WeaveFont: A Description Language for Typographical Fonts

Thiago Leucz Astrizi

thiago@bitbitbit.com.br

Abstract: *This article contains an implementation of WeaveFont, a language created to represent typographical fonts, vector illustrations and animations. The language is strongly based on METAFONT, a description language created by Donald Knuth in 1984. The syntax is very similar for both languages, but WeaveFont was created to run and produce images in real time using OpenGL. Therefore, changes were made to make the language faster for this goal. Contrary to METAFONT, WeaveFont do not support macros and is an imperative language. Several useful features from METAFONT that were implemented with macros, in WeaveFont are a direct part of the language.*

Index

1. Introduction

The first description language for typographical fonts was METAFONT. It was created

on 1984 by Donald Knuth and differs from *other formats for allowing a designer to create different fonts merely by changing basic parameters in the base description of the font. This way, a designer should not create a single typographic font, but a meta-font from which new fonts could be obtained changing these basic parameters. Here we also will call "meta-font" the fonts defined by our system, which should not be confused with the METAFONT language.

The original specification for the METAFONT language can be found in [KNUTH, 1989]. Based on this language, we will define here the language WeaveFont, which will have similar goals, except that it will be focused on defining typographical fonts that could be interpreted and rendered on the fly.

The source-code defined in this article can be used standalone, or together with more files from Weaver Geme Engine, as the language WeaveFont is a part of that bigger project. When we run the source code from inside the game engine, then we have the macro `WEAVER_ENGINE` defined. In this case, we must define the following function:

Section: Function Declaration (metafont.h):

```
#if defined(WEAVER_ENGINE)
void _Wmetafont_loading(void *(*permanent_alloc)(size_t),
                        void (*permanent_free)(void *),
                        void *(*temporary_alloc)(size_t),
                        void (*temporary_free)(void *),
                        void (*before_loading_interface)(void),
                        void (*after_loading_interface)(void),
                        char *source_filename,
                        struct user_interface *target);
#endif
```

This function will read a file (`source_filename`) and create after interpret its WeaveFont code a new vector image or a vector animation.

And we will also need the header for Weaver user interfaces:

Section: Include General Headers (metafont.h):

```
#if defined(WEAVER_ENGINE)
#include "interface.h"
#endif
```

Either if we are running our code from inside the Weaver Engine or not, we always will need a function to load a new typographic font, a meta-font from a file. After reading it, we can render the characters defined there. Add after this, we will need to deallocate the struct containing all the data from that font. Therefore, we will need to export a function to create and another to destroy a meta-font read from a file:

Section: Function Declaration (metafont.h) (continuation):

```
struct metafont *_Wnew_metafont(char *filename);
void _Wdestroy_metafont(struct metafont *mf);
```

In a single project, we could read and keep in memory several different meta-fonts to render the characters and images stored in them. But before creating the first one, it is necessary to call a initialization function that sets which functions will be called in some contexts and also how many dots per inch has our screen (DPI):

Section: Function Declaration (metafont.h) (continuation):

```
bool _Winit_weavefont(void *(*temporary_alloc)(size_t),
                      void (*temporary_free)(void *),
                      void *(*permanent_alloc)(size_t),
                      void (*permanent_free)(void *),
```

```
                    uint64_t (*rand)(void), int dpi);
```

The functions passed as parameters are respectivelly the one that allocates memory temporarily , other to free what was allocated by it, a function to make more permanent allocation, the function that frees what was allocated by it, a function that returns 64 random bits and finally our DPI. The disallocation functions can be set to NULL. The function returns true if the initialization was successful.

After using this library API, the function below should be called to finalize it:

Section: Function Declaration (metafont.h) (continuation):

```
void _Wfinish_weavefont(void);
```

If an error was found in a WeaveFont source code that defines a meta-font, then the following exported function could be used to print a diagnostic message in the screen:

Section: Function Declaration (metafont.h) (continuation):

```
void _Wprint_metafont_error(struct metafont *);
```

1.1. Literate Programming

Our API will be written using the literate programming technique, proposed by Knuth on [Knuth, 1984]. It consist in writting a computer program explaining didactically in a text what is being done while presenting the code. The program is compiled extracting the computer code directly from the didactical text. The code shall be presented in a way and order such that it is best for explaining for a human. Not how it would be easier to compile.

Using this technique, this document is not a simple documentation for our code. It is the code by itself. The part that will be extracted to be compiled can be identified by a gray background. We begin each piece of code by a title that names it. For example, immediately before this subsection we presented a series of function declarations. And how one could deduct by the title, most of them will be positioned in the file `metafont.h`.

We can show the structure of the file `metafont.h`:

File: `src/metafont.h`:

```
#ifndef __WEAVER_METAFONT
#define __WEAVER_METAFONT
#ifdef __cplusplus
extern "C" {
#endif
#include <stdbool.h> // Define  'bool' type
#include <stdlib.h> // Define 'atof' and 'abs' functions
#if defined(__linux__) || defined(BSD) || defined(__EMSCRIPTEN__)
#include <GLES3/gl3.h> // Our language renders with OpenGL
#endif
#if !defined(_WIN32)
#include <sys/param.h> // Needed on BSD, but does not exist on Windows
#endif
            <Section to be inserted: Include General Headers (metafont.h)>
              <Section to be inserted: General Declarations (metafont.h)>
                <Section to be inserted: Data Structures (metafont.h)>
              <Section to be inserted: Function Declaration (metafont.h)>
#ifdef __cplusplus
}
#endif
#endif
```

The cde above shows the default boureaucracy to define a header for our C API. The two first lines and the last one are macros that ensure that this header will not be inserted more than once in a single

compiling unit. The lines 3, 4, 5 and the three lines before the last one make the header adequate to be used in C++ code. This tells the compiler that we are using C code and that therefore, the compiler is free to use optimizations assuming that we will not use C++ exclusive techniques, like operator overloading. Next we include a header that will let us to use boolean variables. And there are some parts in red. Note that one of them is called "Function Declaration (weaver.h)", the same title used in most of the code declared previously. This means that all the previous code with this title will be inserted in that position inside this file. The other parts in red represent code that we will define in the next sections.

If you want to know how is the `metafont.c` file related with this header, its structure is:

File: `src/metafont.c`:

```
#include "metafont.h"
<Section to be inserted: Local Headers (metafont.c)>
<Section to be inserted: Local Macros (metafont.c)>
<Section to be inserted: Local Data Structures (metafont.c)>
<Section to be inserted: Local Variables (metafont.c)>
<Section to be inserted: Local Function Declaration (metafont.c)>
<Section to be inserted: Auxiliary Local Functions (metafont.c)>
<Section to be inserted: API Functions Definition (metafont.c)>
```

All the code presented in this document will be placed in one of these two files. Besides them, no other file will be created.

2. General Auxiliary Code

The code presented in this Section will be an introduction about how we define code in this article. All code here will be widely used in the next Sections and all code will be independant of the specific data structure used by Weaver Metafont.

2.1. Linear Algebra Code

We will use very frequently matrices 3×3, that usually will represent linear transformations over a vector space with 3 dimensions. Such matrices will be an array of 9 elements, representing the matrix content.

The matrix elements will be organized as below, according with how they are stored in array M:

$$\begin{bmatrix} M[0] & M[1] & M[2] \\ M[3] & M[4] & M[5] \\ M[6] & M[7] & M[8] \end{bmatrix}$$

We can initialize an identity matrix with:

Section: Local Macros (metafont.c):

```
#define INITIALIZE_IDENTITY_MATRIX(I) {\
  int _i;\
  for(_i = 0; _i < 9; _i++)\
    I[_i] = ((_i%4)?(0.0):(1.0));\
}
```

Despite our dealings in a vector space with 3 dimensions, on practice all the values we deal will be contained in the plane $\{(x, y, 1)\}$ with x and y being real numbers. The reason for working in 3 dimensions is that we want to represent translations (or shifting points) as linear transforms, and this is only possible if we assume that our 2D plane is inside a 3D vector space.

Given a 3-dimentional vector $(x, y, 1)$, we can transform it in a new vector $(x', y', 1)$ applying a linear transformation represented by the matrix below:

$$[x\ y\ 1] \begin{bmatrix} M[0] & M[1] & 0 \\ M[3] & M[4] & 0 \\ M[6] & M[7] & 1 \end{bmatrix} = [x'\ \ y'\ \ 1]$$

The new coordinates (x', y') can be computed using the macros below:

Section: Local Macros (metafont.c) (continuation):

```
#define LINEAR_TRANSFORM_X(x, y, M) (x * M[0] + y * M[3] + M[6])
#define LINEAR_TRANSFORM_Y(x, y, M) (x * M[1] + y * M[4] + M[7])
```

The matrix multiplication is associative as seen below:

$$((x, y, 1) \cdot A) \cdot B = (x, y, 1) \cdot (A \cdot B)$$

This means that the applying the linear transform AB over a vector is the same than first applying the linear transform A and then the linear transform B. The code below accumulate two linear transforms computing AB and storing the result in A. Notice that our code always assumes that all matrices have $(0, 0, 1)^T$ as the last column:

Section: Local Macros (metafont.c) (continuation):

```
#define MATRIX_MULTIPLICATION(A, B) {\
  float _a0 = A[0], _a1 = A[1], _a3 = A[3], _a4 = A[4], _a6 = A[6],\
        _a7 = A[7];\
  A[0] = _a0 * B[0] + _a1 * B[3];\
  A[1] = _a0 * B[1] + _a1 * B[4];\
  A[3] = _a3 * B[0] + _a4 * B[3];\
  A[4] = _a3 * B[1] + _a4 * B[4];\
  A[6] = _a6 * B[0] + _a7 * B[3] + B[6];\
  A[7] = _a6 * B[1] + _a7 * B[4] + B[7];\
}
```

Some linear transforms are very common. For them, we will create more macros to perform them more easily. For example, rotating a vector $(x, y, 1)$ around the axis $(0, 0, 1)$ by the angle θ. This can be done multiplying by the matrix:

$$[x\ y\ 1] \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -\sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x \cdot cos(\theta) - y \cdot sin(\theta)\ \ x \cdot sin(\theta) + y \cdot cos(\theta)\ \ 1]$$

Applying the linear transform above over a matrix can be done using the macro below:

Section: Local Macros (metafont.c) (continuation):

```
#define TRANSFORM_ROTATE(M, theta) {\
  float _m0 = M[0], _m1 = M[1], _m3 = M[3], _m4 = M[4], _m6 = M[6],\
        _m7 = M[7];\
  double _cos_theta, _sin_theta;\
  _sin_theta = sin(theta);\
  _cos_theta = cos(theta);\
  M[0] = _m0 * _cos_theta - _m1 * _sin_theta;\
  M[1] = _m0 * _sin_theta + _m1 * _cos_theta;\
  M[3] = _m3 * _cos_theta - _m4 * _sin_theta;\
  M[4] = _m3 * _sin_theta + _m4 * _cos_theta;\
  M[6] = _m6 * _cos_theta - _m7 * _sin_theta;\
  M[7] = _m6 * _sin_theta + _m7 * _cos_theta;\
}
```

Another relevant linear transform is changing the vector sizes, compressing or stretching them. To perform this in the axis x, we multiply by the following matrix:

$$[x\ y\ 1] \begin{bmatrix} s & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [sx\ \ y\ \ 1]$$

And this is done by the macro below:

Section: Local Macros (metafont.c) (continuation):

```
#define TRANSFORM_SCALE_X(M, s) {\
  M[0] = M[0] * s;\
  M[3] = M[3] * s;\
  M[6] = M[6] * s;\
}
```

Perform this in the axis y means multiplying:

$$[x\ y\ 1]\begin{bmatrix}1 & 0 & 0\\ 0 & s & 0\\ 0 & 0 & 1\end{bmatrix} = [x\ \ sy\ \ 1]$$

And we use the following code:

Section: Local Macros (metafont.c) (continuation):

```
#define TRANSFORM_SCALE_Y(M, s) {\
  M[1] = M[1] * s;\
  M[4] = M[4] * s;\
  M[7] = M[7] * s;\
}
```

And doing this in both axis means just applying both transforms in any order:

Section: Local Macros (metafont.c) (continuation):

```
#define TRANSFORM_SCALE(M, s) {\
  TRANSFORM_SCALE_X(M, s);\
  TRANSFORM_SCALE_Y(M, s);\
}
```

The translation, or shifting, is the only operation here that requires three dimensions to work as a linear transform. Shifting a vector a in the horizontal and b in the vertical is done by the matrix multiplication:

$$[x\ y\ 1]\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & b & 1\end{bmatrix} = [x+a\ \ y+b\ \ 1]$$

This matrix multiplication is done using the following macro:

Section: Local Macros (metafont.c) (continuation):

```
#define TRANSFORM_SHIFT(M, a, b) {\
  M[6] = M[6] + a;\
  M[7] = M[7] + b;\
}
```

Slant using s as the intensity is performed by the multiplication:

$$[x\ y\ 1]\begin{bmatrix}1 & 0 & 0\\ s & 1 & 0\\ a & b & 1\end{bmatrix} = [x+sy\ \ y\ \ 1]$$

Which we apply over other matrices using the macro:

Section: Local Macros (metafont.c) (continuation):

```
#define TRANSFORM_SLANT(M, s) {\
  M[0] = M[0] + s * M[1];\
  M[3] = M[3] + s * M[4];\
  M[6] = M[6] + s * M[7];\
}
```

The last special transform described here is changing the scale in the complex plane. This means multiplying the points by a pair (s, t), treating all them as complex numbers. This performs a scale change and a rotation at the same time. And it is done multiplying by the matrix:

$$[x\ y\ 1] \begin{bmatrix} s & t & 0 \\ -t & s & 0 \\ 0 & 0 & 1 \end{bmatrix} = [sx - ty \ \ tx + sy \ \ 1]$$

Performing this operation in another matrix can be done using:

Section: Local Macros (metafont.c) (continuation):

```
#define TRANSFORM_SCALE_Z(M, s, t) {\
  float _m0 = M[0], _m1 = M[1], _m3 = M[3], _m4 = M[4], _m6 = M[6],\
        _m7 = M[7];\
  M[0] = _m0 * s - _m1 * t;\
  M[1] = _m0 * t + _m1 * s;\
  M[3] = _m3 * s - _m4 * t;\
  M[4] = _m3 * t + _m4 * s;\
  M[6] = _m6 * s - _m7 * t;\
  M[7] = _m6 * t + _m7 * s;\
}
```

2.1.1. Solving Linear System Equations

Solving systems of linear equations is the central problem in linear algebra. These are equations in which the unknowns are always multiplied by numerical constants and added together (we never will have two unknowns multiplied together). For example:

$$x + 2y + 3z = 6$$

$$2x + 5y + 2z = 4$$

$$6x - 3y + z = 2$$

We can represent the above system of equations in matrix form, associating the first column with x, the second with y and the third with z:

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 2 \\ 6 & -3 & 1 \end{bmatrix} x = \begin{bmatrix} 6 \\ 4 \\ 2 \end{bmatrix}$$

Solving the system means finding a valid value for x. In the above example: $x = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$

A function capable of solving this system could be the one below, which receives a number of dimensions n, a vector already allocated with the n^2 elements of a matrix, an already allocated vector with n elements on the side right side of the equation, and a next vector with n values of the solution, which will be filled by the function:

Section: Local Function Declaration (metafont.c):

```
void solve_linear_system(int n, double *m, double *b, double *x);
```

Before showing how to solve the system, it is useful to define some new macros to help us. Let's assume that every matrix, regardless of size is allocated in a single continuous vector that concatenates each of its lines. It will be very common for us to have to change the order of the lines a matrix to solve the system of equations. Swap lines in matrix M, which represent the left side of a system of equations also means swapping the same rows of b, the matrix representing the right side.

The macro below shows what happens when, in a system with n lines, we swap in the matrix M and in b the lines i and j:

Section: Local Macros (metafont.c) (continuation):

```
#define EXCHANGE_ROWS(n, M, b, i, j) {\
  if(i != j){\
   int _k;\
   double _tmp;\
   _tmp = b[i];\
   b[i] = b[j];\
   b[j] = _tmp;\
   for(_k = 0; _k < n; _k++){\
       _tmp = m[i*n+_k];\
       m[i*n+_k] = m[j*n+_k];\
       m[j*n+_k] = _tmp;}}}
```

Note that changing the lines does not change the result of a system. equations. For example, the three equations below have the same solution ($x = 0$, $y = 0$, $z = 2$) of the system of equations at the beginning of this Subsection:

$$6x - 3y + z = 2$$

$$x + 2y + 3z = 6$$

$$2x + 5y + 2z = 4$$

Another relevant operation is, given a system with n lines, with the matrix M and the matrix b, we must combine the lines i and j, subtracting from line i the contents of line j multiplied by q:

Section: Local Macros (metafont.c) (continuation):

```
#define SUB_MUL_LINES(n, m, b, i, j, q) {\
  int _k;\
  b[i] -= (q * b[j]);\
  for(_k = 0; _k < n; _k++){\
      m[i*n+_k] -= (q * m[j*n+_k]);}}
```

Performing this operation also does not change the result of the equations. For example, in our previous example, we can subtract the second line by the first multiplied by 1/6 and the third by first multiplied by 1/3. The result would be:

$$6x - 3y + z = 2$$

$$5/2y + 17/6z = 17/3$$

$$6y + 5/3z = 10/3$$

Although the result did not change, we were able to simplify the system, leaving the last two equations with only two unknowns. With one more operation, it would be possible to leave the last equation with just one unknown and from then on the solution becomes trivial.

What we want then is to perform the operations of the two previous macros successive times until obtaining a triangular matrix with all elements below the main diagonal null. Then computing the solution becomes straightforward: in the last line where we have just one unknown, we compute directly the result. Then, knowing its value, there remains only one unknown in the penultimate line. And so we continue until we reach first line and no more unknown unknowns remain.

To build the triangular matrix, we have an iteration where we go through one line at a time. In iteration i, our goal is make all elements in column i that are below the line i null. To do this, we choose the current line or below it that has the largest absolute value in column i and swap its position with the current line. This is not strictly necessary, but choosing the largest possible value minimizes operation errors. And then from that we choose suitable values to transform the matrix and solve it:

Section: Auxiliary Local Functions (metafont.c):

```
void solve_linear_system(int n, double *m, double *b, double *x){
```

```
  int i, j;
  for(i = 0; i < n; i ++){ // Para cada linha
    // Acha maior piv possvel na coluna atual e troca de posio com ele:
    int max_line = i;
    double max = fabs(m[i * n + i]);
    for(j = i + 1; j < n; j ++){
      if(fabs(m[j*n+i]) > max){
        max = fabs(m[j*n+i]);
        max_line = j;
      }
    }
    EXCHANGE_ROWS(n, m, b, i, max_line);
    // Torna zero as colunas do piv nas linhas abaixo:
    for(j = i + 1; j < n; j ++){
      double multiplier = m[j * n + i]/m[i * n + i];
      SUB_MUL_LINES(n, m, b, j, i, multiplier);
    }
  }
  // Gerando a soluo  partir da matriz triangular:
  for(i = n - 1; i >= 0; i --){
    x[i] = 0;
    for(j = n - 1; j > i; j --)
      x[i] += x[j] * m[i*n + j];
    x[i] = (b[i] - x[i]) / m[i*n + i];
  }
}
```

2.2. Mutexes and Critical Sections

A mutex is a data sctructure that controls the access for multiple processes to a single common resource. They are treated differently depending on the operating system and environment. Because of its non-portability, we will introduce them here, we need to use slightly different functions and declarations, depending on where the code is compiled.

On Linux and BSD a Mutex is defined using the library `pthread` and follows its naming convention. On Web Assembly, Emscripten implements the same API, but it will work only if the web page uses the correct HTTP headers. Because of this, we will use mutexes on Web Assembly only if the user asks defining macro `W_ALWAYS_USE_THREADS`. On Windows there is a difference between mutexes, which can be shared between multiple programs and critical sections, which can be shared only between threds from a single program. For our use case, we need a critical section, not a mutex.

Section: Mutex Declaration:

```
#if defined(_WIN32)
CRITICAL_SECTION mutex;
#elif defined(__linux__) || defined(BSD) || defined(W_ALWAYS_USE_THREADS)
pthread_mutex_t mutex;
#endif
```

This means that on Linux and BSD we need to insert the header `pthread`. On windows, the default header that we already placed on `metafont.h` is enough.

Section: Include General Headers (metafont.h):

```
#if defined(__linux__) || defined(BSD) || defined(W_ALWAYS_USE_THREADS)
#include <pthread.h>
#endif
```

To initialize a mutex (or critical section), we use these functions:

Section: Local Macros (metafont.c):

```
#if defined(_WIN32)
#define MUTEX_INIT(mutex) InitializeCriticalSection(mutex);
#elif defined(__linux__) || defined(BSD) || defined(W_ALWAYS_USE_THREADS)
#define MUTEX_INIT(mutex) pthread_mutex_init(&mutex, NULL);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_INIT(mutex)
#endif
```

When we do not need a mutex (or critical section) anymore, we need to use some function to close it:

Section: Local Macros (metafont.c) (continuation):

```
#if defined(_WIN32)
#define MUTEX_DESTROY(mutex) DeleteCriticalSection(mutex);
#elif defined(__linux__) || defined(BSD) || defined(W_ALWAYS_USE_THREADS)
#define MUTEX_DESTROY(mutex) pthread_mutex_destroy(&mutex);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_DESTROY(mutex)
#endif
```

When we have a mutex, there are two main operations defined over them. The first is asking to use the mutex. If another process is already using it, then this call would block and we wait until the mutex is free again:

Section: Local Macros (metafont.c) (continuation):

```
#if defined(_WIN32)
#define MUTEX_WAIT(mutex) EnterCriticalSection(mutex);
#elif defined(__linux__) || defined(BSD) || defined(W_ALWAYS_USE_THREADS)
#define MUTEX_WAIT(mutex) pthread_mutex_lock(&mutex);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_WAIT(mutex)
#endif
```

And finally, after we used the mutex, we can free it again, letting other processes use it:

Section: Local Macros (metafont.c) (continuation):

```
#if defined(_WIN32)
#define MUTEX_SIGNAL(mutex) LeaveCriticalSection(mutex);
#elif defined(__linux__) || defined(BSD) || defined(W_ALWAYS_USE_THREADS)
#define MUTEX_SIGNAL(mutex) pthread_mutex_unlock(&mutex);
#elif defined(__EMSCRIPTEN__)
#define MUTEX_SIGNAL(mutex)
#endif
```

2.3. Dealing with Errors

Most of the functions defined here receive as input a metafont struct representing the font being rendered and a context struct with information about the execution. Furthermore, they usually return true if the execution is successful and false otherwise. Some functions may follow the different convention of returning a pointer if they are successful and returning NULL otherwise.

If an error is encountered, we must immediately stop running the code and return false. If an invoked function returns false indicating an error, the function that called it must also stop execution. In addition, we must mark within the metafont structure that an error has occurred, filling in additional information that will allow a diagnostic message to be printed later if the user so desires.

The act of filling in the error information in the metafont struct will be done with the help of macros. Some examples of macros can be `RAISE_NO_MEMORY` or `RAISE_ERROR_FAILED_OPENING_FILE`. Almost all error macros will receive as input the metafont structure (to fill in data about the error), either a context struct or NULL (in case they need to read additional information about the execution) and optionally the line number where the error was found. Depending on the error, more information can be placed in the macro.

To avoid having to define many different types of errors in detail in the body of the text, thus taking the focus away from the algorithms and logic being implemented, we will define the errors and their macros only in Appendix A.

3. Initialization and Finalization

First we will define the initialization function. What it will do is set some static variables that will be used by other functions. It also will set in a variable our screen pixel density in DPI. The variables that will store these information are:

Section: Local Variables (metafont.c):

```
static void *(*temporary_alloc)(size_t);
static void (*temporary_free)(void *);
static void *(*permanent_alloc)(size_t);
static void (*permanent_free)(void *);
static uint64_t (*random_func)(void);
static int dpi;
```

And the initialization function that sets these variables:

Section: API Functions Definition (metafont.c):

```
bool _Winit_weavefont(void *(*t_alloc)(size_t),
                      void (*t_free)(void *),
                      void *(*p_alloc)(size_t),
                      void (*p_free)(void *),
                      uint64_t (*random)(void), int pixel_density){
  temporary_alloc = t_alloc;
  temporary_free = t_free;
  permanent_alloc = p_alloc;
  permanent_free = p_free;
  random_func = random;
  dpi = pixel_density;
                    <Section to be inserted: WeaveFont Initialization>
  return true;
}
```

There is also a finalization function and when we update the initialization function, we may need to update with correnponding code the finalization function:

Section: API Functions Definition (metafont.c):

```
void _Wfinish_weavefont(void){
                    <Section to be inserted: WeaveFont Finalization>
}
```

Global variables and informations about a meta-font that need to be kept on the long term, always will be allocated with the permanent allocation function. Local variables inside the rendering instructions for each character always will be allocated with the temporary allocation function. If the memory manages distinguishes between these two allocations (like in Weaver Game Engine), these will be two different functions. In most other cases, the same function would be passed in both arguments, probably a `malloc`. In this case, both deallocation functions also would be the same `free` function. In Weaver Game Engine, we hgave no deallocation functions: the "permanent" allocation is freed in the end of each game loop and the

temporary allocation is freed at the end of each frame.

As we expect that these functions will be called only once per program, we do not need to use a mutex for them.

4. Lexer

4.1. Token Types

The first thing to be created for a language is its lexer. It will read the source code in a file and will output a list of tokens, where tokens are the most basic unit in the language. A token is like a word: the most basic element in a language.

WeaveFont recognizes the following types of tokens: numeric, strings, symbolic variable, symbolic loop begin, symbolic loop end and symbolic generic. The symbolic generic on practice is subdivided in several other sub types, one for each reserved word in the language. Here is where we define the main types, and where we will place all other subtypes of symbolic generic tokens:

Section: Local Data Structures (metafont.c):

```
enum { // Token types
  TYPE_NUMERIC = 1, TYPE_STRING, TYPE_SYMBOLIC, TYPE_FOR, TYPE_ENDFOR,
  // The basic types (numeric, string, variable, loop begin and end)
  //  are above. Others will be placed below:
          <Section to be inserted: WeaveFont: Symbolic Token Definition>
  // And the last one that shouldn't be used, except to represent errors:
  TYPE_INVALID_TOKEN
};
```

A numeric token will be represented internally as a floating point number. This is different than specified in the original METAFONT, where a custom fixed-point numeric representation was used. This choice will make our operations over numbers faster, as we count with hardware support for this. This is how a numeric token is represented:

Section: Local Data Structures (metafont.c) (continuation):

```
struct numeric_token{
  int type;   // Should be equal 'TYPE_NUMERIC'
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  float value;
};
```

And this is how we will represent string tokens:

Section: Local Data Structures (metafont.c) (continuation):

```
struct string_token{
  int type;   // Should be equal 'TYPE_STRING'
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  char value[5];
  // Pointer for the glyph represented by string (will be used in Section 15.1)
  struct _glyph *glyph;
};
```

We will store only the first 4 bytes of each given string, even if in the source code the string is bigger.

This is so because unlike in the original METAFONT, our only use for strings are saying wich Unicode character each glyph should represent. For this we need only 4 bytes.

In the case of symbolic variable tokens, we need to store a pointer to the variable and its name:

Section: Local Data Structures (metafont.c) (continuation):

```
struct symbolic_token{
  int type;   // Should be equal 'TYPE_SYMBOLIC'
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  void *var;
  char *value;
};
```

A token that begins a loop needs a boolean variable to store if the loop i currently running or not. It also needs a pointer to a numeric variable for loop control, a floating point number as increment (that will be added to the control variable each iteration) and the stop condition, floating point number that, if crossed by the control variable, will break the loop. We will also store a pointer to the end of the loop, so that we can easily exit the loop if needed:

Section: Local Data Structures (metafont.c) (continuation):

```
struct begin_loop_token{
  int type;   // Must be equal 'TYPE_FOR'
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  bool running;
  float *control_var;
  struct linked_token *end;
};
```

Some tokens need to store a pointer for other tokens. An example is the token `endfor` that marks the end of a loop, but must store a pointer to the beginning of the loop, so that we can iterate over the code more easily. Tokens like `if`, `elseif`, `else` and `beginchar` also need to store pointers to other parts of the code, so that it is easier to navigate in the code while interpreting it. These tokens will use the following data structure:

Section: Local Data Structures (metafont.c) (continuation):

```
struct linked_token{
  //'TYPE_ENDFOR'/'TYPE_IF'/'TYPE_ELSEIF'/'TYPE_ELSE'/'TYPE_BEGINCHAR':
  int type;
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  struct generic_token *link;
};
```

For the symbolic generic tokens, all we need to store about them is its sub type. Therefore, we use the following structure:

Section: Local Data Structures (metafont.c) (continuation):

```
struct generic_token{
```

```
  int type;   // A subtype of symbolic token
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
};
```

We just need to define a subtype for each of them:

Section: WeaveFont: Symbolic Token Definition:

```
TYPE_OPEN_PARENTHESIS,  // '('
TYPE_CLOSE_PARENTHESIS, // ')'
TYPE_COMMA,             // ','
TYPE_SEMICOLON,         // ';'
```

We can define later more reserved symbolic tokens. Notice that by C `enum` rules, any token whose type is a number greater or equal than 3 is a symbolic token. And if greater than 5, it is a symbolic generic token.

Every token have a pointer to a next token. This happens because usually they will be part of a linked list.

Tokens are created by the permanent allocation function. We need to keep them on memory because we may need to read and interpret them many times, each one to render a new character. Because of this, we avoid to store in them information that is not needed. For example, their line numbers in the source code is useful for debugging. Font designers and image creators would use this information, but most casual users never would debug the WeaveFont code that they run, Because of this, we store this information only if we are in debugging mode.

To deallocate the memory occupied by a token list, we can use the following function:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void free_token_list(void *token_list){
  if(permanent_free != NULL && token_list != NULL){
    struct generic_token *p, *p_next;
    p = token_list;
    while(p != NULL){
      p_next = p -> next;
      if(p -> type == TYPE_SYMBOLIC)
        permanent_free(((struct symbolic_token *) p) -> value);
      permanent_free(p);
      p = p_next;
    }
  }
}
```

4.2. Lexical Analysis

Now the funtion that represents our lexer. It gets as argument the struct for our meta-font (whose contents we still will define) and a string with a path for the file with METAFONT source code. It will produce a token list and will store in its two last arguments the pointer for the first and last pointer. The function will return true if no error happened and false otherwise.

Section: Auxiliary Local Functions (metafont.c):

```
bool lexer(struct metafont *mf, char *path, struct generic_token **first_token,
           struct generic_token **last_token){
  struct linked_token *aux_stack = NULL;
  FILE *fp;
  char c;
```

```
  int line = 1;
  *first_token = NULL;
  *last_token = NULL;
  fp = fopen(path, "r");
  if(fp == NULL){
    RAISE_ERROR_FAILED_OPENING_FILE(mf, NULL, 0, path);
    return false;
  }
  while((c = fgetc(fp)) != EOF){
    char next_char = fgetc(fp);
    ungetc(next_char, fp);
    if(c == '\n'){
      line ++;
      continue;
    }
```

<Section to be inserted: Lexer: Rule 1>
<Section to be inserted: Lexer: Rule 2>
<Section to be inserted: Lexer: Rule 3>
<Section to be inserted: Lexer: Rule 4>
<Section to be inserted: Lexer: Rule 5>
<Section to be inserted: Lexer: Rule 6>

```
    {
      // No rule applied: error
      char unknown_char[5];
      memset(unknown_char, 0, 5);
      unknown_char[0] = c;
      fgets(&(unknown_char[1]), 4, fp);
      RAISE_ERROR_INVALID_CHAR(mf, NULL, line, unknown_char);
      return false;
    }
    free_token_list(first_token);
    *first_token = NULL;
    *last_token = NULL;
    return false;
  }
```

<Section to be inserted: Lexer: Detect Final Errors>

```
  fclose(fp);
  return true;
}
```

As we used the struct FILE, this means that we need to include the header with data about input/output to be able to read files:

Section: Local Headers (metafont.c):

```
#include <stdio.h>
```

The rules to read tokens consist in reading each line in the source code applying the following rules for each character in the line:

1) If the next character is a space, tabs or a period and is not followed by a period or a decimal digit, then ignore this character and go to the next one.

Section: Lexer: Rule 1:

```
if(c == ' ' || c == '\t' ||
   (c == '.' && next_char != '.' && !isdigit(next_char)))
```

```
  continue;
```

As we are using `isdigit` function, we need to insert the following header that declares this function:

Section: Local Headers (metafont.c):

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

2) If the character is a percent sign, ignore it and also ignore all other following characters in this line. Percentage sign is how we start comments in the language.

Section: Lexer: Rule 2:

```
if(c == '%'){
  do{
    c = fgetc(fp);
  } while(c != '\n' && c != EOF);
  ungetc(c, fp);
  continue;
}
```

3) If the next character is a decimal digit or a period, then the next token in numeric. It will be interpreted from the biggest sequence of decimal digests and a single optional period representing the decimal dot present in the input.

Section: Lexer: Rule 3:

```
if((c == '.' && isdigit(next_char)) || isdigit(c)){
  char buffer[256];
  struct numeric_token *new_token =
    (struct numeric_token *) permanent_alloc(sizeof(struct numeric_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0); // We will define errors on Appendix A.
    return false;
  }
  new_token -> type = TYPE_NUMERIC;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  int i = 0;
  int number_of_dots = (c == '.');
  buffer[i] = c;
  i ++;
  do{
    c = fgetc(fp);
    if(c == '.')
      number_of_dots ++;
    buffer[i] = c;
    i ++;
  } while(isdigit(c) || (c == '.' && number_of_dots == 1));
  ungetc(c, fp);
  i --;
  buffer[i] = '\0';
  new_token -> value = atof(buffer);
  if(*first_token == NULL)
```

```
    *first_token = *last_token = (struct generic_token *) new_token;
  else{
    (*last_token) -> next = (struct generic_token *) new_token;
    *last_token = (struct generic_token *) new_token;
  }
  continue;
}
```

4) If the next character is a double quote, the next token will be a string. Its content will be all other characters until the next double quote not preceded by a back slash that should be in the same line. If we have a double quote opening a string, but the same line do not have another double quote to close the string, this is an error. When reading a back slash, we always ignore the first one, but we always consider the next character. This rule allow us to represent a double quote inside a string (in the form `"\""`) and a backslash inside the string (in the form `"\\"`).

Section: Lexer: Rule 4:

```
if(c == 34){ // 34: ASCII  for double quote
  struct string_token *new_token =
    (struct string_token *) permanent_alloc(sizeof(struct string_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  new_token -> type = TYPE_STRING;
  new_token -> glyph = NULL;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  int i = 0, prev = 0, prev_prev;
  do{
    prev_prev = prev;
    prev = c;
    c = fgetc(fp);
    if(i < 5 && (c !='\\' || prev == '\\')){
      new_token -> value[i] = c;
      i ++;
    }
  } while((c != 34 || (prev == '\\' && prev_prev != '\\')) &&
          c != '\n' && c != EOF);
  i --;
  new_token -> value[i] = '\0';
  if(c == '\n' || c == EOF){
    RAISE_ERROR_UNCLOSED_STRING(mf, NULL, line, new_token -> value);
    if(permanent_free != NULL)
      permanent_free(new_token);
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    return false;
  }
  if(*first_token == NULL)
```

```
    *first_token = *last_token = (struct generic_token *) new_token;
  else{
    (*last_token) -> next = (struct generic_token *) new_token;
    *last_token = (struct generic_token *) new_token;
  }
  continue;
}
```

5) If the next character is a parenthesis, semicolon or a comma, the next token will be symbolic and composed by that single character.

Section: Lexer: Rule 5:

```
if(c == '(' || c == ')' || c == ',' || c == ';'){
  struct generic_token *new_token =
     (struct generic_token *) permanent_alloc(sizeof(struct generic_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL,  line);
    return false;
  }
  if(c == '(')
    new_token -> type = TYPE_OPEN_PARENTHESIS;
  else if(c == ')')
    new_token -> type = TYPE_CLOSE_PARENTHESIS;
  else if(c == ';')
    new_token -> type = TYPE_SEMICOLON;
  else
    new_token -> type = TYPE_COMMA;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  if(*first_token == NULL)
    *first_token = *last_token = (struct generic_token *) new_token;
  else{
    (*last_token) -> next = (struct generic_token *) new_token;
    *last_token = (struct generic_token *) new_token;
  }
  continue;
}
```

6) Otherwise, the next token will be symbolic and will be composed by the longest sequence composed only by characters taken from one of 12 different families:

Section: Lexer: Rule 6:

```
{
  char buffer[256];
  int i = 0;
  buffer[0] = '\0';
  // Buffer is read according with subrules 6-a to 6-l
```

<Section to be inserted: Lexer: Rule A>
<Section to be inserted: Lexer: Rule B>
<Section to be inserted: Lexer: Rule C>

```
                    <Section to be inserted: Lexer: Rule D>
                    <Section to be inserted: Lexer: Rule E>
                    <Section to be inserted: Lexer: Rule F>
                    <Section to be inserted: Lexer: Rule G>
                    <Section to be inserted: Lexer: Rule H>
                    <Section to be inserted: Lexer: Rule I>
                    <Section to be inserted: Lexer: Rule J>
                    <Section to be inserted: Lexer: Rule K>
                    <Section to be inserted: Lexer: Rule L>
  // Depending on buffer content, generates next token
                <Section to be inserted: Lexer: New Control Flow Token>
              <Section to be inserted: Lexer: New Reserved Symbolic Token>
               <Section to be inserted: Lexer: New Generic Symbolic Token>
}
```

a) The first family of letters are the uppercase and lowercase alphabetic letters, digits and underline. A digit cannot be the first character in the sequence, otherwise it would be considered part of a numeric token.

Section: Lexer: Rule A:

```
if(isalpha(c) || c == '_'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(isalpha(c) || c == '_' || isdigit(c));
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

b) The second family is composed by the symbols for greater, equal and lesser, colon and a pipe.

Section: Lexer: Rule B:

```
else if(c == '>' || c == '<' || c == '=' || c == ':' || c == '|'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '>' || c == '<' || c == '=' || c == ':' || c == '|');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

c) Acute and grave accents.

Section: Lexer: Rule C:

```
else if(c == '`' || c == '\''){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '`' || c == '\'');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

d) Plus and minus.

Section: Lexer: Rule D:

```
else if(c == '+' || c == '-'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '+' || c == '-');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

e) Slash, backslash and multiplication symbol.

Section: Lexer: Rule E:

```
else if(c == '\\' || c == '/' || c == '*'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '\\' || c == '/' || c == '*');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

f) Question and exclamation marks.

Section: Lexer: Rule F:

```
else if(c == '?' || c == '!'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '?' || c == '!');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

g) Hash sign, ampersand, at sign and dollar sign.

Section: Lexer: Rule G:

```
else if(c == '#' || c == '&' || c == '@' || c == '$'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '#' || c == '&' || c == '@' || c == '$');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

h) Circumflex accent and tilde.

Section: Lexer: Rule H:

```
else if(c == '^' || c == '~'){
  do{
    buffer[i] = c;
    i ++;
```

```
    c = fgetc(fp);
  } while(c == '^' || c == '~');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

i) Opening brackets.

Section: Lexer: Rule I:

```
else if(c == '['){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '[');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

j) Closing brackets.

Section: Lexer: Rule J:

```
else if(c == ']'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == ']');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

k) Opening and closing braces.

Section: Lexer: Rule K:

```
else if(c == '{' || c == '}'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '{' || c == '}');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

l) Ponto.

Section: Lexer: Rule L:

```
else if(c == '.'){
  do{
    buffer[i] = c;
    i ++;
    c = fgetc(fp);
  } while(c == '.');
  ungetc(c, fp);
  buffer[i] = '\0';
}
```

After reading the characters for our new symbolic token, we check if we have a reserved symbolic token: a token representing a language keyword. We can check this using function `strcmp` comparing the buffer with some keywords. We will store a list of keywords here:

Section: Local Variables (metafont.c) (continuation):

```
static char* list_of_keywords[] = {
                    <Section to be inserted: List of Keywords>
  NULL};
```

Therefore, to know if we are dealing with a token that represents a reserved keyword, we check if it is in this NULL terminated list. If so, we create a token whose type is chosen according with the position in this list:

Section: Lexer: New Reserved Symbolic Token:

```
{
  int token_type = 0;
  for(i = 0; list_of_keywords[i] != NULL; i ++)
    if(!strcmp(buffer, list_of_keywords[i]))
      token_type = i + TYPE_SEMICOLON + 1; // Keywords come after ';'
  if(token_type != 0){
    struct generic_token *new_token =
       (struct generic_token *) permanent_alloc(sizeof(struct generic_token));
    if(new_token == NULL){
      free_token_list(*first_token);
      *first_token = *last_token = NULL;
      RAISE_ERROR_NO_MEMORY(mf, NULL, line);
      return false;
    }
    new_token -> type = token_type;
    new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
    new_token -> line = line;
#endif
    if(*first_token == NULL)
      *first_token = *last_token = (struct generic_token *) new_token;
    else{
      (*last_token) -> next = (struct generic_token *) new_token;
      *last_token = (struct generic_token *) new_token;
    }
    continue;
  }
}
```

To use function `strcmp` we need the following header:

Section: Local Headers (metafont.c):

```
#include <string.h>
```

Control flow tokens like `for`, `endfor`, `if`, `elseif`, `else` and `fi` also are tokens whose names are reserved words. But we give a higher precedence dealing with them and use a different code to initialize them. This must happen because several of these tokens should have initialized pointers for other tokens, so that it becomes easier to navigate in the code while they change the usual flow of the code.

In case of `for`, when we create it, we also create the corresponding `endfor` that later will close the loop, and we initialize both tokens. We make `for` having a pointer to `endfor` and `endfor` having a pointer to the token before `for`. This will make easier for the interpreter to navigate in the iteration. The token

`endfor` is then stored in a stack to be used later.

Section: Lexer: New Control Flow Token:

```
if(!strcmp(buffer, "for")){
  struct generic_token *previous_token = *last_token;
  struct linked_token *endfor_token;
  struct begin_loop_token *new_token = (struct begin_loop_token *)
                              permanent_alloc(sizeof(struct begin_loop_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  new_token -> type = TYPE_FOR;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  new_token -> running = false;
  new_token -> control_var = NULL;
  new_token -> end = NULL;
  if(*first_token == NULL)
    *first_token = *last_token = (struct generic_token *) new_token;
  else{
    (*last_token) -> next = (struct generic_token *) new_token;
    *last_token = (struct generic_token *) new_token;
  }
  endfor_token = (struct linked_token *)
                                  permanent_alloc(sizeof(struct linked_token));
  if(endfor_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  endfor_token -> link = (struct generic_token *) previous_token;
  endfor_token -> type = TYPE_ENDFOR;
  new_token -> end = endfor_token;
  if(aux_stack == NULL){
    aux_stack = endfor_token;
    endfor_token -> next = NULL;
  }
  else{
    endfor_token -> next = (struct generic_token *) aux_stack;
    aux_stack = endfor_token;
  }
  continue;
}
```

When finally we find an “endfor” in the code, we finally use the stacked token, instead of creating a new one. If such token do not exist, we raise an error (which we define later):

Section: Lexer: New Control Flow Token (continuation):

```
if(!strcmp(buffer, "endfor")){
  if(aux_stack == NULL || aux_stack -> type != TYPE_ENDFOR){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_UNBALANCED_ENDING_TOKEN(mf, NULL, OPTIONAL(line), TYPE_ENDFOR);
    return false;
  }
  struct linked_token *new_token = aux_stack;
  aux_stack = (struct linked_token *) aux_stack -> next;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  (*last_token) -> next = (struct generic_token *) new_token;
  *last_token = (struct generic_token *) new_token;
  continue;
}
```

If we read a token `if`, likewise, we alsocreate a token `fi` and put it in the stack. But this will be just a temporary `fi`, not the one that we will place in the final program, because this special `fi` will be a "linked token" with a pointer to the previous `if`, `elseif` or `else` in the same nesting level. This pointer will help us to update such past tokens so that they will correctly point to the next conditional control flow token.

This is how we create a token `if`:

Section: Lexer: New Control Flow Token (continuation):

```
if(!strcmp(buffer, "if")){
  struct linked_token *if_token, *fi_token;
  if_token = (struct linked_token *)
              permanent_alloc(sizeof(struct linked_token));
  if(if_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  if_token -> type = TYPE_IF;
  if_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  if_token -> line = line;
#endif
  if_token -> link = NULL;
  if(*first_token == NULL)
    *first_token = *last_token = (struct generic_token *) if_token;
  else{
    (*last_token) -> next = (struct generic_token *) if_token;
    *last_token = (struct generic_token *) if_token;
  }
  fi_token = (struct linked_token *)
              temporary_alloc(sizeof(struct linked_token));
  if(fi_token == NULL){
    free_token_list(*first_token);
```

```
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  fi_token -> link = (struct generic_token *) if_token;
  fi_token -> type = TYPE_FI;
  if(aux_stack == NULL){
    aux_stack = fi_token;
    fi_token -> next = NULL;
  }
  else{
    fi_token -> next = (struct generic_token *) aux_stack;
    aux_stack = fi_token;
  }
  continue;
}
```

If we find a token `elseif` or `else` in the same nesting level, we check the pointer of the stacked `if` and make the pointed token point to our new `elseif` or `else` found. This will correctly initialize this previous token that should be a `if` or a previous `elseif`. Then, we create a new token to our newly found `elseif` or `else`, insert it in our program and update the stacked `fi` token to point to this newly created token:

Section: Lexer: New Control Flow Token (continuation):

```
if(!strcmp(buffer, "elseif") || !strcmp(buffer, "else")){
  struct linked_token *new_token;
  if(aux_stack == NULL || aux_stack -> type != TYPE_FI ||
     aux_stack -> link -> type == TYPE_ELSE){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_UNBALANCED_ENDING_TOKEN(mf, NULL, OPTIONAL(line),
                                        (buffer[4] == 'i')?TYPE_ELSEIF:TYPE_ELSE);
    return false;
  }
  new_token = (struct linked_token *)
               permanent_alloc(sizeof(struct linked_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  ((struct linked_token *) (aux_stack -> link)) -> link =
                                          (struct generic_token *) new_token;
  if(buffer[4] == 'i')
    new_token -> type = TYPE_ELSEIF;
  else
    new_token -> type = TYPE_ELSE;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
```

```
  new_token -> link = NULL;
  if(*first_token == NULL)
    *first_token = *last_token = (struct generic_token *) new_token;
  else{
    (*last_token) -> next = (struct generic_token *) new_token;
    *last_token = (struct generic_token *) new_token;
  }
  aux_stack -> link = (struct generic_token *) new_token;
  continue;
}
```

Finally, when finding a `fi`, we remove from the stack the temporary token `fi` with a link to the previous `if`, `elseif` or `else`. We update this previous token's pointer to point to immediately before our `fi`, finishing its initialization:

Section: Lexer: New Control Flow Token (continuation):

```
if(!strcmp(buffer, "fi")){
  struct generic_token *new_token;
  if(aux_stack == NULL || aux_stack -> type != TYPE_FI){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_UNBALANCED_ENDING_TOKEN(mf, NULL, OPTIONAL(line), TYPE_FI);
    return false;
  }
  new_token = (struct generic_token *)
               permanent_alloc(sizeof(struct generic_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  ((struct linked_token *) aux_stack -> link) -> link = new_token;
  new_token -> type = TYPE_FI;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  (*last_token) -> next = (struct generic_token *) new_token;
  *last_token = (struct generic_token *) new_token;
  struct linked_token *tmp = aux_stack;
  aux_stack = (struct linked_token *) aux_stack -> next;
  if(temporary_free != NULL)
    temporary_free(tmp);
  continue;
}
```

Another control flow token is the `beginchar`, which must have its pointer initialized to the next `endchar`. We do this exactly like we did to initialize the `if` token: we stack a temporary `endchar` with a link to the last `beginchar`. When we find the next `enchar` in code, we check this pointer to update the information in the referenced `beginchar`. However, in the case of `beginchar`, we cannot have several os these tokens nested. If we find one of them nested inside other, we raise an error:

Section: Lexer: New Control Flow Token (continuation):

```
if(!strcmp(buffer, "beginchar")){
  struct linked_token *beginchar_token, *endchar_token = aux_stack;
  beginchar_token = (struct linked_token *)
                      permanent_alloc(sizeof(struct linked_token));
  if(beginchar_token ==  NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  while(endchar_token != NULL){
    if(endchar_token -> type == TYPE_ENDCHAR){ // Error if nested 'beginchar'
      free_token_list(*first_token);
      *first_token = *last_token = NULL;
      RAISE_ERROR_NESTED_BEGINCHAR(mf, NULL, line);
      return false;
    }
    endchar_token = (struct linked_token *) (endchar_token -> next);
  }
  beginchar_token -> type = TYPE_BEGINCHAR;
  beginchar_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  beginchar_token -> line = line;
#endif
  beginchar_token -> link = NULL;
  if(*first_token == NULL)
    *first_token = *last_token = (struct generic_token *) beginchar_token;
  else{
    (*last_token) -> next = (struct generic_token *) beginchar_token;
    *last_token = (struct generic_token *) beginchar_token;
  }
  endchar_token = (struct linked_token *)
                    temporary_alloc(sizeof(struct linked_token));
  if(endchar_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  endchar_token -> link = (struct generic_token *) beginchar_token;
  endchar_token -> type = TYPE_ENDCHAR;
  if(aux_stack == NULL){
    aux_stack = endchar_token;
    endchar_token -> next = NULL;
  }
  else{
    endchar_token -> next = (struct generic_token *) aux_stack;
    aux_stack = endchar_token;
  }
```

```
  continue;
}
```

And finally, when we find the real `endchar` in the code, we remove the stacked `endchar` after updating the pointer in the last `beginchar`:

Section: Lexer: New Control Flow Token (continuation):

```
if(!strcmp(buffer, "endchar")){
  struct generic_token *new_token;
  if(aux_stack == NULL || aux_stack -> type != TYPE_ENDCHAR){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_UNBALANCED_ENDING_TOKEN(mf, NULL, OPTIONAL(line), TYPE_ENDCHAR);
    return false;
  }
  new_token = (struct generic_token *)
               permanent_alloc(sizeof(struct generic_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  ((struct linked_token *) aux_stack -> link) -> link = new_token;
  new_token -> type = TYPE_ENDCHAR;
  new_token -> next = NULL;
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  (*last_token) -> next = (struct generic_token *) new_token;
  *last_token = (struct generic_token *) new_token;
  struct linked_token *tmp = aux_stack;
  aux_stack = (struct linked_token *) aux_stack -> next;
  if(temporary_free != NULL)
    temporary_free(tmp);
  continue;
}
```

If in the end of our lexer function we detect that we still have a stacked control flow token, this means that a `for` or `if` was opened, but not closed. Then we should raise an error (which will be defined later):

Section: Lexer: Detect Final Errors:

```
if(aux_stack != NULL){
  free_token_list(*first_token);
  *first_token = *last_token = NULL;
  RAISE_ERROR_MISSING_TOKEN(mf, NULL, OPTIONAL(aux_stack -> link -> line),
                            aux_stack -> type);
  return false;
}
```

If we do not have a reserved symbolic token, and if we have something in our buffer, then we generate a new generic symbolic token. This token will represent a variable:

Section: Lexer: New Generic Symbolic Token:

```
if(buffer[0] != '\0'){
```

```
  buffer[255] = '\0';
  size_t buffer_size = strlen(buffer) + 1;
  struct symbolic_token *new_token =
     (struct symbolic_token *) permanent_alloc(sizeof(struct symbolic_token));
  if(new_token == NULL){
    free_token_list(*first_token);
    *first_token = *last_token = NULL;
    RAISE_ERROR_NO_MEMORY(mf, NULL, line);
    return false;
  }
  new_token -> type = TYPE_SYMBOLIC;
  new_token -> next = NULL;
  new_token -> var = NULL;
  // If this has the name of an internal variable, we can set the pointer
  // for its content right now:
```

<Section to be inserted: Set Pointer to Internal Variable>

```
#if defined(W_DEBUG_METAFONT)
  new_token -> line = line;
#endif
  new_token -> value = (char *) permanent_alloc(buffer_size);
  memcpy(new_token -> value, buffer, buffer_size);
  if(*first_token == NULL)
    *first_token = *last_token = (struct generic_token *) new_token;
  else{
    (*last_token) -> next = (struct generic_token *) new_token;
    *last_token = (struct generic_token *) new_token;
  }
  continue;
}
```

5. WeaveFont Programs

5.1. Running Programs

To evaluate a WeaveFont program, we need two additional structures. The first one, which we will call `struct metafont` will contain all the final information extracted from the typographical meta-font and that will store all needed information to render each glyph in that font. The second structure, which we will call `struct context` represents the current state in our parser and represents information that we need to know to interpret correctly the token list. This second structure can be discarded after we read all the tokens from our font.

In short, for each file with WeaveFont source-code, we will produce a single `struct metafont` with data about the font. And each time we execute code from this file (when we read the file for the first time, and each time we render a new character), we create a new `struct context`, which will be discarded after the code execution.

The first thing that our parser needs to know is that a WeaveFont program is a list (possibly empty) of statements:

```
<Program> -> <List of Statements>
```

In the METAFONT language, it was necessary to indicate the end of statements with a final `end` statement. In WeaveFont language this is not necessary becaus e we will assume that all code is in a single file. Therefore, the end of file is the end of statements.

The first parser function to be defined is the one that recognizes an entire program. It first checks if

the program is empty. If so, then the function returns: doing nothing is the correct execution of an empty program. Otherwise, it passes the entire list of statements for the next function that will split them in individual statements and execute each one individually. After we run the code, we return if the execution was successful.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_program(struct metafont *mf, struct context *cx,
                  struct generic_token *first_token,
                  struct generic_token *last_token){
  if(first_token == NULL)
    return true;
  if(!eval_list_of_statements(mf, cx, first_token, last_token))
    return false;
  // Additional code to be defined in the future:
                  <Section to be inserted: After Program Execution>
  return true;
}
```

The context struct still will not be entirely defined. We will show its content while each attribute becames necessary. For now we will just show a single variable in this structure that will store the current UTF-8 character that is being rendered, or the empty string if no character is being rendered:

Section: Data Structures (metafont.h):

```
struct context{
  char current_character[5];
                  <Section to be inserted: Attributes (struct context)>
};
```

For the meta-font struct, for now we will define three variables stored in it. The first one is a mutex, as described in Subsection 2.2. Next we have a string with the filename from where the font source code was read. The last one is a boolean variable that stores if the font is still loading or if it finished. A font that is still loading did not proccessed all the source code in its file. It is still evaluating the source code statements for the first time. After it evaluates all the code, then it is not anymore in loading mode and is ready to render images.

Section: Data Structures (metafont.h) (continuation):

```
struct metafont{
                      <Section to be inserted: Mutex Declaration>
  char *file;
  bool loading;
                  <Section to be inserted: Attributes (struct metafont)>
};
```

Both data structures will have a function that initializes them and also finalizes them (the function that destroys a meta-font struct, `_Wdestroy_metafont`, already was declared in Section 1, when we listed the headers of functions that would be exported for the user):

Section: Local Function Declaration (metafont.c) (continuation):

```
struct metafont *init_metafont(char *filename);
struct context *init_context(struct metafont *mf);
void destroy_context(struct context *cx);
```

The structure `metafont` needs to store more informations, as it could be allocated both using permanent or temporary memory allocations. Meanwhile, the context always will be temporary and will be allocated with temporary functions.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct metafont *init_metafont(char *filename){
  struct metafont *mf;
  mf = (struct metafont *) permanent_alloc(sizeof(struct metafont));
  if(mf == NULL)
    return NULL;
  MUTEX_INIT(mf -> mutex);
  // Copying the filename where the code is read
  size_t filename_size = strlen(filename) + 1;
  mf -> file = (char *) permanent_alloc(filename_size);
  memcpy(mf -> file, filename, filename_size);
  // We initialize in the loading mode
  mf -> loading = true;
  // More code to be defined in the future:
                <Section to be inserted: Initialization (struct metafont)>
  return mf;
}
struct context *init_context(struct metafont *mf){
  struct context *cx;
  cx = (struct context *) temporary_alloc(sizeof(struct context));
  if(cx == NULL){
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0); // We will define errors on Appendix A.
    return NULL;
  }
  cx -> current_character[0] = '\0';
  // To be defined in the future:
                <Section to be inserted: Initialization (struct context)>
  return cx;
}
```

And this is the definition for the functions that frees the memory occupied by these structures:

Section: API Functions Definition (metafont.c) (continuation):

```
void _Wdestroy_metafont(struct metafont *mf){
  if(permanent_free != NULL){
    permanent_free(mf -> file);
    // To be defined in the future:
                <Section to be inserted: Finalization (struct metafont)>
    MUTEX_DESTROY(mf -> mutex);
    permanent_free(mf);
  }
}
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void destroy_context(struct context *cx){
  if(temporary_free != NULL){
    // TO be defined in the future:
                <Section to be inserted: Finalization (struct context)>
    temporary_free(cx);
  }
}
```

Like we commented above, the boolean variable `loading` inside the `struct metafont` must be set

to false after the program is read and executed for the first time. We defined in this Subsection the code that reads the entire program for the first time (function `eval_program`). We just need to update that code to set this variable to the correct value after the program is executed:

Section: After Program Execution:

```
mf -> loading = false;
```

5.2. Splitting List of Statements in Individual Statements

A list of statements is a possibly empty sequence of statements, which can be empty, simple or compound and could be placed in any order:

```
<List of Statements> -> <Empty> |
                        <Empty> ; <List of Statements> |
                        <Simple Statement> ; <List of Statements> |
                        <Composite Statement> <List of Statements>
```

An empty or simple statement always must be terminated by a semicolon. But compound statements do not need the semicolon as delimiter because they already have some sequence of tokens that acts as an opening for them, and another sequence that acts as their closing. Between their opening and closing, they can contain other statements, which also can be simple, empty or compound. For example, an `if` and all the conditional code that is executed if some condition is true are all part of a single compound statement.

To correctly evaluate our programs, we must be able to identify and split single statements from a list of statements. Even when these compound statements are more complex to be delimited. However, for now, in this Subsection we will assume a simple heuristics to split a list of statements in individual statements: we will assume that all statements are delimited by semicolons.

The function that will split and identify individual statemetns is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_list_of_statements(struct metafont *mf, struct context *cx,
                             struct generic_token *begin_list,
                             struct generic_token *end_list);
```

This function job is just split each individual statement from the list and pass the individual statements to another function that will execute them. The code is:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_list_of_statements(struct metafont *mf, struct context *cx,
                             struct generic_token *begin_list,
                             struct generic_token *end_list){
  struct generic_token *begin, *end = NULL;
  begin = begin_list;
```

<Section to be inserted: Before Evaluating Code>

```
  while(begin != NULL){
    // This loop skips empty statements and places 'begin' in the beginning
    // of the next non-empty statement:
    while(begin != NULL && begin -> type == TYPE_SEMICOLON){
      if(begin != end_list)
        begin = begin -> next;
      else
        begin = NULL;
    }
    end = begin;
    // This loop finds the last token before a ';' and places 'end' there:
    if(end != NULL){
      while(end != end_list && end -> next -> type != TYPE_SEMICOLON)
```

```
        end = end -> next;
    }
    // If we found something, we evaluate the individual statement:
    if(begin != NULL){
      if(!eval_statement(mf, cx, begin, &end))
        return false;
      // After evaluation, we place 'begin' in the ';' following 'end':
      if(end != end_list)
        begin = end -> next;
      else
        begin = NULL;
    }
  }
```

<Section to be inserted: After Evaluating Code>

```
  return true;
}
```

The above function walks over the entire list of tokens. The loop invariant for the most external `while` is that when the iteration begins, the pointer `begin` is always either in a semicolon token or in the token that is the beginning of the next non-empty statement. In the first case, we move forward the pointer ignoring the semicolons: we are just skipping empty statements. After `begin` is placed in the beginning of some non-empty statement, we update `end` to the position of the next token before a semicolon. This way, both tokens now delimit an individual statement, which we pass to the `eval_statement` function. After the evaluation, if we found no errrors, we place `begin` in the next token, which should be a semicolon, preserving the loop invariant.

Or at least, this is the illusion kept in that function. In reality, as we passed the `end` token pointer to `eval_statement` by reference, not by value, this pointer position can be updated when the statement is evaluated. This happens when a compound function is evaluated: the pointer can be moved to a backward or forward position based on which is the real next statement to be executed if we take into account the rules for compound statements. As this logic is handled in `eval_statement`, which we will define in the following sections, the function `eval_list_of_statements` becomes very simple.

6. A Compound Statement: The Compound Block

6.1. Supporting Compound Statements

As we seen in the last Section, there are compound statements that can be composed by several other statements. The gramatical rules for them are:

```
<Compound Statement> -> <Compound Block> |
                        <Conditional> |
                        <Iteration> |
                        <Character Definition>
<Compound Block> -> begingroup <List of Statements> endgroup
<Conditional> -> if <Boolean Expression> :
                   <List of Statements>
                 fi
<Iteration> -> for <For Header>:
                 <List of Statements>
               endfor
<Character Definition> -> beginchar <Character Description>
                            <List of Statements>
                          endchar
```

By the above rules, the compound statements are three: they can be blocks (begins with `begingroup` and ends with `endgroup`), conditionals (begins with `if`, ends with `fi`), iterations (begin withs `for` and ends with `endfor`) or charactere definition (begins with `beginchar`, ends with `endchar`).

