1. Intro. This is a filter that inputs the format used by SAT0, SAT1, etc., and outputs the well-known DIMACS format for satisfiability problems.

DIMACS format begins with zero or more optional comment lines, indicated by their first character '`c`'. The next line should say '`p cnf` n m', where n is the number of variables and m is the number of clauses. Then comes a string of m "clauses," which are sequences of nonzero integers of absolute value $\le n$, followed by zero. A literal for the kth variable is represented by k; its complement is represented by $-k$.

SAT format is more flexible, more symbolic, and more complicated; it is explained in the programs cited above. I hacked this program from SAT3.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_flip.h"
#include <time.h>
  time_t myclock;
  typedef unsigned int uint;    /* a convenient abbreviation */
  typedef unsigned long long ullng;    /* ditto */
  ⟨Type definitions 4⟩;
  ⟨Global variables 2⟩;
  main(int argc, char *argv[])
  {
    register uint c, h, i, j, k, l, p, q, r, level, kk, pp, qq, ll;
    ⟨Process the command line 3⟩;
    ⟨Initialize everything 7⟩;
    ⟨Input the clauses 8⟩;
    if (verbose) ⟨Report the successful completion of the input phase 20⟩;
    myclock = time(0);
    printf("c␣file␣created␣by␣SAT-TO-DIMACS␣%s", ctime(&myclock));
    ⟨Output the clauses 21⟩;
  }
```

2. ⟨Global variables 2⟩ ≡

```
  int random_seed = 0;    /* seed for the random words of gb_rand */
  int verbose = 1;    /* level of verbosity */
  int hbits = 8;    /* logarithm of the number of the hash lists */
  int buf_size = 1024;    /* must exceed the length of the longest input line */
```

See also section 6.

This code is used in section 1.

3. On the command line one can say

- '`v`⟨ integer ⟩' to enable various levels of verbose output on *stderr*;
- '`h`⟨ positive integer ⟩' to adjust the hash table size;
- '`b`⟨ positive integer ⟩' to adjust the size of the input buffer; and/or
- '`s`⟨ integer ⟩' to define the seed for any random numbers that are used.

⟨ Process the command line 3 ⟩ ≡

```
for (j = argc - 1, k = 0; j; j--)
  switch (argv[j][0]) {
  case 'v': k |= (sscanf(argv[j] + 1, "%d", &verbose) - 1); break;
  case 'h': k |= (sscanf(argv[j] + 1, "%d", &hbits) - 1); break;
  case 'b': k |= (sscanf(argv[j] + 1, "%d", &buf_size) - 1); break;
  case 's': k |= (sscanf(argv[j] + 1, "%d", &random_seed) - 1); break;
  default: k = 1;      /* unrecognized command-line option */
  }
if (k || hbits < 0 || hbits > 30 || buf_size <= 0) {
  fprintf(stderr, "Usage: %s [v<n>] [h<n>] [b<n>] [s<n>] < foo.dat\n", argv[0]);
  exit(-1);
}
```

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];
  long long 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 stdin */
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 nullclauses;   /* how many of them were null? */
ullng cells;   /* how many occurrences of literals in clauses? */
```

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

See also section 13.

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 clauses 8 ⟩ ≡
  while (1) {
    if (¬fgets(buf, buf_size, stdin)) break;
    clauses++;
    if (buf[strlen(buf) − 1] ≠ '\n') {
      fprintf(stderr, "The␣clause␣on␣line␣%lld␣(%.20s...)␣is␣too␣long␣for␣me;\n", clauses, buf);
      fprintf(stderr, "␣my␣buf_size␣is␣only␣%d!\n", buf_size);
      fprintf(stderr, "Please␣use␣the␣command-line␣option␣b<newsize>.\n");
      exit(−4);
    }
    ⟨ Input the clause in buf 9 ⟩;
  }
  if ((vars ≫ hbits) ≥ 10) {
    fprintf(stderr, "There␣are␣%lld␣variables␣but␣only␣%d␣hash␣tables;\n", vars, 1 ≪ hbits);
    while ((vars ≫ hbits) ≥ 10) hbits++;
    fprintf(stderr, "␣maybe␣you␣should␣use␣command-line␣option␣h%d?\n", hbits);
  }
  clauses −= nullclauses;
  if (clauses ≡ 0) {
    fprintf(stderr, "No␣clauses␣were␣input!\n");
    exit(−77);
  }
  if (vars ≥ #80000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣%llu␣variables!\n", vars);
    exit(−664);
  }
  if (clauses ≥ #80000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣%llu␣clauses!\n", clauses);
    exit(−665);
  }
  if (cells ≥ #100000000) {
    fprintf(stderr, "Whoa,␣the␣input␣had␣%llu␣occurrences␣of␣literals!\n", cells);
    exit(−666);
  }
```

This code is used in section 1.

9. ⟨ Input the clause in *buf* 9 ⟩ ≡

