c;
            for (;;)
            {
                c = *cptr++;
                putc(c, f);
                if (c == quote)
                {
                    FREE(s_line);
                    goto loop;
                }
                if (c == '\n')
                    unterminated_string(s_lineno, s_line, s_cptr);
                if (c == '\\')
                {
                    c = *cptr++;
                    putc(c, f);
                    if (c == '\n')
                    {
                        get_line();
                        if (line == 0)
                            unterminated_string(s_lineno, s_line,
s_cptr);
                    }
                }
            }
        }

    case '/':
        c = *cptr;
        if (c == '/')
        {
            putc('*', f);
            while ((c = *++cptr) != '\n')
            {
                if (c == '*' && cptr[1] == '/')
                    fprintf(f, "* ");
```

```
                else
                    putc(c, f);
            }
            fprintf(f, "*/\n");
            goto next_line;
        }
        if (c == '*')
        {
            int c_lineno = lineno;
            char *c_line = dup_line();
            char *c_cptr = c_line + (cptr - line - 1);

            putc('*', f);
            ++cptr;
            for (;;)
            {
                c = *cptr++;
                putc(c, f);
                if (c == '*' && *cptr == '/')
                {
                    putc('/', f);
                    ++cptr;
                    FREE(c_line);
                    goto loop;
                }
                if (c == '\n')
                {
                    get_line();


                    if (line == 0)
                        unterminated_comment(c_lineno, c_line, c_cptr);
                }
            }
        }
        goto loop;

    default:
        goto loop;
    }
}

int
mark_symbol()
{
    register int c;
    register bucket *bp;
```

```
    c = cptr[1];
    if (c == '%' || c == '\\')
    {
        cptr += 2;
        return (1);
    }

    if (c == '=')
        cptr += 2;
    else if ((c == 'p' || c == 'P') &&
             ((c = cptr[2]) == 'r' || c == 'R') &&
             ((c = cptr[3]) == 'e' || c == 'E') &&
             ((c = cptr[4]) == 'c' || c == 'C') &&
             ((c = cptr[5], !IS_IDENT(c))))
        cptr += 5;
    else
        syntax_error(lineno, line, cptr);

    c = nextc();
    if (isalpha(c) || c == '_' || c == '.' || c == '$')
        bp = get_name();
    else if (c == '\'' || c == '"')
        bp = get_literal();
    else
    {
        syntax_error(lineno, line, cptr);
        /*NOTREACHED*/
    }

    if (rprec[nrules] != UNDEFINED && bp->prec != rprec[nrules])
        prec_redeclared();

    rprec[nrules] = bp->prec;
    rassoc[nrules] = bp->assoc;
    return (0);
}

read_grammar()
{
    register int c;

    initialize_grammar();
    advance_to_start();

    for (;;)
    {
        c = nextc();
```

```
        if (c == EOF) break;
        if (isalpha(c) || c == '_' || c == '.' || c == '$' || c == '\''
|| c == '"')
            add_symbol();
        else if (c == '{' || c == '=')
            copy_action();
        else if (c == '|')
        {
            end_rule();
            start_rule(plhs[nrules-1], 0);
            ++cptr;
        }
        else if (c == '%')
        {
            if (mark_symbol()) break;
        }
        else
            syntax_error(lineno, line, cptr);
    }
    end_rule();
}

free_tags()
{
    register int i;

    if (tag_table == 0) return;

    for (i = 0; i < ntags; ++i)
    {
        assert(tag_table[i]);
        FREE(tag_table[i]);
    }
    FREE(tag_table);
}

pack_names()
{
    register bucket *bp;
    register char *p, *s, *t;

    name_pool_size = 13;  /* 13 == sizeof("$end") + sizeof("$accept") */
    for (bp = first_symbol; bp; bp = bp->next)
        name_pool_size += strlen(bp->name) + 1;
    name_pool = MALLOC(name_pool_size);
    if (name_pool == 0) no_space();
```

```
    strcpy(name_pool, "$accept");
    strcpy(name_pool+8, "$end");
    t = name_pool + 13;
    for (bp = first_symbol; bp; bp = bp->next)
    {
        p = t;
        s = bp->name;
        while (*t++ = *s++) continue;
        FREE(bp->name);
        bp->name = p;
    }
}

check_symbols()
{
    register bucket *bp;
    if (goal->class == UNKNOWN)
        undefined_goal(goal->name);

    for (bp = first_symbol; bp; bp = bp->next)
    {
        if (bp->class == UNKNOWN)
        {
            undefined_symbol_warning(bp->name);
            bp->class = TERM;
        }
    }
}
pack_symbols()
{
    register bucket *bp;
    register bucket **v;
    register int i, j, k, n;

    nsyms = 2;
    ntokens = 1;
    for (bp = first_symbol; bp; bp = bp->next)
    {
        ++nsyms;
        if (bp->class == TERM) ++ntokens;
    }
    start_symbol = ntokens;
    nvars = nsyms - ntokens;

    symbol_name = (char **) MALLOC(nsyms*sizeof(char *));
    if (symbol_name == 0) no_space();
    symbol_value = (short *) MALLOC(nsyms*sizeof(short));
```

