The PKtype processor

(Version 2.3, 23 April 2020)

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926 and MCS-8300984, and by the System Development Foundation.

2* The *banner* string defined here should be changed whenever `PKtype` gets modified.

define *my_name* ≡ `´pktype´`
define *banner* ≡ `´This␣is␣PKtype,␣Version␣2.3´` { printed when the program starts }

4* Both the input and output come from binary files. On line interaction is handled through Pascal's standard *input* and *output* files. Two macros are used to write to the type file, so this output can easily be redirected.

define *print_ln*(#) ≡ *write_ln*(*output*, #)
define *print*(#) ≡ *write*(*output*, #)
define *typ_file* ≡ *stdout*
define *t_print_ln*(#) ≡ *write_ln*(*typ_file*, #)
define *t_print*(#) ≡ *write*(*typ_file*, #)

program *PKtype*(*input*, *output*);
type ⟨ Types in the outer block 9 ⟩
var ⟨ Globals in the outer block 11 ⟩
⟨ Define *parse_arguments* 56* ⟩
procedure *initialize*; { this procedure gets things started properly }
var *i*: *integer*; { loop index for initializations }
begin *kpse_set_program_name*(*argv*[0], *my_name*); *kpse_init_prog*(`´PKTYPE´`, 0, **nil**, **nil**);
parse_arguments; *print*(*banner*); *print_ln*(*version_string*);
⟨ Set initial values 12 ⟩
end;

5* This module is deleted, because it is only useful for a non-local **goto** , which we don't use in C.

6* These constants determine the maximum length of a file name and the length of the terminal line, as well as the widest character that can be translated.

8* We use a call to the external C exit to avoid a non-local **goto** .

define *abort*(#) ≡
begin *print_ln*(#); *uexit*(1)
end

10* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lower case letters. Nowadays, of course, we need to deal with both upper and lower case alphabets in a convenient way, especially in a program like `PKtype`. So we shall assume that the Pascal system being used for `PKtype` has a character set containing at least the standard visible characters of ASCII code (`"!"` through `"~"`).

Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name. In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the output file. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *char* $\equiv$ 0 .. 255
define *text_char* $\equiv$ *char* { the data type of characters in text files }
define *first_text_char* = 0 { ordinal number of the smallest element of *text_char* }
define *last_text_char* = 127 { ordinal number of the largest element of *text_char* }

⟨ Types in the outer block 9 ⟩ +≡
text_file = **packed file of** *text_char*;

31* ⟨ Globals in the outer block 11 ⟩ +≡
pk_file: *byte_file*; { where the input comes from }

32* In C, do path searching.

procedure *open_pk_file*; { prepares to read packed bytes in *pk_file* }
  begin { Don't use *kpse_find_pk*; we want the exact file or nothing. }
  pk_file ← *kpse_open_file*(*cmdline*(1), *kpse_pk_format*); *cur_loc* ← 0;
  end;

33* We need a place to store the names of the input and output file, as well as a byte counter for the output file.

⟨ Globals in the outer block 11 ⟩ +≡
pk_name: *c_string*; { name of input and output files }
cur_loc: *integer*; { how many bytes have we read? }

34* We shall use a set of simple functions to read the next byte or bytes from *pk_file*. There are seven possibilities, each of which is treated as a separate function in order to minimize the overhead for subroutine calls. We comment out the ones we don't need.

```
define pk_byte ≡ get_byte
define pk_loc ≡ cur_loc
function get_byte: integer;  { returns the next byte, unsigned }
  var b: eight_bits;
  begin if eof(pk_file) then get_byte ← 0
  else begin read(pk_file, b); incr(cur_loc); get_byte ← b;
    end;
  end;
  @{
  function signed_byte: integer;  { returns the next byte, signed }
    var b: eight_bits;
    begin read(pk_file, b); incr(cur_loc);
    if b < 128 then signed_byte ← b else signed_byte ← b − 256;
    end;
  @}
function get_two_bytes: integer;  { returns the next two bytes, unsigned }
  var a, b: eight_bits;
  begin read(pk_file, a); read(pk_file, b); cur_loc ← cur_loc + 2; get_two_bytes ← a ∗ 256 + b;
  end;
  @{
  function signed_pair: integer;  { returns the next two bytes, signed }
    var a, b: eight_bits;
    begin read(pk_file, a); read(pk_file, b); cur_loc ← cur_loc + 2;
    if a < 128 then signed_pair ← a ∗ 256 + b
    else signed_pair ← (a − 256) ∗ 256 + b;
    end;
  @}
  @{
  function get_three_bytes: integer;  { returns the next three bytes, unsigned }
    var a, b, c: eight_bits;
    begin read(pk_file, a); read(pk_file, b); read(pk_file, c); cur_loc ← cur_loc + 3;
    get_three_bytes ← (a ∗ 256 + b) ∗ 256 + c;
    end;
  @}
  @{
  function signed_trio: integer;  { returns the next three bytes, signed }
    var a, b, c: eight_bits;
    begin read(pk_file, a); read(pk_file, b); read(pk_file, c); cur_loc ← cur_loc + 3;
    if a < 128 then signed_trio ← (a ∗ 256 + b) ∗ 256 + c
    else signed_trio ← ((a − 256) ∗ 256 + b) ∗ 256 + c;
    end;
  @}
function signed_quad: integer;  { returns the next four bytes, signed }
  var a, b, c, d: eight_bits;
  begin read(pk_file, a); read(pk_file, b); read(pk_file, c); read(pk_file, d); cur_loc ← cur_loc + 4;
  if a < 128 then signed_quad ← ((a ∗ 256 + b) ∗ 256 + c) ∗ 256 + d
  else signed_quad ← (((a − 256) ∗ 256 + b) ∗ 256 + c) ∗ 256 + d;
  end;
```

