# A PROTOCOL SIMULATOR 12

## 12.1 INTRODUCTION

Without the proper tools it may be possible to design a correct protocol, but in most cases it will be impossible to formally establish its correctness with any measure of reliability. So far, we have occupied ourselves mainly with the development of a design discipline based on the usage of formal validation models. In this chapter we extend this discipline with a software tool for simulating the behavior of validation models written in PROMELA. The tool is called SPIN, which is short for: simple PROMELA interpreter[1]. SPIN can simulate the execution of a validation model by interpreting PROMELA statements on the fly. It can be used on either partial or complete protocol designs, at any level of abstraction. It can quickly tell us whether or not we are on the right track with a design and as such it can be a valuable design tool.

The program that we develop in this chapter will not try to validate correctness requirements. That is a task for a validator (Chapter 13). A small exception to that rule is made for PROMELA `assert` statements, since the corresponding requirements are validated as a mere side-effect of their execution. The validation of non-progress cycles, invalid end-states, and temporal claims, however, is outside the scope of a simulator. The complete program for the simulator contains about 2000 lines of text. It includes a lexical analyzer, a parser, and a process scheduler. It does require a fair amount of explanation and some familiarity with C and UNIX to get through this chapter. But, rest assured, it is not necessary to understand the details of the implementation to be able to use the simulator. Below we first discuss the general structure and the type of output the simulator can generate. Then we discuss a small version of the program, only for evaluating PROMELA expressions, and show how it works.

1. The terms *simulator*, *interpreter*, and *evaluator* are used as synonyms in this chapter.

Finally, we extend this program into a complete interpreter, by adding the missing pieces one by one. In the next chapter SPIN is extended with an option for performing fast automated protocol validations. A source listing for the final version of SPIN can be found in Appendices D and E.

## 12.2 SPIN – OVERVIEW

To build the interpreter we rely on the UNIX programming tools *yacc*, *lex*, and *make*. A casual familiarity with these tools is therefore assumed. We concentrate here on what the tools can do for us, rather than explain how they work inside. In case of emergency consult a UNIX manual or refer to the Bibliographic Notes at the end of this chapter for pointers to other literature that may be helpful.

### A SAMPLE SIMULATION RUN

Consider the example program from Chapter 5 for calculating the factorial of a positive integer number.

```
proctype fact(int n; chan p)
{       int result;

        if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                chan child = [1] of { int };
                run fact(n-1, child);
                child?result;
                p!n*result
        fi
}
init
{       int result;
        chan child = [1] of { int };

        run fact(12, child);
        child?result;
        printf("result: %d\n", result)
}
```

Running the analyzer on this program produces the following output:

```
$ spin factorial
result: 479001600
13 processes created
```

where `$` is a UNIX system prompt. And fortunately,

```
12*11*10*9*8*7*6*5*4*3*2*1 = 479001600
```

Running the simulator in verbose mode gives us a little more information about the run, for instance by printing all message transmissions, with the number of the process performing them, the contents of the message being sent, and the name of the destination channel.

```
$ spin -s factorial
proc 12 (fact)  line   5, Send 1         -> queue 12 (p)
proc 11 (fact)  line  10, Send 2         -> queue 11 (p)
proc 10 (fact)  line  10, Send 6         -> queue 10 (p)
proc  9 (fact)  line  10, Send 24        -> queue 9 (p)
proc  8 (fact)  line  10, Send 120       -> queue 8 (p)
proc  7 (fact)  line  10, Send 720       -> queue 7 (p)
proc  6 (fact)  line  10, Send 5040      -> queue 6 (p)
proc  5 (fact)  line  10, Send 40320     -> queue 5 (p)
proc  4 (fact)  line  10, Send 362880    -> queue 4 (p)
proc  3 (fact)  line  10, Send 3628800   -> queue 3 (p)
proc  2 (fact)  line  10, Send 39916800 -> queue 2 (p)
proc  1 (fact)  line  10, Send 479001600 -> queue 1 (p)
result: 479001600
13 processes created
```

The column with the message value now implicitly gives us a running count of the factorial being computed. If still more information is needed, we can also run the simulator with additional flags to print, for instance, message receptions or the values of variables. But the above example suffices for now.

## 12.3 EXPRESSIONS

One specific function that the simulator must perform is the evaluation of expressions. In a statement such as

```
crunch!data(3*12+4/2)
```

the simulator must evaluate three expressions:

- ○ The value of the destination `crunch`
- ○ The value of the message type `data` and
- ○ The value of the argument `3*12+4/2`

The evaluation of expressions may seem insignificant at first, but since PROMELA is founded on the concept of executability, the evaluation of statements in general is really at the core of the simulator. To keep things simple, let us therefore begin with a small program that can do no more than evaluate PROMELA expressions.

We have to tell our program what valid expressions look like and how they should be evaluated. The first issue calls for a grammar specification. If we ignore variable names for a while, the simplest form of expression is a number, say an integer constant. We write a constant as a series of one or more digits, where a digit is any symbol in the range `'0'` to `'9'`. If we formalize this we can write

```
digit   :       '0' | '1' | '2' | '3' | '4'
        |       '5' | '6' | '7' | '8' | '9'
```

The term we are defining is on the left side of the colon, and the defining symbols are on the right side where the vertical bar `'|'` is used to separate alternatives. Recursively then, we can define a constant as a series of one or more digits, as follows:

```
const   :       digit
        |       digit const
```

And, similarly, we can specify that a simple expression is just a number by writing

```
expr    :       const
```

It is now easy to tag on more interesting types of expressions. From any two valid expressions we can make another valid one by adding, subtracting, multiplying, or dividing them. Recursively again, we define

```
expr    :       const
        |       expr '+' expr
        |       expr '-' expr
        |       expr '*' expr
        |       expr '/' expr
        |       '(' expr ')'
```

The last line is for good form: if `2+5` is a valid expression, then so is `(2+5)`. It also allows us to force the order of evaluation of subexpressions, but more about that later.

Formally, what we have defined here are three *production rules*. On the left side of the rule (before the colon) we write the phrase we want to define. On the right side we write a number of alternative ways in which the phrase can be constructed, separated by vertical bars. Quoted characters are called *literals*. Names written in capitals are called *terminals* or *tokens*. Everything else is called a *non-terminal* and must be defined, that is, it must occur somewhere on the left side of a production rule. A lexical analyzer is used to recognize the terminals and literals and pass them on to a parser. The parser can then restrict itself to checking the grammar, and building a ‘‘parse tree.’’ This general structure is illustrated in Figure 12.1.

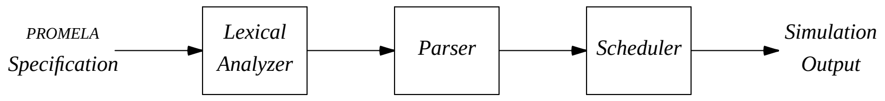

*Figure 12.1 — General Structure of the Simulator*

The scheduler evaluates the program by walking down the parse tree, evaluating its nodes in accordance with the semantics of the language. So it is important that the parser delivers a tree structure to the scheduler that is connected in such a way that it can be evaluated on the fly. But, more about the parser and the scheduler later; let us first look at the lexical analyzer.

A first specification of the lexical analyzer is fairly straightforward to define with the UNIX tool *lex*. In fact, with *lex*, it is easy to consider numbers a special class of token named `CONST` (uppercase, because it is now a terminal), with their numeric value attached as an attribute. In this way the grammar specification can assume the existence of the number tokens, rather than having to parse and check them for syntax. This is what the specification of the lexical analyzer looks like. We call this first version of the program `spine` (SPIN expression evaluator). As always, the line numbers are not part of the program text.

```
 1 /***** spine: lex.l *****/
 2
 3 %{
 4 #include "spin.h"
 5 #include "y.tab.h"
 6
 7 int lineno = 1;
 8 %}
 9
10 %%
11 [ \t]        { /* ignore white space */ }
12 [0-9]+       { yylval.val = atoi(yytext); return CONST; }
13 \n           { lineno++; }
14 .            { return yytext[0]; }
```

The program skips over white space (line 11), counts newlines (line 13), calculates the value of sequences of digits (line 12) and returns it as part of a token of type `CONST`. Everything else is passed on as a literal (line 14), i.e., as a single character. We rely on *yacc* to produce the definition of the name `CONST`. It is contained in the header file `y.tab.h` that is included on line 5. But before we discuss the input for *yacc*, we have to go back briefly to the grammar definition.

We still have to define what an expression such as `2+5*3` means. As far as our program is concerned, it could equally well mean either `(2+5)*3` or `2+(5*3)`. The convention is that multiplication and division have a higher precedence than addition and subtraction, which means that the second interpretation is the correct one. We will see below how these precedence rules can be stated formally in a grammar specification.

All rules and definitions given so far can be expressed with the UNIX parser generator *yacc*. A complete *yacc* specification for the grammar defined so far looks as follows:

```
 1 /***** spine: spin.y *****/
 2
 3 %{
 4 #include "spin.h"           /* defines what Nodes are etc */
 5 %}
 6
 7 %union{                     /* defines the data type of   */
 8     int val;                /* the internal parse stack   */
 9     Node *node;
10 }
11
12 %token      <val>   CONST   /* token CONST has a value    */
13 %type       <node>  expr    /* expressions produce nodes  */
14
15 %left       ’+’ ’-’         /* left associative operators */
16 %left       ’*’ ’/’         /* idem, highest precedence   */
17 %%
18
19 /** Grammar Rules **/
20
21 program : expr              { printf("= %d\n", eval($1)); }
22             ;
23 expr    : ’(’ expr ’)’      { $$ = $2; }
24         | expr ’+’ expr     { $$ = nn( 0, ’+’, $1, $3); }
25         | expr ’-’ expr     { $$ = nn( 0, ’-’, $1, $3); }
26         | expr ’*’ expr     { $$ = nn( 0, ’*’, $1, $3); }
27         | expr ’/’ expr     { $$ = nn( 0, ’/’, $1, $3); }
28         | CONST             { $$ = nn($1, CONST, 0, 0); }
29         ;
30 %%
```

Parts of this specification will look familiar. Lines 23 to 29 define the structure of expressions, and lines 15 and 16 define the precedence rules. Line 12 defines `CONST` to be a token with an attribute of type `val`. Line 21 defines that, in this version, we expect a PROMELA `program` to consist of a single expression.

There are also some new phrases. Line 13, for instance, contains the definition for the internal representation of an expression: a data structure named `node`. We use *yacc* to build a parse tree of a PROMELA program, or in this case a parse tree of an expression. For example, parsing the expression `2+5*3` produces the tree shown in Figure 12.2. The parse tree defines the structure of the program and will help us later to determine how it is to be interpreted.

The *yacc* file is used to generate what is called an *LALR(1)* parser that can build the tree in Figure 12.2. The parser scans its input in one pass from *l*eft-to-right. It builds a *r*ightmost derivation in reverse, using at most *one l*ook-*a*head token. Each node in the parse tree is represented by a data structure that is defined in the include file `spin.h`, on lines 3 to 7. The structure is referred to in `spin.y` on lines 9 and 13.