```
if (symbol_value == 0) no_space();
symbol_prec = (short *) MALLOC(nsyms*sizeof(short));
if (symbol_prec == 0) no_space();
symbol_assoc = MALLOC(nsyms);
if (symbol_assoc == 0) no_space();

v = (bucket **) MALLOC(nsyms*sizeof(bucket *));
if (v == 0) no_space();

v[0] = 0;
v[start_symbol] = 0;

i = 1;
j = start_symbol + 1;
for (bp = first_symbol; bp; bp = bp->next)
{
    if (bp->class == TERM)
        v[i++] = bp;
    else
        v[j++] = bp;
}
assert(i == ntokens && j == nsyms);

for (i = 1; i < ntokens; ++i)
    v[i]->index = i;

goal->index = start_symbol + 1;
k = start_symbol + 2;
while (++i < nsyms)
    if (v[i] != goal)
    {
        v[i]->index = k;
        ++k;
    }

goal->value = 0;
k = 1;
for (i = start_symbol + 1; i < nsyms; ++i)
{
    if (v[i] != goal)
    {
        v[i]->value = k;
        ++k;
    }
}

k = 0;
```

```
for (i = 1; i < ntokens; ++i)
{
    n = v[i]->value;
    if (n > 256)
    {
        for (j = k++; j > 0 && symbol_value[j-1] > n; --j)
            symbol_value[j] = symbol_value[j-1];
        symbol_value[j] = n;
    }
}
if (v[1]->value == UNDEFINED)
    v[1]->value = 256;

j = 0;
n = 257;
for (i = 2; i < ntokens; ++i)
{
    if (v[i]->value == UNDEFINED)
    {
        while (j < k && n == symbol_value[j])
        {
            while (++j < k && n == symbol_value[j]) continue;
            ++n;
        }
        v[i]->value = n;
        ++n;
    }
}

symbol_name[0] = name_pool + 8;
symbol_value[0] = 0;
symbol_prec[0] = 0;
symbol_assoc[0] = TOKEN;
for (i = 1; i < ntokens; ++i)
{
    symbol_name[i] = v[i]->name;
    symbol_value[i] = v[i]->value;
    symbol_prec[i] = v[i]->prec;
    symbol_assoc[i] = v[i]->assoc;
}
symbol_name[start_symbol] = name_pool;
symbol_value[start_symbol] = -1;
symbol_prec[start_symbol] = 0;
symbol_assoc[start_symbol] = TOKEN;
for (++i; i < nsyms; ++i)
{
    k = v[i]->index;
```

```
            symbol_name[k] = v[i]->name;
            symbol_value[k] = v[i]->value;
            symbol_prec[k] = v[i]->prec;
            symbol_assoc[k] = v[i]->assoc;
        }

        FREE(v);
    }

    pack_grammar()
    {
        register int i, j;
        int assoc, prec;

        ritem = (short *) MALLOC(nitems*sizeof(short));
        if (ritem == 0) no_space();
        rlhs = (short *) MALLOC(nrules*sizeof(short));
        if (rlhs == 0) no_space();
        rrhs = (short *) MALLOC((nrules+1)*sizeof(short));
        if (rrhs == 0) no_space();
        rprec = (short *) REALLOC(rprec, nrules*sizeof(short));
        if (rprec == 0) no_space();
        rassoc = REALLOC(rassoc, nrules);
        if (rassoc == 0) no_space();

        ritem[0] = -1;
        ritem[1] = goal->index;
        ritem[2] = 0;
        ritem[3] = -2;
        rlhs[0] = 0;
        rlhs[1] = 0;
        rlhs[2] = start_symbol;
        rrhs[0] = 0;
        rrhs[1] = 0;
        rrhs[2] = 1;

        j = 4;
        for (i = 3; i < nrules; ++i)
        {
            rlhs[i] = plhs[i]->index;
            rrhs[i] = j;
            assoc = TOKEN;
            prec = 0;
            while (pitem[j])
            {
                ritem[j] = pitem[j]->index;
```

```
            if (pitem[j]->class == TERM)
            {
                prec = pitem[j]->prec;
                assoc = pitem[j]->assoc;
            }
            ++j;
        }
        ritem[j] = -i;
        ++j;
        if (rprec[i] == UNDEFINED)
        {
            rprec[i] = prec;
            rassoc[i] = assoc;
        }
    }
    rrhs[i] = j;

    FREE(plhs);
    FREE(pitem);
}

print_grammar()
{
    register int i, j, k;
    int spacing;
    register FILE *f = verbose_file;

    if (!vflag) return;
    k = 1;
    for (i = 2; i < nrules; ++i)
    {
        if (rlhs[i] != rlhs[i-1])
        {
            if (i != 2) fprintf(f, "\n");
            fprintf(f, "%4d  %s :", i - 2, symbol_name[rlhs[i]]);
            spacing = strlen(symbol_name[rlhs[i]]) + 1;
        }
        else
        {
            fprintf(f, "%4d  ", i - 2);
            j = spacing;
            while (--j >= 0) putc(' ', f);
            putc('|', f);
        }
        while (ritem[k] >= 0)
        {
            fprintf(f, " %s", symbol_name[ritem[k]]);
```