35* This module was needed when output was directed to *typ_file*. It is not needed when output goes to *stdout*.

36* As we are reading the packed file, we often need to fetch 16 and 32 bit quantities. Here we have two procedures to do this.

define *get_16* ≡ *get_two_bytes*
define *get_32* ≡ *signed_quad*

52* If any specials are found, we write them out here.

define *four_cases*(#) ≡ #, # + 1, # + 2, # + 3

```
procedure skip_specials;
  var i, j: integer;
  begin repeat flag_byte ← pk_byte;
    if flag_byte ≥ 240 then
      case flag_byte of
      four_cases(pk_xxx1): begin t_print((pk_loc − 1) : 1, ´:␣␣Special:␣´´´); i ← 0;
        for j ← pk_xxx1 to flag_byte do i ← 256 ∗ i + pk_byte;
        for j ← 1 to i do t_print(xchr[pk_byte]);
        t_print_ln(´´´´);
        end;
      pk_yyy: begin t_print((pk_loc − 1) : 1); t_print_ln(´:␣␣Num␣special:␣´, get_32 : 1);
        end;
      pk_post: t_print_ln((pk_loc − 1) : 1, ´:␣␣Postamble´);
      pk_no_op: t_print_ln((pk_loc − 1) : 1, ´:␣␣No␣op´);
      pk_pre, pk_undefined: abort(´Unexpected␣´, flag_byte : 1, ´!´);
      endcases;
  until (flag_byte < 240) ∨ (flag_byte = pk_post);
  end;
```

53*. Terminal communication. There isn't any.

54*. So there is no **procedure** *dialog*.

55*. The main program. Now that we have all the pieces written, let us put them together.

```
begin initialize; open_pk_file; ⟨Read preamble 38⟩;
skip_specials;
while flag_byte ≠ pk_post do
  begin ⟨Unpack and write character 40⟩;
  skip_specials;
  end;
j ← 0;
while ¬eof(pk_file) do
  begin i ← pk_byte;
  if i ≠ pk_no_op then abort('Bad␣byte␣at␣end␣of␣file:␣', i : 1);
  t_print_ln((pk_loc − 1) : 1, ':␣␣No␣op'); incr(j);
  end;
t_print_ln(pk_loc : 1, '␣bytes␣read␣from␣packed␣file.');
end.
```

56*. System-dependent changes. Parse a Unix-style command line.

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)

⟨ Define parse_arguments 56* ⟩ ≡
procedure parse_arguments;
  const n_options = 2;  { Pascal won't count array lengths for us. }
  var long_options: array [0 .. n_options] of getopt_struct;
    getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
  begin ⟨ Define the option table 57* ⟩;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin do_nothing;
      end
    else if getopt_return_val = ´?´ then
        begin usage(my_name);
        end
      else if argument_is(´help´) then
          begin usage_help(PKTYPE_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´Tomas␣Rokicki´, nil);
            end;  { Else it was just a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. }
  if (optind + 1 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣exactly␣one␣file␣argument.´); usage(my_name);
    end;
  end;
```

This code is used in section 4*.

57*. Here are the options we allow. The first is one of the standard GNU options.

```
⟨ Define the option table 57* ⟩ ≡
  current_option ← 0; long_options[current_option].name ← ´help´;
  long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
  long_options[current_option].val ← 0; incr(current_option);
```

See also sections 58* and 59*.

This code is used in section 56*.

58*. Another of the standard options.

```
⟨ Define the option table 57* ⟩ +≡
  long_options[current_option].name ← ´version´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

59*. An element with all zeros always ends the list.

```
⟨ Define the option table 57* ⟩ +≡
  long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```

60* Index. Pointers to error messages appear here together with the section numbers where each identifier is used.

The following sections were changed by the change file: 2, 4, 5, 6, 8, 10, 31, 32, 33, 34, 35, 36, 52, 53, 54, 55, 56, 57, 58, 59, 60.