1. Intro. This program generates clauses that are satisfiable if and only if there's a Boolean chain x_1, $\ldots$, x_{n+r} in n variables x_1, $\ldots$, x_n that computes the functions whose truth tables are T_1, $\ldots$, T_m. The parameters are given on the command line. I assume that $n \le 6$, so that each truth table has at most 64 bits. The truth tables are specified in hexadecimal notation, using 2^{n-2} hex digits each.

The chains are assumed to be "normal"; that is, each function on each step takes $(0,\ldots,0) \mapsto 0$. (If a parameter T_j isn't normal, we compute its complement.)

Steps are indicated in clause-variable names by a single character, beginning with `1`, `2`, $\ldots$, `9`, `a`, `b`, $\ldots$; the first n steps are reserved for the projection functions x_1 through x_n.

The clauses involve several kinds of variables:

- `F`kbb' means that the Boolean binary function at step k has $F_k(b,b') = 1$; here $n < k \le n+r$ and $0 \le b,b' \le 1$, $b+b' > 0$.
- `K`kji means that $x_k = F_k(x_j, x_i)$; here $n < k \le n+r$ and $k > j > i > 0$.
- `Z`ik means that the ith output z_i is x_k; here $1 \le i \le m$ and $n < k \le n+r$.
- `X`$kb_1b_2\ldots b_n$ means that the Boolean function x_k takes $(b_1,\ldots,b_n) \mapsto 1$; here $n < k \le n+r$ and $0 \le b_1,\ldots,b_n \le 1$, $b_1+\cdots+b_n > 0$.

```
#define maxn 6   /* at most this many variables */
#define maxk 36   /* at most this many steps */
#include <stdio.h>
#include <stdlib.h>
  int n, r;   /* command-line parameters */
  unsigned long long t[maxk];   /* truth tables on the command line */
  unsigned long long x[maxn + 1];   /* truth tables for the projections */
  ⟨ Subroutines 8 ⟩;
  main(int argc, char *argv[])
  {
    register int b, bb, bbb, h, i, j, k, m;
    register unsigned long long mask;
    ⟨ Process the command line 2 ⟩;
    for (k = n + 1; k ≤ n + r; k++) ⟨ Generate the clauses for step k 3 ⟩;
    for (i = 1; i ≤ m; i++) ⟨ Generate the clauses for output i 10 ⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
if (argc < 4 ∨ sscanf(argv[1], "%d", &n) ≠ 1 ∨ sscanf(argv[2], "%d", &r) ≠ 1) {
  fprintf(stderr, "Usage:␣%s␣n␣r␣t1␣...␣tm\n", argv[0]);
  exit(−1);
}
if (n < 2 ∨ n > maxn) {
  fprintf(stderr, "n␣should␣be␣between␣2␣and␣%d,␣not␣%d!\n", maxn, n);
  exit(−2);
}
if (n + r > maxk) {
  fprintf(stderr, "n+r␣should␣be␣at␣most␣%d,␣not␣%d!\n", maxk, n + r);
  exit(−3);
}
mask = (n ≡ 6 ? −1 : (1_LL ≪ (1 ≪ n)) − 1);
x[1] = mask ≫ (1 ≪ (n − 1));
for (i = 2; i ≤ n; i++) x[i] = x[i − 1] ⊕ (x[i − 1] ≪ (1 ≪ (n − i)));
m = argc − 3;
if (m > r) {
  fprintf(stderr, "the␣number␣of␣outputs␣should␣be␣at␣most␣r,␣not␣%d!\n", m);
  exit(−4);
}
for (i = 1; i ≤ m; i++) {
  if (sscanf(argv[2 + i], "%llx", &t[i]) ≠ 1) {
    fprintf(stderr, "I␣couldn't␣scan␣truth␣table␣t%d!\n", i);
    exit(−5);
  }
  if (n < 6 ∧ (t[i] ≫ (1 ≪ n))) {
    fprintf(stderr, "Truth␣table␣t%d␣(%llx)␣has␣too␣many␣bits!\n", i, t[i]);
    exit(−6);
  }
  if (t[i] ≫ ((1 ≪ n) − 1)) t[i] = (∼t[i]) & mask;
}
printf("~␣sat-chains␣%d␣%d", n, r);
for (i = 1; i ≤ m; i++) printf("␣%llx", t[i]);
printf("\n");
```

This code is used in section 1.

3. ⟨ Generate the clauses for step k 3 ⟩ ≡

```
{
  ⟨ Generate clauses to say that operation k isn't trivial 4 ⟩;
  ⟨ Generate clauses to say that step k is based on two prior steps 5 ⟩;
  ⟨ Generate clauses to say that step k is used at least once 6 ⟩;
  ⟨ Generate clauses to exploit the completeness of the basis functions 7 ⟩;
  for (i = 1; i < k; i++)
    for (j = i + 1; j < k; j++) ⟨ Generate the main clauses that involve Kkji 9 ⟩;
}
```

This code is used in section 1.