As there are list of statements inside compound statements,we can have multiple compound statements nested inside each other. But each one must be closed in the right order. If the last beginning of compound statement was a `begingroup` and then we find a `fi` or `endchar`, then the program is not correct.

To read correctly compound statements, we need to take into account their delimiting tokens, which we define below:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_BEGINGROUP,            // Symbolic token 'begingroup'
TYPE_ENDGROUP,              // Symbolic token 'endgroup'
TYPE_IF,                    // Symbolic token 'if'
TYPE_FI,                    // Symbolic token 'fi'
TYPE_BEGINCHAR,             // Symbolic token 'beginchar'
TYPE_ENDCHAR,               // Symbolic token 'endchar'
```

And we add them to the list of reserved keywords:

Section: List of Keywords:

```
"begingroup", "endgroup", "if", "fi", "beginchar", "endchar",
```

The tokens that delimit iterations are `TYPE_FOR` and `TYPE_ENDFOR`. They were already defined, when we created our lexer. Their tokens need to store additional information and have a more complex initialization, so we had to define them when creating the lexer.

And now the function that will evaluate individual statements.

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_statement(struct metafont *, struct context *,
                    struct generic_token *begin, struct generic_token **end);
```

The function gets pointers for the first and last tokens that should delimit an individual statement. For simple statements, both pointers are correct. For compound statements, that are not delimited by semicolons as seen in the last Section, the first pointer is correct, but the end pointer is not. Correcting the last pointer is the responsability of `eval_statement`, as this is the function that know the rules for all individual statements.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_statement(struct metafont *mf, struct context *cx,
                    struct generic_token *begin, struct generic_token **end){
                     <Section to be inserted: Statement: Compound>
                     <Section to be inserted: Statement: Declaration>
                     <Section to be inserted: Statement: Assignment>
                      <Section to be inserted: Statement: Command>
  // If we are here, we could not identify the statement:
  RAISE_ERROR_UNKNOWN_STATEMENT(mf, cx, OPTIONAL(begin -> line));
  return false;
}
```

The above function will test if it is evaluating a simple or compound statement. If none of the above options is identified, then we should raise an error of unknown statement.

Now let's deal with the compound statement. The first thing to recall is that it will be important for the language to know in which nesting level we are to know the scope of each declared variable. Because of this, our context will store the current nesting level. Each beginning of compound statement (`begingroup`, `if`, `beginchar` and `for`) should increase by 1 the nesting level and each ending of compound statement (`endgroup`, `fi`, `endchar` and `endfor`) should decrease by 1. Therefore, we also must to know which token should end the most recent nesting level. We should store in a stack the expected tokens that should

end our nestings. Each element in this stack should store a token to the beginning of its compound statement in case we need this information. We declare the nesting level and the stack of expected ending tokens below:

Section: Attributes (struct context):

```
  int nesting_level;
  struct linked_token *end_token_stack;
```

The nesting level should be initialized to zero and the ending token stack should begin empty:

Section: Initialization (struct context):

```
  cx -> nesting_level = 0;
  cx -> end_token_stack = NULL;
```

To help us to manage nesting levels, we will create two auxiliary functions. Both of them get as argument a `struct context` with the evaluation context and a token. The first will begin a new nesting level using that token and the second will end the current nesting level. Both of them will return error if an invalid token is supplied:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool begin_nesting_level(struct metafont *mf, struct context *cx,
                         struct generic_token *tok);
bool end_nesting_level(struct metafont *mf, struct context *cx,
                       struct generic_token *tok);
```

The first function implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool begin_nesting_level(struct metafont *mf, struct context *cx,
                         struct generic_token *tok){
  struct linked_token *end_token;
  end_token = (struct linked_token *)
                    temporary_alloc(sizeof(struct linked_token));
  if(end_token == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(tok -> line));
    return false;
  }
  // Only four types of tokens will be passed for this function:
  switch(tok -> type){
  case TYPE_BEGINGROUP:
    end_token -> type = TYPE_ENDGROUP;
    break;
  case TYPE_IF:
    end_token -> type = TYPE_FI;
    break;
  case TYPE_BEGINCHAR:
    end_token -> type = TYPE_ENDCHAR;
    break;
  case TYPE_FOR:
    end_token -> type = TYPE_ENDFOR;
    break;
  default:
    return false;
  }
  end_token -> link = tok;
#if defined(W_DEBUG_METAFONT)
```

```
  end_token -> line = tok -> line;
#endif
  cx -> nesting_level ++;
  end_token -> next = (struct generic_token *) (cx -> end_token_stack);
  cx -> end_token_stack = end_token;
  return true;
}
```

And the second function:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool end_nesting_level(struct metafont *mf, struct context *cx,
                       struct generic_token *tok){
  struct linked_token *end_tok = cx -> end_token_stack;
  if(end_tok == NULL){
    RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(tok -> line), tok);
    return false;
  }
  else if(end_tok-> type != tok -> type){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(tok -> line),
                               end_tok -> type, tok);
    return false;
  }
  cx -> nesting_level --;
  cx -> end_token_stack = (struct linked_token *) end_tok -> next;
  if(temporary_free != NULL)
    temporary_free(end_tok);
  return true;
}
```

Notice that if an error happens inside a compound statement, we could stop the code execution before calling the `end_nesting_level`. Bacause of this, when freeing the memory for a context, we need to destroy any remaining token in the stack of expected ending tokens:

Section: Finalization (struct context):

```
if(temporary_free != NULL){
  while(cx -> end_token_stack != NULL){
    struct linked_token *end_tok = cx -> end_token_stack;
    cx -> end_token_stack = (struct linked_token *) (end_tok -> next);
    temporary_free(end_tok);
  }
}
```

6.2. The Statement `begingroup...endgroup`

How exactly a statement is evaluated? First, recall that `eval_list_of_statements` function splits the code using semiconlons as delimiters. Next, each part is given as input to function `eval_statement`. As already mentioned, this behaviour may be a little naive, as compound statements are full statements that can contain semicolons and other statements. Consider the following list of tokens:

```
[begingroup][T1][T2][;][T3][T4][;][endgroup][;][T5];
```

In this sequence, the first three tokens would be passed to be evaluated. Next would be `[T3][T4]`. And finally, `[endgroup][T5]`. This is wrong, as we ideally should evaluate first the beginning of a new nesting block with `[begingroup]`. Next, each statement in the block. Next, we end the nesting with `[endgroup]`. And only then we evaluate `[T5]`.

Fortunately, it's easy to evaluate things in this order. When we get the three first tokens in the example above, we will evaluate only `begingroup`, and we will give back the other two tokens to be evaluated later. We can do this changing the position of the pointer that marks the end of current statement, like explained in Subsection 6.1.

When we get the `endgroup`, we will do the same: we evaluate only the `endgroup` part to close the compound statement and give back any remaining tokens.

The code to deal with any statement that begins with `begingroup` is:

Section: Statement: Compound:

```
if(begin -> type == TYPE_BEGINGROUP){
  begin_nesting_level(mf, cx, begin);
  // This sets the beginning of next statement as the token after 'begingroup':
  *end = begin;
  return true;
}
```

In the same way, if we get an statement that begins with `endgroup`, we evaluate only this first token finishing the current nesting level:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_ENDGROUP){
  if(!end_nesting_level(mf, cx, begin))
    return false;
  *end = begin;
  return true;
}
```

What if we finish the evaluation of a program, but we did not finished some compound statement? This kind of error usually is detected by the lexer. Except in case of an unfinished `begingroup`. In this case, we add an error detection at the end of the program to catch this error:

Section: After Program Execution:

```
if(cx -> nesting_level > 0){
  RAISE_ERROR_MISSING_TOKEN(mf, cx, OPTIONAL(cx -> end_token_stack -> line),
                            TYPE_ENDGROUP);
  return false;
}
```

7. Variable Declarations

The variable declaration is the first of the simple statements that we will describe. The syntax to declare variables is:

```
<Simple Statement> -> <Declaration> | ...
<Declaration> -> <Type> <List of Declarations>
<Type> -> boolean | string | path | pen | picture | transform | pair |
          numeric
<List of Declarations> -> <Tag> | <Tag> , <List of Declarations>
```

A "tag" is basically a symbolic token without a predefined meaning in the language. For example, "`tag`" is a tag, but "`begingroup`" is not.

To interpret a declaration, we will introduce the following special tokens:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_T_BOOLEAN,              // Symbolic token 'boolean'
TYPE_T_PATH,                 // Symbolic token 'path'
TYPE_T_PEN,                  // Symbolic token 'pen'
```

```
TYPE_T_PICTURE,                 // Symbolic token 'picture'
TYPE_T_TRANSFORM,               // Symbolic token 'transform'
TYPE_T_PAIR,                    // Symbolic token 'pair'
TYPE_T_NUMERIC,                 // Symbolic token 'numeric'
```

And then we add them to the list of keywords:

Section: List of Keywords (continuation):

```
"boolean", "path", "pen", "picture", "transform", "pair", "numeric",
```

When a variable is declared, we need to do two things:

1) We should allocate a new structure to store its content. The variable cannot have the same name than any reserved keyword nor should have the same name than some special variables like: `w`, `h`, `d`, `currentpicture` ou `currentpen`. Otherwise, we raise an error. A default value depending of its type if stored in the variable, but it should not be used before being initialized.

2) We iterate over the following tokens in the list of tokens until the end of the list or until the current nesting level ends. When we find a symbolic token with the variable name, we update its pointer to the newly allocated variable.

As each variable type has its own information and content, the specifics about how the variable is created can be different. What all variables have in common is the following structure:

Section: Local Data Structures (metafont.c) (continuation):

```
// Generic variable
struct variable{
  int type;
  struct variable *next;
};
```

All variables first store their type, then a pointer for the next variable. Depending on the variable type, more information could be stored after this pointer.

If we are dealing with a global variable, we could want to preserve and store its name, just in case if later the user would want to change the variable content indentifying it by the name. In this case, we would want to use the following structure to store the name with the variable:

Section: Local Data Structures (metafont.c) (continuation):

```
struct named_variable{
  char *name;
  struct named_variable *next;
  struct variable *var;
};
```

And the metafont structure will have pointers for these named global variables, and also for other variables stored there:

Section: Attributes (struct metafont):

```
  struct named_variable *named_variables;
  struct variable *variables;
```

These pointers are initialized with NULL:

Section: Initialization (struct metafont):

```
mf -> named_variables = NULL;
mf -> variables = NULL;
```

To disallocate a list of variables storen on the meta-font, we just iterate over their linked list. Some variables, which are more complex, could need additional operations before removing them, but we will deal with them later. Here we disallocate the variables whose names we did not preserve:

Section: Finalization (struct metafont):

```
if(permanent_free != NULL){
  struct variable *v = (struct variable *) (mf -> variables);
  struct variable *next;
  while(v != NULL){
    next = (struct variable *) (v -> next);
            <Section to be inserted: Finalize Variable 'v' in 'struct metafont'>
    permanent_free(v);
    v = next;
  }
}
```

And to disallocate a list with preserved names, we act in a similar way, we just need to also disallocate the structure where we store their names:

Section: Finalization (struct metafont) (continuation):

```
if(permanent_free != NULL){
  struct named_variable *named = (struct named_variable *)
                                   (mf -> named_variables);
  struct named_variable *next;
  while(named != NULL){
    struct variable *v = (struct variable *) named -> var;
    next = (struct named_variable *) (named -> next);
    permanent_free(named -> name);
            <Section to be inserted: Finalize Variable 'v' in 'struct metafont'>
    permanent_free(v);
    permanent_free(named);
    named = next;
  }
}
```

In the case of variables that are declared inside a character specification to be rendered, they will be stored in contexts, not in the metafont struct. After all, their duration always will be temporary:

Section: Attributes (struct context):

```
  struct variable *variables;
```

The list of variables is initialized as empty:

Section: Initialization (struct context) (continuation):

```
cx -> variables = NULL;
```

And we finalize it exactly as the list of global variables:

Section: Finalization (struct context) (continuation):

```
if(temporary_free != NULL){
  struct variable *v = (struct variable *) (cx -> variables);
  struct variable *next;
  while(v != NULL){
    next = (struct variable *) (v -> next);
            <Section to be inserted: Finalize Variable 'v' in 'struct context'>
    temporary_free(v);
    v = next;
  }
}
```

And now let's interpret the variable declaration:. If we find a variable type in the beginning of a new statement, this means that this statement is a variable declaration. We just need to get the type and iterate

over all the declared variables creating them. THe variable names are always intercalated by commas if there is more than one variable being declared:

Section: Statement: Declaration:

```
else if(begin -> type >= TYPE_T_BOOLEAN && begin -> type <= TYPE_T_NUMERIC){
  int type = begin -> type;
  struct symbolic_token *variable = (struct symbolic_token *) (begin -> next);
  if(variable == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  if(begin == *end){
    RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  do{
    if(variable -> type != TYPE_SYMBOLIC || !strcmp(variable -> value, "w") ||
       !strcmp(variable -> value, "h") || !strcmp(variable -> value, "d") ||
       !strcmp(variable -> value, "currentpen") ||
       !strcmp(variable -> value, "currentpicture")){
      RAISE_ERROR_INVALID_NAME(mf, cx, OPTIONAL(variable -> line),
                               (struct generic_token *) variable,
                               variable -> type);
      return false;
    }
                    <Section to be inserted: Insert Declared Variable>
    if(variable != (struct symbolic_token *) *end)
      variable = (struct symbolic_token *) (variable -> next);
    else{
      variable = NULL;
      continue;
    }
    if(variable -> type != TYPE_COMMA){
      RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(variable -> line),
                                 TYPE_COMMA, (struct generic_token *) variable);
      return false;
    }
    if(variable == (struct symbolic_token *) *end){ // Error: ended with ','
      RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, OPTIONAL((*end) -> line));
      return false;
    }
    variable = (struct symbolic_token *) (variable -> next);
  } while(variable != NULL);
  return true;
}
```

Creating and inserting a new variable in the code fragment above means:

Section: Insert Declared Variable:

```
{
  void *variable_pointer;
  if(mf -> loading){ // Variable should be stored in 'struct metafont'
    if(cx -> nesting_level != 0 || variable -> value[0] == '_')
```

```
      variable_pointer = insert_variable(mf, type, &(mf -> variables));
    else
      variable_pointer = insert_named_global_variable(mf, type, variable);
  }
  else
    variable_pointer = insert_variable(mf, type, &(cx -> variables));
  if(variable_pointer == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(variable -> line));
    return false;
  }
  update_token_pointer_for_variable(variable, variable_pointer);
}
```

The function that inserts a new global variable without its name is:

Section: Local Function Declaration (metafont.c) (continuation):

```
struct variable *insert_variable(struct metafont *mf,
                                 int type,
                                 struct variable **target);
```

It will allocate a new variable and store it in the local pointed by `target`:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct variable *insert_variable(struct metafont *mf,
                                 int type,
                                 struct variable **target){
  struct variable *var;
  size_t var_sizes[] = {
    sizeof(struct boolean_variable), sizeof(struct path_variable),
    sizeof(struct pen_variable), sizeof(struct picture_variable),
    sizeof(struct transform_variable), sizeof(struct pair_variable),
    sizeof(struct numeric_variable)
  };
  size_t var_size = var_sizes[type-TYPE_T_BOOLEAN];
  if(mf -> loading)
    var = (struct variable *) permanent_alloc(var_size);
  else
    var = (struct variable *) temporary_alloc(var_size);
  if(var != NULL){
    var -> type = type;
    var -> next = NULL;
                  <Section to be inserted: Initializing New Variable>
  }
  if(*target == NULL)
    *target = var;
  else{
    struct variable *p = (struct variable *) (*target);
    while(p -> next != NULL)
      p = (struct variable *) p -> next;
    p -> next = var;
  }
  return var;
}
```

Inserting a named variable is done with the following function:

Section: Local Function Declaration (metafont.c) (continuation):

```
struct variable *insert_named_global_variable(struct metafont *mf,
                                              int type,
                                              struct symbolic_token *var);
```

The function works in a similar way, first allocating the structure with the variable nam and then placing the variable there. As we only preserve the name of global variables, we surely should use the permanent allocation function:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct variable *insert_named_global_variable(struct metafont *mf,
                                              int type,
                                              struct symbolic_token *var){
  struct named_variable *named;
  struct variable *new_var;
  size_t name_size;
  named = (struct named_variable *)
              permanent_alloc(sizeof(struct named_variable));
  if(named == NULL)
    return NULL;
  name_size = strlen(var -> value) + 1;
  named -> name = (char *) permanent_alloc(name_size);
  if(named -> name == NULL){
    if(permanent_free != NULL)
      permanent_free(named);
    return NULL;
  }
  memcpy(named -> name, var -> value, name_size);
  named -> next = NULL;
  named -> var = NULL;
  new_var = insert_variable(mf, type, &(named -> var));
  if(new_var == NULL){
    if(permanent_free != NULL){
      permanent_free(named -> name);
      permanent_free(named);
      return NULL;
    }
  }
  if(mf -> named_variables == NULL)
    mf -> named_variables = named;
  else{
    struct named_variable *p = (struct named_variable *)
                                    mf -> named_variables;
    while(p -> next != NULL)
      p = (struct named_variable *) p -> next;
    p -> next = named;
  }
  return new_var;
}
```

And finally, the function that search in a list of tokens for symbolic tokens with the same name that the

allocated new variable and updating each one to make them point to the newly allocated variable position:

Section: Local Function Declaration (metafont.c) (continuation):

```
void update_token_pointer_for_variable(struct symbolic_token *var_token,
                                       struct variable *var_pointer);
```

The function will walk in the linked list starting in the token next to the declared variable. And t will stop only when there is no more tokens or when the variable do not exist anymore because we are not more in its nesting level (it happens when we find the `endgroup`, `fi` or `endchar` which ends the current nesting level):

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void update_token_pointer_for_variable(struct symbolic_token *var_token,
                                       struct variable *var_pointer){
  struct symbolic_token *p = (struct symbolic_token *) var_token -> next;
  int nesting_level = 0;
  while(p != NULL && nesting_level >= 0){
    if(p -> type == TYPE_BEGINGROUP || p -> type == TYPE_IF ||
       p -> type == TYPE_BEGINCHAR || p -> type == TYPE_FOR)
      nesting_level ++;
    else if(p -> type == TYPE_ENDGROUP || p -> type == TYPE_FI ||
            p -> type == TYPE_ENDCHAR || p -> type == TYPE_ENDFOR)
      nesting_level --;
    else if(p -> type == TYPE_SYMBOLIC){
      if(!strcmp(p -> value, var_token -> value)){
        p -> var = var_pointer;
      }
    }
    p = (struct symbolic_token *) (p -> next);
  }
}
```

7.1. Numeric Variables

A numeric variable will be stored in the following structure:

Section: Local Data Structures (metafont.c) (continuation):

```
struct numeric_variable{
  int type; // Deve set 'TYPE_T_NUMERIC'
  void *next;
  float value;
};
```

Comparing numeric variables with generic variables, the difference is that we have an additional floating point variable in the structure called `value`. It will store the numeric value. When we create a numeric variable, we initialize it with NaN, which represents an unknown value:

Section: Initializing New Variable:

```
if(type == TYPE_T_NUMERIC){
  ((struct numeric_variable *) var) -> value = NAN;
}
```

Using the macro `NAN` requires the mathematical header:

Section: Local Headers (metafont.c) (continuation):

```
#include <math.h>
```

In the case of numeric variables, nothing additional is needed when we deallocate them, as they have no additional structure besides a single floating-point number.

However, besides the variables declared by the users, we will have some additional internal variables. They always will be present and do not need to be declared. We have exactly ten internal numeric variables: `pt`, `cm`, `mm`, `color_r`, `color_g`, `color_b`, `color_a`, `monospace`, `w`, `h` and `d`. Basically the first three will store how many pixels correspond to 1pt, 1cm and 1mm respectively. The next one store if we are using a monospace font or not. The next four will store the default color used to render our glyphs. Each one correspond to a different color channel: red, green, blue and alpha (transparency). The next three will store the width, height and depth of some glyph that we are rendering, or will store zero when we are not rendering something.

The first eight of such internal numeric variables will be declared in the meta-font struct:

Section: Attributes (struct metafont) (continuation):

```
struct numeric_variable *internal_numeric_variables;
```

The other three will be declared in the context structure, as they are bound to the context of which character we are rendering (if any):

Section: Attributes (struct context) (continuation):

```
struct numeric_variable *internal_numeric_variables;
```

During the initialization we will allocate space for the seven first variables and fill their values:

Section: Initialization (struct metafont) (continuation):

```
mf -> internal_numeric_variables =
             permanent_alloc(8 * sizeof(struct numeric_variable));
((struct numeric_variable *) mf -> internal_numeric_variables)[0].value =
   ((double) dpi) / 72.0; // 1in = 72 pt
((struct numeric_variable *) mf -> internal_numeric_variables)[1].value =
   ((double) dpi) / 2.54; // 1in = 2.54cm
((struct numeric_variable *) mf -> internal_numeric_variables)[2].value =
   ((double) dpi) / 25.4; // 1in = 25.4mm
// Default color: (0, 0, 0, 1) (black opaque)
((struct numeric_variable *) mf -> internal_numeric_variables)[3].value = 0.0;
((struct numeric_variable *) mf -> internal_numeric_variables)[4].value = 0.0;
((struct numeric_variable *) mf -> internal_numeric_variables)[5].value = 0.0;
((struct numeric_variable *) mf -> internal_numeric_variables)[6].value = 1.0;
// By default, the font do not begin in monospace mode:
((struct numeric_variable *) mf -> internal_numeric_variables)[7].value = 0.0;
{
  int i;
  for(i = 0; i < 8; i ++){
    ((struct numeric_variable *)
      mf -> internal_numeric_variables)[i].type = TYPE_T_NUMERIC;
    ((struct numeric_variable *)
      mf -> internal_numeric_variables)[i].next = NULL;
  }
}
```

The other three ariables will be allocated not when we initialize the font, but when we initialize the execution context:

Section: Initialization (struct context) (continuation):

```
cx -> internal_numeric_variables =
             temporary_alloc(3 * sizeof(struct numeric_variable));
```

```
((struct numeric_variable *) cx -> internal_numeric_variables)[0].value = 0.0;
((struct numeric_variable *) cx -> internal_numeric_variables)[1].value = 0.0;
((struct numeric_variable *) cx -> internal_numeric_variables)[2].value = 0.0;
{
  int i;
  for(i = 0; i < 3; i ++){
    ((struct numeric_variable *)
      cx -> internal_numeric_variables)[i].type = TYPE_T_NUMERIC;
    ((struct numeric_variable *)
      cx -> internal_numeric_variables)[i].next = NULL;
  }
}
```

We also will create the following macros to access each one of these variables more easily:

Section: Local Macros (metafont.c) (continuation):

```
#define INTERNAL_NUMERIC_PT   0
#define INTERNAL_NUMERIC_CM   1
#define INTERNAL_NUMERIC_MM   2
#define INTERNAL_NUMERIC_R    3
#define INTERNAL_NUMERIC_G    4
#define INTERNAL_NUMERIC_B    5
#define INTERNAL_NUMERIC_A    6
#define INTERNAL_NUMERIC_MONO 7
#define INTERNAL_NUMERIC_W    0
#define INTERNAL_NUMERIC_H    1
#define INTERNAL_NUMERIC_D    2
```

To finalize the metafont structure, we need to deallocate these internal variables:

Section: Finalization (struct metafont) (continuation):

```
if(permanent_free != NULL)
  permanent_free(mf -> internal_numeric_variables);
```

Likewise, when we finalize a context, we also deallocate its three numeric variables:

Section: Finalizao (struct context) (continuation):

```
if(temporary_free != NULL)
  temporary_free(cx -> internal_numeric_variables);
```

When our lexer creates a new symbolic token which is not a reserved word, it should check if this token have the same name than an internal variable. If so, we can then set correctly the pointer to the variable:

Section: Set Pointer to Internal Variable:

```
if(!strcmp(buffer, "pt")){
  new_token -> var =
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_PT]);
}
else if(!strcmp(buffer, "cm")){
  new_token -> var =
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_CM]);
}
else if(!strcmp(buffer, "mm")){
  new_token -> var =
```

```
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_MM]);
}
else if(!strcmp(buffer, "color_r")){
  new_token -> var =
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_R]);
}
else if(!strcmp(buffer, "color_g")){
  new_token -> var =
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_G]);
}
else if(!strcmp(buffer, "color_b")){
  new_token -> var =
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_B]);
}
else if(!strcmp(buffer, "color_a")){
  new_token -> var =
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_A]);
}
else if(!strcmp(buffer, "monospace")){
  new_token -> var =
          &(((struct numeric_variable *)
                mf -> internal_numeric_variables)[INTERNAL_NUMERIC_MONO]);
}
```

7.2. Pair Variables

A pair variable will store point coordinates. They will be stored in the following structure:

Section: Local Data Structures (metafont.c) (continuation):

```
struct pair_variable{
  int type; // Must be 'TYPE_T_PAIR'
  void *next;
  float x, y;
};
```

The difference is that they have space for two floating point values instead of one. Initially we will represent the first of them with NaN to represent a non-initialized pair:

Section: Initializing New Variable (continuation):

```
if(type == TYPE_T_PAIR){
  ((struct pair_variable *) var) -> x = NAN;
}
```

7.3. Transform Variables

If a pair variable is a tuple with two numeric values, then a transform variable is a tuple with six different values. Let's assume that a transform variable is (a, b, c, d, e, f), what it represents is a linear transformation in which the pair (x, y) is transformed in (x', y') by the following matrix multiplication:

$$[x\ y\ 1]\begin{bmatrix} c & e & 0 \\ d & f & 0 \\ a & b & 1 \end{bmatrix} = [x'\ \ y'\ \ 1]$$

Or: $(x', y') = (a + cx + dy, b + ex + fy)$.

The order in which we store and use the elements may appear strange, but we store them in that order for compatibility with the notation from original METAFONT created by Knuth.

A transform variable therefore, needs to store a linear transformation in a matrix form:

Section: Local Data Structures (metafont.c) (continuation):

```
struct transform_variable{
  int type; // Deve ser 'TYPE_T_TRANSFORM'
  void *next;
  float value[9];
};
```

A non-initialized transform variable, by default, has its first value marked as NAN:

Section: Initializing New Variable (continuation):

```
if(type == TYPE_T_TRANSFORM)
  ((struct transform_variable *) var) -> value[0] = NAN;
```

There is an internal transform variable that always is presente and that represents no transformation at all. A transform that do not change the point, that in the metrix notation represents multiplying by the identity matrix. This transform is represented by the tuple $(0, 0, 1, 0, 0, 1)$.

The internal transform variables will be stored here:

Section: Attributes (struct metafont) (continuation):

```
struct transform_variable *internal_transform_variables;
```

During initialization we allocate memory for the internal variables and initialize them:

Section: Initialization (struct metafont) (continuation):

```
mf -> internal_transform_variables =
  (struct transform_variable *)
    permanent_alloc(sizeof(struct transform_variable));
if(mf -> internal_transform_variables == NULL){
  if(permanent_free != NULL)
    permanent_free(mf);
  return NULL; //ERRO: Sem memria suficiente
}
// The transform variable 'identity':
mf -> internal_transform_variables[0].type = TYPE_T_TRANSFORM;
INITIALIZE_IDENTITY_MATRIX(mf -> internal_transform_variables[0].value);
```

In the finalization, we need to free the allocated memory:

Section: Finalization (struct metafont) (continuation):

```
if(permanent_free != NULL)
  permanent_free(mf -> internal_transform_variables);
```

The variable `identity` will be stored as the first position in the array of internal transform variables:

Section: Local Macros (metafont.c) (continuation):

```
#define INTERNAL_TRANSFORM_IDENTITY 0
```

And when reading the source code, we can set correctly the pointer each time we find a token referencing this internal variable:

Section: Set Pointer to Internal Variable (continuation):

```
else if(!strcmp(buffer, "identity"))
  new_token -> var =
          &(mf -> internal_transform_variables[INTERNAL_TRANSFORM_IDENTITY]);
```

7.4. Path Variables

Path variables are the most complex variables in the WeaveFont language. They store a sequence of curves, where the next curve begins in the same point where the last ended. Each curve can be represented in several different formats. But all them are cubic Bezir curves. The path may be cyclic, when the last curve ends in the same point where the first curbe begins. To store the path contents, we may need to allocate additional structures. Basically a path variable store most of it data in a "point" array, where each point in the array is a literal point of the curve with additional data, or is a pointer to several points that compose a subpath. Some of these points may have some missing information that we need to fill before trying to use the path. The variable that stores a path is:

Section: Local Data Structures (metafont.c) (continuation):

```
struct path_variable{
  int type; // Must be 'TYPE_T_PATH'
  void *next;
  bool permanent; // Which allocation was used: permanent or temporary
  bool cyclic;
  int length, number_of_points, number_of_missing_directions;
  struct path_points *points;
};
```

The number of points in a path variable is always greater or equal than the variable `length`, as each position in the array with `length` elements is either a point or a subpath with 1 or more points.

In fact, each point in the array may be in one of three different formats:

Section: Local Macros (metafont.c) (continuation):

```
#define UNINITIALIZED_FORMAT 0 // Do not use, it's just for initialization
#define PROVISIONAL_FORMAT   1 // Still collecting data about the point
#define SUBPATH_FORMAT       2 // Pointer for several points in subpath
#define FINAL_FORMAT         3 // Final format, ready for use
```

Now we describe how is the final format, which we expect for our points.

A Cubic Bezir Curve is defined by two extremity points (z_1 and z_4) and two control points (z'_2 and z'_3). The two extremity curves are always part of the curve. To obtain the other curve points given the four points (z_1, z'_2, z'_3, z_4), we use the following procedure:

1) Get three intermediate points: z'_{12} is the point halfway the points z_1 and z'_2, z'_{23} is the point halfway between z'_2 and z'_3 and z_{34} is the point halfway between z'_3 and z_4.

2) Get two intermediate points: z'_{123} is the point halfway between z'_{12} and z'_{23} while z'_{234} is the point halfway between z'_{23} and z'_{34}.

3) Get the new curve point z_{1234} located halfway between z'_{123} and z'_{234}.

4) Generate the remaining curve points repeating this procedure recursively over the points $(z_1, z'_{12}, z'_{123}, z_{1234})$ and over $(z_{1234}, z'_{234}, z'_{34}, z_4)$.

It is also possible to define such curves using a formula. Given the extremity points and the control points, a Cubic Bezir Curve is defined by the following formula if we vary t between 0 and 1:

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z'_2 + 3(1-t) t^2 z'_3 + t^3 z_4$$

The data structure that will represent a sequence of Cubic Bezir Curves is:

Section: Local Data Structures (metafont.c) (continuation):

```
struct path_points{
  int format; // The point format
  union{
    // Final format: extremity point and 2 control points
    struct{
      float x, y; // Extremity point
      float u_x, u_y, v_x, v_y;   // Next 2 control points after it
    } point;
    // Subpath format: pointer to subpath
    struct path_variable *subpath;
    // Provisional format: other info besides the control points
    struct{
      float x, y; // Extremity point
      float dir1_x, dir1_y, dir2_x, dir2_y; // Direction specifier
      float tension1, tension2;
      bool atleast1, atleast2;
    } prov;
  };
};
```

Basically its points will be stored in the pointer `points`, an array of structs whose number of elements is equal `length`. Each structure will represent a point (x, y) or a subpath (if `subpath` is not null, and in this case we ignore x and y). To check the number of points in a path variable, one could read variable `total_length` that considers the points stored recursively in other linked path variables. The variable `length` stores only the size of array `points`.

The control points defined by `u_x`, `u_y`, `v_x` and `v_y` are the control points between the current point or subpath and the next one if it exists. If we have a cyclic path, the control points in the last point or subpath show how it is connected to the first point in the path. If we are not in a cyclic path, then the control points of the last point is ignored.

We will represent a non-initialized path variable setting its length as -1:

Section: Initializing New Variable (continuation):

```
if(type == TYPE_T_PATH){
  ((struct path_variable *) var) -> length = -1;
  ((struct path_variable *) var) -> permanent = mf -> loading;
}
```

When we remove a global variable, if it is a path variable and if it is initialized, then we also need to remove the allocated list of points. If the variable is in the meta-font structure, we use permanent disallocation:

Section: Finalize Variable 'v' in 'struct metafont':

```
if(v -> type == TYPE_T_PATH){
  struct path_variable *path = (struct path_variable *) v;
  if(path -> length != -1 && permanent_free != NULL)
    path_recursive_free(permanent_free, path, false);
}
```

If the variable is in the context structure we do the same, but using a different disallocation function:

Section: Finalize Variable 'v' in 'struct context':

```
if(v -> type == TYPE_T_PATH){
  struct path_variable *path = (struct path_variable *) v;
  if(path -> length != -1 && temporary_free != NULL)
    path_recursive_free(temporary_free, path, false);
}
```

As a path can contain subpaths and each subpath also can contain more subpaths, we will use a recursive function to disallocate their memory. Given a disallocation function and a pointer to a path variable, we disalloc all its subpaths, and then we disalloc the path variable. The function declaration is:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_recursive_free(void (*free_func)(void *),
                         struct path_variable *path,
                         bool free_first_pointer);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_recursive_free(void (*free_func)(void *),
                         struct path_variable *path,
                         bool free_first_pointer){
  if(free_func != NULL){
    int i;
    for(i = 0; i < path -> length; i ++){
      if(path -> points[i].format == SUBPATH_FORMAT)
        path_recursive_free(free_func, (struct path_variable *)
                                       path -> points[i].subpath, true);
    }
    free_func(path -> points);
    if(free_first_pointer)
      free_func(path);
  }
}
```

7.4.1. Removing Recursion from Path Variables

Pathh variables are the most complex variables in WeaveFont language. They can be stored in memory in several different format. However, some of these formats are provisional and has undesirable properties. For example, while we can represent a path storing its subpaths recursively using several pointers, this is not desirable because it makes hard to access quickly random points in the path.

After evaluating a path, it will be in the format in which it was more convenient to assemble given the expression that created the path. But before storing such variables, we should convert its format for the one that it is more convenient to handle in future operations.

The first of these conversions is removing recursion from paths. This will turn `length` and `total_length` equal, removing possible confusions and misunderstandings when these values differ. Besided this, no point in the path will point to a subpath: all point structures will indeeed represent a single point.

For this, we will define a function that copies a path variable, but while performing the copy, it will remove the recursion. The function that performs this copy has the following header, followed by the header of the auxiliary and recursive function that it will call:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool recursive_copy_points(struct metafont *mf, struct context *cx,
                           void *(*alloc)(size_t),
                           struct path_variable **target,
                           struct path_variable *source,
                           bool alloc_target); // Should we alloc target?
void recursive_aux_copy(struct path_points **dst,
                        struct path_variable *origin, int *missing_directions,
                        struct path_points **previous_point);
```

The function will use the allocation function given as argument, which could be the permanent or

temporary. Then it initializes the relevant variables in the target and copy recursively. This is also the moment where we can check if there is missing information in the path points. We will fill for the first time the variable `number_of_missing_directions`:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool recursive_copy_points(struct metafont *mf, struct context *cx,
                           void *(*alloc)(size_t),
                           struct path_variable **target,
                           struct path_variable *source,
                          bool alloc_target){
  struct path_points *p, *previous_point = NULL;
  if(alloc_target){
    *target = (struct path_variable *) alloc(sizeof(struct path_variable));
    if(*target == NULL){
      RAISE_ERROR_NO_MEMORY(mf, cx, 0);
      return false;
    }
  }
  (*target) -> cyclic = source -> cyclic;
  (*target) -> permanent = (alloc == permanent_alloc);
  (*target) -> length = source -> number_of_points;
  (*target) -> number_of_points = source -> number_of_points;
  (*target) -> number_of_missing_directions = 0;
  (*target) -> points = (struct path_points *)
                          alloc(sizeof(struct path_points) *
                                (*target) -> number_of_points);
  if((*target) -> points == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, 0);
    return false;
  }
  p = (*target) -> points;
  recursive_aux_copy(&p, source, &((*target) -> number_of_missing_directions),
                     &previous_point);
  return true;
}
```

The recursive function that copy each point:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void recursive_aux_copy(struct path_points **dst, struct path_variable *origin,
                        int *missing_directions,
                        struct path_points **previous_point){
  int index = 0;
  while(index <= origin -> length - 1){
    if(origin -> points[index].format != SUBPATH_FORMAT &&
       !isnan(origin -> points[index].point.x)){
      memcpy(*dst, &(origin -> points[index]),  sizeof(struct path_points));
      // If we have two successive equal points, there is no missing direction
      // specifiers between them:
      if(*previous_point != NULL &&
         (*previous_point) -> format == PROVISIONAL_FORMAT &&
         (*previous_point) -> prov.x == (*dst) -> prov.x &&
```

```
        (*previous_point) -> prov.y == (*dst) -> prov.y){
      if(isnan((*previous_point) -> prov.dir1_x))
        (*missing_directions) --;
      if(isnan((*previous_point) -> prov.dir2_x))
        (*missing_directions) --;
      (*previous_point) -> format = FINAL_FORMAT;
      (*previous_point) -> point.u_x = (*previous_point) -> point.x;
      (*previous_point) -> point.v_x = (*previous_point) -> point.x;
      (*previous_point) -> point.u_y = (*previous_point) -> point.y;
      (*previous_point) -> point.v_y = (*previous_point) -> point.y;
    }
    if((*dst) -> format == PROVISIONAL_FORMAT){
      // Aproveitamos para remover especificadores de direo invlidos:
      if((*dst) -> prov.dir1_x == 0.0 &&
         (*dst) -> prov.dir1_y == 0.0){
         (*dst) -> prov.dir1_x = NAN;
         (*dst) -> prov.dir1_y = NAN;
      }
      if((*dst) -> prov.dir2_x == 0.0 &&
         (*dst) -> prov.dir2_y == 0.0){
        (*dst) -> prov.dir2_x = NAN;
        (*dst) -> prov.dir2_y = NAN;
      }
      if(isnan((*dst) -> prov.dir1_x))
        (*missing_directions) ++;
      if(isnan((*dst) -> prov.dir2_x))
        (*missing_directions) ++;
    }
    *previous_point = *dst;
    (*dst) ++;
  }
  else if(origin -> points[index].format == SUBPATH_FORMAT){
    recursive_aux_copy(dst, (struct path_variable *)
                           (origin -> points[index].subpath),
                       missing_directions, previous_point);
    path_recursive_free(temporary_free, (struct path_variable *)
                       (origin -> points[index].subpath), true);
  }
  index  ++;
 }
 return;
}
```

7.4.2. Tension and Direction Specifiers

We saw that a point that is part of a path can be on a provisional format, in addition to its final format and its pointer format. When this occurs, it has a very large number of variables. In particular, it has two numerical values representing its tension and has two pairs representing the next two direction specifiers.

The first direction specifier must be considered as a vector containing the turning angle of the curve trajectory at the first extremity point and the second is the same for the next endpoint. The turning angle can be seen as the angle of the "wheels" of a vehicle at that point, if the vehicle was making the trajectory of the curve. We may represent such information in code below representing a section of a path:

`{w0} p1 {w1} .. tension t1 and t2 {w2} p2 {w3}`

Note that if $w_0 \neq w_1$ or $w2 \neq w_2$, then there will be a break in curve at its end point, there will be a "tip", a non-derivable part in the curve.

Tension should represent how "loose" or "tight" is a curve. A curve with greater tension will have a smaller perimeter and will tend to approaches a straight line as tension increases.

Having such information, how to convert the curve point from the provisional format to the final format? Using the segment of curve above, this is done with the help of the formulas below:

$$\theta = arg(w_1/(z_2 - z_1)) \;\; \phi = arg((z_2 - z_1)/w_2)$$

It is not the direction vectors, but the angle computed above that is actually used to compute the control points from the final format of the point. The difference between the first format provisional and the second is that the second has already calculated these angles θ and ϕ and stored the results, discarding the direction vectors.

To compute function arg above, we will need the header:

Section: Local Headers (metafont.c) (continuation):

```
#include <complex.h>
```

The control points u and v are obtained with formulas:

$$u = z_n + e^{i\theta}(z_{n+1} - z_n)f(\theta, \phi)/\alpha$$

$$v = z_{n+1} - e^{-i\phi}(z_{n+1} - z_n)f(\phi, \theta)/\beta$$

With function f defined as:

$$f(\theta, \phi) = \frac{2 + \sqrt{2}(\sin\theta - \frac{1}{16}\sin\phi)(\sin\phi - \frac{1}{16}\sin\theta)(\cos\theta - \cos\phi)}{3(1 + \frac{1}{2}(\sqrt{5} - 1)\cos\theta + \frac{1}{2}(3 - \sqrt{5})\cos\phi)}$$

We can now declare the function to perform the conversion with the function that computes f:

Section: Local Function Declaration (metafont.c) (continuation):

```
void convert_to_final(struct path_variable *p);
double compute_f(double theta, double phi);
```

First we sow the code to compute f:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
double compute_f(double theta, double phi){
  double n = 2 + sqrt(2) * (sin(theta) - 0.0625 * sin(phi)) *
             (sin(phi) - 0.0625 * sin(theta)) * (cos(theta) - cos(phi));
  double d = 3 * (1 + 0.5 * (sqrt(5) - 1) * cos(theta) + 0.5 * (3 - sqrt(5)) *
                  cos(phi));
  return n/d;
}
```

If we assume that the path was already normalized to remove the recursive definitions,we can easily iterate over the points converting all them that are in the provisional format:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void convert_to_final(struct path_variable *p){
  int i;
  for(i = 0; i < p -> length - 1; i ++){
    if(p -> points[i].format == PROVISIONAL_FORMAT){
```

```
      bool atleast0 = p -> points[i].prov.atleast1;
      bool atleast1 = p -> points[i].prov.atleast2;
      double complex u, v;
      struct path_points *p0 = &(p -> points[i]), *p1 = &(p -> points[i+1]);
      double w0_x = p0 -> prov.dir1_x, w0_y = p0 -> prov.dir1_y;
      double w1_x = p0 -> prov.dir2_x, w1_y = p0 -> prov.dir2_y;
      double complex z0 = p0 -> prov.x + p0 -> prov.y * I;
      double complex z1 = p1 -> prov.x + p1 -> prov.y * I;
      double theta = carg((w0_x + w0_y * I) / (z1 - z0));
      double phi = carg((z1 - z0)/(w1_x + w1_y * I));
      u = z0 + (cexp(theta * I) * (z1 - z0) * compute_f(theta, phi)) /
        p -> points[i].prov.tension1;
      v = z1 - (cexp(-phi * I) * (z1 - z0) * compute_f(phi, theta)) /
        p -> points[i].prov.tension2;
      p -> points[i].format = FINAL_FORMAT;
      p -> points[i].point.u_x = creal(u);
      p -> points[i].point.u_y = cimag(u);
      p -> points[i].point.v_x = creal(v);
      p -> points[i].point.v_y = cimag(v);
```

<Section to be inserted: Adjust tension>

```
    }
  }
  if(p -> cyclic)
    memcpy(&(p -> points[p -> length - 1]), &(p -> points[0]),
           sizeof(struct path_points));
}
```

One last remaining complication in the conversion is adjusting the tension. If the boolean variable `atleast0` is true, this means that the tension voltage we had was a minimum value. We must increase it if necessary, to the lowest value such that the first control point falls within a bounded triangle by the extremity points and the point at which the straight lines directed by the direction specifiers intersect:

Section: Adjust tension:

```
if(atleast0)
  correct_tension(p0 -> point.x, p0 -> point.y,
                  p1 -> point.x, p1 -> point.y,
                  w0_x, w0_y, w1_x, w1_y,
                  &(p -> points[i].point.u_x), &(p -> points[i].point.u_y));
if(atleast1)
  correct_tension(p0 -> point.x, p0 -> point.y,
                  p1 -> point.x, p1 -> point.y,
                  w0_x, w0_y, w1_x, w1_y,
                  &(p -> points[i].point.v_x), &(p -> points[i].point.v_y));
```

We declare now the function that adjust the tension used above:

Section: Local Function Declaration (metafont.c) (continuation):

```
void correct_tension(double p0_x, double p0_y, double p1_x, double p1_y,
                     double d0_x, double d0_y, double d1_x, double d1_y,
                     float *control_x, float *control_y);
```

And the function implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void correct_tension(double p0_x, double p0_y, double p1_x, double p1_y,
```

```
                    double d0_x, double d0_y, double d1_x, double d1_y,
                    float *control_x, float *control_y){
  double internal_angle0, internal_angle1, internal_angle2;
  double triangle_angle;
  double p2_x, p2_y;
  internal_angle0 = get_angle(p0_x, p0_y, p1_x, p1_y, p0_x + d0_x, p0_y + d0_y);
  internal_angle1 = get_angle(p1_x, p1_y, p0_x, p0_y, p1_x + d1_x, p1_y + d1_y);
  internal_angle2 = M_PI - internal_angle0 - internal_angle1;
  if(internal_angle0 + internal_angle1 >= M_PI - 0.00002 ||
     internal_angle0 == 0.0 || internal_angle1 == 0.0 ||
     internal_angle2 == 0.0)
    return; // Not a valid triangle
  { // Discover the third vertex coordinate:
    // First compute the triangle side that goes from p0 to the unknown vertex
    double known_side = hypot((p1_x - p0_x), (p1_y - p0_y));
    double triangle_side = known_side * sin(internal_angle0) /
                           sin(internal_angle2);
    // Knowing the triangle side and the angle, we compute the coordinates:
    triangle_angle = get_angle(p0_x, p0_y, p1_x, p1_y, p0_x + 1.0, p0_y);
    p2_x = p1_x + triangle_side * cos(triangle_angle + internal_angle1);
    p2_y = p1_y + triangle_side * sin(triangle_angle + internal_angle1);
  }
  {
```

<Section to be inserted: Check if the point is inside the triangle>

<Section to be inserted: If Not, Adjust the Tension>

```
  }
}
```

This is the function that gets the angle between two vectors:

Section: Local Function Declaration (metafont.c) (continuation):

```
double get_angle(double v_x, double v_y, double c0_x, double c0_y,
                 double c1_x, double c1_y);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
double get_angle(double v_x, double v_y, double c0_x, double c0_y,
                 double c1_x, double c1_y){
  double v0_x, v0_y, v1_x, v1_y;
  v0_x = c0_x - v_x;
  v0_y = c0_y - v_y;
  v1_x = c1_x - v_x;
  v1_y = c1_y - v_y;
  if(fabs(v0_x) <= 0.00002 &&  fabs(v0_y) <= 0.00002)
    return INFINITY;
  if(fabs(v1_x) <= 0.00002 &&  fabs(v1_y) <= 0.00002)
    return INFINITY;
  return acos((v0_x * v1_x + v0_y * v1_y) /
              (hypot(v0_x, v0_y) * hypot(v1_x, v1_y)));
}
```

How do we check that a point is inside a triangle? There are several ways, what we will use is a technique that computes the signed area of three triangles composed by the point and two of the vertex points from the triangle, whose vertices we will pass always in the same clockwise or counter-clockwise order. The signed

area of a triangle is its area, but the result is positive or negative, depending if we passed the vertices in clockwise or counter-clockwise order. If all the signed areas are positive, or if all them are negative, then the point is inside the triangle. If not, the point is outside:

Section: Check if the point is inside the triangle:

```
bool s1, s2, s3; // O sinal das reas
s1 = ((*control_x - p1_x) * (p0_y - p1_y) -
      (p0_x - p1_x) * (*control_y - p1_y)) < 0;
s2 = ((*control_x - p2_x) * (p1_y - p2_y) -
      (p1_x - p2_x) * (*control_y - p2_y)) < 0;
s3 = ((*control_x - p0_x) * (p2_y - p0_y) -
      (p2_x - p0_x) * (*control_y - p0_y)) < 0;
if(s1 == s2 && s2 == s3)
  return;
```

An advantage of the above method is that the signed area that have a different sign than others represents the side which is near the target point. Therefore, if we did not exit the function, we can check the closest side, get its line equation and compute the nearest point in the line. If the point is in the triangle side, this is the new point or our control point. Otherwise, we choose the nearest vertex to the point.

Section: If Not, Adjust the Tension:

```
{
  double x0, y0, x1, y1;
  if(s1 != s2 && s1 != s3){
    x0 = p0_x; y0 = p0_y;
    x1 = p1_x; y1 = p1_y;
  }
  else if(s2 != s1 && s2 != s3){
    x0 = p1_x; y0 = p1_y;
    x1 = p2_x; y1 = p2_y;
  }
  else{
    x0 = p2_x; y0 = p2_y;
    x1 = p0_x; y1 = p0_y;
  }
  if(x1 < x0){ // x1 must be >= x0 (assuming this the logic becames simpler)
    double tmp;
    tmp = x1; x1 = x0; x0 = tmp;
    tmp = y1; y1 = y0; y0 = tmp;
  }
  if(x0 == x1){ // Vertical line (do not use line equation, or we divide by zero)
    *control_x = x0;
    if(*control_y > y0 && *control_y > y1){
      if(y0 > y1)
        *control_y = y0;
      else
        *control_y = y1;
    }
    else if(*control_y < y0 && *control_y < y1){
      if(y0 < y1)
        *control_y = y0;
      else
        *control_y = y1;
```

```
    }
  }
  else if(y0 == y1) // Horizontal line
    *control_y = y0;
  else{ // Use line equation
    // m0 x + b0 = y is the line that contains the triangle side
    double m0 = (y1 - y0) / (x1 - x0);
    double b0 = y1 - m0 * x1;
    // m1 x + b1 = y is the perpendicular line that crosses the point
    double m1 = - m0;
    double b1 = *control_y - m1 * *control_x;
    *control_x = (b1 - b0) / (m0 - m1);
    *control_y = m0 * *control_x + b0;
  }
  if(*control_x > x1){
    *control_x = x1;
    *control_y = y1;
  }
  else if(*control_x < x0){
    *control_x = x0;
    *control_y = y0;
  }
}
```

7.4.3. Deducing Direction Specifiers

When we interpret a path from some expression or piece of code, and after we remove the recursion, it will always come with its points either in the final format, or in the first provisional format. If it is in the first provisional format, it will always come with the tension and `atleast` variables correctly filled out. However, it may come without one of the direction specifiers or even without both. Because of this, all path variables store in variable `number_of_missing_directions` how many direction specifiers are missing.

There can be two cases: we can have direction specifiers where the first value is NaN (Not-a-NUmber) and the second is a number, or both can be NaN. If both do not have a numeric value, the direction specifier is missing and unknown. If only the second is a number, then it represents the "curl" of the curve. The higher the value, the more points end act "curling" the curve, as if it were a string curling in a pulley. A larger part of the curve approaches a straight line, with the sudden change of direction being reduced to a smaller area. Anyway, in both cases the specifier is missing, but the rules to compute it changes.

To fill in the missing specifiers, we must divide it into different segments. Every path with missing points has one or more segments in which there is a direction specifier or a "curl" only in the end region of the segment:

$z_0\{w_0\} \,..\, \text{tension}\, a_0 \,\text{and}\, b_0 \,..\, z_1 \,..\, < etc. > \, z_{n-1} \,..\, \text{tension}\, a_{n-1} \,\text{and}\, b_{n-1}\{w_n\}z_n$

This is the function code that fills in the missing specifiers. Notie that what it does is first check if we have a case in which virtually all specifiers are missing. In this case, our segment is the whole path. Otherwise, it iterates over each segment until there are no more missing specifiers:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool find_missing_directions(struct metafont *mf, struct context *cx,
                             struct path_variable *p);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool find_missing_directions(struct metafont *mf, struct context *cx,
                             struct path_variable *p){
  if(p -> number_of_missing_directions == 0 || p -> length < 2)
```

```
    return true;
  if(p -> cyclic && p -> number_of_missing_directions >= (2 * p -> length))
    return fill_cyclic_missing_directions(mf, cx, p, 0, p -> length - 1);
  if(!(p -> cyclic) &&
     p -> number_of_missing_directions >= (2 * (p -> length - 1) - 1))
    return fill_missing_directions(mf, cx, p, 0, p -> length - 2);
  else{
    int begin_segment = 0, end_segment = 0;
    while(p -> number_of_missing_directions > 0){
      while(p -> points[begin_segment].format != PROVISIONAL_FORMAT ||
            isnan(p -> points[begin_segment].prov.dir1_y) ||
            (!isnan(p -> points[begin_segment].prov.dir1_x) &&
             !isnan(p -> points[begin_segment].prov.dir2_x)))
        begin_segment = (begin_segment + 1) % p -> length;
      end_segment = begin_segment;
      while(isnan(p -> points[end_segment].prov.dir2_y))
        end_segment = (end_segment + 1) % p -> length;
      if(!fill_missing_directions(mf, cx, p, begin_segment, end_segment))
        return false;
    }
    return true;
  }
}
```

The missing values are obtained solving systems of linear equations. Each point that lies at the extremity of the segment will contribute with 1 unknown and 1 equation to the system. All the other points will contribute with 3 unknowns and 3 equations. If the segment is an entire cyclic path, there are no points in the extremities and all points contributes with 3 unknowns and 3 equations. This means that on non-cyclic paths of n points, there are $3(n-2)+2$ unknowns, while in cyclic paths of n points we will have $3n$ unknowns.

The unknowns will be θ_0 for the first point of non-cyclic segment, ϕ_{n-1} for the last point, and for the other points with index i, we have the unknowns θ_i, ψ_i and ϕ_i. The order of each unknown in the system of equations is: $(\theta_0, \theta_1, \psi_1, \phi_1, \ldots, \theta_{n-2}, \psi_{n-2}, \phi_{n-2}, \phi_{n-1})$ if we have more than two points in our system. When we have just two points, the order is simply (θ_0, ϕ_1).

When we are placing a new equation in a matrix M, which stores the left side of a linear system, the unknowns referenced by each equation will be the current one and previous θ, the current one and the next ϕ and also the current ψ. No other unknowns will be referenced. Each line in matriz M will have at most 5 non-null elements.

But what is the position of these five elements in the matrix M? Let's declare five variables to keep track of such indices:

Section: Variable Declaration for Unknowns (Non-Cyclic):

```
int previous_theta, current_theta, current_psi, current_phi, next_phi,
    number_of_equations;
previous_theta = -1;
current_theta = 0;
current_psi = current_phi = -1;
if(begin == end)
  next_phi = 1;
else
  next_phi = 3;
number_of_equations = 0;
```

The initialized values above already work in the non-cyclical scenario when we are in the first point and we want to enter the first equation. Remember that the position of the unknowns in each column of M is

given in the order: $(\theta_0, \theta_1, \psi, \phi_1, \ldots)$. So when we are in point 0, there is no previous θ and no current ψ or ϕ. The positions for current θ_0 and ϕ_1 (which is the next ϕ) are correct.

In the cyclic case, for all points we always will have a current θ, ψ and ϕ. Therefore, we initialize the indices as:

Section: Variable Declaration for Unknowns (Cyclic):

```
int previous_theta, current_theta, current_psi, current_phi, next_phi,
    number_of_equations;
previous_theta = size - 3;
current_theta = 0;
current_psi = 1;
current_phi = 2;
next_phi = 5;
number_of_equations = 0;
```

If our system has "`size`" elements, then the following code snippet changes the position of these values to the next equation, but without changing the current point. This occurs when we are neither in the first nor in the last point. In this case, our current point will include three different equations, without changing the referenced unknowns. In this case , we just need to move to the next line of the matrix M.

Section: Next Equation, Same Point:

```
{
  previous_theta += size;
  current_theta += size;
  current_psi += size;
  current_phi += size;
  next_phi += size;
  number_of_equations ++;
}
```

But when we move from one point to another, we do not want to just move to the next equation, but we also need to update the unknowns that will no longer be the same:

Section: Next Equation, Next Point:

```
{
  previous_theta = current_theta + size;
  current_phi = next_phi + size;
  current_theta += (size + 3);
  current_psi += (size + 3);
  if(p -> cyclic)
    next_phi = number_of_equations * size + ((next_phi + 3) % size) + size;
  else{
    if(next_phi % size == size - 2)
      next_phi += (size + 1);
    else
      next_phi += (size + 3);
  }
  number_of_equations ++;
}
```

Notice that we do not need to treat as a special case when we go from first point and equation to the next point. In such cases, the current ψ_0, which does not exist, is already initialized as having an index -1. When we jump to the next variable (from ψ_n to ψ_{n+1}), in most cases we just increment 3 to access the next unknown. So, when we go to the second point, the value is adjusted correctly to index 2, which is actually the position of ψ_1 in the vector of unknowns x.

Taking into account what was presented about the system of equations to be solved in each case, for a

non-cyclical segment, the function that fills the missing directions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool fill_missing_directions(struct metafont *mf, struct context *cx,
                             struct path_variable *p, int begin, int end);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool fill_missing_directions(struct metafont *mf, struct context *cx,
                             struct path_variable *p, int begin, int end){
  double *M, *x, *b;
  struct path_points *p0 = NULL, *p1 = &(p -> points[(begin) % p -> length]),
                     *p2 = &(p -> points[(begin+1) % p -> length]);
  double complex z0 = NAN, z1 = p1 -> prov.x + p1 -> prov.y * I,
                 z2 = p2 -> prov.x + p2 -> prov.y * I;
  int i = 1, size, number_of_points;
  if(end >= begin)
    number_of_points = (end-begin+2);
  else
    number_of_points = p -> length - (begin - end) + 2;
  size = 3 * ((number_of_points) - 2) + 2; // Size of linear system
      <Section to be inserted: Variable Declaration for Unknowns (Non-Cyclic)>
                 <Section to be inserted: Check for degenerate case>
  // Allocating linear system:
  M = (double *) temporary_alloc(size * size * sizeof(double));
  x = (double *) temporary_alloc(size * sizeof(double));
  b = (double *) temporary_alloc(size * sizeof(double));
  if(M == NULL || x == NULL || b == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, 0);
    if(M != NULL && temporary_free != NULL) temporary_free(M);
    if(x != NULL && temporary_free != NULL) temporary_free(x);
    if(b != NULL && temporary_free != NULL) temporary_free(b);
    return false;
  }
  memset(M, 0, size * size * sizeof(double));
  // Iterating over points and adding equations to the system:
                   <Section to be inserted: Equation: First Point>
  if(begin != end){
    for(i = 1; (begin + i - 1) % p -> length != end; i ++){
      p0 = p1; p1 = p2;
      p2 = &(p -> points[(begin+i+1) % p -> length]);
      z0 = z1; z1 = z2;
      z2 = p2 -> prov.x + p2 -> prov.y * I;
                    <Section to be inserted: Equation: General Case>
    }
  }
                    <Section to be inserted: Equation: Last Point>
  solve_linear_system(size, M, b, x);
                  <Section to be inserted: Fill specifier: first point>
  {
    int theta;
    for(i = 1, theta = 1; i < number_of_points - 1; i ++, theta += 3){
```

```
      // Fill missing direction in p->points[i] using x[theta] value
```
<Section to be inserted: Fill specifier: inner point>
```
    }
  }
```
<Section to be inserted: Fill specifier: last point>
```
  if(temporary_free != NULL){
    temporary_free(M);
    temporary_free(x);
    temporary_free(b);
  }
  return true;
}
```

The function that fills the missing directions for some entire cyclic path is quite similar. It just must take into account that all points on the path have directions to be discovered and will also use a different formula to calculate the size for the system of linear equations, since there will be three unknowns per point:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool fill_cyclic_missing_directions(struct metafont *mf, struct context *cx,
                                    struct path_variable *p, int begin,
                                    int end);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool fill_cyclic_missing_directions(struct metafont *mf, struct context *cx,
                                    struct path_variable *p, int begin,
                                    int end){
  double *M, *x, *b;
  struct path_points *p0 = NULL,
                     *p1 = &(p -> points[end]),
                     *p2 = &(p -> points[begin]);
  double complex z0, z1 = p1 -> prov.x + p1 -> prov.y * I,
                 z2 = p2 -> prov.x + p2 -> prov.y * I;
  int i, size = 3 * (p -> length); // System of equations size
```
<Section to be inserted: Variable Declaration for Unknowns (Cyclic)>
```
  // Allocating system of linear equations:
  M = (double *) temporary_alloc(size * size * sizeof(double));
  x = (double *) temporary_alloc(size * sizeof(double));
  b = (double *) temporary_alloc(size * sizeof(double));
  if(M == NULL || x == NULL || b == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, 0);
    if(M != NULL && temporary_free != NULL) temporary_free(M);
    if(x != NULL && temporary_free != NULL) temporary_free(x);
    if(b != NULL && temporary_free != NULL) temporary_free(b);
    return false;
  }
  memset(M, 0, size * size * sizeof(double));
  // Iterating over all points to add equations:
  for(i = 0; i < p -> length; i ++){
    p0 = p1; p1 = p2;
    p2 = &(p -> points[(begin+i+1) % p -> length]);
    z0 = z1; z1 = z2;
    z2 = p2 -> prov.x + p2 -> prov.y * I;
```

<Section to be inserted: Equation: General Case>

