INTERNET-DRAFT
draft-ietf-idn-dude-02.txt
Expires 2001-Dec-07

Mark Welter
Brian W. Spolarich
Adam M. Costello

2001-Jun-07

Differential Unicode Domain Encoding (DUDE)

Status of this Memo

Abstract

DUDE is a reversible transformation from a sequence of nonnegative integer values to a sequence of letters, digits, and hyphens (LDH characters). DUDE provides a simple and efficient ASCII-Compatible Encoding (ACE) of Unicode strings [UNICODE] for use with Internationalized Domain Names [IDN] [IDNA].

Contents

The IDNA draft [IDNA] describes an architecture for supporting internationalized domain names. Each label of a domain name may begin with a special prefix, in which case the remainder of the label is an ASCII-Compatible Encoding (ACE) of a Unicode string satisfying certain constraints. For the details of the constraints, see [IDNA] and [NAMEPREP]. The prefix has not yet been specified, but see http://www.i-d-n.net/ for prefixes to be used for testing and experimentation.

DUDE is intended to be used as an ACE within IDNA, and has been designed to have the following features:

* Completeness: Every sequence of nonnegative integers maps to an LDH string. Restrictions on which integers are allowed, and on sequence length, may be imposed by higher layers.

* Uniqueness: Every sequence of nonnegative integers maps to at most one LDH string.

* Reversibility: Any Unicode string mapped to an LDH string can be recovered from that LDH string.

* Efficient encoding: The ratio of encoded size to original size is small. This is important in the context of domain names because [RFC1034] restricts the length of a domain label to 63 characters.

* Simplicity: The encoding and decoding algorithms are reasonably simple to implement. The goals of efficiency and simplicity are at odds; DUDE places greater emphasis on simplicity.

An optional feature is described in appendix C "Mixed-case annotation".

2. Terminology

The key words "must", "shall", "required", "should", "recommended", and "may" in this document are to be interpreted as described in RFC 2119 [RFC2119].

LDH characters are the letters A-Z and a-z, the digits 0-9, and hyphen-minus.

A quartet is a sequence of four bits (also known as a nibble or nybble).

A quintet is a sequence of five bits.

Hexadecimal values are shown preceeded by "0x". For example, 0x60 is decimal 96.

As in the Unicode Standard [UNICODE], Unicode code points are denoted by "U+" followed by four to six hexadecimal digits, while a range of code points is denoted by two hexadecimal numbers separated by "..", with no prefixes.

XOR means bitwise exclusive or. Given two nonnegative integer values A and B, A XOR B is the nonnegative integer value whose binary representation is 1 in whichever places the binary representations of A and B disagree, and 0 wherever they agree. For the purpose of applying this rule, recall that an integer's representation begins with an infinite number of unwritten zeros. In some programming languages, care may need to be taken that A and B are stored in variables of the same type and size.

3. Overview

DUDE encodes a sequence of nonnegative integral values as a sequence of LDH characters, although implementations will of course need to represent the output characters somehow, typically as ASCII octets. When DUDE is used to encode Unicode characters, the input values are Unicode code points (integral values in the range 0..10FFFF, but not D800..DFFF, which are reserved for use by UTF-16).

Each value in the input sequence is represented by one or more LDH characters in the encoded string. The value 0x2D is represented by hyphen-minus (U+002D). Each non-hyphen-minus character in the encoded string represents a quintet. A sequence of quintets represents the bitwise XOR between each non-0x2D integer and the previous one.