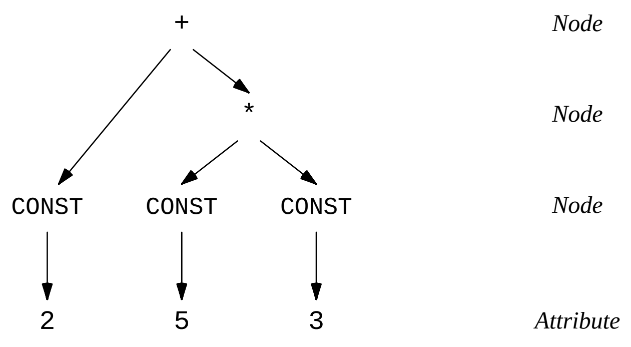

*Figure 12.2 — Parse Tree for* `2+5*3`

```
 1 /***** spine: spin.h *****/
 2
 3 typedef struct Node {
 4     int     nval;           /* value attribute     */
 5     short   ntyp;           /* node type           */
 6     struct Node *lft, *rgt; /* children parse tree */
 7 } Node;
 8
 9 extern Node *nn();          /* allocates nodes     */
10 extern char *emalloc();     /* allocates memory    */
11 extern void exit();
```

We have used only binary arithmetic operators so each node in the tree needs to point to at most two descendants. In `spin.h` these are called `lft` and `rgt`, two pointers to structures of type `Node`. The definition of a `Node` also contains a field for storing the node type, which can be an operator such as `'+'`, and `'*'`, or it can be a terminal node of type `CONST`. Terminal nodes, of course, have no descendants (cf. Figure 12.2), but they do have an attribute of type `nval` which is used to store the numerical value of the constant calculated by the lexical analyzer and passed on to the parser in the `yylval.val` field of the token (line 12 of `lex.l`).

Now, back to the parser. Whenever a subexpression is recognized it is remembered in a structure of type `Node`. In our example we use the function `nn()` to prepare such a structure. Its type is declared in `spin.h` on line 9. The definition of the procedure itself looks as follows:

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

The routine allocates memory for a new node in the parse tree, relying on `emalloc()` to check for error conditions and it returns a pointer to the initialized structure. For instance, when expression `5*3` is parsed, line 28 in `spin.y` is invoked twice, once for `5`, once for `3`. Each call produces a sub-expression of type `CONST` that is passed to line 26. The nodes of type `CONST` have no descendants, but the attribute fields are set to the value of the constant. The value produced by `lex.l` is available in a predefined parameter named `$1`, where the `1` refers to the first field in the production rule on line 28. In this case there is only one field on the right side of that rule, so `$1` is also the only valid parameter. The type of the field is an integer in this case, as defined on lines 8 and 12.

On line 26 a new node is constructed of type `'*'` with the two sub-expressions of type `CONST` as descendants. Arithmetical operators, such as `'*'`, are passed as literals from the lexical analyzer to the parser. The data structures representing the two sub-expressions for the multiplication are again passed by *yacc* in two parameters, named `$1` and `$3`. They point at the first and the third field of the production rule.

When the complete parse tree has been built it is passed to the production rule on line 21 and it can be interpreted. Here is the code for the interpreter.

```
eval(now)
        Node *now;
{
        if (now != (Node *) 0)
        switch (now->ntyp) {
        case CONST: return now->nval;
        case   '/': return (eval(now->lft) / eval(now->rgt));
        case   '*': return (eval(now->lft) * eval(now->rgt));
        case   '-': return (eval(now->lft) - eval(now->rgt));
        case   '+': return (eval(now->lft) + eval(now->rgt));
        default   : printf("spin: bad node type %d\n", now->ntyp);
                    exit(1);
        }
        return 0;
}
```

The default clause defends against unknown node types. The known types of nodes are evaluated recursively until the final answer is produced.

If we put all these pieces together we get the PROMELA expression evaluator. Here is a complete listing, together with the remaining routines that escaped mention above.

```
/***** spine: spin.h *****/

typedef struct Node {
    int     nval;           /* value attribute     */
    short   ntyp;           /* node type           */
    struct Node *lft, *rgt; /* children parse tree */
} Node;

extern Node *nn();          /* allocates nodes     */
extern char *emalloc();     /* allocates memory    */
extern void exit();

/***** spine: lex.l *****/

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

int lineno = 1;
%}

%%
[ \t]       { /* ignore white space */ }
[0-9]+      { yylval.val = atoi(yytext); return CONST; }
\n          { lineno++; }
.           { return yytext[0]; }

/***** spine: spin.y *****/

%{
#include "spin.h"           /* defines what Nodes are etc */
%}

%union{                     /* defines the data type of   */
    int val;                /* the internal parse stack   */
    Node *node;
}

%token      <val>   CONST   /* token CONST has a value    */
%type       <node>  expr    /* expressions produce nodes  */

%left       ’+’ ’-’         /* left associative operators */
%left       ’*’ ’/’         /* idem, highest precedence   */
%%

/** Grammar Rules **/

program : expr              { printf("= %d\n", eval($1)); }
            ;
expr    : ’(’ expr ’)’      { $$ = $2; }
        | expr ’+’ expr     { $$ = nn( 0, ’+’, $1, $3); }
        | expr ’-’ expr     { $$ = nn( 0, ’-’, $1, $3); }
        | expr ’*’ expr     { $$ = nn( 0, ’*’, $1, $3); }
        | expr ’/’ expr     { $$ = nn( 0, ’/’, $1, $3); }
        | CONST             { $$ = nn($1, CONST, 0, 0); }
        ;
%%

/***** spine: run.c *****/

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

eval(now)
    Node *now;
{
    if (now != (Node *) 0)
    switch (now->ntyp) {
    case CONST: return now->nval;
    case   ’/’: return (eval(now->lft) / eval(now->rgt));
    case   ’*’: return (eval(now->lft) * eval(now->rgt));
    case   ’-’: return (eval(now->lft) - eval(now->rgt));
    case   ’+’: return (eval(now->lft) + eval(now->rgt));
    default   : printf("spin: bad node type %d\n", now->ntyp);
                exit(1);
    }
    return 0;
}

/***** spine: main.c *****/

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

main()
{
    yyparse();
    exit(0);
}

yywrap()    /* a dummy routine */
{
    return 1;
}

yyerror(s1, s2)     /* called by yacc on syntax errors */
    char *s1, *s2;
{
    extern int lineno;
    char buf[128];
    sprintf(buf, s1, s2);
    printf("spine: line %d: %s\n", lineno, buf);
}

char *
```

```
emalloc(n)
{   extern char *malloc();  /* library functions */
    extern char *memset();

    char *tmp = malloc(n);
    if (!tmp)
    {       printf("spine: not enough memory\n");
            exit(1);
    }
    memset(tmp,0,n);        /* clear memory */
    return tmp;
}

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

To compile this set of programs, the following *makefile* can be used.

```
# ***** spine: makefile *****

CC=cc           # ANSI C compiler
CFLAGS=         # no flags yet
YFLAGS=-v -d -D # verbose, debugging
OFILES= spin.o lex.o main.o run.o

spine:  $(OFILES)
        $(CC) $(CFLAGS) -o spine $(OFILES)

%.o:    %.c spin.h      # all files depend on spin.h
        $(CC) $(CFLAGS) -c $%.c
```

The *makefile* defines which flags must be passed to *yacc* and *cc* and it records the dependencies among the source files. It states, for instance, that when `spin.h` changes, all object files must be recreated. The *makefile* is read by another UNIX utility called *make* to produce the executable program `spine`. Here is the dialogue that is printed on my system if this program is compiled and invoked with a sample expression.

```
$ make
yacc -v -d -D spin.y
cc -c y.tab.c
rm y.tab.c
mv y.tab.o spin.o
```

```
lex  lex.l
cc -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc -c main.c
cc -c run.c
cc -o spine spin.o lex.o main.o run.o
$ echo "2+5*3" | spine
= 17
$
```

The complete program must of course recognize quite a few other language features before it can simulate PROMELA programs. They can be grouped into five classes:

- ○ Variables
- ○ Statements
- ○ Control-flow constructs
- ○ Processes
- ○ Macro expansion

### VARIABLES (Section 12.4, page 255)

We have to consider variable declarations, assignments to variables and references variables, generally in expressions built from the full range of arithmetic and logical operators, plus the special operators `len` and `run`. We consider variables of the five basic data types `bit`, `bool`, `byte`, `short`, and `int`, plus the declaration of channel type identifiers with the keyword `chan`.

### STATEMENTS (Section 12.5, page 265)

There are two types of unconditional statements: assignments to variables and the print statement we added to the simulator. There are also five basic types of conditional statements: boolean conditions, `timeout`s, send and receive statements, and `assert`. The ‘‘pseudo-statement’’ `skip` can be implemented as the equivalent of the condition `(1)`.

### CONTROL FLOW (Section 12.6, page 275)

We have to consider the sequential control flow specifications: `goto`, `break`, selection, repetition, and `atomic` statements.

### PROCESSES AND MESSAGE TYPES (Section 12.7, page 282)

Most importantly, we have to implement the code for parsing and interpreting global declarations for both process types and message types.

### MACRO EXPANSION (Section 12.8, page 292)

One of the easier parts of the code. Macro expansion is achieved by routing the input to SPIN through the standard C preprocessor before parsing begins.

### TO THE BRAVE

Each of the next four sections focuses on one of these four extensions of the little program `spine`, that was discussed above. This discussion ultimately leads us to the full simulator source of SPIN as it is listed in Appendix D. The discussion below is primarily meant for the benefit of those who would like to expand, or modify the code, or the language that it parses. Feel free to skip sections or to move directly to a safer part of the chapter, such as Section 12.9 explaining the general use of the program. The brave who delve into the code may occasionally want to refer back to this overview or to the previews at the beginning of each section. To see the code in context they may also want to refer to the appendix. Appendix D includes an index of all the code, together with references to the page numbers that explain it.

## 12.4 VARIABLES

Most of the code that is required for the manipulation of variables is contained in two files. The first file, `sym.c`, contains the definition of a general symbol table handler. The second file, `vars.c`, contains the code for storing and manipulating variables of the basic data types. We do not worry too much about variables of type `chan` just yet. The bulk of that will come in the next extension when message passing is implemented. For now, we will just parse the declarations. To fully implement the other types of variables, we need some new hooks in the lexical analyzer, the parser and in the expression evaluation routine. The next four subsections, then, focus on these extensions:

- ○ Extensions to the lexical analyzer (Section 12.4.1, page 255)
- ○ New symbol table routines (Section 12.4.2, page 257)
- ○ Extensions to the parser (Section 12.4.3, page 260)
- ○ New code for the evaluator (Section 12.4.4, page 263)

We begin by taking a look at the changes that have to be made in the lexical analyzer.

### 12.4.1 LEXICAL ANALYZER

There is a range of new tokens that must be added to recognize variable names and declarations in the input to the simulator. There are also several PROMELA operators that consist of more than one character and are best recognized in the lexical analyzer and converted into tokens. Let's first look at variable declarations. Recognizing the five basic data types and the keyword `chan` produces the following extra rules in `lex.l`.

