SPIN VERSION 0 SIMULATOR SOURCE D

The *makefile* for the final version of SPIN, discussed in Chapter 12, is defined as follows.

```
CC=cc           # ANSI C compiler
CFLAGS=-O       # optimizer
YFLAGS=-v -d -D # create y.output, y.debug, and y.tab.h
OFILES= spin.o lex.o sym.o vars.o main.o debug.o \
        mesg.o flow.o sched.o run.o dummy.o

spin:   $(OFILES)
        $(CC) $(CFLAGS) -o spin $(OFILES) -lm

%.o:    %.c spin.h
        $(CC) $(CFLAGS) -c $%.c
```

The remainder of this Appendix lists the contents of the 11 source files (see Table D.1) that are required to compile the program, plus one dummy file as temporary place holder for the analysis routines that are added in Chapter 13 and listed in Appendix E. The program should run on any UNIX system with an ANSI-standard compatible C compiler.

See the introduction to Appendix E for information on retrieving an online copy of the most recent version of SPIN.

Table D.1 – Source File Index

File	Line Number
dummy.c	1907
flow.c	1137
lex.l	194
main.c	487
mesg.c	867
run.c	1378
sched.c	1553
spin.h	1
spin.y	283
sym.c	626
vars.c	717

Table D.2 – Procedures Listed – Appendix D

Procedure	Line	Procedure	Line
a_rcv(q, n, full)	1001	a_snd(q, n)	969
add_el(e, s)	1232	add_seq(n)	1256
addsymbol(r, s)	1719	break_dest()	1348
cast_val(t, v)	791	check_name(s)	268
checkvar(s, n)	745	close_seq()	1158
cnt_mpars(n)	882	complete_rendez()	1688
doq(s, n)	1111	dumpglobals()	815
dumplocal(s)	847	emalloc(n)	582
enable(s, n)	1602	eval(now)	1455
eval_sub(e)	1385	eval_sync(now)	1435
fatal(s1, s2)	573	find_lab(s, c)	1322
findloc(s, n)	1801	gensrc()	1909
get_lab(s)	1287	getglobal(s, n)	769
getlocal(s, n)	1821	getval(s, n)	725
has_lab(e)	1298	hash(s)	634
if_seq(s, tok, lnno)	1209	interprint(n)	1511
ismtype(str)	703	lookup(s)	649
main(argc, argv)	502	make_atomic(s)	1355
match_trail()	1914	mov_lab(z, e, y)	1309
naddsymbol(r, s, k)	1741	new_el(n)	1195
nn(s, v, t, l, r)	592	open_seq(top)	1148
p_talk(e)	1860	pushbreak()	1335
q_is_sync(n)	935	qlen(n)	925
qmake(s)	893	qrecv(n, full)	959
qsend(n)	945	ready(n, p, s)	1589
rem_lab(a, b, c)	607	rem_var(a, b, c)	617
remotelab(n)	1868	remotevar(n)	1880
runnable(s, n)	1575	s_snd(q, n)	1039
sched()	1631	seqlist(s, r)	1184
set_lab(s, e)	1272	setglobal(v, m)	804
setlocal(p, m)	1831	setmtype(m)	686
setparams(r, p, q)	1778	settype(n, t)	670
setval(v, n)	735	sr_mesg(v, j)	1098
sr_talk(n, v, s, a, j, mx, named)	1073	start_claim(n)	1617
talk(e, s)	1848	typck(n, t, s)	1761
typex(n, t)	783	walk_atomic(a, b)	1364
whoruns()	1842	wrapup()	1674
yyerror(s1, s2)	561		

Table D.3 – Procedures Explained – Chapter 12

Procedure	Page	Procedure	Page
a_rcv()	275	a_snd()	274
add_el()	280	add_seq()	280
addsymbol()	289	break_dest()	283
cast_val()	268	check_name()	260
check_name()	269	check_name()	286
checkvar()	267	close_seq()	279
complete_rendez()	276	emalloc()	262
enable()	288	eval()	254
eval()	267	eval()	272
eval()	289	eval()	291
eval_sub()	281	eval_sub()	290
eval_sync()	277	findloc()	291
get_lab()	281	getglobal()	268
getlocal()	291	hash()	263
if_seq()	284	interprint()	271
ismtype()	286	lookup()	262
lookup()	263	main()	293
new_el()	279	nn()	253
nn()	265	open_seq()	279
pushbreak()	282	qlen()	273
qmake()	268	qmake()	273
qrecv()	274	qsend()	274
ready()	287	rem_lab()	292
rem_var()	292	runnable()	287
s_snd()	275	sched()	289
seqlist()	279	set_lab()	281
setglobal()	267	setlocal()	291
setmtype()	286	setparams()	288
setparams()	288		

```
 1 /***** spin: spin.h *****/
 2
 3 typedef struct Symbol {
 4     char    *name;
 5     short   type;           /* variable or chan type      */
 6     int     nel;            /* 1 if scalar, >1 if array   */
 7     int     *val;           /* runtime value(s), initl 0  */
 8     struct Node     *ini;   /* initial value, or chan-def */
 9     struct Symbol   *context; /* 0 if global, or procname */
10     struct Symbol   *next;  /* linked list */
11 } Symbol;
12
13 typedef struct Node {
14     int     nval;           /* value attribute            */
15     short   ntyp;           /* node type                  */
16     Symbol  *nsym;          /* new attribute              */
17     Symbol  *fname;         /* filename of src            */
18     struct SeqList  *seql;  /* list of sequences          */
19     struct Node     *lft, *rgt; /* children in parse tree */
20 } Node;
21
22 typedef struct Queue {
23     short   qid;            /* runtime q index      */
24     short   qlen;           /* nr messages stored   */
25     short   nslots, nflds;  /* capacity, flds/slot  */
26     short   *fld_width;     /* type of each field   */
27     int     *contents;      /* the actual buffer    */
28     struct Queue    *nxt;   /* linked list */
29 } Queue;
30
31 typedef struct Element {
32     Node    *n;             /* defines the type & contents */
33     int     seqno;          /* uniquely identifies this el */
34     unsigned char   status; /* used by analyzer generator  */
35     struct SeqList  *sub;   /* subsequences, for compounds */
36     struct Element  *nxt;   /* linked list */
37 } Element;
38
39 typedef struct Sequence {
40     Element *frst;
41     Element *last;
42 } Sequence;
43
44 typedef struct SeqList {
45     Sequence        *this;  /* one sequence */
46     struct SeqList  *nxt;   /* linked list  */
47 } SeqList;
48
49 typedef struct Label {
50     Symbol  *s;
51     Symbol  *c;
52     Element *e;
53     struct Label    *nxt;
```

```
 54 } Label;
 55
 56 typedef struct Lbreak {
 57     Symbol  *l;
 58     struct Lbreak   *nxt;
 59 } Lbreak;
 60
 61 typedef struct RunList {
 62     Symbol  *n;             /* name            */
 63     int     pid;            /* process id      */
 64     int     maxseq;         /* used by analyzer generator */
 65     Element *pc;            /* current stmnt   */
 66     Symbol  *symtab;        /* local variables */
 67     struct RunList  *nxt;   /* linked list */
 68 } RunList;
 69
 70 typedef struct ProcList {
 71     Symbol  *n;             /* name       */
 72     Node    *p;             /* parameters */
 73     Sequence *s;            /* body       */
 74     struct ProcList *nxt;   /* linked list */
 75 } ProcList;
 76
 77 #ifdef GODEF
 78 typedef struct atom_stack { /* all conflict sets hit in atomic seq. */
 79     char *what;             /* what = "R_LOCK", "W_LOCK"   */
 80     int when;               /* when == Direct or Indirect  */
 81     int cause;              /* type of operation, e.g. 'r' */
 82     Node *n;
 83     struct atom_stack *nxt;
 84 } atom_stack;
 85
 86 #define DONE          1             /* status bits of elements */
 87 #define ATOM          2             /* part of an atomic chain */
 88 #define L_ATOM        4             /* last element in a chain */
 89 #ifdef VARSTACK
 90 #define HIT   8             /* hit in dflow.c search   */
 91 #define Nhash       255             /* size of hash table */
 92
 93 #define PREDEF        5             /* predefined identifier */
 94 #define BIT   1             /* data types    */
 95 #define BYTE          8             /* width in bits */
 96 #define SHORT        16
 97 #define INT  32
 98 #define     CHAN     64
 99
100 #ifdef GODEF
101 #define Direct              1
102 #define Indirect    2
103
104 #define max(a,b) (((a)<(b)) ? (b) : (a))
105
106 /***** Old-Style C - prototype definitions *****/
107 extern char *malloc();
```

```
108 extern char *memcpy();
109 extern char *memset();
110 extern char *mktemp();
111 extern char *strcat();
112 extern char *strcpy();
113 extern long time();
114 extern void exit();
115 extern void srand();
116
117 extern Element      *d_eval_sub();
118 extern Element      *colons();
119 extern Element      *eval_sub();
120 extern Element      *get_lab();
121 extern Element      *huntele();
122 extern Element      *if_seq();
123 extern Element      *new_el();
124 extern Element      *walk_sub();
125 extern Node *nn();
126 extern Node *rem_var();
127 extern Node *rem_lab();
128 extern SeqList      *seqlist();
129 extern Sequence     *close_seq();
130 extern Symbol       *break_dest();
131 extern Symbol       *findloc();
132 extern Symbol       *has_lab();
133 extern Symbol       *lookup();
134 extern char *emalloc();
135 extern void add_el();
136 extern void add_seq();
137 extern void addsymbol();
138 #ifdef DEBUG
139 extern void auto2();
140 extern void auto_atomic();
141 extern void check_proc();
142 extern void cnt_mpars();
143 extern void comment();
144 extern void do_var();
145 extern void doglobal();
146 extern void do_init();
147 extern void dolocal();