4. Base-32 characters

"a" = 0 = 0x00 = 00000	"s" = 16 = 0x10 = 10000
"b" = 1 = 0x01 = 00001	"t" = 17 = 0x11 = 10001
"c" = 2 = 0x02 = 00010	"u" = 18 = 0x12 = 10010
"d" = 3 = 0x03 = 00011	"v" = 19 = 0x13 = 10011
"e" = 4 = 0x04 = 00100	"w" = 20 = 0x14 = 10100
"f" = 5 = 0x05 = 00101	"x" = 21 = 0x15 = 10101
"g" = 6 = 0x06 = 00110	"y" = 22 = 0x16 = 10110
"h" = 7 = 0x07 = 00111	"z" = 23 = 0x17 = 10111
"i" = 8 = 0x08 = 01000	"2" = 24 = 0x18 = 11000
"j" = 9 = 0x09 = 01001	"3" = 25 = 0x19 = 11001
"k" = 10 = 0x0A = 01010	"4" = 26 = 0x1A = 11010

```
"m" = 11 = 0x0B = 01011          "5" = 27 = 0x1B = 11011
"n" = 12 = 0x0C = 01100          "6" = 28 = 0x1C = 11100
"p" = 13 = 0x0D = 01101          "7" = 29 = 0x1D = 11101
"q" = 14 = 0x0E = 01110          "8" = 30 = 0x1E = 11110
"r" = 15 = 0x0F = 01111          "9" = 31 = 0x1F = 11111
```

The digits "0" and "1" and the letters "o" and "l" are not used, to avoid transcription errors.

A decoder must accept both the uppercase and lowercase forms of the base-32 characters (including mixtures of both forms). An encoder should output only lowercase forms or only uppercase forms (unless it uses the feature described in the appendix C "Mixed-case annotation").

5. Encoding procedure

All ordering of bits, quartets, and quintets is big-endian (most significant first).

```
let prev = 0x60
for each input integer n (in order) do begin
  if n == 0x2D then output hyphen-minus
  else begin
    let diff = prev XOR n
    represent diff in base 16 as a sequence of quartets,
      as few as are sufficient (but at least one)
    prepend 0 to the last quartet and 1 to each of the others
    output a base-32 character corresponding to each quintet
    let prev = n
  end
end
```

If an encoder encounters an input value larger than expected (for example, the largest Unicode code point is U+10FFFF, and nameprep [NAMEPREP03] can never output a code point larger than U+EFFFD), the encoder may either encode the value correctly, or may fail, but it must not produce incorrect output. The encoder must fail if it encounters a negative input value.

6. Decoding procedure

```
let prev = 0x60
while the input string is not exhausted do begin
  if the next character is hyphen-minus
  then consume it and output 0x2D
  else begin
    consume characters and convert them to quintets until
      encountering a quintet whose first bit is 0
    fail upon encountering a non-base-32 character or end-of-input
    strip the first bit of each quintet
```

```
      concatenate the resulting quartets to form diff
      let prev = prev XOR diff
      output prev
    end
  end
  encode the output sequence and compare it to the input string
  fail if they do not match (case-insensitively)
```

The comparison at the end is necessary to guarantee the uniqueness property (there cannot be two distinct encoded strings representing the same sequence of integers). This check also frees the decoder from having to check for overflow while decoding the base-32 characters. (If the decoder is one step of a larger decoding process, it may be possible to defer the re-encoding and comparison to the end of that larger decoding process.)

7. Example strings

The first several examples are nonsense strings of mostly unassigned code points intended to exercise the corner cases of the algorithm.

(A) u+0061
DUDE: b

(B) u+2C7EF u+2C7EF
DUDE: u6z2ra

(C) u+1752B u+1752A
DUDE: tzxwmb

(D) u+63AB1 u+63ABA
DUDE: yv47bm

(E) u+261AF u+261BF
DUDE: uyt6rta

(F) u+C3A31 u+C3A8C
DUDE: 6v4xb5p

(G) u+09F44 u+0954C
DUDE: 39ue4si

(H) u+8D1A3 u+8C8A3
DUDE: 27t6dt3sa

(I) u+6C2B6 u+CC266
DUDE: y6u7g4ss7a

(J) u+002D u+002D u+002D u+E848F
DUDE: ---82w8r

(K) u+BD08E u+002D u+002D u+002D
    DUDE: 57s8q---

(L) u+A9A24 u+002D u+002D u+002D u+C05B7
    DUDE: 434we---y393d

(M) u+7FFFFFFF
    DUDE: z999993r or explicit failure

The next several examples are realistic Unicode strings that could be used in domain names. They exhibit single-row text, two-row text, ideographic text, and mixtures thereof. These examples are names of Japanese television programs, music artists, and songs, merely because one of the authors happened to have them handy.