```
  }
  solve_linear_system(size, M, b, x);
  {
    int theta;
    for(theta = 0, i = 0; i < p -> length; i ++, theta += 3){
      // Fill missing direction in p->points[i] using x[theta] value
```

<Section to be inserted: Fill specifier: inner point>

```
    }
  }
  if(temporary_free != NULL){
    temporary_free(M);
    temporary_free(x);
    temporary_free(b);
  }
  return true;
}
```

Let's see now which equations we must insert in each case. Let's go to the first point of a non-cyclical segment. By the rules of how path expressions are interpreted, that first segment point, which we can call z_0 and which is succeeded by z_1 (here we treat both as complex numbers), either already has a known direction specifier (which we call w_0) or it has a "curl" value associated, which we call it γ.

If it already has a direction specifier, its value of θ_0 is already known and given by the equation:

$$\theta_0 = arg(w_0/(z_1 - z_0))$$

Instead it has just a γ, then we will later discover θ_0 with the help of the following equation that we add to the system:

$$k_0\theta_0 + k_1\phi_1 = 0$$

With constants k_0 and k_1 depending on the tensions t_1 and t_2 stored in the initial point:

$$k_0 = t_1^2(t_2^{-1} - 3) - \gamma t_2^2 t_1^{-1}$$

$$k_1 = t_1^2 t_2^{-1} - \gamma t_2^2(t_1^{-1} - 3)$$

The code that adds the equations on both cases is:

Section: Equation: First Point:

```
if(!isnan(p1 -> prov.dir1_x)){ // Known w0
  double w1_x = p1 -> prov.dir1_x, w1_y = p1 -> prov.dir1_y;
  M[0] = 1.0; // \theta_0 is the first unknown
  b[0] = carg((w1_x + w1_y * I) / (z2 - z1));
}
else{ // Known gamma
  double gamma = p1 -> prov.dir1_y;
  double t1 = p1 -> prov.tension1;
  double t2 = p1 -> prov.tension2;
  int phi1;
  if(begin == end)
    phi1 = 1;
  else
    phi1 = 3;
  M[0] = t1*t1*(1/t2 - 3.0) - gamma * t2 * t2 * (1/t1);
```

```
  M[phi1] = t1*t1*(1/t2) - gamma * t2 * t2 * (1/t1 - 3.0);
  b[0] = 0.0;
}
{
  previous_theta = size;
  current_theta = size + 1;
  current_psi = size + 2;
  current_phi = size + 3;
  if(number_of_points > 3)
    next_phi = size + 6;
  else
    next_phi = size + 4;
  number_of_equations ++;
}
```

In the case of the last point, we will also have either a known direction w_n or a known "curl" with value γ_n.

If w_n is known, we already know the value for ϕ_n:

$$\phi_n = arg((z_n - z_{n-1})/w_n)$$

Otherwise, if we know γ_n, then the value of ϕ_n will be discovered later using equation:

$$k_0\theta_{n-1} + k_1\phi_n = 0$$

With the constants k_0 anf k_1 depending on tennsions t_1 and t_2 stored in the previous point:

$$k_0 = t_2^2 t_1^{-1} - \gamma t_1^2(t_2^{-1} - 3)$$
$$k_1 = t_2^2(t_1^{-1} - 3) - \gamma t_1^2 t_2^{-1}$$

The code to add the equation in both cases is seen below:

Section: Equation: Last Point:

```
p1 = &(p -> points[(end) % p -> length]);
p2 = &(p -> points[(end+1) % p -> length]);
z1 = p1 -> prov.x + p1 -> prov.y * I;
z2 = p2 -> prov.x + p2 -> prov.y * I;
if(!isnan(p1 -> prov.dir2_x)){ // Known w_n
  double w1_x = p1 -> prov.dir2_x, w1_y = p1 -> prov.dir2_y;
  M[size * size - 1] = 1.0;
  b[size - 1] = carg((z2 - z1)/(w1_x + w1_y * I));
}
else{ // Known gamma
  double gamma = p1 -> prov.dir2_y;
  double t1 = p1 -> prov.tension1;
  double t2 = p1 -> prov.tension2;
  int last_theta;
  if(begin == end)
    last_theta = 2;
  else
    last_theta = size * size - 4;
  M[last_theta] = t2*t2*(1/t1) - gamma * t1 * t1 * (1/t2 - 3.0);
  M[size * size - 1] = t2*t2*(1/t1 - 3.0) - gamma * t1 * t1 * (1/t2);
  b[size - 1] = 0.0;
```

```
}
```

Finally, we will see the three equations that will be generated for the inner points, which are all points in a cyclical segment, or all points in a non-cyclical segment which are not the first nor the last point.

Given point z_n preceded by z_{n-1} and followed by z_{n+1}, we have:

$$\psi_n = arg((z_{n+1} - z_n)/(z_n - z_{n-1}))$$

$$\theta_n + \psi_n + \phi_n = 0$$

$$k_0\theta_{k-1} + k_1\theta_k + k_2\phi_n + k_3\phi_{n+1} = 0$$

If we use the following segment as example:

$$z_0 \; .. \; tension \; t_0 \; and \; t_1 \; .. \; z_1 \; .. \; tension \; t_2 \; and \; t_3 \; .. z_2$$

The constants k_0, k_1, k_2 and k_3 for point z_1 are given by:

$$k_0 = t_1^2 \|z_1 - z_0\|^{-1} t_0^{-1}$$

$$k_1 = -t_2^2 \|z_2 - z_1\|^{-1} (t_3^{-1} - 3)$$

$$k_2 = t_1^2 \|z_1 - z_0\|^{-1} (t_0^{-1} - 3)$$

$$k_3 = -t_2^2 \|z_2 - z_1\|^{-1} (t_3^{-1})$$

The code that adds all three equations is:

Section: Equation: General Case:

```
M[current_psi] = 1.0;
b[number_of_equations] = carg((z2 - z1)/(z1 - z0));
if(b[number_of_equations] == -M_PI)
  b[number_of_equations] *= -1.0;
                <Section to be inserted: Next Equation, Same Point>
M[current_theta] = M[current_psi] = M[current_phi] = 1.0;
b[number_of_equations] = 0.0;
                <Section to be inserted: Next Equation, Same Point>
{
  double t0 = p0 -> prov.tension1, t1 = p0 -> prov.tension2,
         t2 = p1 -> prov.tension1, t3 = p1 -> prov.tension2;
  M[previous_theta] = t1 * t1 * (1.0/cabs(z1 - z0)) * (1.0/t0);
  M[current_theta] = - t2 * t2 * (1.0/cabs(z2 - z1)) * (1.0/t3 - 3.0);
  M[current_phi] = t1 * t1 * (1.0/cabs(z1 - z0)) * (1.0/t0 - 3.0);
  M[next_phi] = - t2 * t2 * (1.0/cabs(z2 - z1)) * (1.0/t3);
  b[number_of_equations] = 0.0;
                <Section to be inserted: Next Equation, Next Point>
}
```

But what exactly do these equations represent and where did they come from? We basically use the same rules and equations than the language METAFONT. This gives us compatibility with curves produced by that language. The methods and the equations were developed by John Hobby and published in [HOBBY, 1986]. What do these rules mean is that the path is invariant under translation, rotation and change of scale. Furthermore, the formula was developed so that any change at a local point has decreasing influence on the following points (the impact of changes drops exponentially) and to minimize spikes and sudden changes in direction of the curve.

The equation that determines the shape of the curve always has a solution, provided all tensions are greater than 3/4 and the "curl" values are always non-negative. These restrictions on values will be reinforced when we interpret of path expressions. If a value outside these limits is found, an error will be generated.

There is, however, one last degenerate case that we can find, where the equation may have no solution. It occurs when a segment has only two points and both have a "curl" instead of a direction specifier. In this case, it either would produce a system of linear equations with no solution, or it produces equations where the solution is a straight line. Because of this, we always will assume that this case produces a straight line:

Section: Check for degenerate case:

```
if(begin == end && isnan(p1 -> prov.dir1_x) && isnan(p1 -> prov.dir2_x)){
  p1 -> format = FINAL_FORMAT;
  p1 -> point.u_x = p1 -> point.x + (1.0/3) * (p2 -> point.x - p1 -> point.x);
  p1 -> point.v_x = p1 -> point.x + (2.0/3) * (p2 -> point.x - p1 -> point.x);
  p1 -> point.u_y = p1 -> point.y + (1.0/3) * (p2 -> point.y - p1 -> point.y);
  p1 -> point.v_y = p1 -> point.y + (2.0/3) * (p2 -> point.y - p1 -> point.y);
  p -> number_of_missing_directions -= 2;
  return true;
}
```

Now let's start filling in the missing values for direction specifiers, assuming we have already solved the linear equations. In the case of the first point of a non-cyclic segment, the equations gave us the value θ_0 associated with it. If we already had a direction specifier, we do not need to do anything. Otherwise, the w_0 specifier is defined as:

$$\theta_0 = arg(w_0/(z_1 - z_0))$$

And this is the code that fills the first direction specifier if needed:

Section: Fill specifier: first point:

```
if(isnan(p -> points[begin].prov.dir1_x)){
  double complex dir;
  z0 = p -> points[begin].prov.x + p -> points[begin].prov.y * I;
  z1 = p -> points[(begin+1) % p -> length].prov.x +
       p -> points[(begin+1) % p -> length].prov.y * I;
  dir = cos(x[0]) + sin(x[0]) * I;
  dir *= (z1 - z0);
  p -> points[begin].prov.dir1_x = creal(dir);
  p -> points[begin].prov.dir1_y = cimag(dir);
  p -> number_of_missing_directions --;
}
```

In the case of the last point, solving the system of equations revealed to us ϕ_n. If it already had a direction specifier, we do not need to do anything. Otherwise, we may deduct your w_n specifier with the formula:

$$\phi_n = arg((z_n - z_{n-1})/w_n)$$

The code that do this, if needed is:

Section: Fill specifier: last point:

```
if(isnan(p -> points[end].prov.dir2_x)){
  double complex dir;
  z1 = p -> points[end].prov.x + p -> points[end].prov.y * I;
  z2 = p -> points[(end+1) % p -> length].prov.x +
       p -> points[(end+1) % p -> length].prov.y * I;
  dir = cos(x[size - 1]) + sin(x[size - 1]) * I;
  dir /= (z2 - z1);
  dir = 1.0 / dir;
  p -> points[end].prov.dir2_x = creal(dir);
```

```
  p -> points[end].prov.dir2_y = cimag(dir);
  p -> number_of_missing_directions --;
}
```

As for all other points with index k, we have both θ_k as ϕ_k; and we could use either one to discover w_k. But the iteration that goes through all the points to fill missing direction specifiers sets values for `i` corresponding to the number of the point that we are visiting in the iteration and a `theta` index with the position in solution vector x with current point θ. Because of this, we chose to use the θ_k formula:

$$\theta_k = arg(w_k/(z_{k+1} - z_k))$$

The internal points will have 2 direction specifiers to be written. One in the right side (stored in the visited point) and another in the left side (stored in the previous point). We fill both using the code below:

Section: Fill specifier: inner point:

```
{
  double complex dir;
  z0 = p -> points[(begin + i) % p -> length].prov.x +
       p -> points[(begin + i) % p -> length].prov.y * I;
  z1 = p -> points[(begin + i + 1) % p -> length].prov.x +
       p -> points[(begin + i + 1) % p -> length].prov.y * I;
  dir = cos(x[theta]) + sin(x[theta]) * I;
  dir *= (z1 - z0);
  p -> points[(begin + i) % p -> length].prov.dir1_x = creal(dir);
  p -> points[(begin + i) % p -> length].prov.dir1_y = cimag(dir);
  p -> number_of_missing_directions --;
  if(begin + i - 1 >= 0){
    p -> points[(begin + i - 1) % p -> length].prov.dir2_x = creal(dir);
    p -> points[(begin + i - 1) % p -> length].prov.dir2_y = cimag(dir);
    p -> number_of_missing_directions --;
  }
  else if(p -> cyclic){
    p -> points[end].prov.dir2_x = creal(dir);
    p -> points[end].prov.dir2_y = cimag(dir);
    p -> number_of_missing_directions --;
  }
}
```

7.4.4. Normalizing Paths

Now that we defined a function that removes path recursion (`recursive_copy_points`, seen in Subsection 7.4.1), another to ensure that all points will have direction specifiers (`find_missing_directions`, seen in Subsection 7.4.3) and a third one that converts all points of a path from provisional format to final format (`convert_to_final`, seen in Subsection 7.4.2), we can combine all these functions in order to normalize a path, leaving it in its final format: the format in which it can be rendered.

Doing this involves calling each of these functions in order. However, there is one last complication that we must address involving cyclic paths.

We could represent a cyclical path in two ways: assuming that the last point is equal the first one, or assuming that there is a path between the last point and the first one. In the first case, the cyclic path below would store 4 points: $(0,0)$, $(0,3)$, $(4,0)$ and $(0,0)$, with the last one being equal the first. In the second case, we would store just the three points and assume the last one point is connected to the first one, since the path is marked as cyclic.

```
(0, 0) .. (0, 3) .. (4, 0) .. cycle;
```

Apparently representing just three points is more elegant: we do not waste memory with an additional point and, as seen in the code from previous subsection, we can iterate over cyclic paths, starting at any point so that the code below would pass exactly once in each point:

```
for(i = 0; i < path -> length; i ++)
  struct path_point *p = &(path -> points[(begin + i) % path -> length]);
```

Notice how in the code above, it doesn't matter what the value is for `begin`, we visit each point exactly once. But only if we work with the assumption that we are not repeating the first point at the end of the cyclic path. In code from previous subsections, we used a loop like this to traverse a path segment, and the code works even when the end of the segment is stored before the beginning, which can happen in segments inside cyclic paths.

However, there are two cases in which there is an advantage in storing explicitly a copy of the first point at the end position. First, the drawing function could deal with cyclic and non-cyclic segments usaing the same code. In both cases, it draws by passing the pen in each point until reaching the last one. Second, and most importantly, consider the code below that concatenates two paths:

```
p = ((-3, -3) -- (0, 0)) & ((0, 0) .. (0, 3) .. (4, 0) .. cycle);
```

Concatenation destroys the cyclic nature of the path. If we do not explicitly store a point $(0, 0)$ at the end of the second path, then we will need to create and place an additional point when the path stops being cyclical. To do this, we may potentially need to reallocate the array where the points are stored in the second path.

To avoid increasing the complexity of concatenation and drawing, we will choose to explicitly represent a copy of the first point at the end of a cyclic path. But for cases where it is better not having this copy, we can then perform the operations:

1) Decrement variable `length` in the path.

2) Perform the operations assuming that a copy of the first point is not placed at the end.

3) Increment variable `length` and copy to the last position a copy of the first point.

This way, the code from the previous Subsection will work and we can get almost the best of both worlds. Just having to make the above conversion. For example, given the function that normalizes a path leaving it in its final format:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool normalize_path(struct metafont *mf, struct context *cx,
                    struct path_variable *path);
```

We can implement it this way:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool normalize_path(struct metafont *mf, struct context *cx,
                    struct path_variable *path){
  struct path_variable *new;
  void *(*alloc)(size_t);
  void (*disalloc)(void *);
  if(path -> permanent){
    alloc = permanent_alloc;
    disalloc = permanent_free;
  }
  else{
    alloc = temporary_alloc;
    disalloc = temporary_free;
  }
  if(!recursive_copy_points(mf, cx, alloc, &new, path, true))
    return false;
  if(disalloc != NULL)
```

```
    disalloc(path -> points);
  memcpy(path, new, sizeof(struct path_variable));
  disalloc(new);
  if(path -> number_of_missing_directions > 0){
    if(path -> cyclic){
      path -> length --;
      if(isnan(path -> points[path -> length - 1].prov.dir1_x))
        path -> number_of_missing_directions --;
      if(isnan(path -> points[path -> length - 1].prov.dir2_x))
        path -> number_of_missing_directions --;
    }
    if(!find_missing_directions(mf, cx, path)){
      printf("DONE\n");
      return false;
    }
    if(path -> cyclic){
      path -> length ++;
      memcpy(&(path -> points[path -> length - 1]), &(path -> points[0]),
             sizeof(struct path_points));
    }
  }
  convert_to_final(path);
  return true;
}
```

7.5. Pen Variables

A pen variable stores the structure needed to draw lines and forms in METAFONT. The pens specify the diameter and format of all drawed lines and points. They are stored in the following structure:

Section: Local Data Structures (metafont.c) (continuation):

```
struct pen_variable{
  int type; // Must be 'TYPE_T_PEN'
  void *next;
  bool permanent; // Variable allocation is permanent or temporary?
  struct path_variable *format; // The format as a cyclic path
  int flags;
  struct pen_variable *referenced; // Not null only in 'currentpen'
  float gl_matrix[9]; // OpenGL matrix transformation
  // The variables below will be manipulated only in Section 11.2
  GLuint gl_vbo; // The OpenGL vertices after triangulation
  GLsizei indices; // Number of vertices stored above
  // This will store how detailed is our triangulation, which will help us
  // to decide if we need a retriangulation in some cases:
  float triang_resolution;
};
```

Here we will give a brief explanation about these variables and why do we need them:

The variables `type`, `nesting_level` are `next` common for all variable types and are not something new. They are needed so that we will know which variable type we have, its scope and which is the next variable if we are in a linked list.

The variable `format` specifies the pen format as a cyclic path. In the original METAFONT, the format needed to be both cyclic and convex. Here, we will require that the path must be cyclic and also should be

simple. A simple path is a path whose perimeter do not crosss over itself. If the restriction is not respected, then the result will be undefined and we will not ensure that the pen will have the specified format.

It does not matter how complex is the pen format, to draw it in the screen, we need to convert it to a set of triangles in a process called triangulation. Details about how to triangulate or if we need to triangulate at all, will be stored in the variable `flags`. Some of the possible flags are:

Section: Local Macros (metafont.c) (continuation):

```
#define FLAG_CONVEX         1
#define FLAG_STRAIGHT       2
#define FLAG_CIRCULAR       4
#define FLAG_SEMICIRCULAR   8
#define FLAG_SQUARE        16
#define FLAG_NULL          32
// More flags will be defined later, Subsection 11.2
```

The flag `FLAG_CONVEX` stores if our pen is convex. If so, we can use very simple triangulation algorithms that will perform the computations in O(n).

The flag `FLAG_STRAIGHT` stores if the pen format is composed only by straight lines. If so, we can represent it exactly by triangulation, the result is perfect, not just approximated. This means that we do not need to redo the triangulation if the pen changes its size, for example. Conversely, if our pen has non-straight curves, then the triangulation needs to be redone to show more details if the pen size is increased.

The flag `FLAG_CIRCULAR` stores if we know that the pen has a circular format. In this case, we can generate the triangulation without looking at the path variable format. Likewise, the flag `FLAG_SEMICIRCULAR` checks if the pen is a semi-circle, which has the same property.

The flag `FLAG_SQUARE` stores if this is a square pen. If so, we also do not need points from the path variable and probably we can use a premade triangulation.

The flag `FLAG_NULL` stores if this is a null pen. If so, it never will need to be triangulated, as it represents a point without height nor width.

The variable `referenced` will be non-NULL if we are dealing with a temporary pen which is not stored in a variable and if this pen references another pen format and vertices. In this case, we will use if possible the same path format and the same triangulation than the referenced pen variable. This also means that if the pen is deallocated, we should not destroy its format nor throw away the triangulation vertices.

The variable `gl_matrix` stores the OpenGL matrix transformation. As the pens will have a more definitive format which will not be subdivided nor concatenated as in the subpath operations, then we can represent their linear transformations using this matrix instead of updating manually their points.

The other variables will me used more on Section 11.2. The variable `gl_vbo` will reference the pen vertices after the triangulation, when the pen is ready to be drawn. Such vertices, if they exist, will be stored in the video card. Otherwise, this variable will be equal 0. The variables `pen_lft`, `pen_rt`, `pen_top` and `pen_bot` will store respectively the maximum coordinate x, minimum coordinate x, maximum coordinate y and minimum coordinate y for all the points in the pen. Finally, `triang_resolution` will store an internal measure of how detailed is the pen triangulation. It will help us to choose if we need a more detailed triangulation or if what we have is enough.

A pen variable that was declared but not initialized will have its format set as the null pointer, will be non-circular and not triangulated:

Section: Initializing New Variable (continuation):

```
if(type == TYPE_T_PEN){
  ((struct pen_variable *) var) -> format = NULL;
  ((struct pen_variable *) var) -> gl_vbo = 0;
  ((struct pen_variable *) var) -> indices = 0;
  ((struct pen_variable *) var) -> flags = false;
  ((struct pen_variable *) var) -> referenced = NULL;
  ((struct pen_variable *) var) -> permanent = mf -> loading;
```

```
}
```

Which means that when we remove a global variable and it is a pen variable, we need to deallocate the format and throw away its vertices:

Section: Finalize Variable 'v' in 'struct metafont' (continuation):

```
if(v -> type == TYPE_T_PEN){
  struct pen_variable *pen = (struct pen_variable *) v;
  if(pen -> format != NULL && permanent_free != NULL)
    path_recursive_free(permanent_free, pen -> format, true);
  if(pen -> gl_vbo != 0)
    glDeleteBuffers(1, &(pen -> gl_vbo));
}
```

If the variable is not global we do the same, but using a different disallocation function:

Section: Finalize Variable 'v' in 'struct context':

```
if(v -> type == TYPE_T_PEN){
  struct pen_variable *pen = (struct pen_variable *) v;
  if(pen -> format != NULL && temporary_free != NULL)
    path_recursive_free(temporary_free, pen -> format, true);
  if(pen -> gl_vbo != 0)
    glDeleteBuffers(1, &(pen -> gl_vbo));
                <Section to be inserted: Finalize Local Pen 'pen'>
}
```

Besides the pen variables declared by users, we will also support two other internal pen variables that can be used by the users that will always be declared and initialized. One of them will be `currentpen` variable that always will store the pen that we currently are using to draw. The other variable will be a pen with square format that will be already initialized to obtain better performance when used.

The `currentpen` variable will be special: it will be like a global variable. But its state between different executions of WeaveFont code will not be saved: each time we execute a certain code again, it is as if this variable had been reset. In addition, we will store information about it that we do not store for the other pens: its largest and smallest x coordinates, as well as its largest and smallest y coordinates. This variable will not be stored together with the others: it will not be stored in the `metafont` structure that represents the font, but in the structure that represents our execution context:

Section: Attributes (struct context) (continuation):

```
struct pen_variable *currentpen;
```

The variable with our square pen, which will be called `pensquare` will be considered a typical internal variable:

Section: Attributes (struct metafont) (continuation):

```
struct pen_variable *internal_pen_variables;
```

The variable `currentpen` is initialized as an empty variable:

Section: Initialization (struct context) (continuation):

```
cx -> currentpen = (struct pen_variable *)
                     permanent_alloc(sizeof(struct pen_variable));
if(cx -> currentpen == NULL){
  RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
  return NULL;
}
cx -> currentpen -> format = NULL;
cx -> currentpen -> type = TYPE_T_PEN;
cx -> currentpen -> flags = FLAG_NULL;
```

```
cx -> currentpen -> referenced = NULL;
cx -> currentpen -> gl_vbo = 0;
cx -> currentpen -> indices = 0;
cx -> currentpen -> permanent = true;
INITIALIZE_IDENTITY_MATRIX(cx -> currentpen -> gl_matrix);
```

The internal variable with our square pen need to be allocated and initialized to be a square with side length 1:

Section: Initialization (struct metafont) (continuation):

```
mf -> internal_pen_variables = (struct pen_variable *)
                                permanent_alloc(1 * sizeof(struct pen_variable));
if(mf -> internal_pen_variables == NULL){
  if(permanent_free != NULL)
    permanent_free(mf);
  return NULL;
}
mf -> internal_pen_variables[0].format = NULL; // A caneta 'pensquare'
mf -> internal_pen_variables[0].type = TYPE_T_PEN;
mf -> internal_pen_variables[0].flags = FLAG_CONVEX | FLAG_STRAIGHT |
                                        FLAG_SQUARE;
mf -> internal_pen_variables[0].referenced = NULL;
mf -> internal_pen_variables[0].gl_vbo = 0;
mf -> internal_pen_variables[0].indices = 4;
mf -> internal_pen_variables[0].permanent = true;
INITIALIZE_IDENTITY_MATRIX(mf -> internal_pen_variables[0].gl_matrix);
```

The following macro will help us to access more easily the position for the internal variable:

Section: Local Macros (metafont.c) (continuation):

```
#define INTERNAL_PEN_PENSQUARE  0
```

And to finalize the Metafont structure, we need to deallocate the internal variables:

Section: Finalization (struct metafont) (continuation):

```
if(permanent_free != NULL){
  if(mf -> internal_pen_variables[0].format != NULL){
    permanent_free(mf -> internal_pen_variables[0].format -> points);
    permanent_free(mf -> internal_pen_variables[0].format);
  }
  permanent_free(mf -> internal_pen_variables);
}
```

And we also need to free the memory for `currentpen` after finishing the execution context:

Section: Finalization (struct context) (continuation):

```
if(temporary_free != NULL){
  if(cx -> currentpen -> format != NULL){
    temporary_free(cx -> currentpen -> format -> points);
    temporary_free(cx -> currentpen -> format);
  }
  temporary_free(cx -> currentpen);
}
```

And when we read a source code, we need to adjust correctly the variable pointers when we find reference for internal variables (we do not do this with`currenpen` because this variable will be dealt with

very differently):

Section: Set Pointer to Internal Variable (continuation):

```
else if(!strcmp(buffer, "pensquare"))
  new_token -> var =
          &(mf -> internal_pen_variables[INTERNAL_PEN_PENSQUARE]);
```

The pen '`currentpen`' is special because it can store a pointer for other pen instead of its own content. But what happens if we free the memory of a local pen variable that was pointed by `currentepen`? Easy. In this case, `currentpen` bacomes a null pen again, restoring its default values:

Section: Finalize Local Pen 'pen':

```
if(cx -> currentpen -> referenced == pen){
  cx -> currentpen -> format = NULL;
  cx -> currentpen -> type = TYPE_T_PEN;
  cx -> currentpen -> flags = FLAG_NULL;
  cx -> currentpen -> referenced = NULL;
  cx -> currentpen -> gl_vbo = 0;
  cx -> currentpen -> indices = 0;
}
```

7.6. Picture Variables

A picture variable will store a rendered image, possibly created using pens and Weaver Metafont drawing commands. Contrary to original METAFONT, Weaver Metafont requires all pictures to have a known width and height. Therefore, we need to store in this variable its size and also an OpenGL identifier to a texture:

Section: Local Data Structures (metafont.c) (continuation):

```
struct picture_variable{
  int type; // Must be 'TYPE_T_PICTURE'
  void *next;
  int width, height;
  GLuint texture;
};
```

A declared, but not initialized picture variable will have negative height and width. Its texture also will be zero:

Section: Initializing New Variable (continuation):

```
if(type == TYPE_T_PICTURE){
  ((struct picture_variable *) var) -> width = -1;
  ((struct picture_variable *) var) -> height = -1;
  ((struct picture_variable *) var) -> texture = 0;
}
```

When removing a global variable, if it is a picture variable, we need to destroy its texture with OpenGL if the texture exists:

Section: Finalize Variable 'v' in 'struct metafont' (continuation):

```
if(v -> type == TYPE_T_PICTURE){
  struct picture_variable *pic = (struct picture_variable *) v;
  if(pic -> texture != 0)
    glDeleteTextures(1, &(pic -> texture));
}
```

If the variable is not global we do the same:

Section: Finalize Variable 'v' in 'struct context':

```
if(v -> type == TYPE_T_PICTURE){
  struct picture_variable *pic = (struct picture_variable *) v;
  if(pic -> texture != 0)
    glDeleteTextures(1, &(pic -> texture));
}
```

We will have a single internal picture variable, which we will call "`currentpicture`". Like in the case of `currentpen`, this variable will be reinitialized as empty, each time we render a new character, each time we enter a new execution context:

Section: Attributes (struct context) (continuation):

```
struct picture_variable *currentpicture;
```

In the initialization for each execution, we let the variable as non-initialized:

Section: Initialization (struct context) (continuation):

```
cx -> currentpicture = (struct picture_variable *)
                         temporary_alloc(sizeof(struct picture_variable));
if(cx -> currentpicture == NULL){
  RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
  return NULL;
}
```

```
cx -> currentpicture -> type = TYPE_T_PICTURE;
cx -> currentpicture -> width = -1;
cx -> currentpicture -> height = -1;
cx -> currentpicture -> texture = 0;
```

When the time to deallocate intrnal variables comes, we should eliminate its texture if it exists:

Section: Finalization (struct context) (continuation):

```
if(cx -> currentpicture -> texture != 0)
  glDeleteTextures(1, &(cx -> currentpicture -> texture));
if(temporary_free != NULL)
  temporary_free(cx -> currentpicture);
```

7.7. Boolean Variables

The simplest variable type, a boolean variable can store only true or false. Its struct format is:

Section: Local Data Structures (metafont.c) (continuation):

```
struct boolean_variable{
  int type; // Deve ser 'TYPE_T_BOOLEAN'
  void *next;
  short value;
};
```

The value stored by the variable will be 0 if false, 1 if true or -1 if non-initialized:

Section: Initializing New Variable (continuation):

```
if(type == TYPE_T_BOOLEAN)
  ((struct boolean_variable *) var) -> value = -1;
```

8. Assignments

Assignments are how we initialize variables, how we modify their values and how we store the result of expressions.

The syntax for an assignment is:

```
<Simple Statement> -> <Declaration> | <Assignment> | ...
<Assignment> -> <Variable> = <Right Side> |
                <Variable> := <Right Side> |
<Right Side> -> <Expression> | <Assignment>
```

This means that we can chain assignments, for example, in the code below, all variables, if numeric, will store the value 5:

```
a = b = c = 5;
```

If in the beginning of a statement we find a variable, then certainly we are dealing with an assignment.

In an assignment statement, the tokens "=" and ":=" are equivalent:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_EQUAL,             // Symbolic token '='
TYPE_ASSIGNMENT,        // Symbolic token ':='
```

Both types are added to the list of keywords:

Section: List of Keywords (continuation):

```
"=", ":=",
```

Now that we have this new kind of token, the code to evaluate assignments is described below. What it does is walk over the tokens in the assignment detecting all variables that should pass by the assignment. While it does it, it checks if all them are declared, if all them have the same type, if we are assigning to something that is not a variable, if we forgot the assignment symbol, or if we are missing an expression after the assignment. If one of these things are detected, an error is generated.

The part that we still will not show is how we evaluate the expression after the last assignment symbol and how in the end we update the value for each variable to be equal what was evaluated. How we do these things change depending on the variable types, each type have its own expressions and its own way of storing the value in the variables. These details will be shown in the next subsections.

Section: Statement: Assignment:

```
else if(begin -> type == TYPE_SYMBOLIC){
  struct symbolic_token *var = (struct symbolic_token *) begin;
  struct generic_token *begin_expression;
  int type = 0; // Type for the variables being assigned
  int number_of_variables = 0;
  do{
    if(var -> type != TYPE_SYMBOLIC){
      RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(var -> line), TYPE_SYMBOLIC,
                                 (struct generic_token *) var);
      return false;
    }
    // Variables like 'h', 'd', 'w', 'currentpen' and
    // 'currentpicture' are special:
    if(!strcmp(var -> value, "h"))
      var -> var =
        (void *) &(cx -> internal_numeric_variables[INTERNAL_NUMERIC_H]);
    else if(!strcmp(var -> value, "w"))
      var -> var =
        (void *) &(cx -> internal_numeric_variables[INTERNAL_NUMERIC_W]);
    else if(!strcmp(var -> value, "d"))
      var -> var =
        (void *) &(cx -> internal_numeric_variables[INTERNAL_NUMERIC_D]);
```

```
    else if(!strcmp(var -> value, "currentpen"))
      var -> var = (void *) cx -> currentpen;
    else if(!strcmp(var -> value, "currentpicture"))
      var -> var = (void *) cx -> currentpicture;
    if(var -> var == NULL){
      RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(var -> line), var);
      return false;
    }
    number_of_variables ++;
    if(type == 0)
      type = ((struct variable *) (var -> var)) -> type;
    else if(((struct variable *) (var -> var)) -> type != type){
      RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(var -> line),
                                      var,
                                      ((struct variable *)(var -> var)) -> type,
                                      type);
      return false;
    }
    if((void *) var != (struct symbolic_token *) end)
      var = (struct symbolic_token *) (var -> next);
    else
      var = NULL;
    if(var -> type != TYPE_EQUAL && var -> type != TYPE_ASSIGNMENT){
      RAISE_ERROR_UNKNOWN_STATEMENT(mf, cx, OPTIONAL(begin -> line));
      return false;
    }
    if(var != (struct symbolic_token *) *end)
      var = (struct symbolic_token *) (var -> next);
    else
      var = NULL;
  } while(var != NULL && (var -> next -> type == TYPE_EQUAL ||
                          var -> next -> type == TYPE_ASSIGNMENT));
  if(var == NULL){
    RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin -> line), type);
    return false;
  }
  begin_expression = (struct generic_token *) var;
              <Section to be inserted: Assignment for Numeric Variables>
                <Section to be inserted: Assignment for Pair Variables>
              <Section to be inserted: Assignment for Transform Variables>
                <Section to be inserted: Assignment for Path Variables>
                <Section to be inserted: Assignment for Pen Variables>
               <Section to be inserted: Assignment for Picture Variables>
               <Section to be inserted: Assignment for Boolean Variables>
  return true;
}
```

The above assignment code will be executed every time that we find a variable name, a "tag" as the first token in a statement. In a valid program, this happens only when the user is about to perform an assignment.

8.1. Numeric Assignments and Expressions

How do we assign numeric variables once we checked that we had no errors on the left side of the assignment? We use the following code:

Section: Assignment for Numeric Variables:

```
if(type == TYPE_T_NUMERIC){
  int i;
  struct numeric_variable result;
  // Get right-side expression value:
  if(!eval_numeric_expression(mf, cx, begin_expression, *end, &result))
    return false;
  var = (struct symbolic_token *) begin;
  for(i = 0; i < number_of_variables; i ++){
    // The assignment:
    ((struct numeric_variable *) (var -> var)) -> value = result.value;
    // Getting next variable to assign:
    var = (struct symbolic_token *) (var -> next);
    var = (struct symbolic_token *) (var -> next);
  }
}
```

Now we will write how to interpret numeric expressions.

8.1.1. Sum and Subtraction: Normal and Pythagorean

The rules for numeric expressions begins as:

```
<Numeric Expression> -> <Numeric Tertiary>
<Numeric Tertiary> -> <Numeric Secondary> |
                      <Numeric Tertiary> <T-Op> <Numeric Secondary>
<T-Op> -> + | - | ++ | +-+
```

The symbols `+` and `-` represent the usual addition and subtraction. The symbol `++` should not be confused with C language increment and means the pythagorean sum:

$$a + +b = \sqrt{a^2 + b^2}$$

This can be easily computed in C using the function `hypot` from the standard math library.

The symbol `+-+` is the "pythagorean subtraction" defined below:

$$a + - + b = \sqrt{a^2 - b^2} = \sqrt{(a+b)(a-b)} = \sqrt{a+b}\sqrt{a-b}$$

We will compute the pythagorean subtraction using the last definition, as the multiplication of two square roots. This is the computation that minimizes errors involving overflows and underflows.

The four operators above are the ones with smaller precedence. These operations will be done only after all other mathematical operations.

We define below the tokens for the operators:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_SUM,                  // 0 token simblico '+'
TYPE_SUBTRACT,             // 0 token simblico '-'
TYPE_PYTHAGOREAN_SUM,      // 0 token simblico '++'
TYPE_PYTHAGOREAN_SUBTRACT, // 0 token simblico '+-+'
```

However, to correctly identify the tertiary operators insine some expression requires taking into account delimiters like "[", "]", "{" e "}", besides parenthesis, whose tokens were already defined when we introduced the lexer. Because of this, we add new token types for the delimiters that still were not defined:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_OPEN_BRACKETS,          // 0 token simblico '['
TYPE_CLOSE_BRACKETS,         // 0 token simblico ']'
TYPE_OPEN_BRACES,            // 0 token simblico '{'
TYPE_CLOSE_BRACES,           // 0 token simblico '}'
```

And we add all them to the list of keywords:

Section: List of Keywords (continuation):

```
"+", "-", "++", "+-+", "[", "]", "{", "}",
```

Now let's evaluate numeric expressions, which means evaluating tertiary numeric expressions. The function is declared here:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_numeric_expression(struct metafont *mf, struct context *cx,
                             struct generic_token *begin_expression,
                             struct generic_token *end_token_list,
                             struct numeric_variable *result);
```

The tertiary numeric expressions is where we solve additions and subtractions. By the syntax rules, what we need to do is walk over the tokens in the expression until find the last tertiary operator that is not delimited by parenthesis, brackets or braces.

To help with this, the following macros will store, check and count the nesting level for parenthesis, braces and brackets:

Section: Local Macros (metafont.c) (continuation):

```
#define DECLARE_NESTING_CONTROL() int nesting_parenthesis = 0, \
                                      nesting_brackets = 0, \
                                      nesting_braces = 0;
#define COUNT_NESTING(p) {if(p -> type == TYPE_OPEN_PARENTHESIS)           \
                            nesting_parenthesis ++;                        \
                          else if(p -> type == TYPE_CLOSE_PARENTHESIS) \
                            nesting_parenthesis --;                        \
                          else if(p -> type == TYPE_OPEN_BRACKETS)     \
                            nesting_brackets ++;                           \
                          else if(p -> type == TYPE_CLOSE_BRACKETS)    \
                            nesting_brackets --;                           \
                          else if(p -> type == TYPE_OPEN_BRACES)       \
                            nesting_braces ++;                             \
                          else if(p -> type == TYPE_CLOSE_BRACES)      \
                            nesting_braces --;}
#define IS_NOT_NESTED() (nesting_parenthesis == 0 && nesting_brackets == 0 && \
                         nesting_braces == 0)
#define RESET_NESTING_COUNT() nesting_parenthesis = 0; \
                              nesting_brackets = 0; \
                              nesting_braces = 0;
```

After walking over an expression, we may want to raise an error if we opened a parenthesis, a bracket or braces which was not closed. For this, the following macro may be useful:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, line) {\
  if(nesting_parenthesis > 0){\
    RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, line, '(');\
    return false;\
  } else if(nesting_parenthesis < 0){\
```

```
    RAISE_ERROR_UNOPENED_DELIMITER(mf, cx, line, ')');\
    return false;\
  } else if(nesting_brackets > 0){\
    RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, line, '[');\
    return false;\
  } else if(nesting_brackets < 0){\
    RAISE_ERROR_UNOPENED_DELIMITER(mf, cx, line, ']');\
    return false;\
  } else if(nesting_braces > 0){\
    RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, line, '{');\
    return false;\
  } else if(nesting_braces < 0){\
    RAISE_ERROR_UNOPENED_DELIMITER(mf, cx, line, '}');\
    return false;}}
```

The above error checking do not detect all possible errors involving delimiters. For example, we may get a "({)}", which is wrong, but we cannot detect. We will worry about this later. For now, we are caring about delimiters only because we need to check for operators outside them to correctly find the operators in tertiary expressions, and we will worry only about errors that would prevent us to detect this.

In a tertiary expression, everything in the left side of the rightmost tertiary operator should be recursively evaluated as another tertiary expression. Everything in the right side should be interpreted as a secondary numeric expression. Finally, if we did not find any tertiary operator, the entire expression is evaluated as a secondary numeric expression.

However, there are some exceptions, cases in which the symbols `+` and `-` should not be treated as a sum or subtraction. For example:

```
a = +1;
```

In this case, it is just an unary operator that does not change the signal for the next element. If we had a `-`, then it would invert the signal of the next element. There is no sum or subtraction occurring.

This happens if the operators are in the beginning of the numeric expression, or if before them there is a comma, an opening bracket, a multiplication or division symbol, another tertiary operator, or one of the following tokens that we will define later: `length`, `sqrt`, `sind`, `cosd`, `log`, `exp`, `floor`, `uniformdeviate`, `rotated`, `shifted`, `slanted`, `xscaled`, `yscaled`, `zscaled`, `of`, `point`, `precontrol` ou `postcontrol`.

If we have a previous token (`prev`) and the current one (`cur`), we can check if it represents a valid sum or subtraction with the macro below:

Section: Local Macros (metafont.c) (continuation):

```
#define IS_VALID_SUM_OR_SUB(prev, cur)                    \
        ((cur -> type == TYPE_SUM ||                     \
         cur -> type == TYPE_SUBTRACT) &&                \
         (prev != NULL && prev -> type != TYPE_COMMA && \
          prev -> type != TYPE_OPEN_BRACKETS &&          \
          prev -> type != TYPE_MULTIPLICATION &&         \
          prev -> type != TYPE_DIVISION &&               \
          prev -> type != TYPE_SUM &&                    \
          prev -> type != TYPE_SUBTRACT &&               \
          prev -> type != TYPE_PYTHAGOREAN_SUM &&        \
          prev -> type != TYPE_PYTHAGOREAN_SUBTRACT &&   \
          prev -> type != TYPE_LENGTH &&                 \
          prev -> type != TYPE_SQRT &&                   \
          prev -> type != TYPE_SIND &&                   \
          prev -> type != TYPE_COSD &&                   \
```

```
          prev -> type != TYPE_LOG &&                    \
          prev -> type != TYPE_EXP &&                    \
          prev -> type != TYPE_FLOOR &&                  \
          prev -> type != TYPE_ROTATED &&                \
          prev -> type != TYPE_SCALED &&                 \
          prev -> type != TYPE_SHIFTED &&                \
          prev -> type != TYPE_SLANTED &&                \
          prev -> type != TYPE_XSCALED &&                \
          prev -> type != TYPE_YSCALED &&                \
          prev -> type != TYPE_ZSCALED &&                \
          prev -> type != TYPE_OF &&                     \
          prev -> type != TYPE_POINT &&                  \
          prev -> type != TYPE_PRECONTROL &&             \
          prev -> type != TYPE_POSTCONTROL &&            \
          prev -> type != TYPE_UNIFORMDEVIATE))
```

The following code interprets tertiary numeric expressions and identify correctly the tertiary operators with the help of the macro:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_numeric_expression(struct metafont *mf, struct context *cx,
                             struct generic_token *begin,
                             struct generic_token *end,
                             struct numeric_variable *result){
  struct generic_token *end_tertiary = NULL, *begin_secondary,
                       *last_sum = NULL, *p, *prev = NULL;
  DECLARE_NESTING_CONTROL();
  struct numeric_variable a, b;
  p = begin;
  do{ // Find last tertiary operator: '+', '-', '++' or '+-+'
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && (p -> type == TYPE_PYTHAGOREAN_SUM ||
                           p -> type == TYPE_PYTHAGOREAN_SUBTRACT ||
                           IS_VALID_SUM_OR_SUB(prev, p))){
      last_sum = p;
      end_tertiary = prev;
    }
    prev = p;
    if(p != end)
      p = (struct generic_token *) p -> next;
    else
      p = NULL;
  }while(p != NULL);
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
  if(end_tertiary != NULL){ // If we have tertiary operator:
    begin_secondary = (struct generic_token *) (last_sum -> next);
    eval_numeric_expression(mf, cx, begin, end_tertiary, &a);
    eval_numeric_secondary(mf, cx, begin_secondary, end, &b);
    if(last_sum -> type == TYPE_SUM) // Evaluate '+':
      result -> value = a.value + b.value;
    else if(last_sum -> type == TYPE_SUBTRACT) // Evaluate '-':
      result -> value = a.value - b.value;
```

```
    else if(last_sum -> type == TYPE_PYTHAGOREAN_SUM) // Evaluate '++'
      result -> value = hypotf(a.value, b.value);
    else if(last_sum -> type == TYPE_PYTHAGOREAN_SUBTRACT){ // Evaluate '+-+':
      result -> value = sqrtf(a.value + b.value) *
        sqrtf(a.value - b.value);
      if(isnan(result -> value)){
        RAISE_ERROR_NEGATIVE_SQUARE_ROOT(mf, cx, OPTIONAL(last_sum -> line),
                                         a.value - b.value);
        return false;
      }
    }
    return true;
  }
  else // No tertiary operator:
    return eval_numeric_secondary(mf, cx, begin, end, result);
}
```

8.1.2. Multiplication and Division

The rules to deal with secondary numeric expressions are:

```
<Numeric Secondary> -> <Numeric Primary> |
                       <Numeric Secondary> <S-Op> <Numeric Primary>
<S-Op> -> * | /
```

The operators `*` and `/` are respectively the multiplication and division.

Let's add these operators as reserved tokens and define their types:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_MULTIPLICATION,        // Symbolic token '*'
TYPE_DIVISION,              // Symbolic token '/'
```

Section: List of Keywords (continuation):

```
"*", "/",
```

The function that will evaluate numeric secondary expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_numeric_secondary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
                            struct numeric_variable *result);
```

And its definition is very similar with the function that evaluates tertiary expressions, except that this function computes multiplication and division.

However, there is an additional rule that we should be aware: the token "/" will be considered division only if it is not delimited by two numeric tokens. If so, then it represents a fraction and it should be computed with a higher precedence than what we deal here. However, if the token before the previous one already was considered part of a fraction, then in this case we have a division despide being surrounded by numeric tokens. This way the code `1/3/1/3` is interpreted as a division between two fractions (1/3)/(1/3):

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_numeric_secondary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
```

```
                              struct numeric_variable *result){
  struct generic_token *end_secondary = NULL, *begin_primary,
                       *last_mul = NULL, *p, *prev = NULL,
                       *prev_prev = NULL, *last_fraction = NULL;
  DECLARE_NESTING_CONTROL();
  struct numeric_variable a, b;
  b.value = 0.0;
  p = begin;
  do{ // Find last secondary operator '*' or '/'
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && (p -> type == TYPE_MULTIPLICATION ||
                           p -> type == TYPE_DIVISION)){
      if(p -> type == TYPE_DIVISION && prev -> type == TYPE_NUMERIC &&
         p != end &&  p -> next -> type != TYPE_NUMERIC &&
         last_fraction != prev_prev) // Fraction separator
         last_fraction = p;
       else{ // Valid multiplication or division
         last_mul = p;
         end_secondary = prev;
       }
    }
    prev_prev = prev;
    prev = p;
    if(p != end)
      p = (struct generic_token *) p -> next;
    else
      p = NULL;
  }while(p != NULL);
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
  if(end_secondary != NULL){
    begin_primary = (struct generic_token *) (last_mul -> next);
    if(!eval_numeric_secondary(mf, cx, begin, end_secondary, &a))
      return false;
    if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
      return false;
    if(last_mul -> type == TYPE_MULTIPLICATION) // Evaluate '*':
      result -> value = a.value * b.value;
    else if(last_mul -> type == TYPE_DIVISION){ // Evaluate '/':
      if(b.value == 0.0){
        RAISE_ERROR_DIVISION_BY_ZERO(mf, cx, OPTIONAL(last_mul -> line));
        return false;
      }
      result -> value = a.value / b.value;
    }
    return true;
  }
  else // No secondary operator:
    return eval_numeric_primary(mf, cx, begin, end, result);
}
```

In the avove code, we check again for errors in our delimiters, despite the fact that we also checked when evaluating the tertiary expression. This is needed because, for example, the following incorrect code

would not raise any error during tertiary expression evaluation, but would produce the error only in the above code, when evaluating the secondary expression:

```
numeric a;
a=(4*{8)+3}+1;
```

8.1.3. Modulus, Sine, Cosine, Exponentials, Floor and Random Uniform Values

The rules for primary numeric expressions are:

```
<Numeric Primary> -> <Numeric Atom> |
                     length <Numeric Primary> | (...) |
                     <Numeric Operator> <Numeric Primary>
<Numeric Operator> -> sqrt | sind | cosd | mlog | mexp | floor |
                       uniformdeviate |
                       <Scalar Multiplication Operator>
<Scalar Multiplication Operator> -> + | - |
                       <Primary Numeric Token Before Variable>
<Primary Numeric Token> -> <Numeric Token> / <Numeric Token> |
                           <Numeric Token>
```

The operator `length`, when appearing before a numeric primery means that we want its modulus. If it appears before another kind of primary expression, it could mean different things.

All these new operators must be added to the token list:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_LENGTH,         // Symbolic token 'length'
TYPE_SQRT,           // Symbolic token 'sqrt'
TYPE_SIND,           // Symbolic token 'sind'
TYPE_COSD,           // Symbolic token 'cosd'
TYPE_LOG,            // Symbolic token 'log'
TYPE_EXP,            // Symbolic token 'exp'
TYPE_FLOOR,          // Symbolic token 'floor'
TYPE_UNIFORMDEVIATE, // Symbolic token 'uniformdeviate'
```

Section: List of Keywords (continuation):

```
"length", "sqrt", "sind", "cosd", "log", "exp", "floor", "uniformdeviate",
```

The function that evaluates a primary numeric expression is this:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_numeric_primary(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct numeric_variable *result);
```

We can decide which syntax rule should be applied while evaluating the primary numeric expression using seven different interpretation rules:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_numeric_primary(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct numeric_variable *result){
                  <Section to be inserted: Numeric Primary: Rule 1>
                  <Section to be inserted: Numeric Primary: Rule 2>
```

```
                <Section to be inserted: Numeric Primary: Rule 3>
            <Section to be inserted: Numeric Primary: Additional Operators>
                <Section to be inserted: Numeric Primary: Rule 4>
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_NUMERIC);
  return false;
}
```

The rules to recognize the expression are:

1) If the expression is composed by a single token, or if it begins with "(" and ends with ")", or if it is composed by three tokens, a numeric, a division and another numeric, then the entire expression is a numeric atom:

Section: Numeric Primary: Rule 1:

```
if(begin == end || (begin -> type == TYPE_OPEN_PARENTHESIS &&
                    end -> type == TYPE_CLOSE_PARENTHESIS) ||
    (begin -> type == TYPE_NUMERIC &&  begin -> next != end &&
     begin -> next -> type == TYPE_DIVISION && begin -> next -> next == end &&
     end -> type == TYPE_NUMERIC)){
  return eval_numeric_atom(mf, cx, begin, end, result);
}
```

2) If we find operator `length`, then we check if the expression after it is numeric. If so, we compute its modulus. If the expression have another type, we will define later how the operator will proceed. We assume that we have a function that discovers the type of a primary expression.

Section: Numeric Primary: Rule 2:

```
else if(begin -> type == TYPE_LENGTH){
  int expr_type = get_primary_expression_type(mf, cx, begin -> next, end);
  if(expr_type == TYPE_T_NUMERIC){
    struct numeric_variable num;
    if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
      return false;
    result -> value = ((num.value > 0)?(num.value):(-num.value));
    return true;
  }
                    <Section to be inserted: Evaluate 'length'>
  else{
    RAISE_ERROR_UNSUPORTED_LENGTH_OPERAND(mf, cx, OPTIONAL(begin -> line),
                                          expr_type);
    return false;
  }
}
```

Notice that identifying if we have or not a numeric primary expression involves calling `get_primatry_expression` This function will be defined in Subsection 8.8. For now, we just need to know that given the beginning and ending tokens for some expression, it returns the expression type. If the type is numeric, the code above shows how we compute the modulus. Otherwise, we will evaluate it using other rules that we should define later (in Subsection 8.2.4 and 8.4.5).

3) If we find some of the numeric operators, we have a numeric operator followed by a numeric primary expression.

The first numeric operator is the square root:

Section: Numeric Primary: Rule 3:

```
else if(begin -> type == TYPE_SQRT){
  struct numeric_variable num;
```

```
  if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
    return false;
  if(num.value < 0.0){
    RAISE_ERROR_NEGATIVE_SQUARE_ROOT(mf, cx, OPTIONAL(begin -> line),
                                     num.value);
    return false;
  }
  result -> value = sqrtf(num.value);
  return true;
}
```

Next we have the operator `sind`, which interprets the next number in degrees (because of this it has the letter "d" in the end) and computes the sine:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_SIND){
  struct numeric_variable num;
  if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
    return false;
  // 1 degree = 0,0174533 radians
  result -> value = sinf(num.value * 0.0174533);
  return true;
}
```

Computing the cosine:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_COSD){
  struct numeric_variable num;
  if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
    return false;
  // 1 degree = 0,0174533 radians
  result -> value = cosf(num.value * 0.0174533);
  return true;
}
```

Computing the logarithm in e basis:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_LOG){
  struct numeric_variable num;
  if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
    return false;
  if(num.value <= 0.0){
    RAISE_ERROR_NEGATIVE_LOGARITHM(mf, cx, OPTIONAL(begin -> line),
                                   num.value);
    return false;
  }
  result -> value = logf(num.value);
  return true;
}
```

And this is how we compute the exponential which means $expx = e^x$:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_EXP){
  struct numeric_variable num;
  if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
    return false;
  result -> value = expf(num.value);
  return true;
}
```

The floor of a given value:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_FLOOR){
  struct numeric_variable num;
  if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
    return false;
  result -> value = floorf(num.value);
  return true;
}
```

About the operator `uniformdeviate`, it generates a number uniform and random between 0 and the value passed as operand. We can compute this generating a number between 0 and 1 and then multiply this value with the operand.

Generating a floating point number between 0 and 1 following a distribution near to uniform can be done generating a random 64-bit number and then multiplying it with 2^{-64}. Not all possible floating-point numbers could be represented generating them this way. Numbers smaller than 2^{-64} would be ignored and the rounding would make some numbers near 1 more probable, but we also would have less density in the range near 1. However, despithe these drawbacks, the result would be sufficiently precise for our purposes:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_UNIFORMDEVIATE){
  struct numeric_variable num;
  if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
    return false;
  { // Generating the random number
    uint64_t random_bits = random_func();
    float multiplicand = (float) ldexp((double) random_bits, -64);
    result -> value = multiplicand * num.value;
  }
  return true;
}
```

If the next operator is the symbol `+`, this means a multiplication by 1. This operation can be ignored:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_SUM){
  if(!eval_numeric_primary(mf, cx, begin -> next, end, result))
    return false;
  return true;
}
```

But if the operator is a symbol `-`, hen this means a multiplication by -1:

Section: Numeric Primary: Rule 3 (continuation):

```
else if(begin -> type == TYPE_SUBTRACT){
  if(!eval_numeric_primary(mf, cx, begin -> next, end, result))
    return false;
```

```
  result -> value *= -1;
  return true;
}
```

4) In the remaining cases, we have a scalar multiplicationwhere the scalar is a primary numeric token that is not followed by `+`, `-` or another numeric token. To deal with this, we needto identify the beginning and end of the primary numeric token. BY the rules it is a single numeric token, or three tokens (two numeric ones separated by a `/`). After separating the parts, the first part is multiplied by the second (a numeric primary expression):

Section: Numeric Primary: Rule 4:

```
else{
  float token_value;
  struct generic_token *after_token;
  if(begin -> type != TYPE_NUMERIC){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_NUMERIC);
    return false;
  }
  token_value = ((struct numeric_token *) begin) -> value;
  after_token = begin -> next;
  if(after_token -> type == TYPE_DIVISION){
    if(after_token == end){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_NUMERIC);
      return false;
    }
    after_token = after_token -> next;
    token_value /= ((struct numeric_token *) after_token) -> value;
    after_token = after_token -> next;
  }
  if(!eval_numeric_primary(mf, cx, after_token, end, result))
    return false;
  result -> value *= token_value;
  return true;
}
```

8.1.4. Isolated Numbers and Random Normal Values

The final rules for numeric expressions are:

```
<Numeric Atom> -> <Numeric Variable> |
                  <Primary Numeric Token> |
                  ( <Numeric Expression> ) |
                  normaldeviate
```

The only new token in this part is `normaldeviate`:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_NORMALDEVIATE,   // 0 token simblico 'normaldeviate'
```

Section: List of Keywords (continuation):

```
"normaldeviate",
```

This operator creates a new random number taken from a n ormal distribution with mean 0 and standard

deviation 1.

The function that evaluates numeric atoms is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_numeric_atom(struct metafont *mf, struct context *cx,
                       struct generic_token *begin,
                       struct generic_token *end,
                       struct numeric_variable *result);
```

We will decide which rule to apply while evaluating the numeric atom first checking if we have a single token or not, and then applying different rules based on this:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_numeric_atom(struct metafont *mf, struct context *cx,
                       struct generic_token *begin,
                       struct generic_token *end,
                       struct numeric_variable *result){
  if(begin == end){
                    <Section to be inserted: Numeric Atom: Rule 1>
                    <Section to be inserted: Numeric Atom: Rule 2>
                    <Section to be inserted: Numeric Atom: Rule 3>
  }
  else{
                    <Section to be inserted: Numeric Atom: Rule 4>
                    <Section to be inserted: Numeric Atom: Rule 5>
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_NUMERIC);
  return false;
}
```

1) If we have a single token and it is a numeric token, we just return its value:

Section: Numeric Atom: Rule 1:

```
if(begin -> type == TYPE_NUMERIC){
  result -> value = ((struct numeric_token *) begin) -> value;
  return true;
}
```

2) If we have a single token and it is a variable, we return its content. But we first need to check if the variable was declared, if it is numeric and if it was initialized. We also interpres specially the variables `w`, `h` and `d`:

Section: Numeric Atom: Rule 2:

```
if(begin -> type == TYPE_SYMBOLIC){
  struct symbolic_token *var_token = ((struct symbolic_token *) begin);
  struct numeric_variable *var;
  if(!strcmp(var_token -> value, "w"))
    var = &(cx -> internal_numeric_variables[INTERNAL_NUMERIC_W]);
  else if(!strcmp(var_token -> value, "h"))
    var = &(cx -> internal_numeric_variables[INTERNAL_NUMERIC_H]);
  else if(!strcmp(var_token -> value, "d"))
    var = &(cx -> internal_numeric_variables[INTERNAL_NUMERIC_D]);
  else
    var = var_token -> var;
  if(var == NULL){
```

```
    RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(begin -> line),
                                    var_token);
    return false;
  }
  if(var -> type != TYPE_T_NUMERIC){
    RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(begin -> line),
                                    var_token, var -> type,
                                    TYPE_T_NUMERIC);
    return false;
  }
  if(isnan(var -> value)){
    RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(begin -> line),
                                       var_token, TYPE_T_NUMERIC);
    return false;
  }
  result -> value = var -> value;
  return true;
}
```

3) Finally, if we have a single token and it is a `normaldeviate`, then we need to generate a random number taken from a normal distribution. Given that we already have a function to generate random bits, we can simulate a normal distribution following these steps:

a) First we generate two random numbers between -1 and 1. We will call them u and v. e can generate them getting 64 random bits, multiplying the first 63 by 2^{-63} and using the remaining bit to choose the signal. The result is sufficiently close to uniform.

b) If $u^2 + v^2 \geq 1$, the numbers form a point outside a circle with radius 1. In this case, the result should be discarded and we try to generate numbers again. We also discard the result if both numbers are zero, as in this case our method do not work.

c) After this, we can produce two numbers following a normal distribution computing:

$$x_0 = u\sqrt{-2\ln(u^2 + v^2)/(u^2 + v^2)}$$

$$x_1 = v\sqrt{-2\ln(u^2 + v^2)/(u^2 + v^2)}$$

One of the values will be returned. The other can be stored to be returned next time we need a random normal value. We store it in the Metafont struct:

Section: Attributes (struct metafont) (continuation):

```
  bool have_stored_normaldeviate;
  float normaldeviate;
```

Initially the structure will have no value stored. We will store values there only after using the procedure descrbed above:

Section: Initialization (struct metafont) (continuation):

```
mf -> have_stored_normaldeviate = false;
```

So, when we need to generate a normal rando value, we always check first if we already have a pre-generated value, and if not, we generate two values:

Section: Numeric Atom: Rule 3:

```
if(begin -> type == TYPE_NORMALDEVIATE){
  if(mf -> have_stored_normaldeviate){
    mf -> have_stored_normaldeviate = false;
    result -> value = mf -> normaldeviate;
    return true;
```

```
  }
  else{
    uint64_t random_bits;
    float u, v, s;
    do{
      random_bits = random_func();
      u = (float) ldexp((double) (random_bits >> 1), -63) *
            ((random_bits %2)?(-1.0):(+1.0));
      v = (float) ldexp((double) (random_bits >> 1), -63) *
            ((random_bits %2)?(-1.0):(+1.0));
      s = u*u + v*v;
    } while(s >= 1.0 || s == 0.0);
    u *= (float) sqrt((-2.0 * log((double) s))/s);
    v *= (float) sqrt((-2.0 * log((double) s))/s);
    mf -> have_stored_normaldeviate = true;
    mf -> normaldeviate = u;
    result -> value = v;
    return true;
  }
}
```

4) Now the cases where we have more than one token. If the first token in "(" and the last one is ")", we compute the inner expression as a numeric expression, and then return the result discarding the parenthesis:

Section: Numeric Atom: Rule 4:

```
if(begin -> type == TYPE_OPEN_PARENTHESIS &&
   end -> type == TYPE_CLOSE_PARENTHESIS){
   struct generic_token *p = begin;
   while(p -> next != end)
     p = p -> next;
   if(p == begin){
     RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '(');
     return false;
   }
   if(!eval_numeric_expression(mf, cx, begin -> next, p, result))
     return false;
   return true;
}
```

If we find just an empty parenthesis, this is an error. The user forgot to put some expression in the parenthesis.

5) Finally, the case when the numeric atom is a fraction composed by a numeric token, `/` and another numeric token. The result is obtained dividing both tokens:

Section: Numeric Atom: Rule 5:

```
if(begin -> type == TYPE_NUMERIC && end -> type == TYPE_NUMERIC &&
   begin -> next -> type == TYPE_DIVISION){
  if(((struct numeric_token *) end) -> value == 0.0){
    RAISE_ERROR_DIVISION_BY_ZERO(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  result -> value = ((struct numeric_token *) begin) -> value /
           ((struct numeric_token *) end) -> value;
  return true;
```

```
}
```

8.2. Pair Assignments and Expressions

To make the assignment to pair variables, we use the code below:

Section: Assignment for Pair Variables:

```
else if(type == TYPE_T_PAIR){
  int i;
  struct pair_variable result;
  if(!eval_pair_expression(mf, cx, begin_expression, *end, &result))
    return false;
  var = (struct symbolic_token *) begin;
  for(i = 0; i < number_of_variables; i ++){
    struct pair_variable *v = (struct pair_variable *) var -> var;
    v -> x = result.x;
    v -> y = result.y;
    var = (struct symbolic_token *) (var -> next);
    var = (struct symbolic_token *) (var -> next);
  }
}
```

Now let's see how to evaluate pair expressions.

8.2.1. Sum and Subtraction

The grammar rules for pair tertiary expressions begin with sum and subtraction:

```
<Pair Expression> -> <Pair Tertiary>
<Pair Tertiary> -> <Pair Secondary> |
                   <Pair Tertiary> <PT-Op> <Pair Secondary>
<PT-Op> -> + | -
```

Sum and subtraction is evaluated exactly as expected from vector sum and subtraction.

The function that evaluates pair expressions is declared here:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_pair_expression(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct pair_variable *result);
```

The method to evaluate tertiary pair expressions are not different from what we already defined with numeric expressions. We just have fewer tertiary operators here.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_pair_expression(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct pair_variable *result){
  struct generic_token *end_tertiary = NULL, *begin_secondary,
                       *last_sum = NULL, *p, *prev = NULL;
  DECLARE_NESTING_CONTROL();
  struct pair_variable a, b;
  p = begin;
  do{ // Find last tertiary operator '+' or '-'
    COUNT_NESTING(p);
```

```
    if(IS_NOT_NESTED() && IS_VALID_SUM_OR_SUB(prev, p)){
      last_sum = p;
      end_tertiary = prev;
    }
    prev = p;
    if(p != end)
      p = p -> next;
    else
      p = NULL;
  }while(p != NULL);
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
  if(end_tertiary != NULL){
    begin_secondary = last_sum -> next;
    if(!eval_pair_expression(mf, cx, begin, end_tertiary, &a))
      return false;
    if(!eval_pair_secondary(mf, cx, begin_secondary, end, &b))
      return false;
    if(last_sum -> type == TYPE_SUM){ // Evaluate '+'
      result -> x = a.x + b.x;
      result -> y = a.y + b.y;
    }
    else if(last_sum -> type == TYPE_SUBTRACT){ // Evaluate '-'
      result -> x = a.x - b.x;
      result -> y = a.y - b.y;
    }
    return true;
  }
  else // No tertiary operator:
    return eval_pair_secondary(mf, cx, begin, end, result);
}
```

8.2.2. Transformers and Scalar Multiplication and Division

The grammar for secondary pair expressions is:

```
<Pair Secondary> -> <Pair Primary> |
                    <Pair Secondary><Mul or Div><Numeric Primary> |
                    <Numeric Secondary> * <Pair Primary> |
                    <Pair Secondary><Transformer>
<Mul or Div> -> * | /
<Transformer> -> rotated <Numeric Primary> |
                 scaled <Numeric Primary> |
                 shifted <Pair Primary> |
                 slanted <Numeric Primary> |
                 xscaled <Numeric Primary> |
                 yscaled <Numeric Primary> |
                 zscaled <Pair Primary> | (...)
```

The above transformer rule is incomplete because we will see the last existing transformer in Subsection 8.3.1.

For now let's add the seven new keywords representing known transformers:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_ROTATED,  // Symbolic token 'rotated'
```

```
TYPE_SCALED,   // Symbolic token 'scaled'
TYPE_SHIFTED,  // Symbolic token 'shifted'
TYPE_SLANTED,  // Symbolic token 'slanted'
TYPE_XSCALED,  // Symbolic token 'xscaled'
TYPE_YSCALED,  // Symbolic token 'yscaled'
TYPE_ZSCALED,  // Symbolic token 'zscaled'
```

Section: List of Keywords (continuation):

```
"rotated", "scaled", "shifted", "slanted", "xscaled", "yscaled",
"zscaled",
```

The function that evaluates secondary pair expressions is declared below:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_pair_secondary(struct metafont *mf, struct context *cx,
                         struct generic_token *begin,
                         struct generic_token *end,
                         struct pair_variable *result);
```

Evaluating a secondary expression here is very similar to what we did for numeric expressions. We also walk over the token list searching for the rightmost secondary operator, ignoring anything nested inside parenthesis and brackets. We follow the same rules to check when we have a division and when the symbol / is just a fraction separator. The difference is that here we have a total of nine secondary operators including the transformers. Because of the big quantity, we will show each of them separately instead of all them in the code block below:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_pair_secondary(struct metafont *mf, struct context *cx,
                           struct generic_token *begin,
                           struct generic_token *end,
                           struct pair_variable *result){
  struct generic_token *end_secondary = NULL, *begin_primary,
                       *last_mul = NULL, *p, *prev = NULL,
                       *prev_prev = NULL, *last_fraction = NULL;
  DECLARE_NESTING_CONTROL();
  p = begin;
  do{ // Find the rightmost secondary operator
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && (p -> type == TYPE_MULTIPLICATION ||
             p -> type == TYPE_DIVISION || p -> type == TYPE_ROTATED ||
             p -> type == TYPE_SCALED || p -> type == TYPE_SHIFTED ||
             p -> type == TYPE_SLANTED || p -> type == TYPE_XSCALED ||
             p -> type == TYPE_YSCALED || p -> type == TYPE_ZSCALED ||
             // Operator 'transformed'  will be defined in Subsection 8.3.1.
             p -> type == TYPE_TRANSFORMED)){
      if(p -> type == TYPE_DIVISION && prev -> type == TYPE_NUMERIC &&
         p != end && p -> next -> type != TYPE_NUMERIC &&
         last_fraction != prev_prev) // Fraction separator
         last_fraction = p;
       else{ //  Diviso ou operadores vlidos
         last_mul = p;
         end_secondary = prev;
       }
```

```
    }
    prev_prev = prev;
    prev = p;
    if(p != end)
      p = p -> next;
    else
      p = NULL;
  }while(p != NULL);
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
  if(end_secondary != NULL){
    begin_primary = last_mul -> next;
            <Section to be inserted: Secondary Pair Operator: Multiplication>
              <Section to be inserted: Secondary Pair Operator: Division>
              <Section to be inserted: Secondary Pair Operator: Rotation>
               <Section to be inserted: Secondary Pair Operator: Scaling>
              <Section to be inserted: Secondary Pair Operator: Shifting>
              <Section to be inserted: Secondary Pair Operator: Slanting>
             <Section to be inserted: Secondary Pair Operator: X-Scaling>
             <Section to be inserted: Secondary Pair Operator: Y-Scaling>
              <Section to be inserted: Secondary Pair Operator: Z-Scaling>
        <Section to be inserted: Secondary Pair Operator: Additional Operators>
  }
  else
    return eval_pair_primary(mf, cx, begin, end, result);
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PAIR);
  return false;
}
```

The first operator is the multiplication. Notice that we can have two different multiplications: a pair multiplicated by a numeric or a numeric multiplicated by a pair. To identify which of the two should be applied, we must check the primary expression to the right of the operator.

