1. Intro. This program is sort of a reverse of the preprocessor SAT12: Suppose F is a set of clauses for a satisfiability problem, and SAT12 transforms F to F' and outputs the file `/tmp/erp`. Then if some other program finds a solution to F', this program inputs that solution (in *stdin*) together with `/tmp/erp` and outputs a solution to F.

The reader is supposed to be familiar with SAT12, or at least with those parts of SAT12 where the input format and the `erp` file format are specified.

(I hacked this program in a big hurry. It has nothing complicated to do.)

Note: The standard UNIX pipes aren't versatile enough to use this program without auxiliary intermediate files. For instance,

```
sat12 < foo.dat | sat11k | sat12-pre
```

does not work; `sat12-pre` will start to read file `/tmp/erp` before `sat12` has written it! Instead, you must say something like

```
sat12 < foo.dat >! /tmp/bar.dat; sat11k < /tmp/bar.dat | sat12-pre
```

or

```
sat12 < foo.dat | sat11k >! /tmp/bar.sol; sat12-pre < /tmp/bar.sol
```

to get the list of satisfying literals. The second alternative is generally better, because `/tmp/bar.sol` is a one-line file with at most as many literals as there are variables in the reduced clauses, while `/tmp/bar.dat` has the full set of those clauses.

I could probably get around this problem by using named pipes. But I don't want to go to the trouble of creating and destroying them.

```
#define O "%"    /* used for percent signs in format strings */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_flip.h"
  typedef unsigned int uint;    /* a convenient abbreviation */
  typedef unsigned long long ullng;    /* ditto */
  ⟨Type definitions 4⟩;
  ⟨Global variables 2⟩;
  ⟨Subroutines 29⟩;
  main(int argc, char *argv[])
  {
    register uint c, h, i, j, k, kk, l, p, v, vv;
    ⟨Process the command line 3⟩;
    ⟨Initialize everything 7⟩;
    ⟨Input the erp file 8⟩;
    if (¬clauses) fprintf(stderr, "(The␣erp␣file␣is␣empty!)\n");
    ⟨Input the solution 21⟩;
    ⟨Check input anomalies 10⟩;
    ⟨Output the new solution 22⟩;
  }
```

2. Here I'm mostly copying miscellaneous lines of code from SAT12, editing it lightly, and keeping more of it than actually necessary.

```
⟨ Global variables 2 ⟩ ≡
  int random_seed = 0;   /* seed for the random words of gb_rand */
  int hbits = 8;   /* logarithm of the number of the hash lists */
  int buf_size = 1024;   /* must exceed the length of the longest erp input line */
  FILE *erp_file;   /* file to allow reverse preprocessing */
  char erp_file_name[100] = "/tmp/erp";   /* its name */
```

See also sections 6 and 24.

This code is used in section 1.

3. On the command line one can specify nondefault values for any of the following parameters:

- 'h⟨ positive integer ⟩' to adjust the hash table size.
- 'b⟨ positive integer ⟩' to adjust the size of the input buffer.
- 's⟨ integer ⟩' to define the seed for any random numbers that are used.
- 'e⟨ filename ⟩' to change the name of the `erp` output file.

```
⟨ Process the command line 3 ⟩ ≡
  for (j = argc − 1, k = 0; j; j−−)
    switch (argv[j][0]) {
    case 'h': k |= (sscanf(argv[j] + 1, ""O"d", &hbits) − 1); break;
    case 'b': k |= (sscanf(argv[j] + 1, ""O"d", &buf_size) − 1); break;
    case 's': k |= (sscanf(argv[j] + 1, ""O"d", &random_seed) − 1); break;
    case 'e': sprintf(erp_file_name, ""O".99s", argv[j] + 1); break;
    default: k = 1;   /* unrecognized command-line option */
    }
  if (k ∨ hbits < 0 ∨ hbits > 30 ∨ buf_size < 11) {
    fprintf(stderr, "Usage:␣"O"s␣[v<n>]␣[h<n>]␣[b<n>]␣[s<n>]␣[efoo.erp]␣[m<n>]", argv[0]);
    fprintf(stderr, "␣[c<n>]␣<␣foo.dat\n");
    exit(−1);
  }
  if (¬(erp_file = fopen(erp_file_name, "r"))) {
    fprintf(stderr, "I␣couldn't␣open␣file␣"O"s␣for␣reading!\n", erp_file_name);
    exit(−16);
  }
```

This code is used in section 1.

