1. Introduction. This program translates input from `/dev/kbd` to Chinese runes. The idea is based on `ktrans` but this is a rewrite to have much cleaner code and the efficiency for handling a much bigger dictionary.

The usage is the almost the same as `ktrans`: just invoke `cim` before `rio`.

`^T` gives you English mode, `^N` enters Cangjie mode. `^L` rotates selection.

I know you might want Pinyin but Cangjie is a little bit easier to implement and way easier to use when there is no prompt.

2. Setup keyboard I/O. See `kbdfs(8)`. Reading `/dev/kbd` gives keycodes and the `c` messages are what we want to process. Besides handling the event produced by the operating system, we need also to ensure other programs that read `/dev/kbd` can acquire the translated result by faking a new keyboard device to shadow the original one.

```
⟨ Setup keyboard 2 ⟩ ≡
  if ((kbd = open("/dev/kbd", OREAD)) < 0)
        sysfatal("open␣kbd:␣%r");
  ⟨ Shadowing /dev/kbd 4 ⟩
```

This code is used in section 30.

3.
```
⟨ Variables 3 ⟩ ≡
  int kbd, newkbd;
```

See also section 6.

This code is used in section 30.

4. The call *bind*(`"#|"`, *dir*, `MREPL`) would create a pair of pipes named `data` and `data1` in *dir*.

```
⟨ Shadowing /dev/kbd 4 ⟩ ≡
  if (bind("#|", "/n/temp", MREPL) < 0)
        sysfatal("bind␣/n/temp:␣%r");
  if ((newkbd = open("/n/temp/data1", OWRITE)) < 0)
        sysfatal("open␣kbd␣pipe:␣%r");
  if (bind("/n/temp/data", "/dev/kbd", MREPL) < 0)
        sysfatal("bind␣kbd␣pipe:␣%r");
  unmount(nil, "/n/temp");
```

This code is used in section 2.

5. This loop would be used to intercept input and output translated result. When the start of the message is `'c'` the following stting contains one character. The function *nextstate*( ) would output runes to *newkbd* when possible.

```
⟨ Key translation loop 5 ⟩ ≡
  while ((n = read(kbd, buf, sizeof (buf))) > 0) {
    buf[n − 1] = 0;
    if (n < 2 ∨ buf[0] ≠ 'c')
          write(newkbd, buf, n);
    else nextstate(buf, n, newkbd);
  }
```

This code is used in section 30.

6.
```
⟨ Variables 3 ⟩ +≡
  int n;
  char buf[128];
```

7. Dictionary. The dictionary is defined as several C arrays in a separated file named `dict.c`. It has to be generated by a external program and that detail would not be discussed here. The requirement for dictionary is that all entries need to be sorted with lexicographical order. An **int** is used as a key to encode the string of at most three alphabets.

```
format Rune char    /* Rune is the type for Unicode characters */
```

⟨ Declarations 7 ⟩ ≡

```
typedef struct Map Map;
struct Map {
  int key;
  Rune *val;
};
extern const Map dict[];
```

See also section 8.

This code is used in section 30.

8. There are two precomputed indices to speed up lookup.

⟨ Declarations 7 ⟩ +≡

```
extern const int lvl1[];
extern const int lvl2[];
```

9. *lvl1* contains indices from `"a"` to `"z"`.

⟨ Lookup in *lvl1* 9 ⟩ ≡

```
return &dict[lvl1[key[0] - 'a']];
```

This code is used in section 11.

10. *lvl2* contains indices from `"aa"`, `"ab"` to `"zz"`. Since it is likely the indices are sparse, a negative value indicates not found.

⟨ Lookup in *lvl2* 10 ⟩ ≡

```
{
  int index = lvl2[(key[0] - 'a') * 26 + (key[1] - 'a')];
  return (index < 0) ? nil : &dict[index];
}
```

This code is used in section 11.

11. Find in the dictionary. Since the indices are very sparse after level 2, only linear search would be needed. The optional argument *table* can be used to resume a search.

⟨ Subroutines 11 ⟩ ≡

```
const Map *match(const char *key, int len, const Map *table)
{
  switch (len) {
  case 1: ⟨ Lookup in lvl1 9 ⟩
  case 2: ⟨ Lookup in lvl2 10 ⟩
  default:
    if (table ≠ nil) ⟨ Linear search 13 ⟩
    else ⟨ Speedup with lvl2 12 ⟩
  }
}
```

See also section 16.

This code is used in section 30.

12. ⟨ Speedup with *lvl2* 12 ⟩ ≡

```
return match(key, len, match(key, 2, nil));
```

This code is used in section 11.

13. Since the keys are ordered, if the compare result is negative, return *nil*. Only codes after the first two characters are stored in the key. If the key's length is less than 3 it is only stored as 0.

⟨ Linear search 13 ⟩ ≡

```
{
  int cmp;
  int keycode;
  ⟨ Compute keycode 14 ⟩
  for ⟨ First none 0 key until next 0 15 ⟩ {
    cmp = (table[i].key ≫ (5 * (3 − (len − 2)))) − keycode;
    if (cmp ≡ 0) return &table[i];
    else if (cmp > 0) return nil;
  }
  return nil;
}
```

This code is used in section 11.