Section: Secondary Pair Operator: Multiplication:

```
if(last_mul -> type == TYPE_MULTIPLICATION){
  if(get_primary_expression_type(mf, cx, begin_primary, end) == TYPE_T_PAIR){
    struct numeric_variable a;
    struct pair_variable b;
    if(!eval_numeric_secondary(mf, cx, begin, end_secondary, &a))
      return false;
    if(!eval_pair_primary(mf, cx, begin_primary, end, &b))
      return false;
    result -> x = b.x * a.value;
    result -> y = b.y * a.value;
    return true;
  }
  else{
    struct pair_variable a;
    struct numeric_variable b;
    if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
      return false;
    if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
      return false;
```

```
    result -> x = a.x * b.value;
    result -> y = a.y * b.value;
    return true;
  }
}
```

If we have a division, then this always will be a pair divided by a numeric. We should generate an error in case of division by zero:

Section: Secondary Pair Operator: Division:

```
else if(last_mul -> type == TYPE_DIVISION){
  struct pair_variable a;
  struct numeric_variable b;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
    return false;
  if(b.value == 0.0){
    RAISE_ERROR_DIVISION_BY_ZERO(mf, cx, OPTIONAL(last_mul -> line));
    return false;
  }
  result -> x = a.x / b.value;
  result -> y = a.y / b.value;
  return true;
}
```

If we have a rotation, we rotate our pair counter-clockwise using the origin as the axis. We interpret angles in degrees, not in radians:

Section: Secondary Pair Operator: Rotation:

```
else if(last_mul -> type == TYPE_ROTATED){
  struct pair_variable a;
  struct numeric_variable b;
  double sin_theta, cos_theta, theta;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
    return false;
  theta = 0.0174533 * b.value; // 1 degree = 0.0174533 radians
  sin_theta = sin(theta);
  cos_theta = cos(theta);
  result -> x = a.x * cos_theta - a.y * sin_theta;
  result -> y = a.x * sin_theta + a.y * cos_theta;
  return true;
}
```

A scaling is the same thing than a multiplication:

Section: Secondary Pair Operator: Scaling:

```
else if(last_mul -> type == TYPE_SCALED){
  struct pair_variable a;
  struct numeric_variable b;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
```

```
    return false;
  result -> x = a.x * b.value;
  result -> y = a.y * b.value;
  return true;
}
```

A shifting is equal a sum, but this operator have a higher precedence:

Section: Secondary Pair Operator: Shifting:

```
else if(last_mul -> type == TYPE_SHIFTED){
  struct pair_variable a, b;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_pair_primary(mf, cx, begin_primary, end, &b))
    return false;
  result -> x = a.x + b.x;
  result -> y = a.y + b.y;
  return true;
}
```

The slanting operator shifts a point more to the right depending on how above the axis x it is and more to the left depending on how below the axis x it is:

Section: Secondary Pair Operator: Slanting:

```
else if(last_mul -> type == TYPE_SLANTED){
  struct pair_variable a;
  struct numeric_variable b;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
    return false;
  result -> x = a.x + b.value * a.y;
  result -> y = a.y;
  return true;
}
```

A x-scaling multiplies a numeric scalar to the first value in the pair:

Section: Secondary Pair Operator: X-Scaling:

```
else if(last_mul -> type == TYPE_XSCALED){
  struct pair_variable a;
  struct numeric_variable b;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
    return false;
  result -> x = a.x * b.value;
  result -> y = a.y;
  return true;
}
```

Likewise a y-scaling multiplies a numeric scalar to the second value in the pair:

Section: Secondary Pair Operator: Y-Scaling:

```
else if(last_mul -> type == TYPE_YSCALED){
  struct pair_variable a;
  struct numeric_variable b;
```

```
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_numeric_primary(mf, cx, begin_primary, end, &b))
    return false;
  result -> x = a.x;
  result -> y = a.y * b.value;
  return true;
}
```

Finally, a z-scaling interprets two pairs as complex numbers and multiply them:

$$(a + bi)(c + di) = ac + (ad)i + (cb)i + (bd)i^2 = (ac - bd) + (cb + ad)i$$

Section: Secondary Pair Operator: Z-Scaling:

```
else if(last_mul -> type == TYPE_ZSCALED){
  struct pair_variable a, b;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_pair_primary(mf, cx, begin_primary, end, &b))
    return false;
  result -> x = a.x * b.x - a.y * b.y;
  result -> y = a.x * b.y + b.x * a.y;
  return true;
}
```

8.2.3. Pair Intermediary Values, Literals and Variables

The final grammar rules for pair expressions are:

```
<Pair Primary> -> <Pair Variable> |
                  ( <Numeric Expression> , <Numeric Expression> ) |
                  ( <Pair Expression> ) |
                  (...) |
                  <Numeric Atom> [ <Pair Expression, <Pair Expression>] |
                  <Scalar Multiplication Operator><Pair Primary>
```

We ommited some rules above because there are primary operators that we will define later, in Subsections 8.4.6 and 11.4.

The novel operator is the constructionn $a[b, c]$, where b and c are pairs. It represents intermediary values between b end c. It is evaluated as $a(b + c)$. This means that $.5[a, b]$ is half the path between b and c.

The other rules are analogous to what we already described in the grammar for numeric expressions.

The function that will evaluate pair primary expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_pair_primary(struct metafont *mf, struct context *cx,
                       struct generic_token *begin,
                       struct generic_token *end,
                       struct pair_variable *result);
```

Each one of the five grammar rules will be tested separately to discover which one we should apply when we find a primary expression:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_pair_primary(struct metafont *mf, struct context *cx,
                       struct generic_token *begin,
                       struct generic_token *end,
```

```
                        struct pair_variable *result){
  if(begin == end){
                    <Section to be inserted: Pair Primary: Rule 1>
  }
  else if(begin -> type == TYPE_OPEN_PARENTHESIS &&
          end -> type == TYPE_CLOSE_PARENTHESIS){
                    <Section to be inserted: Pair Primary: Rule 2>
                    <Section to be inserted: Pair Primary: Rule 3>
  }
       <Section to be inserted: Pair Primary: Other Rules to Be Defined Later>
                  <Section to be inserted: Pair Primary: Rule 4>
                  <Section to be inserted: Pair Primary: Rule 5>
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PAIR);
  return false;
}
```

The first rule will be applied when we have a single token in the expression. The only case when it happens is when we are evaluating a pair variable:

Section: Pair Primary: Rule 1:

```
struct symbolic_token *tok = (struct symbolic_token *) begin;
struct pair_variable *var;
if(tok -> type != TYPE_SYMBOLIC){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PAIR);
  return false;
}
var = (struct pair_variable *) tok -> var;
if(var == NULL){
  RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(begin -> line), tok);
  return false;
}
if(var -> type != TYPE_T_PAIR){
  RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(begin -> line),
                                  tok, var -> type,
                                  TYPE_T_PAIR);
  return false;
}
if(isnan(var -> x)){
  RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(begin -> line),
                                     tok, TYPE_T_PAIR);
  return false;
}
result -> x = var -> x;
result -> y = var -> y;
return true;
```

If the expression is delimited by parenthesis, we could be facing a literal representation of a pair in the form (a, b), or we could be dealing with a pair expression inside parenthesis like in $(pair1 + (a, b))$. We can differentiate the two cases checking the presence of a comma inside the parenthesis which is not nested in other internal parenthesis.

Section: Pair Primary: Rule 2:

```
  struct generic_token *begin_a, *end_a, *begin_b, *end_b, *comma;
  if(begin -> next == end){
```

```
    RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '(');
    return false;
  }
  begin_a = (struct generic_token *) begin -> next;
  end_a = begin_a;
  DECLARE_NESTING_CONTROL();
  bool literal = true;
  while(end_a != NULL){
    COUNT_NESTING(end_a);
    if(IS_NOT_NESTED() &&
       ((struct generic_token *) end_a -> next) -> type == TYPE_COMMA)
      break;
    if(end_a -> next != end)
      end_a = (struct generic_token *) end_a -> next;
    else{
      literal = false;
      break;
    }
  }
  if(literal){
    struct numeric_variable a, b;
    comma = (struct generic_token *) end_a -> next;
    begin_b = (struct generic_token *) comma -> next;
    if(begin_b == end){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(comma -> line),
                                     TYPE_T_PAIR);
      return false;
    }
    end_b = begin_b;
    while(end_b -> next != end)
      end_b = (struct generic_token *) end_b -> next;
    if(!eval_numeric_expression(mf, cx, begin_a, end_a, &a))
      return false;
    if(!eval_numeric_expression(mf, cx, begin_b, end_b, &b))
      return false;
    result -> x = a.value;
    result -> y = b.value;
    return true;
  }
```

In the code above we identify if we are dealing with a literal checking for a comma. If so, we set in the boolean variable `literal` as true. Otherwise, immediately after the last `if` above, we run the following `else` as the next rule:

Section: Pair Primary: Rule 3:

```
else
  return eval_pair_expression(mf, cx, begin_a, end_a, result);
```

If the last token is a `]`, then we have a construction of type $a[B, C]$ for some numeric a and pairs B and C. Our task is separate the three parts a, B and C, evaluate them and return $B + a(C - B)$:

Section: Pair Primary: Rule 4:

```
else if(end -> type == TYPE_CLOSE_BRACKETS){
  struct generic_token *begin_a, *end_a, *begin_b, *end_b, *begin_c,
```

```
                      *end_c;
struct numeric_variable a;
struct pair_variable b, c;
DECLARE_NESTING_CONTROL();
begin_a = begin;
end_a = begin_a;
while(end_a != end){ // a: From beginning expression to token before '['
  COUNT_NESTING(end_a);
  if(IS_NOT_NESTED() && end_a -> next -> type == TYPE_OPEN_BRACKETS)
    break;
  end_a = end_a -> next;
}
if(end_a == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PAIR);
  return false;
}
begin_b = end_a -> next;
begin_b = begin_b -> next; // b: Begin after '['
end_b = begin_b;
while(end_b != end){ // b: Ends before ','
  COUNT_NESTING(end_b);
  if(IS_NOT_NESTED() && end_b -> next -> type == TYPE_COMMA)
    break;
  end_b = end_b -> next;
}
if(end_b == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PAIR);
  return false;
}
begin_c = end_b -> next;
begin_c = begin_c -> next; // c: Begins after ','
end_c = begin_c;
while(end_c != end){ // c: ends in the second to last token
  if(end_c -> next == end)
    break;
  end_c =  end_c -> next;
}
if(end_c == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PAIR);
  return false;
}
if(!eval_numeric_atom(mf, cx, begin_a, end_a, &a)) // Eval 'a'
  return false;
if(!eval_pair_expression(mf, cx, begin_b, end_b, &b)) // Eval 'b'
  return false;
if(!eval_pair_expression(mf, cx, begin_c, end_c, &c)) // Eval 'c'
  return false;
result -> x = b.x + a.value * (c.x - b.x); // Computes b + a(c-b)
```

```
  result -> y = b.y + a.value * (c.y - b.y);
  return true;
}
```

The last rule is when we have a scalar multiplication operator. The operator can be +, −, a single numeric token or a fraction. We have one example for each of the four cases below:

```
pair a, b, c, d;
a = +(1, 2);
b = -(1, 2);
c = 2a;
d = 1/2b;
```

The code that deals with such expressions are:

Section: Pair Primary: Rule 5:

```
else{
  if(begin -> type == TYPE_SUM) // Unary '+' before expression
    return eval_pair_primary(mf, cx, begin -> next, end, result);
  else if(begin -> type == TYPE_SUBTRACT){ // Unary '-'
    if(!eval_pair_primary(mf, cx, begin -> next, end, result))
      return false;
    result -> x = - (result -> x);
    result -> y = - (result -> y);
    return true;
  }
  else if(begin -> type == TYPE_NUMERIC){ // Number/fraction before expression
    struct generic_token *tok;
    float value = ((struct numeric_token *) begin) -> value;
    tok = begin -> next;
    if(tok -> type == TYPE_DIVISION){ // It is a fraction
      tok = begin -> next;
      if(tok == end || tok -> next == end || tok -> type != TYPE_NUMERIC){
        RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                       TYPE_T_PAIR);
        return false;
      }
      if(((struct numeric_token *) tok) -> value == 0.0){
        RAISE_ERROR_DIVISION_BY_ZERO(mf, cx, OPTIONAL(begin -> line));
        return false;
      }
      value /= ((struct numeric_token *) tok) -> value;
      tok = tok -> next;
    }
    if(!eval_pair_primary(mf, cx, begin -> next, end, result))
      return false;
    result -> x *= value;
    result -> y *= value;
    return true;
  }
}
```

8.2.4. Pairs in Numeric Expressions

Numeric subexpressions can appear inside pair expressions. For example, in $a[b, c]$, a is a numeric atom. Likewise, pair subexpressions can appear inside numeric expressions. We did not define this in the section about numeric expressions because we still had not defined how to evaluate pair expressions. There are four numeric primary operators involving pairs:

```
<Numeric Primary> -> length <Pair Primary> | xpart <Pair Primary> |
                     ypart <Pair Primary> | angle <Pair Primary>
```

This require defining the following three new token types:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_XPART,  // Symbolic token 'xpart'
TYPE_YPART,  // Symbolic token 'ypart'
TYPE_ANGLE,  // Symbolic token 'angle'
```

They correspond to the following keywords:

Section: List of Keywords (continuation):

```
"xpart", "ypart", "angle",
```

The first case, which did not need a new token, othe operator `length` was already being used to get the modulus of a given number and we had written that it could be used on other types, not only in numerics. In the case of pairs, this operators returns the euclidean norm:

Section: Evaluate 'length':

```
else if(expr_type == TYPE_T_PAIR){
  struct pair_variable p;
  if(!eval_pair_primary(mf, cx, begin -> next, end, &p))
    return false;
  result -> value = (float) hypot(p.x, p.y);
  return true;
}
```

The operator `xpart` returns the first value in a pair. However, we must take an additioonal care: this operator can be use in contexts other than getting value from a pair, as we will still see in Sub-subsection 8.3.3. Therefore, first we must check if what we have after the operator is a pair. If so, we return the first value:

Section: Numeric Primary: Additional Operators:

```
else if(begin -> type == TYPE_XPART){
  struct pair_variable p;
  int expr_type = get_primary_expression_type(mf, cx, begin -> next, end);
  if(expr_type == TYPE_T_PAIR){
    if(!eval_pair_primary(mf, cx, begin -> next, end, &p))
      return false;
    result -> value = p.x;
    return true;
  }
  else{
            <Section to be inserted: Numeric Primary: X-Part in Non-Pair>
  }
}
```

While the operator `ypart` returns the second value in a pair. In this case we also must take care to check if we have as operand is a pair or some other thing:

Section: Numeric Primary: Additional Operators (continuation):

```
else if(begin -> type == TYPE_YPART){
```

```
  struct pair_variable p;
  int expr_type = get_primary_expression_type(mf, cx, begin -> next, end);
  if(expr_type == TYPE_T_PAIR){
    if(!eval_pair_primary(mf, cx, begin -> next, end, &p))
      return false;
    result -> value = p.y;
    return true;
  }
  else{
            <Section to be inserted: Numeric Primary: Y-Part in Non-Pair>
  }
}
```

Finally, the last operator, `angle`, measure the angle of a pair. Which means the angle between the segment that connects the origin to the pair coordintate and the segment that connects the origin to $(1, 0)$. An error will be generated if you try to measure the angle of $(0, 0)$:

Section: Numeric Primary: Additional Operators (continuation):

```
else if(begin -> type == TYPE_ANGLE){
  struct pair_variable p;
  if(!eval_pair_primary(mf, cx, begin -> next, end, &p))
    return false;
  if(p.x == 0.0 && p.y == 0.0){
    RAISE_ERROR_NULL_VECTOR_ANGLE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  result -> value = (float) acos(p.x / (hypot(p.x, p.y)));
  result -> value *= 57.2958; // Radians to degrees
  return true;
}
```

8.3. Transform Assignments and Expressions

One of the moments where we expect to eval a transform expression is when making assignments to transform variables. In this case, we apply a function to evaluate the expression and get the result. Then, we assign the result for each variable in the left side of the assignment expression:

Section: Assignment for Transform Variables:

```
else if(type == TYPE_T_TRANSFORM){
  int i;
  struct transform_variable result;
  if(!eval_transform_expression(mf, cx, begin_expression, *end, &result))
    return false;
  var = (struct symbolic_token *) begin;
  for(i = 0; i < number_of_variables; i ++){
    memcpy(((struct transform_variable *) var -> var) -> value, result.value,
           sizeof(float) * 9);
    var = (struct symbolic_token *) (var -> next);
    var = (struct symbolic_token *) (var -> next);
  }
}
```

Now let's see how do we evaluate a transform expression.

8.3.1. Transforming Transformers

The grammar rules to evaluate a transform expression begins with:

```
<Transform Expression> -> <Transform Tertiary>
<Transform Tertiary> -> <Transform Secondary>
<Transform Secondary> -> <Transform Secondary> <Transformer> |
                         <Transform Primary>
<Transformer> -> rotated <Numeric Primary> |
                 scaled <Numeric Primary> |
                 shifted <Pair Primary> |
                 slanted <Numeric Primary> |
                 xscaled <Numeric Primary> |
                 yscaled <Numeric Primary> |
                 zscaled <Pair Primary>
                 transformed <Transform Primary>
```

We have here a new type of token, representing the last type of transformer that we ommited in the last subsection about pair expressions:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_TRANSFORMED,  // Symbolic token 'transformed'
```

This token correspond to the following keyword:

Section: List of Keywords (continuation):

```
"transformed",
```

About the grammar rules, they mean that there is no tertiary operators for transform expressions. Anyway, we will define a function to evaluate tertiary expressions just to keep our API uniform between different functions that evaluate expressions and also to keep a function ready to be modified if in the future a tertiary operator is added to the language:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_transform_expression(struct metafont *mf, struct context *cx,
                               struct generic_token *begin,
                               struct generic_token *end,
                               struct transform_variable *result);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_transform_expression(struct metafont *mf, struct context *cx,
                               struct generic_token *begin,
                               struct generic_token *end,
                               struct transform_variable *result){
  return eval_transform_secondary(mf, cx, begin, end, result);
}
```

The function that evaluates secondary transform expressions is this:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_transform_secondary(struct metafont *mf, struct context *cx,
                              struct generic_token *begin,
                              struct generic_token *end,
                              struct transform_variable *result);
```

This function works walking over the list os tokens in the expression, ignoring tokens nested in parenthesis, brackets and braces. Each time it finds a secondary operator, it stores its position. After walking all

tokens, if we did not find a secondary operator, we evaluate the entire expression as a primary transform expression. Otherwise, everything before the last operator is evaluated as a secondary expression and we apply the operator over the resulting transform that we find after evaluating the expression:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_transform_secondary(struct metafont *mf, struct context *cx,
                              struct generic_token *begin,
                              struct generic_token *end,
                              struct transform_variable *result){
  struct generic_token *p, *last_transform = NULL, *last_token = NULL,
                       *end_secondary = NULL;
  DECLARE_NESTING_CONTROL();
  p = begin;
  do{ // FInds rightmost transform operator:
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() &&
       (p -> type == TYPE_ROTATED || p -> type == TYPE_SCALED ||
        p -> type == TYPE_SHIFTED || p -> type == TYPE_SLANTED ||
        p -> type == TYPE_XSCALED || p -> type == TYPE_YSCALED ||
        p -> type == TYPE_ZSCALED || p -> type == TYPE_TRANSFORMED)){
      last_transform = p;
      end_secondary = last_token;
    }
    last_token = p;
    if(p != end)
      p = p -> next;
    else
      p = NULL;
  } while(p != NULL);
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
  if(last_transform != NULL){ // We have a secondary operator:
    if(end_secondary == NULL){ // But there is nothing before it:
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_TRANSFORM);
      return false;
    }
    if(!eval_transform_secondary(mf, cx, begin, end_secondary, result))
      return false;
        <Section to be inserted: Apply Secondary Transformer over Transforms>
  }
  else // NO secondary operator:
    return eval_transform_primary(mf, cx, begin, end, result);
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_TRANSFORM);
  return false;
}
```

Now let's apply the different transformer operators over our transform variables and results. The first kind of transformer is the rotation, which we apply with the code below:

Section: Apply Secondary Transformer over Transforms:

```
if(last_transform -> type == TYPE_ROTATED){
  struct numeric_variable theta;
```

```
  double angle_radian;
  if(!eval_numeric_primary(mf, cx, last_transform -> next, end, &theta))
    return false;
  // 1 degree = 0,0174533 radians
  angle_radian = theta.value * 0.0174533;
  TRANSFORM_ROTATE(result -> value, angle_radian);
  return true;
}
```

Changing the scale for a transform means running the folllowing code:

Section: Apply Secondary Transformer over Transforms (continuation):

```
else if(last_transform -> type == TYPE_SCALED){
  struct numeric_variable scale;
  if(!eval_numeric_primary(mf, cx, last_transform -> next, end, &scale))
    return false;
  TRANSFORM_SCALE(result -> value, scale.value);
  return true;
}
```

The next transformer makes a translation, or shifting over a distance (x, y).

Section: Apply Secondary Transformer over Transforms (continuation):

```
else if(last_transform -> type == TYPE_SHIFTED){
  struct pair_variable shift;
  if(!eval_pair_primary(mf, cx, last_transform -> next, end, &shift))
    return false;
  TRANSFORM_SHIFT(result -> value, shift.x, shift.y);
  return true;
}
```

Now the transformer that slants a transform with intensity s:

Section: Apply Secondary Transformer over Transforms (continuation):

```
else if(last_transform -> type == TYPE_SLANTED){
  struct numeric_variable slant;
  if(!eval_numeric_primary(mf, cx, last_transform -> next, end, &slant))
    return false;
  TRANSFORM_SLANT(result -> value, slant.value);
  return true;
}
```

Changing the scale only in axis x shrinks or expand the transform in that axis by some factor s. This operation is done by the following code:

Section: Apply Secondary Transformer over Transforms (continuation):

```
else if(last_transform -> type == TYPE_XSCALED){
  struct numeric_variable scale;
  if(!eval_numeric_primary(mf, cx, last_transform -> next, end, &scale))
    return false;
  TRANSFORM_SCALE_X(result -> value, scale.value);
  return true;
}
```

We can also change the scale only in the axis y by some factor s:

Section: Apply Secondary Transformer over Transforms (continuation):

```
else if(last_transform -> type == TYPE_YSCALED){
  struct numeric_variable scale;
  if(!eval_numeric_primary(mf, cx, last_transform -> next, end, &scale))
    return false;
  TRANSFORM_SCALE_Y(result -> value, scale.value);
  return true;
}
```

Finally, we can change the scale in the complex plane, multiplying each point by (s, t), interpreting everything as complex numbers. The result is a simultaneous rotation and scaling. This transformer is implemented by the code below:

Section: Apply Secondary Transformer over Transforms (continuation):

```
else if(last_transform -> type == TYPE_ZSCALED){
  struct pair_variable scale;
  if(!eval_pair_primary(mf, cx, last_transform -> next, end, &scale))
    return false;
  TRANSFORM_SCALE_Z(result -> value, scale.x, scale.y);
  return true;
}
```

Finally, the last transformer combines two transforms in a single one. This is done multiplying the matrices that represent the transform:

Section: Apply Secondary Transformer over Transforms (continuation):

```
else if(last_transform -> type == TYPE_TRANSFORMED){
  struct transform_variable b;
  if(!eval_transform_primary(mf, cx, last_transform -> next, end, &b))
    return false;
  MATRIX_MULTIPLICATION(result -> value, b.value);
  return true;
}
```

8.3.2 Transform Tertiary Expressions: Literals and Variables

The remaining rules in the grammar for transform expressions are:

```
<Transform Primary> -> <Transform Variable> |
                       ( <Transform Tertiary> ) |
                       ( <Numeric Expression> , <Numeric Expression> ,
                         <Numeric Expression> , <Numeric Expression> ,
                         <Numeric Expression> , <Numeric Expression> )
```

In the first case, we have a single symbolic token with a transform variable. In the other two cases, we have more than one token where the first one is an open parenthesis. In one we have a single transform expression in the parenthesis and in the other we have comma-separated six numeric expressions expressing a transform literal.

The function that evaluates primary transform expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_transform_primary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
                            struct transform_variable *result);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_transform_primary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
                            struct transform_variable *result){
  if(begin == end && begin -> type == TYPE_SYMBOLIC){ // It's a variable?
                <Section to be inserted: Primary Transform: Variable>
  }
  else if(begin != end && begin -> type == TYPE_OPEN_PARENTHESIS &&
          end -> type == TYPE_CLOSE_PARENTHESIS){
    if(begin -> next == end){
      RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '(');
      return false;
    }
    struct generic_token *p = begin -> next;
    DECLARE_NESTING_CONTROL();
    bool has_comma = false;
    do{
      COUNT_NESTING(p);
      if(IS_NOT_NESTED() && p -> type == TYPE_COMMA){
        RESET_NESTING_COUNT();
        has_comma = true;
        break;
      }
      if(p != end)
        p = p -> next;
      else
        p = NULL;
    } while(p != NULL);
    if(has_comma){
                  <Section to be inserted: Primary Transform: Literal>
    }
    else{
                <Section to be inserted: Primary Transform: Parenthesis>
    }
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_TRANSFORM);
  return false;
}
```

The above code identifies if we are dealing with one of the three cases of transform primary expressions: a variable, a literal or a subexpression. The first case is the easiest: we have a variable if there is a single symbolic token. The other two are always delimited by parenthesis, but we can distinguish between both because a transform literal is composed by commas delimiting its values.

The simplest case is evaluating a transform variable:

Section: Primary Transform: Variable:

```
struct symbolic_token *v = (struct symbolic_token *) begin;
struct transform_variable *content = v -> var;
if(content == NULL){
```

```
  RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(v -> line), v);
  return false;
}
if(content -> type != TYPE_T_TRANSFORM){
  RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(v -> line),
                                  v -> value, content -> type,
                                  TYPE_T_TRANSFORM);
  return false;
}
if(isnan(content -> value[0])){
  RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(v -> line), v,
                                     TYPE_T_TRANSFORM);
  return false;
}
memcpy(result -> value, content -> value, sizeof(float) * 9);
return true;
```

The next case is when we have a subexpression between parenthesis:

Section: Primary Transform: Parenthesis:

```
struct generic_token *end_expr;
for(end_expr = begin -> next; end_expr -> next != end;
    end_expr = end_expr -> next);
return eval_transform_expression(mf, cx, begin -> next, end_expr, result);
```

And finally, the last case, where we need to interpret six numeric expressions to obtain a transform literal:

Section: Primary Transform: Literal:

```
int i;
struct generic_token *begin_numeric_expr, *end_numeric_expr;
struct numeric_variable numeric_result;
end_numeric_expr = begin_numeric_expr = begin -> next;
float values[6]; // Each one of the 6 values will be stored here
for(i = 0; i < 6; i ++){
  p = begin_numeric_expr;
  do{
    if(p != end){ // Ignoring last ')'
      COUNT_NESTING(p);
    }
    if(IS_NOT_NESTED() && ((i < 5 && p -> type == TYPE_COMMA) ||
                           (i == 5 && p -> type == TYPE_CLOSE_PARENTHESIS))){
      break;
    }
    end_numeric_expr = p;
    if(p != end)
      p = (struct generic_token *) p -> next;
    else
      p = NULL;
  } while(p != NULL);
  if(p == NULL){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_TRANSFORM);
    return false;
```

```
  }
  if(!eval_numeric_expression(mf, cx, begin_numeric_expr, end_numeric_expr,
                              &numeric_result))
    return false;
  values[i] = numeric_result.value;
  begin_numeric_expr = p -> next;
  end_numeric_expr = begin_numeric_expr;
}
// Storing in the right order inside the matrix:
result -> value[0] = values[2]; result -> value[1] = values[4];
result -> value[2] = 0.0;
result -> value[3] = values[3]; result -> value[4] = values[5];
result -> value[5] = 0.0;
result -> value[6] = values[0]; result -> value[7] = values[1];
result -> value[8] = 1.0;
return true;
```

The above code has two loops. The first one has as invariant the property that in the beginning of each iteration, `p` and `begin_numeric_expr` points to the first token in the next numeric expression in the literal that we still did not evaluate. The second loop will update `end_numeric_expr` so that it will mark the end of the next non-evaluated numeric expression. After the inner loop exits, we have the numeric expression correctly delimited and we can evaluate it and store its value as part of the transform. We repeat this siz times to read the entire transform literal.

8.3.3. Transforms in Numeric Expressions

Transforms can be placed in primary numeric expressions. The following grammar rules allow them:

```
<Numeric Primary> -> <Transform Part><Transform Primary>
<Transform Part> -> xpart | ypart | xxpart | xypart | yxpart | yypart
```

The six operators above extract each one of the six numeric values in a transform, in the same order that they are listed.

As `xpart` and `ypart` already were defined when we were defining pair expressions, we just need to define the next four new tokens for the new operators:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_XXPART,  // Symbolic token xxpart'
TYPE_XYPART,  // Symbolic token xypart'
TYPE_YXPART,  // Symbolic token yxpart'
TYPE_YYPART,  // Symbolic token yypart'
```

And the keywords associated with them:

Section: List of Keywords (continuation):

```
"xxpart", "xypart", "yxpart", "yypart",
```

These four new operators are evaluated in the function that evaluates primary numeric expressions:

Section: Numeric Primary: Additional Operators (continuation):

```
else if(begin -> type >= TYPE_XXPART && begin -> type <= TYPE_YYPART){
  struct transform_variable t;
  if(!eval_transform_primary(mf, cx, begin -> next, end, &t))
    return false;
  if(begin -> type == TYPE_XXPART)
    result -> value = t.value[0];
  else if(begin -> type == TYPE_XYPART)
```

```
    result -> value = t.value[3];
  else if(begin -> type == TYPE_YXPART)
    result -> value = t.value[1];
  else if(begin -> type == TYPE_YYPART)
    result -> value = t.value[4];
  return true;
}
```

But what about `xpart` and `ypart`? They were already treated when we were evaluating primary numeric expressions in Subsection 8.2.4. But we need to augment the definition of this operator present there to also deal with cases where the operand is not a pair, but a transform. This is how we do this for the `xpart` operator:

Section: Numeric Primary: X-Part in Non-Pair:

```
struct transform_variable t;
if(!eval_transform_primary(mf, cx, begin -> next, end, &t))
  return false;
result -> value = t.value[6];
return true;
```

And the same must be done for the `ypart` operator:

Section: Numeric Primary: Y-Part in Non-Pair:

```
struct transform_variable t;
if(!eval_transform_primary(mf, cx, begin -> next, end, &t))
  return false;
result -> value = t.value[7];
return true;
```

8.3.4. Transforms in Pair Expressions

Transforms also appear in grammar rules about pair expressions. It is needed to deal woth transform expressions in pair secondary expressions:

```
<Pair Secondary> -> <Pair Secondary><Transformer>
<Transformer> -> (...) | transformed <Transform Primary>
```

This operator apply the linear transform in the right-side to the pair in the left-side. We can implement it with the code below that augment the already written code about transformers in Subsection 8.2.2 about pair secondary expressions.

Section: Secondary Pair Operator: Additional Operators:

```
else if(last_mul -> type == TYPE_TRANSFORMED){
  struct pair_variable a;
  struct transform_variable b;
  if(!eval_pair_secondary(mf, cx, begin, end_secondary, &a))
    return false;
  if(!eval_transform_primary(mf, cx, begin_primary, end, &b))
    return false;
  result -> x = LINEAR_TRANSFORM_X(a.x, a.y, b.value);
  result -> y = LINEAR_TRANSFORM_Y(a.x, a.y, b.value);
  return true;
}
```

8.4. Path Assignments and Expressions

To assign a path to a variable with the right type, we use the following code:

Section: Assignment for Path Variables:

```
else if(type == TYPE_T_PATH){
  int i;
  struct path_variable result;
  void *(*alloc)(size_t);
  void (*disalloc)(void *);
  if(!eval_path_expression(mf, cx, begin_expression, *end, &result))
    return false;
  var = (struct symbolic_token *) begin; // 'var' begins in assigned variable
  for(i = 0; i < number_of_variables; i ++){
    struct path_variable *dst = (struct path_variable *) var -> var;
    if(dst -> permanent){ // How to allocate and dis-allocate data
      alloc = permanent_alloc;
      disalloc = permanent_free;
    }
    else{
      alloc = temporary_alloc;
      disalloc = temporary_free;
    }
    // If destiny is not empty, free the allocated memory:
    if(dst -> length != -1 && disalloc != NULL)
      path_recursive_free(disalloc, dst, false);
    // Copy origin to destiny:
    if(!recursive_copy_points(mf, NULL, alloc, &dst, &result, false))
      return false;
    var = (struct symbolic_token *) (var -> next); // 'var' points to '='
    var = (struct symbolic_token *) (var -> next); // 'var' points to next var
  }
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &result, false);
}
```

In next Subsection we will check the grammar rules about how to interpret path expressions.

8.4.1. Joining Paths

The grammar for path expressions begin with:

```
<Path Expression> -> <Pair Expression> | <Path Tertiary> |
                     <Path Subexpression><Direction Specifier> |
                     <Path Subexpression><Path Join> cycle
<Path Join> -> <Direction Specifier><Basic Join><Direction Specifier>
<Basic Join> -> & | .. | .. <Tension> .. | .. <Controls> .. | --
<Tension> -> tension <Tension Amount> |
             tension <Tension Amount> and <Tension Amount>
<Tension Amount> -> <Numeric Primary> | atleast <Numeric Primary>
<Controls> -> controls <Pair Primary> |
              controls <Pair Primary> and <Pair Primary>
<Direction Specifier> -> Empty |
                         { <Pair Expression } |
                         { <Numeric Expression> , <Numeric Expression> } |
                         { curl <Numeric Expression> }
<Path Subexpression> -> <Path Expression> |
```

<Path Subexpression><Path Join><Path Tertiary>

This requires the register of these new types of tokens:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_CYCLE,          // Symbolic token 'cycle'
TYPE_AMPERSAND,      // Symbolic token '&'
TYPE_JOIN,           // Symbolic token '..'
TYPE_TENSION,        // Symbolic token 'tension'
TYPE_AND,            // Symbolic token 'and'
TYPE_ATLEAST,        // Symbolic token 'atleast'
TYPE_CONTROLS,       // Symbolic token 'controls'
TYPE_CURL,           // Symbolic token 'curl'
TYPE_STRAIGHT_JOIN,  // Symbolic token '--'
```

And for each one we register a new correspondent keyword:

Section: List of Keywords (continuation):

```
"cycle", "&", "..", "tension", "and", "atleast", "controls", "curl", "--",
```

The first thing that we will define is how to count the number of joins in a path expression. For this we will use the following function:

Section: Local Function Declaration (metafont.c) (continuation):

```
int count_path_joins(struct generic_token *begin, struct generic_token *end);
```

The function works counting the number of "&", "–" and also counting the ".." that appear alone or that appear for the second time in a join that specifies tension or control points. A joining inside a sub-expression inside delimiters like parenthesis is not counted:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
int count_path_joins(struct generic_token *begin, struct generic_token *end){
int count = 0;
  DECLARE_NESTING_CONTROL();
  struct generic_token *p = begin;
  while(p != NULL){
    COUNT_NESTING(p);
    if(IS_NOT_NESTED()){
      if(p -> type == TYPE_AMPERSAND)
        count ++;
      else if(p -> type == TYPE_STRAIGHT_JOIN)
        count ++;
      else if(p -> type == TYPE_JOIN){
        struct generic_token *next = (struct generic_token *) p -> next;
        if(p == end || (next -> type !=  TYPE_TENSION &&
                        next -> type != TYPE_CONTROLS))
          count ++;
      }
    }
    if(p != end)
      p = (struct generic_token *) p -> next;
    else
      p = NULL;
  }
  return count;
```

```
}
```

Counting the number of joins is important for knowing how many points should we allocate for a path variable. After having how to obtain such information, we can begin dealing with path expressions. The function that evaluates them has the following header:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_path_expression(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct path_variable *result);
```

The number of points in a path variable will be initially set to the number of join operators plus 1:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_path_expression(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct path_variable *result){
  int expected_length;
  int number_of_joins = count_path_joins(begin, end);
  expected_length = number_of_joins + 1;
        <Section to be inserted: Path Expression: When There is No Joins>
        <Section to be inserted: Path Expression: Allocating Path Variable>
          <Section to be inserted: Path Expression: Iterating over Joins>
  // Paths are normalized as seen in Subsection 7.4.4:
  return normalize_path(mf, cx, result);
}
```

And what if the number of joins is zero? In this case we could have a pair expression like below:

```
path p;
p = (2, 3);
```

Or we could have a tertiary expression where the joins are inside delimiters like parenthesis:

```
path p;
p = ((2, 3)..(1, 2));
```

Anyway, we could have in the end zero or more direction specifiers, which should be ignored. In the first case because direction specifiers have no effect over isolated points. In the second because the path is normalized and already have all the control points when we finish to evaluate the parenthesis. Therefore, direction specifiers have no effect.

We can check if we have additional direction specifiers checking if the last token is "}". In this case, we change the position of the end of expression for the last token before the direction specifier and consider just this part as the expression to be evaluated. However, we first need to do some validation to ensure that the direction specifier follows the grammar rules:

Section: Path Expression: When There is No Joins:

```
if(number_of_joins == 0){
  if(end -> type == TYPE_CLOSE_BRACES){
    float dir_x, dir_y;
    struct generic_token *p = begin;
    DECLARE_NESTING_CONTROL();
    while(p != end){
      COUNT_NESTING(p);
      if(IS_NOT_NESTED() &&
```

```
        p -> next -> type == TYPE_OPEN_BRACES)
      break;
    p = p -> next;
  }
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
  // The specifier is ignored, but needs to be validated:
  if(!eval_direction_specifier(mf, cx, p -> next, end, &dir_x, &dir_y))
    return false;
  end = p;
 }
 result -> permanent = false;
 if(!eval_path_tertiary(mf, cx, begin, end, result))
   return false;
 return normalize_path(mf, cx, result);
}
```

If there are one or more joins, then we need to allocate the new points in our path variable. The number of allocated points is given by variable `expected_length`:

Section: Path Expression: Allocating Path Variable:

```
result -> points = (struct path_points *)
                    temporary_alloc(sizeof(struct path_points) *
                    expected_length);
if(result -> points == NULL){
  RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
  return false;
}
result -> length = 0; // Initialization
result -> permanent = false;
result -> number_of_points = 0;
result -> cyclic = false;
```

Now we will evaluate the joins. We need to iterate over all the tokens finding the joins in the format:

$$z_1 dje\_2$$

Therefore, we need variables to store the begin and end of each part while we perform the iteration. And then we create a loop where we will iterate over each join in the format above and mark the beginning and end token for each part:

Section: Path Expression: Iterating over Joins:

```
{
  struct generic_token *begin_z1, *end_z1 = NULL, *begin_z2, *end_z2;
  struct generic_token *begin_d = NULL, *end_d = NULL, *begin_e, *end_e;
  struct generic_token * begin_j, *end_j;
  struct path_points *z0_point = NULL, *z1_point = NULL, *z2_point = NULL;
  struct path_variable *z1_parent;
  begin_z1 = begin;
  end_z1 = begin_z1;
```

<Section to be inserted: Path Expression: Initial Value for z1>

```
  while(end_z1 != end || result -> length < expected_length){
```

<Section to be inserted: Path Expression: Delimit Join Tokens>

<Section to be inserted: Path Expression: Interpret Join Extremities>

<Section to be inserted: Path Expression: Interpret Direction Specifiers>

<Section to be inserted: Path Expression: Interpret the Join>

```
    begin_z1 = begin_z2;
    end_z1 = end_z2;
  }
```

<Section to be inserted: Path Expression: After the Joins>

```
}
```

The first element z_1, in the first iteration can be delimited choosing all tokens before the next "{", "&", "" or ".." in the expression, assuming that we are not inside parenthesis or brackets. In all other iterations, the element z_1 will be equal to what was element z_2 in the previous iteration, as seen in the previous code:

Section: Path Expression: Initial Value for z1:

```
{
  DECLARE_NESTING_CONTROL();
  int next_type;
  end_z1 = begin_z1;
  while(end_z1 != end){
    COUNT_NESTING(end_z1);
    next_type = end_z1 -> next -> type;
    if(IS_NOT_NESTED() &&
       (next_type == TYPE_OPEN_BRACES || next_type == TYPE_JOIN ||
        next_type == TYPE_AMPERSAND || next_type == TYPE_STRAIGHT_JOIN))
      break;
    end_z1 = (struct generic_token *) end_z1 -> next;
  }
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
}
```

Now we will delimit the first direction specifier. First we read the next token to check if we have a "{". If not, there is no first direction specifier. If so, then we delimit it until finding the closing "}":

Section: Path Expression: Delimit Join Tokens:

```
begin_d = end_z1 -> next;
if(begin_d -> type != TYPE_OPEN_BRACES){ // No specifier
  begin_d = NULL;
  end_d = NULL;
}
else{
  DECLARE_NESTING_CONTROL();
  if(begin_d == end){
    RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, OPTIONAL(begin_d -> line), '{');
    return false;
  }
  end_d = begin_d -> next;
  while(end_d != end){
    if(IS_NOT_NESTED() && end_d -> type == TYPE_CLOSE_BRACES)
      break;
    COUNT_NESTING(end_d);
    end_d = end_d -> next;
  }
  if(end_d -> type != TYPE_CLOSE_BRACES){
    RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, OPTIONAL(begin_d -> line), '{');
    return false;
  }
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin_d -> line));
```

```
}
```

Now we will delimit the join. If the join is "&" or a "–", then it is a single token. If the join begins with "..", then it could be a single token if the next token is not "`controls`" or "`tension`". But if the next token is these specifiers, then the join is delimited by the first ".." and the second "..":

Section: Path Expression: Delimit Join Tokens (continuation):

```
if(end_d == NULL)
  begin_j = end_z1 -> next; // Begins after z1
else
  begin_j = end_d -> next; // Or after direction specifier
end_j = begin_j; // We first assume that it is a single token
if(begin_j == end){ // But we need to have something after it
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
}
if(begin_j -> type == TYPE_JOIN && // Testing if it is more than 1 token
   (begin_j -> next -> type == TYPE_CONTROLS ||
    begin_j -> next -> type == TYPE_TENSION)){
  DECLARE_NESTING_CONTROL();
  end_j = end_j -> next;
  while(end_j != end){ // If so, the ending is next '..'
    COUNT_NESTING(end_j);
    if(IS_NOT_NESTED() && end_j -> type == TYPE_JOIN)
      break;
    end_j = end_j -> next;
  }
  if(end_j == end){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                   TYPE_T_PATH);
    return false;
  }
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
}
else if(begin_j -> type != TYPE_JOIN && begin_j -> type != TYPE_AMPERSAND &&
        begin_j -> type != TYPE_STRAIGHT_JOIN){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                 TYPE_T_PATH);
  return false;
}
```

The next direction delimiter may exist or not, depending if the next token is "{":

Section: Path Expression: Delimit Join Tokens (continuation):

```
begin_e = end_j -> next;
if(begin_e -> type != TYPE_OPEN_BRACES){ // No specifier
  begin_e = NULL;
  end_e = NULL;
} else{
  DECLARE_NESTING_CONTROL();
  if(begin_e == end){
    RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, OPTIONAL(begin_e -> line), '{');
    return false;
```

```
  }
  end_e = begin_e -> next;
  while(end_e != end){
    if(IS_NOT_NESTED() && end_e -> type == TYPE_CLOSE_BRACES)
      break;
    COUNT_NESTING(end_e);
    end_e = end_e -> next;
  }
  if(end_e -> type != TYPE_CLOSE_BRACES){
    RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, OPTIONAL(begin_e -> line), '{');
    return false;
  }
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin_e -> line));
}
```

Finally, the last join element is the next point or subpath joining the previous one. This is always a tertiary path expression. This last element will have its end delimited by an ampersand, opening braces or by a join token:

Section: Path Expression: Delimit Join Tokens (continuation):

```
{
  DECLARE_NESTING_CONTROL();
  if(end_e == NULL)
    begin_z2 = (struct generic_token *) end_j -> next;
  else
    begin_z2 = (struct generic_token *) end_e -> next;
  end_z2 = begin_z2;
  while(end_z2 != end){
    COUNT_NESTING(end_z2);
    if(IS_NOT_NESTED() &&
       (end_z2 -> next -> type == TYPE_OPEN_BRACES ||
        end_z2 -> next -> type == TYPE_JOIN ||
        end_z2 -> next -> type == TYPE_AMPERSAND ||
        end_z2 -> next -> type == TYPE_STRAIGHT_JOIN))
      break;
    end_z2 = end_z2 -> next;
  }
  if(end_z2 == end)
    COUNT_NESTING(end_z2);
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin_z2 -> line));
}
```

Once we delimited each part of the join, now we need to interpret the parts. First we will care about the first extremity point. For this, first we get the result of the first extremity point, given by a path tertiary expression. We copy it to the correct position in the path being interpreted.

Section: Path Expression: Initial Value for z1 (continuation):

```
{
  struct path_variable z1;
  if(!eval_path_tertiary(mf, cx, begin_z1, end_z1, &z1))
    return false;
  // If z1 is a single point:
  if(z1.length == 1 && z1.points[0].format != SUBPATH_FORMAT){
```

```
    result -> points[0].format = PROVISIONAL_FORMAT;
    result -> points[0].prov.x = z1.points[0].prov.x;
    result -> points[0].prov.y = z1.points[0].prov.y;
    result -> points[0].prov.dir1_x = result -> points[0].prov.dir1_y = NAN;
    result -> points[0].prov.dir2_x = result -> points[0].prov.dir2_y = NAN;
    result -> points[0].prov.tension1 = 1.0;
    result -> points[0].prov.tension2 = 1.0;
    result -> points[0].prov.atleast1 = false;
    result -> points[0].prov.atleast2 = false;
    result -> number_of_points ++;
  } else{ // If z1 is a subpath:
    result -> points[0].format = SUBPATH_FORMAT;
    if(!recursive_copy_points(mf, cx, temporary_alloc,
                              &(result -> points[0].subpath), &z1, true))
      return false;
    result -> number_of_points +=
                               result -> points[0].subpath -> number_of_points;
  }
  result -> cyclic = false;
  result -> length ++;
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &z1, false);
}
```

At the beginning of the iteration when we are iterating over the points to be interpreted, we will always have a variable z_1 which is a point or subpath that will begin as the first part of the path being joined with the others. And we will also have a variable z_2 which is what follows z_1 and it will be the value that in the next iteration will be assigned to z_1.

The code below interprets z_2. In this case, we may find a `cycle` token. If so, we must mark the path interpreted as cyclical and we must copy the starting point to this position. Otherwise, we conclude that the path is not cyclical and interpret z_1 in a similar way that we interpreted the first point z_0:

Section: Path Expression: Interpret Join Extremities:

```
if(begin_z2 == end_z2 && begin_z2 -> type == TYPE_CYCLE){ // Found 'cycle'
  struct path_points *p = result -> points;
  // The first point may be inside a subpath:
  while(p[0].format == SUBPATH_FORMAT)
    p = ((struct path_variable *) p[0].subpath) -> points;
  memcpy(&(result -> points[result -> length]), p, sizeof(struct path_points));
  result -> length ++;
  result -> number_of_points ++;
  result -> cyclic = true;
}
else{
  struct path_variable z2;
  if(!eval_path_tertiary(mf, cx, begin_z2, end_z2, &z2))
    return false;
  result -> cyclic = false;
  if(z2.length == 1 && z2.points[0].format != SUBPATH_FORMAT){ // z2 is point:
    result -> points[result -> length].format = PROVISIONAL_FORMAT;
    result -> points[result -> length].prov.x = z2.points[0].prov.x;
    result -> points[result -> length].prov.y = z2.points[0].prov.y;
    result -> points[result -> length].prov.dir1_x = NAN;
```

```
    result -> points[result -> length].prov.dir1_y = NAN;
    result -> points[result -> length].prov.dir2_x = NAN;
    result -> points[result -> length].prov.dir2_y = NAN;
    result -> points[result -> length].prov.tension1 = 1.0;
    result -> points[result -> length].prov.tension2 = 1.0;
    result -> points[result -> length].prov.atleast1 = false;
    result -> points[result -> length].prov.atleast2 = false;
    result -> number_of_points ++;
  }
  else{ // z2 is subpath:
    result -> points[result -> length].format = SUBPATH_FORMAT;
    if(!recursive_copy_points(mf, cx, temporary_alloc,
                              &(result -> points[result -> length].subpath),
                              &z2, true))
      return false;
    result -> number_of_points +=
           result -> points[result -> length].subpath -> number_of_points;
  }
  result -> length ++;
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &z2, false);
}
```

Notice that a code like below is perfectly valid:

```
path p;
p = (0, 0) .. (1, 2) .. cycle .. (2, 4) .. cycle .. cycle;
```

Each point different than `cycle` makes the interpreter conclude that the path is not cyclical, and each `cycle` makes him reach the opposite conclusion. So is only the last element of the join that determines whether the path will be treated as cyclical or not. In other cases, this token is just a synonym for the first point.

The code we have written so far ensures that all extremity points will be correctly stored in the resulting variable. But we won't store only them, but we also store information about how they are joined, given control points, tension and direction specifiers. To fill in this information, we need a pointer that shows what are the exact values of the extremity points that we are joining (points z_1 and z_2) together with the previous point if it exist (point z_0). At first, we could obtain them directly at `result -> points[result -> length - 2]` and `result -> points[result -> length - 1]` for points z_1 and z_2. However, these values may not be single points, they could be sub-paths. In this case, to find the true extremity points, we have to walk through the subpath:

Section: Path Expression: Interpret Join Extremities (continuation):

```
// z0 is the previous point, z1 is the current one and z2 is the next one
z1_point = &(result -> points[result -> length - 2]);
z1_parent = result;
z0_point = NULL;
while(z1_point -> format == SUBPATH_FORMAT){
  struct path_variable *p = (struct path_variable *) z1_point -> subpath;
  z1_point = &(p -> points[p -> length - 1]);
  z1_parent = p;
  if(p -> length  != 1 && z1_point -> format != SUBPATH_FORMAT)
    z0_point = &(p -> points[p -> length - 2]);
}
if(z0_point == NULL && result -> length > 2){
```

```
  z0_point = &(result -> points[result -> length - 3]);
  while(z0_point -> format == SUBPATH_FORMAT){
    struct path_variable *p = (struct path_variable *) z0_point -> subpath;
    z0_point = &(p -> points[p -> length - 1]);
  }
}
z2_point = &(result -> points[result -> length - 1]);
while(z2_point -> format == SUBPATH_FORMAT){
  struct path_variable *p = (struct path_variable *) z2_point -> subpath;
  z2_point = &(p -> points[0]);
}
```

Now we need to interpret both direction specifiers. They could be empty (which is equivalent to have direction $(0, 0)$), they could be a pair specifying a direction vector or they could be a numeric "curl" value.

A function that evaluates direction specifiers is called and places the read values in our current point:

Section: Path Expression: Interpret Direction Specifiers:

```
if(!eval_direction_specifier(mf, cx, begin_d, end_d,
                             &(z1_point -> prov.dir1_x),
                             &(z1_point -> prov.dir1_y)))
  return false;
if(!eval_direction_specifier(mf, cx, begin_e, end_e,
                             &(z1_point -> prov.dir2_x),
                             &(z1_point -> prov.dir2_y)))
  return false;
```

The function that evaluates direction specifiers is declared here:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_direction_specifier(struct metafont *mf, struct context *cx,
                              struct generic_token *begin,
                              struct generic_token *end, float *w_x,
                              float *w_y);
```

And the function works checking for 4 different cases: when there is no direction specifier, when it is a curl specification, when it is a direction specified by two numeric values and when it is a direction specified by a pair:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_direction_specifier(struct metafont *mf, struct context *cx,
                              struct generic_token *begin,
                              struct generic_token *end, float *w_x,
                              float *w_y){
  // Detecting error of empty specifier '{}':
  if(begin != NULL && begin -> next == end){
    RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '{');
    return false;
  }
                  <Section to be inserted: Direction Specifier: Case 1>
                  <Section to be inserted: Direction Specifier: Case 2>
                  <Section to be inserted: Direction Specifier: Case 3>
                  <Section to be inserted: Direction Specifier: Case 4>
  return false;
}
```

When a specifier does not exist, we store nothing and return, letting the values indefinite:

Section: Direction Specifier: Case 1:

```
if(begin == NULL || end == NULL){
  return true;
}
```

Next we check if we have "$\{curl\gamma\}$" where γ is a numeric expression. In this case, we store γ in the y coordinate of the direction specifier and keep the x coordinate as Not-a-Number:

Section: Direction Specifier: Case 2:

```
if(begin -> next -> type == TYPE_CURL){
  struct numeric_variable gamma;
  struct generic_token *begin_n, *end_n;
  begin_n = begin -> next -> next;
  end_n = begin_n;
  if(end_n == end){
    RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(end_n -> line),
                                   TYPE_T_NUMERIC);
    return false;
  }
  while(end_n -> next != end)
    end_n = end_n -> next;
  if(!eval_numeric_expression(mf, cx, begin_n, end_n, &gamma))
    return false;
  *w_x = NAN;
  *w_y = gamma.value;
  return true;
}
```

Now we will check if we have the case "$\{a, b\}$" where a and b are numbers. To check for this case, we will try to delimit a and b searching for the comma. We will interpret case 3 only if the comma is found:

Section: Direction Specifier: Case 3:

```
DECLARE_NESTING_CONTROL();
struct generic_token *begin_a, *end_a, *begin_b = NULL, *end_b;
begin_a = (struct generic_token *) begin -> next;
end_a = begin_a;
while(end_a -> next != end){
  COUNT_NESTING(end_a);
  if(IS_NOT_NESTED() && end_a -> next -> type == TYPE_COMMA){
    begin_b = (struct generic_token *) end_a -> next -> next;
    break;
  }
  end_a = (struct generic_token *) end_a -> next;
}
if(begin_b != NULL){
  struct numeric_variable a, b;
  end_b = begin_b;
  while(end_b -> next != end)
    end_b = (struct generic_token *) end_b -> next;
  if(!eval_numeric_expression(mf, cx, begin_a, end_a, &a))
    return false;
  if(!eval_numeric_expression(mf, cx, begin_b, end_b, &b))
```

```
    return false;
  *w_x = a.value;
  *w_y = b.value;
  return true;
}
```

And finally, the last case, where we have an element $\{a\}$, where a is a pair expression. To deal with this case, recall that in the previous case we already delimited a. If a comma was not found, we did not delimited b, but still delimited a. Therefore, we can interpret the tokens as a pair expression:

Section: Direction Specifier: Case 4:

```
else{ // if in the previous case there was no comma
  COUNT_NESTING(end_a);
  RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin_a -> line));
  struct pair_variable a;
  if(!eval_pair_expression(mf, cx, begin_a, end_a, &a))
    return false;
  *w_x = a.x;
  *w_y = a.y;
  return true;
}
```

With the code we have written so far we are populating in the resulting path all direction specifiers explicitly written in the path expression. However, when no specifier explicit is placed, there are some rules that allow us to deduce implicit values for them.

Firstly, when we have a concatenation (`&`), a straight line (`--`) or two identical points in succession, the direction specifier is irrelevant in that segment. In the first case, the endpoints being concatenated are equal, therefore there is no direction between them. In the second case, the segment will be directly stored in the final format as a straight line, without needing a direction specifier. In the third case, we have a redundant point being repeated twice and there is no space between two copies of the same point. However, we must place an implicit `curl 1` before and after the points being joined. Basically we transform the expressions below:

```
z0 .. z1 & z2 .. z3
z0 .. z1 -- z2 .. z3
```

In the expressions:

```
z0 .. {curl 1}z1 & z2{curl 1} .. z3
z0 .. {curl 1}z1 -- z2{curl 1} .. z3
```

And this is done by the code below:

Section: Path Expression: Interpret Direction Specifiers (continuation):

```
if((begin_j == end_j && (begin_j -> type == TYPE_AMPERSAND ||
                         begin_j -> type == TYPE_STRAIGHT_JOIN)) ||
   (z1_point -> prov.x == z2_point -> prov.x &&
    z1_point -> prov.y == z2_point -> prov.y)){
  if(z0_point != NULL && z0_point -> format == PROVISIONAL_FORMAT &&
     isnan(z0_point -> prov.dir2_y)){
    z0_point -> prov.dir2_x = NAN;
    z0_point -> prov.dir2_y = 1.0;
  }
  if(z2_point -> format == PROVISIONAL_FORMAT &&
     isnan(z2_point -> prov.dir1_y)){
    z2_point -> prov.dir1_x = NAN;
```

```
    z2_point -> prov.dir1_y = 1.0;
  }
  // If z1 == z2, we also place explicit control points if they do not exit:
  if(z1_point -> prov.x == z2_point -> prov.x &&
     z1_point -> prov.y == z2_point -> prov.y &&
     z1_point -> format == PROVISIONAL_FORMAT){
    z1_point -> format = FINAL_FORMAT;
    z1_point -> point.u_x = z1_point -> point.v_x = z1_point -> point.x;
    z1_point -> point.u_y = z1_point -> point.v_y = z1_point -> point.y;
  }
}
```

Another implicit deduction for direction specifiers occurs when either the previous or the next point is in final format with explicit control points. Then, we convert the expressions below:

```
z0 .. controls u and v .. z1 .. z2 .. z3
z0 ..  z1 .. z2 .. controls u and v ..z3
```

Interpreting them as:

```
z0 .. controls u and v .. z1{z1-v} .. z2 ..z3
z0 ..  z1 .. {u-z2}z2 .. controls u and v ..z3
```

We deal with this case when we read explicit control points in the joining:

Section: Path Expression: Found Explicit Control Points:

```
if(z0_point != NULL && z0_point -> format == PROVISIONAL_FORMAT &&
   isnan(z0_point -> prov.dir2_y)){
  z0_point -> prov.dir2_x = z1_point -> point.u_x - z1_point -> point.x;
  z0_point -> prov.dir2_y = z1_point -> point.u_y - z1_point -> point.y;
  if(z0_point -> prov.dir2_x == 0.0 && z0_point -> prov.dir2_y == 0.0){
    z0_point -> prov.dir2_x = NAN;
    z0_point -> prov.dir2_y = 1.0;
  }
}
if(z2_point -> format == PROVISIONAL_FORMAT && isnan(z2_point -> prov.dir1_y)){
  z2_point -> prov.dir1_x = z2_point -> prov.x - z1_point -> point.v_x;
  z2_point -> prov.dir1_y = z2_point -> prov.y - z1_point -> point.v_y;
  if(z2_point -> prov.dir1_x == 0.0 && z2_point -> prov.dir1_y == 0.0){
    z2_point -> prov.dir1_x = NAN;
    z2_point -> prov.dir1_y = 1.0;
  }
}
```

Direction specifiers also are implicitly copied from one side of a given point to the other. This means that the expressions below:

```
z0 .. {w1}z1 .. z2 ..z3
z0 .. z1{w1} .. z2 ..z3
```

Would be interpreted as:

```
z0 .. {w1}z1{w1} .. z2 ..z3
z0 .. {w1}z1{w1} .. z2 ..z3
```

And this is done by the code below:

Section: Path Expression: Interpret Direction Specifiers (continuation):

```
if(!isnan(z1_point -> prov.dir2_y) &&
   z2_point -> format == PROVISIONAL_FORMAT && isnan(z2_point -> prov.dir1_y)){
  z2_point -> prov.dir1_x = z1_point -> prov.dir2_x;
  z2_point -> prov.dir1_y = z1_point -> prov.dir2_y;
}
if(z0_point != NULL && isnan(z0_point -> prov.dir2_y) &&
   !isnan(z1_point -> prov.dir1_y)){
  z0_point -> prov.dir2_x = z1_point -> prov.dir1_x;
  z0_point -> prov.dir2_y = z1_point -> prov.dir1_y;
}
```

Now we will finally interpret the join itself. A join may be in one of the following formats:

```
z1 & z2
z1 -- z2
z1 .. controls u and v .. z2
z1 .. tension a and b .. z2
z1 .. z2
```

If our join is a concatenation, characterized by the "&" token, we are concatenating tow points or (more probably) two sub-paths. In this case we should check is the two joining points occupy the same place (which means having a distance lesser than 0.00002). If not, we should raise an error. If so, we join both paths removing the first copy of the point where the join is happening.

Section: Path Expression: Interpret the Join:

```
if(begin_j == end_j && begin_j -> type == TYPE_AMPERSAND){
  double dif_x = z1_point -> prov.x - z2_point -> prov.x;
  double dif_y = z1_point -> prov.y - z2_point -> prov.y;
  if(hypot(dif_x, dif_y) > 0.00002){
    RAISE_ERROR_DISCONTINUOUS_PATH(mf, cx, OPTIONAL(begin_j -> line),
                                   z1_point -> prov.x, z1_point -> prov.y,
                                   z2_point -> prov.x, z2_point -> prov.y);
    return false;
  }
  // This will overwrite the first extremity point
  result -> number_of_points --;
  if(z1_parent != result){
    z1_parent -> length --;
    z1_parent -> number_of_points --;
  }
  z1_point -> point.x = NAN;
  z1_point -> point.y = NAN;
}
```

The next joining case will be when the points are connected by a straight line. In this case, our point will have its format modified to the final one and control points will be placed on the same line segment delimited by extremity points:

Section: Path Expression: Interpret the Join (continuation):

```
else if(begin_j == end_j && begin_j -> type == TYPE_STRAIGHT_JOIN){
  z1_point -> format = FINAL_FORMAT;
  z1_point -> point.u_x = z1_point -> point.x + (1.0/3.0) *
                          (z2_point -> prov.x - z1_point -> point.x);
  z1_point -> point.u_y = z1_point -> point.y + (1.0/3.0) *
                          (z2_point -> prov.y - z1_point -> point.y);
```

```
  z1_point -> point.v_x = z1_point -> point.x + (2.0/3.0) *
                          (z2_point -> prov.x - z1_point -> point.x);
  z1_point -> point.v_y = z1_point -> point.y + (2.0/3.0) *
                          (z2_point -> prov.y - z1_point -> point.y);
}
```

If the join is simple, without any information, it will be interpreted as equivalent to having a tension equal to 1. The two expressions below are equivalent:

```
z1 .. z2
z1 .. tension 1 and 1 .. z2
```

Storing the tension in this case is performed by the code below:

Section: Path Expression: Interpret the Join (continuation):

```
else if(begin_j == end_j && begin_j -> type == TYPE_JOIN){
  z1_point -> prov.tension1 = 1.0;
  z1_point -> prov.tension2 = 1.0;
}
```

The next case is when we have explicit control points. There are two different formats:

```
z1 .. controls u and v .. z2
z1 .. controls u .. z2
```

The second case is equivalent to having both control points with the same value. The way we differentiate both cases is due to the token `and`:

Section: Path Expression: Interpret the Join (continuation):

```
else if(begin_j -> type == TYPE_JOIN && begin_j != end_j &&
        begin_j -> next -> type == TYPE_CONTROLS){
  DECLARE_NESTING_CONTROL();
  struct generic_token *begin_u, *end_u, *begin_v = NULL, *end_v = NULL;
  struct pair_variable u, v;
  if(begin_j -> next == end_j || begin_j -> next -> next == end_j){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                   TYPE_T_PATH);
    return false;
  }
  begin_u = begin_j -> next -> next;
  end_u = begin_u;
  while(end_u -> next != end_j){
    COUNT_NESTING(end_u);
    if(IS_NOT_NESTED() && end_u -> next -> type == TYPE_AND)
      break;
    end_u = end_u -> next;
  }
  if(end_u -> next != end_j){
    if(end_u -> next -> next == end_j){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                     TYPE_T_PATH);
      return false;
    }
    begin_v = end_u -> next -> next;
    end_v = begin_v;
    while(end_v -> next != end_j)
```

```
      end_v = end_v -> next;
  }
  if(!eval_pair_primary(mf, cx, begin_u, end_u, &u))
    return false;
  z1_point -> format = FINAL_FORMAT;
  z1_point -> point.u_x = u.x;
  z1_point -> point.u_y = u.y;
  if(begin_v != NULL){
    if(!eval_pair_primary(mf, cx, begin_v, end_v, &v))
      return false;
    z1_point -> point.v_x = v.x;
    z1_point -> point.v_y = v.y;
  }
  else{
    z1_point -> point.v_x = u.x;
    z1_point -> point.v_y = u.y;
  }
```

<Section to be inserted: **Path Expression: Found Explicit Control Points**>

```
}
```

And finally, the junction can be in the tension description format. These are the different possibilities:

```
z1 .. tension t0 .. z2
z1 .. tension atleast t0 .. z2
z1 .. tension t0 and t1 .. z2
z1 .. tension atleast t0 and t1 .. z2
z1 .. tension t0 and atleast t1 .. z2
z1 .. tension atleast t0 and atleast t1 .. z2
```

The code that evaluates the join, given a tension description is given below:

Section: Path Expression: Interpret the Join (continuation):

```
else if(begin_j -> type == TYPE_JOIN && begin_j != end_j &&
        begin_j -> next -> type == TYPE_TENSION){
  DECLARE_NESTING_CONTROL();
  struct generic_token *begin_t0, *end_t0, *begin_t1 = NULL, *end_t1 = NULL;
  struct numeric_variable t0, t1;
  if(begin_j -> next == end_j || begin_j -> next -> next == end_j){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                   TYPE_T_PATH);
    return false;
  }
  begin_t0 = begin_j -> next -> next;
  z1_point -> prov.atleast1 = (begin_t0 -> type == TYPE_ATLEAST);
  if(begin_t0 -> type == TYPE_ATLEAST){
    begin_t0 = begin_t0 -> next;
    if(begin_t0 == end_j){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                     TYPE_T_PATH);
      return false;
    }
  }
  end_t0 = begin_t0;
```

```
while(end_t0 -> next != end_j){
  COUNT_NESTING(end_t0);
  if(IS_NOT_NESTED() && end_t0 -> next -> type == TYPE_AND)
    break;
  end_t0 = end_t0 -> next;
}
if(end_t0 -> next != end_j){
  if(begin_t0 -> next -> next == end_j){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                   TYPE_T_PATH);
    return false;
  }
  begin_t1 = end_t0 -> next -> next;
  z1_point -> prov.atleast2 = (begin_t1 -> type == TYPE_ATLEAST);
  if(begin_t1 -> type == TYPE_ATLEAST){
    begin_t1 = (struct generic_token *) begin_t1 -> next;
    if(begin_t1 == end_j){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_j -> line),
                                     TYPE_T_PATH);
      return false;
    }
  }
  end_t1 = begin_t1;
  while(end_t1 -> next != end_j)
    end_t1 = end_t1 -> next;
}
if(!eval_numeric_primary(mf, cx, begin_t0, end_t0, &t0))
  return false;
z1_point -> prov.tension1 = t0.value;
if(begin_t1 != NULL){
  if(!eval_numeric_primary(mf, cx, begin_t1, end_t1, &t1))
    return false;
  z1_point -> prov.tension2 = t1.value;
}
else{
  z1_point -> prov.atleast2 = z1_point -> prov.atleast1;
  z1_point -> prov.tension2 = z1_point -> prov.tension1;
}
if(z1_point -> prov.tension1 < 0.75){
  RAISE_ERROR_INVALID_TENSION(mf, cx, OPTIONAL(begin_t0 -> line),
                              z1_point -> prov.tension1, 0,
                              z1_point -> prov.x, z1_point -> prov.y,
                              z2_point -> prov.x, z2_point -> prov.y);
  return false;
}
if(z1_point -> prov.tension2 < 0.75){
  RAISE_ERROR_INVALID_TENSION(mf, cx, OPTIONAL(begin_t0 -> line),
                              z1_point -> prov.tension2, 1,
                              z1_point -> prov.x, z1_point -> prov.y,
                              z2_point -> prov.x, z2_point -> prov.y);
  return false;
```

```
  }
}
```

Like seen in Subsection 7.4.3, the tension value cannot be smaller than 3/4, otherwise the system of equations that determines the curve shape could have no solution.

Once we populate the resulting path with all tension values, joinings and directions, there is just a little more operations that we need to do before normalize and return the expression result. First, we will adjust our pointer z_0 to point to the first point in the path. The pointers z_1 and z_2 we will not change: after going through all iterations, z_1 is the last but one point and z_2 is the last one:

Section: Path Expression: After the Joins:

```
// z0 is now the first point, z1 is the last but one and z2 is the last one
z0_point = &(result -> points[0]);
while(z0_point -> format == SUBPATH_FORMAT){
  struct path_variable *p = (struct path_variable *) z0_point -> subpath;
  z0_point = &(p -> points[0]);
}
```

Now after we have iterated over all the joins, it is possible that we still did not interpret everything in the expression. That's because we can have a last direction specifier on the right side of the last point. If it exists, it needs to be validated. On a non-cyclic path, it will be copied to the left side of that point if there is no direction specifier. In a cyclical path, this can also happen, but the specifier will also have an effect on the point where he is:

Section: Path Expression: After the Joins (continuation):

```
if(end_z1 != end){
  float w_x = NAN, w_y = NAN;
  if(!eval_direction_specifier(mf, cx, end_z1 -> next, end, &w_x, &w_y))
    return false;
  if(z1_point -> format == PROVISIONAL_FORMAT &&
     isnan(z1_point -> prov.dir2_y)){
    z1_point -> prov.dir2_x = w_x;
    z1_point -> prov.dir2_y = w_y;
  }
  if(result -> cyclic){
    z1_point -> prov.dir1_x = w_x;
    z1_point -> prov.dir1_y = w_y;
  }
}
```

Now let's apply some additional rules to detect implicit direction specifiers, which can only be made after we read the entire expression.

First, if we have a non-cyclical path, if the first and last direction specifier is empty, we replace it by `curl 1`. This means that the following two expressions are equivalent:

```
p = z1 .. z2 .. z3;
p = z1{curl 1} .. z2 .. {curl 1}z3;
```

Besides this, the last point of a non-cyclical path do not have anything after it. Therefore, we convert it to the final format, with the control points being equal the point coordinate:

Section: Path Expression: After the Joins:

```
if(!(result -> cyclic)){
  if(z0_point -> format == PROVISIONAL_FORMAT &&
     isnan(z0_point -> prov.dir1_y)){
    z0_point -> prov.dir1_x = NAN;
    z0_point -> prov.dir1_y = 1.0;
```

```
  }
  if(z1_point -> format == PROVISIONAL_FORMAT &&
    isnan(z1_point -> prov.dir2_y)){
    z1_point -> prov.dir2_x = NAN;
    z1_point -> prov.dir2_y = 1.0;
  }
  if(z2_point -> format == PROVISIONAL_FORMAT){
    z2_point -> format = FINAL_FORMAT;
    z2_point -> point.u_x = z2_point -> point.v_x = z2_point -> point.x;
    z2_point -> point.u_y = z2_point -> point.v_y = z2_point -> point.y;
  }
}
```

If we have a cyclical path, the first and last points are equal. After evaluating all the expressions, we can guarantee that the coordinates on the first and last points are matching. However, they could be in a different format. If the first point is in final format and the last one is not, we transform the last point to final form too, copying the control points:

Section: Path Expression: After the Joins (continuation):

```
if(result -> cyclic && z0_point -> format == FINAL_FORMAT){
  memcpy(z2_point, z0_point, sizeof(struct path_points));
  if(z1_point -> format == PROVISIONAL_FORMAT &&
     isnan(z1_point -> prov.dir2_y)){
    z1_point -> prov.dir2_x = z2_point -> point.u_x - z2_point -> point.x;
    z1_point -> prov.dir2_y = z2_point -> point.u_y - z2_point -> point.y;
    if(z1_point -> prov.dir2_x == 0.0 && z1_point -> prov.dir2_y == 0.0){
      z1_point -> prov.dir2_x = NAN;
      z1_point -> prov.dir2_x = 1.0;
    }
  }
}
```

If both points are in provisional format, we copy all non-empty direction specifiers from one to the other. If there is a conflict, as in the example below, the information from the first point will have priority over the last:

```
% Example of conflict:
p = (0, 0){1, 2} .. (1, 3) .. cycle{1, 3};
% Like in METAFONT language, this is interpreted as:
p = (0, 0){1, 2} .. (1, 3) .. {1, 3}cycle{1, 2};
```

This is the code that makes the two points consistent:

Section: Path Expression: After the Joins (continuation):

```
else if(result -> cyclic){
  if(!isnan(z0_point -> prov.dir1_y)){
    z2_point -> prov.dir1_x = z0_point -> prov.dir1_x;
    z2_point -> prov.dir1_y = z0_point -> prov.dir1_y;
    if(z1_point -> format == PROVISIONAL_FORMAT &&
       isnan(z1_point -> prov.dir2_y)){
      z1_point -> prov.dir2_x = z2_point -> prov.dir1_x;
      z1_point -> prov.dir2_y = z2_point -> prov.dir1_y;
    }
  }
  else{
```

```
    z0_point -> prov.dir1_x = z2_point -> prov.dir1_x;
    z0_point -> prov.dir1_y = z2_point -> prov.dir1_y;
  }
  z2_point -> prov.dir2_x = z0_point -> prov.dir2_x;
  z2_point -> prov.dir2_y = z0_point -> prov.dir2_y;
}
```

This finalizes how a curve should be built given its points and subpaths. Now we will see how we read the points and subpaths.

8.4.2. Tertiary Path Expressions

The grammar for tertiary path expressions is:

```
<Path Tertiary> -> <Pair Tertiary> | <Path Secondary>
```

And it is only this. Therefore, to interpret a tertiary path expression, we should walk over the expression and check if we find a tertiary pair expression. If so, we should interpret the entire expression like a pair expression. Otherwise, we interpret as a secondary path expression. If we interpret the expression like a pair, we should convert the result from a pair to a path with a single point.

Anyway, after interpreting the expression and obtain a path as result, we return the result.

The function declaration is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_path_tertiary(struct metafont *mf, struct context *cx,
                        struct generic_token *begin,
                        struct generic_token *end,
                        struct path_variable *result);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_path_tertiary(struct metafont *mf, struct context *cx,
                        struct generic_token *begin,
                        struct generic_token *end,
                        struct path_variable *result){
  struct generic_token *p, *prev = NULL;
  bool this_expression_is_pair = false;
  DECLARE_NESTING_CONTROL();
  p = begin;
  do{
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && IS_VALID_SUM_OR_SUB(prev, p)){
      this_expression_is_pair = true;
      break;
    }
    prev = p;
    if(p != end)
      p = p -> next;
    else
      p = NULL;
  }while(p != NULL);
  if(this_expression_is_pair){
    struct pair_variable pair;
    if(!eval_pair_expression(mf, cx, begin, end, &pair))
      return false;
```

```
    result -> cyclic = false;
    result -> length = 1;
    result -> number_of_points = 1;
    result -> points = (struct path_points *)
                         temporary_alloc(sizeof(struct path_points));
    if(result -> points == NULL){
      RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
      return false;
    }
    result -> points -> format = FINAL_FORMAT;
    result -> points -> point.x = pair.x;
    result -> points -> point.y = pair.y;
    result -> points -> point.u_x = pair.x;
    result -> points -> point.u_y = pair.y;
    result -> points -> point.v_x = pair.x;
    result -> points -> point.v_y = pair.y;
    return true;
  }
  else
    return eval_path_secondary(mf, cx, begin, end, result);
}
```

8.4.3. Secondary Path Expressions: Transformers

The grammar for secondary path expressions is:

```
<Path Secondary> -> <Pair Secondary> | <Path Primary> |
                    <Path Secondary><Transformer>
```

The transformers are the same that were presented for pair expressions plus the `transformed` operator introduced in the section about transforms. The first thing that we should test in theses expressions is if we have a transformer at the end of the expression. If so, then we should apply the last rule in the grammar above. If not, we should check if we have a secondary pair operator (multiplication or division). If so, we apply the second rule, treating the path secondary expression as a pair secondary. Otherwise we apply the first rule and interpret the expression as a primery path expression.

Applying a transformer to rotate, scale and others means applying the transformation in each extremity point and each control point in the path.

The declaration for the function that evaluates path secondary expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_path_secondary(struct metafont *mf, struct context *cx,
                         struct generic_token *begin,
                         struct generic_token *end,
                         struct path_variable *result);
```

And the implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_path_secondary(struct metafont *mf, struct context *cx,
                         struct generic_token *begin,
                         struct generic_token *end,
                         struct path_variable *result){
  bool have_transform = false, have_pair_operator = false;
  struct generic_token *p, *last_fraction = NULL, *transform_op = NULL,
                       *before_transform = NULL, *prev = NULL,
```

```
                      *prev_prev = NULL;
DECLARE_NESTING_CONTROL();
p = begin;
do{
  COUNT_NESTING(p);
  if(IS_NOT_NESTED() && (p -> type == TYPE_MULTIPLICATION ||
           p -> type == TYPE_DIVISION || p -> type == TYPE_ROTATED ||
           p -> type == TYPE_SCALED || p -> type == TYPE_SHIFTED ||
           p -> type == TYPE_SLANTED || p -> type == TYPE_XSCALED ||
           p -> type == TYPE_YSCALED || p -> type == TYPE_ZSCALED ||
           p -> type == TYPE_TRANSFORMED)){
    if(p -> type == TYPE_DIVISION && prev -> type == TYPE_NUMERIC &&
      p != end && p -> next -> type != TYPE_NUMERIC &&
       last_fraction != prev_prev) // Fraction separator
       last_fraction = p;
     else if(p -> type == TYPE_DIVISION || p -> type == TYPE_MULTIPLICATION)
       have_pair_operator = true;
     else{
       have_transform = true;
       transform_op = p;
       before_transform = prev;
     }
  }
  prev_prev = prev;
  prev = p;
  if(p != end)
    p = p -> next;
  else
    p = NULL;
}while(p != NULL);
RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
if(have_transform){
  if(transform_op -> next == NULL){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PATH);
    return false;
  }
  if(!eval_path_secondary(mf, cx, begin, before_transform, result))
    return false;
                <Section to be inserted: Path Transform: Rotate>
                 <Section to be inserted: Path Transform: Scale>
                 <Section to be inserted: Path Transform: Shift>
                 <Section to be inserted: Path Transform: Slant>
                <Section to be inserted: Path Transform: X-Scale>
                <Section to be inserted: Path Transform: Y-Scale>
                <Section to be inserted: Path Transform: Z-Scale>
           <Section to be inserted: Path Transform: Generic Transform>
}
else if(have_pair_operator){
  struct pair_variable pair;
  if(!eval_pair_secondary(mf, cx, begin, end, &pair))
```

```
      return false;
    result -> cyclic = false;
    result -> length = 1;
    result -> number_of_points = 1;
    result -> points = (struct path_points *)
                        temporary_alloc(sizeof(struct path_points));
    if(result -> points == NULL){
      RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
      return false;
    }
    result -> points -> format = FINAL_FORMAT;
    result -> points[0].point.x = pair.x;
    result -> points[0].point.y = pair.y;
    result -> points[0].point.u_x = pair.x;
    result -> points[0].point.u_y = pair.y;
    result -> points[0].point.v_x = pair.x;
    result -> points[0].point.v_y = pair.y;
    return true;
  }
  else
    return eval_path_primary(mf, cx, begin, end, result);
}
```

In the code above, when we check if we have a transformer in the expression, we always store the last transformer in the variable `transform_op` and the last token before the transformer in the variable `before_transform`. This allow us to divide the expression in its parts and interpret the code to obtain the path to be transformed, as is shown above.

We can assume that every time we apply a transformer, we will do so on an already normalized path, that is, without recursions and with all points in the final format. This is because normalization happens in the when the `eval_path_expression` function returns. And the transformation in the code above will always occur on the result of path primary expressions. In the primary expressions we will have variables (which to have been stored, were returned by `eval_path_expression`) or sub-expressions (which are interpreted recursively by a `eval_path_expression`). Fortunately, this will limit the complexity of the code that handles such operations.

If we have a rotation, to interpret the totation transformer after we got the path to be rotated in the `result` variable, we can use the code below:

Section: Path Transform: Rotate:

```
if(transform_op -> type == TYPE_ROTATED){
  struct numeric_variable a;
  double theta, sin_theta, cos_theta;
  if(!eval_numeric_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  theta = 0.0174533 * a.value; // Converting degrees to radians
  sin_theta = sin(theta);
  cos_theta = cos(theta);
  path_rotate(result, sin_theta, cos_theta);
  return true;
}
```

The function that performs the rotation:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_rotate(struct path_variable *p, double sin_theta,
```

```
                double cos_theta);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_rotate(struct path_variable *p, double sin_theta,
                double cos_theta){
  int i;
  for(i = 0; i < p -> length; i ++){
    double x = p -> points[i].prov.x, y = p -> points[i].prov.y;
    p -> points[i].prov.x = x * cos_theta - y * sin_theta;
    p -> points[i].prov.y = x * sin_theta + y * cos_theta;
    x = p -> points[i].point.u_x;
    y = p -> points[i].point.u_y;
    p -> points[i].point.u_x = x * cos_theta - y * sin_theta;
    p -> points[i].point.u_y = x * sin_theta + y * cos_theta;
    x = p -> points[i].point.v_x;
    y = p -> points[i].point.v_y;
    p -> points[i].point.v_x = x * cos_theta - y * sin_theta;
    p -> points[i].point.v_y = x * sin_theta + y * cos_theta;
  }
}
```

Now we will interpret the scaling operator. It shoult interpret a numeric value and then multiply all points in the path by such value:

Section: Path Transform: Scale:

```
if(transform_op -> type == TYPE_SCALED){
  struct numeric_variable a;
  if(!eval_numeric_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  path_xyscale(result, a.value, a.value);
  return true;
}
```

The function that iterates over path points changing the scale:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_xyscale(struct path_variable *p, float x, float y);
```

Notice that by the function signature we can pass different values to stretch the path horizontally and vertically. This transformer stretches the path always in the same proportion, but other transformers will also use this function to stretch not preserving proportions. Its implementation is:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_xyscale(struct path_variable *p, float x, float y){
  int i;
  for(i = 0; i < p -> length; i ++){
    p -> points[i].point.x *= x;
    p -> points[i].point.y *= y;
    p -> points[i].point.u_x *= x;
    p -> points[i].point.u_y *= y;
    p -> points[i].point.v_x *= x;
    p -> points[i].point.v_y *= y;
  }
}
```

The next transformer translate, or shift a given path. We first read a pair and this pair sets how the path should be shifted in the x and y axis:

Section: Path Transform: Shift:

```
if(transform_op -> type == TYPE_SHIFTED){
  struct pair_variable a;
  if(!eval_pair_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  path_shift(result, a.x, a.y);
  return true;
}
```

And the recursive function that performs the shifting:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_shift(struct path_variable *p, float x, float y);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_shift(struct path_variable *p, float x, float y){
  int i;
  for(i = 0; i < p -> length; i ++){
    p -> points[i].point.x += x;
    p -> points[i].point.y += y;
    p -> points[i].point.u_x += x;
    p -> points[i].point.u_y += y;
    p -> points[i].point.v_x += x;
    p -> points[i].point.v_y += y;
  }
}
```

Now to the slanting transformer. This transformer shifts to the right the points based on how above the origin they are in tha y axis and shifts to the left based on how below the origin they are in the y axis:

Section: Path Transform: Slant:

```
if(transform_op -> type == TYPE_SLANTED){
  struct numeric_variable a;
  if(!eval_numeric_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  path_slant(result, a.value);
  return true;
}
```

And the function that slants:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_slant(struct path_variable *p, float s);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_slant(struct path_variable *p, float s){
  int i;
  for(i = 0; i < p -> length; i ++){
    p -> points[i].point.x += s * p -> points[i].point.y;
    p -> points[i].point.u_x += s * p -> points[i].point.u_y;
    p -> points[i].point.v_x += s * p -> points[i].point.v_y;
  }
```

```
}
```

The next transformer change the horizontal size of the path while preserving the vertical size. We need to read a numeric value and it determines how much the path should be horizontally stretched:

Section: Path Transform: X-Scale:

```
else if(transform_op -> type == TYPE_XSCALED){
  struct numeric_variable a;
  if(!eval_numeric_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  path_xyscale(result, a.value, 1.0);
  return true;
}
```

Here we are using a function already defined. We do not need to redefine it again. And we also can stretch the path vertically:

Section: Path Transform: Y-Scale:

```
else if(transform_op -> type == TYPE_YSCALED){
  struct numeric_variable a;
  if(!eval_numeric_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  path_xyscale(result, 1.0, a.value);
  return true;
}
```

The last but one transformer is the Z-scale which reads a pair, interprets it as a complex number, interpret all points in the path as complex numbers and multiply them:

$$(a+bi)(c+di) = ac + (ad)i + (cb)i + (bd)i^2 = (ac-bd) + (cb+ad)i$$

We interpret and make the transformation below, remembering that this is the last transformation and that an error should be raised if we need to apply a transformation and this is not the correct:

Section: Path Transform: Z-Scale:

```
else if(transform_op -> type == TYPE_ZSCALED){
  struct pair_variable a;
  if(!eval_pair_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  path_zscale(result, a.x, a.y);
  return true;
}
```

And the function:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_zscale(struct path_variable *p, float x, float y);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_zscale(struct path_variable *p, float x, float y){
  int i;
  for(i = 0; i < p -> length; i ++){
    float x0 = p -> points[i].point.x;
    float y0 = p -> points[i].point.y;
    p -> points[i].point.x = x0 * x - y0 * y;
    p -> points[i].point.y = x0 * y + y0 * x;
    x0 = p -> points[i].point.u_x;
    y0 = p -> points[i].point.u_y;
```

```
    p -> points[i].point.u_x = x0 * x - y0 * y;
    p -> points[i].point.u_y = x0 * y + y0 * x;
    x0 = p -> points[i].point.v_x;
    y0 = p -> points[i].point.v_y;
    p -> points[i].point.v_x = x0 * x - y0 * y;
    p -> points[i].point.v_x = x0 * y + y0 * x;
  }
}
```

The last transformer is a generic linear transform, where we will get a transform specifying how each point in the path should be transformed. We deal with this case as below:

Section: Path Transform: Generic Transform:

```
else if(transform_op -> type == TYPE_TRANSFORMED){
  struct transform_variable a;
  if(!eval_transform_primary(mf, cx, transform_op -> next, end, &a))
    return false;
  path_transform(result, a.value);
  return true;
}
else{
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
}
```

For this, we use the function below:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_transform(struct path_variable *p, float *M);
```

Which is defined as below:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_transform(struct path_variable *p, float *M){
  int i;
  for(i = 0; i < p -> length; i ++){
    float x0 = p -> points[i].point.x;
    float y0 = p -> points[i].point.y;
    p -> points[i].point.x = LINEAR_TRANSFORM_X(x0, y0, M);
    p -> points[i].point.y = LINEAR_TRANSFORM_Y(x0, y0, M);
    x0 = p -> points[i].point.u_x;
    y0 = p -> points[i].point.u_y;
    p -> points[i].point.u_x = LINEAR_TRANSFORM_X(x0, y0, M);
    p -> points[i].point.u_y = LINEAR_TRANSFORM_Y(x0, y0, M);
    x0 = p -> points[i].point.v_x;
    y0 = p -> points[i].point.v_y;
    p -> points[i].point.v_x = LINEAR_TRANSFORM_X(x0, y0, M);
    p -> points[i].point.v_x = LINEAR_TRANSFORM_Y(x0, y0, M);
  }
}
```

8.4.4. Primary Path Expressions: Variables, Reverses and Subpath

The grammar for primary path expressions is:

```
<Path Primary> -> <Pair Primary> | <Path Variable> |
```

```
( <Path Expression> ) |
reverse <Path Primary> |
subpath <Pair Expression> of <Pair Primary> |
...
```

The rules are incomplete because an additional primary operator related to pen variables will be defined in Subsection 8.5.4.

For now we need to register the new tokens `reverse`, `subpath` and `of`:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_REVERSE,          // 0 token simblico 'reverse'
TYPE_SUBPATH,          // 0 token simblico 'subpath'
TYPE_OF,               // 0 token simblico 'of'
```

And for each one of them, we add its string to the list of reserved words:

Section: List of Keywords (continuation):

```
"reverse", "subpath", "of",
```

What the grammar rules say is that in the end of each path expression, we will find a path variable, a parenthesis, some of these new path operators. In all other cases, we get a pair primary. Therefore, we should first test these first cases and if we are not in one of them, we evaluate as a primary pair.

The function that will interpret primary path expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_path_primary(struct metafont *mf, struct context *cx,
                       struct generic_token *begin,
                       struct generic_token *end,
                       struct path_variable *result);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_path_primary(struct metafont *mf, struct context *cx,
                       struct generic_token *begin,
                       struct generic_token *end,
                       struct path_variable *result){
  if(begin -> type == TYPE_REVERSE){
                    <Section to be inserted: Primary Path: Reverse>
  }
  else if(begin -> type == TYPE_SUBPATH){
                    <Section to be inserted: Primary Path: Subpath>
  }
  else if(begin == end && begin -> type == TYPE_SYMBOLIC){
                    <Section to be inserted: Primary Path: Variable>
  }
  else if(begin -> type == TYPE_OPEN_PARENTHESIS &&
          end -> type == TYPE_CLOSE_PARENTHESIS){
                    <Section to be inserted: Primary Path: Parenthesis>
  }
              <Section to be inserted: Primary Path: Other Expressions>
  { // If we still did not return, then it is a primary pair
                  <Section to be inserted: Primary Path: Primary Pair>
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
```

```
}
```

The first case: we must compute the reverse of a path. For this, we need to reverse the order for all extremity points and move the control points for its new position. This operation is always performed over normalized path variables, with recursion and points in the non-final format removed.

Section: Primary Path: Reverse:

```
struct path_variable tmp;
if(begin -> next == NULL || begin == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
}
if(!eval_path_primary(mf, cx, begin -> next, end, &tmp))
  return false;
if(!reverse_path(mf, cx, result, &tmp))
  return false;
if(temporary_free != NULL)
  path_recursive_free(temporary_free, &tmp, false);
return true;
```

The function that performs the reversion:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool reverse_path(struct metafont *mf, struct context *cx,
                  struct path_variable *dst,
                  struct path_variable *origin);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool reverse_path(struct metafont *mf, struct context *cx,
                  struct path_variable *dst,
                  struct path_variable *origin){
  int i;
  dst -> cyclic = origin -> cyclic;
  dst -> length = origin -> number_of_points;
  dst -> number_of_points = origin -> number_of_points;
  dst -> points = (struct path_points *)
     temporary_alloc(sizeof(struct path_points) * dst -> length);
  if(dst -> points == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, 0);
    return false;
  }
  for(i = 0; i < origin -> length - 1; i ++){
    dst -> points[i].format = FINAL_FORMAT;
    dst -> points[i].point.x = origin-> points[origin->length-1-i].point.x;
    dst -> points[i].point.y = origin-> points[origin->length-1-i].point.y;
    dst -> points[i].point.u_x = origin-> points[origin->length-2-i].point.v_x;
    dst -> points[i].point.u_y = origin-> points[origin->length-2-i].point.v_y;
    dst -> points[i].point.v_x = origin-> points[origin->length-2-i].point.u_x;
    dst -> points[i].point.v_y = origin-> points[origin->length-2-i].point.u_y;
  }
  dst -> points[i].format = FINAL_FORMAT;
  dst -> points[i].point.x = origin -> points[0].point.x;
  dst -> points[i].point.y = origin -> points[0].point.y;
```

```
  dst -> points[i].point.u_x = origin -> points[0].point.x;
  dst -> points[i].point.u_y = origin -> points[0].point.y;
  dst -> points[i].point.v_x = origin -> points[0].point.x;
  dst -> points[i].point.v_y = origin -> points[0].point.y;
  return true;
}
```

The next step is compute the subpath. A subpath creates a new path that is part of another known bigger path. For example:

```
subpath (0, 2) of p1 -- p2 -- p3;
```

The code above evaluates to the new path `p1 -- p2`, assuming that both `p1` and `p2` represent a single point.

In the original METAFONT, we could pass non-integer point positions. For example, we could select a subpath between the conrol points 0.5 and 1.8. With such positions, METAFONT would generate new intermediary control points betwwen the first and the second and between the second and third as indicated. New extemity and control points would be generated. But as this is a little messy computation, here we will support only subpaths between integer positions.

In the case of a non-cyclic path, if we try to specify a subpath using an index lesser than zero, this index will be considered zero. If we try to use an index greater than the maximum allowed index, it will be treated as the maximum index.

In the case of a cyclic path, negative indices are counted walking over the cycle in the path opposite direction. And there is no maximum allowed index. This way, you can create a subpath from a cyclic path that is bigger than the original path. The cyclic nature also will always be lost after the operation.

Section: Primary Path: Subpath:

```
DECLARE_NESTING_CONTROL();
struct pair_variable a;
struct path_variable b;
struct generic_token *of, *end_pair_expr = begin;
struct generic_token *begin_subexpr;
if(begin -> next == NULL ||  end_pair_expr -> type == TYPE_OF || begin == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
}
of = end_pair_expr -> next;
while(of != NULL && of != end){
  COUNT_NESTING(of);
  if(IS_NOT_NESTED() && of -> type == TYPE_OF)
    break;
  end_pair_expr = of;
  of = of -> next;
}
RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
if(of == NULL || of == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
}
if(!eval_pair_expression(mf, cx, begin -> next, end_pair_expr, &a))
  return false;
begin_subexpr = of -> next;
if(!eval_path_primary(mf, cx, begin_subexpr, end, &b))
  return false;
```

```
{
  int final_path_size, offset, i;
  result -> cyclic = false;
  // Ignore last point if equal the first:
  if(b.cyclic)
    b.length = b.length - 1;
  if(a.x < 0 && !b.cyclic)
    a.x = 0;
  if(a.y < 0  && !b.cyclic)
    a.y = 0;
  if(a.x >= b.length && !b.cyclic)
    a.x = b.length - 1;
  if(a.y >= b.length  && !b.cyclic)
    a.y = b.length - 1;
  final_path_size = a.y - a.x;
  if(final_path_size < 0)
    final_path_size *= -1;
  final_path_size ++;
  offset = ((int) ((a.x <= a.y)?(a.x):(a.y))) % b.length;
  if(offset < 0)
    offset *= -1;
  result -> length = final_path_size;
  result -> number_of_points = final_path_size;
  result -> points = (struct path_points *)
                       temporary_alloc(final_path_size *
                                       sizeof(struct path_points));
  if(result -> points == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  for(i = 0; i < result -> length; i ++)
    memcpy(&(result -> points[i]), &(b.points[(offset + i) % b.length]),
           sizeof(struct path_points));
  // Adjust final control points if we lost the cyclcic property:
  result -> points[result -> length - 1].point.u_x =
                                result -> points[result -> length - 1].point.x;
  result -> points[result -> length - 1].point.u_y =
                                result -> points[result -> length - 1].point.y;
  result -> points[result -> length - 1].point.v_x =
                                result -> points[result -> length - 1].point.x;
  result -> points[result -> length - 1].point.v_y =
                                result -> points[result -> length - 1].point.y;
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &b, false);
  if(a.x > a.y){
    struct path_variable c;
    if(!reverse_path(mf, cx, &c, result))
      return false;
    if(temporary_free != NULL)
      temporary_free(result -> points);
    result -> points = c.points;
```

```
  }
  return true;
}
```

The next step is the path primary expression when we read a pair variable or a path variable. In this case, we allocate the memory for the result points and copy the variable content to the result:

Section: Primary Path: Variable:

```
{
  struct symbolic_token *v = (struct symbolic_token *) begin;
  struct path_variable *var = v -> var;
  if(var == NULL){
    RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(begin -> line), v);
    return false;
  }
  if(((struct pair_variable *) var) -> type == TYPE_T_PAIR){
    if(isnan(((struct pair_variable *) var) -> x)){
      RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(begin -> line),
                                         v, TYPE_T_PAIR);
      return false;
    }
    result -> length = 1;
    result -> number_of_points = 1;
    result -> cyclic = false;
    result -> points = (struct path_points *)
                        temporary_alloc(sizeof(struct path_points));
    result -> points[0].format = FINAL_FORMAT;
    result -> points[0].point.x = ((struct pair_variable *) var) -> x;
    result -> points[0].point.y = ((struct pair_variable *) var) -> y;
    result -> points[0].point.u_x = ((struct pair_variable *) var) -> x;
    result -> points[0].point.u_y = ((struct pair_variable *) var) -> y;
    result -> points[0].point.v_x = ((struct pair_variable *) var) -> x;
    result -> points[0].point.v_y = ((struct pair_variable *) var) -> y;
    return true;
  }
  else if(var -> type == TYPE_T_PATH){
    if(var -> length == -1){
      RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(begin -> line),
                                         v, TYPE_T_PATH);
      return false;
    }
    return recursive_copy_points(mf, cx, temporary_alloc, &result, var, false);
  }
  else{
    RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(v -> line), v -> value,
                                    var -> type, TYPE_T_PATH);
    return false;
  }
}
```

The next primary path expression is when we have an expression between parenthesis. And this expression is not a pair, it is another path expression that need to be evaluated. First we check if there is nothing between parenthesis and produce and error in this case. Next, we check if we do not have a non-nested

comma inside the parenthesis. If so, we ignore the expression because we have a pair primary. In all other cases, we evaluate whatever expression we have inside the parenthesis as a path expression:

Section: Primary Path: Parenthesis:

```
struct generic_token *t = begin -> next;
bool found_comma = false;
DECLARE_NESTING_CONTROL();
if(begin -> next == end){
  RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '(');
  return false;
}
while(t != NULL && t -> next != end){
  COUNT_NESTING(t);
  if(IS_NOT_NESTED() && t -> type == TYPE_COMMA){
    found_comma = true;
    break;
  }
  t = t -> next;
}
if(!found_comma){
  return eval_path_expression(mf, cx, begin -> next, t, result);
}
```

And finally, the last kind of primary expression for pairs is when we have a pair primary expression. In this case, we just interpret the expression, allocate the single point in our path and copy the result of the expression to that point:

Section: Primary Path: Primary Pair:

```
struct pair_variable v;
if(!eval_pair_primary(mf, cx, begin, end, &v))
  return false;
result -> length = 1;
result -> number_of_points = 1;
result -> cyclic = false;
result -> points = (struct path_points *)
                    temporary_alloc(sizeof(struct path_points));
result -> points[0].format = FINAL_FORMAT;
result -> points[0].point.x = v.x;
result -> points[0].point.y = v.y;
result -> points[0].point.u_x = v.x;
result -> points[0].point.u_y = v.y;
result -> points[0].point.v_x = v.x;
result -> points[0].point.v_y = v.y;
return true;
```

8.4.5. Path in Numeric Expressions

We can nuse the numeric expression with the operator `length` to obtain the number of extremity points in a path minus one. The syntax is:

```
<Numeric Primary> -> length <Path Primary>
```

We do not need a new type of token. The expression `length` was already used to compute modulus for numbers and pairs. For the case in which this operator gets a path, the implementation is even simpler:

Section: Evaluate 'length' (continuation):

```
else if(expr_type == TYPE_T_PATH){
  struct path_variable p;
  if(!eval_path_primary(mf, cx, begin -> next, end, &p))
    return false;
  result -> value = (float) (p.number_of_points - 1);
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &p, false);
  return true;
}
```

8.4.6. Path in Pair Expressions

Given a path, we can extract pairs from it. The pair can be an extremity point or some of the control points. The syntax for this is:

```
<Pair Primary> -> point <Numeric Expression> of <Path Primary> |
                  precontrol <Numeric Expression> of <Path Primary> |
                  postcontrol <Numeric Expression> of <Path Primary>
```

This requires adding the following new token types:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_POINT,             // 0 token simblico 'point'
TYPE_PRECONTROL,        // 0 token simblico 'precontrol'
TYPE_POSTCONTROL,       // 0 token simblico 'postcontrol'
```

And we add them also to the list of reserved words:

Section: List of Keywords (continuation):

```
"point", "precontrol", "postcontrol",
```

All these operators require that we get a point in a given position n inside a path. For this, it will be helpful to have an auxiliary function that will get a path variable and an index to return the point in the path corresponding to that index. The function first will check if the length and the total lenght in the path are the same. If so, we will assume that we are in a path without recursive subpaths and in this case, returning the correct point is easy and fast. Otherwise, we will call a recursive function to get for us the correct point:

Section: Local Function Declaration (metafont.c) (continuation):

```
struct path_points *get_point(struct path_variable *v, int n);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct path_points *get_point(struct path_variable *v, int n){
  if(v -> length == v -> number_of_points){
    struct path_points *ret = (struct path_points *) &(v -> points[n]);
    while(ret -> format == SUBPATH_FORMAT)
      ret = &(((struct path_variable *) (ret -> subpath)) -> points[0]);
    return ret;
  }
  else{
    int count = 0;
    return _get_point(v, n, &count);
  }
}
```

This auxiliar recursive fnction that will get the point for recursively defined paths will walk over each point in the path in the usual order and will return the current point when it walks over the correct number of points. For this, it needs the path variable pointer, the point index and also how many points it already visited:

Section: Local Function Declaration (metafont.c) (continuation):

```
struct path_points *_get_point(struct path_variable *v, int n, int *count);
```

The function definition is:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct path_points *_get_point(struct path_variable *v, int n, int *count){
  int i;
  for(i = 0; i < v -> length; i ++){
    if(v -> points[i].format != SUBPATH_FORMAT) {
      if(*count == n && !isnan(v -> points[i].point.x))
        return ((struct path_points *) &(v -> points[i]));
      else if(!isnan(v -> points[i].point.x))
        (*count) ++;
    }
    else{
      struct path_points *r =
        _get_point((struct path_variable *) (v -> points[i].subpath),
                   n, count);
      if(r != NULL)
        return r;
    }
  }
  return NULL;
}
```

For the first operator, `point`, it returns a given extremity point from a path. In a non-cyclic path, indices lesser than zero are interpreted as zero (the first index) and indices greater or equal than the number of extremity points will be interpreted as the last point's index. If the path is cyclic, we count the indices following the cycle. In the original METAFONT language, non-integer indices were allowed, but here we will interpret all indices as integers, casting them to C int type.

The operator `postcontrol` is similar, but it gets the first control point after the given extremity point pointed by the index. And the operator `precontrol` gets the control point before the given extremity point.

Section: Pair Primary: Other Rules to Be Defined Later:

```
if(begin -> type == TYPE_POINT ||
   begin -> type == TYPE_PRECONTROL ||
   begin -> type == TYPE_POSTCONTROL){
  DECLARE_NESTING_CONTROL();
  struct generic_token *begin_num, *end_num , *of = NULL, *begin_path, *end_path;
  struct numeric_variable a;
  struct path_variable b;
  begin_num = begin -> next;
  end_num = begin_num;
  int index;
  while(end_num != NULL && end_num -> next != end){
    COUNT_NESTING(end_num);
    if(IS_NOT_NESTED() &&
```

```
     ((struct generic_token *) end_num -> next) -> type == TYPE_OF){
    of = end_num -> next;
    break;
  }
  end_num = end_num -> next;
}
if(of == NULL || of -> next == NULL){ // Code ends after 'of'
  RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
  return false;
}
if(of == end){ // Statement ends after 'of':
  RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, OPTIONAL(begin -> line));
  return false;
}
begin_path = of -> next;
end_path = end;
if(!eval_numeric_expression(mf, cx, begin_num, end_num, &a))
  return false;
if(!eval_path_primary(mf, cx, begin_path, end_path, &b))
  return false;
if(b.cyclic){
  index = ((int)(a.value)) % (b.number_of_points - 1);
  if(begin -> type == TYPE_PRECONTROL)
    index = (index - 1) % (b.number_of_points - 1);
}
else{
  index = (int) (a.value);
  if(index < 0) index = 0;
  if(index >= b.number_of_points) index = b.number_of_points - 1;
  if(begin -> type == TYPE_PRECONTROL)
    index --;
}
if(begin -> type == TYPE_POINT){
  result -> x = get_point(&b, index) -> point.x;
  result -> y = get_point(&b, index) -> point.y;
}
else if(begin -> type == TYPE_PRECONTROL){
  if(index < 0){
    result -> x = get_point(&b, 0) -> point.x;
    result -> y = get_point(&b, 0) -> point.y;
  }
  else{
    result -> x = get_point(&b, index) -> point.v_x;
    result -> y = get_point(&b, index) -> point.v_y;
  }
}
else{
  result -> x = get_point(&b, index) -> point.u_x;
  result -> y = get_point(&b, index) -> point.u_y;
}
if(temporary_free != NULL)
```

```
    path_recursive_free(temporary_free, &b, false);
  return true;
}
```

8.5. Pen Assignments and Expressions

To assign a pen to a variable with the right type, we use the following code:

Section: Assignment for Pen Variables:

```
else if(type == TYPE_T_PEN){
  int i;
  struct pen_variable result;
  if(!eval_pen_expression(mf, cx, begin_expression, *end, &result))
    return false;
  var = (struct symbolic_token *) begin;
  for(i = 0; i < number_of_variables; i ++){
    if(!assign_pen_variable(mf, cx, (struct pen_variable *) var -> var,
                            &result))
      return false;
    var = (struct symbolic_token *) (var -> next);
    var = (struct symbolic_token *) (var -> next);
  }
  if(temporary_free != NULL && result.format !=  NULL)
    path_recursive_free(temporary_free, result.format, true);
  if(result.gl_vbo != 0 && result.referenced == NULL)
    glDeleteBuffers(1, &(result.gl_vbo));
}
```

The declaration for the function that makes the assignment:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool assign_pen_variable(struct metafont *mf, struct context *cx,
                         struct pen_variable *target,
                         struct pen_variable *source);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool assign_pen_variable(struct metafont *mf, struct context *cx,
                         struct pen_variable *target,
                         struct pen_variable *source){
  void *(*alloc)(size_t);
  void (*disalloc)(void *);
  bool permanent = target -> permanent;
  struct variable *next = target -> next;
  if(permanent){
    disalloc = permanent_free;
    alloc = permanent_alloc;
  }
  else{
    disalloc = temporary_free;
    alloc = temporary_alloc;
  }
  if(target -> format != NULL && disalloc != NULL)
```

```
    path_recursive_free(disalloc, target -> format, true);
  if(target -> gl_vbo != 0)
    glDeleteBuffers(1, &(target -> gl_vbo));
  memcpy(target, source, sizeof(struct pen_variable));
  target -> type = TYPE_T_PEN;
  target -> next = next;
  target -> permanent = permanent;
  if(! (source -> flags & (FLAG_CIRCULAR | FLAG_SQUARE | FLAG_NULL)))
    if(!recursive_copy_points(mf, NULL, alloc, &(target -> format),
                              source -> format, true))
      return false;
  target -> gl_vbo = 0;
  target -> indices = 0;
  target -> referenced = NULL;
  // If we are assigning to 'currentpen', then we need to retriangulate
  // the pen, as will be described in Section 11.2:
  if(target == cx -> currentpen)
    triangulate_pen(mf, cx, target, target -> gl_matrix);
  return true;
}
```

8.5.1. Pen Tertiary Expression

For now we have no tertiry operator for pen expressions:

```
<Pen Expression> -> <Pen Tertiary>
<Pen Tertiary> -> <Pen Secondary>
```

So the function that evaluates tertiary expression, for now, can just call directly the funtion that evaluates secondary pen expressions and use some lines to detect common errors:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_pen_expression(struct metafont *mf, struct context *cx,
                         struct generic_token *begin,
                         struct generic_token *end,
                         struct pen_variable *result);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_pen_expression(struct metafont *mf, struct context *cx,
                         struct generic_token *begin,
                         struct generic_token *end,
                         struct pen_variable *result){
  if(begin -> type != TYPE_SYMBOLIC && begin -> type != TYPE_NULLPEN &&
     begin -> type != TYPE_OPEN_PARENTHESIS &&
     begin -> type != TYPE_PENCIRCLE && begin -> type != TYPE_PENSEMICIRCLE &&
     begin -> type != TYPE_MAKEPEN){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PEN);
    return false;
  }
  return eval_pen_secondary(mf, cx, begin, end, result);
}
```

8.5.2. Pen Transformers

The syntax for secondary pen expressions is:

```
<Pen Secondary> -> <Pen Primary> | <Pen Secondary> <Transformer>
<Transformer> -> rotated <Numeric Primary> |
                 scaled <Numeric Primary> |
                 shifted <Pair Primary> |
                 slanted <Numeric Primary> |
                 xscaled <Numeric Primary> |
                 yscaled <Numeric Primary> |
                 zscaled <Pair Primary>
                 transformed <Transform Primary>
```

Transformers are not new, they were already used in pair, transforms and path expressions. But apply them over pens is a little different, as instead of modifying each one of its points, we modify only its OpenGL matrix transformation.

The declaration for the function that evaluates pen secondary expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_pen_secondary(struct metafont *mf, struct context *cx,
                        struct generic_token *begin,
                        struct generic_token *end,
                        struct pen_variable *result);
```

And its implementation consists in trying to find the last transformer in the expression. If it is not found, the function just calls another function to evaluate primary expression. If a transformer is found, then everything before it is evaluated as a secondary expression and then we apply the transformer. Rotate a pen never requires the retriangulation of that pen.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_pen_secondary(struct metafont *mf, struct context *cx,
                        struct generic_token *begin,
                        struct generic_token *end,
                        struct pen_variable *pen){
  DECLARE_NESTING_CONTROL();
  struct generic_token *p, *prev = NULL, *last_transformer = NULL,
                       *before_last_transformer = begin;
  p = begin;
  do{
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && (p -> type == TYPE_ROTATED ||
       p -> type == TYPE_SCALED || p -> type == TYPE_SHIFTED ||
       p -> type == TYPE_SLANTED || p -> type == TYPE_XSCALED ||
       p -> type == TYPE_YSCALED || p -> type == TYPE_ZSCALED ||
       p -> type == TYPE_TRANSFORMED)){
      last_transformer = p;
      before_last_transformer = prev;
    }
    prev = p;
    if(p != end)
      p = (struct generic_token *) p -> next;
    else
      p = NULL;
  }while(p != NULL);
```

```
  if(last_transformer == NULL)
    return eval_pen_primary(mf, cx, begin, end, pen);
  else{
    if(!eval_pen_secondary(mf, cx, begin, before_last_transformer, pen))
      return false;
    if(last_transformer -> type == TYPE_ROTATED){
                    <Section to be inserted: Pen Secondary: Rotation>
    }
    else if(last_transformer -> type == TYPE_SCALED){
                     <Section to be inserted: Pen Secondary: Scaling>
    }
    else if(last_transformer -> type == TYPE_SHIFTED){
                      <Section to be inserted: Pen Secondary: Shift>
    }
    else if(last_transformer -> type == TYPE_SLANTED){
                     <Section to be inserted: Pen Secondary: Slanting>
    }
    else if(last_transformer -> type == TYPE_XSCALED){
                    <Section to be inserted: Pen Secondary: X-Scaling>
    }
    else if(last_transformer -> type == TYPE_YSCALED){
                    <Section to be inserted: Pen Secondary: Y-Scaling>
    }
    else if(last_transformer -> type == TYPE_ZSCALED){
                    <Section to be inserted: Pen Secondary: Z-Scaling>
    }
    else if(last_transformer -> type == TYPE_TRANSFORMED){
                <Section to be inserted: Pen Secondary: Generic Transform>
    }
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PEN);
    return false;
  }
}
```

First we will take care of rotation by an algle θ. First we perform the conversion between degrees used by WeaveFONT and radians, used by the C sandard library. Rotating means applying the rotation linear transform over the pen transform matrix:

Section: Pen Secondary: Rotation:

```
struct numeric_variable r;
double rotation;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &r))
  return false;
rotation = 0.017453292519943295 * r.value;
TRANSFORM_ROTATE(pen -> gl_matrix, rotation);
return true;
```

To perform the scaling tranformation, we also apply such linear transform over the pen matrix:

Section: Pen Secondary: Scaling:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
```

```
TRANSFORM_SCALE(pen -> gl_matrix, a.value);
// Curved pens need retriangulation if the size increase:
if(pen -> gl_vbo != 0 && a.value > 1.0 && !(pen -> flags & FLAG_STRAIGHT)){
  glDeleteBuffers(1, &(pen -> gl_vbo));
  pen -> gl_vbo = 0;
  pen -> indices = 0;
}
return true;
```

The shift transformer shifts each point in the pen adding a pair to it.

Section: Pen Secondary: Shift:

```
struct pair_variable pair;
if(!eval_pair_primary(mf, cx, last_transformer -> next, end, &pair))
  return false;
TRANSFORM_SHIFT(pen -> gl_matrix, pair.x, pair.y);
return true;
```

The next transformer is the slanting operator.