4. The I/O wrapper. The following routines read the input and absorb it into temporary data areas from which all of the "real" data structures can readily be initialized. My intent is to incorporate these routines in all of the SAT-solvers in this series. Therefore I've tried to make the code short and simple, yet versatile enough so that almost no restrictions are placed on the sizes of problems that can be handled. These routines are supposed to work properly unless there are more than $2^{32}-1 = 4{,}294{,}967{,}295$ occurrences of literals in clauses, or more than $2^{31}-1 = 2{,}147{,}483{,}647$ variables or clauses.

In these temporary tables, each variable is represented by four things: its unique name; its serial number; the clause number (if any) in which it has most recently appeared; and a pointer to the previous variable (if any) with the same hash address. Several variables at a time are represented sequentially in small chunks of memory called "vchunks," which are allocated as needed (and freed later).

```
#define vars_per_vchunk 341   /* preferably (2^k − 1)/3 for some k */

⟨Type definitions 4⟩ ≡
  typedef union {
    char ch8[8];
    uint u2[2];
    ullng lng;
  } octa;
  typedef struct tmp_var_struct {
    octa name;   /* the name (one to eight ASCII characters) */
    uint serial;   /* 0 for the first variable, 1 for the second, etc. */
    int stamp;   /* m if positively in clause m; −m if negatively there */
    struct tmp_var_struct *next;   /* pointer for hash list */
  } tmp_var;

  typedef struct vchunk_struct {
    struct vchunk_struct *prev;   /* previous chunk allocated (if any) */
    tmp_var var[vars_per_vchunk];
  } vchunk;
```

See also section 5.

This code is used in section 1.

5. Each clause in the temporary tables is represented by a sequence of one or more pointers to the **tmp_var** nodes of the literals involved. A negated literal is indicated by adding 1 to such a pointer. The first literal of a clause is indicated by adding 2. Several of these pointers are represented sequentially in chunks of memory, which are allocated as needed and freed later.

```
#define cells_per_chunk 511   /* preferably 2^k − 1 for some k */

⟨Type definitions 4⟩ +≡
  typedef struct chunk_struct {
    struct chunk_struct *prev;   /* previous chunk allocated (if any) */
    tmp_var *cell[cells_per_chunk];
  } chunk;
```

6. ⟨ Global variables 2 ⟩ +≡

```
char *buf;   /* buffer for reading the lines (clauses) of erp_file */
tmp_var **hash;   /* heads of the hash lists */
uint hash_bits[93][8];   /* random bits for universal hash function */
vchunk *cur_vchunk;   /* the vchunk currently being filled */
tmp_var *cur_tmp_var;   /* current place to create new tmp_var entries */
tmp_var *bad_tmp_var;   /* the cur_tmp_var when we need a new vchunk */
chunk *cur_chunk;   /* the chunk currently being filled */
tmp_var **cur_cell;   /* current place to create new elements of a clause */
tmp_var **bad_cell;   /* the cur_cell when we need a new chunk */
ullng vars;   /* how many distinct variables have we seen? */
ullng clauses;   /* how many clauses have we seen? */
ullng cells;   /* how many occurrences of literals in clauses? */
int kkk;   /* how many clauses should follow the current erp file group */
```

7. ⟨ Initialize everything 7 ⟩ ≡

```
gb_init_rand(random_seed);
buf = (char *) malloc(buf_size * sizeof(char));
if (¬buf) {
  fprintf(stderr, "Couldn't␣allocate␣the␣input␣buffer␣(buf_size="O"d)!\n", buf_size);
  exit(−2);
}
hash = (tmp_var **) malloc(sizeof(tmp_var) ≪ hbits);
if (¬hash) {
  fprintf(stderr, "Couldn't␣allocate␣"O"d␣hash␣list␣heads␣(hbits="O"d)!\n", 1 ≪ hbits, hbits);
  exit(−3);
}
for (h = 0; h < 1 ≪ hbits; h++) hash[h] = Λ;
```

See also section 15.