(N) 3<nen>b<gumi><kinpachi><sensei> (Latin, kanji)
    u+0033 u+5E74 u+0062 u+7D44 u+91D1 u+516B u+5148 u+751F
    DUDE: xdx8whx8tgz7ug863f6s5kuduwxh

(O) <amuro><namie>-with-super-monkeys (Latin, kanji, hyphens)
    u+5B89 u+5BA4 u+5948 u+7F8E u+6075 u+002D u+0077 u+0069 u+0074
    u+0068 u+002D u+0073 u+0075 u+0070 u+0065 u+0072 u+002D u+006D
    u+006F u+006E u+006B u+0065 u+0079 u+0073
    DUDE: x58jupu8nuy6gt99m-yssctqtptn-tmgftfth-trcbfqtnk

(P) maji<de>koi<suru>5<byou><mae> (Latin, hiragana, kanji)
    u+006D u+0061 u+006A u+0069 u+3067 u+006B u+006F u+0069 u+3059
    u+308B u+0035 u+79D2 u+524D
    DUDE: pnmdvssqvssnegvsva7cvs5qz38hu53r

(Q) <pafii>de<runba> (Latin, katakana)
    u+30D1 u+30D5 u+30A3 u+30FC u+0064 u+0065 u+30EB u+30F3 u+30D0
    DUDE: vs5bezgxrvs3ibvs2qtiud

(R) <sono><supiido><de> (hiragana, katakana)
    u+305D u+306E u+30B9 u+30D4 u+30FC u+30C9 u+3067
    DUDE: vsvpvd7hypuivf4q

8. Security considerations

Users expect each domain name in DNS to be controlled by a single authority. If a Unicode string intended for use as a domain label could map to multiple ACE labels, then an internationalized domain name could map to multiple ACE domain names, each controlled by a different authority, some of which could be spoofs that hijack service requests intended for another. Therefore DUDE is designed so that each Unicode string has a unique encoding.

However, there can still be multiple Unicode representations of the "same" text, for various definitions of "same". This problem is addressed to some extent by the Unicode standard under the topic of

canonicalization, and this work is leveraged for domain names by "nameprep" [NAMEPREP03].

9. References

[IDN] Internationalized Domain Names (IETF working group), http://www.i-d-n.net/, idn@ops.ietf.org.

[IDNA] Patrik Faltstrom, Paul Hoffman, "Internationalizing Host Names In Applications (IDNA)", draft-ietf-idn-idna-01.

[NAMEPREP03] Paul Hoffman, Marc Blanchet, "Preparation of Internationalized Host Names", 2001-Feb-24, draft-ietf-idn-nameprep-03.

[RFC952] K. Harrenstien, M. Stahl, E. Feinler, "DOD Internet Host Table Specification", 1985-Oct, RFC 952.

[RFC1034] P. Mockapetris, "Domain Names - Concepts and Facilities", 1987-Nov, RFC 1034.

[RFC1123] Internet Engineering Task Force, R. Braden (editor), "Requirements for Internet Hosts -- Application and Support", 1989-Oct, RFC 1123.

[RFC2119] Scott Bradner, "Key words for use in RFCs to Indicate Requirement Levels", 1997-Mar, RFC 2119.

[SFS] David Mazieres et al, "Self-certifying File System", http://www.fs.net/.

[UNICODE] The Unicode Consortium, "The Unicode Standard", http://www.unicode.org/unicode/standard/standard.html.

A. Acknowledgements

The basic encoding of integers to quartets to quintets to base-32 comes from earlier IETF work by Martin Duerst. DUDE uses a slight variation on the idea.

Paul Hoffman provided helpful comments on this document.

The idea of avoiding 0, 1, o, and l in base-32 strings was taken from SFS [SFS].