148 extern void doq();
149 extern void dumpglobals();
150 extern void dumplocal();
151 extern void dumpskip();
152 extern void dumpsrc();
153 extern void end_labs();
154 extern void explain();
155 extern void fatal();
156 extern void genaddproc();
157 extern void genaddqueue();
158 extern void genaddclaim();
159 extern void genheader();
160 extern void genother();
161 extern void gensrc();
```

```
162 extern void genunio();
163 extern void lost_trail();
164 extern void main();
165 extern void make_atomic();
166 extern void match_trail();
167 extern void mov_lab();
168 extern void naddsymbol();
169 extern void ncases();
170 extern void ntimes();
171 extern void open_seq();
172 extern void p_talk();
173 extern void patch_atomic();
174 extern void pushbreak();
175 extern void put_pinit();
176 extern void put_ptype();
177 extern void putname();
178 extern void putnr();
179 extern void putremote();
180 extern void putproc();
181 extern void putprogress();
182 extern void putseq();
183 extern void putseq_el();
184 extern void putseq_lst();
185 extern void putskip();
186 extern void putsrc();
187 extern void putstmnt();
188 extern void ready();
189 extern void runnable();
190 extern void sched();
191 extern void set_lab();
192 extern void setmtype();
193 extern void setparams();
194 extern void settype();
195 extern void sr_mesg();
196 extern void sr_talk();
197 extern void start_claim();
198 extern void talk();
199 extern void typ2c();
200 extern void typex();
201 extern void undostmnt();
202 extern void walk_atomic();
203 extern void whoruns();
204
205 /***** spin: lex.l *****/
206
207 %{
208 #include "spin.h"
209 #include "y.tab.h"
210
211 int         lineno=1;
212 unsigned char       in_comment=0;
213 extern Symbol       *Fname;
214
215 #define Token       if (!in_comment) return
```

```
%}

%%
"/*"        { in_comment=1; }
"*/"        { in_comment=0; }
\n          { lineno++; }
[ \t]       { /* ignore white space */ }
[0-9]+      { yylval.val = atoi(yytext); Token CONST; }
\#\ [0-9]+\ \"[^\"]*\"[ 0-9]* {     /* preprocessor directive */
            int i=1;
            while (yytext[i] == ' ') i++;
            lineno = atoi(&yytext[i])-1;
            while (yytext[i] != ' ') i++;
            Fname = lookup(&yytext[i+1]);
    }
\"[^\"]*\"  { yylval.sym = lookup(yytext); Token STRING; }
"never" { yylval.sym = lookup(":never:"); Token CLAIM; }
"init"      { yylval.sym = lookup("_init"); Token INIT; }
"int"       { yylval.val =   INT; Token TYPE; }
"short"     { yylval.val = SHORT; Token TYPE; }
"byte"      { yylval.val =  BYTE; Token TYPE; }
"bool"      { yylval.val =   BIT; Token TYPE; }
"bit"       { yylval.val =   BIT; Token TYPE; }
"chan"      { yylval.val =  CHAN; Token TYPE; }
"skip"      { yylval.val =  1; Token CONST; }
[a-zA-Z_][a-zA-Z_0-9]* { Token check_name(yytext); }
"::"        { yylval.val = lineno; Token SEP; }
"="         { yylval.val = lineno; Token ASGN; }
"!"         { yylval.val = lineno; Token SND; }
"?"         { yylval.val = lineno; Token RCV; }
"->"        { Token ';'; /* statement separator */ }
"<<"        { Token LSHIFT; /* shift bits left  */ }
">>"        { Token RSHIFT; /* shift bits right */ }
"<="        { Token     LE; /* less than or equal to */ }
">="        { Token     GE; /* greater than or equal to */ }
"=="        { Token     EQ; /* equal to */ }
"!="        { Token     NE; /* not equal to */ }
"&&"        { Token    AND; /* logical and */ }
"||"        { Token     OR; /* logical or */ }
.           { Token yytext[0]; }
%%

static struct {
    char *s;        int tok;
} Names[] = {
    "Symmetry",     SYMMETRY,
    "assert",       ASSERT,
    "atomic",       ATOMIC,
    "break",        BREAK,
    "do",           DO,
    "fi",           FI,
    "goto",         GOTO,
    "if",           IF,
    "len",          LEN,
```

```
270     "mtype",        MTYPE,
271     "od",           OD,
272     "of",           OF,
273     "printf",       PRINT,
274     "proctype",     PROCTYPE,
275     "run",          RUN,
276     "timeout",      TIMEOUT,
277     0,              0,
278 };
279
280 check_name(s)
281     char *s;
282 {
283     register int i;
284     for (i = 0; Names[i].s; i++)
285             if (strcmp(s, Names[i].s) == 0)
286             {       yylval.val = lineno;
287                     return Names[i].tok;
288             }
289     if (yylval.val = ismtype(s))
290             return CONST;
291     yylval.sym = lookup(s); /* symbol table */
292     return NAME;
293 }
294
295 /***** spin: spin.y *****/
296
297 %{
298 #include "spin.h"
299 #define YYDEBUG     0
300 #define Stop        nn(0,lineno,'@',0,0)
301 extern Symbol       *context;
302 extern int  lineno, u_sync, u_async;
303 char                *claimproc = (char *) 0;
304 %}
305
306 %union{
307         int      val;
308         Node     *node;
309         Symbol   *sym;
310         Sequence *seq;
311         SeqList  *seql;
312 }
313
314 %token      <val>   RUN LEN OF
315 %token      <val>   CONST TYPE ASGN
316 %token      <sym>   NAME CLAIM
317 %token      <sym>   STRING INIT
318 %token      <val>   ASSERT SYMMETRY
319 %token      <val>   GOTO BREAK MTYPE SEP
320 %token      <val>   IF FI DO OD ATOMIC
321 %token      <val>   SND RCV PRINT TIMEOUT
322 %token      <val>   PROCTYPE
323
```

```
%type       <sym>   var ivar
%type       <node>  expr var_list stmnt
%type       <node>  args arg arglist typ_list decl
%type       <node>  decl_lst one_decl any_decl
%type       <node>  prargs margs varref step ch_init
%type       <seql>  options
%type       <seq>   option body

%right              ASGN
%left               SND RCV
%left               OR
%left               AND
%left               '|'
%left               '&'
%left               EQ NE
%left               '>' '<' GE LE
%left               LSHIFT RSHIFT
%left               '+' '-'
%left               '*' '/' '%'
%right              '~' UMIN NEG
%%

/** PROMELA Grammar Rules **/

program : units             { sched(); }
        ;
units   : unit | units unit
        ;
unit    : proc
        | claim
        | init
        | one_decl
        | mtype
        ;
proc    : PROCTYPE NAME     { context = $2; }
            '(' decl ')'
          body              { ready($2, $5, $7);
                              context = (Symbol *) 0;
                            }
        ;
claim   : CLAIM             { context = $1;
                              if (claimproc)
                                yyerror("claim %s redefined",
                                            claimproc);
                              claimproc = $1->name;
                            }
          body              { ready($1, (Node *) 0, $3);
                              context = (Symbol *) 0;
                            }
        ;
init    : INIT              { context = $1; }
          body              { runnable($3, $1);
                              context = (Symbol *) 0;
                            }
```

```
378         ;
379 mtype   : MTYPE ASGN '{' args '}'   { setmtype($4); }
380     | SYMMETRY ASGN arglist         { syms($3); }
381         | ';'       /* optional ; as separator of units */
382         ;
383 arglist     : '{' arg '}'                   { $$ = $2; }
384     | '{' arg '}' ',' arglist       { $$ = nn(0, 0, ';', $2, $5); }
385     ;
386 body    : '{'               { open_seq(1); }
387             sequence        { add_seq(Stop); }
388           '}'               { $$ = close_seq();
389                             }
390         ;
391 sequence: step              { add_seq($1); }
392         | sequence ';' step { add_seq($3); }
393         ;
394 step    : any_decl stmnt    { $$ = $2; }
395         ;
396 any_decl: /* empty */       { $$ = (Node *) 0; }
397         | one_decl ';' any_decl { $$ = nn(0, 0, ',', $1, $3); }
398         ;
399 one_decl: TYPE var_list     { settype($2, $1); $$ = $2; }
400         ;
401 decl_lst: one_decl          { $$ = nn(0, 0, ',', $1,  0); }
402         | one_decl ';' decl_lst { $$ = nn(0, 0, ',', $1, $3); }
403         ;
404 decl    : /* empty */       { $$ = (Node *) 0; }
405         | decl_lst          { $$ = $1; }
406         ;
407 var_list: ivar              { $$ = nn($1, 0, TYPE,  0,  0); }
408         | ivar ',' var_list { $$ = nn($1, 0, TYPE,  0, $3); }
409         ;
410 ivar    : var               { $$ = $1; }
411         | var ASGN expr     { $1->ini = $3; $$ = $1; }
412         | var ASGN ch_init  { $1->ini = $3; $$ = $1; }
413 /* for compatibility with Unix v.10: */