```
            ++k;
        }
        ++k;
        putc('\n', f);
    }
}

reader()
{
    write_section(banner);
    create_symbol_table();
    read_declarations();
    read_grammar();
    free_symbol_table();
    free_tags();
    pack_names();
    check_symbols();
    pack_symbols();
    pack_grammar();
    free_symbols();
    print_grammar();
}
```

49.2.2.9 Skeleton.c

```
#include "defs.h"

/* The banner used here should be replaced with an #ident directive  */
/* if the target C compiler supports #ident directives.              */
/*                                                                   */
/* If the skeleton is changed, the banner should be changed so that  */
/* the altered version can easily be distinguished from the original.*/

char *banner[] =
{
    "#ifndef lint",
    "static char yysccsid[] = \"@(#)yaccpar     1.7 (Berkeley)
09/09/90\";",
    "#endif",
    "#define YYBYACC 1",
    0
};

char *tables[] =
{
    "extern short yylhs[];",
    "extern short yylen[];",
```

```
    "extern short yydefred[];",
    "extern short yydgoto[];",
    "extern short yysindex[];",
    "extern short yyrindex[];",
    "extern short yygindex[];",
    "extern short yytable[];",
    "extern short yycheck[];",
    "#if YYDEBUG",
    "extern char *yyname[];",
    "extern char *yyrule[];",
    "#endif",
    0
};

char *header[] =
{
    "#define yyclearin (yychar=(-1))",
    "#define yyerrok (yyerrflag=0)",
    "#ifdef YYSTACKSIZE",
    "#ifndef YYMAXDEPTH",
    "#define YYMAXDEPTH YYSTACKSIZE",
    "#endif",
    "#else",
    "#ifdef YYMAXDEPTH",
    "#define YYSTACKSIZE YYMAXDEPTH",
    "#else",
    "#define YYSTACKSIZE 600",
    "#define YYMAXDEPTH 600",
    "#endif",
    "#endif",
    "int yydebug;",
    "int yynerrs;",
    "int yyerrflag;",
    "int yychar;",
    "short *yyssp;",
    "YYSTYPE *yyvsp;",
    "YYSTYPE yyval;",
    "YYSTYPE yylval;",
    "short yyss[YYSTACKSIZE];",
    "YYSTYPE yyvs[YYSTACKSIZE];",
    "#define yystacksize YYSTACKSIZE",
    0
};

char *body[] =
{
    "#define YYABORT goto yyabort",
```

```
    "#define YYACCEPT goto yyaccept",
    "#define YYERROR goto yyerrlab",
    "int",
    "yyparse()",
    "{",
    "    register int yym, yyn, yystate;",
    "#if YYDEBUG",
    "    register char *yys;",
    "    extern char *getenv();",
    "",
    "    if (yys = getenv(\"YYDEBUG\"))",
    "    {",
    "        yyn = *yys;",
    "        if (yyn >= '0' && yyn <= '9')",
    "            yydebug = yyn - '0';",
    "    }",
    "#endif",
    "",
    "    yynerrs = 0;",
    "    yyerrflag = 0;",
    "    yychar = (-1);",
    "",
    "    yyssp = yyss;",
    "    yyvsp = yyvs;",
    "    *yyssp = yystate = 0;",
    "",
    "yyloop:",
    "    if (yyn = yydefred[yystate]) goto yyreduce;",
    "    if (yychar < 0)",
    "    {",
    "        if ((yychar = yylex()) < 0) yychar = 0;",
    "#if YYDEBUG",
    "        if (yydebug)",
    "        {",
    "            yys = 0;",
    "            if (yychar <= YYMAXTOKEN) yys = yyname[yychar];",
    "            if (!yys) yys = \"illegal-symbol\";",
    "            printf(\"yydebug: state %d, reading %d (%s)\\n\",
yystate,",
    "                    yychar, yys);",
    "        }",
    "#endif",
    "    }",
    "    if ((yyn = yysindex[yystate]) && (yyn += yychar) >= 0 &&",
    "            yyn <= YYTABLESIZE && yycheck[yyn] == yychar)",
    "    {",
    "#if YYDEBUG",
```

```
    "        if (yydebug)",
    "            printf(\"yydebug: state %d, shifting to state
%d\\n\",",
    "                    yystate, yytable[yyn]);",
    "#endif",
    "        if (yyssp >= yyss + yystacksize - 1)",
    "        {",
    "            goto yyoverflow;",
    "        }",
    "        *++yyssp = yystate = yytable[yyn];",
    "        *++yyvsp = yylval;",
    "        yychar = (-1);",
    "        if (yyerrflag > 0)  --yyerrflag;",
    "        goto yyloop;",
    "    }",
    "    if ((yyn = yyrindex[yystate]) && (yyn += yychar) >= 0 &&",
    "            yyn <= YYTABLESIZE && yycheck[yyn] == yychar)",
    "    {",
    "        yyn = yytable[yyn];",
    "        goto yyreduce;",
    "    }",
    "    if (yyerrflag) goto yyinrecovery;",
    "#ifdef lint",
    "    goto yynewerror;",
    "#endif",
    "yynewerror:",
    "    yyerror(\"syntax error\");",
    "#ifdef lint",
    "    goto yyerrlab;",
    "#endif",
    "yyerrlab:",
    "    ++yynerrs;",
    "yyinrecovery:",
    "    if (yyerrflag < 3)",
    "    {",
    "        yyerrflag = 3;",
    "        for (;;)",
    "        {",
    "            if ((yyn = yysindex[*yyssp]) && (yyn += YYERRCODE) >= 0
&&",
    "                    yyn <= YYTABLESIZE && yycheck[yyn] ==
YYERRCODE)",
    "            {",
    "#if YYDEBUG",
    "                if (yydebug)",
    "                    printf(\"yydebug: state %d, error recovery
shifting\\",
```