This code is used in section 1.

8. The hash address of each variable name has h bits, where h is the value of the adjustable parameter *hbits*. Thus the average number of variables per hash list is $n/2^h$ when there are n different variables. A warning is printed if this average number exceeds 10. (For example, if h has its default value, 8, the program will suggest that you might want to increase h if your input has 2560 different variables or more.)

All the hashing takes place at the very beginning, and the hash tables are actually recycled before any SAT-solving takes place; therefore the setting of this parameter is by no means crucial. But I didn't want to bother with fancy coding that would determine h automatically.

⟨ Input the `erp` file 8 ⟩ ≡

```
while (1) {
  k = fscanf(erp_file, ""O"10s␣<-"O"d", buf, &kkk);
  if (k ≠ 2) break;
  clauses++;
  ⟨ Input one literal 20 ⟩;
  *(cur_cell − 1) = hack_in(*(cur_cell − 1), 4);   /* special marker */
  if (¬fgets(buf, buf_size, erp_file) ∨ buf[0] ≠ '\n') confusion("erp␣group␣intro␣line␣format");
  ⟨ Input kkk clauses 9 ⟩;
}
```

This code is used in section 1.

9. ⟨ Input *kkk* clauses 9 ⟩ ≡

```
for (kk = 0; kk < kkk; kk++) {
  if (¬fgets(buf, buf_size, erp_file)) break;
  clauses++;
  if (buf[strlen(buf) − 1] ≠ '\n') {
    fprintf(stderr, "The clause on line "O"lld ("O".20s...) is too long for me;\n", clauses,
        buf);
    fprintf(stderr, " my buf_size is only "O"d!\n", buf_size);
    fprintf(stderr, "Please use the command-line option b<newsize>.\n");
    exit(−4);
  }
  ⟨ Input the clause in buf 11 ⟩;
}
if (kk < kkk) {
  fprintf(stderr, "file "O"s ended prematurely: "O"d clauses missing!\n", erp_file_name,
      kkk − kk);
  exit(−667);
}
```

This code is used in section 8.

10. ⟨ Check input anomalies 10 ⟩ ≡

```
if ((vars ≫ hbits) ≥ 10) {
  fprintf(stderr, "There are "O"lld variables but only "O"d hash tables;\n", vars, 1 ≪ hbits);
  while ((vars ≫ hbits) ≥ 10) hbits++;
  fprintf(stderr, " maybe you should use command-line option h"O"d?\n", hbits);
}
if (clauses ≡ 0) {
  fprintf(stderr, "No clauses were input!\n");
  exit(−77);
}
if (vars ≥ #80000000) {
  fprintf(stderr, "Whoa, the input had "O"llu variables!\n", vars);
  exit(−664);
}
if (clauses ≥ #80000000) {
  fprintf(stderr, "Whoa, the input had "O"llu clauses!\n", clauses);
  exit(−665);
}
if (cells ≥ #100000000) {
  fprintf(stderr, "Whoa, the input had "O"llu occurrences of literals!\n", cells);
  exit(−666);
}
```

This code is used in section 1.

11. ⟨ Input the clause in *buf* 11 ⟩ ≡

```
for (j = k = 0; ; ) {
  while (buf[j] ≡ '␣') j++;    /* scan to nonblank */
  if (buf[j] ≡ '\n') break;
  if (buf[j] < '␣' ∨ buf[j] > '~') {
    fprintf(stderr, "Illegal␣character␣(code␣#"O"x)␣in␣the␣clause␣on␣line␣"O"lld!\n",
        buf[j], clauses);
    exit(−5);
  }
  if (buf[j] ≡ '~') i = 1, j++;
  else i = 0;
  ⟨ Scan and record a variable; negate it if i ≡ 1 12 ⟩;
}
if (k ≡ 0) {
  fprintf(stderr, "Empty␣line␣"O"lld␣in␣file␣"O"s!\n", clauses, erp_file_name);
  exit(−663);
}
cells += k;
```

This code is used in section 9.