14. Each alphabet occupies 5 bits. There are 3 alphabets in the keycode.

⟨ Compute keycode 14 ⟩ ≡

```
keycode = 0;
for (int i = 2; i < len; i++) {
  keycode ≪= 5;
  keycode += key[i] − 'a' + 1;
}
```

This code is used in section 13.

15. When a two character key doesn't exist, the index for next key with same prefix is stored.

⟨ First none 0 key until next 0 15 ⟩ ≡

```
(int i = (table[0].key) ? 0 : 1; table[i].key; i++)
```

This code is used in section 13.

16. State machine. Right now this is a bit of messy.

```
⟨ Subroutines 11 ⟩ +≡
  void nextstate(char *oldbuf, int n, int outfd)
  {
    if (len > 5) {
      len = 0;
      laststate = nil;
    }
    if (¬natural ∧ islower(oldbuf[1])) {
      ⟨ Copy input character to input buffer 26 ⟩
      ⟨ Search and output 19 ⟩
      return;
    }
    else ⟨ Handle mode change and selection 17 ⟩
    ⟨ Directly write to output 28 ⟩
    return;
  }
```

17. ⟨ Handle mode change and selection 17 ⟩ ≡

```
  if (oldbuf[1] ≡ 20) {      /* ^T */
    natural = 1;
    ⟨ Reset state 25 ⟩
    return;
  }
  else if (oldbuf[1] ≡ 14) {      /* ^N */
    natural = 0;
    ⟨ Reset state 25 ⟩
    return;
  }
  else if (¬natural ∧ oldbuf[1] ≡ '␣') {
    ⟨ Reset state 25 ⟩
    return;
  }
  else if (oldbuf[1] ≡ 12) {      /* ^L */
    ⟨ Rotate selection 20 ⟩
    return;
  }
  else if (oldbuf[1] ≡ '\b') {
    ⟨ Reset state 25 ⟩
  }
```

This code is used in section 16.

18. This indicates the English/Chinese state.

```
⟨ Global states 18 ⟩ ≡
  int natural = 0;
```

See also sections 21 and 24.

This code is used in section 30.

19. The only **goto** statement in this program. It enables auto start of next input if the last character is already complete.

```
⟨ Search and output 19 ⟩ ≡
  const Map *result;
Search: result = match(input, len, laststate);
  if (result ≠ nil) {
    if (laststate ≠ nil) ⟨ Backspace 23 ⟩
    candidate = result→val;
    ⟨ Write current candidate 22 ⟩
    laststate = result;
  }
  else if (laststate ≠ nil) {
    ⟨ Flush input buffer until the but last character 27 ⟩
    laststate = nil;
    goto Search;
  }
```

This code is used in section 16.

20.

```
⟨ Rotate selection 20 ⟩ ≡
  if (laststate ≠ nil) {
    ⟨ Backspace 23 ⟩
    candidate++;
    if (¬*candidate) candidate = laststate→val;
    ⟨ Write current candidate 22 ⟩
  }
```

This code is used in section 17.

21.

```
⟨ Global states 18 ⟩ +≡
  Rune *candidate;
```

22.

```
⟨ Write current candidate 22 ⟩ ≡
  char buf[128];
  int n = snprint(buf, sizeof (buf), "c%C", *candidate) + 1;
  write(outfd, buf, n);
```

This code is used in sections 19 and 20.

23.

```
⟨ Backspace 23 ⟩ ≡
  write(outfd, "c\b\0", 3);
```

This code is used in sections 19 and 20.

24. These states keeps a record of current input.

```
⟨ Global states 18 ⟩ +≡
  char input[20];
  int len = 0;
  const Map *laststate = nil;
```

25.

```
⟨ Reset state 25 ⟩ ≡
  len = 0;
  laststate = nil;
```

This code is used in section 17.

26. ⟨ Copy input character to input buffer 26 ⟩ ≡
```
input[len] = oldbuf[1];
len += 1;
input[len] = '\0';
```
This code is used in section 16.

27. ⟨ Flush input buffer until the but last character 27 ⟩ ≡
```
input[0] = input[len - 1];
input[1] = '\0';
len = 1;
```
This code is used in section 19.

28. ⟨ Directly write to output 28 ⟩ ≡
```
write(outfd, oldbuf, n);
```
This code is used in section 16.

29. Program entry. A standard Plan9 C program would use these headers.

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
```

30. The *main*( ) function starts the daemon process and call *exits*(*nil*).

```
⟨ Declarations 7 ⟩
⟨ Global states 18 ⟩
⟨ Subroutines 11 ⟩
void main(int argc, char **argv)
{
  ⟨ Variables 3 ⟩;
  if (argc > 1) {
    ⟨ Test dictionary 31 ⟩
    exits(nil);
  }
  ⟨ Setup keyboard 2 ⟩
  if (fork( )) exits(nil);
  ⟨ Key translation loop 5 ⟩
}
```

31. ⟨ Test dictionary 31 ⟩ ≡

```
const Map *result = match(argv[1], strlen(argv[1]), nil);
if (result ≡ nil) print("not␣found!\n");
else print("%S\n", result→val);
```

This code is used in section 30.

32. Bugs. The translation could fail if typing is too fast.

CIM - The Cangjie Input Method for 9front