```
"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; }
```

Each declarator is passed as a token of type `TYPE`. The attribute is a constant, defined in `spin.h`, that specifies the width of each data type in bits (except for `CHAN`).

```
#define BIT      1              /* data types    */
#define BYTE     8              /* width in bits */
#define SHORT   16
#define INT     32
#define CHAN    64
```

The term `Token` is also defined in a macro

```
#define Token   if (!in_comment) return
```

which in combination with the following two lines

```
"/*"    { in_comment=1; }
"*/"    { in_comment=0; }
```

guarantees that no lexical tokens are passed to the parser within PROMELA comments. The two-character operators are recognized by the following new *lex* rules.

```
"<<"    { 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 */ }
```

Most single character operators, such as `'<'` or `'>'` are still passed by the last line of the rules section:

```
.       { Token yytext[0]; }
```

which was basically unmodified from the version used in `spine`. An exception is the assignment operator `'='`, which is converted into a real token, named `ASGN`. It is also given a value attribute that is set to the line number on which the assignment was found.

```
"="     { yylval.val = lineno; Token ASGN; }
```

To facilitate debugging, as many tokens as possible are tagged with a line number that refers back to the PROMELA source file. The above *lex* rule shows how that works for the assignment statement. The treatment of tokens for alphanumeric names is implemented slightly differently. To simplify the code that *lex* has to produce a little bit, we look up predefined alphanumeric names in a static table with procedure `check_name()`. The procedure is called from within the *lex* rule

```
[a-zA-Z_][a-zA-Z_0-9]* { Token check_name(yytext); }
```

An arbitrary name starts with an upper or lower case letter, and is followed by zero or more letters or digits. Underscores are allowed in names. To recognize the keywords `len`, `run`, and `of` (used in `chan` initializers), for instance, we can write:

```
static struct {
        char *s;        int tok;
} Names[] = {
        "len",          LEN,
        "run",          RUN,
        "of",           OF,
        0,              0,
};

check_name(s)
        char *s;
{
        register int i;
        for (i = 0; Names[i].s; i++)
                if (strcmp(s, Names[i].s) == 0)
                {       yylval.val = lineno;
                        return Names[i].tok;
                }
        yylval.sym = lookup(s); /* symbol table */
        return NAME;
}
```

Unrecognized names go through to the symbol table routines, all others come back immediately with a line number attached.

All new token names must be defined properly in the *yacc* grammar specification, but we will come to that later. Let us first look at the way in which names are stored in the symbol table. So far we have no separate processes, so all names are necessarily global. The routine `lookup()` is defined in a new file `sym.c` with a symbol table handler.

### 12.4.2 SYMBOL TABLE HANDLER

Each new name is stored in a data structure of type `Symbol` that is defined in the new version of `spin.h`.

```
typedef struct Symbol {
        char    *name;
        short   type;           /* variable or chan type      */
        int     nel;            /* 1 if scalar, >1 if array   */
        int     *val;           /* runtime value(s), initl 0  */
        struct Node     *ini;   /* initial value, or chan-def */
        struct Symbol   *next;  /* linked list */
} Symbol;
```

The structure `Symbol` contains the full name of the symbol being stored as a pointer to a character string. It also holds its type and the number of elements that are accessible if the name is defined to be an array. The pointer `ini` points to a parse tree fragment that can be evaluated to initialize a new identifier: a channel initializer for message channels, or an integer expression for variables of the five basic data types. The integer pointer `val` points to a location where the runtime values of global variables are stored.

The last element of the structure, `next`, points to another symbol. In its simplest form then, the symbol table can be implemented as a single linked list pointed to by

```
Symbol *symtab.
```

Initially the list is empty:

```
symtab = (Symbol *) 0.
```

Inserting a new symbol `sp` at the front list takes just two assignments in C:

```
sp->next = symtab;
symtab = sp;
```

Linked lists will come back a few more times in the other extensions we make. We use them, for instance, to implement message channels and for storing process references in the scheduler. Figure 12.3 illustrates how the linked list is used for the symbol table routines.

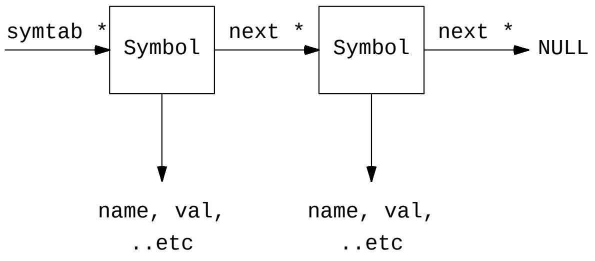

*Figure 12.3 — Linked List for the Symbol Table*

The complete lookup routine, using the linked list, can be written as follows:

```
Symbol *
lookup(s)
        char *s;
{
        Symbol *sp;

        /* check if symbol is already in the list */
        for (sp = symtab; sp; sp = sp->next)
                if (strcmp(sp->name, s) == 0)
                        return sp;      /* yes it is */

        /* if not, create a new symbol */
        sp = (Symbol *) emalloc(sizeof(Symbol));
        sp->name = (char *) emalloc(strlen(s) + 1);
        strcpy(sp->name, s);
        sp->nel = 1;            /* scalar */
        /* insert it in the list */
        sp->next = symtab;
        symtab = sp;

        return sp;
}
```

The routine `emalloc()` allocates and clears memory for us, so by default the type and the initial value of a new variable are both zero. The lookup routine is called by the lexical analyzer that, as defined, has not enough context to determine the type of a variable name or even if it is used as a scalar or as a vector. A separate routine `settype()` is called by the parser to initialize the type field of variables when that information has been collected. In the same check we can also make sure that an array was not accidentally given a negative dimension. The first argument to `settype()` is a linked list of names, with a pointer into the symbol table for each name.

```
settype(n, t)
        Node *n;
{
        while (n)
        {       if (n->nsym->type)
                  yyerror("redeclaration of ‘%s’", n->nsym->name);
                n->nsym->type = t;
                if (n->nsym->nel <= 0)
                  yyerror("bad array size for ‘%s’", n->nsym->name);
                n = n->rgt;
        }
}
```

The code maintains the same bookkeeping for all variables, scalars and arrays alike. A scalar is simply an array of size one.

Using only a single linked list to store all names, however, makes the lexical analyzer spend a disproportionate amount of time looking up variable names in the initial `for` loop of routine `lookup()`. For each new name, the routine would be forced to look at *all* previously entered names, before it can finally decide that a new symbol must be created. The longer the list, the more severe the time penalty becomes. A standard solution to this problem is to use a *hash-table* lookup scheme. We use the name of the symbol to calculate a unique index into an array of symbol tables (the hash-table), and store the symbol there. The average search time goes down linearly with the size of the hash table. It leads to the following version, with `Nhash` a constant defined in `spin.h`.
The value of `Nhash` must be of the type $2^n - 1$, with arbitrary `n`.

```
Symbol  *symtab[Nhash+1];
```

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

        return sp;
}
```

### 12.4.3 PARSER

Now let us look at the extensions that we must make in the parser to recognize variable names, declarations, and the new operators. First, recall that a token of type `NAME` has an attribute of type `Symbol`. We must therefore extend the definition of the parser's stack.

```
%union{
        int val;
        Node *node;
        Symbol *sym;
}
```

There are also some new token names and quite a few new precedence rules. Here is what we added.

```
%token  <val>   LEN OF
%token  <val>   CONST TYPE ASGN
%token  <sym>   NAME

%type   <sym>   var ivar
%type   <node>  expr var_list
%type   <node>  args arg typ_list

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

The assignment operator ASGN is right associative and gets the lowest precedence of all operators. The new token of type NAME has a symbol attribute pointing to a slot in the symbol table.
A sequence of zero or more variable declarations is parsed as follows:

```
any_decl: /* empty */          { $$ = (Node *) 0; }
        | one_decl ';' any_decl { $$ = nn(0, 0, ',', $1, $3); }
        ;
one_decl: TYPE var_list        { settype($2, $1); $$ = $2; }
        ;
```

As soon as a complete variable declaration is recognized, in the last production above, the routine settype() is called to store the extra type information in the symbol table. Before we look at the definition of a var_list, note that a variable name can be followed by an array index. Variables are therefore defined in the grammar as a non-terminal var, of type sym.

```
var     : NAME                 { $1->nel =  1; $$ = $1; }
        | NAME '[' CONST ']'   { $1->nel = $3; $$ = $1; }
```

For the time being we just remember the array size specified, and check it for its validity later when a procedure settype() is called. A variable can also have an initialization field. An initialized variable can be defined as a non-terminal ivar, as follows:

```
ivar    : var                   { $$ = $1; }
        | var ASGN expr         { $1->ini = $3; $$ = $1; }
        | var ASGN ch_init      { $1->ini = $3; $$ = $1; }
        ;
ch_init : '[' CONST ']' OF '{' typ_list '}'
                                { if ($2) u_async++; else u_sync++;
                                  cnt_mpars($6);
                                  $$ = nn(0, $2, CHAN, 0, $6);
                                }
        ;
```

The initializer can be either an expression returning a value or a channel specification. The production rules above allow both. Some statistics are gathered about the number of message parameters used and the number of synchronous and asynchronous channels that are declared. The list of data types in a channel initializer is also quickly defined.

```
typ_list: TYPE                  { $$ = nn(0, 0, $1, 0,  0); }
        | TYPE ',' typ_list     { $$ = nn(0, 0, $1, 0, $3); }
        ;
```

The check that, for instance, a channel is not initialized with an expression can be placed in the code that performs the actual initializations. The set of production rules that deals with variable declarations can be completed by defining

```
var_list: ivar                  { $$ = nn($1, 0, TYPE, 0,  0); }
        | ivar ',' var_list     { $$ = nn($1, 0, TYPE, 0, $3); }
        ;
```

The node allocation routine `nn()` must now also handle the new symbol-table references. It is extended as follows:

```
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;
}
```

Next, we have to prepare to parse the new operators that we have added. Most of it is straightforward. The series of production rules for expressions merely grows a bit.