12. We need a hack to insert the bit codes 1, 2, and/or 4 into a pointer value.

```
#define hack_in(q, t) (tmp_var *)(t | (ullng) q)
```

⟨ Scan and record a variable; negate it if $i \equiv 1$ 12 ⟩ ≡

```
{
  register tmp_var *p;
  if (cur_tmp_var ≡ bad_tmp_var) ⟨ Install a new vchunk 13 ⟩;
  ⟨ Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 16 ⟩;
  ⟨ Find cur_tmp_var→name in the hash table at p 17 ⟩;
  if (p→stamp ≡ clauses ∨ p→stamp ≡ −clauses) {
    fprintf(stderr, "Duplicate␣literal␣encountered␣on␣line␣"O"lld!\n", clauses);
    exit(−669);
  } else {
    p→stamp = (i ? −clauses : clauses);
    if (cur_cell ≡ bad_cell) ⟨ Install a new chunk 14 ⟩;
    *cur_cell = p;
    if (i ≡ 1) *cur_cell = hack_in(*cur_cell, 1);
    if (k ≡ 0) *cur_cell = hack_in(*cur_cell, 2);
    cur_cell++, k++;
  }
}
```

This code is used in sections 11 and 20.

13. ⟨Install a new **vchunk** 13⟩ ≡

```
{
  register vchunk *new_vchunk;
  new_vchunk = (vchunk *) malloc(sizeof(vchunk));
  if (¬new_vchunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣vchunk!\n");
    exit(−6);
  }
  new_vchunk→prev = cur_vchunk, cur_vchunk = new_vchunk;
  cur_tmp_var = &new_vchunk→var[0];
  bad_tmp_var = &new_vchunk→var[vars_per_vchunk];
}
```

This code is used in section 12.

14. ⟨Install a new **chunk** 14⟩ ≡

```
{
  register chunk *new_chunk;
  new_chunk = (chunk *) malloc(sizeof(chunk));
  if (¬new_chunk) {
    fprintf(stderr, "Can't␣allocate␣a␣new␣chunk!\n");
    exit(−7);
  }
  new_chunk→prev = cur_chunk, cur_chunk = new_chunk;
  cur_cell = &new_chunk→cell[0];
  bad_cell = &new_chunk→cell[cells_per_chunk];
}
```

This code is used in section 12.

15. The hash code is computed via "universal hashing," using the following precomputed tables of random bits.

⟨Initialize everything 7⟩ +≡

```
for (j = 92; j; j−−)
  for (k = 0; k < 8; k++) hash_bits[j][k] = gb_next_rand();
```

16. ⟨ Put the variable name beginning at *buf*[*j*] in *cur_tmp_var→name* and compute its hash code *h* 16 ⟩ ≡

```
cur_tmp_var→name.lng = 0;
for (h = l = 0; buf[j + l] > '␣' ∧ buf[j + l] ≤ '~'; l++) {
  if (l > 7) {
    fprintf(stderr, "Variable␣name␣%.9s...␣in␣the␣clause␣on␣line␣%lld␣is␣too␣long!\n",
        buf + j, clauses);
    exit(−8);
  }
  h ⊕= hash_bits[buf[j + l] − '!'][l];
  cur_tmp_var→name.ch8[l] = buf[j + l];
}
if (l ≡ 0) {
  fprintf(stderr, "Illegal␣appearance␣of␣~␣on␣line␣%lld!\n", clauses);
  exit(−668);
}
j += l;
h &= (1 ≪ hbits) − 1;
```

This code is used in section 12.

17. ⟨ Find *cur_tmp_var→name* in the hash table at *p* 17 ⟩ ≡

```
for (p = hash[h]; p; p = p→next)
  if (p→name.lng ≡ cur_tmp_var→name.lng) break;
if (¬p) {   /* new variable found */
  p = cur_tmp_var++;
  p→next = hash[h], hash[h] = p;
  p→serial = vars++;
  p→stamp = 0;
}
```

This code is used in section 12.

18. ⟨ Move *cur_cell* backward to the previous cell 18 ⟩ ≡

```
if (cur_cell > &cur_chunk→cell[0]) cur_cell−−;
else {
  register chunk *old_chunk = cur_chunk;

  cur_chunk = old_chunk→prev; free(old_chunk);
  bad_cell = &cur_chunk→cell[cells_per_chunk];
  cur_cell = bad_cell − 1;
}
```

This code is used in sections 26 and 27.