414         | NAME '[' CONST ']' OF '{' typ_list '}' {
415                               $1->nel = 1;
416                               if ($3) u_async++; else u_sync++;
417                               $1->ini = nn(0, $3, CHAN, 0, $7);
418                               cnt_mpars($7);
419                               $$ = $1;
420                             }
421         ;
422 ch_init : '[' CONST ']' OF '{' typ_list '}'
423                             { if ($2) u_async++; else u_sync++;
424                               cnt_mpars($6);
425                               $$ = nn(0, $2, CHAN, 0, $6);
426                             }
427         ;
428 var     : NAME              { $1->nel =  1; $$ = $1; }
429         | NAME '[' CONST ']' { $1->nel = $3; $$ = $1; }
430         ;
431 varref  : NAME              { $$ = nn($1, 0,  NAME,  0,  0); }
```

```
432         | NAME '[' expr ']' { $$ = nn($1, 0,  NAME, $3,  0); }
433         ;
434 stmnt   : varref ASGN expr  { $$ = nn($1->nsym, $2, ASGN, $1, $3); }
435         | varref RCV margs  { $$ = nn($1->nsym, $2,  'r', $1, $3); }
436         | varref SND margs  { $$ = nn($1->nsym, $2,  's', $1, $3); }
437         | PRINT '(' STRING prargs ')' { $$ = nn($3, $1,PRINT, $4, 0); }
438         | ASSERT expr       { $$ = nn(0, $1, ASSERT, $2, 0); }
439         | GOTO NAME         { $$ = nn($2, $1, GOTO, 0, 0); }
440         | expr              { $$ = nn(0, lineno, 'c', $1, 0); }
441         | NAME ':' stmnt    { $$ = nn($1, $3->nval,':',$3, 0); }
442         | IF options FI     { $$ = nn(0, $1, IF, 0, 0);
443                               $$->seql = $2;
444                             }
445         | DO                { pushbreak(); }
446           options OD        { $$ = nn(0, $1, DO, 0, 0);
447                               $$->seql = $3;
448                             }
449         | BREAK             { $$ = nn(break_dest(),$1,GOTO,0,0); }
450         | ATOMIC
451           '{'               { open_seq(0); }
452              sequence
453           '}'               { $$ = nn(0, $1, ATOMIC, 0, 0);
454                               $$->seql = seqlist(close_seq(), 0);
455                               make_atomic($$->seql->this);
456                             }
457         ;
458 options : option            { $$ = seqlist($1, 0); }
459         | option options    { $$ = seqlist($1, $2); }
460         ;
461 option  : SEP               { open_seq(0); }
462           sequence          { $$ = close_seq(); }
463         ;
464 expr    : '(' expr ')'              { $$ = $2; }
465         | expr '+' expr             { $$ = nn(0, 0,  '+', $1, $3); }
466         | expr '-' expr             { $$ = nn(0, 0,  '-', $1, $3); }
467         | expr '*' expr             { $$ = nn(0, 0,  '*', $1, $3); }
468         | expr '/' expr             { $$ = nn(0, 0,  '/', $1, $3); }
469         | expr '%' expr             { $$ = nn(0, 0,  '%', $1, $3); }
470         | expr '&' expr             { $$ = nn(0, 0,  '&', $1, $3); }
471         | expr '|' expr             { $$ = nn(0, 0,  '|', $1, $3); }
472         | expr '>' expr             { $$ = nn(0, 0,  '>', $1, $3); }
473         | expr '<' expr             { $$ = nn(0, 0,  '<', $1, $3); }
474         | expr GE expr              { $$ = nn(0, 0,   GE, $1, $3); }
475         | expr LE expr              { $$ = nn(0, 0,   LE, $1, $3); }
476         | expr EQ expr              { $$ = nn(0, 0,   EQ, $1, $3); }
477         | expr NE expr              { $$ = nn(0, 0,   NE, $1, $3); }
478         | expr AND expr             { $$ = nn(0, 0,  AND, $1, $3); }
479         | expr OR  expr             { $$ = nn(0, 0,   OR, $1, $3); }
480         | expr LSHIFT expr  { $$ = nn(0, 0,LSHIFT,$1, $3); }
481         | expr RSHIFT expr  { $$ = nn(0, 0,RSHIFT,$1, $3); }
482         | '~' expr          { $$ = nn(0, 0,  '~', $2,  0); }
483         | '-' expr %prec UMIN       { $$ = nn(0, 0, UMIN, $2,  0); }
484         | SND expr %prec NEG        { $$ = nn(0, 0,  '!', $2,  0); }
485         | RUN NAME '(' args ')'     { $$ = nn($2,$1, RUN, $4,  0); }
```

```
486         | LEN '(' varref ')'        { $$ = nn($3->nsym, $1, LEN, $3,  0); }
487         | varref RCV '[' margs ']'  { $$ = nn($1->nsym,$2,'R',$1,$4); }
488         | varref                    { $$ = $1; }
489         | CONST                     { $$ = nn(0,$1,  CONST,  0,  0); }
490         | TIMEOUT                   { $$ = nn(0,$1,TIMEOUT,  0,  0); }
491         | NAME '[' expr ']' '.' varref { $$ = rem_var($1, $3, $6); }
492         | NAME '[' expr ']' ':' NAME   { $$ = rem_lab($1, $3, $6); }
493         ;
494 typ_list: TYPE                       { $$ = nn(0, 0, $1, 0,  0); }
495         | TYPE ',' typ_list           { $$ = nn(0, 0, $1, 0, $3); }
496         ;
497 args    : /* empty */               { $$ = (Node *) 0; }
498         | arg                       { $$ = $1; }
499         ;
500 arg     : expr                      { $$ = nn(0, 0, ',', $1,  0); }
501         | expr ',' arg              { $$ = nn(0, 0, ',', $1, $3); }
502         ;
503 prargs  : /* empty */               { $$ = (Node *) 0; }
504         | ',' arg                   { $$ = $2; }
505         ;
506 margs   : arg                       { $$ = $1; }
507         | expr '(' arg ')'          { $$ = nn(0, 0, ',', $1, $3); }
508         ;
509 %%
510
511 /***** spin: main.c *****/
512
513 #include <stdio.h>
514 #include "spin.h"
515 #include "y.tab.h"
516
517 extern Symbol       *context;
518 extern int  lineno;
519 extern FILE *yyin;
520 Symbol      *Fname;
521 int verbose = 0;
522 int analyze = 0;
523 int s_trail = 0;
524 int m_loss  = 0;
525 int Named = 0;
526 int nr_errs = 0;
527
528 void
529 main(argc, argv)
530     char *argv[];
531 {
532     Symbol *s;
533     int T = (int) time((long *)0);
534
535     while (argc > 1 && argv[1][0] == '-')
536     {       switch (argv[1][1]) {
537             case 'a': analyze  =  1; break;
538             case 'g': verbose +=  1; break;
539             case 'l': verbose +=  2; break;
```

```
540             case 'm': m_loss   =  1; break;
541             case 'N': Named    =  1; break;
542             case 'n': T = atoi(&argv[1][2]); break;
543             case 'p': verbose +=  4; break;
544             case 'r': verbose +=  8; break;
545             case 's': verbose += 16; break;
546             case 'v': verbose += 32; break;
547             case 't': s_trail  =  1; break;
548             default : printf("use: spin -[agmlpqrst] [-nN] file\n");
549                       printf("\t-a produce an analyzer\n");
550                       printf("\t-g print all global variables\n");
551                       printf("\t-l print all local variables\n");
552                       printf("\t-m lose msgs sent to full queues\n");
553                       printf("\t-nN seed for random nr generator\n");
554                       printf("\t-p print all statements\n");
555                       printf("\t-r print receive events\n");
556                       printf("\t-s print send events\n");
557                       printf("\t-v verbose, more warnings\n");
558                       printf("\t-t follow a simulation trail\n");
559                       printf("\t-N force Naming of message types in trails\n");
560                       exit(1);
561             }
562             argc--, argv++;
563     }
564     if (argc > 1)
565     {       char outfile[17], cmd[64];
566             Fname = lookup(argv[1]);
567             mktemp(strcpy(outfile, "/tmp/spin.XXXXXX"));
568             sprintf(cmd, "/lib/cpp %s > %s", argv[1], outfile);
569             if (system(cmd))
570             {       unlink(outfile);
571                     exit(1);
572             } else if (!(yyin = fopen(outfile, "r")))
573             {       printf("cannot open %s\n", outfile);
574                     exit(1);
575             }
576             unlink(outfile);
577     } else
578             Fname = lookup("<stdin>");
579     srand(T);
580     s = lookup("_p"); s->type = PREDEF;
581     yyparse();
582     exit(nr_errs);
583 }
584
585 yywrap()    /* dummy routine */
586 {
587     return 1;
588 }
589
590 yyerror(s1, s2)
591     char *s1, *s2;
592 {
593     extern int yychar;
```

```
    printf("spin: %s line %d: ", Fname->name, lineno);
    if (s2)
            printf(s1, s2);
    else
            printf(s1);
    if (yychar) printf("    saw '%d'.", yychar);
    printf("\n"); fflush(stdout);
    nr_errs++;
}

void
fatal(s1, s2)
    char *s1, *s2;
{
    yyerror(s1, s2);
    fflush(stdout);
    exit(1);
}

char *
emalloc(n)