B. Author contact information

Mark Welter <mwelter@walid.com>
Brian W. Spolarich <briansp@walid.com>
WALID, Inc.
State Technology Park

2245 S. State St.
Ann Arbor, MI 48104
+1 734 822 2020

Adam M. Costello <amc@cs.berkeley.edu>
University of California, Berkeley
http://www.cs.berkeley.edu/~amc/

C. Mixed-case annotation

In order to use DUDE to represent case-insensitive Unicode strings, higher layers need to case-fold the Unicode strings prior to DUDE encoding. The encoded string can, however, use mixed-case base-32 (rather than all-lowercase or all-uppercase as recommended in section 4 "Base-32 characters") as an annotation telling how to convert the folded Unicode string into a mixed-case Unicode string for display purposes.

Each Unicode code point (unless it is U+002D hyphen-minus) is represented by a sequence of base-32 characters, the last of which is always a letter (as opposed to a digit). If that letter is uppercase, it is a suggestion that the Unicode character be mapped to uppercase (if possible); if the letter is lowercase, it is a suggestion that the Unicode character be mapped to lowercase (if possible).

DUDE encoders and decoders are not required to support these annotations, and higher layers need not use them.

Example: In order to suggest that example O in section 7 "Example strings" be displayed as:

    <amuro><namie>-with-SUPER-MONKEYS

one could capitalize the DUDE encoding as:

    x58jupu8nuy6gt99m-yssctqtptn-tMGFtFtH-tRCBFQtNK

D. Differences from draft-ietf-idn-dude-01

Four changes have been made since draft-ietf-idn-dude-01 (DUDE-01):

1) DUDE-01 computed the XOR of each integer with the previous one in order to decide how many bits of each integer to encode, but now the XOR itself is encoded, so there is no need for a mask.

2) DUDE-01 made the first quintet of each sequence different from the rest, while now it is the last quintet that differs, so it's easier for the decoder to detect the end of the sequence.

3) The base-32 map has changed to avoid 0, 1, o, and l, to help

humans avoid transcription errors.

4) The initial value of the previous code point has changed from 0 to 0x60, making the encodings of a few domain names shorter and none longer.

