1. SAT. This program reads a CNF formula and answers SAT or UNSAT. The first line of the input starts with the string `"p␣cnf␣"` followed by the number of variables m and the number of clauses n. Variables are numbers in the interval $[1\,..\,m]$ and clauses are sets of variables. Starting from the second line, each clause is described by listing its variables and finishing with a 0. This format is understood also by MiniSAT.

2. The program systematically explores all truth assignments to variables until a satisfying truth assignment is found or all possibilities are exhausted. The state is a partially evaluated CNF.

3. The main function reads the input, backtracks, and reports the result. Overall, the program looks as follows.

```
⟨ Includes 9 ⟩
⟨ Preprocessor definitions ⟩
⟨ Data types 4 ⟩
⟨ Global data 5 ⟩
⟨ Debug code 19 ⟩
⟨ Functions 13 ⟩
int main( )
{
  ⟨ Read the input 7 ⟩;
  printf(sat(0) ? "SAT\n" : "UNSAT\n");
  ⟨ Clean up 21 ⟩;
  return 0;
}
```

4. Occurrences of the same literal (across different clauses) are linked. Literals in the same clause are also linked. That is why the literal node structure has two pairs of (previous, next) pointers. Clauses of the same size are linked.

All lists are doubly linked, circular, and start with a dummy that acts as a list representative.

```
⟨ Data types 4 ⟩ ≡
  struct lit_node {
    struct lit_node *pc;   /* previous literal in the same clause */
    struct lit_node *nc;   /* next literal in the same clause */
    struct lit_node *p;   /* previous occurrence of this literal */
    struct lit_node *n;   /* next occurrence of this literal */
    struct clause_node *clause;   /* the clause of this occurrence */
    int literal;   /* a value in [−m . . −1] or in [1 . . m] or 0 for the dummy */
  };
  struct clause_node {
    struct clause_node *p;   /* previous clause with the same size as this */
    struct clause_node *n;   /* next clause with the same size as this */
    struct lit_node *clause;   /* Λ for the dummy */
    int size;   /* the number of literals in this clause */
  };
```

This code is used in section 3.

5. We can start visiting nodes if we are given a clause size or a literal.

```
⟨Global data 5⟩ ≡
  struct clause_node *formula;    /* formula[k] is the dummy of the list containing clauses of size k */
  struct lit_node *positive;
    /* positive[v] contains the dummy element of the list of occurrences of v in the formula */
  struct lit_node *negative;
    /* negative[v] contains the dummy element of the list of occurrences of ¬v in the formula */
```

See also section 6.

This code is used in section 3.

6. These data structures are built while reading the input. Since the size of the longest clause is not stored in the header we first build one big array with all clauses. Only after reading all clauses we know how much space should be allocated for *formula* and we can distribute clauses according to their sizes.

```
⟨Global data 5⟩ +≡
  int var_cnt;    /* the number of variables */
  int clause_cnt;    /* the number of clauses */
  int max_clause_size;    /* the size of the biggest clause */
  struct clause_node *all_clauses;    /* buffer for storing all clauses */
```

7. ⟨Read the input 7⟩ ≡

```
  ⟨Read the header and initialize 8⟩;
  ⟨Read clauses into one big list and construct lists with literals 10⟩;
  ⟨Distribute clauses according to their size 12⟩;
```

This code is used in section 3.

8. Initial lines not starting with "p␣cnf␣" are ignored.

```
⟨Read the header and initialize 8⟩ ≡
  char linebuf[1 ≪ 10];    /* buffer for reading lines */

  max_clause_size = 0;
  while (fgets(linebuf, 1 ≪ 10, stdin) ∧ strncmp("p␣cnf", linebuf, 5)) ;
  if (sscanf(linebuf, "p␣cnf␣%d␣%d", &var_cnt, &clause_cnt) ≠ 2) {
    fprintf(stderr, "Input␣must␣contain\np␣cnf␣<varcnt>␣<clausecnt>\n");
    return 1;
  }
  all_clauses = malloc(clause_cnt * sizeof(struct clause_node));
  positive = malloc((var_cnt + 1) * sizeof(struct lit_node));
  negative = malloc((var_cnt + 1) * sizeof(struct lit_node));
  for (int i = 1; i ≤ var_cnt; ++i) {
    positive[i].n = positive[i].p = &positive[i];
    negative[i].n = negative[i].p = &negative[i];
    positive[i].literal = negative[i].literal = 0;
  }
```

This code is used in section 7.