```
expr    : '(' expr ')'          { $$ = $2; }
        | expr '+' expr         { $$ = nn(0, 0,  '+', $1, $3); }
        | expr '-' expr         { $$ = nn(0, 0,  '-', $1, $3); }
        ...
        | expr AND expr         { $$ = nn(0, 0,  AND, $1, $3); }
        | expr OR  expr         { $$ = nn(0, 0,   OR, $1, $3); }
        ...
        | expr LSHIFT expr      { $$ = nn(0, 0,LSHIFT,$1, $3); }
        | expr RSHIFT expr      { $$ = nn(0, 0,RSHIFT,$1, $3); }
        | '˜' expr              { $$ = nn(0, 0,  '˜', $2,  0); }
        | '-' expr %prec UMIN   { $$ = nn(0, 0, UMIN, $2,  0); }
        | '!' expr %prec NEG    { $$ = nn(0, 0,  '!', $2,  0); }
        | LEN '(' varref ')'    { $$ = nn($3->nsym,$1,LEN,$3,0); }
        | varref                { $$ = $1; }
        | CONST                 { $$ = nn(0, $1, CONST, 0, 0); }
        ;
```

Perhaps the most interesting additions are the new types of expressions for the bitwise, arithmetic and logical unary operations: bitwise complement `'˜'`, unary minus and boolean negation. Since the minus operator can also be used as a binary operator we have to set its precedence explicitly with the *yacc* keyword `%prec`. And, of course, the negation operator `'!'` will double in one of the next extensions as a binary send operator, so its precedence in this grammar rule is also explicitly set. In the final version we replace the character token `'!'` with a token `SND`, which is again labeled with a line number to improve error reporting.

The variable references, used in the last two production rules, are defined as follows:

```
varref  : NAME                  { $$ = nn($1, 0,  NAME,  0,  0); }
        | NAME '[' expr ']'     { $$ = nn($1, 0,  NAME, $3,  0); }
        ;
```

There are just two types of variable references, one for scalars and one for arrays. They both return a node of type `NAME` with the first field referring to the symbol that defines the variable name. The left pointer specifies the optional array index. A null pointer in place of an index trivially evaluates to zero. It should produce an error if we try to determine the ‘‘length’’ of anything other than a channel variable. The parser, however, does not check this.

### 12.4.4 EVALUATOR

The last routine we must look at for the addition of variables is the evaluator code in `run.c` Luckily, the extension is almost trivial. There are merely some new cases in the switch. For instance:

```
switch (now->ntyp) {
...
case     NE: return (eval(now->lft) != eval(now->rgt));
case     EQ: return (eval(now->lft) == eval(now->rgt));
case     OR: return (eval(now->lft) || eval(now->rgt));
case    AND: return (eval(now->lft) && eval(now->rgt));
case LSHIFT: return (eval(now->lft) << eval(now->rgt));
case RSHIFT: return (eval(now->lft) >> eval(now->rgt));
...
case  ASGN: return setval(now->lft, eval(now->rgt));
case  NAME: return getval(now->nsym, eval(now->lft));
...
```

The only interesting new cases are the variable references on the last two lines above. They are implemented with two procedure calls: `getval()` and `setval()`.

To process an assignment, first the value to be assigned is determined by evaluating the expression pointed to via `now->rgt`. Next, the target variable, and possibly its index, must be found. All the information is available via the left pointer, which is passed to `setval()`. A node of type `NAME` contains a pointer to the symbol table, with the alpha-numeric variable name, and it contains an index for array references or a null pointer for scalars. The routines `getval()` and `setval()` are defined in `vars.c`. For global variables, the only type of variables we have so far, the routine `setval()` hands the job to routine `setglobal()`, which checks the index and if necessary allocates memory for the variables.

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

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

with `checkvar()` defined as follows:

```
checkvar(s, n)
        Symbol *s;
{
        int i;

        if (n >= s->nel || n < 0)
        {       yyerror("array indexing error, ‘%s’", s->name);
                return 0;
        }
        if (s->type == 0)
        {       yyerror("undecl var ‘%s’ (assuming int)", s->name);
                s->type = INT;
        }
        if (s->val == (int *) 0)        /* uninitialized */
        {       s->val = (int *) emalloc(s->nel*sizeof(int));
                for (i = 0; i < s->nel; i++)
                {       if (s->type != CHAN)
                                s->val[i] = eval(s->ini);
                        else
                                s->val[i] = qmake(s);
        }       }
        return 1;
}
```

A plain variable is initialized by evaluating the expression in the initialization field of the symbol. A null pointer in this field, again, trivially evaluates to the default initial value zero. A channel variable is initialized by passing the initialization pointer to a routine `qmake()` that is discussed in the next section. A type clash between initializer and variable triggers an error in either the evaluator or in the channel building routine.

The routine `getval()` hands off to `getglobal()`.

```
getglobal(s, n)
        Symbol *s;
{
        if (checkvar(s, n))
                return cast_val(s->type, s->val[n]);
        return 0;
}
```

A zero result is returned if the `checkvar` routine fails, that is, if an indexing error was detected. The routine `cast_val()` interprets the type of the variable and masks or casts variable values accordingly.

```
cast_val(t, v)
{       int i=0; short s=0; unsigned char u=0;

        if (t == INT || t == CHAN) i = v;
        else if (t == SHORT) s = (short) v;
        else if (t == BYTE)  u = (unsigned char)v;
        else if (t == BIT)   u = (unsigned char)(v&1);

        if (v != i+s+u)
                yyerror("value %d truncated in assignment", v);
        return (int)(i+s+u);
}
```

It is considered an error if a value changes due to type casting. Variables are only cast to the right value when they are read. Internally, all values are stored in 32-bit integers.

## 12.5 STATEMENTS

Adding statements is relatively easy at this point. We discuss four sets of extensions:

- ○ Extensions to the lexical analyzer (Section 12.5.1, page 266)
- ○ Extensions to the parser (Section 12.5.2, page 266)
- ○ Extensions to the evaluation routines (Section 12.5.3, page 267)

○ The implementation of message passing (Section 12.5.4, page 268)

To implement message passing statements we add a new file `mesg.c`. We begin by adding five types of statements: boolean conditions, `timeout`s, assignments, `printf`, and `assert` statements, plus the pseudo-statement `skip`.

### 12.5.1 LEXICAL ANALYZER

Three of the statements, `assert`, `printf`, and `timeout`, produce new entries into the static lookup table of alphanumeric names in the lexical analyzer.

```
static struct {
        char *s;        int tok;
} Names[] = {
        "assert",       ASSERT,
        "printf",       PRINT,
        "timeout",      TIMEOUT,
        ...
};
```

The line number attached to the tokens by the routine `checkname()` allows us to produce the right feedback to a user when an `assert` statement fails during a simulation run.

The pseudo statement `skip` is translated into the equivalent constant with the *lex* rule

```
"skip"          { yylval.val = 1; return CONST; }
```

To implement the `printf` statement properly, we must define strings. The only place where string arguments are used is in `printf`'s, so we can treat it as a special token with a symbol attribute. The symbol pointer than holds the string, rather than the variable name. This produces one more `lex` rule.

```
\".*\"          { yylval.sym = lookup(yytext); return STRING; }
```

Finally, we add a rule for translating arrows into semicolons.

```
"->"            { return ';'; /* statement separator */ }
```

### 12.5.2 PARSER

To update the parser we must add some new production rules again for parsing statements. So far, a statement can be an assignment, a print statement, an assertion, a jump, or an expression (a condition). The `timeout` is implemented as a special predefined variable that can be part of a condition. It is added in the code for parsing expressions.

```
stmnt   : varref ASGN expr      { $$ = nn($1->nsym,$2, ASGN,$1,$3); }
        | PRINT '(' STRING prargs ')' { $$=nn($3, $1, PRINT,$4, 0); }
        | ASSERT expr           { $$ = nn(0, $1, ASSERT, $2, 0); }
        | GOTO NAME             { $$ = nn($2,$1,   GOTO,  0, 0); }
        | expr                  { $$ = nn(0,lineno, 'c', $1, 0); }
        ...
expr    : TIMEOUT               { $$ = nn(0, $1,TIMEOUT,  0, 0); }
        ...
```

We have made a separate production rule for parsing variable references, named `varref`. It can be used in a few more cases later.

A *sequence* of statements is defined as follows:

```
sequence: step                  { add_seq($1); }
        | sequence ';' step     { add_seq($3); }
        ;
step    : any_decl stmnt        { $$ = $2; }
        ;
```

There is no need to group declarations at the start of a program body in PROMELA, so in the rules above we have allowed each statement to be preceded by one or more declarations. The routine `add_seq()` is defined in `flow.c` to tag statements onto a linked list, but we look at that in more detail in Section 12.6. For now, a program body is just a sequence of statements, which is parsed as follows:

```
body    : '{'                   { open_seq(1); }
             sequence           { add_seq(Stop); }
          '}'                   { $$ = close_seq(); }
        ;
```

The first open brace starts a new linked list to store the statements via a call on routine `open_seq()`. When a complete sequence is recognized a special `Stop` node is tagged onto the end and the sequence is closed and passed up in the parse tree. We look at the routines for manipulating the sequences in `flow.c` in more detail later when we discuss compound statements. Let us first now consider the additions we have to make in the interpreter code to evaluate the statements added so far.

### 12.5.3 EVALUATOR

The additions are still modest. The interpreter only has to handle these new node types.

```
case TIMEOUT: return Tval;
case     'c': return eval(now->lft); /* condition */
case   PRINT: return interprint(now);
case  ASSERT: if (eval(now->lft)) return 1;
              yyerror("assertion violated", (char *)0);
              wrapup(); exit(1);
```

Timeouts are a modeling feature of PROMELA; they are implemented as a test on a predefined variable here. In the final simulator the scheduler can explicitly enable or disable timeout events to test if the protocol can recover from exception conditions. Conditions, identified by the internal node type `'c'`, are evaluated recursively and return a zero or non-zero status. The `assert` statement is interpreted directly and causes an error exit if it fails, printing the line number that was duly carried along as a token attribute. We have moved the grubby details of the implementation of the print statements into a separate procedure `interprint()`. It can be defined as follows:

```
interprint(n)
        Node *n;
{
        Node *tmp = n->lft;
        char c, *s = n->nsym->name;
        int i, j;

        for (i = 0; i < strlen(s); i++)
                switch (s[i]) {
                default:   putchar(s[i]); break;
                case '\"': break; /* ignore */
                case '\\':
                         switch(s[++i]) {
                         case 't': putchar('\t'); break;
                         case 'n': putchar('\n'); break;
                         default:  putchar(s[i]); break;
                         }
                         break;
                case  '%':
                         if ((c = s[++i]) == '%')
                         {      putchar('%'); /* literal */
                                break;
                         }
                         if (!tmp)
                         {      yyerror("too few print args %s", s);
                                break;
                         }
                         j = eval(tmp->lft);
                         tmp = tmp->rgt;
                         switch(c) {
                         case 'c': printf("%c", j); break;
                         case 'd': printf("%d", j); break;
                         case 'o': printf("%o", j); break;
                         case 'u': printf("%u", j); break;
                         case 'x': printf("%x", j); break;
                         default:  yyerror("unrecognized print cmd %%'%c'", c);
                                   break;
                         }
                         break;
                }
        fflush(stdout);
        return 1;
}
```

which recognizes a modest number of conversions of the UNIX library function `printf()`.

### 12.5.4 IMPLEMENTING MESSAGE PASSING

The synchronous and asynchronous message passing primitives are good for another two to three hundred lines of source text. The easiest part is the extension of the parser to pass the new statements on to the interpreter.

The sending and receiving get a relatively low evaluation priority, equivalent to assignment. Three new types of statements are added.

```
stmnt   : ...
        | varref RCV margs      { $$ = nn($1->nsym, $2, 'r',$1,$3); }
        | varref SND margs      { $$ = nn($1->nsym, $2, 's',$1,$3); }
```

with message arguments defined as follows:

```
margs   : arg                   { $$ = $1; }
        | expr '(' arg ')'      { $$ = nn(0, 0, ',', $1, $3); }
        ;
arg     : expr                  { $$ = nn(0, 0, ',', $1,  0); }
        | expr ',' arg          { $$ = nn(0, 0, ',', $1, $3); }
        ;
```

The arguments of a receive operation can only be constants or names, but it is easier to verify that part of the syntax later. There is also one new type of expression

```
expr    : ...
        | varref RCV '[' margs ']' { $$ = nn($1->nsym,$2, 'R', $1, $4); }
```

which corresponds to a side-effect free test of the executability of a receive statement (see Chapter 5).

The interpreter now has three extra clauses for the new internal node types `'r'`, `'s'`, and `'R'`. This corresponds to the following code in `run.c`.