```
    " to state %d\\n\", *yyssp, yytable[yyn]);",
    "#endif",
    "                if (yyssp >= yyss + yystacksize - 1)",
    "                {",
    "                    goto yyoverflow;",
    "                }",
    "                *++yyssp = yystate = yytable[yyn];",
    "                *++yyvsp = yylval;",
    "                goto yyloop;",
    "            }",
    "            else",
    "            {",
    "#if YYDEBUG",
    "                if (yydebug)",
    "                    printf(\"yydebug: error recovery discarding
state %d\
\\n\",",
    "                            *yyssp);",
    "#endif",
    "                if (yyssp <= yyss) goto yyabort;",
    "                --yyssp;",
    "                --yyvsp;",
    "            }",
    "        }",
    "    }",
    "    else",
    "    {",
    "        if (yychar == 0) goto yyabort;",
    "#if YYDEBUG",
    "        if (yydebug)",
    "        {",
    "            yys = 0;",
    "            if (yychar <= YYMAXTOKEN) yys = yyname[yychar];",
    "            if (!yys) yys = \"illegal-symbol\";",
    "            printf(\"yydebug: state %d, error recovery discards
token %d\
 (%s)\\n\",",
    "                    yystate, yychar, yys);",
    "        }",
    "#endif",
    "        yychar = (-1);",
    "        goto yyloop;",
    "    }",
    "yyreduce:",
    "#if YYDEBUG",
    "    if (yydebug)",
```

```
    "        printf(\"yydebug: state %d, reducing by rule %d
(%s)\\n\",",
    "                yystate, yyn, yyrule[yyn]);",
    "#endif",
    "    yym = yylen[yyn];",
    "    yyval = yyvsp[1-yym];",
    "    switch (yyn)",
    "    {",
    0
};


char *trailer[] =
{
    "    }",
    "    yyssp -= yym;",
    "    yystate = *yyssp;",
    "    yyvsp -= yym;",
    "    yym = yylhs[yyn];",
    "    if (yystate == 0 && yym == 0)",
    "    {",
    "#if YYDEBUG",
    "        if (yydebug)",
    "            printf(\"yydebug: after reduction, shifting from state
0 to\\",
    " state %d\\n\", YYFINAL);",
    "#endif",
    "        yystate = YYFINAL;",
    "        *++yyssp = YYFINAL;",
    "        *++yyvsp = yyval;",
    "        if (yychar < 0)",
    "        {",
    "            if ((yychar = yylex()) < 0) yychar = 0;",
    "#if YYDEBUG",
    "            if (yydebug)",
    "            {",
    "                yys = 0;",
    "                if (yychar <= YYMAXTOKEN) yys = yyname[yychar];",
    "                if (!yys) yys = \"illegal-symbol\";",
    "                printf(\"yydebug: state %d, reading %d (%s)\\n\",",
    "                        YYFINAL, yychar, yys);",
    "            }",
    "#endif",
    "        }",
    "        if (yychar == 0) goto yyaccept;",
    "        goto yyloop;",
    "    }",
```

```
    "    if ((yyn = yygindex[yym]) && (yyn += yystate) >= 0 &&",
    "            yyn <= YYTABLESIZE && yycheck[yyn] == yystate)",
    "        yystate = yytable[yyn];",
    "    else",
    "        yystate = yydgoto[yym];",
    "#if YYDEBUG",
    "    if (yydebug)",
    "        printf(\"yydebug: after reduction, shifting from state %d
\\",
    "to state %d\\n\", *yyssp, yystate);",
    "#endif",
    "    if (yyssp >= yyss + yystacksize - 1)",
    "    {",
    "        goto yyoverflow;",
    "    }",
    "    *++yyssp = yystate;",
    "    *++yyvsp = yyval;",
    "    goto yyloop;",
    "yyoverflow:",
    "    yyerror(\"yacc stack overflow\");",
    "yyabort:",
    "    return (1);",
    "yyaccept:",
    "    return (0);",
    "}",
    0
};

write_section(section)
char *section[];
{
    register int i;
    register FILE *fp;

    fp = code_file;
    for (i = 0; section[i]; ++i)
    {
        ++outline;
        fprintf(fp, "%s\n", section[i]);
    }
}
```

49.2.2.10 Symtab.c

```
#include "defs.h"
/* TABLE_SIZE is the number of entries in the symbol table. */
/* TABLE_SIZE must be a power of two.                        */
```