Section: Pen Secondary: Slanting:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SLANT(pen -> gl_matrix, a.value);
// Slant non-circular curved pens always require retriangulation:
if(pen -> gl_vbo != 0 && !(pen -> flags & FLAG_STRAIGHT) &&
   !(pen -> flags & FLAG_CIRCULAR)){
  glDeleteBuffers(1, &(pen -> gl_vbo));
  pen -> gl_vbo = 0;
  pen -> indices = 0;
}
return true;
```

Scaling in the x-axis is analogous to the general scaling, but the pen is stretched only in the x-axis:

Section: Pen Secondary: X-Scaling:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SCALE_X(pen -> gl_matrix, a.value);
// Curved pens need retriangulation if the size increase:
if(pen -> gl_vbo != 0 && a.value > 1.0 && !(pen -> flags & FLAG_STRAIGHT)){
  glDeleteBuffers(1, &(pen -> gl_vbo));
  pen -> gl_vbo = 0;
  pen -> indices = 0;
}
return true;
```

Likewise, changing the scale only in the vertical axis, is done by the code below:

Section: Pen Secondary: Y-Scaling:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SCALE_Y(pen -> gl_matrix, a.value);
// Curved pens need retriangulation if the size increase:
```

```
if(pen -> gl_vbo != 0 && a.value > 1.0 && !(pen -> flags & FLAG_STRAIGHT)){
  glDeleteBuffers(1, &(pen -> gl_vbo));
  pen -> gl_vbo = 0;
  pen -> indices = 0;
}
return true;
```

Scaling in z-index means interpreting the point and a pair as complex numbers and multiply them:

$$(x + yi)(a + bi) = ax + (bx)i + (ay)i + (by)i^2 = (ax - by) + (bx + ay)i$$

We can perform the operation above with the following code:

Section: Pen Secondary: Z-Scaling:

```
struct pair_variable pair;
if(!eval_pair_primary(mf, cx, last_transformer -> next, end, &pair))
  return false;
TRANSFORM_SCALE_Z(pen -> gl_matrix, pair.x, pair.y);
// Curved pens need retriangulation in this case:
if(pen -> gl_vbo != 0 && !(pen -> flags & FLAG_STRAIGHT)){
  glDeleteBuffers(1, &(pen -> gl_vbo));
  pen -> gl_vbo = 0;
  pen -> indices = 0;
}
return true;
```

Finally, the last transform is the generic one, which applies whichever transform is stored in a transformer:

Section: Pen Secondary: Generic Transform:

```
struct transform_variable t;
if(!eval_transform_primary(mf, cx, last_transformer -> next, end, &t))
  return false;
MATRIX_MULTIPLICATION(pen -> gl_matrix, t.value);
// Curved pens need retriangulation in this case:
if(pen -> gl_vbo != 0 && !(pen -> flags & FLAG_STRAIGHT)){
  glDeleteBuffers(1, &(pen -> gl_vbo));
  pen -> gl_vbo = 0;
  pen -> indices = 0;
}
return true;
```

8.5.3. Variables, Null Pen, Circular Pen and Arbitrary Pen

The grammar for pen primary expressions are defined as:

```
<Primrio de Caneta> -> <Varivel de Caneta> |
                       nullpen | ( <Expresso de Caneta> ) |
                       pencircle | pensemicircle |
                       makepen <Primrio de Caminho>
```

This requires four new reserved words:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_NULLPEN,        // The symbolic token 'nullpen'
TYPE_PENCIRCLE,      // The symbolic token 'pencircle'
TYPE_PENSEMICIRCLE,  // The symbolic token 'pensemicircle'
```

```
TYPE_MAKEPEN,        // The symbolic token 'makepen'
```

And we add a string with their names in the list of reserved keywords:

Section: List of Keywords (continuation):

```
"nullpen", "pencircle", "pensemicircle", "makepen",
```

The function that will interpret the primary pen expressions:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_pen_primary(struct metafont *mf, struct context *cx,
                      struct generic_token *begin,
                      struct generic_token *end,
                      struct pen_variable *result);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_pen_primary(struct metafont *mf, struct context *cx,
                      struct generic_token *begin,
                      struct generic_token *end,
                      struct pen_variable *result){
  if(begin == end){
    if(begin -> type == TYPE_SYMBOLIC){
                      <Section to be inserted: Pen Primary: Variable>
    }
    else if(begin -> type == TYPE_NULLPEN){
                      <Section to be inserted: Pen Primary: Null Pen>
    }
    else if(begin -> type == TYPE_PENCIRCLE){
                     <Section to be inserted: Pen Primary: Circle Pen>
    }
    else if(begin -> type == TYPE_PENSEMICIRCLE){
                   <Section to be inserted: Pen Primary: Semicircle Pen>
    }
  }
  else{
    if(begin -> type == TYPE_OPEN_PARENTHESIS &&
      end -> type == TYPE_CLOSE_PARENTHESIS){
                    <Section to be inserted: Pen Primary: Parenthesis>
    }
    else if(begin -> type == TYPE_MAKEPEN){
                  <Section to be inserted: Pen Primary: Custom Format>
    }
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PEN);
  return false;
}
```

If the expression evaluates to a variable, we need to copy the variable content to the result of the evaluation. If the evaluation return a variable that points to other (the `currentpen`), instead we copy the content of the referenced pen. But after the copy we multiply the obtained matrix by the pointer matrix.

Section: Pen Primary: Variable:

```
struct symbolic_token *v = (struct symbolic_token *) begin;
```

```
struct pen_variable *content = v -> var, *to_copy = v -> var;
if(!strcmp(v -> value, "currentpen"))
  content = cx -> currentpen;
if(content == NULL){
  RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(v -> line), v);
  return false;
}
if(content -> type != TYPE_T_PEN){
  RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(v -> line), v -> value,
                                  ((struct variable *)(v -> var)) -> type,
                                  TYPE_T_PEN);
  return false;
}
if(content -> format == NULL && content -> flags == false){
  RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(v -> line), v,
                                     TYPE_T_PEN);
  return false;
}
if(content -> referenced != NULL)
  to_copy = content -> referenced;
memcpy(result, to_copy, sizeof(struct pen_variable));
if(to_copy -> format != NULL)
  if(!recursive_copy_points(mf, cx, temporary_alloc, &(result -> format),
                            to_copy -> format, true))
    return false;
if(to_copy != content)
  MATRIX_MULTIPLICATION(result -> gl_matrix, content -> gl_matrix);
return true;
```

If the expression evaluates to a null pen, we should just create a new pen with the flag FLAG_NULL active. This pen never will be triangulated and never will produce any drawing:

Section: Pen Primary: Null Pen:

```
result -> format = NULL;
result -> flags = FLAG_NULL;
result -> referenced = NULL;
result -> gl_vbo = 0;
result -> indices = 0;
INITIALIZE_IDENTITY_MATRIX(result -> gl_matrix);
return true;
```

If the expression evaluates to a circular pen, we generate a new pen with circular and convex flags. We also initialize the transform matrix as an identity matrix:

Section: Pen Primary: Circle Pen:

```
result -> format = NULL;
result -> flags = FLAG_CONVEX | FLAG_CIRCULAR;
result -> referenced = NULL;
result -> gl_vbo = 0;
result -> indices = 0;
INITIALIZE_IDENTITY_MATRIX(result -> gl_matrix);
return true;
```

If the pen is a a semicircle, the code is very similar:

Section: Pen Primary: Semicircle Pen:

```
result -> format = NULL;
result -> flags = FLAG_CONVEX | FLAG_SEMICIRCULAR;
result -> referenced = NULL;
result -> gl_vbo = 0;
result -> indices = 0;
INITIALIZE_IDENTITY_MATRIX(result -> gl_matrix);
return true;
```

Evaluate parenthesis requires to evaluate as a pen expression every token after the open parenthesis until the position before the closing of the parenthesis:

Section: Pen Primary: Parenthesis:

```
struct generic_token *t = begin -> next;
DECLARE_NESTING_CONTROL();
if(begin -> next == end){ // Empty parenthesis: '()':
  RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '(');
  return false;
}
while(t != NULL && t -> next != end){
  COUNT_NESTING(t);
  t = t -> next;
}
RAISE_ERROR_AND_EXIT_IF_WRONG_NESTING(mf, cx, OPTIONAL(begin -> line));
return eval_pen_expression(mf, cx, begin -> next, t, result);
```

The last case of pen expression is when the user chooses a custom format for the pen. In this case, we should evaluate the path expression and create a new pen with that given format:

Section: Pen Primary: Custom Format:

```
struct generic_token *p = begin -> next;
result -> format =
        (struct path_variable *) temporary_alloc(sizeof(struct path_variable));
if(result -> format == NULL){
  RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
  return false;
}
if(!eval_path_primary(mf, cx, p, end, result -> format))
  return false;
if(!(result -> format -> cyclic)){
  RAISE_ERROR_NONCYCLICAL_PEN(mf, cx, OPTIONAL(begin -> line));
  return false;
}
result -> flags = read_flags(result -> format); // To be implemented below
result -> referenced = NULL;
result -> gl_vbo = 0;
result -> indices = 0;
INITIALIZE_IDENTITY_MATRIX(result -> gl_matrix);
return true;
```

The code for custom pen creation needs a function that walks over the points in a path and return the flags that a pen with that format would have:

Section: Local Function Declaration (metafont.c) (continuation):

```
int read_flags(struct path_variable *path);
```

We already know that the pen never will be circular, as perfect circles cannot be expressed using Bezir curves. We will verify only if the pen is straight or convex:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
int read_flags(struct path_variable *path){
  int i, positive_cross_product = -1;
  int flag = FLAG_CONVEX | FLAG_STRAIGHT;
  for(i = 0; i < path -> number_of_points - 1; i ++){
    struct path_points *current, *next;
    current = get_point(path, i);
    next = get_point(path, i + 1);
    if(flag & FLAG_STRAIGHT){
      // It is straight if the auxiliary and control points are colinear.
      // They are colinear if they form a triangle with area zero (or
      // sufficiently next to zero):
      double area =  current -> point.x * (current -> point.u_y - next -> point.y) +
                     current -> point.u_x * (next -> point.y - current -> point.y) +
                     next -> point.x * (current -> point.y - current -> point.u_y);
      if(area > 0.01 || area < -0.01)
        flag -= FLAG_STRAIGHT;
      area =  current -> point.x * (current -> point.v_y - next -> point.y) +
              current -> point.v_x * (next -> point.y - current -> point.y) +
              next -> point.x * (current -> point.y - current -> point.v_y);
      if((flag & FLAG_STRAIGHT) && (area > 0.01 || area < -0.01))
        flag -= FLAG_STRAIGHT;
    }
    if(flag & FLAG_CONVEX){
      // It is convex only if the z component of the cross product of vectors
      // composed by the extremity and control points in the drawing order
      // is always non-negative or always non-positive
      int j;
      double d1_x, d1_y, d2_x, d2_y, z_cross_product;
      double p1_x, p1_y, p2_x, p2_y, p3_x, p3_y; // Pontos
      for(j = 0; j < 3; j ++){
        switch(j){
          case 0:
            p1_x = current -> point.x; p1_y = current -> point.y;
            p2_x = current -> point.u_x; p2_y = current -> point.u_y;
            p3_x = current -> point.v_x; p3_y = current -> point.v_y;
            break;
          case 1:
            p1_x = p2_x; p1_y = p2_y;
            p2_x = p3_x; p2_y = p3_y;
            p3_x = next -> point.x; p3_y = next -> point.y;
            break;
          default:
            p1_x = p2_x; p1_y = p2_y;
            p2_x = p3_x; p2_y = p3_y;
            p3_x = next -> point.u_x; p3_y = next -> point.u_y;
            break;
```

```
        }
        d1_x = p2_x - p1_x;
        d1_y = p2_y - p1_y;
        d2_x = p3_x - p2_x;
        d2_y = p3_y - p2_y;
        z_cross_product = d1_x * d2_y - d1_y * d2_x;
        if(z_cross_product > 0.01 || z_cross_product < -0.01){
          if(positive_cross_product == -1)
            positive_cross_product = (z_cross_product > 0);
          else if((z_cross_product > 0) != positive_cross_product){
            flag -= FLAG_CONVEX;
            break;
          }
        }
      }
    }
  }
  return flag;
}
```

8.5.4. Pens in Path Expressions

Primary pen expressions also can appear inside path primary expressions. This happens when we try to extract a pen format as a path. The grammar for this operation is:

```
<Path Primary> -> makepath <Pen Primary>
```

We need to register a new token for "makepath":

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_MAKEPATH,       // 0 token simblico 'makepath'
```

And we add the string with the token name to the list of reserved words:

Section: List of Keywords (continuation):

```
"makepath",
```

The operator "makepath" evaluates all tokens after itself as a pen primary expression. After this, it takes the resulting pen and extract its format as a path. Sometimes the pen variable will already have its format stored as a path and we just need to extract and copy it. But in some cases, if we have a `nullpen` or a `circlepen`, or even if we have a square pen, we need to generate a path with a correct format:

Section: Primary Path: Other Expressions:

```
else if(begin -> type == TYPE_MAKEPATH){
  struct pen_variable tmp;
  if(begin -> next == NULL || begin == end){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PATH);
    return false;
  }
  if(!eval_pen_primary(mf, cx, begin -> next, end, &tmp))
    return false;
  if(tmp.flags & FLAG_NULL){ // nullpen: Avalia para ponto nico (0, 0)
                    <Section to be inserted: 'makepath': 'nullpen'>
  }
  else if(tmp.flags & FLAG_CIRCULAR){ // circlepen: Circle approximation
```

```
                    <Section to be inserted: 'makepath': 'pencircle'>
  }
  else if(tmp.flags & FLAG_SEMICIRCULAR){
                  <Section to be inserted: 'makepath': 'pensemicircle'>
  }
  else if(tmp.flags & FLAG_SQUARE){ // Caneta quadrada
                   <Section to be inserted: 'makepath': 'pensquare'>
  }
  else{ // Caneta com formato personalizado
               <Section to be inserted: 'makepath': Formato Personalizado>
  }
             <Section to be inserted: 'makepath': Apply Linear Transform>
  return true;
}
```

As in the original METAFONT language, the `nullpen` represents a single point in position $(0, 0)$ that never will be drawn:

Section: 'makepath': 'nullpen':

```
result -> length = 1;
result -> number_of_points = 1;
result -> cyclic = false;
result -> points =
  (struct path_points *) temporary_alloc(sizeof(struct path_points));
result -> points[0].format = FINAL_FORMAT;
result -> points[0].point.x = 0.0;
result -> points[0].point.y = 0.0;
result -> points[0].point.u_x = 0.0;
result -> points[0].point.u_y = 0.0;
result -> points[0].point.v_x = 0.0;
result -> points[0].point.v_y = 0.0;
```

If we have a circular pen, then we use as a circle approximation the same values that the original METAFONT uses in a macro to represent its `fullcircle` path. The result is close enough to a circle:

Section: 'makepath': 'pencircle':

```
result -> length = 9;
result -> number_of_points = 9;
result -> points =
   (struct path_points *) temporary_alloc(sizeof(struct path_points) * 9);
result -> points[0].format = FINAL_FORMAT;
result -> points[0].point.x = 0.5; result -> points[0].point.y = 0.0;
result -> points[0].point.u_x = 0.5; result -> points[0].point.u_y = 0.13261;
result -> points[0].point.v_x = 0.44733; result -> points[0].point.v_y = 0.2598;
result -> points[1].format = FINAL_FORMAT;
result -> points[1].point.x = 0.35356; result -> points[1].point.y = 0.35356;
result -> points[1].point.u_x = 0.2598; result -> points[1].point.u_y = 0.44733;
result -> points[1].point.v_x = 0.13261; result -> points[1].point.v_y = 0.5;
result -> points[2].format = FINAL_FORMAT;
result -> points[2].point.x = 0.0; result -> points[2].point.y = 0.5;
result -> points[2].point.u_x = -0.13261; result -> points[2].point.u_y = 0.5;
result -> points[2].point.v_x = -0.2598; result -> points[2].point.v_y = 0.44733;
result -> points[3].format = FINAL_FORMAT;
result -> points[3].point.x = -0.35356; result -> points[3].point.y = 0.35356;
```

```
result -> points[3].point.u_x = -0.44733; result -> points[3].point.u_y = 0.2598;
result -> points[3].point.v_x = -0.5; result -> points[3].point.v_y = 0.13261;
result -> points[4].format = FINAL_FORMAT;
result -> points[4].point.x = -0.5; result -> points[4].point.y = 0.0;
result -> points[4].point.u_x = -0.5; result -> points[4].point.u_y = -0.13261;
result -> points[4].point.v_x = -0.44733; result -> points[4].point.v_y = -0.2598;
result -> points[5].format = FINAL_FORMAT;
result -> points[5].point.x = -0.35356; result -> points[5].point.y = -0.35356;
result -> points[5].point.u_x = -0.2598; result -> points[5].point.u_y = -0.44733;
result -> points[5].point.v_x = -0.13261; result -> points[5].point.v_y = -0.5;
result -> points[6].format = FINAL_FORMAT;
result -> points[6].point.x = 0.0; result -> points[6].point.y = -0.5;
result -> points[6].point.u_x = 0.13261; result -> points[6].point.u_y = -0.5;
result -> points[6].point.v_x = 0.2598; result -> points[6].point.v_y = -0.44733;
result -> points[7].format = FINAL_FORMAT;
result -> points[7].point.x = 0.35356; result -> points[7].point.y = -0.35356;
result -> points[7].point.u_x = 0.44733; result -> points[7].point.u_y = -0.2598;
result -> points[7].point.v_x = 0.5; result -> points[7].point.v_y = -0.13261;
result -> points[7].format = FINAL_FORMAT;
result -> points[8].point.x = 0.5; result -> points[8].point.y = 0.0;
result -> points[8].point.u_x = 0.5; result -> points[8].point.u_y = 0.13261;
result -> points[8].point.v_x = 0.44733; result -> points[8].point.v_y = 0.2598;
result -> points[8].format = FINAL_FORMAT;
result -> cyclic = true;
```

For a semicircular pen, it is the same, we just use half of the above points:

Section: 'makepath': 'pensemicircle':

```
result -> length = 5;
result -> number_of_points = 5;
result -> points =
   (struct path_points *) temporary_alloc(sizeof(struct path_points) * 9);
result -> points[0].format = FINAL_FORMAT;
result -> points[0].point.x = 0.5; result -> points[0].point.y = 0.0;
result -> points[0].point.u_x = 0.5; result -> points[0].point.u_y = 0.13261;
result -> points[0].point.v_x = 0.44733; result -> points[0].point.v_y = 0.2598;
result -> points[1].format = FINAL_FORMAT;
result -> points[1].point.x = 0.35356; result -> points[1].point.y = 0.35356;
result -> points[1].point.u_x = 0.2598; result -> points[1].point.u_y = 0.44733;
result -> points[1].point.v_x = 0.13261; result -> points[1].point.v_y = 0.5;
result -> points[2].format = FINAL_FORMAT;
result -> points[2].point.x = 0.0; result -> points[2].point.y = 0.5;
result -> points[2].point.u_x = -0.13261; result -> points[2].point.u_y = 0.5;
result -> points[2].point.v_x = -0.2598; result -> points[2].point.v_y = 0.44733;
result -> points[3].format = FINAL_FORMAT;
result -> points[3].point.x = -0.35356; result -> points[3].point.y = 0.35356;
result -> points[3].point.u_x = -0.44733; result -> points[3].point.u_y = 0.2598;
result -> points[3].point.v_x = -0.5; result -> points[3].point.v_y = 0.13261;
result -> points[4].format = FINAL_FORMAT;
result -> points[4].point.x = -0.5; result -> points[4].point.y = 0.0;
result -> points[4].point.u_x = -0.33333; result -> points[4].point.u_y = 0.0;
result -> points[4].point.v_x =  0.33333; result -> points[4].point.v_y = 0.0;
```

```
result -> cyclic = true;
```

If we have a square pen, we generate its true format filling its extremity and control points:

Section: 'makepath': 'pensquare':

```
result -> length = 5;
result -> number_of_points = 5;
result -> points =
   (struct path_points *) temporary_alloc(sizeof(struct path_points) * 5);
result -> points[0].format = FINAL_FORMAT;
result -> points[0].point.x = -0.5; result -> points[0].point.y = -0.5;
result -> points[0].point.u_x = (-0.5+(1.0/3.0));
result -> points[0].point.u_y = -0.5;
result -> points[0].point.v_x = (-0.5+(2.0/3.0));
result -> points[0].point.v_y = -0.5;
result -> points[1].format = FINAL_FORMAT;
result -> points[1].point.x = 0.5; result -> points[1].point.y = -0.5;
result -> points[1].point.u_x = 0.5;
result -> points[1].point.u_y = (-0.5+(1.0/3.0));
result -> points[1].point.v_x = 0.5;
result -> points[1].point.v_y = (-0.5+(2.0/3.0));
result -> points[2].format = FINAL_FORMAT;
result -> points[2].point.x = 0.5; result -> points[2].point.y = 0.5;
result -> points[2].point.u_x = (0.5-(1.0/3.0));
result -> points[2].point.u_y = 0.5;
result -> points[2].point.v_x = (0.5-(2.0/3.0));
result -> points[2].point.v_y = 0.5;
result -> points[3].format = FINAL_FORMAT;
result -> points[3].point.x = -0.5; result -> points[3].point.y = 0.5;
result -> points[3].point.u_x = -0.5;
result -> points[3].point.u_y = (0.5-(1.0/3.0));
result -> points[3].point.v_x = -0.5;
result -> points[3].point.v_y = (0.5-(2.0/3.0));
result -> points[4].format = FINAL_FORMAT;
result -> points[4].point.x = -0.5; result -> points[4].point.y = -0.5;
result -> points[4].point.u_x = (-0.5+(1.0/3.0));
result -> points[4].point.u_y = -0.5;
result -> points[4].point.v_x = (-0.5+(2.0/3.0));
result -> points[4].point.v_y = -0.5;
result -> cyclic = true;
```

Finally, if we have a pen with a custom format, then it already stores its format as a path variable. We just need to copy it and then deallocate the temporary pen evaluation:

Section: 'makepath': Formato Personalizado:

```
if(!recursive_copy_points(mf, cx, temporary_alloc, &result, tmp.format, false))
  return false;
if(temporary_free != NULL){
  temporary_free(tmp.format -> points);
  temporary_free(tmp.format);
}
```

And finally, after obtaining the path, we need to apply all the linear transformations stored in the transform matrix, applying any rotation, shifting, slanting, scaling or other transformations stored there:

Section: 'makepath': Apply Linear Transform:

```
{
  int i;
  for(i = 0; i < result -> length; i ++){
    float x0 = result -> points[i].point.x, y0 = result -> points[i].point.y;
    result -> points[i].point.x = LINEAR_TRANSFORM_X(x0, y0, tmp.gl_matrix);
    result -> points[i].point.y = LINEAR_TRANSFORM_Y(x0, y0, tmp.gl_matrix);
    x0 = result -> points[i].point.u_x;
    y0 = result -> points[i].point.u_y;
    result -> points[i].point.u_x = LINEAR_TRANSFORM_X(x0, y0, tmp.gl_matrix);
    result -> points[i].point.u_y = LINEAR_TRANSFORM_Y(x0, y0, tmp.gl_matrix);
    x0 = result -> points[i].point.v_x;
    y0 = result -> points[i].point.v_y;
    result -> points[i].point.v_x = LINEAR_TRANSFORM_X(x0, y0, tmp.gl_matrix);
    result -> points[i].point.v_y = LINEAR_TRANSFORM_Y(x0, y0, tmp.gl_matrix);
  }
}
```

8.6. Picture Assignments and Expressions

To make a picture assignment, we use the code below. It first evaluates a picture expression and get a picture variable as result. The picture variable has an initialized and filled OpenGL texture. If our assignment has a single variable, we can just copy the texture identifier and the other variable information. If we have more than one variables as destiny in our assignment, then for the first variable we just copy the texture identifier, and for the other we will need to generate new textures and copy the content.

Section: Assignment for Picture Variables:

```
else if(type == TYPE_T_PICTURE){
  int i;
  struct picture_variable result;
  if(!eval_picture_expression(mf, cx, begin_expression, *end, &result))
    return false;
  var = (struct symbolic_token *) begin;
  for(i = 0; i < number_of_variables; i ++){
    if(i == 0){
      struct picture_variable *pic = (struct picture_variable *) var -> var;
      if(pic -> texture != 0)
        glDeleteTextures(1, &(pic -> texture));
      pic -> width = result.width;
      pic -> height = result.height;
      pic -> texture = result.texture;
      pic -> type = TYPE_T_PICTURE;
      // If assigning to 'currentpicture', additional code is executed
      if(pic == cx -> currentpicture){
        // The code below will be defined in Subsection 14.1:
                    <Section to be inserted: Create new 'currentpicture'>
      }
    }
    else
      assign_picture_variable(mf, cx, (struct picture_variable *) var -> var,
                              &result);
    var = (struct symbolic_token *) (var -> next);
```

```
    var = (struct symbolic_token *) (var -> next);
  }
}
```

The function `assign_picture_variable` above needs to generate a new texture in the destiny (removing the existing texture if it exists) and copy the content of the origin texture in the destiny. This means that we need to render the content of one texture in the other.

To render something, we first need vertices. As all that we will render will be rectangular textures, we will need only the following vertices for all the rendering:

Section: Local Variables (metafont.c) (continuation):

```
static const float square[20] = {
  -1.0, -1.0, // First vertice
  0.0, 0.0,  // Texture coordinate
  1.0, -1.0, // Second vertice
  1.0, 0.0, // Texture
  1.0, 1.0, // Third vertice
  1.0, 1.0, // Texture
  -1.0, 1.0, // Fourth vertice
  0.0, 1.0}; // Texture
static GLuint vbo; // OpenGL Vertex Buffer Object
```

Each vertice above also stores information about the texture coordinate. So we can sent to the GPU both informations at once. Notice that we defined above a square with length 2 centered at the origin. This is the correct size to render the vertices occupying all the available space according with OpenGL conventions. We also defined the vertices in counter-clockwise order, which means that we are looking at the front side os the polycon according with OpenGL conventions. This is necessary so that the polygon will be rendered even if optimizations to avoid rendering the back of polygons are active.

When initializing Weaver Metafont, we sent these vertices to the graphics card:

Section: WeaveFont Initialization:

```
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
// Sending vertices to the graphics card
glBufferData(GL_ARRAY_BUFFER, sizeof(square), square, GL_STATIC_DRAW);
```

And during finalization, we remove and erase these vertices:

Section: WeaveFont Finalization:

```
glDeleteBuffers(1, &vbo);
```

To render the vertices, we need a vertex shader running in the GPU. Our vertex shader must be versatile enough to get as argument a matrix with linear transformation to be done over the vertices. Besides the matrix, the input will be the vertex coordinate, and the associated texture coordinate. The output that will be passed to the fragment shader os the final coordinate after the linear transformation and the texture coordinate.

Section: Local Variables (metafont.c) (continuation):

```
static const char vertex_shader[] =
  "#version 100\n"
  "attribute vec4 vertex_data;\n"
  "uniform mat3 model_view_matrix;\n"
  "varying mediump vec2 texture_coordinate;\n"
  "void main(){\n"
  "  highp vec3 coord;\n"
  "  coord = vec3(vertex_data.xy, 1.0) * model_view_matrix;\n"
  "  gl_Position = vec4(coord.x, coord.y, 0.0, 1.0);\n"
```

```
  "  texture_coordinate = vertex_data.zw;\n"
  "}\n";
```

The fragment shader get as input the vertex shader output and a texture:

Section: Local Variables (metafont.c) (continuation):

```
static const char fragment_shader[] =
  "#version 100\n"
  "precision mediump float;\n"
  "varying mediump vec2 texture_coordinate;\n"
  "uniform sampler2D texture1;\n"
  "void main(){\n"
  "  vec4 texture = texture2D(texture1, texture_coordinate);\n"
  "  gl_FragColor = texture;"
  "}\n";
static GLuint program; // Store the program after compiling the above shaders
GLint uniform_matrix; // Stores the matrix position in the program above
GLint uniform_texture; // Stores the texture position in the above code
```

These simple shaders must be compiled during initialization. Let's create a local auxiliary function to help us with the compilation:

Section: Local Function Declaration (metafont.c) (continuation):

```
GLuint compile_shader_program(const char *vertex_shader_source,
                              const char *fragment_shader_source);
```

This function will return the compiled program's identifier given the shader source code passed as argument. In case of error, it will return zero.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
GLuint compile_shader_program(const char *vertex_shader_source,
                              const char *fragment_shader_source){
  GLuint vertex, fragment, prog;
  GLint status = GL_TRUE;
  // Creating vertex and fragment shader
  vertex = glCreateShader(GL_VERTEX_SHADER);
  fragment = glCreateShader(GL_FRAGMENT_SHADER);
  // Passing their source code
  glShaderSource(vertex, 1, &vertex_shader_source, NULL);
  glShaderSource(fragment, 1, &fragment_shader_source, NULL);
  // Compiling vertex shader:
  glCompileShader(vertex);
  glGetShaderiv(vertex, GL_COMPILE_STATUS, &status);
  if(status == GL_FALSE){
    fprintf(stderr,
            "ERROR: Weaver Metafont vertex shader compilation failed!\n");
    return 0;
  }
  // Compiling fragment shader:
  glCompileShader(fragment);
  glGetShaderiv(fragment, GL_COMPILE_STATUS, &status);
  if(status == GL_FALSE){
    fprintf(stderr,
            "ERROR: Weaver Metafont fragment shader compilation failed!\n");
```

```
    return 0;
  }
  // Creating the program:
  prog = glCreateProgram();
  // Linking the program:
  glAttachShader(prog, vertex);
  glAttachShader(prog, fragment);
  glBindAttribLocation(prog, 0, "vertex_data");
  glLinkProgram(prog);
  glGetProgramiv(prog, GL_LINK_STATUS, &status);
  if(status == GL_FALSE){
    fprintf(stderr, "ERROR: Weaver Metafont shader linking failed!\n");
    return false;
  }
  // Finalization:
  glDeleteShader(vertex);
  glDeleteShader(fragment);
  return prog;
}
```

And we can use now this function to compile our default shader program:

Section: WeaveFont Initialization (continuation):

```
{
  program = compile_shader_program(vertex_shader, fragment_shader);
  if(program == 0)
    return false;
  uniform_matrix = glGetUniformLocation(program, "model_view_matrix");
  uniform_texture = glGetUniformLocation(program, "texture1");
}
```

During finalization we will delete the program:

Section: WeaveFont Finalization (continuation):

```
glDeleteProgram(program);
```

To render the content of one texture in another, we will need to create a new framebuffer with a new texture attached using a given width and height. The function below will help us to achieve this. It will create a new framebuffer with a new attached texture and activate both. Everything that will be rendered after this, will be rendered in the texture instead of the screen.

Section: Local Function Declaration (metafont.c) (continuation):

```
bool get_new_framebuffer(GLuint *new_framebuffer, GLuint *new_texture,
                         int width, int height);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool get_new_framebuffer(GLuint *new_framebuffer, GLuint *new_texture,
                         int width, int height){
  glGenFramebuffers(1, new_framebuffer);
  glGenTextures(1, new_texture);
  glBindTexture(GL_TEXTURE_2D, *new_texture);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, NULL);
  glBindFramebuffer(GL_FRAMEBUFFER, *new_framebuffer);
  glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                         *new_texture, 0);
  if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE){
    glBindTexture(GL_TEXTURE_2D, 0);
    return false;
  }
  glBindTexture(GL_TEXTURE_2D, 0);
  return true;
}
```

We also need code to render a picture variable in the current framebuffer using the defined shader program. The function that will perform this is:

Section: Local Function Declaration (metafont.c) (continuation):

```
void render_picture(struct picture_variable *pic, float *matrix, int dst_width,
                    int dst_height, bool clear_background);
// XXX:
void print_picture(struct picture_variable *pic);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void render_picture(struct picture_variable *pic, float *matrix, int dst_width,
                    int dst_height, bool clear_background){
  glColorMask(true, true, true, true);
  glViewport(0, 0, dst_width, dst_height);
  glBindBuffer(GL_ARRAY_BUFFER, vbo);
  glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void *) 0);
  glEnableVertexAttribArray(0);
  glUseProgram(program);
  glUniformMatrix3fv(uniform_matrix, 1, true, matrix);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, pic -> texture);
  glUniform1i(uniform_texture, 0);
  if(clear_background){
    // Clearing the destiny to transparent white before drawing
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
  }
  glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
  glBindTexture(GL_TEXTURE_2D, 0);
}
// XXX:
void print_picture(struct picture_variable *pic){
  float model_view_matrix[9] = {1.0, 0.0, 0.0,
                                0.0, 1.0, 0.0,
                                0.0, 0.0, 1.0};
  GLuint framebuffer;
  GLuint texture;
  unsigned char data[100000];
```

```
  get_new_framebuffer(&framebuffer, &texture, pic -> width, pic -> height);
  render_picture(pic, model_view_matrix, pic -> width, pic -> height, true);
  // Ler dados do framebuffer:
  glFinish();
  glReadPixels(0, 0, pic -> width, pic -> height, GL_RGBA, GL_UNSIGNED_BYTE, data);
  {
    int i, j;
    for(i = pic -> width * (pic -> height - 1) * 4;
        i >= 0; i -= (pic -> width * 4)){
      for(j = 0; j < (pic -> width * 4); j += 4)
        printf("(%hu %hu %hu %hu)", (unsigned char) data[i + j], (unsigned char)
data[i+j+1], (unsigned char) data[i+j+2], (unsigned char) data[i+j+3]);
        printf("\n");
    }
  }
}
```

Now let's declare the function that assigns the content of a source variable to a target variable, generating a new texture and copying the contents os one texture to the other:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool assign_picture_variable(struct metafont *mf, struct context *cx,
                             struct picture_variable *target,
                             struct picture_variable *source);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool assign_picture_variable(struct metafont *mf, struct context *cx,
                             struct picture_variable *target,
                             struct picture_variable *source){
  GLuint temporary_framebuffer;
  GLint previous_framebuffer;
  float model_view_matrix[9] = {1.0, 0.0, 0.0,
                                0.0, 1.0, 0.0,
                                0.0, 0.0, 1.0};
  if(target -> texture != 0)
    glDeleteTextures(1, &(target -> texture));
  glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_framebuffer);
  if(!get_new_framebuffer(&temporary_framebuffer, &(target -> texture),
                          source -> width, source -> height)){
    RAISE_ERROR_OPENGL_FRAMEBUFFER(mf, NULL, 0);
    return false;
  }
  render_picture(source, model_view_matrix, source -> width, source -> height, true);
  // Finalization:
  glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
  glBindTexture(GL_TEXTURE_2D, 0);
  glDeleteFramebuffers(1, &temporary_framebuffer);
  if(target == cx -> currentpicture){
    // The code below will be defined on Subsection 14.1:
                <Section to be inserted: Create new 'currentpicture'>
  }
```

```
  return true;
}
```

The only error that could happen above is in the OpenGL library, if it failed to create a new framebuffer for the new image variable. This should not happen during normal execution.

8.6.1. Picture Tertiary Expressions: Sum and Subtraction

The tertiary picture expressions invoves sums and subtractions:

```
<Picture Expression> -> <Picture Tertiary>
<Picture Tertiary> -> <Picture Tertiary><'+' or '-'> <Picture Secondary>
<'+' or '-'> -> + | -
```

Pictures can be added or subtracted. The result of $p_1 + p_2$ represents a new image composed by the pixels from the first image plus the pixels in the second image. And $p_1 - p_2$ means erasing from the first image the pixels from the second. The resulting image always will have the biggest height and width from the operands. And the pixel adding or subtracting will happen when the two images are centered.

The function that evaluates picture tertiary expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_picture_expression(struct metafont *mf, struct context *cx,
                             struct generic_token *begin,
                             struct generic_token *end,
                             struct picture_variable *result);
```

The function implementation tries to identify the last operator + or − in the expression. If it does not exist, then we need to evaluate the entire expression as a secondary expression. If it exists, then the left side of the operator is evaluated as a tertiary expression and the right side as a secondary expression. After this, the two results are added or subtracted:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_picture_expression(struct metafont *mf, struct context *cx,
                             struct generic_token *begin,
                             struct generic_token *end,
                             struct picture_variable *result){
  DECLARE_NESTING_CONTROL();
  struct generic_token *p = begin, *prev = NULL;
  struct generic_token *last_operator = NULL, *before_last_operator = NULL;
  while(p != end){
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && p != begin &&
       (p -> type == TYPE_SUM || p -> type == TYPE_SUBTRACT) &&
       prev -> type != TYPE_SUM && prev -> type != TYPE_SUBTRACT){
      last_operator = p;
      before_last_operator = prev;
    }
    prev = p;
    p = p -> next;
  }
  if(last_operator == NULL || before_last_operator == NULL){
    struct picture_variable a;
    struct picture_variable *sec = &a;
        <Section to be inserted: Picture: Evaluate Secondary Expression in 'sec'>
    return true;
  }
```

```
  else{
    struct picture_variable a, b;
    struct picture_variable *sec = &b;
    if(last_operator == end){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_PICTURE);
      return false;
    }
    if(!eval_picture_expression(mf, cx, begin, before_last_operator, &a))
      return false;
        <Section to be inserted: Picture: Evaluate Secondary Expression in 'sec'>
            <Section to be inserted: Image Expression: Sum or Subtract>
    if(a.texture != 0)
      glDeleteTextures(1, &(a.texture));
    if(b.texture != 0)
      glDeleteTextures(1, &(b.texture));
    return true;
  }
}
```

To add or subtract images, we first need to determine the final image area. It will have the biggest width and height from the combined images. After this, we create and initialize it as a transparent image. Both in the sum as in the subtraction we begin drawing the first operand to the destination. After this, in the case of a sum, we just draw the second image blending with the initial image by multiplying the new color (y_R, y_G, y_B, y_A) by its alpha value and adding with the previous color (x_R, x_G, x_B, x_A) multiplied by the complement of the alpha component of the new color:

$$(1 - y_A)(x_R, x_G, x_B, x_A) + y_A(y_R, y_G, y_B, y_A)$$

The subtraction case is more complex and to understand it, we need to understand the rules about how the textures are represented and how Weaver Metafont deals with them. Weaver Metafont assumes that an empty initialized picture of a given size has a fully transparent white color (represented by $(1, 1, 1, 0)$ following OpenGL convention). If we put ink in a image, we are putting a fully opaque black color (represented by $(0, 0, 0, 1)$ following OpenGL conventions). To Weaver Metafont, white transparent is the neutral operator in a subtraction: removing transparent white from a picture changes nothing. And removint opaque black removes all the color. These rules contrast with white being represented by 1 and black being represented by 0 in OpenGL. Because of this, our blending equation to combine an existing color (x_R, x_G, x_B, x_A) with a new color (y_R, y_G, y_B, y_A) is:

$$(max(x_R, 1 - y_R), max(x_G, 1 - y_G), max(x_B, 1 - y_B), x_A - y_A)$$

Unfortunately, it is not possible to compute the above equation using only OpenGL blending equations. Open GL ES 3.0 and 4 supports the usage of max function, but when using it, the blending factors are ignored. This means that it is possible to compute $max(x_R, y_R)$, but not $max(x_R, 1 - y_R)$. This inversion neeeds to be done in the shader. The fragment shader that invert the colors, but not the image alpha is:

Section: Local Variables (metafont.c) (continuation):

```
static const char fragment_shader_inverse[] =
  "#version 100\n"
  "precision mediump float;\n"
  "varying mediump vec2 texture_coordinate;\n"
  "uniform sampler2D texture1;\n"
  "void main(){\n"
  "  vec4 texture = texture2D(texture1, texture_coordinate);\n"
```

```
  "  gl_FragColor = vec4(1.0 - texture.r, 1.0 - texture.g, 1.0 - texture.b, \n"
  "                      texture.a);\n"
  "}\n";
static GLuint inv_program; // The compiled program whose code is above
static GLint uniform_inv_texture; // Texture position in the program
static GLint uniform_inv_matrix; // Matrix position in the program
```

A new shader program to render the inverse RGB must be compiled during initialization:

Section: WeaveFont Initialization (continuation):

```
{
  inv_program = compile_shader_program(vertex_shader, fragment_shader_inverse);
  uniform_inv_matrix = glGetUniformLocation(program, "model_view_matrix");
  uniform_inv_texture = glGetUniformLocation(program, "texture1");
}
```

In the finalization, we destroy the program:

Section: WeaveFont Finalization (continuation):

```
glDeleteProgram(inv_program);
```

Section: Image Expression: Sum or Subtract:

```
// Allocating and declaring data, creating transparent initial image
GLuint temporary_framebuffer = 0;
GLint previous_framebuffer;
float model_view_matrix[9] = {1.0, 0.0, 0.0,
                              0.0, 1.0, 0.0,
                              0.0, 0.0, 1.0};
result -> width = ((a.width >= b.width)?(a.width):(b.width));
result -> height = ((a.height >= b.height)?(a.height):(b.height));
glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_framebuffer);
get_new_framebuffer(&temporary_framebuffer, &(result -> texture),
                    result -> width, result -> height);
// Rendering image 'a'
model_view_matrix[0] = (double) a.width / (double) result -> width;
model_view_matrix[4] = (double) a.height / (double) result -> height;
render_picture(&a, model_view_matrix, result -> width, result -> height, true);
// Rendering image 'b'
model_view_matrix[0] = (double) b.width / (double) result -> width;
model_view_matrix[4] = (double) b.height / (double) result -> height;
if(last_operator -> type == TYPE_SUBTRACT){
  glEnable(GL_BLEND);
  // The factors in the blending equation
  glBlendFunc(GL_ONE, GL_ONE);
  // Max function for RGB and subtraction for A:
  glBlendEquationSeparate(GL_MAX, GL_FUNC_REVERSE_SUBTRACT);
  glUseProgram(inv_program);
  glUniformMatrix3fv(uniform_inv_matrix, 1, true, model_view_matrix);
  glUniform1i(uniform_inv_texture, 0);
  glBindTexture(GL_TEXTURE_2D, b.texture);
  glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
  glBlendEquation(GL_FUNC_ADD);
  glDisable(GL_BLEND);
```

```
}
else{ // Sum:
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glBlendEquation(GL_FUNC_ADD);
  render_picture(&b, model_view_matrix, result -> width, result -> height, false);
  glDisable(GL_BLEND);
}
// Finalization
glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
glBindTexture(GL_TEXTURE_2D, 0);
glDeleteFramebuffers(1, &temporary_framebuffer);
```

8.6.2. Picture Secondary Expressions: Transformers

A picture secondary expression has the following syntax:

```
<Picture Seconary> -> <Picture Primary> |
                      <Picture Secondary><Transformer>
<Transformer> -> rotated <Numeric Primary> |
                 scaled <Numeric Primary> |
                 shifted <Pair Primary> |
                 slanted <Numeric Primary> |
                 xscaled <Numeric Primary> |
                 yscaled <Numeric Primary> |
                 zscaled <Pair Primary> |
                 transformed <Transform Primary>
```

The same transformers that we can use in pairs, transforms, paths and pens also can be used in images. But how we do apply them is a little different. Like what we do for pens, we first store all the required transformations in a matrix to accumulate transformations before applying them. But contrary to pens, picture variables do not store their transform matrix.

The matrix will exist only when evaluating secondary and tertiary picture expressions. And they will be applied only after evaluating a secondary expression in the tertiary expression. This way, if we had the following code:

```
a = img totated 45 slanted 0.2 zscaled(2, 3)
```

Instead of making three potentially expensive transforms, we will accumulate all the three transforms in a matrix. When we have no more transforms, which is when we return from all secondary expression evaluations to a tertiary evaluation, we finally apply the transforms stored in the matrix. This means that the function that evaluates picture secondary expressions will have two additional parameters: a matrix pre-initialized as the identity matrix and a pointer to a boolean variable pre-initialized to false. The boolean variable will became true only if the matrix is changed during secondary evaluation.

Section: Picture: Evaluate Secondary Expression in 'sec':

```
{
  float matrix[9];
  bool modified = false;
  INITIALIZE_IDENTITY_MATRIX(matrix);
  if(last_operator == NULL){
     if(!eval_picture_secondary(mf, cx, begin, end, sec, matrix, &modified))
      return false;
  }
  else if(!eval_picture_secondary(mf, cx, last_operator -> next,
```

```
                                    end, sec, matrix, &modified))
    return false;
  if(modified){
    if(!apply_image_transformation(mf, result, sec, matrix)){
      RAISE_ERROR_OPENGL_FRAMEBUFFER(mf, cx, OPTIONAL(begin -> line));
      return false;
    }
    if(sec -> texture != 0)
      glDeleteTextures(1, &(sec -> texture));
  }
  else{
    result -> width = sec -> width;
    result -> height = sec -> height;
    result -> texture = sec -> texture;
  }
}
```

The function that evaluates secondary expressions for pictures is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_picture_secondary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
                            struct picture_variable *result,
                            float *matrix, bool *modified);
```

And its implementation involves looping over each token in the expression until finding the last secondary transform operator. If we do not find any, we evaluate everything as a primary expression. Otherwise, we evaluate everything before the last operator as a secondary expression and then we apply the transformation in the matrix:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_picture_secondary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
                            struct picture_variable *result,
                            float *matrix, bool *modified){
  DECLARE_NESTING_CONTROL();
  struct generic_token *p, *prev = NULL, *last_transformer = NULL,
                       *before_last_transformer = begin;
  p = begin;
  do{
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && (p -> type == TYPE_ROTATED ||
       p -> type == TYPE_SCALED || p -> type == TYPE_SHIFTED ||
       p -> type == TYPE_SLANTED || p -> type == TYPE_XSCALED ||
       p -> type == TYPE_YSCALED || p -> type == TYPE_ZSCALED ||
       p -> type == TYPE_TRANSFORMED)){
      last_transformer = p;
      before_last_transformer = prev;
    }
    prev = p;
    if(p != end)
```

```
      p = (struct generic_token *) p -> next;
    else
      p = NULL;
  }while(p != NULL);
  if(last_transformer == NULL)
    return eval_picture_primary(mf, cx, begin, end, result);
  else{
    if(!eval_picture_secondary(mf, cx, begin, before_last_transformer, result,
                               matrix, modified))
      return false;
    if(last_transformer -> type == TYPE_ROTATED){
                  <Section to be inserted: Picture Secondary: Rotation>
    }
    else if(last_transformer -> type == TYPE_SCALED){
                   <Section to be inserted: Picture Secondary: Scaling>
    }
    else if(last_transformer -> type == TYPE_SHIFTED){
                  <Section to be inserted: Picture Secondary: Shifting>
    }
    else if(last_transformer -> type == TYPE_SLANTED){
                  <Section to be inserted: Picture Secondary: Slanting>
    }
    else if(last_transformer -> type == TYPE_XSCALED){
                 <Section to be inserted: Picture Secondary: X-Scaling>
    }
    else if(last_transformer -> type == TYPE_YSCALED){
                 <Section to be inserted: Picture Secondary: Y-Scaling>
    }
    else if(last_transformer -> type == TYPE_ZSCALED){
                 <Section to be inserted: Picture Secondary: Z-Scaling>
    }
    else if(last_transformer -> type == TYPE_TRANSFORMED){
              <Section to be inserted: Picture Secondary: Generic Transform>
    }
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PICTURE);
    return false;
  }
}
```

Evaluating each different linear transformation means modifying the secondary expression matrix, multiplying its previous value by a new matrix. Exactly like we did for pens.

This is scale changing:

Section: Picture Secondary: Scaling:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SCALE(matrix, a.value);
*modified = true;
return true;
```

This is the code for rotation:

Section: Picture Secondary: Rotation:

```
struct numeric_variable r;
double rotation;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &r))
  return false;
rotation = 0.017453292519943295 * r.value;
TRANSFORM_ROTATE(matrix, rotation);
*modified = true;
return true;
```

The code that shifts the image by coordinate (x, y) in pixels. Before performing the linear transform in the matrix, we conver the pixels in OpenGL coordinates that depend on the picture size:

Section: Picture Secondary: Shifting:

```
struct pair_variable pair;
if(!eval_pair_primary(mf, cx, last_transformer -> next, end, &pair))
  return false;
pair.x = 2.0 * (pair.x / result -> width);
pair.y = 2.0 * (pair.y / result -> height);
TRANSFORM_SHIFT(matrix, pair.x, pair.y);
*modified = true;
return true;
```

The code that slants an image:

Section: Picture Secondary: Slanting:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SLANT(matrix, a.value);
*modified = true;
return true;
```

The code that scales an image only in the x-axis:

Section: Picture Secondary: X-Scaling:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SCALE_X(matrix, a.value);
*modified = true;
return true;
```

The code that scales the image only in axis y:

Section: Picture Secondary: Y-Scaling:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SCALE_Y(matrix, a.value);
*modified = true;
return true;
```

The z-scaling which changes the scale in the complex plane:

Section: Picture Secondary: Z-Scaling:

```
struct pair_variable pair;
if(!eval_pair_primary(mf, cx, last_transformer -> next, end, &pair))
```

```
  return false;
TRANSFORM_SCALE_Z(matrix, pair.x, pair.y);
*modified = true;
return true;
```

Finally, the generic transform that applies the linear transform stored in a transformer:

Section: Picture Secondary: Generic Transform:

```
struct transform_variable t;
if(!eval_transform_primary(mf, cx, last_transformer -> next, end, &t))
  return false;
MATRIX_MULTIPLICATION(matrix, t.value);
*modified = true;
return true;
```

Now we just need to define how do we apply the linear transformation over an image given the transform matrix, the origin picture variable and the destiny picture variable. This is done with the help of the following auxiliary function:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool apply_image_transformation(struct metafont *mf,
                                struct picture_variable *dst,
                                struct picture_variable *org,
                                float *matrix);
```

Apply the image transformation means using the transformation matrix to transform the original image to generate the new image in the destiny. For this, we need the following steps:

1) First we need to discover the size in pixels for the final image. To discover the values, we multiply the coordinates in the origin by the transformation matrix. But in this case, we measure the coordinates in pixels, not using OpenGL coordinates. If our image is a square measuring 5, one of its vertices is $(-5/2, -5/2)$ and the other will be $(5/2, 5/2)$. The resulting transformation will be the coordinates in pixels, from which we obtain the final size in pixels. We can ignore the translation in this step.

2) To take into account the translation, we should add twice the translation distance in pixels to the image size. Recall that the translation should not change the central pixel in the image. Therefore, if we shift an original image composed by a single black pixel by $(1, 1)$, we would obtain a new image 3×3, with the black pixel in the upper corner. The image center now is a white transparent pixel, which is the black pixel position before the shift.

3) Changing the size and scale is done just rendering the image in a texture with different size. So we do not need anymore the information in the matrix about scaling. We remove this from tha matrix and also correct the shifting coordinates, so that if the original image is shifted a single pixel,in the final image we get it after changing a simple pixel. This is done applying a correction over x and y positions in the matrix.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool apply_image_transformation(struct metafont *mf,
                                struct picture_variable *dst,
                                struct picture_variable *origin,
                                float *matrix){
  int i;
  GLuint temporary_framebuffer = 0;
  GLint previous_framebuffer;
  // To compute the final size, we store the greatest and smallest
  // coordinates found in the picture vertices in both axisafter
  // the transformation. (Step 1):
  float min_x = INFINITY, min_y = INFINITY, max_x = -INFINITY, max_y = -INFINITY;
  float origin_coordinates[8];
```

```
  origin_coordinates[0] = -((float) origin -> width) / 2.0;
  origin_coordinates[1] = -((float) origin -> height) / 2.0;
  origin_coordinates[2] = ((float) origin -> width) / 2.0;
  origin_coordinates[3] = -((float) origin -> height) / 2.0;
  origin_coordinates[4] = ((float) origin -> width) / 2.0;
  origin_coordinates[5] = ((float) origin -> height) / 2.0;
  origin_coordinates[6] = -((float) origin -> width) / 2.0;
  origin_coordinates[7] = ((float) origin -> height) / 2.0;
  for(i = 0; i < 8; i += 2){
    float x = LINEAR_TRANSFORM_X(origin_coordinates[i],
                                 origin_coordinates[i + 1], matrix);
    float y = LINEAR_TRANSFORM_Y(origin_coordinates[i],
                                 origin_coordinates[i + 1], matrix);
    if(x > max_x) max_x = x;
    if(x < min_x) min_x = x;
    if(y > max_y) max_y = y;
    if(y < min_y) min_y = y;
  }
  // Adjusting the image size according with translation (Step 2):
  dst -> width = (int) (max_x - min_x) +
                 (int) (origin -> width * matrix[6]);
  dst -> height = (int) (max_y - min_y) +
                  (int) (origin -> height * matrix[7]);
  // Removing from the matrix scaling data and correcting the translation
  // (Step 3):
  {
    double x_correction = ((double) origin -> width) / (double) dst -> width;
    double y_correction = ((double) origin -> height) / (double) dst -> height;
    matrix[0] = matrix[0] * x_correction;
    matrix[3] = matrix[3] * x_correction;
    matrix[6] = matrix[6] * x_correction;
    matrix[1] = matrix[1] * y_correction;
    matrix[4] = matrix[4] * y_correction;
    matrix[7] = matrix[7] * y_correction;
  }
  // Generating framebuffer and initial texture
  glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_framebuffer);
  if(!get_new_framebuffer(&temporary_framebuffer, &(dst -> texture), dst -> width,
                          dst -> height)){
    RAISE_ERROR_OPENGL_FRAMEBUFFER(mf, NULL, 0);
    return false;
  }
  // Rendering:
  render_picture(origin, matrix, dst -> width, dst -> height, true);
  // Finalization:
  glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
  glDeleteFramebuffers(1, &temporary_framebuffer);
  return true;
}
```

8.6.3. Inverters, Identity and Empty Images

The grammar for primary picture expressions is:

```
<Picture Primary> -> <Picture Variable> |
                     nullpicture <Pair Primary> |
                     ( <Picture Expression> ) |
                     <'+' or '-'> <Picture Primary> |
                     subpicture <Pair Primary> and <Pair Primary> of
                                <Picture Primary>
```

This means that we need to register a new token for "nullpicture" and "subpicture":

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_NULLPICTURE,        // Symbolic token 'nullpicture'
TYPE_SUBPICTURE,         // Symbolic token 'subpicture'
```

And this also requires adding these strings to the list of reserved keywords:

Section: List of Keywords (continuation):

```
"nullpicture", "subpicture",
```

The function that evaluates primary picture expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_picture_primary(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct picture_variable *result);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_picture_primary(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct picture_variable *result){
  if(begin == end){
    if(begin -> type == TYPE_SYMBOLIC){
                    <Section to be inserted: Primary Picture: Variable>
    }
  }
  else{
    if(begin -> type == TYPE_OPEN_PARENTHESIS &&
      end -> type == TYPE_CLOSE_PARENTHESIS){
                  <Section to be inserted: Primary Picture: Parenthesis>
    }
    else if(begin -> type == TYPE_NULLPICTURE){
                  <Section to be inserted: Primary Picture: 'nullpicture'>
    }
    else if(begin -> type == TYPE_SUM){
                    <Section to be inserted: Primary Picture: Identity>
    }
    else if(begin -> type == TYPE_SUBTRACT){
                    <Section to be inserted: Primary Picture: Inverse>
    }
    else if(begin -> type == TYPE_SUBPICTURE){
```

<Section to be inserted: Primary Picture: Subpicture>

```
    }
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PICTURE);
  return false;
}
```

The first kind of primary expression is reading an image from a variable. In this case, we need to copy the content of the variable rendering it in the destiny:

Section: Primary Picture: Variable:

```
GLuint temporary_framebuffer = 0;
GLint previous_framebuffer;
float identity_matrix[9] = {1.0, 0.0, 0.0,
                            0.0, 1.0, 0.0,
                            0.0, 0.0, 1.0};
struct symbolic_token *v = (struct symbolic_token *) begin;
struct picture_variable *content = v -> var;
if(!strcmp(v -> value, "currentpicture"))
  content = cx -> currentpicture;
if(content == NULL){
  RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(v -> line), v);
  return false;
}
if(content -> type != TYPE_T_PICTURE){
  RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(v -> line), v,
                                  ((struct variable *)(v -> var)) -> type,
                                  TYPE_T_PICTURE);
  return false;
}
if(content -> width == -1 && content -> height == -1){
  RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(v -> line), v,
                                     TYPE_T_PICTURE);
  return false;
}
// Preparing the rendering:
result -> width = content -> width;
result -> height = content -> height;
glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_framebuffer);
get_new_framebuffer(&temporary_framebuffer, &(result -> texture),
                    result -> width, result -> height);
// Rendering
render_picture(content, identity_matrix, result -> width, result -> height, true);
// Finalization:
glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
glDeleteFramebuffers(1, &temporary_framebuffer);
return true;
```

Dealing with parenthesis is exactly like what we did in other expressions. We walk over the token list until we find the last token before the closing parenthesis. The tokens between the two parenthesis are evaluated as a new picture expression:

Section: Primary Picture: Parenthesis:

```
struct generic_token *t = begin -> next;
if(begin -> next == end){
  RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '(');
  return false;
}
while(t != NULL && t -> next != end)
  t = t -> next;
return eval_picture_expression(mf, cx, begin -> next, t, result);
```

The next picture primary expression is creating an empty picture with a given size. We first evaluate the tokens after `nullpicture` token as a primary pair expression. Then, we obtain the picture size from the pair and create the texture:

Section: Primary Picture: 'nullpicture':

```
struct generic_token *begin_pair_expression, *end_pair_expression;
struct pair_variable p;
unsigned char *data;
begin_pair_expression = begin -> next;
end_pair_expression = end;
if(begin == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PAIR);
  return false;
}
if(!eval_pair_primary(mf, cx, begin_pair_expression, end_pair_expression, &p))
  return false;
result -> width = p.x;
result -> height = p.y;
data = temporary_alloc(p.x * p.y * 4);
if(data == NULL){
  RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
  return false;
}
// Making the new texture fully white
memset(data, 255, result -> width * result -> height * 4);
{ // And making it fully transparent:
  int i, size = result -> width * result -> height * 4;
  for(i = 3; i < size; i += 4)
    data[i] = 0;
}
glGenTextures(1, &(result -> texture));
glBindTexture(GL_TEXTURE_2D, result -> texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, result -> width, result -> height, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glBindTexture(GL_TEXTURE_2D, 0);
if(temporary_free != NULL)
  temporary_free(data);
return true;
```

Now the antepenultimate case: when we have a token `+` before the picture. In this case the operator

is not doing anything. It is an identity operator. So we just ignore it and evaluate the next tokens as the resulting picture:

Section: Primary Picture: Identity:

```
struct generic_token *p = begin -> next;
if(begin == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PICTURE);
  return false;
}
return eval_picture_primary(mf, cx, p, end, result);
```

For the operator `-`, which generates the inverse of a picture, we will again render activating a blend equation. This time, we will obtain the final color usaing opaque white $(1, 1, 1, 1)$ as the initial color and we render subtracting from this the new color. This is done choosing the blend factors to be one (GL_ONE) e usando subtraction as the operator (GL_FUNC_SUBTRACT):

Section: Primary Picture: Inverse:

```
struct picture_variable p;
GLuint temporary_framebuffer = 0;
GLint previous_framebuffer;
float identity_matrix[9] = {1.0, 0.0, 0.0,
                            0.0, 1.0, 0.0,
                            0.0, 0.0, 1.0};
if(begin == end){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PICTURE);
  return false;
}
if(!eval_picture_primary(mf, cx, begin -> next, end, &p))
  return false;
// Preparing to render:
result -> width = p.width;
result -> height = p.height;
glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_framebuffer);
get_new_framebuffer(&temporary_framebuffer, &(result -> texture),
                    result -> width, result -> height);
// Initializing the new texture to opaque white (1, 1, 1, 1)
glClearColor(1.0, 1.0, 1.0, 1.0);
glClear(GL_COLOR_BUFFER_BIT);
// Adjusting the blending equation to subtract the color from white opaque:
glEnable(GL_BLEND);
glBlendFuncSeparate(GL_ONE, GL_ONE, GL_ONE, GL_ONE);
glBlendEquationSeparate(GL_FUNC_REVERSE_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT);
// Rendering:
render_picture(&p, identity_matrix, result -> width, result -> height, false);
// Ending:
glDisable(GL_BLEND);
glDeleteTextures(1, &(p.texture));
glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
glDeleteFramebuffers(1, &temporary_framebuffer);
return true;
```

FInally, the primary expression to compute subpictures. This expression begins with token `subpic-`

`ture`, get a primary pair (offset in pixels), a token `and`, a second pair (with subpicture size), a token `of` and a primary expression with a picture (from where the subpicture will be extracted).

The first part for evaluate this expression is reading the two pairs and the picture from the subexpressions. The second part is extract the subpicture as requested:

Section: Primary Picture: Subpicture:

```
struct pair_variable pair_offset, subpicture_size;
struct picture_variable original_picture;
        <Section to be inserted: Subpicture: Extract Subexpressions>
          <Section to be inserted: Subimagem: Extract Subpicture>
return false;
```

Extract the subexpressions involves delimiting the begin and end for each subexpression while searching for the auxiliary tokens `of` and `and`. We also store an state to check if we are reading the tokens in the right order, or if we are in a malformed expressions where the tokens are present, but in the wrong order, or without subexpressions:

Section: Subpicture: Extract Subexpressions:

```
{
  DECLARE_NESTING_CONTROL();
  struct generic_token *begin_pair1 = NULL, *end_pair1 = NULL,
                       *begin_pair2 = NULL, *end_pair2 = NULL,
                       *begin_pic = NULL, *end_pic = NULL,
                       *p = begin -> next,
                       *last_token = begin;
  int state = 0;
  begin_pair1 = p;
  while(p != end && p != NULL){
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && p -> type == TYPE_AND){
      if(state != 0 || last_token -> type == TYPE_SUBPICTURE){
        RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(p -> line), p);
        return false;
      }
      end_pair1 = last_token;
      state ++;
      begin_pair2 = p -> next;
    }
    else if(IS_NOT_NESTED() && p -> type == TYPE_OF){
      if(state != 1 || last_token -> type == TYPE_AND){
        RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(p -> line), p);
        return false;
      }
      end_pair2 = last_token;
      state ++;
      begin_pic = p -> next;
    }
    last_token = p;
    p = p -> next;
  }
  if(p == NULL){
    if(state < 2){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
```

```
                                    TYPE_T_PAIR);
    }
    else{
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_PICTURE);
    }
    return false;
  }
  end_pic = p;
  if(!eval_pair_primary(mf, cx, begin_pair1, end_pair1, &pair_offset))
    return false;
  if(!eval_pair_primary(mf, cx, begin_pair2, end_pair2, &subpicture_size))
    return false;
  if(!eval_picture_primary(mf, cx, begin_pic, end_pic, &original_picture))
    return false;
}
```

Extract the subpicture requires generating a new texture with the size indicated by the second pair and rendering the original image in the new texture changing its size and offset according with what is stored in the two pairs:

Section: Subimagem: Extract Subpicture:

```
{
  GLuint temporary_framebuffer = 0;
  GLint previous_framebuffer;
  float render_matrix[9];
  INITIALIZE_IDENTITY_MATRIX(render_matrix);
  glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_framebuffer);
  get_new_framebuffer(&temporary_framebuffer, &(result -> texture),
                      subpicture_size.x, subpicture_size.y);
  result -> width = subpicture_size.x;
  result -> height = subpicture_size.y;
  // Initializing the new texture as transparent white (1, 1, 1, 0)
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  // Adjusting the size to render in the new texture
  render_matrix[0] = original_picture.width / subpicture_size.x;
  render_matrix[4] = original_picture.height / subpicture_size.y;
  // Adjusting the offset in the render matrix
  render_matrix[6] = -2.0 * (pair_offset.x +
                      0.5 * (subpicture_size.x - original_picture.width)) /
                      subpicture_size.x;
  render_matrix[7] = -2.0 * (pair_offset.y +
                      0.5 * (subpicture_size.y - original_picture.height)) /
                      subpicture_size.y;
  // Rendering:
  render_picture(&original_picture, render_matrix, result -> width, result -> height,
                 false);
```

```
  // Finalization:
  glDisable(GL_BLEND);
  glDeleteTextures(1, &(original_picture.texture));
```

```
  glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
  glDeleteFramebuffers(1, &temporary_framebuffer);
  return true;
}
```

8.6.4. Pictures in Numeric Expressions

There are cases where we need to evaluate a picture expression inside expressions of other types. We have three additional numeric expressions:

```
<Numeric Primary> -> totalweight <Picture Primary> |
                     width <Picture Primary> |
                     height <Picture Primary>
```

The two last operators just return respectively the picture width and height. The operator `totalweight` does is evaluate a picture expression, get the resulting picture and return the sum of its pixels weight. Each pixel weight is computed as the euclidean distance between its RGB component and the color white, and then normalizing the result dividing by $\sqrt{3}$. So the color white has weight 0 and the black color has weight 1. The computed value is then multiplied by the alpha component.

First we need a new token types for the three new operators:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_TOTALWEIGHT,        // Symbolic token 'totalweight'
TYPE_WIDTH,              // Symbolic token 'width'
TYPE_HEIGHT,             // Symbolic token 'height'
```

And we add their names to the list of keywords:

Section: List of Keywords (continuation):

```
"totalweight", "width", "height",
```

For the first operator, we will recover the pixels from a texture first rendering the texture to a temporary framebuffer and then reading the valuesd using function `glReadPixels` to get the framebuffer content:

Section: Numeric Primary: Additional Operators (continuation):

```
else if(begin -> type == TYPE_TOTALWEIGHT){
  struct picture_variable p;
  char *data;
  GLuint temporary_framebuffer = 0;
  GLint previous_framebuffer;
  GLuint temporary_texture = 0;
  float identity_matrix[9] = {1.0, 0.0, 0.0,
                              0.0, 1.0, 0.0,
                              0.0, 0.0, 1.0};
  if(begin == end){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PICTURE);
    return false;
  }
  if(!eval_picture_primary(mf, cx, begin -> next, end, &p))
    return false;
  data = temporary_alloc(p.width * p.height * 4);
  if(data == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
```

```
  glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_framebuffer);
  if(!get_new_framebuffer(&temporary_framebuffer, &(temporary_texture),
                          p.width, p.height)){
    RAISE_ERROR_OPENGL_FRAMEBUFFER(mf, NULL, 0);
    return false;
  }
  // Rendering:
  render_picture(&p, identity_matrix, p.width, p.height, true);
  // Reading framebuffer data:
  glReadPixels(0, 0, p.width, p.height, GL_RGBA, GL_UNSIGNED_BYTE, data);
  {
    int i, size = p.width * p.height * 4;
    double sum = 0.0;
    for(i = 0; i < size; i += 4){
      // If the values are equal, let's avoid rounding errors:
      if(data[i] == data[i+1] && data[i+1] == data[i+2]){
        sum += ((255 - (unsigned char) data[i]) / 255.0) *
               (((unsigned char) data[i+3]) / 255.0);
      }
      else{
        double r = ((255 - (unsigned char) data[i]) / 255.0) * 0.2989,
               g = ((255 - (unsigned char) data[i+1]) / 255.0) * 0.5870,
               b = ((255 - (unsigned char) data[i+2]) / 255.0) * 0.1140,
               a = ((unsigned char) data[i+3]) / 255.0;
        sum += ((r+g+b) * a);
      }
    }
    result -> value = sum;
  }
  // Ending:
  if(temporary_free != NULL)
    temporary_free(data);
  glDeleteTextures(1, &temporary_texture);
  glDeleteTextures(1, &(p.texture));
  glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
  glDeleteFramebuffers(1, &temporary_framebuffer);
  return true;
}
```

For the next two operators, `width` and `height`, everything is simpler. We just need to get the image and get their width and height respectively. For the width:

Section: Numeric Primary: Additional Operators (continuation):

```
else if(begin -> type == TYPE_WIDTH){
  struct picture_variable p;
  if(begin == end){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PICTURE);
    return false;
  }
  if(!eval_picture_primary(mf, cx, begin -> next, end, &p))
    return false;
```

```
  result -> value = (float) p.width;
  return true;
}
```

And for the height:

Section: Numeric Primary: Additional Operators (continuation):

```
else if(begin -> type == TYPE_HEIGHT){
  struct picture_variable p;
  if(begin == end){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PICTURE);
    return false;
  }
  if(!eval_picture_primary(mf, cx, begin -> next, end, &p))
    return false;
  result -> value = (float) p.height;
  return true;
}
```

8.7. Boolean Assignments and Expressions

Like for all other expressions and assignments, we begin defining the code to assign variables after evaluate an expression:

Section: Assignment for Boolean Variables:

```
else if(type == TYPE_T_BOOLEAN){
  int i;
  bool ret;
  struct boolean_variable result;
  ret = eval_boolean_expression(mf, cx, begin_expression, *end, &result);
  if(!ret)
    return false;
  var = (struct symbolic_token *) begin;
  for(i = 0; i < number_of_variables; i ++){
    ((struct boolean_variable *) var -> var) -> value = result.value;
    var = (struct symbolic_token *) (var -> next);
    var = (struct symbolic_token *) (var -> next);
  }
}
```

Now that we can assign the result of expressions, we can write the code to proper evaluate the expressions.