9. ⟨Includes 9⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

This code is used in section 3.

10. ⟨ Read clauses into one big list and construct lists with literals 10 ⟩ ≡

```
for (int i = 0; i < clause_cnt; ++i) {
   struct clause_node *clause = &all_clauses[i];    /* clause being processed */
   clause→size = 0;
   clause→clause = malloc(sizeof(struct lit_node));
   clause→clause→pc = clause→clause→nc = clause→clause;
   clause→clause→literal = 0;
   while (1) ⟨ Read one literal and add it to clause or break if done 11 ⟩;
   if (max_clause_size < clause→size) max_clause_size = clause→size;
}
```

This code is used in section 7.

11. TODO: I should probably not trust *malloc* to do a good job here.

```
#define lit_head(l)   ((l) > 0 ? &positive[l] : &negative[−(l)])
          /* extracts the global list of occurrences of the literal l */
```

⟨ Read one literal and add it to *clause* or **break** if done 11 ⟩ ≡

```
{
   int l;    /* for reading literal identifiers */
   struct lit_node *literal;    /* the literal being read */
   if (scanf("%d", &l) ≠ 1 ∨ ¬l) break;
   literal = malloc(sizeof(struct lit_node));
   literal→literal = l;
   literal→clause = clause;
   literal→p = lit_head(l), literal→n = lit_head(l)→n;
   literal→p→n = literal→n→p = literal;
   literal→nc = clause→clause→nc, literal→pc = clause→clause;
   literal→pc→nc = literal→nc→pc = literal;
   ++clause→size;
}
```

This code is used in section 10.

12. ⟨ Distribute clauses according to their *size* 12 ⟩ ≡

```
formula = malloc((max_clause_size + 1) * sizeof(struct clause_node));
for (int i = 0; i ≤ max_clause_size; ++i) {
   formula[i].p = formula[i].n = &formula[i];
   formula[i].clause = Λ;
}
for (struct clause_node *c = all_clauses; c ≠ all_clauses + clause_cnt; ++c) {
   c→n = formula[c→size].n, c→p = &formula[c→size];
   c→p→n = c→n→p = c;
}
```

This code is used in section 7.

13. Backtracking is implemented using recursion. A partial evaluation step consists in removing literals from their clause list and in removing clauses from their list. To undo such a step we need to keep track of the literal that set to *true* and a list with the clauses that were removed.

The first thing to do is to check if the empty clause was derived.

```
⟨ Functions 13 ⟩ ≡
  int sat(int depth)
  {
    int l;    /* the literal that gets set to true */
    ⟨ Check if trivially SAT or UNSAT 17 ⟩;
    ⟨ Pick a literal l 18 ⟩;
    ⟨ Set l to true and recurse 14 ⟩;
    l = −l;
    ⟨ Set l to true and recurse 14 ⟩;
    return 0;
  }
```

This code is used in section 3.

14. The information that must be saved during a partial evaluation step so that it can be undone consist of (1) the list of removed clauses and (2) the list of literal occurrences in removed clauses (other than l). This means that in order to remove a clause we spend time proportional to its size, which isn't great. The hope is that this saves time later on because we never have to look at removed clauses again.

There is no need to explicitly remove occurrences of l since it doesn't appear in any non-removed clause, so it won't be picked again by a subsequent partial evaluation step.

```
⟨ Set l to true and recurse 14 ⟩ ≡
  {
    struct lit_node *ln, *lln;    /* literal nodes of interest */
    struct clause_node *cn, *ccn;    /* clause nodes of interest */
    struct clause_node rc;    /* removed clauses */
    int rsat;    /* recursive result */
    rc.p = rc.n = &rc, rc.clause = Λ;
    if (verbose) {
      for (int d = 0; d < depth; ++d) printf("␣");
      printf("set␣%d\n", l);
    }
    ⟨ Partially evaluate formula for l ≡ true 15 ⟩;
    if (verbose) printformula(depth);
    rsat = sat(depth + 1);
    if (verbose) {
      for (int d = 0; d < depth; ++d) printf("␣");
      printf("unset␣%d\n", l);
    }
    ⟨ Undo partial evaluation 16 ⟩;
    if (verbose) printformula(depth);
    if (rsat) return 1;
  }
```

This code is used in section 13.