{   char *tmp = malloc(n);

    if (!tmp)
            fatal("not enough memory", (char *)0);
    memset(tmp, 0, n);
    return tmp;
}

Node *
nn(s, v, t, l, r)
    Symbol *s;
    Node *l, *r;
{
    Node *n = (Node *) emalloc(sizeof(Node));
    n->nval = v;
    n->ntyp = t;
    n->nsym = s;
    n->fname = Fname;
    n->lft  = l;
    n->rgt  = r;
    return n;
}

Node *
rem_lab(a, b, c)
    Symbol *a, *c;
    Node *b;
{
    if (!context || strcmp(context->name, ":never:") != 0)
            yyerror("warning: illegal use of ':' (outside never claim)", (char *)0);
    return  nn((Symbol *)0, 0, EQ,
            nn(lookup("_p"), 0, 'p', nn(a, 0, '?', b, (Node *)0), (Node *)0),
            nn(c, 0, 'q', nn(a, 0, NAME, (Node *)0, (Node *)0), (Node *)0));
```

```
}

Node *
rem_var(a, b, c)
    Symbol *a;
    Node *b, *c;
{
    Node *tmp;
    if (!context || strcmp(context->name, ":never:") != 0)
            yyerror("warning: illegal use of '.' (outside never claim)", (char *)0);
    tmp = nn(a, 0, '?', b, (Node *)0);
    return nn(c->nsym, 0, 'p', tmp, c->lft);
}

/***** spin: sym.c *****/

#include "spin.h"
#include "y.tab.h"

Symbol      *symtab[Nhash+1];
Symbol      *context = (Symbol *) 0;

hash(s)
    char *s;
{
    int h=0;

    while (*s)
    {       h += *s++;
            h <<= 1;
            if (h&(Nhash+1))
                    h |= 1;
    }
    return h&Nhash;
}

Symbol *
lookup(s)
    char *s;
{
    Symbol *sp;
    int h=hash(s);

    for (sp = symtab[h]; sp; sp = sp->next)
            if (strcmp(sp->name, s) == 0 && sp->context == context)
                    return sp;                      /* found  */
    if (context)                                    /* local  */
    for (sp = symtab[h]; sp; sp = sp->next)
            if (strcmp(sp->name, s) == 0 && !sp->context)
                    return sp;                      /* global */
    sp = (Symbol *) emalloc(sizeof(Symbol));        /* add    */
    sp->name = (char *) emalloc(strlen(s) + 1);
    strcpy(sp->name, s);
    sp->nel = 1;
```

```
702     sp->context = context;
703     sp->next = symtab[h];
704     symtab[h] = sp;
705
706     return sp;
707 }
708
709 void
710 settype(n, t)
711     Node *n;
712 {
713     while (n)
714     {       if (n->nsym->type)
715               yyerror("redeclaration of ‘%s’", n->nsym->name);
716             n->nsym->type = t;
717             if (n->nsym->nel <= 0)
718               yyerror("bad array size for ‘%s’", n->nsym->name);
719             n = n->rgt;
720     }
721 }
722
723 Node *Mtype = (Node *) 0;
724
725 void
726 setmtype(m)
727     Node *m;
728 {
729     Node *n = m;
730     if (Mtype)
731             yyerror("mtype redeclared", (char *)0);
732
733     Mtype = n;
734     while (n)       /* syntax check */
735     {       if (!n->lft || !n->lft->nsym
736             ||  (n->lft->ntyp != NAME)
737             ||   n->lft->lft)       /* indexed variable */
738                     fatal("bad mtype definition", (char *)0);
739             n = n->rgt;
740     }
741 }
742
743 Node *Symnode = 0;
744
745 void
746 syms(m)
747     Node *m;
748 {
749     if (Symnode)
750             yyerror("Duplicate Symmetry definition", (char *)0);
751
752     Symnode = m;
753 }
754 #include <stdio.h>
755
```

```
int SymCnt=0;
void
putsyms(fd, fh)
    FILE *fd, *fh;
{
    if (!Symnode || !Symnode->rgt || !Symnode->lft)
    {       fprintf(fh, "#define Normalize  0\n");
            return;
    }
    fprintf(fh, "#ifndef WHICH\n");
    fprintf(fh, "#define WHICH      1       /* must be either 1 or -1 */\n");
    fprintf(fh, "#endif\n");

    putdescend(fd, Symnode->lft, Symnode->rgt);

    fprintf(fh, "#define Normalize  1 ");
    while (SymCnt > 0)
            fprintf(fh, "\\\n\t\t&& Symmer%d() == WHICH ", --SymCnt);
    fprintf(fh, "\n");
}

void
putdescend(fd, a, b)
    FILE *fd;
    Node *a, *b;
{
    if (b->ntyp != ';')
            putasym(fd, a, b);
    else
    {       putasym(fd, a, b->lft);
            putasym(fd, a, b->rgt);
    }
}

void
putasym(fd, m, n)
    FILE *fd;
    Node *m, *n;
{
    Node *t1 = m;
    Node *t2 = n;
    fprintf(fd, "Symmer%d()\n{      long d;\n", SymCnt++);
    for ( ; t1 && t2; t1 = t1->rgt, t2 = t2->rgt)
    {       if (!t1->lft || !t2->lft)
                    fatal("bad Symmetry list", (char *)0);

            fprintf(fd, "   d = ");
            putstmnt(fd, t1->lft, 0);
            fprintf(fd, " - ");
            putstmnt(fd, t2->lft, 0);
            fprintf(fd, ";\n");

            fprintf(fd, "   if (d < 0) return -1;\n");
            fprintf(fd, "   if (d > 0) return  1;\n\n");
```

```
810     }
811     fprintf(fd, "   return 0;\n}\n");
812 }
813
814 ismtype(str)
815     char *str;
816 {
817     Node *n;
818     int cnt = 1;
819
820     for (n = Mtype; n; n = n->rgt)
821     {       if (strcmp(str, n->lft->nsym->name) == 0)
822                     return cnt;
823             cnt++;
824     }
825     return 0;
826 }
827
828 /***** spin: vars.c *****/
829
830 #include <stdio.h>
831 #include "spin.h"
832 #include "y.tab.h"
833
834 extern RunList      *X;
835 int Noglobal=0;
836
837 getval(s, n)
838     Symbol *s;
839 {
840     if (strcmp(s->name, "_p") == 0)
841             return (X && X->pc)?X->pc->seqno:0;
842     if (s->context && s->type)
843             return getlocal(s, n);
844     if (Noglobal)
845             return 0;
846     if (!s->type)   /* not declared locally */
847             s = lookup(s->name); /* try global */
848     return getglobal(s, n);
849 }
850
851 setval(v, n)
852     Node *v;
853 {
854     if (v->nsym->context && v->nsym->type)
855             return setlocal(v, n);
856     if (!v->nsym->type)
857             v->nsym = lookup(v->nsym->name);
858     return setglobal(v, n);
859 }
860
861 checkvar(s, n)
862     Symbol *s;
863 {
```

```
864     int i;
865
866     if (n >= s->nel || n < 0)
867     {       yyerror("array indexing error, `%s'", s->name);
868             return 0;
869     }
870     if (s->type == 0)
871     {       yyerror("undecl var `%s' (assuming int)", s->name);
872             s->type = INT;
873     }
874     if (s->val == (int *) 0)        /* uninitialized */
875     {       s->val = (int *) emalloc(s->nel*sizeof(int));
876             for (i = 0; i < s->nel; i++)
877             {       if (s->type != CHAN)
878                             s->val[i] = eval(s->ini);
879                     else
880                             s->val[i] = qmake(s);
881     }       }
882     return 1;
883 }
884
885 getglobal(s, n)
886     Symbol *s;
887 {
888     int i;
889     if (s->type == 0 && X && (i = find_lab(s, X->n)))
890             return i;
891     if (checkvar(s, n))
892             return cast_val(s->type, s->val[n]);
893     return 0;
894 }
895
896 void
897 typex(n, t)
898     Node *n;
899 {
900     if (n->ntyp == NAME && n->nsym->type != t
901     && (t == CHAN || n->nsym->type == CHAN))
902             yyerror("type clash (chan) in mesg pars", 0);
903 }
904
905 cast_val(t, v)
906 {   int i=0; short s=0; unsigned char u=0;
907
908     if (t == INT || t == CHAN) i = v;
909     else if (t == SHORT) s = (short) v;
910     else if (t == BYTE)  u = (unsigned char)v;
911     else if (t == BIT)   u = (unsigned char)(v&1);
912
913     if (v != i+s+u)
914             yyerror("value %d truncated in assignment", v);
915     return (int)(i+s+u);
916 }
917
```

```
setglobal(v, m)
    Node *v;
{
    int n = eval(v->lft);

    if (checkvar(v->nsym, n))
            v->nsym->val[n] = m;
    return 1;
}

void
dumpglobals()
{   extern Symbol *symtab[Nhash+1];
    register Symbol *sp;
    register int i, j, k, n, m;

    for (i = 0; i <= Nhash; i++)
    for (sp = symtab[i]; sp; sp = sp->next)
    {       if (!sp->type || sp->context)
                    continue;
            for (j = 0, m = -1; j < sp->nel; j++)
            {       if (sp->type == CHAN)
                    {       doq(sp, j);
                            k = 0;
                            continue;
                    }
                    n = getglobal(sp, j);
                    if (j == 0 || n != k)
                    {       if (m != j-1)
                                    printf("\t\t...\n");
                            if (sp->nel > 1)
                              printf("\t\t%s[%d] = %d\n",
                                    sp->name, j, n);
                            else
                              printf("\t\t%s = %d\n",
                                    sp->name, n);
                            m = j;
                    }
                    k = n;