E. Example implementation

```
/*****************************************/
/* dude.c 0.2.3 (2001-May-31-Thu)        */
/* Adam M. Costello <amc@cs.berkeley.edu> */
/*****************************************/

/* This is ANSI C code (C89) implementing */
/* DUDE (draft-ietf-idn-dude-02).         */


/*********************************************************/
/* Public interface (would normally go in its own .h file): */

#include <limits.h>

enum dude_status {
  dude_success,
  dude_bad_input,
  dude_big_output  /* Output would exceed the space provided. */
};

enum case_sensitivity { case_sensitive, case_insensitive };

#if UINT_MAX >= 0x1FFFFF
typedef unsigned int u_code_point;
#else
typedef unsigned long u_code_point;
#endif

enum dude_status dude_encode(
  unsigned int input_length,
  const u_code_point input[],
  const unsigned char uppercase_flags[],
  unsigned int *output_size,
  char output[] );

    /* dude_encode() converts Unicode to DUDE (without any           */
    /* signature).  The input must be represented as an array        */
    /* of Unicode code points (not code units; surrogate pairs       */
    /* are not allowed), and the output will be represented as       */
    /* null-terminated ASCII.  The input_length is the number of code */
```

```
    /* points in the input.  The output_size is an in/out argument:   */
    /* the caller must pass in the maximum number of characters       */
    /* that may be output (including the terminating null), and on    */
    /* successful return it will contain the number of characters     */
    /* actually output (including the terminating null, so it will be */
    /* one more than strlen() would return, which is why it is called */
    /* output_size rather than output_length).  The uppercase_flags   */
    /* array must hold input_length boolean values, where nonzero     */
    /* means the corresponding Unicode character should be forced     */
    /* to uppercase after being decoded, and zero means it is         */
    /* caseless or should be forced to lowercase.  Alternatively,     */
    /* uppercase_flags may be a null pointer, which is equivalent     */
    /* to all zeros.  The encoder always outputs lowercase base-32    */
    /* characters except when nonzero values of uppercase_flags       */
    /* require otherwise.  The return value may be any of the         */
    /* dude_status values defined above; if not dude_success, then    */
    /* output_size and output may contain garbage.  On success, the   */
    /* encoder will never need to write an output_size greater than   */
    /* input_length*k+1 if all the input code points are less than 1  */
    /* << (4*k), because of how the encoding is defined.              */

enum dude_status dude_decode(
  enum case_sensitivity case_sensitivity,
  char scratch_space[],
  const char input[],
  unsigned int *output_length,
  u_code_point output[],
  unsigned char uppercase_flags[] );

    /* dude_decode() converts DUDE (without any signature) to         */
    /* Unicode.  The input must be represented as null-terminated     */
    /* ASCII, and the output will be represented as an array of       */
    /* Unicode code points.  The case_sensitivity argument influences */
    /* the check on the well-formedness of the input string; it       */
    /* must be case_sensitive if case-sensitive comparisons are       */
    /* allowed on encoded strings, case_insensitive otherwise.        */
    /* The scratch_space must point to space at least as large        */
    /* as the input, which will get overwritten (this allows the      */
    /* decoder to avoid calling malloc()).  The output_length is      */
    /* an in/out argument: the caller must pass in the maximum        */
    /* number of code points that may be output, and on successful    */
    /* return it will contain the actual number of code points        */
    /* output.  The uppercase_flags array must have room for at       */
    /* least output_length values, or it may be a null pointer if     */
    /* the case information is not needed.  A nonzero flag indicates  */
    /* that the corresponding Unicode character should be forced to   */
    /* uppercase by the caller, while zero means it is caseless or    */
    /* should be forced to lowercase.  The return value may be any    */
    /* of the dude_status values defined above; if not dude_success,  */
    /* then output_length, output, and uppercase_flags may contain    */
    /* garbage.  On success, the decoder will never need to write     */
```

```
    /* an output_length greater than the length of the input (not     */
    /* counting the null terminator), because of how the encoding is  */
    /* defined.                                                       */


/*********************************************************/