```
#define     TABLE_SIZE 1024

bucket **symbol_table;
bucket *first_symbol;
bucket *last_symbol;

int
hash(name)
char *name;
{
    register char *s;
    register int c, k;

    assert(name && *name);
    s = name;
    k = *s;
    while (c = *++s)
      k = (31*k + c) & (TABLE_SIZE - 1);

    return (k);
}

bucket *
make_bucket(name)
char *name;
{
    register bucket *bp;

    assert(name);
    bp = (bucket *) MALLOC(sizeof(bucket));
    if (bp == 0) no_space();
    bp->link = 0;
    bp->next = 0;
    bp->name = MALLOC(strlen(name) + 1);
    if (bp->name == 0) no_space();
    bp->tag = 0;
    bp->value = UNDEFINED;
    bp->index = 0;
    bp->prec = 0;
    bp-> class = UNKNOWN;
    bp->assoc = TOKEN;

    if (bp->name == 0) no_space();
    strcpy(bp->name, name);

    return (bp);
}
```

```
bucket *
lookup(name)
char *name;
{
    register bucket *bp, **bpp;

    bpp = symbol_table + hash(name);
    bp = *bpp;

    while (bp)
    {
      if (strcmp(name, bp->name) == 0) return (bp);
      bpp = &bp->link;
      bp = *bpp;
    }

    *bpp = bp = make_bucket(name);
    last_symbol->next = bp;
    last_symbol = bp;

    return (bp);
}

create_symbol_table()
{
    register int i;
    register bucket *bp;

    symbol_table = (bucket **) MALLOC(TABLE_SIZE*sizeof(bucket *));
    if (symbol_table == 0) no_space();
    for (i = 0; i < TABLE_SIZE; i++)
      symbol_table[i] = 0;

    bp = make_bucket("error");
    bp->index = 1;
    bp->class = TERM;

    first_symbol = bp;
    last_symbol = bp;
    symbol_table[hash("error")] = bp;
}

free_symbol_table()
{
    FREE(symbol_table);
    symbol_table = 0;
}
```

```
free_symbols()
{
    register bucket *p, *q;

    for (p = first_symbol; p; p = q)
    {
      q = p->next;
      FREE(p);
    }
}
```

49.2.2.11 Verbose.c

```
#include "defs.h"

static short *null_rules;

verbose()
{
    register int i;

    if (!vflag) return;

    null_rules = (short *) MALLOC(nrules*sizeof(short));
    if (null_rules == 0) no_space();
    fprintf(verbose_file, "\f\n");
    for (i = 0; i < nstates; i++)
      print_state(i);
    FREE(null_rules);

    if (nunused)
      log_unused();
    if (SRtotal || RRtotal)
      log_conflicts();

    fprintf(verbose_file, "\n\n%d terminals, %d nonterminals\n",
ntokens,
          nvars);
    fprintf(verbose_file, "%d grammar rules, %d states\n", nrules - 2,
nstates);
}

log_unused()
{
    register int i;
    register short *p;
```

```
    fprintf(verbose_file, "\n\nRules never reduced:\n");
    for (i = 3; i < nrules; ++i)
    {
      if (!rules_used[i])
      {
          fprintf(verbose_file, "\t%s :", symbol_name[rlhs[i]]);
          for (p = ritem + rrhs[i]; *p >= 0; ++p)
            fprintf(verbose_file, " %s", symbol_name[*p]);
          fprintf(verbose_file, "  (%d)\n", i - 2);
      }
    }
}

log_conflicts()
{
    register int i;

    fprintf(verbose_file, "\n\n");
    for (i = 0; i < nstates; i++)
    {
      if (SRconflicts[i] || RRconflicts[i])
      {
          fprintf(verbose_file, "State %d contains ", i);
          if (SRconflicts[i] == 1)
            fprintf(verbose_file, "1 shift/reduce conflict");
          else if (SRconflicts[i] > 1)
            fprintf(verbose_file, "%d shift/reduce conflicts",
                  SRconflicts[i]);
          if (SRconflicts[i] && RRconflicts[i])
            fprintf(verbose_file, ", ");
          if (RRconflicts[i] == 1)
            fprintf(verbose_file, "1 reduce/reduce conflict");
          else if (RRconflicts[i] > 1)
            fprintf(verbose_file, "%d reduce/reduce conflicts",
                  RRconflicts[i]);
          fprintf(verbose_file, ".\n");
      }
    }
}

print_state(state)
int state;
{
    if (state)
      fprintf(verbose_file, "\n\n");
    if (SRconflicts[state] || RRconflicts[state])
      print_conflicts(state);
```

```
    fprintf(verbose_file, "state %d\n", state);
    print_core(state);
    print_nulls(state);
    print_actions(state);
}

print_conflicts(state)
int state;
{
    register int symbol;
    register action *p, *q, *r;

    for (p = parser[state]; p; p = q->next)
    {
      q = p;
      if (p->action_code == ERROR || p->suppressed == 2)
          continue;

      symbol = p->symbol;
      while (q->next && q->next->symbol == symbol)
          q = q->next;
      if (state == final_state && symbol == 0)
      {
          r = p;
          for (;;)
          {
            fprintf(verbose_file, "%d: shift/reduce conflict \
(accept, reduce %d) on $end\n", state, r->number - 2);
            if (r == q) break;
            r = r->next;
          }
      }
      else if (p != q)
      {
          r = p->next;
          if (p->action_code == SHIFT)
          {
            for (;;)
            {
                if (r->action_code == REDUCE && p->suppressed != 2)
                  fprintf(verbose_file, "%d: shift/reduce conflict \
(shift %d, reduce %d) on %s\n", state, p->number, r->number - 2,
                        symbol_name[symbol]);
                if (r == q) break;
                r = r->next;
            }
          }
```