```
        case LEN:       return qlen(now);
        case 's':       return qsend(now);
        case 'r':       return qrecv(now, 1); /* full-receive */
        case 'R':       return qrecv(now, 0); /* test only    */
```

The three procedures called here, together with the procedure for the initialization of new channels `qmake()` mentioned in passing before, are expanded in the file `mesg.c`. Let us first look at the implementation of `qmake()`.

The descriptions of the message channels are stored in a linked list of structures of type `Queue`, using the following definition from `spin.h`.

```
typedef struct Queue {
        short   qid;            /* runtime q index      */
        short   qlen;           /* nr messages stored   */
        short   nslots, nflds;  /* capacity, flds/slot  */
        short   *fld_width;     /* type of each field   */
        int     *contents;      /* the actual buffer    */
        struct Queue    *nxt;   /* linked list */
} Queue;
```

In `mesg.c` the head of the list is defined like this.

```
Queue *qtab = (Queue *) 0;       /* linked list */
int nqs = 0;                     /* number of queues   */
```

The rest is easy. We give zero length (rendezvous) channels one slot to temporarily hold a message as it is passed from a sender to a receiver.

```
qmake(s)
        Symbol *s;
{
        Node *m;
        Queue *q;
        int i; extern int analyze;

        if (!s->ini)
                return 0;
        if (s->ini->ntyp != CHAN)
                fatal("bad channel initializer for %s\n", s->name);
        if (nqs >= MAXQ)
                fatal("too many queues (%s)", s->name);

        q = (Queue *) emalloc(sizeof(Queue));
        q->qid = ++nqs;
        q->nslots = s->ini->nval;
        for (m = s->ini->rgt; m; m = m->rgt)
                q->nflds++;
        i = max(1, q->nslots);  /* 0-slot qs get 1 slot minimum */

        q->contents  = (int *) emalloc(q->nflds*i*sizeof(int));
        q->fld_width = (short *) emalloc(q->nflds*sizeof(short));
        for (m = s->ini->rgt, i = 0; m; m = m->rgt)
                q->fld_width[i++] = m->ntyp;
        q->nxt = qtab;
        qtab = q;
        ltab[q->qid-1] = q;

        return q->qid;
}
```

Of course, a channel can only be created if an initializer is provided. It is a fatal error if the initializer has the wrong type or if too many channels already exist. For efficiency only, an index to all active channels is also maintained in a linear list named `ltab`. Implementing `qlen()` is now straightforward.

```
qlen(n)
        Node *n;
{
        int whichq = eval(n->lft)-1;

        if (whichq < MAXQ && whichq >= 0 && ltab[whichq])
                return ltab[whichq]->qlen;
        return 0;
}
```

We check that the index calculated is within the correct range, that the corresponding queue exists, and return the length field from the corresponding data structure. The hard part remains: the implementation of synchronous and asynchronous versions of `qsend()` and `qrecv()`. We use a simple interface to decide which routine to use.

```
qsend(n)
        Node *n;
{
        int whichq = eval(n->lft)-1;
        if (whichq == -1)
        {       printf("Error: sending to an uninitialized chan\n");
                whichq = 0;
        }
        if (whichq < MAXQ && whichq >= 0 && ltab[whichq])
        {       if (ltab[whichq]->nslots > 0)
                        return a_snd(ltab[whichq], n);
                else
                        return s_snd(ltab[whichq], n);
        }
        return 0;
}
qrecv(n, full)
        Node *n;
{
        int whichq = eval(n->lft)-1;

        if (whichq == -1)
        {       printf("Error: receiving from an uninitialized chan\n");
                whichq = 0;
        }
        if (whichq < MAXQ && whichq >= 0 && ltab[whichq])
                return a_rcv(ltab[whichq], n, full);
        return 0;
}
```

A rendezvous is triggered by the sender of a message. If, at that time, at least one process is blocked on the matching receive operation, a rendezvous can take place. The rendezvous receive operation, therefore, can be the same for both synchronous and asynchronous operations. First we consider the asynchronous case. Its basic form looks as follows (the version in Appendix D contains a few more features that are not relevant here):

```
a_snd(q, n)
        Queue *q;
        Node *n;
{
        Node *m;
        int i = q->qlen*q->nflds;       /* q offset */
        int j = 0;                      /* q field# */

        if (q->nslots > 0 && q->qlen >= q->nslots)
                return 0;       /* q is full */

        for (m = n->rgt; m && j < q->nflds; m = m->rgt, j++)
          q->contents[i+j] = cast_val(q->fld_width[j], eval(m->lft));
        q->qlen++;
        return 1;
}
```

If the channel is full, a zero status is returned, which means that the statement is currently unexecutable. If there is at least one free slot in the queue, the fields are copied and masked with the predefined field widths. The receive operation is a little more involved.

```
a_rcv(q, n, full)
        Queue *q;
        Node *n;
{
        Node *m;
        int j, k;

        if (q->qlen == 0) return 0;     /* q is empty */

        for (m = n->rgt, j=0; m && j<q->nflds; m = m->rgt, j++)
        {       if (m->lft->ntyp == CONST
                &&  q->contents[j] != m->lft->nval)
                        return 0;       /* no match */
        }
        for (m = n->rgt, j=0; j<q->nflds; m = (m)?m->rgt:m, j++)
        {       if (m && m->lft->ntyp == NAME)
                        setval(m->lft, q->contents[j]);
                for (k = 0; full && k < q->qlen-1; k++)
                        q->contents[k*q->nflds+j] =
                        q->contents[(k+1)*q->nflds+j];
        }
        if (full) q->qlen--;
        return 1;
}
```

The argument `full` is zero when the receive operation is used as a condition, as in `qname?[ack]`, and non-zero otherwise. In the first case the procedure has no side-effects and merely returns the executability status of the receive. There are two `for` loops in the procedure. The first one checks that all message parameters that are declared as constants are properly matched and it checks that the other parameters are variable names. The second cycle copies the data from the queue into the variables specified and, as required, shifts message fields up one slot.

Only the synchronous version of the send operation remains to be expanded.

```
s_snd(q, n)
        Queue *q;
        Node *n;
{
        Node *m;
        int i, j = 0;   /* q field# */
        for (m = n->rgt; m && j < q->nflds; m = m->rgt, j++)
          q->contents[j] = cast_val(q->fld_width[j], eval(m->lft));

        q->qlen = 1;
        if (complete_rendez())
                return 1;
        q->qlen = 0;
        return 0;
}
```

The sender first appends the message and then checks if there is a receiver that can execute the matching receive. There is no need to check the queue length here: when the send operation is executable, all rendezvous channels are guaranteed to be empty. If no matching receive is found, the send operation fails, cancels the message in the queue and returns a zero. If there is a matching receive operation, it will execute before the routine `complete_rendez()` returns, and both sender and receiver proceed to the next statement, again leaving the channel empty. The routine `complete_rendez()` is in fact a tiny portion of the scheduler and is listed in `sched.c`. It blocks all statements except a synchronous receive and the code that is required for evaluating expressions.

```
complete_rendez()
{       RunList *orun = X;
        Element *e;
        int res=0;

        Rvous = 1;
        for (X = run; X; X = X->nxt)
                if (X != orun && (e = eval_sub(X->pc)))
                {       X->pc = e;
                        res = 1;
                        break;
                }
        Rvous = 0;
        X = orun;
        return res;
}
```

The routine first sets a global flag `Rvous` to make sure that only receive operations are enabled. The variable `X` is then pointed to every runnable process to check if it can complete the rendezvous handshake. The evaluation routine that looks into every option of a compound for a possible match is called `eval_sub()`.

```
Element *
eval_sub(e)
        Element *e;
{
        Element *f, *g;
        SeqList *z;
        int i, j, k;
        ...
        if (e->sub)
        {       for (z = e->sub, j=0; z; z = z->nxt)
                        j++;
                k = rand()%j;   /* nondeterminism */
                for (i = 0, z = e->sub; i < j+k; i++)
                {       if (i >= k && f = eval_sub(z->this->frst))
                                return f;
                        z = (z->nxt)?z->nxt:e->sub;
                }
        } else
        {       if (e->n->ntyp == ATOMIC)
                {       ...
                } else if (Rvous)
                {       if (eval_sync(e->n))
                                return e->nxt;
                } else
                        return (eval(e->n))?e->nxt:(Element *)0;
        }
        return (Element *)0;
}
```

The evaluation routine recursively searches through the options of compounds. If there is more than one option, the scheduler picks one at random, using the library routine `rand()`. When, at the lowest level in the recursion, it finds a statement instead of a compound to evaluate, it checks for the value of `Rvous` and calls this routine when it is set.

```
eval_sync(now)
        Node *now;
{       /* allow only synchronous receives
        /* and related node types    */

        if (now)
        switch (now->ntyp) {
        case TIMEOUT:   case PRINT:     case ASSERT:
        case RUN:       case LEN:       case 's':
        case 'c':       case ASGN:      case BREAK:
        case IF:        case DO:        case '.':
                return 0;
        case 'R':
        case 'r':
                if (!q_is_sync(now))
                        return 0;
        }
        return eval(now);
}
```

We will come back to the details later when we look more closely at the scheduler code itself.

## 12.6 CONTROL FLOW

Our next job is to bring some structure into the language by implementing the selection, repetition, `break`, `goto` and `atomic` statements. We have to build up a program as a coherent set of statements, with a control flow discipline that defines which statement from this set is to be evaluated by the scheduler at each execution step. All routines that deal explicitly with the control flow are placed in the file `flow.c`. This section discusses:

- ○ Code for manipulating sequences (Section 12.6.1, page 275)
- ○ Keeping track of labels (Section 12.6.2, page 278)
- ○ Code for parsing compound statements (Section 12.6.3, page 279)

### 12.6.1 SEQUENCES

Recall the definition of the non-terminal `program` in `spin.y`, which we discussed before.

```
body:
         '{'                    { open_seq(1); }
            sequence            { add_seq(Stop); }
         '}'                    { $$ = close_seq(); }
sequence:
          step                  { add_seq($1); }
        | sequence ';' step     { add_seq($3); }
```

A program body is a sequence of statements terminated by a special `Stop` node. But this time we have to carry around some extra information for compound statements. A compound statement is basically a fork in the execution sequence, where one sequence divides into a number of option sequences. We store each individual sequence in a structure of type `Sequence` defined in `spin.h`.

```
typedef struct Sequence {
        Element *frst;
        Element *last;
} Sequence;
```

A set of sequences is stored as a linked list, as follows:

```
typedef struct SeqList {
        Sequence        *this;  /* one sequence */
        struct SeqList  *nxt;   /* linked list  */
} SeqList;
```

And, of course the nodes of the parse tree will have to accommodate the new information, so the data structure for a `Node` is expanded somewhat more.