    }       }
}

void
dumplocal(s)
    Symbol *s;
{
    Symbol *z;
    int i;

    for (z = s; z; z = z->next)
    for (i = 0; i < z->nel; i++)
    {       if (z->type == CHAN)
                    doq(z, i);
            else
```

```
972             {       if (z->nel > 1)
973                       printf("\t\t%s[%d] = %d\n",
974                       z->name, i, getval(z,i));
975                     else
976                       printf("\t\t%s = %d\n",
977                       z->name, getval(z,0));
978     }       }
979 }
980
981 /***** spin: mesg.c *****/
982
983 #include <stdio.h>
984 #include "spin.h"
985 #include "y.tab.h"
986
987 #define MAXQ        2500            /* default max # queues  */
988
989 extern      int lineno, verbose;
990 Queue       *qtab = (Queue *) 0;    /* linked list of queues */
991 Queue       *ltab[MAXQ];            /* linear list of queues */
992 int nqs=0;
993 int Mpars=0;        /* max nr of message parameters  */
994
995 void
996 cnt_mpars(n)
997     Node *n;
998 {
999     Node *m;
1000    int i=0;
1001
1002    for (m=n; m; m = m->rgt)
1003            i++;
1004    Mpars = max(Mpars, i);
1005 }
1006
1007 qmake(s)
1008    Symbol *s;
1009 {
1010    Node *m;
1011    Queue *q;
1012    int i; extern int analyze;
1013
1014    if (!s->ini)
1015            return 0;
1016    if (s->ini->ntyp != CHAN)
1017            fatal("bad channel initializer for %s\n", s->name);
1018    if (nqs >= MAXQ)
1019            fatal("too many queues (%s)", s->name);
1020
1021    q = (Queue *) emalloc(sizeof(Queue));
1022    q->qid = ++nqs;
1023    q->nslots = s->ini->nval;
1024    for (m = s->ini->rgt; m; m = m->rgt)
1025            q->nflds++;
```

```
1026    i = max(1, q->nslots);  /* 0-slot qs get 1 slot minimum */
1027
1028    q->contents  = (int *) emalloc(q->nflds*i*sizeof(int));
1029    q->fld_width = (short *) emalloc(q->nflds*sizeof(short));
1030    for (m = s->ini->rgt, i = 0; m; m = m->rgt)
1031            q->fld_width[i++] = m->ntyp;
1032    q->nxt = qtab;
1033    qtab = q;
1034    ltab[q->qid-1] = q;
1035
1036    return q->qid;
1037 }
1038
1039 qlen(n)
1040    Node *n;
1041 {
1042    int whichq = eval(n->lft)-1;
1043
1044    if (whichq < MAXQ && whichq >= 0 && ltab[whichq])
1045            return ltab[whichq]->qlen;
1046    return 0;
1047 }
1048
1049 q_is_sync(n)
1050    Node *n;
1051 {
1052    int whichq = eval(n->lft)-1;
1053
1054    if (whichq < MAXQ && whichq >= 0 && ltab[whichq])
1055            return (ltab[whichq]->nslots == 0);
1056    return 0;
1057 }
1058
1059 qsend(n)
1060    Node *n;
1061 {
1062    int whichq = eval(n->lft)-1;
1063    if (whichq == -1)
1064    {       printf("Error: sending to an uninitialized chan\n");
1065            whichq = 0;
1066    }
1067    if (whichq < MAXQ && whichq >= 0 && ltab[whichq])
1068    {       if (ltab[whichq]->nslots > 0)
1069                    return a_snd(ltab[whichq], n);
1070            else
1071                    return s_snd(ltab[whichq], n);
1072    }
1073    return 0;
1074 }
1075
1076 qrecv(n, full)
1077    Node *n;
1078 {
1079    int whichq = eval(n->lft)-1;
```

```
1080
1081    if (whichq == -1)
1082    {       printf("Error: receiving from an uninitialized chan\n");
1083            whichq = 0;
1084    }
1085    if (whichq < MAXQ && whichq >= 0 && ltab[whichq])
1086            return a_rcv(ltab[whichq], n, full);
1087    return 0;
1088 }
1089
1090 a_snd(q, n)
1091    Queue *q;
1092    Node *n;
1093 {
1094    Node *m; extern int m_loss;
1095    int i = q->qlen*q->nflds;       /* q offset */
1096    int j = 0;                      /* q field# */
1097
1098    if (q->nslots > 0 && q->qlen >= q->nslots)
1099            return m_loss;  /* q is full */
1100
1101    for (m = n->rgt; m && j < q->nflds; m = m->rgt, j++)
1102    {       q->contents[i+j] =
1103                    cast_val(q->fld_width[j], eval(m->lft));
1104            typex(m->lft, q->fld_width[j]); /* after eval(m->lft) */
1105            if (verbose&16)
1106                    sr_talk(n, eval(m->lft), "Send", "->", j,
1107                      q->nflds, m->lft && m->lft->ntyp == CONST);
1108    }
1109    if (verbose&16)
1110    {       for (i = j; i < q->nflds; i++)
1111                    sr_talk(n, 0, "Send", "->", i, q->nflds, 0);
1112            if (verbose&32)
1113            { if (j < q->nflds)
1114                    printf("\twarning: missing params in send\n");
1115              if (m)
1116                    printf("\twarning: too many params in send\n");
1117    }       }
1118    q->qlen++;
1119    return 1;
1120 }
1121
1122 a_rcv(q, n, full)
1123    Queue *q;
1124    Node *n;
1125 {
1126    Node *m;
1127    int j, k;
1128    if (q->qlen == 0)
1129            return 0;       /* q is empty */
1130
1131    for (m = n->rgt, j=0; m && j < q->nflds; m = m->rgt, j++)
1132    {       if (m->lft->ntyp == CONST)
1133            {       if (q->contents[j] != m->lft->nval)
```

```
1134                            return 0;       /* no match */
1135            } else if (m->lft->ntyp != NAME)
1136                    fatal("bad arg in receive", (char *)0);
1137    }
1138    if (verbose&8 && verbose&32)
1139    {       if (j < q->nflds)
1140              printf("\twarning: missing params in next recv\n");
1141            else if (m)
1142              printf("\twarning: too many params in next recv\n");
1143    }
1144    for (m = n->rgt, j=0; j<q->nflds; m = (m)?m->rgt:m, j++)
1145    {       if (verbose&8)
1146            sr_talk(n, q->contents[j], (full)?"Recv":"[Recv]", "<-", j,
1147                            q->nflds, m && m->lft->ntyp == CONST);
1148            if (m && m->lft->ntyp == NAME)
1149            {       setval(m->lft, q->contents[j]);
1150                    typex(m->lft, q->fld_width[j]);
1151            }
1152            for (k = 0; full && k < q->qlen-1; k++)
1153                    q->contents[k*q->nflds+j] =
1154                      q->contents[(k+1)*q->nflds+j];
1155    }
1156    if (full) q->qlen--;
1157    return 1;
1158 }
1159
1160 s_snd(q, n)
1161    Queue *q;
1162    Node *n;
1163 {
1164    Node *m;
1165    int i, j = 0;   /* q field# */
1166
1167    for (m = n->rgt; m && j < q->nflds; m = m->rgt, j++)
1168            q->contents[j] = cast_val(q->fld_width[j], eval(m->lft));
1169
1170    q->qlen = 1;
1171    if (!complete_rendez())
1172    {       q->qlen = 0;
1173            return 0;
1174    }
1175    if (verbose&16)
1176    {       m = n->rgt;
1177            for (j = 0; m && j < q->nflds; m = m->rgt, j++)
1178            {       sr_talk(n, eval(m->lft), "Sent", "->", j,
1179                      q->nflds, m->lft && m->lft->ntyp == CONST);
1180                    typex(m->lft, q->fld_width[j]);
1181            }
1182            for (i = j; i < q->nflds; i++)
1183                    sr_talk(n, 0, "Sent", "->", i, q->nflds, 0);
1184            if (verbose&32)
1185            {       if (j < q->nflds)
1186                      printf("\twarning: missing params in send\n");
1187                    if (m)
```

```
1188                        printf("\twarning: too many params in send\n");
1189    }       }
1190    return 1;
1191 }
1192
1193 void
1194 sr_talk(n, v, s, a, j, mx, named)
1195    Node *n;
1196    char *s, *a;
1197 {
1198    extern int Named;
1199    if (j == 0)
1200    {       whoruns();
1201            printf("line %3d, %s ", n->nval, s);
1202            sr_mesg(v, named||Named);
1203    } else
1204    {       printf(",");
1205            sr_mesg(v, named);
1206    }
1207    if (j == mx-1)
1208    {       printf("\t%s queue %d", a, eval(n->lft));
1209            if (n->nsym->type == CHAN)
1210                    printf(" (%s", n->nsym->name);
1211            else
1212                    printf(" (%s", lookup(n->nsym->name)->name);
1213            if (n->lft->lft)
1214                    printf("[%d]", eval(n->lft->lft));
1215            printf(")\n");
1216    }
1217    fflush(stdout);
1218 }
1219
1220 void
1221 sr_mesg(v, j)
1222 {  extern Node *Mtype;
1223
1224    int cnt = 1;
1225    Node *n;
1226    for (n = Mtype; n && j; n = n->rgt, cnt++)
1227            if (cnt == v)
1228            {       printf("%s", n->lft->nsym->name);
1229                    return;
1230            }
1231    printf("%d", v);
1232 }
1233
1234 void
1235 doq(s, n)
1236    Symbol *s;
1237 {
1238    Queue *q;
1239    int j, k;
1240    if (!s->val)    /* uninitialized queue */
1241            return;
```

```
1242    for (q = qtab; q; q = q->nxt)
1243    if (q->qid == s->val[n])
1244    {       if (s->nel != 1)