```
            else
            {
              for (;;)
              {
                  if (r->action_code == REDUCE && p->suppressed != 2)
                    fprintf(verbose_file, "%d: reduce/reduce conflict \
(reduce %d, reduce %d) on %s\n", state, p->number - 2, r->number - 2,
                          symbol_name[symbol]);
                  if (r == q) break;
                  r = r->next;
              }
            }
        }
      }
}

print_core(state)
int state;
{
    register int i;
    register int k;
    register int rule;
    register core *statep;
    register short *sp;
    register short *sp1;

    statep = state_table[state];
    k = statep->nitems;

    for (i = 0; i < k; i++)
    {
      sp1 = sp = ritem + statep->items[i];

      while (*sp >= 0) ++sp;
      rule = -(*sp);
      fprintf(verbose_file, "\t%s : ", symbol_name[rlhs[rule]]);

        for (sp = ritem + rrhs[rule]; sp < sp1; sp++)
          fprintf(verbose_file, "%s ", symbol_name[*sp]);

      putc('.', verbose_file);

      while (*sp >= 0)
      {
          fprintf(verbose_file, " %s", symbol_name[*sp]);
          sp++;
      }
```

```
        fprintf(verbose_file, "  (%d)\n", -2 - *sp);
      }
  }

  print_nulls(state)
  int state;
  {
      register action *p;
      register int i, j, k, nnulls;

      nnulls = 0;
      for (p = parser[state]; p; p = p->next)
      {
        if (p->action_code == REDUCE &&
              (p->suppressed == 0 || p->suppressed == 1))
        {
            i = p->number;
            if (rrhs[i] + 1 == rrhs[i+1])
            {
              for (j = 0; j < nnulls && i > null_rules[j]; ++j)
                  continue;

              if (j == nnulls)
              {
                  ++nnulls;
                  null_rules[j] = i;
              }
              else if (i != null_rules[j])
              {
                  ++nnulls;
                  for (k = nnulls - 1; k > j; --k)
                    null_rules[k] = null_rules[k-1];
                  null_rules[j] = i;
              }
            }
        }
      }

      for (i = 0; i < nnulls; ++i)
      {
        j = null_rules[i];
        fprintf(verbose_file, "\t%s : .  (%d)\n", symbol_name[rlhs[j]],
              j - 2);
      }
      fprintf(verbose_file, "\n");
  }
```

```
print_actions(stateno)
int stateno;
{
    register action *p;
    register shifts *sp;
    register int as;

    if (stateno == final_state)
      fprintf(verbose_file, "\t$end  accept\n");
    p = parser[stateno];
    if (p)
    {
      print_shifts(p);
      print_reductions(p, defred[stateno]);
    }

    sp = shift_table[stateno];
    if (sp && sp->nshifts > 0)
    {
      as = accessing_symbol[sp->shift[sp->nshifts - 1]];
      if (ISVAR(as))
          print_gotos(stateno);
    }
}

print_shifts(p)
register action *p;
{
    register int count;
    register action *q;

    count = 0;
    for (q = p; q; q = q->next)
    {
      if (q->suppressed < 2 && q->action_code == SHIFT)
          ++count;
    }
    if (count > 0)
    {
      for (; p; p = p->next)
      {
          if (p->action_code == SHIFT && p->suppressed == 0)
            fprintf(verbose_file, "\t%s  shift %d\n",
                      symbol_name[p->symbol], p->number);
      }
    }
}
```

```
print_reductions(p, defred)
register action *p;
register int defred;
{
    register int k, anyreds;
    register action *q;

    anyreds = 0;
    for (q = p; q ; q = q->next)
    {
      if (q->action_code == REDUCE && q->suppressed < 2)
      {
          anyreds = 1;
          break;
      }
    }
    if (anyreds == 0)
      fprintf(verbose_file, "\t.  error\n");
    else
    {
      for (; p; p = p->next)
      {
          if (p->action_code == REDUCE && p->number != defred)
          {
            k = p->number - 2;
            if (p->suppressed == 0)
                fprintf(verbose_file, "\t%s  reduce %d\n",
                      symbol_name[p->symbol], k);
          }
      }

        if (defred > 0)
          fprintf(verbose_file, "\t.  reduce %d\n", defred - 2);
    }
}

print_gotos(stateno)
int stateno;
{
    register int i, k;
    register int as;
    register short *to_state;
    register shifts *sp;

    putc('\n', verbose_file);
    sp = shift_table[stateno];
    to_state = sp->shift;
```

```
    for (i = 0; i < sp->nshifts; ++i)
    {
      k = to_state[i];
      as = accessing_symbol[k];
      if (ISVAR(as))
          fprintf(verbose_file, "\t%s  goto %d\n", symbol_name[as], k);
    }
}
```

49.2.2.12 Warshall.c