```
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␣#%x)␣in␣the␣clause␣on␣line␣%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 10 ⟩;
}
if (k ≡ 0) {
  fprintf(stderr, "(Empty␣line␣%lld␣is␣being␣ignored)\n", clauses);
  nullclauses++;    /* strictly speaking it would be unsatisfiable */
}
goto clause_done;
empty_clause: ⟨ Remove all variables of the current clause 17 ⟩;
clause_done: cells += k;
```

This code is used in section 8.

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

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

⟨ Scan and record a variable; negate it if $i \equiv 1$ 10 ⟩ ≡

```
{
  register tmp_var *p;
  if (cur_tmp_var ≡ bad_tmp_var) ⟨ Install a new vchunk 11 ⟩;
  ⟨ Put the variable name beginning at buf[j] in cur_tmp_var→name and compute its hash code h 14 ⟩;
  ⟨ Find cur_tmp_var→name in the hash table at p 15 ⟩;
  if (p→stamp ≡ clauses ∨ p→stamp ≡ −clauses) ⟨ Handle a duplicate literal 16 ⟩
  else {
    p→stamp = (i ? −clauses : clauses);
    if (cur_cell ≡ bad_cell) ⟨ Install a new chunk 12 ⟩;
    *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 section 9.

11. ⟨ Install a new **vchunk** 11 ⟩ ≡

```
{
  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 10.

12. ⟨ Install a new **chunk** 12 ⟩ ≡

```
{
  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 10.

13. 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();
```

14. ⟨ Put the variable name beginning at *buf*[*j*] in *cur_tmp_var*→*name* and compute its hash code *h* 14 ⟩ ≡

```
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) goto empty_clause;    /* '~' by itself is like 'true' */
j += l;
h &= (1 ≪ hbits) − 1;
```

This code is used in section 10.

15. ⟨ Find *cur_tmp_var→name* in the hash table at *p* 15 ⟩ ≡

```
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 10.

16. The most interesting aspect of the input phase is probably the "unwinding" that we might need to do when encountering a literal more than once in the same clause.

⟨ Handle a duplicate literal 16 ⟩ ≡

```
{
  if ((p→stamp > 0) ≡ (i > 0)) goto empty_clause;
}
```

This code is used in section 10.

17. An input line that begins with '~␣' is silently treated as a comment. Otherwise redundant clauses are logged, in case they were unintentional. (One can, however, intentionally use redundant clauses to force the order of the variables.)

⟨ Remove all variables of the current clause 17 ⟩ ≡

```
while (k) {
  ⟨ Move cur_cell backward to the previous cell 18 ⟩;
  k−−;
}
if ((buf[0] ≠ '~') ∨ (buf[1] ≠ '␣'))
  fprintf(stderr, "(The␣clause␣on␣line␣%lld␣is␣always␣satisfied)\n", clauses);
else if (vars ≡ 0) printf("c␣%s", buf + 2);      /* retain opening comments */
nullclauses++;
```

This code is used in section 9.

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 17 and 24.

19. Here I must omit '*free*(*old_vchunk*)' from the code that's usually in this section, because the variable data will be used later.

⟨ 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;   /* and don't 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 22.

20. ⟨ Report the successful completion of the input phase 20 ⟩ ≡

```
fprintf(stderr, "(%lld␣variables,␣%lld␣clauses,␣%llu␣literals␣successfully␣read)\n", vars,
    clauses, cells);
```

This code is used in section 1.

21. The output phase. I had to input everything first because DIMACS format specifies the number of variables and clauses right at the beginning.

⟨ Output the clauses 21 ⟩ ≡
  ⟨ Show the variable names as comments 22 ⟩;
  printf ("p␣cnf␣%lld␣%lld\n", *vars*, *clauses*);
  ⟨ Translate all the temporary cells into the simple DIMACS form 23 ⟩;
  ⟨ Check consistency 25 ⟩;

This code is used in section 1.

22. This section is optional, but I'm including it today while I remember how to provide it.

⟨ Show the variable names as comments 22 ⟩ ≡
  for ($c = vars$; c; $c--$) {
    ⟨ Move *cur_tmp_var* backward to the previous temporary variable 19 ⟩;
    printf ("c␣%.8s␣->␣%d\n", *cur_tmp_var*→*name.ch8*, *c*);
  }

This code is used in section 21.

23. ⟨ Translate all the temporary cells into the simple DIMACS form 23 ⟩ ≡
  for ($c = clauses$; c; $c--$) {
    ⟨ Translate the cells for the literals of clause c 24 ⟩;
    printf ("␣0\n");
  }

This code is used in section 21.

24. **#define** *hack_out*(q) (((**ullng**) q) & #3)
#define *hack_clean*(q) ((**tmp_var** ∗)((**ullng**) q & -4))

⟨ Translate the cells for the literals of clause c 24 ⟩ ≡
  for ($i = 0$; $i < 2$; $j{+}{+}$) {
    ⟨ Move *cur_cell* backward to the previous cell 18 ⟩;
    i = *hack_out*(∗*cur_cell*);
    p = *hack_clean*(∗*cur_cell*)→*serial*;
    printf ("␣%s%d", i & 1 ? "-" : "", $p + 1$);
  }

This code is used in section 23.