8.7.1. Comparisons

Comparisons are made using relations. They allow us to check if two values are equal or not, and also which value is bigger or smaller. The grammar rules to compare values are:

```
<Boolean Expression> -> <Boolean Tertiary> |
                        <Numeric Expression> <Relation> <Numeric Tertiary> |
                        <Pair Expression> <Relation> <Pair Tertiary>       |
                        <Boolean Expression> <Relation> <Boolean Tertiary> |
                        <Transform Expression> <Relation> <Transform Tertiary>
<Relation> -> < | <= | > | >= | = | <>
```

This mean that comparisons using these relations are in fact quarternary expressions. They have a precedence order even smaller that tertiary operators. For other types of expressions, we considered a tertiary expression as synonym for expressions of that type. But for boolean expressions, tertiary expressions are an inner type for general boolean expressions.

The relations above require new token types to represent them:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_LT,                    // 0 token simblico '<'
TYPE_LEQ,                   // 0 token simblico '<='
TYPE_GT,                    // 0 token simblico '>'
TYPE_GEQ,                   // 0 token simblico '>='
TYPE_NEQ,                   // 0 token simblico '<>'
```

And we also add their names to the list of reserved keywords:

Section: List of Keywords (continuation):

```
"<", "<=", ">", ">=", "<>",
```

The declaration for the funtion that will evaluate boolean expressions is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_boolean_expression(struct metafont *mf, struct context *cx,
                             struct generic_token *begin,
                             struct generic_token *end,
                             struct boolean_variable *result);
```

This function shall check if we have one of the relation operators. If not, the entire expression is evaluated as a tertiary booleana expression. If we have one, we first need to take the rightmost operator and evaluate it after evaluating the other two expressions that produce its operands. But to evaluate an expression, we need to know its type. We will assume that we have a function that, given an tertiary expression, returns its type. We will call it `get_tertiary_expression_type`. We will define it later, in Subsection 8.8. But for now we will just assume that the function exists.

The implementation is:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_boolean_expression(struct metafont *mf, struct context *cx,
                             struct generic_token *begin,
                             struct generic_token *end,
                             struct boolean_variable *result){
  DECLARE_NESTING_CONTROL();
  struct generic_token *p = begin, *prev = NULL;
  struct generic_token *last_operator = NULL, *before_last_operator = NULL;
  while(p != end){
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && p != begin &&
       (p -> type == TYPE_LT || p -> type == TYPE_LEQ ||
        p -> type == TYPE_GT || p -> type == TYPE_GEQ ||
        p -> type == TYPE_NEQ || p -> type == TYPE_EQUAL)){
      last_operator = p;
      before_last_operator = prev;
    }
    prev = p;
    p = p -> next;
  }
  if(last_operator == NULL)
```

```
  return eval_boolean_tertiary(mf, cx, begin, end, result);
else{
  int type;
  if(before_last_operator == NULL || last_operator == end){
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_BOOLEAN);
    return false;
  }
  type = get_tertiary_expression_type(mf, cx,  last_operator -> next, end);
  switch(type){
  case TYPE_T_NUMERIC:
  {
    struct numeric_variable a, b;
    if(!eval_numeric_expression(mf, cx, begin, before_last_operator, &a))
      return false;
    if(!eval_numeric_expression(mf, cx, last_operator -> next, end, &b))
      return false;
    switch(last_operator -> type){
    case TYPE_LT:
      result -> value = a.value < b.value;
      return true;
    case TYPE_LEQ:
      result -> value = a.value <= b.value;
      return true;
    case TYPE_GT:
      result -> value = a.value > b.value;
      return true;
    case TYPE_GEQ:
      result -> value = a.value >= b.value;
      return true;
    case TYPE_EQUAL:
      result -> value = a.value == b.value;
      return true;
    case TYPE_NEQ:
      result -> value = a.value != b.value;
      return true;
    }
    break;
  }
  case TYPE_T_PAIR:
  {
    struct pair_variable a, b;
    if(!eval_pair_expression(mf, cx, begin, before_last_operator, &a))
      return false;
    if(!eval_pair_expression(mf, cx, last_operator -> next, end, &b))
      return false;
    switch(last_operator -> type){
    case TYPE_LT:
      result -> value = (a.x < b.x) || (a.x == b.x && a.y < b.y);
      return true;
    case TYPE_LEQ:
```

```
      result -> value = (a.x < b.x) || (a.x == b.x && a.y <= b.y);
      return true;
    case TYPE_GT:
      result -> value = (a.x > b.x) || (a.x == b.x && a.y > b.y);
      return true;
    case TYPE_GEQ:
      result -> value = (a.x > b.x) || (a.x == b.x && a.y > b.y);
      return true;
    case TYPE_EQUAL:
      result -> value = (a.x == b.x && a.y == b.y);
      return true;
    case TYPE_NEQ:
      result -> value = (a.x != b.x || a.y != b.y);
      return true;
    }
    break;
  }
  case TYPE_T_TRANSFORM:
  {
    struct transform_variable a, b;
    int i, order[6] = {6, 7, 0, 3, 1, 4};
    if(!eval_transform_expression(mf, cx, begin, before_last_operator, &a))
      return false;
    if(!eval_transform_expression(mf, cx, last_operator -> next, end, &b))
      return false;
    switch(last_operator -> type){
      case TYPE_LT:
        for(i = 0; i < 5; i ++)
          if(a.value[order[i]] != b.value[order[i]]){
            result -> value = (a.value[order[i]] < b.value[order[i]]);
            return true;
          }
        result -> value = (a.value[order[i]] < b.value[order[i]]);
        return true;
      case TYPE_LEQ:
        for(i = 0; i < 5; i ++)
          if(a.value[order[i]] != b.value[order[i]]){
            result -> value = (a.value[order[i]] < b.value[order[i]]);
            return true;
          }
        result -> value = (a.value[order[i]] <= b.value[order[i]]);
        return true;
      case TYPE_GT:
        for(i = 0; i < 5; i ++)
          if(a.value[order[i]] != b.value[order[i]]){
            result -> value = (a.value[order[i]] > b.value[order[i]]);
            return true;
          }
        result -> value = (a.value[order[i]] > b.value[order[i]]);
        return true;
      case TYPE_GEQ:
```

```
        for(i = 0; i < 5; i ++)
          if(a.value[order[i]] != b.value[order[i]]){
            result -> value = (a.value[order[i]] > b.value[order[i]]);
            return true;
          }
        result -> value = (a.value[order[i]] >= b.value[order[i]]);
        return true;
      case TYPE_EQUAL:
        for(i = 0; i < 5; i ++)
          if(a.value[order[i]] != b.value[order[i]]){
            result -> value = false;
            return true;
          }
        result -> value = (a.value[order[i]] == b.value[order[i]]);
        return true;
      case TYPE_NEQ:
        for(i = 0; i < 5; i ++)
          if(a.value[order[i]] != b.value[order[i]]){
            result -> value = true;
            return true;
          }
        result -> value = (a.value[order[i]] != b.value[order[i]]);
        return true;
    }
    break;
  }
  case TYPE_T_BOOLEAN:
  {
    struct boolean_variable a, b;
    a.value = b.value = -1;
    if(!eval_boolean_expression(mf, cx, begin, before_last_operator, &a))
      return false;
    if(!eval_boolean_tertiary(mf, cx, last_operator -> next, end, &b))
      return false;
    switch(last_operator -> type){
    case TYPE_LT:
      result -> value = a.value < b.value;
      return true;
    case TYPE_LEQ:
      result -> value = a.value <= b.value;
      return true;
    case TYPE_GT:
      result -> value = a.value > b.value;
      return true;
    case TYPE_GEQ:
      result -> value = a.value >= b.value;
      return true;
    case TYPE_EQUAL:
      result -> value = (a.value == b.value);
      return true;
    case TYPE_NEQ:
```

```
        result -> value = (a.value != b.value);
        return true;
      }
      break;
    }
    default:
      RAISE_ERROR_INVALID_COMPARISON(mf, cx, OPTIONAL(begin -> line),
                                     last_operator, type);
      return false;
    }
    return true;
  }
}
```

8.7.2. Operation OR

The boolean operator OR is the sole tertiary boolean operator. The grammar for boolean tertiary expressions is:

```
<Boolean Tertiary> -> <Boolean Tertiary> or <Boolean Secondary> |
                      <Boolean Secondary>
```

To implement the operator, we need to define a new symbolic token representing the `or` opeartor:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_OR,                    // Symbolic token 'or'
```

Which should be stored in the list of reserved keywords:

Section: List of Keywords (continuation):

```
"or",
```

Now we declare the function that evaluates secondary boolean expressions:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_boolean_tertiary(struct metafont *mf, struct context *cx,
                           struct generic_token *begin,
                           struct generic_token *end,
                           struct boolean_variable *result);
```

This function implementation follows the expected model from previous similar functions. It walks over the list of tokens trying to find the last non-nested `or` operator. Then, it applies the operator over the results of the two subexpressions that delimits it. If there ire no `or` operators, then the entire expression is evaluated as a secondary boolean operator:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_boolean_tertiary(struct metafont *mf, struct context *cx,
                           struct generic_token *begin,
                           struct generic_token *end,
                           struct boolean_variable *result){
  DECLARE_NESTING_CONTROL();
  struct generic_token *p = begin, *prev = NULL;
  struct generic_token *last_operator = NULL, *before_last_operator = NULL;
  while(p != end){
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && p != begin && p -> type == TYPE_OR){
      last_operator = p;
```

```
      before_last_operator = prev;
    }
    prev = p;
    p = p -> next;
  }
  if(last_operator == NULL)
    return eval_boolean_secondary(mf, cx, begin, end, result);
  else{
    if(last_operator == end){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_BOOLEAN);
      return false;
    }
    struct boolean_variable a, b;
    a.value = b.value = -1;
    if(!eval_boolean_tertiary(mf, cx, begin, before_last_operator, &a))
      return false;
    if(!eval_boolean_secondary(mf, cx, last_operator -> next, end, &b))
      return false;
    result -> value =  (a.value || b.value);
    return true;
  }
}
```

8.7.3. Operator AND

The boolean operator AND is the sole boolean secondary operator. The grammar for secondary boolean expressions is:

```
<Boolean Secondary> -> <Boolean Secondary> and <Boolean Primary> |
                       <Boolean Primary>
```

The token and was alereadly defined. We used it to separate two control points when describing a path curve.

The function that evaluates secondary boolean operators is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_boolean_secondary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
                            struct boolean_variable *result);
```

The function works exactly as the tertiary function that evaluates the operator OR. The difference is that it works searching for AND operators and applying the leftmost AND operator over the delimiting subexpressions. If there are no AND operators, the entire expression is evaluated as a primary boolean expression:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_boolean_secondary(struct metafont *mf, struct context *cx,
                            struct generic_token *begin,
                            struct generic_token *end,
                            struct boolean_variable *result){
  DECLARE_NESTING_CONTROL();
  struct generic_token *p = begin, *prev = NULL;
```

```
  struct generic_token *last_operator = NULL, *before_last_operator = NULL;
  while(p != end){
    COUNT_NESTING(p);
    if(IS_NOT_NESTED() && p != begin && p -> type == TYPE_AND){
      last_operator = p;
      before_last_operator = prev;
    }
    prev = p;
    p = p -> next;
  }
  if(last_operator == NULL)
    return eval_boolean_primary(mf, cx, begin, end, result);
  else{
    if(last_operator == end){
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_BOOLEAN);
      return false;
    }
    struct boolean_variable a, b;
    a.value = b.value = -1;
    if(!eval_boolean_secondary(mf, cx, begin, before_last_operator, &a))
      return false;
    if(!eval_boolean_primary(mf, cx, last_operator -> next, end, &b))
      return false;
    result -> value = (a.value && b.value);
    return true;
  }
}
```

8.7.4. Boolean Literals, Variables, NOT and Simple Predicates

The grammar for boolean primary expressions is:

```
<Boolean Primary> -> <Boolean Variable> | true | false |
                     cycle <Path Primary> | odd <Numeric Primary> |
                     not <Boolean Primary> |
                     ( <Boolean Expression> )
```

Most of these expressions have obvious meanings. A boolean variable is evaluated to what vaue it has stored. The values `true` and `false` evaluated to the boolean value correspinding to their names. The `not` operator is the boolean NOT. Parenthesis can be used to change the order of evaluation for operators. The token `odd` checks if the following numeric is odd after rounding it to the nearest integer. And `cycle` checks if a path is cyclic.

The following new tokens are defined:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_TRUE,                  // 0 token simblico 'true'
TYPE_FALSE,                 // 0 token simblico 'false'
TYPE_ODD,                   // 0 token simblico 'odd'
TYPE_NOT,                   // 0 token simblico 'not'
```

Each of their names should be added to the list of reserved kyewords:

Section: List of Keywords (continuation):

```
"true", "false", "odd", "not",
```

The function that evaluates boolean primaries is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool eval_boolean_primary(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct boolean_variable *result);
```

In boolean primary expressions, it is possible to determine which grammar rule to follow just checking the first token. Therefore, this function just checks the first token and apply the correct rule:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool eval_boolean_primary(struct metafont *mf, struct context *cx,
                          struct generic_token *begin,
                          struct generic_token *end,
                          struct boolean_variable *result){
  struct symbolic_token *symbol;
  struct path_variable path;
  struct numeric_variable num;
  struct boolean_variable b;
  switch(begin -> type){
    case TYPE_SYMBOLIC: // Varivel
      symbol = ((struct symbolic_token *) begin);
      struct boolean_variable *var = symbol -> var;
      if(var == NULL){
        RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(begin -> line),
                                        symbol);
        return false;
      }
      if(var -> type != TYPE_T_BOOLEAN){
        RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(begin -> line),
                                        symbol, var -> type,
                                        TYPE_T_BOOLEAN);
        return false;
      }
      if(var -> value == -1){
        RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(begin -> line),
                                           symbol, TYPE_T_BOOLEAN);
        return false;
      }
      result -> value = var -> value;
      return true;
    break;
    case TYPE_TRUE: // True
      if(end != begin){
        RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                       TYPE_T_BOOLEAN);
        return false;
      }
      result -> value = 1;
      return true;
```

```
    break;
    case TYPE_FALSE: // Falso
      if(end != begin){
        RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                       TYPE_T_BOOLEAN);
        return false;
      }
      result -> value = 0;
      return true;
    break;
    case TYPE_CYCLE: // 'cycle'
      if(!eval_path_primary(mf, cx, begin -> next, end, &path))
        return false;
      result -> value = path.cyclic;
      if(temporary_free != NULL)
        path_recursive_free(temporary_free, &path, false);
      return true;
    break;
    case TYPE_ODD: // 'odd'
      if(!eval_numeric_primary(mf, cx, begin -> next, end, &num))
        return false;
      result -> value = (((int) round(num.value)) % 2);
      return true;
    break;
    case TYPE_NOT: // 'not'
      if(!eval_boolean_primary(mf, cx, begin -> next, end, &b))
        return false;
      result -> value = !(b.value);
      return true;
    break;
    case TYPE_OPEN_PARENTHESIS: // '('
      if(end -> type != TYPE_CLOSE_PARENTHESIS){
        RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                       TYPE_T_BOOLEAN);
        return false;
      }
      struct generic_token *last_token = begin;
      while(last_token -> next != end)
        last_token = last_token -> next;
      return eval_boolean_expression(mf, cx, begin -> next, last_token,
                                     result);
    break;
    default:
      RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_BOOLEAN);
      return false;
  }
}
```

Several times we have an expression and want to identify its type. For example, in boolean expressions we can compare values from numeric, pair, transform and boolean expressions. In a numeric expression we can use `length` operator over numeric or pair expressions. We can multiply between two numeric expressions or between a numeric and a pair expression. Operators like `xpart` works in pairs and transforms.

Notice that in none of the cases above we expect to identify a picture or pen expression. Therefore, identify these expressions is less prioritary. However, identifying them still is useful, as it allow us to produce more useful error messages, warning the user that we found an expression of some specific type instead of another expected type.

Identify the type of some expression can be complex. But the most common cases are simple and can be evaluated fast. We will use different functions to evaluate the type for primary, secondary and tertiary expressions:

Section: Local Function Declaration (metafont.c) (continuation):

```
int get_primary_expression_type(struct metafont *mf, struct context *cx,
                                struct generic_token *begin_expr,
                                struct generic_token *end_expr);
int get_secondary_expression_type(struct metafont *mf, struct context *cx,
                                  struct generic_token *begin_expr,
                                  struct generic_token *end_expr);
int get_tertiary_expression_type(struct metafont *mf, struct context *cx,
                                 struct generic_token *begin_expr,
                                 struct generic_token *end_expr);
```

To identify primary expressions, first we can treat the simplest cases (the expression have a single token) and then we deal with the more complex cases. The rules that allow us to identify the types are:

1) A variable is an expression with the variable type. Only some special variables do not have a type encoded because they are temporary variables and they are handled differently.

2) The expression is boolean if it is the single token `true` or `false`. Or if it begins with `cycle`, `odd` or `not`.

3) The expression is a pair if it begins with `point`, `precontrol` or `postcontrol`. Or if in the expression we have a `[`. And also if it begins with a numeric token and have parenthesis or the beginning of a primary pair.

4) The expressison is numeric if it has a single numeric token or the token `normaldeviate`. Or if it is a numeric token followed by `/`. Or if it begins with `length`, `xpart`, `ypart`, `xxpart`, `xypart`, `yxpart`, `yypart`, `angle`, `sqrt`, `sind`, `cosd`, `log`, `mexp`, `floor`, `uniformdeviate`.

5) If we have a `+` ou `-`, we discover the type ignoring this first token and checking the type of the remaining expression.

6) If the first token is `subpath`, `makepath` or `reverse`, it is a path expression.

7) If the first token is `nullpen`, `pencircle`, `pensemicircle` or `makepen`, then it is a pen expression.

8) If the first token is `nullpicture` or `subpicture`, it is a picture expression.

9) If we begin and end with parenthesis, and have no comma, we should evaluate the type of the tertiary expression inside the parenthesis. If we have a single comma, we have a pair. If we have five commas, we have a transform.

10) In the other cases, we have an unknown expression and some error may be occurred.

The code that tries to encompass these rules (but not in the above order) is:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
int get_primary_expression_type(struct metafont *mf, struct context *cx,
                                struct generic_token *begin_expr,
                                struct generic_token *end_expr){
  if(begin_expr == end_expr){
    if(begin_expr -> type == TYPE_SYMBOLIC){ // Case 1
      struct variable *var = (struct variable *)
```

```
                              ((struct symbolic_token *) begin_expr) -> var;
    if(var == NULL){
      if(!strcmp(((struct symbolic_token *) begin_expr) -> value, "w") ||
         !strcmp(((struct symbolic_token *) begin_expr) -> value, "h") ||
         !strcmp(((struct symbolic_token *) begin_expr) -> value, "d"))
        return TYPE_T_NUMERIC;
      if(!strcmp(((struct symbolic_token *) begin_expr) -> value,
                 "currentpen"))
        return TYPE_T_PEN;
      if(!strcmp(((struct symbolic_token *) begin_expr) -> value,
                 "currentpicture"))
        return TYPE_T_PICTURE;
      return -1;
    }
    else
      return var -> type;
  }
  if(begin_expr -> type == TYPE_TRUE || begin_expr -> type == TYPE_FALSE ||
     begin_expr -> type == TYPE_NOT) // Case 2
    return TYPE_T_BOOLEAN;
  if(begin_expr -> type == TYPE_NUMERIC ||
     begin_expr -> type == TYPE_NORMALDEVIATE) // Case 4
    return TYPE_T_NUMERIC;
  if(begin_expr -> type == TYPE_NULLPEN || // Case 7
     begin_expr -> type == TYPE_PENCIRCLE ||
     begin_expr -> type == TYPE_PENSEMICIRCLE)
     return TYPE_T_PEN;
  else
    return -1;
}
else{
  if(begin_expr -> type == TYPE_CYCLE || begin_expr -> type == TYPE_ODD ||
     begin_expr -> type == TYPE_NOT) // Case 2
    return TYPE_T_BOOLEAN;
  // Pair expression (case 3):
  if(begin_expr -> type == TYPE_POINT ||
     begin_expr -> type == TYPE_PRECONTROL ||
     begin_expr -> type == TYPE_POSTCONTROL ||
     begin_expr -> type == TYPE_BOT || begin_expr -> type == TYPE_TOP ||
     begin_expr -> type == TYPE_LFT || begin_expr -> type == TYPE_RT)
    return TYPE_T_PAIR;
  // Numeric expression (Case 4):
  if(begin_expr -> type == TYPE_LENGTH || begin_expr -> type == TYPE_XPART ||
     begin_expr -> type == TYPE_YPART || begin_expr -> type == TYPE_ANGLE ||
     begin_expr -> type == TYPE_XXPART || begin_expr -> type == TYPE_FLOOR ||
     begin_expr -> type == TYPE_XYPART || begin_expr -> type == TYPE_SIND ||
     begin_expr -> type == TYPE_YXPART || begin_expr -> type == TYPE_SQRT ||
     begin_expr -> type == TYPE_YYPART || begin_expr -> type == TYPE_LOG ||
     begin_expr -> type == TYPE_COSD ||   begin_expr -> type == TYPE_EXP ||
     begin_expr -> type == TYPE_UNIFORMDEVIATE) // Caso 4
    return TYPE_T_NUMERIC;
```

```
  // Case 5:
  if(begin_expr -> type == TYPE_SUM || begin_expr -> type == TYPE_SUBTRACT)
    return get_primary_expression_type(mf, cx, begin_expr -> next, end_expr);
  // Numeric token, it could be a numeric or pair expression:
  if(begin_expr -> type == TYPE_NUMERIC){
    struct generic_token *t = begin_expr;
    while(t != NULL && t != end_expr){
      if(t -> type == TYPE_OPEN_BRACKETS || t -> type == TYPE_PRECONTROL ||
         t -> type == TYPE_OPEN_PARENTHESIS || t -> type == TYPE_POINT ||
         t -> type == TYPE_POSTCONTROL)
        return TYPE_T_PAIR;
      t = t -> next;
    }
    return TYPE_T_NUMERIC;
  }
  // Case 6:
  if(begin_expr -> type == TYPE_SUBPATH ||
     begin_expr -> type == TYPE_MAKEPATH ||
     begin_expr -> type == TYPE_REVERSE)
    return TYPE_T_PATH;
  if(begin_expr -> type == TYPE_MAKEPEN) // Case 7
    return TYPE_T_PEN;
  if(begin_expr -> type == TYPE_NULLPICTURE || // Case 8
     begin_expr -> type == TYPE_SUBPICTURE)
    return TYPE_T_PICTURE;
  // The remaining cases not encompassed by the more general rules
  // are expressions that begins with parenthesis but do not end
  // with them: (1+1)[p1, p1] and also: normaldeviate[p1, p2]
  if((begin_expr -> type == TYPE_OPEN_PARENTHESIS &&
      end_expr -> type != TYPE_CLOSE_PARENTHESIS) ||
     begin_expr -> type == TYPE_NORMALDEVIATE)
    return TYPE_T_PAIR;
  if(begin_expr -> type == TYPE_OPEN_PARENTHESIS &&
     end_expr -> type == TYPE_CLOSE_PARENTHESIS &&
     begin_expr -> next != end_expr){
    DECLARE_NESTING_CONTROL();
    int number_of_commas = 0;
    struct generic_token *t = begin_expr -> next;
    while(t != NULL && t -> next != end_expr){
      COUNT_NESTING(t);
      if(IS_NOT_NESTED() && t -> type == TYPE_COMMA)
        number_of_commas ++;
      t = t -> next;
    }
    if(number_of_commas == 0)
      return get_tertiary_expression_type(mf, cx, begin_expr -> next, t);
    else if(number_of_commas == 1)
      return TYPE_T_PAIR;
    else if(number_of_commas == 5)
      return TYPE_T_TRANSFORM;
  }
```

```
    return -1;
  }
}
```

Now the secondary expressions. The rules that we will use are:

1) If we have a single token, we evaluate everything as a primary expression.

2) If we have `and`, the expression is boolean.

3) If we have a transformer (`transformed`, `rotated`, `scaled`, `shifted`, `slanted`, `xscaled`, `yscaled`, `zscaled`), we ignore it and all the tokens after it and re-evaluate the expression. In this case we will have either a transform, a pair, a path, a pen or a picture.

4) In case of multiplications and divisions, the most important is the operator more at the right. In the case of multiplication, if one of the operands is a pair, then we have a pair expression. If both are numeric, we have a numeric expression. In the case of division, the left operand type is the operation type. A token `/` is a division when it is not delimited by numeric tokens. If it is, then we have a fraction, not a division operator. Except when one of the tokens is already part of some fraction: in this case, we have a division operator. Therefore, 1/3 is a fraction, not a division. But 1/3/1/3 is a single division that divide two fractions.

5) In the other cases, if we cannot identify the type, we try again interpreting the entire expression as a primary expression.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
int get_secondary_expression_type(struct metafont *mf, struct context *cx,
                                  struct generic_token *begin_expr,
                                  struct generic_token *end_expr){
  DECLARE_NESTING_CONTROL();
  struct generic_token *t = begin_expr, *prev = NULL, *last_op = NULL;
  struct generic_token *last_fraction = NULL;
  struct generic_token *before_last_op = NULL, *prev_prev = NULL;
  if(begin_expr == end_expr)
    return get_primary_expression_type(mf, cx, begin_expr, end_expr);
  while(t != end_expr && t != NULL){
    COUNT_NESTING(t);
    if(IS_NOT_NESTED()){
      if(t -> type == TYPE_AND)
        return TYPE_T_BOOLEAN;
      if(t -> type == TYPE_TRANSFORMED || t -> type == TYPE_ROTATED ||
         t -> type == TYPE_SCALED || t -> type == TYPE_SHIFTED ||
         t -> type == TYPE_SLANTED || t -> type == TYPE_XSCALED ||
         t -> type == TYPE_YSCALED || t -> type == TYPE_ZSCALED){
        if(prev == NULL)
          return -1;
        return get_secondary_expression_type(mf, cx, begin_expr, prev);
      }
      if(t -> type == TYPE_MULTIPLICATION || t -> type == TYPE_DIVISION){
        if(t -> type == TYPE_DIVISION && prev -> type == TYPE_NUMERIC &&
           t != end_expr &&
           ((struct generic_token *) t -> next) -> type != TYPE_NUMERIC &&
           last_fraction != prev_prev)
          last_fraction = t;
        else{
          last_op = t;
          before_last_op = prev;
        }
```

```
      }
    }
    prev_prev = prev;
    prev = t;
    t = t -> next;
  }
  if(last_op != NULL){
    int s = get_primary_expression_type(mf, cx, last_op -> next, end_expr) +
            get_secondary_expression_type(mf, cx, begin_expr, before_last_op);
    if(s == 2 * TYPE_T_NUMERIC)
      return TYPE_T_NUMERIC;
    else if(s == TYPE_T_NUMERIC + TYPE_T_PAIR)
      return TYPE_T_PAIR;
    else return -1;
  }
  else return get_primary_expression_type(mf, cx, begin_expr, end_expr);
}
```

Finally, the tertiary expressions. In this case, the rules are:

1) If we have a single token, evaluate its type as a primary expression.

2) If we have an `or` or a relation (`<`, `<=`, `>`, `>=`, `=`, `<>`), then it is a boolean expression. Technically, the relations are not tertiary expressions, but quarternary ones. But we can deal with them here, as we are not evaluating the expression value, where it is mandatory to follow precedence rules.

3) If we have tokens from path joining operations, like `&`, `..` or `--`, then it also is not a tertiary expression, but an expression with even less precedence. Anyway, we can return the information that we have a path expression. The same happens if we find a token that open braces, as they are used only for direction specifiers in path expressions.

4) If we have a pyhtagoric sum or subtraction, (`++`, `+-+`), it is a numeric expression.

5) In the case of a sum or subtraction, the expression type is the same that its operands type.

6) I other cases, we try again to check the type, but interpreting the expression as a secondary expression.

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
int get_tertiary_expression_type(struct metafont *mf, struct context *cx,
                                 struct generic_token *begin_expr,
                                 struct generic_token *end_expr){
  DECLARE_NESTING_CONTROL();
  struct generic_token *t = begin_expr, *prev = NULL, *last_op = NULL;
  if(begin_expr == end_expr)
    return get_primary_expression_type(mf, cx, begin_expr, end_expr);
  while(t != end_expr && t != NULL){
    if(IS_NOT_NESTED() && t -> type == TYPE_OPEN_BRACES)
      return TYPE_T_PATH; // 1st nesting '{' means direction specifier (Case 3)
    COUNT_NESTING(t);
    if(IS_NOT_NESTED()){
      if(t -> type == TYPE_OR || t -> type == TYPE_LT ||
           t -> type == TYPE_GT || t -> type == TYPE_GEQ ||
           t -> type == TYPE_LEQ || t -> type == TYPE_EQUAL ||
           t -> type == TYPE_NEQ)
        return TYPE_T_BOOLEAN;
      if(t -> type == TYPE_AMPERSAND || t -> type == TYPE_JOIN ||
         t -> type == TYPE_STRAIGHT_JOIN)
        return TYPE_T_PATH;
```

```
      if(t -> type == TYPE_PYTHAGOREAN_SUM ||
         t -> type == TYPE_PYTHAGOREAN_SUBTRACT)
        return TYPE_T_NUMERIC;
      if(IS_VALID_SUM_OR_SUB(prev, t) && t != end_expr)
        last_op = t;
    }
    prev = t;
    t = t -> next;
  }
  if(last_op != NULL)
    return get_secondary_expression_type(mf, cx, last_op -> next, end_expr);
  else return get_secondary_expression_type(mf, cx, begin_expr, end_expr);
}
```

9. Compound Statements: Conditional Statement

A conditional statement is a `if`, it ensures that some piece of code will be evaluated only if some conditions are true. The complete gramamr for these statements is:

```
<Conditional Block> -> if <Boolean Expression> :
                         <List of Statements>
                       <Alternatives>
                     fi
<Alternativas> -> <Empty> |
                  elseif <Boolean Expression>:
                    <List of Statements>
                  <Alternatives> |
                  else: <List of Statements>
```

We need to declare two new kinds of tokens:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_ELSEIF,  // Symbolic token 'elseif'
TYPE_ELSE,    // Symbolic token 'else'
TYPE_COLON,   // Symbolic token ':'
```

Which should be associated with the following reserved words:

Section: List of Keywords (continuation):

```
"elseif", "else", ":",
```

In Section 6.2, we introduced the first compound statement: the one that begins with `begingroup` and ends with `endgroup`. The conditional statement using `if` is also a compound statement, which can contain other inner statements.

Remember thar the code is first read in teh function `eval_list_of_statement`, which splits the tokens using semicolons as delimiters and give each part to the function `eval_statement`. This second function evaluates the given code, but it can also change the pointer that delimits the last evaluated token, which is used to determine the next tokens to be evaluated. It's in this function that our `if` will be evaluated.

Like in the `begingroup` case, seen in Section 6.2, when we find a `if` in the first position in the list of tokens to be evalueated, we increase the nesting level and correct the position of the end pointer, ensuring that the next tokens to be evaluated will be the right ones. Here we do the same, but the logic to choose the right pointer is more complex and involves evaluating boolean expressions.

Given a `if`, we first evaluate the boolean expression next to it. If true, the position for the pointer will be the colon after the expression and we will assume that all other `elseif` and `else` are false and should not be executed. Otherwise, the pointer will be set to the one stored in the `if` token, which was initialized by our lexer.

If we find there a token `elseif`, we proceed as we did in the `if` case, checking the boolean expression. If we find a token `else`, we place the pointer in the colon next to `else`. And in the last case, we found the end of the compound statement and no true boolean expression. We place the pointer in the `fi`, and doing this, we skip all the statements inside the `if`.

However, the code that we ignore and do not evaluate because it is inside a block with a false boolean expression in a conditional statement also need to be checked for some critical syntax errors. For example, we could have tokens `endgroup` or `endchar` inside a `if` that were not preceded by a `begingroup` or `beginchar`. These errors could indicate that the user wrote an incorrect `if`, perhaps placing the `fi` in the wrong place. Therefore, these errors should not be ignorated. While we walk over the non-evaluated tokens in a `if`, we still would run code to update the nesting level when we find tokens like `begingroup`, `if`, `beginchar` and their corresponding closing tokens.

The code that evaluates our `if` statement is:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_IF){
  struct generic_token *begin_bool, *end_bool;
  struct boolean_variable b;
  // Delimiting boolean expression
  begin_bool = begin -> next;
  end_bool = begin_bool;
  while(end_bool != NULL && end_bool != *end && end_bool -> next -> type != TYPE_COLON)
    end_bool = end_bool -> next;
  if(end_bool == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
     return false;
  }
  if(end_bool == *end){
    RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  // Evaluating boolean expression:
  if(!eval_boolean_expression(mf, cx, begin_bool, end_bool, &b))
    return false;
  if(b.value == 1){ // True: Code inside this 'if' must be evaluated
    *end = end_bool -> next;
    return true;
  }
  else{ // False: We skip code in this 'if'
    struct generic_token *t = ((struct linked_token *) begin) -> link;
    while(t != NULL){
      if(t -> type == TYPE_FI){
        // We found the 'fi' corresponding to this 'if'
        *end = t;
        return true;
      }
      else if(t -> type == TYPE_ELSE){
        // We found the 'else' corresponding to this 'if'
        *end = t -> next;
        if((*end) == NULL){
          RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
          return false;
        }
```

```
        else if((*end) -> type != TYPE_COLON){
          RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL((*end) -> line),
                                     TYPE_COLON, (*end));
          return false;
        }
        return true;
      }
      else if(t -> type == TYPE_ELSEIF){
        // We found the 'elseif' corresponding to our 'if'
        begin_bool = t -> next;
        end_bool = begin_bool;
        while(end_bool != NULL && end_bool != *end &&
              end_bool -> next -> type != TYPE_COLON)
          end_bool = end_bool -> next;
        if(end_bool == NULL){
          RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
          return false;
        }
        if(!eval_boolean_expression(mf, cx, begin_bool, end_bool, &b))
          return false;
        if(b.value == 1){ // True: Code in 'elseif' must be evaluated
          *end = end_bool -> next;
          return true;
        }
      }
      t = ((struct linked_token *) t) -> next;
    }
  }
  // This should never happen: the lexer should detect if we have 'if' without 'fi':
  RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
  return false;
}
```

Back to the code that evaluates the `if`, finding the correct code to be evaluated depending on the conditional, is the responsibility of the code shown above, before the error dealing. If we find a `elseif` or `else` in the beginning of a new statement, one of two things is happening: we either are dealing with a syntax error, where these tokens do not have a corresponding `if`, or we are inside some `if` where we already evaluated the right conditional code and found the other conditional expression with code that should not be evaluated. In the second case, we should ignore the following tokens until we find the `fi` corresponding to our `f`:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_ELSEIF || begin -> type == TYPE_ELSE){
  struct generic_token *t;
  t = ((struct linked_token *) begin) -> link;
  while(t != NULL){
    if(t -> type == TYPE_FI){
      // We found the 'fi' corresponding to our 'if'
      *end = t;
      return true;
    }
    t = ((struct linked_token *) t) -> link;
```

```
  }
  // Isso nunca deve ocorrer: o lexer deve detectar se h 'if' sem 'fi':
  RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
  return false;
}
```

Finally, we also can find a `fi` in the beginning of a new statement to be evaluated. In this case, we just ignore it and proceed to interpret what comes after it:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_FI){
  *end = begin;
  return true;
}
```

10. Compound Statements: Iterations

An iteration command is a composite command `for` that executes a sequence of other commands an arbitrary number of times. The grammar for this command is:

```
<Iteration> -> for <For Header>:
                 <List of Statements>
               endfor
<For Header> -> <Numeric Variable>  = <Numeric Expression><Progression> |
                <Numeric Variable> := <Numeric Expression><Progression>
<Progresso> -> step <Numeric Expression> until <Numeric Expression>
```

For example, the following iteration is valid:

```
numeric i;
for i = 5 step 10 until 50:
  draw (i, 0);
endfor
```

In the above iteration, the drawing command is executed over points $(5, 0)$, $(15, 0)$, $(25, 0)$, $(35, 0)$, $(45, 0)$. The value of variable `i` begins as 5, and each iteration increases by 10. In the last increment, the variable becomes 55, which is greater than 50, the value set as the limit for our loop. Therefore, this prevents the execution of the iteration and of the internal drawing command.

We will need the following new token types:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_STEP,  // Symbolic token 'step'
TYPE_UNTIL, // Symbolic token 'until'
```

And we add their names to the list of reserved words:

Section: List of Keywords (continuation):

```
"step", "until",
```

Executing a loop means performing an initialization (should be done if the loop still is not running) and testing the loop condition to determine if it should be interrupted or should keep running:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_FOR){
  struct numeric_variable *control;
  struct numeric_variable increment;
  struct begin_loop_token *for_token = (struct begin_loop_token *) begin;
  struct generic_token *current_token = begin, *begin_expr, *end_expr;
```

```
  if(!(for_token -> running)){
    begin_nesting_level(mf, cx, begin);
                    <Section to be inserted: Iteration: Prepare Loop>
  }
                <Section to be inserted: Iteration: Check Loop Condition>
}
```

To prepare our iteration, we must read its header, composed by a numeric variable and two numeric expressions delimited by keywords `step` and `until`. The numeric variable is found after the `for` token:

Section: Iteration: Prepare Loop:

```
{
  struct symbolic_token *var_token = (struct symbolic_token *)for_token -> next;
  if(var_token == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(var_token -> line));
    return false;
  }
  if(var_token -> type != TYPE_SYMBOLIC){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(var_token -> line),
                               TYPE_SYMBOLIC, (struct generic_token *) var_token);
    return false;
  }
  if(var_token -> var == NULL){
    RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(var_token -> line),
                                    var_token);
    return false;
  }
  control = var_token -> var;
  if(control -> type != TYPE_T_NUMERIC){
    RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(var_token -> line),
                                    var_token, control -> type,
                                    TYPE_T_NUMERIC);
    return false;
  }
  for_token -> control_var = &(control -> value);
}
```

After the numeric variable, we must get a "=" or ":=":

Section: Iteration: Prepare Loop (continuation):

```
{
  current_token = for_token -> next -> next;
  if(current_token == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(current_token -> line));
    return false;
  }
  if(current_token -> type != TYPE_ASSIGNMENT &&
     current_token -> type != TYPE_EQUAL){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(current_token -> line),
                               TYPE_EQUAL, current_token);
    return false;
  }
}
```

Next we delimit the following expression that begins after the assignment token and ends right before the next `step` token. The evaluated value should initialize the control variable. A numeric expression never have a `step` token, so we do not need to worry about nesting of expressions in this part:

Section: Iteration: Prepare Loop (continuation):

```
{
  begin_expr = current_token -> next;
  if(begin_expr == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(current_token -> line));
    return false;
  }
  end_expr = begin_expr;
  while(end_expr -> next != NULL && end_expr -> next -> type != TYPE_STEP)
    end_expr = end_expr -> next;
  if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, control))
    return false;
  current_token = end_expr;
}
```

The next step is evaluate the expression after `step` and before `until` to determine how the control variable should be incremented. Notice that the increment should not be done during initialization, but after each loop iteration. And after each iteration, this expression must be evaluated again, to ensure that the result did not change.

Section: Iteration: Check Loop Condition:

```
{
  while(current_token != NULL && current_token -> type != TYPE_STEP)
    current_token = current_token -> next;
  if(current_token == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(current_token -> line));
    return false;
  }
  begin_expr = current_token -> next;
  end_expr = begin_expr;
  while(end_expr != NULL && end_expr -> next != NULL &&
        end_expr -> next -> type != TYPE_UNTIL)
    end_expr = end_expr -> next;
  if(end_expr == NULL || end_expr -> next == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(current_token -> line));
    return false;
  }
  if(end_expr -> next -> type == TYPE_SEMICOLON ||
     end_expr -> next -> type == TYPE_COLON){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(current_token -> line),
                               TYPE_UNTIL, end_expr -> next);
    return false;
  }
  if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &increment))
    return false;
  if(for_token -> running)
    *(for_token -> control_var) += increment.value;
  else
    for_token -> running = true;
```

```
  current_token = end_expr;
}
```

Finally, we must check if we should continue running the loop or if we need to stop it. For this, we need to evaluate another numeric expression found between tokens "`until`" and "`:`":

Section: Iteration: Check Loop Condition (continuation):

```
{
  struct numeric_variable limit;
  current_token = current_token -> next;
  if(current_token -> next == NULL){
    RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(current_token -> line),
                                   TYPE_T_NUMERIC);
    return false;
  }
  begin_expr = current_token -> next;
  end_expr = begin_expr;
  while(end_expr -> next != NULL && end_expr -> next -> type != TYPE_COLON){
    if(end_expr -> next -> type == TYPE_SEMICOLON){
      RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(current_token -> line),
                                 TYPE_SEMICOLON, end_expr -> next);
      return false;
    }
    end_expr = end_expr -> next;
  }
  if(end_expr -> next == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(current_token -> line));
    return false;
  }
  if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &limit))
    return false;
  if((increment.value > 0 && *(for_token -> control_var) > limit.value) ||
     (increment.value < 0 && *(for_token -> control_var) < limit.value)){
    // Interrupt loop:
    for_token -> running = false;
    *end = (struct generic_token *) for_token -> end;
    if(!end_nesting_level(mf, cx, *end))
      return false;
    return true;
  }
  else{
    // Keep running the loop:
    *end = end_expr -> next;
    return true;
  }
}
```

When we find an `endfor` token, we set our interpreter to choose as the next statement, not what comes after the `endfor`, but the corresponding `for`. This is done changing a pointer from the end of statement to the token that comes before the `for`. The address of such region is stored in the `endfor` token since the token creation in our lexer.

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_ENDFOR){
  struct linked_token *endfor_token = (struct linked_token *) begin;
  *end = endfor_token -> link;
  return true;
}
```

11. The pickup Command

Npw we can begin defining our first command. The syntax for command `pickup` is:

```
<Command> -> <'pickup' Command>
<'pickup' Command> -> pickup <Pen to be Picked> <Optional Transformers>
<Pen to be Picked> -> nullpen | pencircle | pensemicircle | <Pen Variable>
<Optional Transformers> -> <Empty> |
                           <Transformer><Optional Transformers>
```

The command requires a new token and reserver keyword:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_PICKUP,  // Symbolic token 'pickup'
```

Section: List of Keywords (continuation):

```
"pickup",
```

What this command does is store a new value in the variable `currentpen`, which we will use to make drawings (which we will define in the following Section).

11.1. Extremity Points in Pens

When we pick a pen, we will store internally 4 values related to the pen. Two of them will store the biggest and smallest values in the axis x when the pen is rendered and the two others will store the smallest and biggest values in the axis y. Storing these values will be useful to better adjust the points that we will draw. For example, we could want to draw with our pen the nearest possible to the left lower corner of the image, without the pen drawing outside the image area. Such control requires to know the pen size. Because of this, we will store globally such information in `pen_lft`, `pen_rt`, `pen_top` e `pen_bot`. These values will be stored in our execution context:

Section: Attributes (struct context) (continuation):

```
float pen_lft, pen_rt, pen_top, pen_bot;
```

These values begin as zero, because `currentpen` begins as a `nullpen` in each execution context:

Section: Initialization (struct context) (continuation):

```
cx -> pen_lft = cx -> pen_rt = cx -> pen_top = cx -> pen_bot = 0.0;
```

An operation very common in this Section will be update these 4 variables in a pen when we are preparing their points. Assume that we will be able to iterate over the points in their perimeter. For each point, we will check if it is at the left of `pen_lft`, at the right of `pen_rt`, above `pen_top` or below `pen_bot`. In these cases, we need to update the information about the extremity points in the pen.

To reduce our code size, we will create the macro below that will declare the temporary variables where we store the biggest and smallest value in axis x and y that we found in a pen perimeter:

Section: Local Macros (metafont.c) (continuation):

```
#define DECLARE_PEN_EXTREMITIES() float _max_x = -INFINITY, _min_x = INFINITY,\
                                  _max_y = -INFINITY, _min_y = INFINITY;
```

Given a new perimeter point, we will use the macro below to get its coordinate (x, y), multiply by its transform matrix and check if we found a point with the biggest or smallest value in axis x or y:

Section: Local Macros (metafont.c) (continuation):

```
#define CHECK_PEN_EXTREMITIES(x, y, matrix) {\
   float _x, _y;\
   _x = LINEAR_TRANSFORM_X(x, y, matrix);\
   _y = LINEAR_TRANSFORM_Y(x, y, matrix);\
   if(_x < _min_x) _min_x = _x;\
   if(_x > _max_x) _max_x = _x;\
   if(_y < _min_y) _min_y = _y;\
   if(_y > _max_y) _max_y = _y;\
  }
// If the matrix is the identity, we can use this:
#define CHECK_PEN_EXTREMITIES_I(x, y) {\
   if(x < _min_x) _min_x = x;\
   if(x > _max_x) _max_x = x;\
   if(y < _min_y) _min_y = y;\
   if(y > _max_y) _max_y = y;\
  }
```

After checking all the points in the pen perimeter, we can update which are its extremity coordinates:

Section: Local Macros (metafont.c) (continuation):

```
#define UPDATE_PEN_EXTREMITIES() {\
  cx -> pen_lft = _min_x;\
  cx -> pen_rt = _max_x;\
  cx -> pen_top = _max_y;\
  cx -> pen_bot = _min_y;\
}
```

The previous macros is enough if we will iterate over each point in the pen perimeter. Doing this, we can call `CHECK_PEN_EXTREMITIES` in all points and then we get who has the biggest and smallest value in each axis. But in cases when we will not iterate over each point, we will need more functions to deduce their values. For example, if we are using a `pickup` command over `pencircle` or `pensemicircle` with a linear transform stored in a matrix. The function below can be used to get this pen extremity points:

Section: Local Function Declaration (metafont.c) (continuation):

```
void pencircular_extremity_points(struct context *cx,
                                  float *matrix, bool fullcircle);
```

A default `pencircle` or `pensemicircle`, with no linear transform is a pen with radius 1/2. Which means that every pen coordinate (x, y) follows the formula:

$$x^2 + y^2 = 0.25$$

Or, equivalently:

$$x = \pm\sqrt{0.25 - y^2}$$

$$y = \pm\sqrt{0.25 - x^2}$$

In the case of a semicircle, the same formulas are valid, except that we have the restriction that $y \geq 0$. Given a linear transform defined by matrix N, we get new values (x, y) using th formulas:

$$f_x(x) = M_{11}x \pm M_{21}\sqrt{0.25 - x^2} + M_{31}$$

$$f_y(y) = M_{22}y \pm M_{12}\sqrt{0.25 - y^2} + M_{32}(circle)$$

$$f_y(y) = M_{22}y + M_{12}\sqrt{0.25 - y^2} + M_{32}(semicircle)$$

When $M_{11} = 0$, the most extreme points in axis x will be $(-0.5, 0)$ and $(0.5, 0)$ after passing by the linear transform defined by our matrix. When $M_{21} = 0$, the most extreme points in axis x will be $(0, -0.5)$ and $(0, 0.5)$ after applying the linear transform. The same logic applies for the axis y if $M_{22} = 0$ or $M_{12} = 0$. In all these cases, we get the biggest and smallest values maximizing or minimizing a single coordinate.

For all other cases, finding the maximum and minimum values in these functions requires getting the derivative of the function and equalizing to zero:

$$f'_x(x) = M_{11} \pm (M_{21}x)/(\sqrt{0.25 - x^2}) = 0$$

$$f'_y(y) = M_{22} \pm (M_{12}y)/(\sqrt{0.25 - y^2}) = 0(circle)$$

$$f'_y(y) = M_{22} + (M_{12}y)/(\sqrt{0.25 - y^2}) = 0(semicircle)$$

As there is no closed formula to compute the soluction directly, we need to employ an interactive method to find the solution. We will first try the Newton Method. To use this method, we need to compute the second order derivative for the functions:

$$f''_x(x) = \pm(M_{21}\sqrt{0.25 - x^2} + (M_{21}x^2)/(\sqrt{0.25 - x^2}))$$

$$f''_y(y) = \pm(M_{12}\sqrt{0.25 - x^2} + (M_{12}x^2)/(\sqrt{0.25 - x^2}))$$

And using these values, by the Newton method, if we have a good guess x_n or y_n for the solution, we can find an even better guess x_{n+1} or y_{n+1} computing:

$$x_{n+1} = x_n - (f'_x(x_n)/f''_x(x_n))$$

$$y_{n+1} = y_n - (f'_y(y_n)/f''_y(y_n))$$

The problem of Newton Method in this function is that it is defined only in the interval between -0.5 and +0.5 (or between 0 and 0.5 for f_y in case of a semicircle). If the value that we are searching is near the extremity, it is possible that we step outside the function Domain. If this happens, we will change to another method. We will use the slower Bissection Method. For this method, notice that $f'(-0.5)$ and $f'(+0.5)$ has opposite signals. The root of this functions is in this interval. Next, we make the interval slower computing f' in the middle of the interval and checking the signal. And we repeat until we find the value or the interval becames too small.

After reviewing the theory, we finally can write the function that gets the extremity points for `pencircle`:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void pencircular_extremity_points(struct context *cx,
                                  float *matrix, bool fullcircle){
  DECLARE_PEN_EXTREMITIES();
  int i, index[4] = {0, 3, 4, 1};
  for(i = 0; i < 2; i ++){ // i=0 computes axis x, i=1 computes axis y
    // Primeiro os casos mais simples:
    if((i == 0 && matrix[3] == 0.0) ||
       (i == 1 && matrix[4] == 0.0)){
```

```
    CHECK_PEN_EXTREMITIES(-0.5, 0.0, matrix);
    CHECK_PEN_EXTREMITIES(0.5, 0.0, matrix);
  }
  else if((i == 0 && matrix[0] == 0.0) ||
          (i == 1 && matrix[1] == 0.0)){
    CHECK_PEN_EXTREMITIES(0.0, 0.5, matrix);
    if(fullcircle)
      CHECK_PEN_EXTREMITIES(0.0, -0.5, matrix);
  }
  else{
    // Newton Method
    float x0 = INFINITY, x1 = 0.0;
    do{
      x0 = x1;
      x1 = x0 - ((matrix[index[2*i]]+(matrix[index[2*i+1]]*
                x0/sqrt(0.25-x0*x0))) /
           (matrix[index[2*i+1]]*sqrt(0.25-x0*x0)+
                ((matrix[index[2*i+1]]*x0*x0)/
           (sqrt(0.25-x0*x0)))));
      if(x1 <= -0.5 || x1 >= 0.5){
        // Failed, using Bissection Method
        float y1;
        x0 = -0.5;
        x1 = 0.5;
        y1 = matrix[3-i*2] * sqrt(0.25-x1*x1) +
             (matrix[index[3-i*2]]*x1/sqrt(0.25-x1*x1));
        while(x0 != x1){
          float x2 = (x0+x1)/2;
          float y2 = matrix[index[3-i*2]] * sqrt(0.25-x2*x2) +
                     (matrix[index[3-i*2]]*x2/sqrt(0.25-x2*x2));
          if(y2 == 0.0 || x0 == x2 || x1 == x2)
            x0 = x1 = x2;
          else if(y2 > 0){
            if(y1 > 0)
              x1 = x2;
            else
              x0 = x2;
          }
          else{
            if(y1 > 0)
              x0 = x2;
            else
              x1 = x2;
          }
        }
      }
    } while(x0 != x1);
    if(i == 0){
      CHECK_PEN_EXTREMITIES(x0, sqrt(0.25-x0*x0), matrix);
      if(fullcircle)
        CHECK_PEN_EXTREMITIES(x0, -sqrt(0.25-x0*x0), matrix);
```

```
        CHECK_PEN_EXTREMITIES(-x0, sqrt(0.25-x0*x0), matrix);
        if(fullcircle)
          CHECK_PEN_EXTREMITIES(-x0, -sqrt(0.25-x0*x0), matrix);
      }
      else{
        CHECK_PEN_EXTREMITIES(sqrt(0.25-x0*x0), x0, matrix);
        CHECK_PEN_EXTREMITIES(-sqrt(0.25-x0*x0), x0, matrix);
        if(fullcircle){
          CHECK_PEN_EXTREMITIES(sqrt(0.25-x0*x0), -x0, matrix);
          CHECK_PEN_EXTREMITIES(-sqrt(0.25-x0*x0), -x0, matrix);
        }
      }
    }
  }
  UPDATE_PEN_EXTREMITIES();
}
```

We also could want to find the extremity points in a pen defined as a list of Bzier Curves:

Section: Local Function Declaration (metafont.c) (continuation):

```
void path_extremity_points(struct context *cx,
                           struct path_variable *p, float *matrix);
```

Curiously, in this case, the formulas are simpler. A cubic Bzier Curve like the ones that we use have the following formula, with t ranging between 0 and 1 and with z_1, z_4 being the extremity points and with z_2 and z_3 being the control points:

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2 + 3(1-t) t^2 z_3 + t^3 z_4$$

The function derivative is:

$$z'(t) = (-3z_1 + 9z_2 - 9z_3 + 3z_4)t^2 + (6z_1 - 12z_2 + 6z_3)t + (-3z_1 + 3z_2)$$

Discovering in which points we have a zero requires just using Bhaskara formula.

Finding the extremity points will be done iterating over the extremity points and computing the roots of $z'(t)$. If we find a root between 0 and 1, we check the value if it is an extremity point. The points z_1, z_2, z_3, z_4 that we use are the stored ones in the pen, after applying the linear transform of the corresponding matrix:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void path_extremity_points(struct context *cx,
                           struct path_variable *p, float *matrix){
  int i, j, length = p -> length;
  DECLARE_PEN_EXTREMITIES();
  for(i = 0; i < length; i ++){
    float x0, y0, u_x, u_y, v_x, v_y, x1, y1;
    CHECK_PEN_EXTREMITIES(p -> points[i].point.x, p -> points[i].point.y,
                          matrix);
    x0 = LINEAR_TRANSFORM_X(p -> points[i].point.x, p -> points[i].point.y,
                            matrix);
    y0 = LINEAR_TRANSFORM_Y(p -> points[i].point.x, p -> points[i].point.y,
                            matrix);
    u_x = LINEAR_TRANSFORM_X(p -> points[i].point.u_x,
                             p -> points[i].point.u_y, matrix);
    u_y = LINEAR_TRANSFORM_Y(p -> points[i].point.u_x,
                             p -> points[i].point.u_y, matrix);
```

```
    v_x = LINEAR_TRANSFORM_X(p -> points[i].point.v_x,
                             p -> points[i].point.v_y, matrix);
    v_y = LINEAR_TRANSFORM_Y(p -> points[i].point.v_x,
                            p -> points[i].point.v_y, matrix);
    x1 = LINEAR_TRANSFORM_X(p -> points[(i+1)%length].point.x,
                            p -> points[i].point.y, matrix);
    y1 = LINEAR_TRANSFORM_Y(p -> points[(i+1)%length].point.y,
                            p -> points[i].point.y, matrix);
    // Bhaskara Formula (axis x)
    float a, b, c, deltah, t;
    a = (-3*x0+9*u_x-9*v_x+3*x1);
    b = (6*x0-12*u_x+6*v_x);
    c = (-3*x0+3*u_x);
    deltah = b * b - 4 * a * c;
    for(j = -1; j < 2; j += 2){
      t = (-b + j * sqrt(deltah)) / (2 * a);
      if(t > 0.0 && t < 1.0){
        float x, y;
        x = (1-t)*(1-t)*(1-t)*x0+3*(1-t)*(1-t)*t*u_x+3*(1-t)*t*t*v_x+
            t*t*t*x1;
        y = (1-t)*(1-t)*(1-t)*y0+3*(1-t)*(1-t)*t*u_y+3*(1-t)*t*t*v_y+
            t*t*t*y1;
        CHECK_PEN_EXTREMITIES_I(x, y);
      }
    }
    // Bhaskara Formula (axis y)
    a = (-3*y0+9*u_y-9*v_y+3*y1);
    b = (6*y0-12*u_y+6*v_y);
    c = (-3*y0+3*u_y);
    deltah = b * b - 4 * a * c;
    for(j = -1; j < 2; j += 2){
      t = (-b + j * sqrt(b * b - 4 * a * c)) / (2 * a);
      if(t > 0.0 && t < 1.0){
        float x, y;
        x = (1-t)*(1-t)*(1-t)*x0+3*(1-t)*(1-t)*t*u_x+3*(1-t)*t*t*v_x+
            t*t*t*x1;
        y = (1-t)*(1-t)*(1-t)*y0+3*(1-t)*(1-t)*t*u_y+3*(1-t)*t*t*v_y+
            t*t*t*y1;
        CHECK_PEN_EXTREMITIES_I(x, y);
      }
    }
  }
  UPDATE_PEN_EXTREMITIES();
}
```

11.2. Triangulation

Now let's deal with the triangulation code. Video cards and OpenGL works drawing triangles in the screen. Therefore, each pen should be transformed in a set of triangles before drawing them. This was mentioned in Subsection 7.5. about pen variables. But the code for triangulation still needs to be defined.