1245              printf("\t\tqueue %d (%s[%d]): ", q->qid, s->name, n);
1246            else
1247              printf("\t\tqueue %d (%s): ", q->qid, s->name);
1248            for (k = 0; k < q->qlen; k++)
1249            {       printf("[");
1250                    for (j = 0; j < q->nflds; j++)
1251                    {       if (j > 0) printf(",");
1252                            sr_mesg(q->contents[k*q->nflds+j], j==0);
1253                    }
1254                    printf("]");
1255            }
1256            printf("\n");
1257            break;
1258    }
1259 }
1260
1261 /***** spin: flow.c *****/
1262
1263 #include "spin.h"
1264 #include "y.tab.h"
1265
1266 Label      *labtab = (Label *) 0;
1267 Lbreak     *breakstack = (Lbreak *) 0;
1268 SeqList    *cur_s = (SeqList *) 0;
1269 int        Elcnt, break_id=0;
1270
1271 void
1272 open_seq(top)
1273 {  SeqList *t;
1274    Sequence *s = (Sequence *) emalloc(sizeof(Sequence));
1275
1276    t = seqlist(s, cur_s);
1277    cur_s = t;
1278    if (top) Elcnt = 1;
1279 }
1280
1281 Sequence *
1282 close_seq()
1283 {  Sequence *s = cur_s->this;
1284    Symbol *z;
1285
1286    if (s->frst == s->last)
1287    {       if ((z = has_lab(s->frst))
1288            && (strncmp(z->name, "progress", 8) == 0
1289            ||  strncmp(z->name, "accept", 6) == 0
1290            ||  strncmp(z->name, "end", 3) == 0))
1291            {       Element *y =    /* insert a skip */
1292                    new_el(nn((Symbol *)0, s->frst->n->nval, 'c',
1293                            nn((Symbol *)0, 1, CONST, (Node *)0,
1294                            (Node *)0), (Node *)0));
1295                    if (s->frst->n->ntyp == GOTO
```

```
1296                    ||  s->frst->n->ntyp == BREAK)
1297                    {       s->frst = y;
1298                            y->nxt = s->last;
1299                    } else
1300                    {       mov_lab(z, s->frst, y);
1301                            s->frst->nxt = y;
1302                            s->last = y;
1303    }       }       }
1304    cur_s = cur_s->nxt;
1305    return s;
1306 }
1307
1308 SeqList *
1309 seqlist(s, r)
1310    Sequence *s;
1311    SeqList *r;
1312 {
1313    SeqList *t = (SeqList *) emalloc(sizeof(SeqList));
1314    t->this = s;
1315    t->nxt = r;
1316    return t;
1317 }
1318
1319 Element *
1320 new_el(n)
1321    Node *n;
1322 {
1323    Element *m;
1324
1325    if (n && (n->ntyp == IF || n->ntyp == DO))
1326            return if_seq(n->seql, n->ntyp, n->nval);
1327    m = (Element *) emalloc(sizeof(Element));
1328    m->n = n;
1329    m->seqno = Elcnt++;
1330    return m;
1331 }
1332
1333 Element *
1334 if_seq(s, tok, lnno)
1335    SeqList *s;
1336 {
1337    Element *e = new_el((Node *) 0);
1338    Element *t = new_el(nn((Symbol *) 0, lnno, '.',
1339                    (Node *)0, (Node *)0)); /* target */
1340    SeqList *z;
1341
1342    e->n = nn((Symbol *)0, lnno, tok, (Node *)0, (Node *)0);
1343    e->sub = s;
1344    for (z = s; z; z = z->nxt)
1345            add_el(t, z->this);
1346    if (tok == DO)
1347    {       add_el(t, cur_s->this);
1348            t = new_el(nn((Symbol *)0, lnno, BREAK, (Node *)0, (Node *)0));
1349            set_lab(break_dest(), t);
```

```
1350            breakstack = breakstack->nxt;   /* pop stack */
1351    }
1352    add_el(e, cur_s->this);
1353    add_el(t, cur_s->this);
1354    return e;       /* destination node for label */
1355 }
1356
1357 void
1358 add_el(e, s)
1359    Element *e;
1360    Sequence *s;
1361 {
1362    if (e->n->ntyp == GOTO)
1363    {       Symbol *z;
1364            if ((z = has_lab(e))
1365            && (strncmp(z->name, "progress", 8) == 0
1366            ||  strncmp(z->name, "accept", 6) == 0
1367            ||  strncmp(z->name, "end", 3) == 0))
1368            {       Element *y =    /* insert a skip */
1369                    new_el(nn((Symbol *)0, e->n->nval, 'c',
1370                            nn((Symbol *)0, 1, CONST, (Node *)0,
1371                                    (Node *)0), (Node *)0));
1372                    mov_lab(z, e, y); /* gets its label */
1373                    add_el(y, s);
1374    }       }
1375    if (!s->frst)
1376            s->frst = e;
1377    else
1378            s->last->nxt = e;
1379    s->last = e;
1380 }
1381
1382 Node       *innermost;
1383
1384 Element *
1385 colons(n)
1386    Node *n;
1387 {
1388    if (!n)
1389            return (Element *) 0;
1390    if (n->ntyp == ':')
1391    {       Element *e = colons(n->lft);
1392            set_lab(n->nsym, e);
1393            return e;
1394    }
1395    innermost = n;
1396    return new_el(n);
1397 }
1398
1399 void
1400 add_seq(n)
1401    Node *n;
1402 {
1403    Element *e;
```

```
1404    if (!n) return;
1405    innermost = n;
1406    e = colons(n);
1407    if (innermost->ntyp != IF && innermost->ntyp != DO)
1408            add_el(e, cur_s->this);
1409 }
1410
1411 void
1412 set_lab(s, e)
1413    Symbol *s;
1414    Element *e;
1415 {
1416    Label *l; extern Symbol *context;
1417    if (!s) return;
1418    l = (Label *) emalloc(sizeof(Label));
1419    l->s = s;
1420    l->c = context;
1421    l->e = e;
1422    l->nxt = labtab;
1423    labtab = l;
1424 }
1425
1426 Element *
1427 get_lab(s)
1428    Symbol *s;
1429 {
1430    Label *l;
1431    for (l = labtab; l; l = l->nxt)
1432            if (s == l->s)
1433                    return (l->e);
1434    fatal("undefined label %s", s->name);
1435    return 0;       /* doesn't get here */
1436 }
1437
1438 Symbol *
1439 has_lab(e)
1440    Element *e;
1441 {
1442    Label *l;
1443    for (l = labtab; l; l = l->nxt)
1444            if (e == l->e)
1445                    return (l->s);
1446    return (Symbol *) 0;
1447 }
1448
1449 void
1450 mov_lab(z, e, y)
1451    Symbol *z;
1452    Element *e, *y;
1453 {
1454    Label *l;
1455    for (l = labtab; l; l = l->nxt)
1456            if (e == l->e)
1457            {       l->e = y;
```

```
1458                    return;
1459            }
1460    fatal("cannot happen - mov_lab %s", z->name);
1461 }
1462
1463 find_lab(s, c)
1464    Symbol *s, *c;
1465 {
1466    Label *l;
1467    for (l = labtab; l; l = l->nxt)
1468    {       if (strcmp(s->name, l->s->name) == 0
1469            &&  strcmp(c->name, l->c->name) == 0)
1470                    return (l->e->seqno);
1471    }
1472    return 0;
1473 }
1474
1475 void
1476 pushbreak()
1477 {  Lbreak *r = (Lbreak *) emalloc(sizeof(Lbreak));
1478    Symbol *l;
1479    char buf[32];
1480
1481    sprintf(buf, ":b%d", break_id++);
1482    l = lookup(buf);
1483    r->l = l;
1484    r->nxt = breakstack;
1485    breakstack = r;
1486 }
1487
1488 Symbol *
1489 break_dest()
1490 {  if (!breakstack)
1491            fatal("misplaced break statement", (char *)0);
1492    return breakstack->l;
1493 }
1494
1495 void
1496 make_atomic(s)
1497    Sequence *s;
1498 {
1499    walk_atomic(s->frst, s->last);
1500    s->last->status &= ~ATOM;
1501    s->last->status |= L_ATOM;
1502 }
1503
1504 void
1505 walk_atomic(a, b)
1506    Element *a, *b;
1507 {
1508    Element *f;
1509    SeqList *h;
1510    for (f = a; ; f = f->nxt)
1511    {       f->status |= ATOM;
```

```
1512            for (h = f->sub; h; h = h->nxt)
1513                    walk_atomic(h->this->frst, h->this->last);
1514            if (f == b)
1515                    break;
1516    }
1517 }
1518
1519 /***** spin: run.c *****/
1520
1521 #include <stdio.h>
1522 #include "spin.h"
1523 #include "y.tab.h"
1524
1525 Element *
1526 eval_sub(e)
1527    Element *e;
1528 {
1529    Element *f, *g;
1530    SeqList *z;
1531    int i, j, k;
1532    extern int Rvous, lineno;
1533
1534    if (!e->n)
1535            return (Element *)0;
1536    if (e->n->ntyp == GOTO)
1537            return (!Rvous)?get_lab(e->n->nsym):(Element *)0;
1538    if (e->sub)
1539    {       for (z = e->sub, j=0; z; z = z->nxt)
1540                    j++;
1541            k = rand()%j;   /* nondeterminism */
1542            for (i = 0, z = e->sub; i < j+k; i++)
1543            {       if (i >= k && (f = eval_sub(z->this->frst)))
1544                            return f;
1545                    z = (z->nxt)?z->nxt:e->sub;
1546            }
1547    } else
1548    {       if (e->n->ntyp == ATOMIC)
1549            {       f = e->n->seql->this->frst;
1550                    g = e->n->seql->this->last;
1551                    g->nxt = e->nxt;
1552                    if (!(g = eval_sub(f))) /* atomic guard */
1553                            return (Element *)0;
1554                    Rvous=0;