25. ⟨ Check consistency 25 ⟩ ≡
  if (*cur_cell* ≠ &*cur_chunk*→*cell*[0] ∨ *cur_chunk*→*prev* ≠ Λ ∨ *cur_tmp_var* ≠
      &*cur_vchunk*→*var*[0] ∨ *cur_vchunk*→*prev* ≠ Λ) {
    fprintf (*stderr*, "This␣can't␣happen␣(consistency␣check␣failure)!\n");
    exit(−14);
  }

This code is used in section 21.

26. Index.

argc: 1, 3.
argv: 1, 3.
bad_cell: 6, 10, 12, 18.
bad_tmp_var: 6, 10, 11, 19.
buf: 6, 7, 8, 9, 14, 17.
buf_size: 2, 3, 7, 8.
c: 1.
cell: 5, 12, 18, 25.
cells: 6, 8, 9, 20.
cells_per_chunk: 5, 12, 18.
chunk: 5, 6, 12, 18.
chunk_struct: 5.
ch8: 4, 14, 22.
clause_done: 9.
clauses: 6, 8, 9, 10, 14, 17, 20, 21, 23.
ctime: 1.
cur_cell: 6, 10, 12, 18, 24, 25.
cur_chunk: 6, 12, 18, 25.
cur_tmp_var: 6, 10, 11, 14, 15, 19, 22, 25.
cur_vchunk: 6, 11, 19, 25.
empty_clause: 9, 14, 16.
exit: 3, 7, 8, 9, 11, 12, 14, 25.
fgets: 8.
fprintf: 3, 7, 8, 9, 11, 12, 14, 17, 20, 25.
free: 18, 19.
gb_init_rand: 7.
gb_next_rand: 13.
gb_rand: 2.
h: 1.
hack_clean: 24.
hack_in: 10.
hack_out: 24.
hash: 6, 7, 15.
hash_bits: 6, 13, 14.
hbits: 2, 3, 7, 8, 14.
i: 1.
j: 1.
k: 1.
kk: 1.
l: 1.
level: 1.
ll: 1.
lng: 4, 14, 15.
main: 1.
malloc: 7, 11, 12.
myclock: 1.
name: 4, 14, 15, 22.
new_chunk: 12.
new_vchunk: 11.
next: 4, 15.
nullclauses: 6, 8, 9, 17.
octa: 4.
old_chunk: 18.
old_vchunk: 19.
p: 1, 10.
pp: 1.
prev: 4, 5, 11, 12, 18, 19, 25.
printf: 1, 17, 21, 22, 23, 24.
q: 1.
qq: 1.
r: 1.
random_seed: 2, 3, 7.
serial: 4, 15, 24.
sscanf: 3.
stamp: 4, 10, 15, 16.
stderr: 3, 7, 8, 9, 11, 12, 14, 17, 20, 25.
stdin: 6, 8.
strlen: 8.
time: 1.
tmp_var: 4, 5, 6, 7, 10, 24.
tmp_var_struct: 4.
uint: 1, 4, 6.
ullng: 1, 6, 10, 24.
u2: 4.
var: 4, 11, 19, 25.
vars: 6, 8, 15, 17, 20, 21, 22.
vars_per_vchunk: 4, 11, 19.
vchunk: 4, 6, 11, 19.
vchunk_struct: 4.
verbose: 1, 2, 3.

⟨ Check consistency 25 ⟩ Used in section 21.
⟨ Find *cur_tmp_var*→*name* in the hash table at p 15 ⟩ Used in section 10.
⟨ Global variables 2, 6 ⟩ Used in section 1.
⟨ Handle a duplicate literal 16 ⟩ Used in section 10.
⟨ Initialize everything 7, 13 ⟩ Used in section 1.
⟨ Input the clause in *buf* 9 ⟩ Used in section 8.
⟨ Input the clauses 8 ⟩ Used in section 1.
⟨ Install a new **chunk** 12 ⟩ Used in section 10.
⟨ Install a new **vchunk** 11 ⟩ Used in section 10.
⟨ Move *cur_cell* backward to the previous cell 18 ⟩ Used in sections 17 and 24.
⟨ Move *cur_tmp_var* backward to the previous temporary variable 19 ⟩ Used in section 22.
⟨ Output the clauses 21 ⟩ 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 14 ⟩ Used in section 10.
⟨ Remove all variables of the current clause 17 ⟩ Used in section 9.
⟨ Report the successful completion of the input phase 20 ⟩ Used in section 1.
⟨ Scan and record a variable; negate it if $i \equiv 1$ 10 ⟩ Used in section 9.
⟨ Show the variable names as comments 22 ⟩ Used in section 21.
⟨ Translate all the temporary cells into the simple DIMACS form 23 ⟩ Used in section 21.
⟨ Translate the cells for the literals of clause c 24 ⟩ Used in section 23.
⟨ Type definitions 4, 5 ⟩ Used in section 1.

SAT-TO-DIMACS

	Section	Page
Intro	1	1
The I/O wrapper	4	3
The output phase	21	10
Index	26	11