```
#include "defs.h"

transitive_closure(R, n)
unsigned *R;
int n;
{
    register int rowsize;
    register unsigned mask;
    register unsigned *rowj;
    register unsigned *rp;
    register unsigned *rend;
    register unsigned *ccol;
    register unsigned *relend;
    register unsigned *cword;
    register unsigned *rowi;

    rowsize = WORDSIZE(n);
    relend = R + n*rowsize;

    cword = R;
    mask = 1;
    rowi = R;
    while (rowi < relend)
    {
      ccol = cword;
      rowj = R;

      while (rowj < relend)
      {
          if (*ccol & mask)
          {
            rp = rowi;
            rend = rowj + rowsize;
            while (rowj < rend)
                *rowj++ |= *rp++;
          }
```

```
            else
            {
              rowj += rowsize;
            }

            ccol += rowsize;
        }

        mask <<= 1;
        if (mask == 0)
        {
            mask = 1;
            cword++;
        }

        rowi += rowsize;
      }
}

reflexive_transitive_closure(R, n)
unsigned *R;
int n;
{
    register int rowsize;
    register unsigned mask;
    register unsigned *rp;
    register unsigned *relend;

    transitive_closure(R, n);

    rowsize = WORDSIZE(n);
    relend = R + n*rowsize;

    mask = 1;
    rp = R;
    while (rp < relend)
    {
      *rp |= mask;
      mask <<= 1;
      if (mask == 0)
      {
          mask = 1;
          rp++;
      }
      rp += rowsize;
    }
}
```

49.2.2.13 Main.c

```
#include <signal.h>
#include "defs.h"

char dflag;
char lflag;
char rflag;
char tflag;
char vflag;

char *file_prefix = "y";
char *myname = "yacc";
#ifdef MSDOS
char *temp_form = "yaccXXXXXXX";
#else
char *temp_form = "yacc.XXXXXXX";
#endif

int lineno;
int outline;

char *action_file_name;
char *defines_file_name;
char *input_file_name = "";
char *output_file_name;
char *code_file_name;
char *text_file_name;
char *union_file_name;
char *verbose_file_name;

FILE *action_file;   /* a temp file, used to save actions associated  */
                     /*  with rules until the parser is written       */
FILE *defines_file;  /*  y.tab.h                                       */
FILE *input_file;    /*  the input file                                */
FILE *output_file;   /*  y.tab.c                                       */
FILE *code_file;   /* y.code.c (used when the -r option is specified) */
FILE *text_file;     /*  a temp file, used to save text until all     */
                     /*  symbols have been defined                     */
FILE *union_file;    /*  a temp file, used to save the union          */
                     /*  definition until all symbol have been        */
                     /*  defined                                       */
FILE *verbose_file;  /*  y.output                                      */

int nitems;
int nrules;
int nsyms;
```

```
int ntokens;
int nvars;

int   start_symbol;
char  **symbol_name;
short *symbol_value;
short *symbol_prec;
char  *symbol_assoc;

short *ritem;
short *rlhs;
short *rrhs;
short *rprec;
char  *rassoc;
short **derives;
char *nullable;

extern char *mktemp();
extern char *getenv();

done(k)
int k;
{
    if (action_file) { fclose(action_file); unlink(action_file_name); }
    if (text_file) { fclose(text_file); unlink(text_file_name); }
    if (union_file) { fclose(union_file); unlink(union_file_name); }
    exit(k);
}

void onintr()  /* last revision deletes the "void" */
{
    done(1);
}

set_signals()
{
#ifdef SIGINT
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);
#endif
#ifdef SIGTERM
    if (signal(SIGTERM, SIG_IGN) != SIG_IGN)
        signal(SIGTERM, onintr);
#endif
#ifdef SIGHUP
    if (signal(SIGHUP, SIG_IGN) != SIG_IGN)
        signal(SIGHUP, onintr);
```

```
#endif
}

usage()
{
    fprintf(stderr, "Yacc (Berkeley) 09/09/90\n");
    fprintf(stderr, "Usage: %s [-dlrtv] [-b file_prefix] filename\n\n",
myname);
    fprintf(stderr, "\t-b file_prefix  change the default file prefix
\"y.\"\n");
    fprintf(stderr, "\t-d\t\twrite the header file \"y.tab.h\"\n");
    fprintf(stderr, "\t-l\t\texclude the #line directives in files\n");
    fprintf(stderr, "\t-r\t\tseperate code and tables into \"y.code.c\"
and \"y.tab.c\"\n");
    fprintf(stderr, "\t-t\t\tinclude the debugging code in files\n");

fprintf(stderr, "\t-v\t\twrite the parser description file
\"y.output\"\n");
    exit(1);
}

getargs(argc, argv)
int argc;
char *argv[];
{
    register int i;
    register char *s;

    if (argc > 0) myname = argv[0];
    for (i = 1; i < argc; ++i)
    {
        s = argv[i];
        if (*s != '-') break;
        switch (*++s)
        {
        case '\0':
            input_file = stdin;
            if (i + 1 < argc) usage();
            return;

        case '-':
            ++i;
            goto no_more_options;

        case 'b':
            if (*++s)
                 file_prefix = s;
```

```
    else if (++i < argc)
        file_prefix = argv[i];
    else
        usage();
    continue;