1555                    while (g && (g->status & (ATOM|L_ATOM))
1556                    && !(f->status & L_ATOM))
1557                    {       f = g;
1558                            g = eval_sub(f);
1559                    }
1560                    if (!g)
1561                    {       wrapup();
1562                            lineno = f->n->nval;
1563                            fatal("atomic seq blocks", (char *)0);
1564                    }
1565                    return g;
```

```
1566            } else if (Rvous)
1567            {       if (eval_sync(e->n))
1568                            return e->nxt;
1569            } else
1570                    return (eval(e->n))?e->nxt:(Element *)0;
1571    }
1572    return (Element *)0;
1573 }
1574
1575 eval_sync(now)
1576    Node *now;
1577 {  /* allow only synchronous receives
1578    /* and related node types    */
1579
1580    if (now)
1581    switch (now->ntyp) {
1582    case TIMEOUT:   case PRINT:     case ASSERT:
1583    case RUN:       case LEN:       case 's':
1584    case 'c':       case ASGN:      case BREAK:
1585    case IF:        case DO:        case '.':
1586            return 0;
1587    case 'R':
1588    case 'r':
1589            if (!q_is_sync(now))
1590                    return 0;
1591    }
1592    return eval(now);
1593 }
1594
1595 eval(now)
1596    Node *now;
1597 {
1598    extern int Tval;
1599
1600    if (now)
1601    switch (now->ntyp) {
1602    case CONST: return now->nval;
1603    case   '!': return !eval(now->lft);
1604    case  UMIN: return -eval(now->lft);
1605    case   '~': return ~eval(now->lft);
1606
1607    case   '/': return (eval(now->lft) / eval(now->rgt));
1608    case   '*': return (eval(now->lft) * eval(now->rgt));
1609    case   '-': return (eval(now->lft) - eval(now->rgt));
1610    case   '+': return (eval(now->lft) + eval(now->rgt));
1611    case   '%': return (eval(now->lft) % eval(now->rgt));
1612    case   '<': return (eval(now->lft) <  eval(now->rgt));
1613    case   '>': return (eval(now->lft) >  eval(now->rgt));
1614    case   '&': return (eval(now->lft) &  eval(now->rgt));
1615    case   '|': return (eval(now->lft) |  eval(now->rgt));
1616    case    LE: return (eval(now->lft) <= eval(now->rgt));
1617    case    GE: return (eval(now->lft) >= eval(now->rgt));
1618    case    NE: return (eval(now->lft) != eval(now->rgt));
1619    case    EQ: return (eval(now->lft) == eval(now->rgt));
```

```
1620    case    OR: return (eval(now->lft) || eval(now->rgt));
1621    case   AND: return (eval(now->lft) && eval(now->rgt));
1622    case LSHIFT: return (eval(now->lft) << eval(now->rgt));
1623    case RSHIFT: return (eval(now->lft) >> eval(now->rgt));
1624
1625    case TIMEOUT: return Tval;
1626
1627    case   RUN: return enable(now->nsym, now->lft);
1628    case   LEN: return qlen(now);
1629    case   's': return qsend(now);          /* send         */
1630    case   'r': return qrecv(now, 1);       /* full-receive */
1631    case   'R': return qrecv(now, 0);       /* test only    */
1632    case   'c': return eval(now->lft);      /* condition    */
1633    case   'p': return remotevar(now);
1634    case   'q': return remotelab(now);
1635    case PRINT: return interprint(now);
1636    case  ASGN: return setval(now->lft, eval(now->rgt));
1637    case  NAME: return getval(now->nsym, eval(now->lft));
1638    case ASSERT: if (eval(now->lft)) return 1;
1639                 yyerror("assertion violated", (char *) 0);
1640                 wrapup(); exit(1);
1641    case  IF: case DO: case BREAK:  /* compound structure */
1642    case   '.': return 1;   /* return label for compound */
1643    case   '@': return 0;   /* stop state */
1644    default   : printf("spin: bad node type %d (run)\n", now->ntyp);
1645                fflush(stdout);
1646                exit(1);
1647    }
1648    return 0;
1649 }
1650
1651 interprint(n)
1652    Node *n;
1653 {
1654    Node *tmp = n->lft;
1655    char c, *s = n->nsym->name;
1656    int i, j;
1657
1658    for (i = 0; i < strlen(s); i++)
1659            switch (s[i]) {
1660            default:   putchar(s[i]); break;
1661            case '\"': break; /* ignore */
1662            case '\\':
1663                    switch(s[++i]) {
1664                    case 't': putchar('\t'); break;
1665                    case 'n': putchar('\n'); break;
1666                    default:  putchar(s[i]); break;
1667                    }
1668                    break;
1669            case  '%':
1670                    if ((c = s[++i]) == '%')
1671                    {      putchar('%'); /* literal */
1672                           break;
1673                    }
```

```
1674                    if (!tmp)
1675                    {      yyerror("too few print args %s", s);
1676                           break;
1677                    }
1678                    j = eval(tmp->lft);
1679                    tmp = tmp->rgt;
1680                    switch(c) {
1681                    case 'c': printf("%c", j); break;
1682                    case 'd': printf("%d", j); break;
1683                    case 'o': printf("%o", j); break;
1684                    case 'u': printf("%u", j); break;
1685                    case 'x': printf("%x", j); break;
1686                    default:  yyerror("unrecognized print cmd %%'%c'", c);
1687                              break;
1688                    }
1689                    break;
1690            }
1691    fflush(stdout);
1692    return 1;
1693 }
1694
1695 /***** spin: sched.c *****/
1696
1697 #include <stdio.h>
1698 #include "spin.h"
1699 #include "y.tab.h"
1700
1701 int nproc = 0;
1702 int nstop = 0;
1703 int Tval  = 0;
1704 int Rvous = 0;
1705 int depth = 0;
1706
1707 RunList            *X   = (RunList  *) 0;
1708 RunList            *run = (RunList  *) 0;
1709 ProcList   *rdy = (ProcList *) 0;
1710 Element            *eval_sub();
1711 extern int verbose, lineno, s_trail, analyze;
1712 extern Symbol      *Fname;
1713 extern char        *claimproc;
1714 extern int Noglobal;
1715 int Have_claim=0;
1716
1717 void
1718 runnable(s, n)
1719    Sequence *s;    /* body */
1720    Symbol *n;      /* name */
1721 {
1722    RunList *r = (RunList *) emalloc(sizeof(RunList));
1723    r->n = n;
1724    r->pid = nproc++;
1725    r->pc = s->frst;
1726    r->maxseq = s->last->seqno;
1727    r->nxt = run;
```

```
1728    run = r;
1729 }
1730
1731 void
1732 ready(n, p, s)
1733    Symbol *n;      /* process name */
1734    Node *p;        /* formal parameters */
1735    Sequence *s;    /* process body */
1736 {
1737    ProcList *r = (ProcList *) emalloc(sizeof(ProcList));
1738    r->n = n;
1739    r->p = p;
1740    r->s = s;
1741    r->nxt = rdy;
1742    rdy = r;
1743 }
1744
1745 enable(s, n)
1746    Symbol *s;      /* process name */
1747    Node *n;        /* actual parameters */
1748 {
1749    ProcList *p;
1750    for (p = rdy; p; p = p->nxt)
1751            if (strcmp(s->name, p->n->name) == 0)
1752            {       runnable(p->s, p->n);
1753                    setparams(run, p, n);
1754                    return (nproc-nstop-1); /* pid */
1755            }
1756    return 0; /* process not found */
1757 }
1758
1759 void
1760 start_claim(n)
1761 {  ProcList *p;
1762    int i;
1763
1764    for (p = rdy, i=1; p; p = p->nxt, i++)
1765            if (i == n)
1766            {       runnable(p->s, p->n);
1767                    Have_claim = 1;
1768                    return;
1769            }
1770    fatal("couldn't find claim", (char *) 0);
1771 }
1772
1773 void
1774 sched()
1775 {  Element *e;
1776    RunList *Y;     /* previous process in run queue */
1777    int i;
1778
1779    if (analyze)
1780    {       gensrc();
1781            return;
```

```
1782    } else if (s_trail)
1783    {       match_trail();
1784            return;
1785    }
1786    if (claimproc)
1787            printf("warning: claims are ignored in simulations\n");
1788
1789    for (Tval=i=0; Tval < 2; Tval++, i=0)
1790    {       while (i < nproc-nstop)
1791            for (X=run, Y=0, i=0; X; X = X->nxt)
1792            {       lineno = X->pc->n->nval;
1793                    Fname  = X->pc->n->fname;
1794                    if (e = eval_sub(X->pc))
1795                    {       X->pc = e; Tval=0;
1796                            talk(e, X->symtab);
1797                    } else
1798                    {       if (X->pc->n->ntyp == '@'
1799                            &&  X->pid == (nproc-nstop-1))
1800                            {       if (Y)