```
typedef struct Node {
        int     nval;           /* value attribute            */
        short   ntyp;           /* node type                  */
        Symbol  *nsym;          /* new attribute              */
        Symbol  *fname;         /* filename of src            */
        struct SeqList  *seql;  /* list of sequences          */
        struct Node     *lft, *rgt; /* children in parse tree */
} Node;
```

Statements are added one by one to a sequence with `add_seq()`. The statements are most conveniently stored in structures of type `Element`. The definition in `spin.h` looks as follows:

```
typedef struct Element {
        Node    *n;             /* defines the type & contents */
        int     seqno;          /* uniquely identifies this el */
        unsigned char   status; /* used by analyzer generator  */
        struct SeqList  *sub;   /* subsequences, for compounds */
        struct Element  *nxt;   /* linked list */
} Element;
```

Each element in a sequence is labeled with a unique sequence (or state) number, that will prove to be useful in building the validator in the next chapter. The numbers are handed out by routine `new_el()`.

```
Element *
new_el(n)
        Node *n;
{
        Element *m;

        if (n && (n->ntyp == IF || n->ntyp == DO))
                return if_seq(n->seql, n->ntyp, n->nval);
        m = (Element *) emalloc(sizeof(Element));
        m->n = n;
        m->seqno = Elcnt++;
        return m;
}
```

The `sub` field of an element points to the options of a compound. It is set, only for those types of statements, in routine `if_seq()`, which is examined in detail in Section 12.6.3.
The sequence numbers are kept in a global counter `Elcnt` that is reset to one at the start of every new process. In the code below this happens when `open_seq()` is called with a non-zero argument.
The open brace of a `body` initializes a new sequence by a call on procedure `open_seq()`. The closing brace closes the sequence and passes it via `close_seq()`. Initializing a new sequence of elements or returning a completed one is fairly straightforward. In its simplest form it looks like this.

```
void
open_seq(top)
{       SeqList *t;
        Sequence *s = (Sequence *) emalloc(sizeof(Sequence));

        t = seqlist(s, cur_s);
        cur_s = t;
        if (top) Elcnt = 1;
}

Sequence *
close_seq()
{       Sequence *s = cur_s->this;

        cur_s = cur_s->nxt;
        return s;
}
```

The listing in Appendix D performs some extra checks that are relevant only to the validator.

The routine `seqlist()` attaches a new sequence to a linked list of sequences.

```
SeqList *
seqlist(s, r)
        Sequence *s;
        SeqList *r;
{
        SeqList *t = (SeqList *) emalloc(sizeof(SeqList));
        t->this = s;
        t->nxt = r;
        return t;
}
```

Adding an element to the current sequence happens as follows:

```
add_seq(n)
        Node *n;
{
        Element *e;
        if (!n) return;
        innermost = n;
        e = colons(n);
        if (innermost->ntyp != IF && innermost->ntyp != DO)
                add_el(e, cur_s->this);
}
```

The routine that allocates memory for a new element `new_el()` filters out the compound statements and does all the hard work for them, so that need not be repeated above. The routine `add_el()` which is used here is not too exciting.

```
add_el(e, s)
        Element *e;
        Sequence *s;
{
        if (!s->frst)
                s->frst = e;
        else
                s->last->nxt = e;
        s->last = e;
}
```

### 12.6.2 JUMPS AND LABELS

Routine `add_seq()` also catches labels, identified by a node type `’:’`, and remembers them in another linked list.

```
typedef struct Label {
        Symbol  *s;
        Symbol  *c;
        Element *e;
        struct Label    *nxt;
} Label;
```

The code is again straightforward.

```
set_lab(s, e)
        Symbol *s;
        Element *e;
{
        Label *l; extern Symbol *context;
        if (!s) return;
        l = (Label *) emalloc(sizeof(Label));
        l->s = s;
        l->c = context;
        l->e = e;
        l->nxt = labtab;
        labtab = l;
}
```

When the scheduler has to determine the destination of a `goto` jump, it consults that list, and retrieves a pointer to the element that carried the label.

```
Element *
get_lab(s)
        Symbol *s;
{
        Label *l;
        for (l = labtab; l; l = l->nxt)
                if (s == l->s)
                        return (l->e);
        fatal("undefined label %s", s->name);
        return 0;       /* doesn’t get here */
}
```

The routine is called at the top of every invocation of the generic evaluation routine `eval_sub()`, as follows:

```
          if (e->n->ntyp == GOTO)
                  return get_lab(e->n->nsym);
```

Of course, to get the `goto` statements and labels at the right places into the parse tree, we must add a few more rules to the *lex* and *yacc* files. The lookup table in `lex.l` is expanded with

```
          "goto",         GOTO,
```

while adding a token `GOTO` to `spin.y`. The two production rules that recognize jumps and labels are:

```
          | GOTO NAME             { $$ = nn($2, $1, GOTO, 0, 0); }
          | NAME ':' stmnt        { $$ = nn($1, $3->nval, ':', $3, 0); }
```

The line number for a labeled statement is extracted from the node that is passed up through the third parameter `$3.` The `GOTO` token carries its own line number that is copied from `$1`.

### 12.6.3 COMPOUND STATEMENTS

The real challenge is to process the compound statements. There are a few new tokens to be handled, such as `if`, `fi`, `do`, `od`, `::`, `break`, and `atomic`. The lexical analyzer is again extended with one line for each. Three new statement types and two new production rules are added to the production rules in `spin.y`.

```
stmnt   : ...
        | IF options FI { $$ = nn(0, $1, IF, 0, 0);
                          $$->seql = $2;
                        }
        | DO            { pushbreak(); }
          options OD    { $$ = nn(0, $1, DO, 0, 0);
                          $$->seql = $3;
                        }
        | BREAK         { $$ = nn(break_dest(),$1,GOTO,0,0); }
        | ATOMIC
          '{'           { open_seq(0); }
             sequence
          '}'           { $$ = nn(0,$1, ATOMIC, 0, 0);
                          $$->seql = seqlist(close_seq(), 0);
                          make_atomic($$->seql->this);
                        }
        ;
options : option        { $$ = seqlist($1, 0); }
        | option options { $$ = seqlist($1, $2); }
        ;
option  : SEP           { open_seq(0); }
          sequence      { $$ = close_seq(); }
        ;
```

Every `option` in a compound statement can be a sequence of statements and is again captured in a data structure of type `Sequence`. Multiple options are again grouped into a linked list of sequences of the type `SeqList` that was defined before.

Repetition statements can be terminated with `break` statements. To keep track of the proper destinations we use `pushbreak()` to push an internal label onto a stack for every new repetition structure that is entered, and a routine `breakdest()` to retrieve the current destination of a `break` statement, much as with the labels and `goto` jumps we discussed earlier.

```
typedef struct Lbreak {
        Symbol  *l;
        struct Lbreak   *nxt;
} Lbreak;

pushbreak()
{       Lbreak *r = (Lbreak *) emalloc(sizeof(Lbreak));
        Symbol *l;
        char buf[32];

        sprintf(buf, ":b%d", break_id++);
        l = lookup(buf);
        r->l = l;
        r->nxt = breakstack;
        breakstack = r;
}

Symbol *
break_dest()
{       if (!breakstack)
                fatal("misplaced break statement", (char *)0);
        return breakstack->l;
}
```

A `break` statement, if it occurs, is translated with a call on `break_dest()` into a jump to the last `break` statement that was pushed onto the stack.

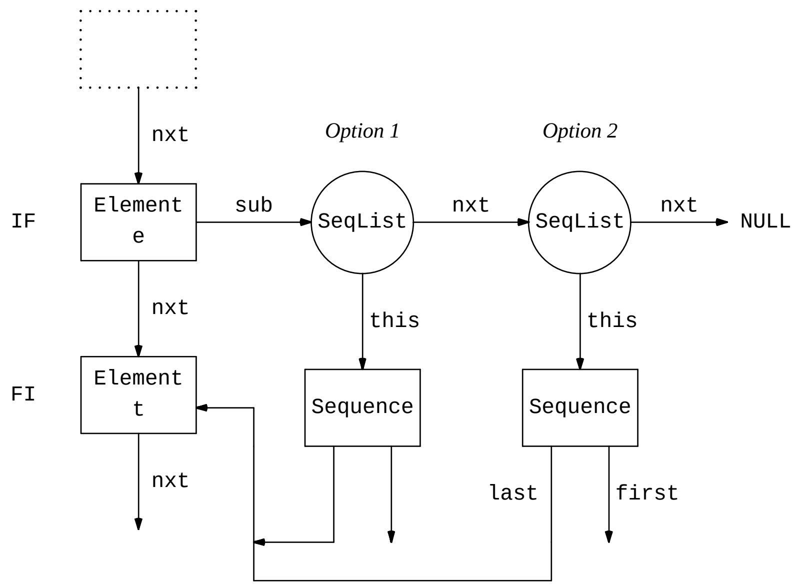

*Figure 12.4 — Parse Tree for a Selection Structure*

Now it is time to turn to one of the toughest procedures for parsing the compound statements: `if_seq()`. A selection or repetition structure must push several elements into the statement sequence. Figure 12.4 illustrates the node structure that is built by the procedure `if_seq()` to include selection statements into a sequence. Only compound statements (selections and repetitions) attach any nodes to the `sub` field of an `Element`. That field of sub-sequences starts a linked list (a `SeqList`) of options, with one complete `Sequence` structure per option in the compound statement. Figure 12.4 shows a selection statement with two options.

The `last` pointer of the sub sequence that defines an option is connected to the element that immediately follows the one that contains the original `sub` field. In the figure this is the `Element` labeled `t` (for target). It formalizes that a selection sequence is terminated whenever an option terminates.

The structure build for a repetition statement is almost the same, with just one exception: the `last` field of each sub sequence is now pointed at an `Element` that is placed immediately preceding the compound, at the place of the dotted box in Figure 12.4. It formalizes that a repetition structure is repeated when an option terminates. The `break` statement in the repetition structure will still point to the target `Element t`. The code that makes all this happen looks as follows:

```
Element *
if_seq(s, tok, lnno)
        SeqList *s;
{
        Element *e = new_el((Node *) 0);
        Element *t = new_el(nn((Symbol *) 0, lnno, '.',
                        (Node *)0, (Node *)0)); /* target */
        SeqList *z;

        e->n = nn((Symbol *)0, lnno, tok, (Node *)0, (Node *)0);
        e->sub = s;
        for (z = s; z; z = z->nxt)
                add_el(t, z->this);
        if (tok == DO)
        {       add_el(t, cur_s->this);
                t = new_el(nn((Symbol *)0, lnno, BREAK, (Node *)0, (Node *)0));
                set_lab(break_dest(), t);
                breakstack = breakstack->nxt;   /* pop stack */
        }
        add_el(e, cur_s->this);
        add_el(t, cur_s->this);
        return e;       /* destination node for label */
}
```

## 12.7 PROCESSES AND MESSAGE TYPES

The main thing missing from our simulator at this point is the concept of a process. With that extension we can place some finishing touches on the software by also coding the scheduler, by adding the distinction between local and global variables. The code for the scheduler is confined to a file named `sched.c`. The extension of the lexical analyzer is minimal at this point: the mere addition of the keywords `proctype`, `init`, and `mtype`. The other extensions are more substantial:

- Extensions to the parser (Section 12.7.1, page 282)
- The process scheduler (Section 12.7.2, page 285)
- Interpreting local variables (Section 12.7.3, page 289)

### 12.7.1 PARSER

A complete PROMELA program is constructed from a series of program units defined as follows:

```
program : units                 { sched(); }
        ;
units   : unit | units unit
        ;
unit    : proc
        | init
        | claim
        | one_decl
        | mtype
        ;
```

```
proc    : PROCTYPE NAME         { context = $2; }
                '(' decl ')'
          body                  { ready($2, $5, $7);
                                  context = (Symbol *) 0;
                                }
        ;
mtype   : MTYPE ASGN '{' args '}' { setmtype($4); }
        | ';'   /* optional ; as separator of units */
        ;
        ...
decl    : /* empty */           { $$ = (Node *) 0; }
        | decl_lst              { $$ = $1; }
        ;
decl_lst: one_decl              { $$ = nn(0, 0, ',', $1,  0); }
        | one_decl ';' decl_lst { $$ = nn(0, 0, ',', $1, $3); }
        ;
```

A `unit` is either a process declaration, a list of message types, a temporal claim, or an `init` specification, each of which can be preceded by one or more global variable declarations. The `init` module and the temporal claims are defined as special type of processes:

```
init    : INIT                  { context = $1; }
          body                  { runnable($3, $1);
                                  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;
                                }
        ;
```

There can be only one `init` and one temporal claim per specification. The first is required, the second optional. The presence of a claim is flagged in the global pointer `claimproc`.

Process declarations are placed in a ready queue of process bodies. To allow us to remember type declarations of formal parameters, declarations must now return a node containing the parameter list. Before the `body` of a process declaration is parsed, though, a context variable is set to identify any variable names and parameters to be recognized as local to the process declaration. Initially, only the `init` process is labeled runnable. Procedure `setmtype()` logically belongs in `sym.c` and can be implemented as follows:

```
Node *Mtype = (Node *) 0;
```

```
void
setmtype(m)
        Node *m;
{
        Node *n = m;
        if (Mtype)
                yyerror("mtype redeclared", (char *)0);

        Mtype = n;
        while (n)       /* syntax check */
        {       if (!n->lft || !n->lft->nsym
                ||  (n->lft->ntyp != NAME)
                ||   n->lft->lft)       /* indexed variable */
                        fatal("bad mtype definition", (char *)0);
                n = n->rgt;
        }
}
Node *Symnode = 0;

void
syms(m)
        Node *m;
{
        if (Symnode)
                yyerror("Duplicate Symmetry definition", (char *)0);

        Symnode = m;
}
```

The procedure merely checks syntax and stores the arguments for later processing. The message-type definitions can be hidden completely from the rest of the program if we let the lexical analyzer check the list whenever it sees a `NAME` and map all message names found there onto constants. We can do that in procedure `check_name()`.

```
check_name(s)
        char *s;
{
        register int i;
        for (i = 0; Names[i].s; i++)
                if (strcmp(s, Names[i].s) == 0)
                {       yylval.val = lineno;
                        return Names[i].tok;
                }
        if (yylval.val = ismtype(s))
                return CONST;
        yylval.sym = lookup(s); /* symbol table */
        return NAME;
}
```

The routine `ismtype()` looks up names in the list of message types.

```
ismtype(str)
        char *str;
{
        Node *n;
        int cnt = 1;

        for (n = Mtype; n; n = n->rgt)
        {       if (strcmp(str, n->lft->nsym->name) == 0)
                        return cnt;
                cnt++;
        }
        return 0;
}
```

### 12.7.2 SCHEDULER

The procedures `ready()` and `runnable()` require little imagination; they need merely store their arguments in linked lists where the scheduler can find them. The list of runnable processes is defined as follows:

```
typedef struct ProcList {
        Symbol  *n;             /* name       */
        Node    *p;             /* parameters */
        Sequence *s;            /* body       */
        struct ProcList *nxt;   /* linked list */
} ProcList;
```

And the routine that fills the list is

```
runnable(s, n)
        Sequence *s;    /* body */
        Symbol *n;      /* name */
{
        RunList *r = (RunList *) emalloc(sizeof(RunList));
        r->n = n;
        r->pid = nproc++;
        r->pc = s->frst;
        r->maxseq = s->last->seqno;
        r->nxt = run;
        run = r;
}
```

The actual runlist of executing processes has a program counter `pc` and a pointer to current values of local variables, which we call `symtab` again, since it is basically another symbol table list.

```
typedef struct RunList {
        Symbol  *n;             /* name            */
        int     pid;            /* process id      */
        int     maxseq;         /* used by analyzer generator */
        Element *pc;            /* current stmnt   */
        Symbol  *symtab;        /* local variables */
        struct RunList  *nxt;   /* linked list */
} RunList;
```

Inserting a process into the list is easy.

```
ready(n, p, s)
        Symbol *n;      /* process name */
        Node *p;        /* formal parameters */
        Sequence *s;    /* process body */
{
        ProcList *r = (ProcList *) emalloc(sizeof(ProcList));
        r->n = n;
        r->p = p;
        r->s = s;
        r->nxt = rdy;
        rdy = r;
}
```

Moving a process from the process list to the run list is a little more involved, since also the parameter fields must be initialized.

```
enable(s, n)
        Symbol *s;      /* process name */
        Node *n;        /* actual parameters */
{
        ProcList *p;
        for (p = rdy; p; p = p->nxt)
                if (strcmp(s->name, p->n->name) == 0)
                {       runnable(p->s, p->n);
                        setparams(run, p, n);
                        return (nproc-nstop-1); /* pid */
                }
        return 0; /* process not found */
}
```

where

```
setparams(r, p, q)
        RunList *r;
        ProcList *p;
        Node *q;
{
        Node *f, *a;    /* formal and actual pars */
        Node *t;        /* list of pars of 1 type */

        for (f = p->p, a = q; f; f = f->rgt) /* one type at a time */
        for (t = f->lft; t; t = t->rgt, a = (a)?a->rgt:a)
        {       int k;
                if (!a) fatal("missing actual parameters: ’%s’", p->n->name);
                k = eval(a->lft);        /* must be initialized*/
                if (typck(a, t->nsym->type, p->n->name))
                {       if (t->nsym->type == CHAN)
                                naddsymbol(r, t->nsym, k); /* copy */
                        else
                        {       t->nsym->ini = a->lft;
                                addsymbol(r, t->nsym);
                        }
                }
        }
}
```

with

```
naddsymbol(r, s, k)
        RunList *r;
        Symbol  *s;
{
        Symbol *t = (Symbol *) emalloc(sizeof(Symbol));
        int i;

        t->name = s->name;
        t->type = s->type;
        t->nel  = s->nel;
        t->ini  = s->ini;
        t->val = (int *) emalloc(s->nel*sizeof(int));
        if (s->nel != 1)
        fatal("array in formal parameter list, %s", s->name);
        for (i = 0; i < s->nel; i++)
                t->val[i] = k;
        t->next = r->symtab;
        r->symtab = t;
}
```

and

```
addsymbol(r, s)
        RunList *r;
        Symbol  *s;
{
        Symbol *t = (Symbol *) emalloc(sizeof(Symbol));
        int i;

        t->name = s->name;
        t->type = s->type;
        t->nel  = s->nel;
        t->ini  = s->ini;
        if (s->val)             /* if initialized, copy it */
        {       t->val = (int *) emalloc(s->nel*sizeof(int));
                for (i = 0; i < s->nel; i++)
                        t->val[i] = s->val[i];
        } else
                checkvar(t, 0); /* initialize it */
        t->next = r->symtab;    /* add it */
        r->symtab = t;
}
```

To be able to create new process instantiations on the fly during a simulation, we expand the `run` statement in `run.c` as follows:

```
        case   RUN: return enable(now->nsym, now->lft);
```

Processes can only be deleted from the run-list in reverse order of creation: a process can only disappear if all its children have disappeared first (Chapter 5). The `pid`s can therefore be recycled in stack order. In the value returned by `enable()`, `(nproc-nstop-1)`, the count `nproc` equals the total number of processes created, and `nstop`, the number of processes that deleted.

The most interesting routine left to discuss is the scheduling routine proper: `sched()`. Its relevant portion looks as follows:

```
sched()
{       Element *e, *eval_sub();
        RunList *Y;     /* previous process in run queue */
        int i=0;
        ...
        for (Tval = 0; Tval < 2; Tval++)
        {       while (i < nproc-nstop)
                for (X=run, Y=0, i=0; X; X = X->nxt)
                {       lineno = X->pc->n->nval;
                        Fname  = X->pc->n->fname;
                        if (e = eval_sub(X->pc))
                        {       X->pc = e; Tval=0;
                        } else  /* process terminated? */
                        {       if (X->pc->n->ntyp == '@'
                                &&  X->pid == (nproc-nstop-1))
                                {       if (Y)
                                                Y->nxt = X->nxt;
                                        else
                                                run = X->nxt;
                                        nstop++; Tval=0;
                                } else
                                        i++;
                        }
                        Y = X;
        }       }
        wrapup();
}
```

The scheduler executes one statement in each runnable process in round-robin fashion. It calls the routine `eval_sub()`, which we saw earlier, to recursively evaluate compound statements and atomic sequences. The evaluation of an atomic sequence only succeeds if the whole sequence can be completed. The code is part of `eval_sub()`.

```
        if (e->n->ntyp == ATOMIC)
        {       f = e->n->seql->this->frst;
                g = e->n->seql->this->last;
                g->nxt = e->nxt;
                if (!(g = eval_sub(f)))
                        return (Element *)0;
                Rvous = 0;
                while (g && (g->status & (ATOM|L_ATOM))
                && !(f->status & L_ATOM))
                {       f = g;
                        g = eval_sub(f);
                }
                if (!g)
                {       wrapup();
                        lineno = f->n->nval;
                        fatal("atomic sequence blocks", (cha *)0);
                }
                return g;
        } else if (Rvous)
                ...
```

It is a fatal error if an atomic sequence blocks. If a process hits an unexecutable statement the scheduler checks to see if it is in a stop state. If so, the process is removed from the run queue and the count of terminated processes `nstop` is incremented. A global variable `X` points to the currently executing process. It is mainly used by the new routines for manipulating local variables, `getlocal()` and `setlocal()`, to determine in which process structure the variables are located. The scheduler also maintains a value `Tval` that is used by the interpreter in `run.c` to determine the executability of the `timeout` statement.

```
        case TIMEOUT: return Tval;
```

During normal execution, when the system is not blocked, `Tval` is zero and conditions that include a `timeout` are unexecutable. To recover from a potential deadlock, the scheduler can enable the `timeout` statements by incrementing `Tval`. If the system does not recover, the scheduler declares a true hang state and gives up.

### 12.7.3 LOCAL VARIABLES

Since local variables are created on the fly, upon the instantiation of new processes, the logical place for the code that manipulates them is in the scheduler. If a variable name is local, two special variants of `getvar()` and `setvar()` are used.