case 'd':
    dflag = 1;
    break;

case 'l':
    lflag = 1;
    break;

case 'r':
rflag = 1;
break;

case 't':
    tflag = 1;
    break;

case 'v':
    vflag = 1;
    break;

default:
    usage();
}

for (;;)
{
    switch (*++s)
    {
    case '\0':
        goto end_of_option;

    case 'd':
        dflag = 1;
        break;

    case 'l':
        lflag = 1;
        break;

case 'r':
rflag = 1;
```

```
        break;

            case 't':
                tflag = 1;
                break;

            case 'v':
                vflag = 1;
                break;

            default:
                usage();
            }
        }
end_of_option:;
    }

no_more_options:;
    if (i + 1 != argc) usage();
    input_file_name = argv[i];
}

char *
allocate(n)
unsigned n;
{
    register char *p;

    p = NULL;
    if (n)
    {
        p = CALLOC(1, n);
        if (!p) no_space();
    }
    return (p);
}

create_file_names()
{
    int i, len;
    char *tmpdir;

#ifdef MSDOS
    (tmpdir = getenv("TMPDIR")) ||
       (tmpdir = getenv("TMP")) ||
       (tmpdir = ".");
#else
```

```
    tmpdir = getenv("TMPDIR");
    if (tmpdir == 0) tmpdir = "/tmp";
#endif

    len = strlen(tmpdir);
    i = len + 13;
    if (len && tmpdir[len-1] != '/')
        ++i;

    action_file_name = MALLOC(i);
    if (action_file_name == 0) no_space();
    text_file_name = MALLOC(i);
    if (text_file_name == 0) no_space();
    union_file_name = MALLOC(i);
    if (union_file_name == 0) no_space();

    strcpy(action_file_name, tmpdir);
    strcpy(text_file_name, tmpdir);
    strcpy(union_file_name, tmpdir);

    if (len && tmpdir[len - 1] != '/')
    {
        action_file_name[len] = '/';
        text_file_name[len] = '/';
        union_file_name[len] = '/';
        ++len;
    }

    strcpy(action_file_name + len, temp_form);
    strcpy(text_file_name + len, temp_form);
    strcpy(union_file_name + len, temp_form);

    action_file_name[len + 5] = 'a';
    text_file_name[len + 5] = 't';
    union_file_name[len + 5] = 'u';

    mktemp(action_file_name);
    mktemp(text_file_name);
    mktemp(union_file_name);

    len = strlen(file_prefix);

     output_file_name = MALLOC(len + 7);
     if (output_file_name == 0)
        no_space();
     strcpy(output_file_name, file_prefix);
     strcpy(output_file_name + len, OUTPUT_SUFFIX);
```

```
     if (rflag)
     {
        code_file_name = MALLOC(len + 8);
        if (code_file_name == 0)
            no_space();
        strcpy(code_file_name, file_prefix);
        strcpy(code_file_name + len, CODE_SUFFIX);
     }
     else
        code_file_name = output_file_name;

    if (dflag)
    {
        /*  the number 7 below is the size of ".tab.h"; sizeof is not
used  */
        /* because of a C compiler that thinks sizeof(".tab.h") == 6  */
        defines_file_name = MALLOC(len + 7);
        if (defines_file_name == 0)
                no_space();
        strcpy(defines_file_name, file_prefix);
        strcpy(defines_file_name + len, DEFINES_SUFFIX);
    }

    if (vflag)
    {
        verbose_file_name = MALLOC(len + 8);
        if (verbose_file_name == 0)
                no_space();
        strcpy(verbose_file_name, file_prefix);
        strcpy(verbose_file_name + len, VERBOSE_SUFFIX);
    }
}

open_files()
{
    create_file_names();

    if (input_file == 0)
    {
        input_file = fopen(input_file_name, "r");
        if (input_file == 0)
                open_error(input_file_name);
    }

    action_file = fopen(action_file_name, "w");
    if (action_file == 0) open_error(action_file_name);
```

```
    text_file = fopen(text_file_name, "w");
    if (text_file == 0) open_error(text_file_name);

    if (vflag)
    {
        verbose_file = fopen(verbose_file_name, "w");
        if (verbose_file == 0) open_error(verbose_file_name);
    }
    if (dflag)
    {
        defines_file = fopen(defines_file_name, "w");
        if (defines_file == 0) open_error(defines_file_name);
        union_file = fopen(union_file_name, "w");
        if (union_file ==  0) open_error(union_file_name);
    }

    output_file = fopen(output_file_name, "w");
    if (output_file == 0) open_error(output_file_name);

     if (rflag)
     {
        code_file = fopen(code_file_name, "w");
        if (code_file == 0)
            open_error(code_file_name);
     }
     else
        code_file = output_file;
}

int
main(argc, argv)
int argc;
char *argv[];
{
    set_signals();
    getargs(argc, argv);
    open_files();
    reader();
    lr0();
    lalr();
    make_parser();
    verbose();
    output();
    done(0);
    /*NOTREACHED*/
}
```

49.2.3 Compiling BYACC

In order to compile all the above files create a project file called `Byacc.prj` and add all the above files to it. Then make EXE file for that project file. Now you get a YACC for DOS. Use it with your own set of grammar.