1801                                            Y->nxt = X->nxt;
1802                                    else
1803                                            run = X->nxt;
1804                                    nstop++; Tval=0;
1805                                    if (verbose&4)
1806                                    {       whoruns();
1807                                            printf("terminates\n");
1808                                    }
1809                            } else
1810                                    i++;
1811                    }
1812                    Y = X;
1813    }       }
1814    wrapup();
1815 }
1816
1817 wrapup()
1818 {  if (depth)      /* for guided simulations, Chapter 12 */
1819            printf("step %d, ", depth);
1820    if (nproc != nstop)
1821    {       printf("#processes: %d\n", nproc-nstop);
1822            dumpglobals();
1823            verbose &= ~1;  /* no more globals */
1824            verbose |=  4;  /* add process states */
1825            for (X = run; X; X = X->nxt)
1826                    talk(X->pc, X->symtab);
1827    }
1828    printf("%d processes created\n", nproc);
1829 }
1830
1831 complete_rendez()
1832 {  RunList *orun = X;
1833    Element *e;
1834    int res=0;
1835
```

```
1836    if (s_trail)    /* for guided simulations, Chapter 12 */
1837            return 1;
1838    Rvous = 1;
1839    for (X = run; X; X = X->nxt)
1840            if (X != orun && (e = eval_sub(X->pc)))
1841            {       X->pc = e;
1842                    if (verbose&4)
1843                    {       printf("rendezvous: %s ",X->n->name);
1844                            printf("<-> %s\n", orun->n->name);
1845                            printf("=r==:   ");
1846                            talk(e, X->symtab);
1847                            printf("=s==:   ");
1848                            X = orun;
1849                            talk(X->pc, X->symtab);
1850                    }
1851                    res = 1;
1852                    break;
1853            }
1854    Rvous = 0;
1855    X = orun;
1856    return res;
1857 }
1858
1859 /***** Runtime - Local Variables *****/
1860
1861 void
1862 addsymbol(r, s)
1863    RunList *r;
1864    Symbol  *s;
1865 {
1866    Symbol *t = (Symbol *) emalloc(sizeof(Symbol));
1867    int i;
1868
1869    t->name = s->name;
1870    t->type = s->type;
1871    t->nel  = s->nel;
1872    t->ini  = s->ini;
1873    if (s->val)             /* if initialized, copy it */
1874    {       t->val = (int *) emalloc(s->nel*sizeof(int));
1875            for (i = 0; i < s->nel; i++)
1876                    t->val[i] = s->val[i];
1877    } else
1878            checkvar(t, 0); /* initialize it */
1879    t->next = r->symtab;    /* add it */
1880    r->symtab = t;
1881 }
1882
1883 void
1884 naddsymbol(r, s, k)
1885    RunList *r;
1886    Symbol  *s;
1887 {
1888    Symbol *t = (Symbol *) emalloc(sizeof(Symbol));
1889    int i;
```

```
1890
1891    t->name = s->name;
1892    t->type = s->type;
1893    t->nel  = s->nel;
1894    t->ini  = s->ini;
1895    t->val = (int *) emalloc(s->nel*sizeof(int));
1896    if (s->nel != 1)
1897    fatal("array in formal parameter list, %s", s->name);
1898    for (i = 0; i < s->nel; i++)
1899            t->val[i] = k;
1900    t->next = r->symtab;
1901    r->symtab = t;
1902 }
1903
1904 typck(n, t, s)
1905    Node *n;
1906    char *s;
1907 {
1908    if (!n || !n->lft
1909    ||  (n->lft->ntyp == NAME && n->lft->nsym->type != t
1910         && n->lft->nsym->type != 0
1911         && (t == CHAN || n->lft->nsym->type == CHAN))
1912    ||  (n->lft->ntyp == NAME && n->lft->nsym->type == 0
1913         && lookup(n->lft->nsym->name)->type != t) )
1914    {       yyerror("error in parameters of run %s(...)", s);
1915            return 0;
1916    }
1917    return 1;
1918 }
1919
1920 void
1921 setparams(r, p, q)
1922    RunList *r;
1923    ProcList *p;
1924    Node *q;
1925 {
1926    Node *f, *a;    /* formal and actual pars */
1927    Node *t;        /* list of pars of 1 type */
1928
1929    for (f = p->p, a = q; f; f = f->rgt) /* one type at a time */
1930    for (t = f->lft; t; t = t->rgt, a = (a)?a->rgt:a)
1931    {       int k;
1932            if (!a) fatal("missing actual parameters: '%s'", p->n->name);
1933            k = eval(a->lft);        /* must be initialized*/
1934            if (typck(a, t->nsym->type, p->n->name))
1935            {       if (t->nsym->type == CHAN)
1936                            naddsymbol(r, t->nsym, k); /* copy */
1937                    else
1938                    {       t->nsym->ini = a->lft;
1939                            addsymbol(r, t->nsym);
1940                    }
1941            }
1942    }
1943 }
```

```
1944
1945 Symbol *
1946 findloc(s, n)
1947    Symbol *s;
1948 {
1949    Symbol *r = (Symbol *) 0;
1950
1951    if (n >= s->nel || n < 0)
1952    {       yyerror("array indexing error %s", s->name);
1953            return (Symbol *) 0;
1954    }
1955
1956    if (!X)
1957    {       if (analyze)
1958                    fatal("error, cannot evaluate variable '%s'", s->name);
1959            else
1960                    yyerror("error, cannot evaluate variable '%s'", s->name);
1961            return (Symbol *) 0;
1962    }
1963    for (r = X->symtab; r; r = r->next)
1964            if (strcmp(r->name, s->name) == 0)
1965                    break;
1966    if (!r && !Noglobal)
1967    {       addsymbol(X, s);
1968            r = X->symtab;
1969    }
1970    return r;
1971 }
1972
1973 getlocal(s, n)
1974    Symbol *s;
1975 {
1976    Symbol *r;
1977
1978    r = findloc(s, n);
1979    if (r) return cast_val(r->type, r->val[n]);
1980    return 0;
1981 }
1982
1983 setlocal(p, m)
1984    Node *p;
1985 {
1986    int n = eval(p->lft);
1987    Symbol *r = findloc(p->nsym, n);
1988
1989    if (r) r->val[n] = m;
1990    return 1;
1991 }
1992
1993 void
1994 whoruns()
1995 {  if (!X) return;
1996
1997    if (Have_claim && X->pid >= 1)
```

```
1998    {       if (X->pid == 1)
1999                    printf("proc  - (%s)    ", X->n->name);
2000            else
2001                    printf("proc %2d (%s)   ", X->pid-1, X->n->name);
2002    } else
2003            printf("proc %2d (%s)   ", X->pid, X->n->name);
2004 }
2005
2006 void
2007 talk(e, s)
2008    Element *e;
2009    Symbol *s;
2010 {
2011    if (verbose&4)
2012    {       p_talk(e);
2013            if (verbose&1) dumpglobals();
2014            if (verbose&2) dumplocal(s);
2015    }
2016 }
2017
2018 void
2019 p_talk(e)
2020    Element *e;
2021 {
2022    whoruns();
2023    printf("line %d (state %d)\n",
2024            (e && e->n && e->n->nval)?e->n->nval:-1, e->seqno);
2025 }
2026
2027 remotelab(n)
2028    Node *n;
2029 {
2030    int i;
2031
2032    if (n->nsym->type)
2033            fatal("not a labelname: ‘%s’", n->nsym->name);
2034    if ((i = find_lab(n->nsym, n->lft->nsym)) == 0)
2035            fatal("unknown labelname: %s", n->nsym->name);
2036    return i;
2037 }
2038
2039 remotevar(n)
2040    Node *n;
2041 {
2042    int pno, i, j, trick=0;
2043    RunList *Y, *oX = X;
2044
2045    if (!n->lft->lft)
2046    {       yyerror("missing pid in %s", n->nsym->name);
2047            return 0;
2048    }
2049    pno = eval(n->lft->lft); /* pid */
2050 TryAgain:
2051    i = nproc - nstop;
```

```
2052    for (Y = run; Y; Y = Y->nxt)
2053    if (--i == pno)
2054    {       if (strcmp(Y->n->name, n->lft->nsym->name))
2055            {       if (!trick && Have_claim)
2056                    {       trick = 1; pno++;
2057                            /* assumes user only guessed the pid */
2058                            goto TryAgain;
2059                    }
2060                    printf("remote ref %s[%d] refers to %s\n",
2061                            n->lft->nsym->name, pno, Y->n->name);
2062                    yyerror("wrong proctype %s", Y->n->name);
2063            }
2064            { extern int Noglobal;
2065              Noglobal=1; /* make sure it's not created by default */
2066              if (n->nsym->type == 0) n->nsym->type = INT;
2067              X = Y; j = getval(n->nsym, eval(n->rgt)); X = oX;
2068              Noglobal=0;
2069            }
2070            return j;
2071    }
2072    printf("remote ref: %s[%d] ", n->lft->nsym->name, pno);
2073    yyerror("variable %s not found", n->nsym->name);
2074    return 0;
2075 }
2076
2077 /***** spin: dummy.c *****/
2078
2079 gensrc()
2080 {
2081    printf("analyze: not defined\n");
2082 }
2083
2084 match_trail()
2085 {
2086    printf("trails: not defined\n");
2087 }
```