/* Implementation (would normally go in its own .c file): */

#include <string.h>

/* Character utilities: */

/* base32[q] is the lowercase base-32 character representing  */
/* the number q from the range 0 to 31.  Note that we cannot  */
/* use string literals for ASCII characters because an ANSI C */
/* compiler does not necessarily use ASCII.                   */

static const char base32[] = {
  97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107,     /* a-k */
  109, 110,                                               /* m-n */
  112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122,  /* p-z */
  50, 51, 52, 53, 54, 55, 56, 57                          /* 2-9 */
};

/* base32_decode(c) returns the value of a base-32 character, in the */
/* range 0 to 31, or the constant base32_invalid if c is not a valid */
/* base-32 character.                                                */

enum { base32_invalid = 32 };

static unsigned int base32_decode(char c)
{
  if (c < 50) return base32_invalid;
  if (c <= 57) return c - 26;
  if (c < 97) c += 32;
  if (c < 97 || c == 108 || c == 111 || c > 122) return base32_invalid;
  return c - 97 - (c > 108) - (c > 111);
}

/* unequal(case_sensitivity,s1,s2) returns 0 if the strings s1 and s2 */
/* are equal, 1 otherwise.  If case_sensitivity is case_insensitive,  */
/* then ASCII A-Z are considered equal to a-z respectively.           */

static int unequal( enum case_sensitivity case_sensitivity,
                    const char s1[], const char s2[]        )
{
  char c1, c2;

  if (case_sensitivity != case_insensitive) return strcmp(s1,s2) != 0;

  for (;;) {
```

```
    c1 = *s1;
    c2 = *s2;
    if (c1 >= 65 && c1 <= 90) c1 += 32;
    if (c2 >= 65 && c2 <= 90) c2 += 32;
    if (c1 != c2) return 1;
    if (c1 == 0) return 0;
    ++s1, ++s2;
  }
}


/* Encoder: */

enum dude_status dude_encode(
  unsigned int input_length,
  const u_code_point input[],
  const unsigned char uppercase_flags[],
  unsigned int *output_size,
  char output[] )
{
  unsigned int max_out, in, out, k, j;
  u_code_point prev, codept, diff, tmp;
  char shift;

  prev = 0x60;
  max_out = *output_size;

  for (in = out = 0;  in < input_length;  ++in) {

    /* At the start of each iteration, in and out are the number of */
    /* items already input/output, or equivalently, the indices of  */
    /* the next items to be input/output.                           */

    codept = input[in];

    if (codept == 0x2D) {
      /* Hyphen-minus stands for itself. */
      if (max_out - out < 1) return dude_big_output;
      output[out++] = 0x2D;
      continue;
    }

    diff = prev ^ codept;

    /* Compute the number of base-32 characters (k): */
    for (tmp = diff >> 4, k = 1;  tmp != 0;  ++k, tmp >>= 4);

    if (max_out - out < k) return dude_big_output;
    shift = uppercase_flags && uppercase_flags[in] ? 32 : 0;
    /* shift controls the case of the last base-32 digit. */
```

```
    /* Each quintet has the form 1xxxx except the last is 0xxxx. */
    /* Computing the base-32 digits in reverse order is easiest. */

    out += k;
    output[out - 1] = base32[diff & 0xF] - shift;

    for (j = 2;  j <= k;  ++j) {
      diff >>= 4;
      output[out - j] = base32[0x10 | (diff & 0xF)];
    }

    prev = codept;
  }

  /* Append the null terminator: */
  if (max_out - out < 1) return dude_big_output;
  output[out++] = 0;

  *output_size = out;
  return dude_success;
}


/* Decoder: */

enum dude_status dude_decode(
  enum case_sensitivity case_sensitivity,
  char scratch_space[],
  const char input[],
  unsigned int *output_length,
  u_code_point output[],
  unsigned char uppercase_flags[] )
{
  u_code_point prev, q, diff;
  char c;
  unsigned int max_out, in, out, scratch_size;
  enum dude_status status;

  prev = 0x60;
  max_out = *output_length;

  for (c = input[in = 0], out = 0;  c != 0;  c = input[++in], ++out) {

    /* At the start of each iteration, in and out are the number of */
    /* items already input/output, or equivalently, the indices of  */
    /* the next items to be input/output.                           */

    if (max_out - out < 1) return dude_big_output;

    if (c == 0x2D) output[out] = c;  /* hyphen-minus is literal */
    else {
```

```
      /* Base-32 sequence.  Decode quintets until 0xxxx is found: */

      for (diff = 0;  ;  c = input[++in]) {
        q = base32_decode(c);
        if (q == base32_invalid) return dude_bad_input;
        diff = (diff << 4) | (q & 0xF);
        if (q >> 4 == 0) break;
      }

      prev = output[out] = prev ^ diff;
    }

    /* Case of last character determines uppercase flag: */
    if (uppercase_flags) uppercase_flags[out] = c >= 65 && c <= 90;
  }

  /* Enforce the uniqueness of the encoding by re-encoding */
  /* the output and comparing the result to the input:     */

  scratch_size = ++in;
