1. Introduction. This program is a quick-and-dirty hack to convert Fontographer Type 3 font output to METAFONT code. I assume that the input file has been hand-converted to a mixture of the `afm` file and the `ps` file output by Fontographer; I also assume that the output file will be hand-tailored to make a complete METAFONT program.

More precisely, this program reads blocks of material like

```
C 33 ; WX 220 ; N exclam ; B 34 442 187 664 ;
/exclam{220 0 4.0426 422.858 217.66 683.983 Cache
187.702 623.194 moveto
-0.297607 -29.4158 -111.106 -124.124 -24 -28 rrcurveto
-19 10 rlineto
16.42 45.8054 35.7433 110.536 23 36 rrcurveto
8.12329 12.7366 11.0454 6.84058 15.4363 -0.159424 rrcurveto
23.3605 -0.159424 21.7024 -18.0087 -0.297607 -23.8059 rrcurveto
closepath
0 FillStroke
}def
```

and writes corresponding blocks of material like

```
beginchar(33,220u#,664u#,0u#);
stroke (188,623)
 ...(187,594,76,470,52,442)
 --(33,452)
 ...(50,497,85,608,108,644)
 ...(117,657,128,664,143,663)
 ...(166,663,188,645,188,621)
 --cycle;
endchar;
```

(operating from standard input to standard output).

It does absolutely nothing fancy.

On a closed shape like the letter O, the user has to change some `stroke` commands to `unstroke`, because Fontographer gives the outside contour and then the inside contour (in opposite directions). The inside contour needs to be erased, not filled, so we want to `unstroke` it.

```
#include <stdio.h>
  char buffer[100], *pos = buffer;
  char token[100];
  ⟨ Subroutines for input 2 ⟩
  main( )
  {
    register int j, k;
    double x, y, z;
    register char *p, *q;
    while (1) {
      ⟨ Process font metric info 4 ⟩;
      ⟨ Process stroke info 5 ⟩;
    }
  }
```

2. Low-level input. At the bottom I need a way to parse the input into tokens. A token is either a number or a string of nonblank characters.

To make things simple, *get_token* just finds a string of nonblank characters. The calling routine will easily be able to convert a numeric string to the actual number.

```
#define get_token gtok()
⟨ Subroutines for input 2 ⟩ ≡
  gtok()
  {
    register char *p;
    if (*pos ≡ 0 ∨ *pos ≡ '\n') {
      if (¬fgets(buffer, 100, stdin)) exit(0);      /* normal exit at end of file */
      pos = buffer;
    }
    for ( ; *pos ≡ '␣'; pos++) ;      /* move to next nonspace */
    for (p = token; *pos ≠ '␣' ∧ *pos ∧ *pos ≠ '\n'; p++, pos++) *p = *pos;
    *p = 0;
    for ( ; *pos ≡ '␣'; pos++) ;
  }
```

See also section 3.

This code is used in section 1.

3. If the input contains any surprises, we give up immediately.

```
#define get_num gnum()
#define panic(str)
        {
          fprintf(stderr, "Oops!␣%s:\n%s", str, buffer);
          exit(−1);
        }
⟨ Subroutines for input 2 ⟩ +≡
  double gnum()
  {
    double xx;
    if (sscanf(token, "%lf", &xx) ≠ 1) panic("Unreadable␣number");
    return xx;
  }
```

4. Reading the font metrics. If the first line of the input is, say,

```
C 36 ; WX 482 ; N dollar ; B 23 -205 437 751 ;
```

we want to define character number 36, whose width is 482 units. The name of the character is unimportant (Fontographer assigned it based solely of the character number). The bounding box is also mostly unimportant except for the y coordinates; in this example we give the character a depth of 205 units and a height of 751.

Another line such as

```
/dollar{482 0 -44.1428 -260.8 504.143 806.8 Cache
```

immediately follows in the input, but we totally ignore it.

```
#define check(str, err)
        {
          get_token;
          if (strcmp(token, str) ≠ 0) panic(err);
        }
⟨ Process font metric info 4 ⟩ ≡
  check("C", "Expected␣‘C’");
  get_token;
  printf("beginchar(%s,", token);
  check(";", "Expected␣‘;’");
  check("WX", "Expected␣‘WX’")get_token;
  printf("%su#,", token);
  check(";", "Expected␣‘;’");
  check("N", "Expected␣‘N’");
  get_token;
  check(";", "Expected␣‘;’");
  check("B", "Expected␣‘B’");
  get_token;
  get_token;
  k = (int)(get_num + .5);
  if (k > 0) k = 0;
  else k = −k;
  get_token;
  get_token;
  printf("%su#,%du#);\n", token, k);
  check(";", "Expected␣‘;’");
  get_token;
  check("0", "Expected␣‘0’");
  get_token;
  get_token;
  get_token;
  get_token;
  check("Cache", "Expected␣‘Cache’");
```

This code is used in section 1.

5. The strokes. Each shape to be filled is presented as a sequence of lines beginning with 'x y `moveto`' and followed by lines that say either 'x y `rlineto`' or 'x_1 y_1 x_2 y_2 x_3 y_3 `rrcurveto`'; finally '`closepath`' ends the shape. Each pair (x, y) is an increment to be added to the previous coordinates.

The final stroke is followed by '`0 Fillstroke }def`'.

```
⟨ Process stroke info 5 ⟩ ≡
  while (1) {
    get_token;
    x = get_num;
    get_token;
    if (strcmp(token, "FillStroke") ≡ 0) break;
    y = get_num;
    check("moveto", "Expected␣'moveto'");
    printf("stroke␣(%d,%d)\n", (int)(x + .5), (int)(y + .5));
    while (1) {
      get_token;
      if (strcmp(token, "closepath") ≡ 0) break;
      x += get_num;
      get_token;
      y += get_num;
      get_token;
      if (strcmp(token, "rlineto") ≡ 0) printf("␣--(%d,%d)\n", (int)(x + .5), (int)(y + .5));
      else {
        printf("␣...(%d,%d", (int)(x + .5), (int)(y + .5));
        x += get_num;
        get_token;
        y += get_num;
        printf(",%d,%d", (int)(x + .5), (int)(y + .5));
        get_token;
        x += get_num;
        get_token;
        y += get_num;
        printf(",%d,%d)\n", (int)(x + .5), (int)(y + .5));
        check("rrcurveto", "Expected␣'rrcurveto'");
      }
    }
    ;
    printf("␣--cycle;\n");
  }
  printf("endchar;\n");
  check("}def", "Expected␣'}def'");
```

This code is used in section 1.

6. Index.

⟨ Process font metric info 4 ⟩ Used in section 1.
⟨ Process stroke info 5 ⟩ Used in section 1.
⟨ Subroutines for input 2, 3 ⟩ Used in section 1.

FOG2MF