```
getlocal(s, n)
        Symbol *s;
{
        Symbol *r;

        r = findloc(s, n);
        if (r) return cast_val(r->type, r->val[n]);
        return 0;
}

setlocal(p, m)
        Node *p;
{
        int n = eval(p->lft);
        Symbol *r = findloc(p->nsym, n);

        if (r) r->val[n] = m;
        return 1;
}
```

The routine `findloc()` locates the name in the symbol table of the currently executing process, pointed to by `X->symtab`.

```
Symbol *
findloc(s, n)
        Symbol *s;
{
        Symbol *r = (Symbol *) 0;

        if (n >= s->nel || n < 0)
        {       yyerror("array indexing error %s", s->name);
                return (Symbol *) 0;
        }

        if (!X)
        {       if (analyze)
                        fatal("error, cannot evaluate variable ’%s’", s->name);
                else
                        yyerror("error, cannot evaluate variable ’%s’", s->name);
                return (Symbol *) 0;
        }
        for (r = X->symtab; r; r = r->next)
                if (strcmp(r->name, s->name) == 0)
                        break;
        if (!r && !Noglobal)
        {       addsymbol(X, s);
                r = X->symtab;
        }
        return r;
}
```

The local variables and the process states of any running process can be referred to in assertions and temporal claims. The hooks in the parser that enable remote referencing are simple. References to remote variables and process states require the last two production rules:

```
        | NAME ’[’ expr ’]’ ’.’ varref  { $$ = rem_var($1, $3, $6); }
        | NAME ’[’ expr ’]’ ’:’ NAME    { $$ = rem_lab($1, $3, $6); }
```

with

```
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);
}
```

and

```
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));
}
```

The reference is implemented as a condition on the control flow state of a process, represented by the internal variable _p, and the state value of a label name. The value of the special variable _p is determined, just like the other remote variables, using a node of type 'p'. The label name is determined with a new node of type 'q'. The node type '?' is only used as a temporary place holder.

In the evaluator two new node types trigger calls on these two routines:

```
        case   'p': return remotevar(now);
        case   'q': return remotelab(now);
```

The first routine, for referencing the current value of a local variable in a remote process, is implemented with a context switch in the scheduler, as follows:

```
remotevar(n)
        Node *n;
{
        int pno, i, j;
        RunList *Y, *oX = X;

        pno = eval(n->lft->lft);        /* pid */
        i = nproc - nstop;
        for (Y = run; Y; Y = Y->nxt)
        if (--i == pno)
        {       if (strcmp(Y->n->name, n->lft->nsym->name))
                        yyerror("wrong proctype %s", Y->n->name);
                X = Y; j = getval(n->nsym, eval(n->rgt)); X = oX;
                return j;
        }
        yyerror("remote ref: proc %s not found", n->nsym->name);
        return 0;
}
```

The second routine, for determining the control flow state in a remote process that corresponds to a given label name, is implemented with a search in the list of labels, as follows:

```
remotelab(n)
        Node *n;
{
        int i;

        if (n->nsym->type)
                fatal("not a labelname: ‘%s’", n->nsym->name);
        if ((i = find_lab(n->nsym, n->lft->nsym)) == 0)
                fatal("unknown labelname: %s", n->nsym->name);
        return i;
}
```

## 12.8 MACRO EXPANSION

The final version of SPIN in Appendix D has an expanded version of `main()` that properly interprets option flags, accepts a file argument and routes its input through the C preprocessor in `/lib/cpp` for macro expansion. The output of the preprocessor is dumped into a temporary file that is immediately unlinked to make sure that it disappears from the file system, even if the run of the simulator is interrupted. The parser, however, keeps a link to the file in the predefined file pointer `yyin`.

```
        if (argc > 1)
        {       char outfile[17], cmd[64];
                strcpy(filename, argv[1]);
                mktemp(strcpy(outfile, "/tmp/spin.XXXXXX"));
                sprintf(cmd, "/lib/cpp %s > %s", argv[1], outfile);
                if (system(cmd))
                {       unlink(outfile);
                        exit(1);
                } else if (!(yyin = fopen(outfile, "r")))
                {       printf("cannot open %s\n", outfile);
                        exit(1);
                }
                unlink(outfile);
        } else
                strcpy(filename, "<stdin>");
```

The preprocessor drops lines into the file that look like

```
# 1 "spin.examples/lynch"
```

The lexical analyzer can pick them up and interpret them with an extra rule that is defined as follows:

```
\#\ [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]);
        }
```

The line number is remembered in variable `lineno`. The file name is stored in the symbol table and a global pointer to it is kept in `Fname`. But most of these remaining features are cosmetic and can be either changed or ignored without undue risk.

## 12.9 SPIN — OPTIONS

The simulator recognizes eight command line options which can be used in any combination. Two (flags *a* and *t*) are specific to the analysis code that we have yet to develop in Chapter 13. The other six are discussed below.

***spin –s***

Prints a line on the display for every message that is sent. Example:

```
$ spin -s factorial
proc 12 (fact)  line   5, Send 1        -> queue 12 (p)
proc 11 (fact)  line  10, Send 2        -> queue 11 (p)
proc 10 (fact)  line  10, Send 6        -> queue 10 (p)
proc  9 (fact)  line  10, Send 24       -> queue 9 (p)
proc  8 (fact)  line  10, Send 120      -> queue 8 (p)
...
```

***spin –r***

Prints a line on the display for every message received. It prints the name and `pid` of the running process, and a source line number for its current state. Example:

```
$ spin  -s -r factorial
proc 12 (fact)  line   5, Send 1        -> queue 12 (p)
proc 11 (fact)  line   9, Recv 1        <- queue 12 (child)
proc 11 (fact)  line  10, Send 2        -> queue 11 (p)
proc 10 (fact)  line   9, Recv 2        <- queue 11 (child)
proc 10 (fact)  line  10, Send 6        -> queue 10 (p)
proc  9 (fact)  line   9, Recv 6        <- queue 10 (child)
....
```

***spin –p***

Prints a line on the display for every statement executed. Example:

```
$ spin -p factorial
proc  0 (_init) line 18 (state 2)
proc  1 (fact)  line 8 (state 4)
proc  1 (fact)  line 9 (state 5)
proc  2 (fact)  line 8 (state 4)
proc  2 (fact)  line 9 (state 5)
proc  3 (fact)  line 8 (state 4)
...
proc  3 (fact)  terminates
```

***spin –l***

Adds the value of all local variables to the output. This option, like the next one, is most useful in combination with *–p*. Example:

```
$ spin -p -l factorial
...
proc 12 (fact)  line 12 (state 9)
                queue 12 (p):
                n = 1
```

```
      proc 11 (fact)  line 4 (state 8)
                      result = 1
                      queue 12 (child):
                      queue 11 (p): [2]
                      n = 2
      ...
      proc 12 (fact)  terminates
      ...
```

***spin –g***

Adds the current values of all global variables to the listings.

***spin –t12345***

Initializes the random number generator with the user specified seed 12345 to secure a simulation run that can be reproduced exactly.

## 12.10 SUMMARY

The last version of SPIN contains about 2000 lines of source code, more than ten times the size of the little expression evaluator that we started this chapter with. To give an indication of the performance of SPIN we ran the following program to calculate Fibonacci numbers. It creates and runs a total of 1000 processes.

```
/***** Fibonacci Sequence *****/

proctype fib(short n)
{       short a = 0;
        short b = 1;
        short c;

        atomic
        {       do
                :: (b < n) ->
                        c = b;
                        b = a+b;
                        a = c
                :: (b >= n) ->
                        break
                od
        }
}

init
{       int i = 1;
        atomic
        {       do
                :: (i < 1000) -> i = i+1; run fib(1000)
                :: (i > 999) -> break
                od
        }
}
```

On a DEC-VAX/8550 computer, one simulation run takes about 7.6 second of user time. A program optimized for calculation can run a similar program two to ten times faster, but of course does not have the synchronization and multi-process features of

PROMELA. The most expensive function calls of a simulation run can be found with the UNIX utility `prof`. For the Fibonacci test 80% of the runtime is spend in the following routines:

```
 %    Time     #Calls     Name
23   2.633   433411     _eval
13   1.533   166433     _findloc
11   1.233   116950     _eval_sub
 9   1.050        0     _strcmp
 8   0.917        0     mcount
 6   0.717   117482     _getlocal
 5   0.617   117482     _cast_val
 5   0.617   117482     _getval
```

If at some point SPIN's efficiency must be improved, a good target for optimizations would be procedure `eval()`. (See also the Exercises.) For protocol simulations, however, the program is sufficiently fast.

In the next chapter we will see how we can extend SPIN's capabilities with a generator for exhaustive protocol validations. As discussed in Chapter 11, the potential performance bottlenecks in protocol validators do require careful attention if we are to produce a tool of practical value. We therefore shift most efficiency considerations to that part of the SPIN software.

## EXERCISES

**12-1.** Change the semantics of the `timeout` statement by allowing a timeout count to be specified. Add a predefined variable `time` that is incremented once for each cycle through the list on running processes by the scheduler. Apply and test the new features with a sample protocol. □

**12-2.** The simulator and the language PROMELA use a 32-bit signed quantity as the largest number. This puts restrictions on the use of SPIN as a calculator. What is the largest factorial that can be computed with the factorial program from Section 12.2? □

**12-3.** Run the Fibonacci test on your system and measure the run-time. Make the atomic sequences non-atomic and repeat the test. Explain the result. □

**12-4.** Model and simulate an arbitrary example PROMELA program from this book. □

**12-5.** Add more features to PROMELA. For instance,
- ○ Allow inline C code fragments
- ○ Allow a keyword `else` in compound statements
- ○ Add C-like data structures
- ○ (Rob Pike) Add *device* channels

A device channel is a prefined message queue that connects a PROMELA program to the outside world (for instance, add a terminal-screen and a keyboard channel).

□

**12-6.** Add a pre-simulation run optimizer that rewrites parts of the parse tree on the fly. Good candidates for optimization, for instance, are expressions involving only constant references, such as `(5*3+2)`. □

**12-7.** Use the simulator to implement a random walk validation strategy. Consider the feasibility of validating each of the correctness criteria discussed in Chapter 6. □

## BIBLIOGRAPHIC NOTES

The best reference to the C programming language is still Kernighan and Ritchie [1978, 1988]. The second edition of this book, published in 1988, is an excellent reference to the new ANSI standard version of the C language. Another good discussion of the ANSI standard definition can be found in Harbison and Steele [1987].

Much more about the design of parsers and lexical analyzers can be found in the famous *dragon* books by Al Aho and others. See for instance Aho and Ullman [1977], and Aho, Sethi and Ullman [1986]. A very useful guide to the usage of the UNIX tools *yacc* and *lex* can also be found in Schreiner and Friedman [1985].

An outstanding tutorial on C program development can be found in Kernighan and Pike [1984]. Chapter 8 of that book is especially recommended.

Device channels (Exercise 12-5) were also defined in the language Squeak, and its successor Newsqueak. See, for instance, Cardelli and Pike [1985].