  status = dude_encode(out, output, uppercase_flags,
                       &scratch_size, scratch_space);
  if (status != dude_success || scratch_size != in ||
      unequal(case_sensitivity, scratch_space, input)
     ) return dude_bad_input;

  *output_length = out;
  return dude_success;
}


/****************************************************************/
/* Wrapper for testing (would normally go in a separate .c file): */

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

/* For testing, we'll just set some compile-time limits rather than */
/* use malloc(), and set a compile-time option rather than using a  */
/* command-line option.                                             */

enum {
  unicode_max_length = 256,
  ace_max_size = 256,
  test_case_sensitivity = case_insensitive
                          /* suitable for host names */
};
```

```
static void usage(char **argv)
{
  fprintf(stderr,
    "%s -e reads code points and writes a DUDE string.\n"
    "%s -d reads a DUDE string and writes code points.\n"
    "Input and output are plain text in the native character set.\n"
    "Code points are in the form u+hex separated by whitespace.\n"
    "A DUDE string is a newline-terminated sequence of LDH characters\n"
    "(without any signature).\n"
    "The case of the u in u+hex is the force-to-uppercase flag.\n"
    , argv[0], argv[0]);
  exit(EXIT_FAILURE);
}


static void fail(const char *msg)
{
  fputs(msg,stderr);
  exit(EXIT_FAILURE);
}

static const char too_big[] =
  "input or output is too large, recompile with larger limits\n";
static const char invalid_input[] = "invalid input\n";
static const char io_error[] = "I/O error\n";


/* The following string is used to convert LDH      */
/* characters between ASCII and the native charset: */

static const char ldh_ascii[] =
  "................"
  "................"
  ".............-.."
  "0123456789......"
  ".ABCDEFGHIJKLMNO"
  "PQRSTUVWXYZ....."
  ".abcdefghijklmno"
  "pqrstuvwxyz";


int main(int argc, char **argv)
{
  enum dude_status status;
  int r;
  char *p;

  if (argc != 2) usage(argv);
  if (argv[1][0] != '-') usage(argv);
  if (argv[1][2] != 0) usage(argv);
```

```
if (argv[1][1] == 'e') {
  u_code_point input[unicode_max_length];
  unsigned long codept;
  unsigned char uppercase_flags[unicode_max_length];
  char output[ace_max_size], uplus[3];
  unsigned int input_length, output_size, i;

  /* Read the input code points: */

  input_length = 0;

  for (;;) {
    r = scanf("%2s%lx", uplus, &codept);
    if (ferror(stdin)) fail(io_error);
    if (r == EOF || r == 0) break;

    if (r != 2 || uplus[1] != '+' || codept > (u_code_point)-1) {
      fail(invalid_input);
    }

    if (input_length == unicode_max_length) fail(too_big);

    if (uplus[0] == 'u') uppercase_flags[input_length] = 0;
    else if (uplus[0] == 'U') uppercase_flags[input_length] = 1;
    else fail(invalid_input);

    input[input_length++] = codept;
  }

  /* Encode: */

  output_size = ace_max_size;
  status = dude_encode(input_length, input, uppercase_flags,
                       &output_size, output);
  if (status == dude_bad_input) fail(invalid_input);
  if (status == dude_big_output) fail(too_big);
  assert(status == dude_success);

  /* Convert to native charset and output: */

  for (p = output;  *p != 0;  ++p) {
    i = *p;
    assert(i <= 122 && ldh_ascii[i] != '.');
    *p = ldh_ascii[i];
  }

  r = puts(output);
  if (r == EOF) fail(io_error);
  return EXIT_SUCCESS;
}
```

```
  if (argv[1][1] == 'd') {
    char input[ace_max_size], scratch[ace_max_size], *pp;
    u_code_point output[unicode_max_length];
    unsigned char uppercase_flags[unicode_max_length];
    unsigned int input_length, output_length, i;

    /* Read the DUDE input string and convert to ASCII: */

    fgets(input, ace_max_size, stdin);
    if (ferror(stdin)) fail(io_error);
    if (feof(stdin)) fail(invalid_input);
    input_length = strlen(input);
    if (input[input_length - 1] != '\n') fail(too_big);
    input[--input_length] = 0;

    for (p = input;  *p != 0;  ++p) {
      pp = strchr(ldh_ascii, *p);
      if (pp == 0) fail(invalid_input);
      *p = pp - ldh_ascii;
    }

    /* Decode: */

    output_length = unicode_max_length;
    status = dude_decode(test_case_sensitivity, scratch, input,
                         &output_length, output, uppercase_flags);
    if (status == dude_bad_input) fail(invalid_input);
    if (status == dude_big_output) fail(too_big);
    assert(status == dude_success);

    /* Output the result: */

    for (i = 0;  i < output_length;  ++i) {
      r = printf("%s+%04lX\n",
                 uppercase_flags[i] ? "U" : "u",
                 (unsigned long) output[i] );
      if (r < 0) fail(io_error);
    }

    return EXIT_SUCCESS;
  }

  usage(argv);
  return EXIT_SUCCESS;  /* not reached, but quiets compiler warning */
}
```