19. ⟨ Move *cur_tmp_var* backward to the previous temporary variable 19 ⟩ ≡

```
if (cur_tmp_var > &cur_vchunk→var[0]) cur_tmp_var−−;
else {
  register vchunk *old_vchunk = cur_vchunk;

  cur_vchunk = old_vchunk→prev; free(old_vchunk);
  bad_tmp_var = &cur_vchunk→var[vars_per_vchunk];
  cur_tmp_var = bad_tmp_var − 1;
}
```

This code is used in section 25.

20. ⟨ Input one literal 20 ⟩ ≡
```
if (buf[0] ≡ '~') i = j = 1;
else i = j = 0;
⟨ Scan and record a variable; negate it if i ≡ 1 12 ⟩;
```
This code is used in sections 8 and 21.

21. ⟨ Input the solution 21 ⟩ ≡
```
clauses++;
k = 0;
while (1) {
  if (scanf(""O"10s", buf) ≠ 1) break;
  if (buf[0] ≡ '~' ∧ buf[1] ≡ 0) {
    printf("~\n");    /* it was unsatisfiable */
    exit(0);
  }
  ⟨ Input one literal 20 ⟩;
}
```
This code is used in section 1.

22. Doing it. When the input phase is done, k literals will have been stored as if they are one huge clause. They are preceded by other groups of clauses, where each group begins with a literal-to-be-defined, identified by a hacked-in 4 bit.

We unwind that data, seeing it backwards as in other programs of this series. Two trivial data structures make the process easy: One for the names of the variables, and one for the current values of the literals.

```
⟨ Output the new solution 22 ⟩ ≡
  ⟨ Allocate the main arrays 23 ⟩;
  for (l = 2; l < vars + vars + 2; l++) lmem[l] = unknown;
  ⟨ Copy all the temporary variable nodes to the vmem array in proper format 25 ⟩;
  if (k) ⟨ Absorb and echo the literals of the given solution 26 ⟩;
  ⟨ Use the erp data to compute the rest of the solution 27 ⟩;
  ⟨ Check consistency 28 ⟩;
  printf("\n");
```

This code is used in section 1.

23. A single **octa** is enough information for each variable, and a single **char** is (more than) enough for each literal.

```
#define true 1
#define false −1
#define unknown 0
#define thevar(l) ((l) ≫ 1)
#define bar(l) ((l) ⊕ 1)   /* the complement of l */
#define litname(l) (l) & 1 ? "~" : "", vmem[thevar(l)].ch8   /* used in printouts */

⟨ Allocate the main arrays 23 ⟩ ≡
  vmem = (octa *) malloc((vars + 1) * sizeof(octa));
  if (¬vmem) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣vmem␣array!\n");
    exit(−10);
  }
  lmem = (char *) malloc((vars + vars + 2) * sizeof(char));
  if (¬lmem) {
    fprintf(stderr, "Oops,␣I␣can't␣allocate␣the␣lmem␣array!\n");
  }
```

This code is used in section 22.

24.
```
⟨ Global variables 2 ⟩ +≡
  octa *vmem;   /* array of variable names */
  char *lmem;   /* array of literal values */
```

25.
```
⟨ Copy all the temporary variable nodes to the vmem array in proper format 25 ⟩ ≡
  for (c = vars; c; c−−) {
    ⟨ Move cur_tmp_var backward to the previous temporary variable 19 ⟩;
    vmem[c].lng = cur_tmp_var→name.lng;
  }
```

This code is used in section 22.

26. **#define** *hack_out*(*q*) (((**ullng**) *q*) & #7)
#define *hack_clean*(*q*) ((**tmp_var** ∗)((**ullng**) *q* & −8))

⟨ Absorb and echo the literals of the given solution 26 ⟩ ≡

```
{
  for (i = 0; i < 2; ) {
    ⟨ Move cur_cell backward to the previous cell 18 ⟩;
    i = hack_out(*cur_cell);
    p = hack_clean(*cur_cell)→serial;
    p += p + (i & 1) + 2;
    printf("␣"O"s"O"s", litname(p));
    lmem[p] = true, lmem[bar(p)] = false;
  }
}
```

This code is used in section 22.