4. **#define** $e(t)$ $((t) \le 9 ? \texttt{'0'} + t : \texttt{'a'} + t - 10)$

⟨ Generate clauses to say that operation k isn't trivial 4 ⟩ ≡

```
printf("F%c01␣F%c10␣F%c11\n", e(k), e(k), e(k));
printf("F%c01␣~F%c10␣~F%c11\n", e(k), e(k), e(k));
printf("~F%c01␣F%c10␣~F%c11\n", e(k), e(k), e(k));
```

This code is used in section 3.

5. ⟨ Generate clauses to say that step k is based on two prior steps 5 ⟩ ≡

```
for (i = 1; i < k; i++)
  for (j = i + 1; j < k; j++) printf("␣K%c%c%c", e(k), e(j), e(i));
printf("\n");
```

This code is used in section 3.

6. ⟨ Generate clauses to say that step k is used at least once 6 ⟩ ≡

```
for (i = 1; i ≤ m; i++) printf("␣Z%c%c", e(i), e(k));
for (j = k + 1; j ≤ n + r; j++)
  for (i = 1; i < k; i++) printf("␣K%c%c%c", e(j), e(k), e(i));
for (j = k + 1; j ≤ n + r; j++)
  for (i = k + 1; i < j; i++) printf("␣K%c%c%c", e(j), e(i), e(k));
printf("\n");
```

This code is used in section 3.

7. If x_k depends only on x_j and x_i, we can assume that no future step will combine x_k with either x_j or x_i. (Because that future step might as well act directly on x_j and x_i.)

⟨ Generate clauses to exploit the completeness of the basis functions 7 ⟩ ≡

```
for (i = 1; i < k; i++)
  for (j = i + 1; j < k; j++)
    for (h = k + 1; h ≤ n + r; h++) {
      printf("~K%c%c%c␣~K%c%c%c\n", e(k), e(j), e(i), e(h), e(k), e(j));
      printf("~K%c%c%c␣~K%c%c%c\n", e(k), e(j), e(i), e(h), e(k), e(i));
    }
```

This code is used in section 3.

8. ⟨ Subroutines 8 ⟩ ≡

```
void printX(char *s, int k, int t)
{
  register int i;
  if (k > n) {
    printf("␣%s%c", s, e(k));
    for (i = 1; i ≤ n; i++) printf("%d", (t ≫ (n − i)) & 1);
  }
}
```

This code is used in section 1.

9. **#define** $bit\_h(i)$ $(\mathbf{int})((x[i] \gg ((1 \ll n) - 1 - h)) \mathbin{\&} 1)$

⟨ Generate the main clauses that involve Kkji 9 ⟩ ≡

```
{
  for (h = 1; h < (1 ≪ n); h++) {
    for (b = 0; b ≤ 1; b++)
      for (bb = 0; bb ≤ 1; bb++) {
        if (j ≤ n ∧ bit_h(j) ≠ b) continue;
        if (i ≤ n ∧ bit_h(i) ≠ bb) continue;
        if (b + bb ≡ 0) {
          printf("~K%c%c%c", e(k), e(j), e(i));
          printX("X", j, h);
          printX("X", i, h);
          printX("~X", k, h);
          printf("\n");   /* (0,0) ↦ 0 */
        } else
          for (bbb = 0; bbb ≤ 1; bbb++) {
            printf("~K%c%c%c", e(k), e(j), e(i));
            if (b) printX("~X", j, h);
            else printX("X", j, h);
            if (bb) printX("~X", i, h);
            else printX("X", i, h);
            if (bbb) printX("~X", k, h);
            else printX("X", k, h);
            printf("␣%sF%c%d%d\n", bbb ? "" : "~", e(k), b, bb);
          }
      }
  }
}
```

This code is used in section 3.

10. ⟨ Generate the clauses for output i 10 ⟩ ≡

```
{
  ⟨ Generate clauses to say that output i is present 11 ⟩;
  for (k = n + 1; k ≤ n + r; k++) ⟨ Generate the clauses that involve Zik 12 ⟩;
}
```

This code is used in section 1.

11. ⟨ Generate clauses to say that output i is present 11 ⟩ ≡

```
for (k = n + 1; k ≤ n + r; k++) printf("␣Z%c%c", e(i), e(k));
printf("\n");
```

This code is used in section 10.

12. ⟨ Generate the clauses that involve Zik 12 ⟩ ≡

```
{
  for (h = 1; h < (1 ≪ n); h++) {
    printf("~Z%c%c", e(i), e(k));
    if (t[i] & (1LL ≪ ((1 ≪ n) - 1 - h))) printX("X", k, h);
    else printX("~X", k, h);
    printf("\n");
  }
}
```

This code is used in section 10.

13. Index.

SAT-CHAINS