15. To partially evaluate a formula we remove occurrences of $-l$ from their clauses (but not from the global list of occurrences of $-l$). We also remove clauses that contain l and remove all ocurrences of other literals in removed clauses from their global lists (but not from the removed clause); the literal l is an exception from this rule.

```
⟨ Partially evaluate formula for $l \equiv true$ 15 ⟩ ≡
  for ($ln = lit\_head(-l)\to n$; $ln \neq lit\_head(-l)$; $ln = ln\to n$) {
    $ln\to pc\to nc = ln\to nc, ln\to nc\to pc = ln\to pc$;
    $ln\to pc = ln\to nc = ln$;
    $cn = ln\to clause$;
    $cn\to p\to n = cn\to n, cn\to n\to p = cn\to p$;
    $--cn\to size$;
    $cn\to p = \&formula[cn\to size], cn\to n = formula[cn\to size].n$;
    $cn\to p\to n = cn\to n\to p = cn$;
  }
  for ($ln = lit\_head(l)\to n$; $ln \neq lit\_head(l)$; $ln = ln\to n$) {
    $cn = ln\to clause$;
    for ($lln = cn\to clause\to nc$; $lln \neq cn\to clause$; $lln = lln\to nc$)
      if ($lln\to literal \neq ln\to literal$) {
        $lln\to p\to n = lln\to n, lln\to n\to p = lln\to p$;
        $lln\to n = lln\to p = lln$;
      }
    $cn\to p\to n = cn\to n, cn\to n\to p = cn\to p$;
    $cn\to p = \&rc, cn\to n = rc.n$;
    $cn\to p\to n = cn\to n\to p = cn$;
  }
```

This code is used in section 14.

16. Let's see if we know how to undo what we just did.

```
⟨ Undo partial evaluation 16 ⟩ ≡
  for ($cn = rc.n$; $cn \neq \&rc$; $cn = ccn$) {
    for ($ln = cn\to clause\to nc$; $ln \neq cn\to clause$; $ln = ln\to nc$)
      if ($ln\to literal \neq l$) {
        $ln\to p = lit\_head(ln\to literal), ln\to n = lit\_head(ln\to literal)\to n$;
        $ln\to p\to n = ln\to n\to p = ln$;
      }
    $ccn = cn\to n$;
    $cn\to p = \&formula[cn\to size], cn\to n = formula[cn\to size].n$;
    $cn\to p\to n = cn\to n\to p = cn$;
  }
  for ($ln = lit\_head(-l)\to n$; $ln \neq lit\_head(-l)$; $ln = ln\to n$) {
    $cn = ln\to clause$;
    $ln\to pc = cn\to clause, ln\to nc = cn\to clause\to nc$;
    $ln\to pc\to nc = ln\to nc\to pc = ln$;
    $cn\to p\to n = cn\to n, cn\to n\to p = cn\to p$;
    $++cn\to size$;
    $cn\to p = \&formula[cn\to size], cn\to n = formula[cn\to size].n$;
    $cn\to p\to n = cn\to n\to p = cn$;
  }
```

This code is used in section 14.

17. Recursion stops when the empty clause was derived, or no clause remains to be satisfied.

```
⟨ Check if trivially SAT or UNSAT 17 ⟩ ≡
  int sz;    /* clause size */
  if (formula[0].n ≠ &formula[0]) return 0;
  for (sz = 1; sz ≤ max_clause_size ∧ formula[sz].n ≡ &formula[sz]; ++sz) ;
  if (sz > max_clause_size) return 1;
```

This code is used in section 13.

18. The literal we pick is some literal from one of the smallest clauses left.

```
⟨ Pick a literal l 18 ⟩ ≡
  l = formula[sz].n→clause→nc→literal;
```

This code is used in section 13.

19. Debugging stuff.

```
#define verbose 0

⟨ Debug code 19 ⟩ ≡
  void printformula(int depth)
  {
    for (int d = 0; d < depth; ++d) printf("␣");
    for (int sz = 0; sz ≤ max_clause_size; ++sz) {
      for (struct clause_node *cn = formula[sz].n; cn ≠ &formula[sz]; cn = cn→n) {
        for (struct lit_node *ln = cn→clause→nc; ln ≠ cn→clause; ln = ln→nc) {
          printf("␣%d", ln→literal);
        }
        printf("|");
      }
    }
    printf("\n");
  }
```

This code is used in section 3.

20. TODO.

21. ⟨ Clean up 21 ⟩ ≡

This code is used in section 3.

SAT