27. At last we get to the heart of this program: Clauses are evaluated (in reverse order of their appearance in the `erp` file) until we come back to a definition point.

⟨ Use the erp data to compute the rest of the solution 27 ⟩ ≡

```
v = true;
for (c = clauses − 1; c; c−−) {
  vv = false;
  for (i = 0; i < 2; ) {
    ⟨ Move cur_cell backward to the previous cell 18 ⟩;
    i = hack_out(*cur_cell);
    p = hack_clean(*cur_cell)→serial;
    p += p + (i & 1) + 2;
    if (i ≥ 4) break;
    if (lmem[p] ≡ unknown) {
      printf("␣"O"s"O"s", litname(p));   /* assign an arbitrary value */
      lmem[p] = true, lmem[bar(p)] = false;
    }
    if (lmem[p] ≡ true) vv = true;   /* vv is OR of literals in clause */
  }
  if (i < 4) {
    if (vv ≡ false) v = false;   /* v is AND of clauses in group */
  } else {   /* defining an eliminated variable */
    lmem[p] = v, lmem[bar(p)] = −v;
    if (v ≡ true) printf("␣"O"s"O"s", litname(p));
    else printf("␣"O"s"O"s", litname(bar(p)));
    v = true;
  }
}
```

This code is used in section 22.

28. ⟨ Check consistency 28 ⟩ ≡

```
if (cur_cell ≠ &cur_chunk→cell[0] ∨ cur_chunk→prev ≠ Λ ∨ cur_tmp_var ≠
      &cur_vchunk→var[0] ∨ cur_vchunk→prev ≠ Λ) confusion("consistency");
free(cur_chunk); free(cur_vchunk);
```

This code is used in section 22.

29. ⟨ Subroutines 29 ⟩ ≡

```
void confusion(char *id)
{    /* an assertion has failed */
  fprintf(stderr, "This can't happen (%s)!\n", id);
  exit(-69);
}
void debugstop(int foo)
{    /* can be inserted as a special breakpoint */
  fprintf(stderr, "You rang(%d)?\n", foo);
}
```

This code is used in section 1.

30. Index.

⟨ Absorb and echo the literals of the given solution 26 ⟩ Used in section 22.
⟨ Allocate the main arrays 23 ⟩ Used in section 22.
⟨ Check consistency 28 ⟩ Used in section 22.
⟨ Check input anomalies 10 ⟩ Used in section 1.
⟨ Copy all the temporary variable nodes to the *vmem* array in proper format 25 ⟩ Used in section 22.
⟨ Find *cur_tmp_var*→*name* in the hash table at p 17 ⟩ Used in section 12.
⟨ Global variables 2, 6, 24 ⟩ Used in section 1.
⟨ Initialize everything 7, 15 ⟩ Used in section 1.
⟨ Input one literal 20 ⟩ Used in sections 8 and 21.
⟨ Input the `erp` file 8 ⟩ Used in section 1.
⟨ Input the clause in *buf* 11 ⟩ Used in section 9.
⟨ Input the solution 21 ⟩ Used in section 1.
⟨ Input *kkk* clauses 9 ⟩ Used in section 8.
⟨ Install a new **chunk** 14 ⟩ Used in section 12.
⟨ Install a new **vchunk** 13 ⟩ Used in section 12.
⟨ Move *cur_cell* backward to the previous cell 18 ⟩ Used in sections 26 and 27.
⟨ Move *cur_tmp_var* backward to the previous temporary variable 19 ⟩ Used in section 25.
⟨ Output the new solution 22 ⟩ Used in section 1.
⟨ Process the command line 3 ⟩ Used in section 1.
⟨ Put the variable name beginning at *buf*[j] in *cur_tmp_var*→*name* and compute its hash code h 16 ⟩ Used in section 12.
⟨ Scan and record a variable; negate it if $i \equiv 1$ 12 ⟩ Used in sections 11 and 20.
⟨ Subroutines 29 ⟩ Used in section 1.
⟨ Type definitions 4, 5 ⟩ Used in section 1.
⟨ Use the erp data to compute the rest of the solution 27 ⟩ Used in section 22.

SAT12-ERP