To recap, these are the relevant variables stored inside a `struct pen_variable` that we will use during triangulation:

* `struct path_variable *format`: A cyclic path with the pen shape.

* `GLuint gl_vbo`: References que vertices stored in the video card. Will store 0 if the pen was not triangulated.

* `float triang_resolution`: An internal measure of the detail level in our triangulation. Relevant to know if we need to retriangulate to generate a more precise representation if our pen has curves and now will be rendered in a different size.

* `int flags`: Stores information if the above shape is null, is concave or convex, it is a polycon or a shape with curves, if it is a square or a circle. Besides these flags that were already presented,we will also define and use here two other flags:

Section: Local Macros (metafont.c) (continuation):

```
#define FLAG_ORIENTATION      64
#define FLAG_COUNTERCLOCKWISE 128
```

The second flag above will store if we are storing the shape of the pen in the counterclockwise order (if the flag is set) or in the clockwise order (if the flag is not set). But this information is initialized and is reliable only if the first flag above is set.

We will use the following function to check the pen orientation:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool is_pen_counterclockwise(struct pen_variable *pen);
```

The function works checking if we already have the information stored in the flag and returning it if so. Otherwise, it tries to discover the orientation. This is done searching for the vertex with the smallest y coordinate (and biggest x in case of tie) Let A be this vertex, P the previous vertex and N the next. The signal of the scalar product between AP and AN inform the orientation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool is_pen_counterclockwise(struct pen_variable *pen){
  int i, index = 0;
  int size = pen -> format -> length;
  float smallest_y = INFINITY, biggest_x = -INFINITY;
  if(pen -> flags & FLAG_ORIENTATION)
    return (pen -> flags & FLAG_COUNTERCLOCKWISE);
  if(pen -> format == NULL || size <= 0)
    return true;
  for(i = 0; i < size; i ++){
    if(pen -> format -> points[i].point.y < smallest_y ||
       (pen -> format -> points[i].point.y == smallest_y &&
        pen -> format -> points[i].point.y > biggest_x)){
      smallest_y = pen -> format -> points[i].point.y;
      biggest_x = pen -> format -> points[i].point.x;
      index = i;
    }
  }
  {
    int n = (index - 1) % size, p = (index + 1) % size;
    if(n < 0)
      n += size;
    if(p < 0)
      p += size;
    while(pen -> format -> points[index].point.x ==
                                    pen -> format -> points[n].point.x &&
          pen -> format -> points[index].point.y ==
```

```
                                    pen -> format -> points[n].point.y)
      n = (n + 1) % size;
    while(pen -> format -> points[index].point.x ==
                                    pen -> format -> points[p].point.x &&
          pen -> format -> points[index].point.y ==
                                       pen -> format -> points[p].point.y)
      p = (p - 1) % size;
    float ap_x = pen -> format -> points[p].point.x -
                 pen -> format -> points[index].point.x;
    float ap_y = pen -> format -> points[p].point.y -
                 pen -> format -> points[index].point.y;
    float an_x = pen -> format -> points[n].point.x -
                 pen -> format -> points[index].point.x;
    float an_y = pen -> format -> points[n].point.y -
                 pen -> format -> points[index].point.y;
    float prod = ap_x * an_x + ap_y * an_y;
    pen -> flags += FLAG_ORIENTATION;
    pen -> flags += FLAG_COUNTERCLOCKWISE * (prod > 0);
    return (prod > 0);
  }
}
```

The function that will perform the triangulation is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool triangulate_pen(struct metafont *mf, struct context *cx,
                     struct pen_variable *pen, float *transform_matrix);
```

And its implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool triangulate_pen(struct metafont *mf, struct context *cx,
                     struct pen_variable *pen, float *transform_matrix){
                   <Section to be inserted: Triangulation: Null Pen>
                 <Section to be inserted: Triangulation: Square Pen>
               <Section to be inserted: Triangulation: Convex Polygon>
             <Section to be inserted: Triangulation: Circle and Semicircle>
                <Section to be inserted: Triangulation: Convex Shape>
                <Section to be inserted: Triangulation: Concave Shape>
}
```

The simplest case is when we have a null pen created by `nullpen`. These pens never will draw anything, so they do not need to be triangulated. They are considered points at the $(0, 0)$ coordinate, but a transform matrix can shift the point to some other position. We deal with this simplest case below:

Section: Triangulation: Null Pen:

```
if((pen -> flags & FLAG_NULL)){
  pen -> indices = 0;
  DECLARE_PEN_EXTREMITIES();
  CHECK_PEN_EXTREMITIES(0, 0, transform_matrix);
  UPDATE_PEN_EXTREMITIES();
  return true;
}
```

The next case is when we need to triangulate a square pen. This kind of pen do not need to be

triangulated because during initialization we will create a single triangulation that will be used for all square pen. The vertices will be stored here:

Section: Local Variables (metafont.c) (continuation):

```
static GLuint pensquare_vbo;
```

The triangulation code thet will be run in the initialization:

Section: WeaveFont Initialization (continuation):

```
{
  float square_vertices[8] = {-0.5, -0.5,
                              +0.5, -0.5,
                              +0.5, +0.5,
                              -0.5, +0.5};
  glGenBuffers(1, &pensquare_vbo);
  glBindBuffer(GL_ARRAY_BUFFER, pensquare_vbo);
  glBufferData(GL_ARRAY_BUFFER, 8 * sizeof(float), square_vertices,
               GL_STATIC_DRAW);
}
```

And in the finalization we can remove the vertices:

Section: WeaveFont Finalization (continuation):

```
glDeleteBuffers(1, &pensquare_vbo);
```

So, when asked to triangulate a square pen, we do not need to triangulate it. Nevertheless, we still need to compute the extremity points:

Section: Triangulation: Square Pen:

```
if((pen -> flags & FLAG_SQUARE)){
  float square_vertices[8] = {-0.5, -0.5, 0.5, -0.5, 0.5, 0.5, -0.5, 0.5};
  pen -> indices = 4;
  DECLARE_PEN_EXTREMITIES();
  int i;
  for(i = 0; i < 4; i ++)
    CHECK_PEN_EXTREMITIES(square_vertices[2 * i], square_vertices[2 * i + 1],
                          transform_matrix);
  UPDATE_PEN_EXTREMITIES();
  return true;
}
```

The next simplest case is when we have a convex polygon. For them, we can use a very simple algorithm. We choose any vertex as the pivot and perform a triangulation creating triangles that always use this vertex and each other adjacent pair of vertices in our shape. It's the fan triangulation:

To use this, we can just pass all the pen vertices in the same order that they are stored and later, when drawing them, we will tell OpenGL to use fan triangulation:

Section: Triangulation: Convex Polygon:

```
if((pen -> flags & FLAG_STRAIGHT) && (pen -> flags & FLAG_CONVEX)){
  int i, index, increment;
  DECLARE_PEN_EXTREMITIES();
  GLsizei size = sizeof(float) * 2 * pen -> format -> length;
  float *data = (float *) temporary_alloc(size);
  if(data == NULL){
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
    return false;
  }
  if(is_pen_counterclockwise(pen)){
    index = 0;
    increment = 1;
  }
  else{
    index = pen -> format -> length - 1;
    increment = -1;
  }
  for(i = 0; i < pen -> format -> length; i ++){
    data[2 * i] = pen -> format -> points[index].point.x;
    data[2 * i + 1] = pen -> format -> points[index].point.y;
    CHECK_PEN_EXTREMITIES(data[2 * i], data[2 * i + 1], transform_matrix);
    index += increment;
  }
  if(pen -> gl_vbo == 0){
    glGenBuffers(1, &(pen -> gl_vbo));
    glBindBuffer(GL_ARRAY_BUFFER, pen -> gl_vbo);
    glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
    pen -> indices = pen -> format -> length;
  }
  if(temporary_free != NULL)
    temporary_free(data);
  UPDATE_PEN_EXTREMITIES();
  return true;
}
```

Now let's deal with the next case for triangulation: circles and semicircles. A circular pen is created using expression `pencircle`. For this kind of pen, we must generate its vertices during triangulation: we will not use the stored vertices. But how many vertices should be generated? If the circle have a single pixel as diameter, perhaps 4 vertices in the perimeter is enough, nobody would notice that this is a square instead of a circle. But if the circle becames bigger, we would need more vertices to correctly approximate its shape. The same applies for semicircles: for them we need half the vertices needed for a circle, but the required number of vertices increases with the semicircle size.

The number of required vertices is given by the formula of circle circumference, in pixels: we need $2\pi r$ vertices in the perimeter for circles and πr vertices for semicircles.

We will use the same fan triangulation as before. But we will use as axis the center of the circle. Therefore, as we need one more vertice to be the axis, we will store $2\pi r + 1$ vertices for circles and $\pi r + 1$ for semicircles.

The radius will be approximated as half the biggest side of the square where our circular pen is inscribed. We compute this after performing the linear transformation in the square using the circle transform matrix.

Therefore, we will get a suitable upper bound for the radius, considering that our circular pen could have been transformed in a ellipse or semi-ellipse after the linear transformation.

This biggest side of the rectangle where our circle is inscribed after the linear transformation is also how we compute the resolution of a circular pen triangulation. If our circular pen is already triangulated and we need to draw using a smaller circle, then we do not need to triangulate it again. But if we need to draw a bigger circle or semicircle, then we need to triangulate, even if we already had a triangulation before.

The code to triangulate the circular pen is:

Section: Triangulation: Circle and Semicircle:

```
if((pen -> flags & FLAG_CIRCULAR) || (pen -> flags & FLAG_SEMICIRCULAR)){
  float radius;
  GLsizei size;
  // Get extremity points:
  pencircular_extremity_points(cx, transform_matrix,
                               (pen -> flags & FLAG_CIRCULAR));
  // Checking resolution (radius):
  {
    float side1, side2;
    side1 = fabs(cx -> pen_rt - cx -> pen_lft);
    side2 = fabs(cx -> pen_top - cx -> pen_bot);
    radius = ((side1 >= side2)?(side1):(side2))/ 2.0;
  }
  // Need to retriangulate only if radius is bigger than already triangulated:
  if(pen -> gl_vbo != 0){
    if(radius > pen -> triang_resolution)
      glDeleteBuffers(1, &(pen -> gl_vbo));
    else
      return true;
  }
  pen -> triang_resolution = radius;
  if(pen -> flags & FLAG_CIRCULAR)
    size = sizeof(float) * 2 * (((int) (2 * M_PI * radius)) + 4);
  else
    size = sizeof(float) * 2 * (((int) (M_PI * radius)) + 4);
  float *data = (float *) temporary_alloc(size);
  if(data == NULL){
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
    return false;
  }
  {
    size_t i;
    float angle = 0.0;
    data[0] = 0.0;
    data[1] = 0.0; // Circle center
    for(i = 2; i < (size / sizeof(float)); i ++){
      data[i] = 0.5 * cos(angle);
      i ++;
      data[i] = 0.5 * sin(angle);
      angle += 1/radius;
      if((pen -> flags & FLAG_SEMICIRCULAR) && angle > M_PI)
        angle = M_PI;
    }
```

```
  }
  glGenBuffers(1, &(pen -> gl_vbo));
  glBindBuffer(GL_ARRAY_BUFFER, pen -> gl_vbo);
  glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
  pen -> indices = (size / (2 * sizeof(float)));
  if(temporary_free != NULL)
    temporary_free(data);
  return true;
}
```

Now we will deal with the last possible shape for a convex pen: when the pen is not a polygon, it has curves. In this case, like in the circle, we need to consider the size of our shape, including the matrix transformation, to decide how many vertices we should use to approximate correctly our curve. But this time, we need to do this iterating over the extremity points and control points that form the shape of our pen.

For each pair of extremity points (A, D), with the control points (B, C), we will use the sum of the distances AB, BC and CD to decide the number of vertices. This estimation should produce reasonable values, except when our curve crosses itself. However, we do not support pens with non-simple forms, therefore we should not care about this unsupported case.

The sum of all the distances, when we iterate over the pen curves is how we measure the pen triangulation resolution. If we are triangulating an already triangulated pen and the resolution is lesser or equal than the current triangulation, we do not need to triangulate the pen again.

If the sum of the distances $AB + BC + CD$ is equal the distance AD, then we have a straight line. In this case, we just need to consider the vertices for the extremity points, not the intermediate points.

After deciding the number of vertices that should be generated, if we need intermediate points between the extremities, then we can use the following formula, where we change the t value between 0 and 1 (with values 0 and 1 corresponding to the extremity points):

$$P(t) = (1-t)^3 A + 3(1-t)^2 tB + 3(1-t)t^2 C + t^3 D$$

Naturally, we should walk over the points in the counterclockwise orientation, which non necessarily is the stored orientation. When measuring the distances, we should also consider the linear transform stored in the transform matrix, but when generating the vertices we should ignore the transform, as it will be performed by the video card when rendering the pen.

The code to triangulate a pen in this case is:

Section: Triangulation: Convex Shape:

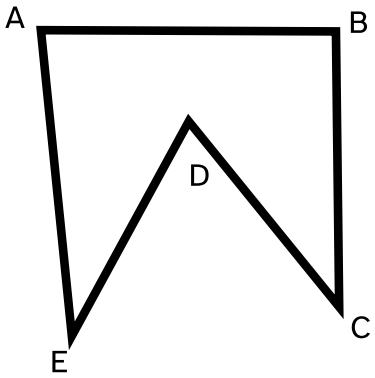
```
if((pen -> flags & FLAG_CONVEX)){
  bool counterclockwise = is_pen_counterclockwise(pen);
  int i, number_of_vertices = 1;
  // Get extremity points:
  path_extremity_points(cx, pen -> format, transform_matrix);
  for(i = 0; i < pen -> format -> length - 1; i ++){
    int distance = 0;
    float x0, y0, u_x, u_y, v_x, v_y, x1, y1;
    float dx, dy;
    x0 = LINEAR_TRANSFORM_X(pen -> format -> points[i].point.x,
                            pen -> format -> points[i].point.y, transform_matrix);
    y0 = LINEAR_TRANSFORM_Y(pen -> format -> points[i].point.x,
                            pen -> format -> points[i].point.y, transform_matrix);
    u_x = LINEAR_TRANSFORM_X(pen -> format -> points[i].point.u_x,
                            pen -> format -> points[i].point.u_y, transform_matrix);
    u_y = LINEAR_TRANSFORM_Y(pen -> format -> points[i].point.u_x,
                            pen -> format -> points[i].point.u_y, transform_matrix);
```

```
  dx = u_x - x0;
  dy = u_y - y0;
  distance += (int) round(sqrt(dx * dx + dy * dy));
  v_x = LINEAR_TRANSFORM_X(pen -> format -> points[i].point.v_x,
                           pen -> format -> points[i].point.v_y, transform_matrix);
  v_y = LINEAR_TRANSFORM_Y(pen -> format -> points[i].point.v_x,
                           pen -> format -> points[i].point.v_y, transform_matrix);
  dx = v_x - u_x;
  dy = v_y - u_y;
  distance += (int) round(sqrt(dx * dx + dy * dy));
  x1 = LINEAR_TRANSFORM_X(pen -> format -> points[i + 1].point.x,
                           pen -> format -> points[i + 1].point.y,
                           transform_matrix);
  y1 = LINEAR_TRANSFORM_Y(pen -> format -> points[i + 1].point.x,
                           pen -> format -> points[i + 1].point.y,
                           transform_matrix);
  dx = x1 - v_x;
  dy = y1 - v_y;
  distance += (int) round(sqrt(dx * dx + dy * dy));
  dx = x1 - x0;
  dy = y1 - y0;
  if(distance == (int) round(sqrt(dx * dx + dy * dy)))
    number_of_vertices ++; // Linha reta
  else
    number_of_vertices += distance;
}
if(pen -> gl_vbo != 0){
  if(number_of_vertices <= pen -> triang_resolution)
    return true; // No need to triangulate again
  else
    glDeleteBuffers(1, &(pen -> gl_vbo)); // Need to retriangulate
}
pen -> triang_resolution = number_of_vertices;
float *data = (float *) temporary_alloc(number_of_vertices * 2 *
                                        sizeof(float));
  if(data == NULL){
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
    return false;
}
{
  struct path_points *p0, *p1;
  int v;
  if(counterclockwise)
    p0 = &(pen -> format -> points[0]);
  else
    p0 = &(pen -> format -> points[pen -> format -> length - 1]);
  data[0] = p0 -> point.x;
  data[1] = p0 -> point.y;
  v = 2;
  for(i = 0; i < pen -> format -> length - 1; i ++){
    float b_x, b_y, c_x, c_y, dx, dy, x0, y0, x1, x2, y1, y2;
```

```
int distance = 0;
if(counterclockwise){
  p1 = &(pen -> format -> points[1 + i]);
  b_x = p0 -> point.u_x;
  b_y = p0 -> point.u_y;
  c_x = p0 -> point.v_x;
  c_y = p0 -> point.v_y;
}
else{
  p1 = &(pen -> format -> points[pen -> format -> length - 2 - i]);
  b_x = p1 -> point.v_x;
  b_y = p1 -> point.v_y;
  c_x = p1 -> point.u_x;
  c_y = p1 -> point.u_y;
}
x0 = LINEAR_TRANSFORM_X(p0 -> point.x, p0 -> point.y, transform_matrix);
y0 = LINEAR_TRANSFORM_Y(p0 -> point.x, p0 -> point.y, transform_matrix);
x2 = LINEAR_TRANSFORM_X(b_x, b_y, transform_matrix);
y2 = LINEAR_TRANSFORM_Y(b_x, b_y, transform_matrix);
dx = x2 - x0;
dy = y2 - y0;
distance += (int) round(sqrt(dx * dx + dy * dy));
x1 = x2;
y1 = y2;
x2 = LINEAR_TRANSFORM_X(c_x, c_y, transform_matrix);
y2 = LINEAR_TRANSFORM_Y(c_x, c_y, transform_matrix);
dx = x2 - x1;
dy = y2 - y1;
distance += (int) round(sqrt(dx * dx + dy * dy));
x1 = x2;
y1 = y2;
x2 = LINEAR_TRANSFORM_X(p1 -> point.x, p1 -> point.y, transform_matrix);
y2 = LINEAR_TRANSFORM_Y(p1 -> point.x, p1 -> point.y, transform_matrix);
dx = x2 - x1;
dy = y2 - y1;
distance += (int) round(sqrt(dx * dx + dy * dy));
dx = x2 - x0;
dy = y2 - y0;
if(distance == (int) round(sqrt(dx * dx + dy * dy))){
  data[v++] = p1 -> point.x;
  data[v++] = p1 -> point.y;
}
else{
  int j;
  float dt = 1.0 / ((float) distance);
  for(j = 1; j <= distance; j ++){
    float t = dt * j;
    data[v++] = (1-t)*(1-t)*(1-t)* p0 -> point.x + 3*(1-t)*(1-t)*t * b_x +
                3*(1-t)*t*t * c_x + t * t * t * p1 -> point.x;
    data[v++] = (1-t)*(1-t)*(1-t)* p0 -> point.y + 3*(1-t)*(1-t)*t * b_y +
                3*(1-t)*t*t * c_y + t * t * t * p1 -> point.y;
```

```
        }
      }
      p0 = p1;
    }
  }
  glGenBuffers(1, &(pen -> gl_vbo));
  glBindBuffer(GL_ARRAY_BUFFER, pen -> gl_vbo);
  glBufferData(GL_ARRAY_BUFFER, number_of_vertices * 2 *
                                sizeof(float), data, GL_STATIC_DRAW);
  pen -> indices = number_of_vertices;
  if(temporary_free != NULL)
    temporary_free(data);
  return true;
}
```

Finally, the most complex triangulation: if we have a concave form. We cannot triangulate using the simple algorithms seen above, just choosing a pivot and creating a triangles that share the pivot as a common vertex. Consider the concave polygon below, for example:

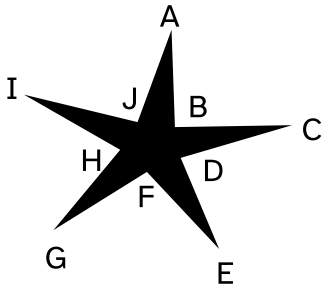


If we choose vertex E as pivot, we would produce the triangle ECD. But this would be a wrong triangle: even if their vertices are part of the polygon, it represents a region that is outside the polygon and that should not be triangulated.

To help triangulating polygons with more general shapes, we will implement the algorithms discussed in [DE BERG, 2000].

Triangulating concave polygons is easier in a particular case: when we have a x-monotone polygon. This happens when we walk from the leftmost vertex in the polygon to the rightmost vertex both in clockwise and counter-clockwise direction, and in both cases, when going from a vertex to the next one, the x coordinate never becomes smaller: it either increases or keeps the same value. For example, the polygon above is x-monotone. If we rotate it 90 degrees, it ceases to be x-monotone, but becomes y-monotone following an equivalente definition for the y-axis.

The polygon below, otherwise, is not x-monotone neither y-monotone:



To triangulate a x-monotone polygon, we can assume that we already checked that it is x-monotone by traversing its vertices from the leftmost to the rightmost, both taking the upper path and the lower path. In the process we mark each vertex as belonging to the upper and/or lower path. The initial vertex and final one belong to both paths. In the process we also order each vertex according to its x coordinate. if two vertices have the same x coordinate, we order them according with the coordinate y. For example, in the case of the first polygon concave that we represent in the images above, the order of its vertices would be (A, E, D, B, C). Where (A, B, C) is the upper path and (A, E, D, C) is the lower path.

Once the vertices are ordered and marked, we can go through them in the order in which we left them. Every time we read a new vertex, we try to create a diagonal using non-neighbor previous vertices, forming a new triangle. When this is not possible, we store the vertex in a stack. When we find an upper vertex,

while all the vertices in our stack are from the bottom path, we can create diagonal between all of them, and we can empty the stack. If we read a vertex of the bottom path, when there are only vertices from the upper path in the stack, the same thing happens. Otherwise, when we read a vertex from the same path than other vertices in the stack, we can try to create a new diagonal generating a triangle, but we can do this only if this diagonal is entirely inside the polygon (which is easier to check if we know if each vertex is in the top or bottom part of the polygon). If it is not possible to create the diagonal in this case, we let the vertices accumulate on the stack and we follow the algorithm, knowing that soon we will find a vertex of the opposite path that will allow triangulation.

For example, in the case of our x-monotone polygon whose ordered vertices are (A, E, D, B, C), we can start by reading (A, E, D). As unfortunately, D and E are both from the bottom path of the polygon, triangulation is not automatic. We then check whether we can create a diagonal DA, forming the triangle DAE. Fortunately we can, because the diagonal DA passes entirely inside the polygon (after all, DE are from the bottom and the vertex E is below the two points diagonal DA). After generating the DAE triangle, we can get rid of E and we now keep (A, D) on our stack. We read the next vertex B. Since B is from the top and D is from the bottom, a diagonal DB can be created. This produces the triangle ADB. We can then remove the vertex A and we are left with only (D, B, C), which is the final triangle of triangulation.

When our pen is not only made up of straight segments and has curves, we can first convert it to a polygon with a sufficiently large number of edges to apply the method. During the conversion we can identify which vertex is furthest left to be able to later test whether it is a x-monotone polygon. But first we have to define a structure to store a linked list of vertices:

Section: Local Data Structures (metafont.c) (continuation):

```
#define FLAG_UPPER 1
#define FLAG_LOWER 2
#define NEW_POLYGON_VERTEX() \
  (struct polygon_vertex *) temporary_alloc(sizeof(struct polygon_vertex))
#define DESTROY_POLYGON_VERTEX(v) \
  ((temporary_free != NULL)?(temporary_free(v)):(true))
struct polygon_vertex{
  int flag; // Will store if it is a vertex from upper or lower path
  float x, y;
  struct polygon_vertex *prev, *next;
        <Section to be inserted: struct polygon“˙vertex: Additional Variables>
};
```

To order vertices in a polygon according with their x-coordinate, but using the y-coordinate to break the tie if both are in the same x-coordinate, is done with the following macro:

Section: Local Data Structures (metafont.c) (continuation):

```
#define XMONOTONE_LEQ(v1, v2) ((v1->x==v2->x)?(v1->y<=v2->y):(v1->x<=v2->x))
```

When we have a double linked list of vertices, if we want to destroy it and free the memory, we can use the auxiliary function:

Section: Local Function Declaration (metafont.c) (continuation):

```
void destroy_vertex_linked_list(struct polygon_vertex *poly);
```

Which we implement below:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void destroy_vertex_linked_list(struct polygon_vertex *poly){
  if(temporary_alloc != NULL && poly != NULL){
    poly -> prev -> next = NULL;
    while(poly -> next != NULL){
      poly = poly -> next;
      temporary_free(poly -> prev);
```

```
    }
    temporary_free(poly);
  }
}
```

The doubled linked list of vertices will be created from a pen using the following function:

Section: Local Function Declaration (metafont.c) (continuation):

```
struct polygon_vertex *polygon_from_pen(struct metafont *mf,
                                        struct pen_variable *,
                                        float *transform_matrix,
                                        int *number_of_vertices);
```

And the function implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct polygon_vertex *polygon_from_pen(struct metafont *mf,
                                        struct pen_variable *p,
                                        float *transform_matrix,
                                        int *number_of_vertices){
  int i, j;
  *number_of_vertices = 0;
  struct polygon_vertex *first, *last, *leftmost;
  // Getting the path from our pen:
  struct path_variable *path = p -> format;
  if(path == NULL){
    RAISE_GENERIC_ERROR(mf, NULL, 0, ERROR_UNKNOWN);
    return NULL; // This should not happen
  }
  // First vertex:
  first = NEW_POLYGON_VERTEX();
  if(first == NULL){
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
    return NULL;
  }
  first -> next = NULL;
  first -> prev = NULL;
  first -> x = LINEAR_TRANSFORM_X(path -> points[0].point.x,
                                  path -> points[0].point.y, transform_matrix);
  first -> y = LINEAR_TRANSFORM_Y(path -> points[0].point.x,
                                  path -> points[0].point.y, transform_matrix);
  leftmost = first;
  last = first;
  *number_of_vertices = 1;
  // Loop over the path extremity points
  for(i = 0; i < path -> length - 1; i ++){
    double x0, y0, u_x, u_y, v_x, v_y, x1, y1;
    bool straight_line = (p -> flags & FLAG_STRAIGHT);
    x0 = last -> x;
    y0 = last -> y;
    u_x = LINEAR_TRANSFORM_X(path -> points[i].point.u_x,
                             path -> points[i].point.u_y, transform_matrix);
    u_y = LINEAR_TRANSFORM_Y(path -> points[i].point.u_x,
```

```
                          path -> points[i].point.u_y, transform_matrix);
  v_x = LINEAR_TRANSFORM_X(path -> points[i].point.v_x,
                          path -> points[i].point.v_y, transform_matrix);
  v_y = LINEAR_TRANSFORM_Y(path -> points[i].point.v_x,
                          path -> points[i].point.v_y, transform_matrix);
  x1 = LINEAR_TRANSFORM_X(path -> points[i + 1].point.x,
                          path -> points[i + 1].point.y, transform_matrix);
  y1 = LINEAR_TRANSFORM_Y(path -> points[i + 1].point.x,
                          path -> points[i + 1].point.y, transform_matrix);
  if(!straight_line){
    double slope1, slope2, slope3;
    slope1 = (u_x - x0) / (u_y - y0);
    slope2 = (v_x - u_x) / (v_y - u_y);
    slope3 = (x1 - v_x) / (y1 - v_y);
    if((y0 == u_y && u_y == v_y && v_y == y1) ||
       (slope1 == slope2 && slope2 == slope3))
      straight_line = true;
  }
  if(straight_line){ // In a straight line we need just 2 vertices
    if(i < path -> length - 2){
      last -> next = NEW_POLYGON_VERTEX();
      if(last -> next == NULL){
        RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
        return NULL;
      }
      last -> next -> prev = last;
      last -> next -> next = NULL;
      last = last -> next;
      last -> x = x1;
      last -> y = y1;
      (*number_of_vertices) ++;
      if(last -> x < leftmost -> x ||
        (last -> x == leftmost -> x && last -> y < leftmost -> y))
        leftmost = last;
    }
  }
  else{ // In a curve we need several vertices
    double dt;
    int distance = (int) ceil(sqrt(pow(u_x-x0, 2.0)+pow(u_y-y0, 2.0)));
    distance += (int) ceil(sqrt(pow(v_x-u_x, 2.0)+pow(v_y-u_y, 2.0)));
    distance += (int) ceil(sqrt(pow(x1-v_x, 2.0)+pow(y1-v_y, 2.0)));
    dt = 1.0/((double) distance);
    for(j = 1; j <= distance; j ++){
      double t = dt * j;
      float x = (1-t)*(1-t)*(1-t) * x0 + 3*(1-t)*(1-t)*t * u_x +
                3*(1-t)*t*t * v_x + t*t*t * x1;
      float y = (1-t)*(1-t)*(1-t) * y0 + 3*(1-t)*(1-t)*t * u_y +
                3*(1-t)*t*t * v_y + t*t*t * y1;
      if(*number_of_vertices > 0){
        if(last -> y == x && last -> y == y){
          last -> x = x;
```

```
            last -> y = y;
            continue;
          }
        }
        if(*number_of_vertices > 1){ // Do we need previous vertex?
          double slope1, slope2;
          slope1 = (x - last->x)/(y - last->y);
          slope2 = (last->x - last->prev->x) /
                   (last->y - last->prev->y);
          if((y == last->y && y == last->prev->y) || slope1 == slope2){
          // If not needed:
            last -> x = x;
            last -> y = y;
            continue;
          }
        }
        // Create new vertex:
        last -> next = NEW_POLYGON_VERTEX();
        if(last -> next == NULL){
          RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
          return NULL;
        }
        last -> next -> prev = last;
        last -> next -> next = NULL;
        last = last -> next;
        last -> x = x;
        last -> y = y;
        (*number_of_vertices) ++;
        if(last -> x < leftmost -> x ||
          (last -> x == leftmost -> x && last -> y < leftmost -> y))
          leftmost = last;
      }
    }
  }
  first -> prev = last;
  last -> next = first;
  return leftmost;
}
```

The previous function works by first extracting the format path from the pen and generating the first vertex from the first point. Then, if we have a straight line to the next extremity point, we just add one more vertex. Otherwise, we estimate the number of vertices needed to approximate the curve by the straight path distance of the first extremity point to the next, passing through the control points. We then generate this number of vertices, but each time we detect that a new vertex, together with the previous two, are in the same line, we overwrite the previous vertex with the current one, since the previous one is redundant. The number of operations that the above function executes to insert vertices is asymptotically equivalent to pen perimeter (in pixels) for curved pens and the number of vertices for polygonal pens.

The function that checks if a polygon is x-monotone is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool is_xmonotone(struct polygon_vertex *poly);
```

It assumes that the polygon given as argument is represented as a pointer to the leftmost vertex in

the double linked list. The function first checks if the polygon has its vertices in the clockwise or counter-clockwise order. Basing on this information, it walks over the vertices adjusting its flag as `FLAG_UPPER` or `FLAG_LOWER`. If we find a vertex not compatible with a x-monotone polygon, we return false and the vertex flags should be ignored. This is the function implementation:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
bool is_xmonotone(struct polygon_vertex *poly){
  bool clockwise;
  struct polygon_vertex *upper, *lower;
  upper = lower = poly;
  upper -> flag = (FLAG_UPPER | FLAG_LOWER);
  while(upper -> next -> y == lower -> prev -> y){
    if(upper -> next -> x >= upper -> x)
      upper = upper -> next;
    else if(lower -> prev -> x >= lower -> x)
      lower = lower -> prev;
    else break;
  }
  clockwise = (upper -> next -> y >= lower -> prev -> y);
  upper = lower = poly;
  do{
    if(XMONOTONE_LEQ(upper, upper -> next)){
      upper = upper -> next;
      if(clockwise){
        upper -> flag = FLAG_UPPER;
      }
      else{
        upper -> flag = FLAG_LOWER;
      }
    }
    else if(XMONOTONE_LEQ(lower, lower -> prev)){
      lower = lower -> prev;
      if(clockwise){
        lower -> flag = FLAG_LOWER;
      }
      else{
        lower -> flag = FLAG_UPPER;
      }
    }
    else
      return false;
  } while(upper != lower);
  upper -> flag = (FLAG_UPPER | FLAG_LOWER);
  return true;
}
```

When we triangulate a concave polygon, we will not use the optimizations where we assume that all triangles share a same point (which allows us to spend less space to store the triangles) just as we did to triangulate circles, squares, semicircles and convex polygons. Instead, each triangle is entirely represented by three coordinates of your vertex. We assume that we will have allocated an array with $3n$ elements and during triangulation we will fill it. And for that, we will need always an index t ($0 \leq t < 3n$) to indicate where inside the array we can place the next triangle.

The function that triangulates x-monotone polygons has the objective of filling this array. It assumes that it was already allocated with sufficient space, so it doesn't do any checking for array bounds. It implements the algorithm over some polygon P which we described more informally above, but which is described more formally below:

1. Sort all polygon vertices according with x coordinate. Let u_1, ..., u_n be the sorted sequence.
2. Initialize a stack S and push u_1 and u_2.
3. Do $j \leftarrow 3$ **until** $n - 1$:
4. **If** u_j and the next vertex in S are on different paths (above and below):
5. Pop all vertices from S.
6. Create a diagonal from u_j to the popped vertex, except the last one.
7. Push u_{j-1} and u_j.
8. **Else**:
9. Pop a vertex from S.
10. Pop from S while the diagonal from u_j to the vertex is inside P.
11. Create a diagonal from u_j to each popped vertice.
12. Push the last popped vertex to S.
13. Push u_j in S.
14. Add diagonals from u_n to each vertice in the stack.

A diagonal is an additional edge placed on the polygon. After adding all the diagonals, the polygon will be triangulated. In practice, each diagonal allows us to add a new triangle and allows us to remove a vertex from our polygon.

The first step in the above algorithm is ordering the vertices from the least x-coordinate to the greatest one. As we have a x-monotone polygon, we can do this in $O(n)$ time, just walking from the first polygon to the upper and lower path until the last one. Instead of creating a new structure for the linked list of ordered vertices, we will add a new pointer to the existing structure, that will point for the next vertex on the ordered list:

Section: struct polygon_vertex: Additional Variables:

```
  struct polygon_vertex *succ;
```

We initialize this pointer running the following function that orders the vertices:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static void order_vertices_on_xmonotone_polygon(struct polygon_vertex *p){
  struct polygon_vertex *upper, *lower, *last;
  last = upper = lower = p;
  while(lower -> prev != upper && upper -> next != lower){
    if(XMONOTONE_LEQ(upper -> next, lower -> prev)){
      last -> succ = upper -> next;
      last = upper -> next;
      upper = upper -> next;
    }
    else{
      last -> succ = lower -> prev;
      last = lower -> prev;
      lower = lower -> prev;
    }
  }
  last -> succ = NULL;
}
```

And finally, the triangulation function:

Section: Local Function Declaration (metafont.c) (continuation):

```
static bool triangulate_xmonotone_polygon(struct polygon_vertex *p,
```

```
                                        float **triangles,
                                        int *number_of_triangles,
                                        struct polygon_vertex **stack);
```

The function gets the Metafont structure (to be able to warn about errors), a polygon p to be triangulated, a pointer to where the triangles generated must be stored and a pointer to the total number of triangles already generated, which must have its content updated as more triangles are generated. The last pointer is an allocated buffer that must have enough space to store all vertices of the polygon if we want. The implementation of the algorithm is:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static bool triangulate_xmonotone_polygon(struct polygon_vertex *p,
                                        float **triangles,
                                        int *number_of_triangles,
                                        struct polygon_vertex **stack){
  float *data = *triangles;
  int stack_size;
  if(p -> next -> next == p -> prev){ // Single triangle
    ADD_TRIANGLE(data, p -> x, p -> y, p -> next -> x, p -> next -> y,
                 p -> prev -> x, p -> prev -> y);
    DESTROY_POLYGON_VERTEX(p -> next);
    DESTROY_POLYGON_VERTEX(p -> prev);
    DESTROY_POLYGON_VERTEX(p);
    *triangles = data;
    return true;
  }
  // 1: Order all polygon vertices
  order_vertices_on_xmonotone_polygon(p);
  // 2: Initialize stack 'S' and push 'u_1' and 'u_2':
  stack[0] = p;
  p = p -> succ;
  stack[1] = p;
  stack_size = 2;
  // 3: Looping 'j' from 3 until 'n-1':
  while(p != NULL && p -> succ != NULL && p -> succ -> succ != NULL){
    p = p -> succ;
    // 4: If 'u_j' and the next vertex are in different paths:
    if(((p -> flag & FLAG_UPPER) && (stack[stack_size-1] -> flag & FLAG_LOWER)) ||
       ((p -> flag & FLAG_LOWER) && (stack[stack_size-1] -> flag & FLAG_UPPER))){
      int d;
      // 5 and 6: Pop everything from 'S' and create diagonal to 'u_j':
      for(d = 1; d < stack_size; d ++){
        ADD_DIAGONAL(data, p, stack[d]);
      }
      // 7: Push 'u_{j-1}' and 'u_j':
      stack[0] = stack[stack_size - 1];
      stack[1] = p;
      stack_size = 2;
    } // 8: Else (if 'u_j' and the next vertex are in the same path):
    else{
      // 9 and 10: Pop and repeat while creating diagonals is possible:
      int d = stack_size - 2, last_popped_vertex = stack_size - 1;
```

```
      // 11: Create diagonals between 'u_j' and each popped vertex:
      while(d>=0 && IS_DIAGONAL_INSIDE(stack[d], p)){
        ADD_DIAGONAL(data, p, stack[d]);
        last_popped_vertex = d;
        d --;
      }
      // 12 and 13: Push last popped vertex and 'u_j':
      stack[last_popped_vertex + 1] = p;
      stack_size = last_popped_vertex + 2;
    }
  }
  {
    int d;
    if(p -> succ != NULL)
      p = p -> succ;
    // 14: Add diagonal from 'u_n' to each vertex in the stack:
    for(d = stack_size - 2; d >= 1; d --){
      ADD_DIAGONAL(data, p, stack[d]);
    }
    ADD_DIAGONAL(data, p, p -> next);
    DESTROY_POLYGON_VERTEX(p -> next);
    DESTROY_POLYGON_VERTEX(p);
  }
  *triangles = data;
  return true;
}
```

In the code above we use the ADD_DIAGONAL macro to insert a diagonal. That is, we generate a triangle by establishing a new edge between two previously unconnected vertices which had a common neighbor. By doing this we create a triangle. One of the main tasks of this macro is to establish which is the common neighbor of both vertices. Once this is done, we can generate the triangle and we can remove one of the vertices of the polygon:

Section: Local Macros (metafont.c) (continuation):

```
#define ADD_DIAGONAL(data, v1, v2) \
        if(v1 -> prev == v2 -> next){\
          ADD_TRIANGLE(data, v1 -> x, v1 -> y, v2 -> x, v2 -> y,\
                       v1 -> prev -> x, v1 -> prev -> y);\
          v1 -> prev = v2;\
          if(v1 -> succ == v2 -> next) v1 -> succ = v1 -> succ -> succ;\
          DESTROY_POLYGON_VERTEX(v2 -> next);\
          v2 -> next = v1;\
        }\
        else if(v1 -> next == v2 -> prev){\
          ADD_TRIANGLE(data, v1 -> x, v1 -> y, v2 -> x, v2 -> y,\
                       v1 -> next -> x, v1 -> next -> y);\
          v1 -> next = v2;\
          if(v1 -> succ == v2 -> prev) v1 -> succ = v1 -> succ -> succ;\
          DESTROY_POLYGON_VERTEX(v2 -> prev);\
          v2 -> prev = v1;\
        }\
        else printf("WARNING: This should not happen (%p<-v1->%p)(%p<-v2->%p)!\n",
```

```
v1->prev, v1->next,v2->prev,v2->next);
```

The macro that destroys vertices already has been written. But the macro that inserts triangles was not. Its task will be inserting triangles in an array, ensuring that the vertices will be in counterclockwise order, so that it can be correctly rendered by the video card. After doing this, you must update the pointer for the place where the next triangle must be stored:

Section: Local Macros (metafont.c) (continuation):

```
#define ADD_TRIANGLE(data, x1, y1, x2, y2, x3, y3) \
  if(x1*y2+y1*x3+x2*y3-x3*y2-x2*y1-x1*y3 > 0){\
    data[0] = x1; data[1] = y1;\
    data[2] = x2; data[3] = y2;\
    data[4] = x3; data[5] = y3;\
  } else{\
    data[0] = x1; data[1] = y1;\
    data[2] = x3; data[3] = y3;\
    data[4] = x2; data[5] = y2;\
  }\
  (*number_of_triangles) ++;\
  data += 6;
```

Finally, we need a macro to test if a diagonal is inside the polygon or not. We assume that the connecting vertices are from the same path in the polygon (otherwise we would already know that the diagonal is internal to the polygon). The macro that does this is:

Section: Local Macros (metafont.c) (continuation):

```
#define COMMON_VERTEX(v1, v2) ((v1 -> next == v2 -> prev)?\
                               (v1 -> next):(v1 -> prev))
#define IS_DIAGONAL_INSIDE(v1, v2)\
  ((v1 -> y != v2 -> y) && \
   (((v1 -> flag & FLAG_UPPER) && \
     (COMMON_VERTEX(v1, v2)->x)*(v2->x-v1->x)/(v2->y-v1->y)+v1->y < \
      COMMON_VERTEX(v1,v2)->y)\
    ||\
   ((v1 -> flag & FLAG_LOWER) && \
    (COMMON_VERTEX(v1, v2)->x)*(v2->x-v1->x)/(v2->y-v1->y)+v1->y > \
     COMMON_VERTEX(v1,v2)->y)))
```

Having written all functions for x-monotone polygons, we can begin triangulating this specific case of concave polygons. First we convert the pen to a polygon. Then, based on the number of vertices, we allocate space for the triangles, and also check if we need to triangulate again if the pen is already triangulated. If so, we allocate space for the vertices and run the triangulation function. Finally, we use the corresponding OpenGL functions to send the triangles to the video card.

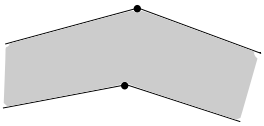
Section: Triangulation: Concave Shape:

```
if(!(pen -> flags & FLAG_CONVEX)){
  float *triang, *last_triang;
  struct polygon_vertex *poly;
  int number_of_vertices = 0, number_of_triangles = 0;
  poly = polygon_from_pen(mf, pen, transform_matrix, &number_of_vertices);
  if(poly == NULL)
    return false;
  if(pen -> gl_vbo != 0){
    if(number_of_vertices <= pen -> triang_resolution){
      destroy_vertex_linked_list(poly);
```

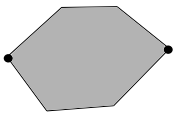
```
      return true; // Already triangulated with sufficient resolution
    }
    else
      glDeleteBuffers(1, &(pen -> gl_vbo)); // Needs to re-triangulate
  }
  pen -> triang_resolution = number_of_vertices;
  triang = temporary_alloc(3 * number_of_vertices * 2 * sizeof(float));
  if(triang == NULL){
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
    return false;
  }
  last_triang = triang;
  if(is_xmonotone(poly)){
    struct polygon_vertex **stack;
    stack = (struct polygon_vertex **)
             temporary_alloc(sizeof(struct polygon_vertex *) *
             number_of_vertices);
    if(stack == NULL){
      RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
      if(temporary_free != NULL) temporary_free(triang);
      return false;
    }
    if(!triangulate_xmonotone_polygon(poly, &last_triang, &number_of_triangles,
                                      stack)){
      if(temporary_free != NULL) temporary_free(triang);
      return false;
    }
    if(temporary_free != NULL)
      temporary_free(stack);
    glGenBuffers(1, &(pen -> gl_vbo));
    glBindBuffer(GL_ARRAY_BUFFER, pen -> gl_vbo);
    glBufferData(GL_ARRAY_BUFFER, number_of_triangles * 3 * 2 *
                 sizeof(float), triang, GL_STATIC_DRAW);
    pen -> indices = number_of_triangles * 3;
    if(temporary_free != NULL)
      temporary_free(triang);
    return true;
  }
  else{
          <Section to be inserted: Triangulation: Non-monotonous concave form>
    return false;
  }
}
return false;
```

But what if our polygon is not x-monotone? In this case, the most efficient known solution involves partitioning the polygon so that each partition is x-monotone. After this, we triangulate separately each partition. To analyze how to do so, we need to categorize the different types of vertices.

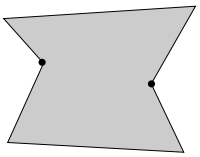
The first type is the regular vertex: a vertex which is between its neighbors in the x-axis. In a x-monotone polygon, we expect that most vertices will be regular (except for the first one and last one). The image below shows two regular vertices. The first one has an internal angle lesser than 180 degrees and the vertex below has an internal angle greater than 180 degrees.



The two next types of vertices are the start and the end vertices. A start vertex has both neighbors at its right and an internal angle lesser than 180 degrees. An end vertex has both neighbors at its left and its internal angle is lesser than 180 degrees. A x-monotone polygon should have a single start vertex and a single end vertex. The picture below shows both kind of vertices highlighted:



Finally, the last two types of vertices are the merge and split vertices. A merge vertex has both neighbors at its left, but its internal angle is greater than 180 degrees. A split vertex has both neighbors at the right, but the internal angle is greater than 180 degrees. The figure below shows a polygon with a merge vertex (at the left) and a split vertex (at the right), both highlighted:

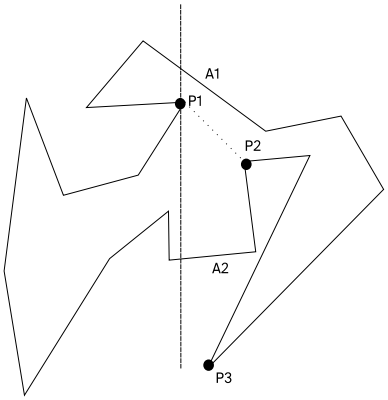


The split and merge vertices are a source of non-monotonicity. Every polygon without these types of vertices is a x-monotonous polygon. Therefore, if we could remove all split and merge vertices, we would obtain a x-monotone polygon.

For this we can create a new diagonal from each join vertex to some other vertex at its right, partitioning the polygon with such diagonal. And each merge vertex can be connected to the another vertex at its left. If we do this without creating an intersection with a previous edge, we will be able to remove all split and merge vertices.

The way in which we connect a join and merge vertex to another one involves traversing the ordered vertices, to the right or left, depending on the type of vertex. We must also identify the pair of edges that have an intersection with the vertical line that intersects the vertex that is a source of non-monotonicity. When traversing the ordered vertices, we must choose the next vertex that lies between the two edges. If there are none, we will end up choosing a vertex that belongs to such edges and we make the connection, partitioning the polygon creating a new edge in this region.

For example: in the image below, when traversing the vertices on the left to the right, we find the vertex P1, which is a source of non monotonicity. It is a join vertex. We draw an imaginary dashed vertical line, that tells us that this vertex is between the edge A1 and A2. If there were another more external edge (if vertex P3 extended further to the left), it would be ignored. So we transverse the vertices looking to the next one at the right. The first one we find is P3, which is ignored because it is not between the edges A1 and A2. After we find P2 which is between both edges and so we choose P1 and P2 as the vertices where we will "cut" our polygon partitioning it:



This means that we need to define new flags to represent the different types of vertices that we could find:

Section: Local Data Structures (metafont.c) (continuation):

```
#define TYPE_UNKNOWN_VERTEX  0
#define TYPE_REGULAR_VERTEX  4
#define TYPE_BEGIN_VERTEX    8
#define TYPE_END_VERTEX     12
#define TYPE_SPLIT_VERTEX   16
#define TYPE_MERGE_VERTEX   20
#define GET_VERTEX_TYPE(v) (((v -> flag) >> 2) << 2)
```

Now we must write code that initializes these flags in the vertex and also orders the vertices according to their x coordinate. We already have a pointer in each vertex to link them in an ordered linked list. But now, as we have a non-monotone polygon, and we may need to generate an ordering which cannot be formed just by adding new vertices in the end of the linked list. Therefore, we will need to use a double linked list. This requires an additional pointer to the predecessor of each vertex in the doubly linked list:

Section: struct polygon_vertex: Additional Variables:

```
  struct polygon_vertex *pred;
```

Finally, to generate our ordered double linked list, we will first traverse all the vertices in the order in which it are drawn. While we're doing this, we can classify each vertex and also connect them in the double linked list, which at first would be unordered. After that, we sort it using a "merge sort". By doing this, we can order and organize the vertices in $O(n \log n)$.

The function that merges two ordered double linked lists in a single one, and which is the core of our merge sort is given below:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct polygon_vertex *merge(struct polygon_vertex *begin1,
                             struct polygon_vertex *begin2){
  struct polygon_vertex *ret = NULL, *v = NULL;
  if(XMONOTONE_LEQ(begin1, begin2)){
    ret = v = begin1;
    begin1 = begin1 -> succ;
  }
  else{
    ret = v = begin2;
    begin2 = begin2 -> succ;
  }
  while(begin1 != NULL || begin2 != NULL){
    if(begin1 == NULL){
      v -> succ = begin2;
      begin2 -> pred = v;
      v = v -> succ;
      begin2 = begin2 -> succ;
    }
    else if(begin2 == NULL){
      v -> succ = begin1;
      begin1 -> pred = v;
      v = v -> succ;
      begin1 = begin1 -> succ;
    }
    else if(XMONOTONE_LEQ(begin1, begin2)){
      v -> succ = begin1;
      begin1 -> pred = v;
      v = v -> succ;
      begin1 = begin1 -> succ;
    }
    else{
      v -> succ = begin2;
      begin2 -> pred = v;
      v = v -> succ;
      begin2 = begin2 -> succ;
    }
  }
  return ret;
}
```

And the merge sort follows below:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
struct polygon_vertex *merge_sort(struct polygon_vertex *p, int size){
  if(size == 1)
    return p;
  else{
    int i = 0;
    struct polygon_vertex *p2 = p;
    while(i < size/2){
      p2 = p2 -> succ;
      i ++;
```

```
    }
    p2 -> pred -> succ = NULL;
    p2 -> pred = NULL;
    p = merge_sort(p, i);
    p2 = merge_sort(p2, size - i);
    p = merge(p, p2);
    return p;
  }
}
```

Now let's create the double linked list itself. By doing this, we must also fill in the flags of each vertex. Which includes information about the vertex type or whether it is a vertex of the bottom or top of the polygon. Remember that we know that the first vertex is not a join or merge vertex. So we know that its internal angle is not greater than 180 degrees. If we follow the path from this vertex, passing through the edges either clockwise or anticlockwise, if we are at a x-monotone vertex, then all the turns in the path will be either to the right, or to the left. But if we arrive at a join or merge vertex, with an internal angle greater than 180 degrees, then we will turn into a different direction than the initial one. Therefore, we can identify whether we are at a split or merge vertex just by observing the direction in which we turn. If all our turns are to the right, then we are following the path in counter-clockwise direction. And if they are to the left, we are in clockwise direction.

Using this logic we can create an iteration that goes through all the vertices by identifying their types and creating a doubled linked list. After that, we sort everything with a "merge sort". And finally, after sorting, as we already know which is the leftmost and rightmost vertex:

Section: Local Function Declaration (metafont.c) (continuation):

```
void prepare_non_monotonous(struct polygon_vertex *p, int number_of_vertices);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void prepare_non_monotonous(struct polygon_vertex *p, int number_of_vertices){
  bool turn_left = (p -> next -> y < p -> y);
  struct polygon_vertex *first_vertex = p;
  do{
    bool is_current_turn_left = is_turning_left(p -> prev, p, p -> next);
    if((XMONOTONE_LEQ(p, p -> next) && XMONOTONE_LEQ(p -> prev, p)) ||
       (XMONOTONE_LEQ(p, p -> prev) && XMONOTONE_LEQ(p -> next, p)))
      p -> flag = TYPE_REGULAR_VERTEX;
    // Begin or split vertex
    else if(XMONOTONE_LEQ(p, p -> next) && XMONOTONE_LEQ(p, p -> prev)){
      if(p == first_vertex)
        p -> flag = TYPE_BEGIN_VERTEX | FLAG_UPPER | FLAG_LOWER;
      else if((turn_left && is_current_turn_left) ||
              (!turn_left && !is_current_turn_left))
        p -> flag = TYPE_BEGIN_VERTEX | FLAG_UPPER | FLAG_LOWER;
      else
        p -> flag = TYPE_SPLIT_VERTEX;
    }
    else{ // End or merge vertex
      if((turn_left && is_current_turn_left) ||
         (!turn_left && !is_current_turn_left))
        p -> flag = TYPE_END_VERTEX | FLAG_UPPER | FLAG_LOWER;
      else
        p -> flag = TYPE_MERGE_VERTEX;
```

```
    }
    p -> succ = p -> next;
    p -> next -> pred = p;
    p = p -> next;
  } while(p != first_vertex);
  p -> pred -> succ = NULL;
  p -> pred = NULL;
  p = merge_sort(first_vertex, number_of_vertices);
}
```

But how can we know if we are turning to the right or to the left when we follow a path defined by three vertices? To do this, we can represent such a path by two vectors and calculate the cross product to obtain the z component. If the result is positive, walking along the vertices means turning towards the left. If it's negative, we're turning to the right. So the function below can be used to give us the answer:

Section: Local Function Declaration (metafont.c) (continuation):

```
static bool is_turning_left(struct polygon_vertex *p1,
                            struct polygon_vertex *p2,
                            struct polygon_vertex *p3);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static bool is_turning_left(struct polygon_vertex *p1,
                            struct polygon_vertex *p2,
                            struct polygon_vertex *p3){
  float v1_x, v1_y, v2_x, v2_y;
  v1_x = p2 -> x - p1 -> x;
  v1_y = p2 -> y - p1 -> y;
  v2_x = p3 -> x - p2 -> x;
  v2_y = p3 -> y - p2 -> y;
  return ((v1_x * v2_y - v1_y * v2_x) > 0);
}
```

Now go back to the last image we showed of a concave polygon. We had commented that the way to partitioning it is separating it in x-monotone polygons. To do so, we imagine a vertical line that moves through the region occupied by the polygon, from left to right. Using this straight line, we would know which edges exist in the same coordinate x of each vertex, and so we could know when we can make a new partition connecting two different vertices.

Our vertical line will be represented by two different pieces of information: which vertex it is passing through at this exact moment (we can simulate it traveling through the polygon area from left to right making her jump from one vertex to another, following the order of vertices from left to right in the double linked list) and which edges it is crossing in a given time. Furthermore, each edge that is below some vertex must store as additional information what is its "helper". The "helper" is the vertex to the left of our imaginary line closest to it such that the segment that connects such a vertex to the given edge is internal to the polygon. As the imaginary vertical line travels from left to right, the "helper" of each edge can change. Storing "helpers" is important because they are possible candidates that we should consider when we want to create a new partition.

The list of edges, where each one might potentially store a "helper", is more efficiently represented as a binary tree. It allows us to search for relevant edges in $O(\log n)$, which helps to better handle pathological polygons with a very large number of edges crossing the same coordinate on x-axis. The nodes in the tree will be sorted according with their position in the axis y (using axis x as tiebreaker). Each node in our tree is:

Section: Local Data Structures (metafont.c) (continuation):

```
struct polygon_edge{
```

```
  float x1, y1, x2, y2; // Pontos da aresta
  struct polygon_vertex *helper;
  struct polygon_edge *parent, *left, *right;
};
#define NEW_POLYGON_EDGE() \
  (struct polygon_edge *) temporary_alloc(sizeof(struct polygon_edge));
#define INITIALIZE_POLYGON_EDGE(p, x1, y1, x2, y2, helper) {\
 p -> x1 = x1; p -> x2 = x2; p -> y1 = y1; p -> y2 = y2;\
 p -> helper = helper;\
 p -> parent = p -> left = p -> right = NULL;}
#define DESTROY_POLYGON_EDGE(p) \
  ((temporary_free != NULL)?(temporary_free(p)):(true))
```

To insert a new edge in the tree, we can use the function below. We will write the function in a way that the user calling the function will pass as argument which function we will use to compare which edge is lesser or equal other. This allow us to use the tree to sort edges in different ways: according with their first vertex, according with the helper vertex, etc. The user optionally could also use as last argument a function used to compare if two edges are equal. If she does so, then we use it to avoid storing equal edges in the tree:

Section: Local Function Declaration (metafont.c) (continuation):

```
static struct polygon_edge *insert_polygon_edge(struct polygon_edge **,
                                                float, float, float, float,
                                                struct polygon_vertex *,
                                               bool (*)(struct polygon_edge *,
                                                        struct polygon_edge *),
                                               bool (*)(struct polygon_edge *,
                                                        struct polygon_edge *));
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static void insert_polygon_edge_aux(struct polygon_edge *tree,
                                    struct polygon_edge *new_edge,
                                    bool (*leq)(struct polygon_edge *p1,
                                                struct polygon_edge *p2),
                                    bool (*eq)(struct polygon_edge *p1,
                                               struct polygon_edge *p2)){
  if(leq(new_edge, tree)){
    if(eq != NULL && eq(new_edge, tree)){
      DESTROY_POLYGON_EDGE(new_edge);
      return;
    }
    if(tree -> left == NULL)
      tree -> left = new_edge;
    else
      insert_polygon_edge_aux(tree -> left, new_edge, leq, eq);
  }
  else{
    if(tree -> right == NULL)
      tree -> right = new_edge;
    else
      insert_polygon_edge_aux(tree -> right, new_edge, leq, eq);
  }
```

```
}
static struct polygon_edge *insert_polygon_edge(struct polygon_edge **tree,
                                                float x1, float y1,
                                                float x2, float y2,
                                                struct polygon_vertex *helper,
                                           bool (*leq)(struct polygon_edge *p1,
                                                       struct polygon_edge *p2),
                                           bool (*eq)(struct polygon_edge *p1,
                                                      struct polygon_edge *p2)){
  struct polygon_edge *new_edge;
  new_edge = NEW_POLYGON_EDGE();
  if(new_edge == NULL)
    return NULL;
  INITIALIZE_POLYGON_EDGE(new_edge, x1, y1, x2, y2, helper);
  if(*tree == NULL)
    *tree = new_edge;
  else{
    struct polygon_edge *current = *tree;
    insert_polygon_edge_aux(current, new_edge, leq, eq);
  }
  return new_edge;
}
```

But how should we order the elements in our tree? We need a function to create a order relationship, so that we can use the above functions:

Section: Local Function Declaration (metafont.c) (continuation):

```
static bool leq_by_vertex(struct polygon_edge *, struct polygon_edge *);
```

For the order relationship, consider that one of the use cases for this tree will be storing the edges that an imaginary vertical line is crossing. And that we may need to search in these edges which is the edge directly below a given point. This means that it will be useful to order the edges according with the point y where our imaginary vertical line is crossing. As we will not have an edge that crosses another edge, and as we will remove from the tree any edge not being crossed by the imaginary vertical line, we can get such ordering just checking, given two edges, which have the leftmost coordinate, obtaining a line equation that extends it, and using this line equation to check if the leftmost coordinate of the other edge is above or below it:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static bool leq_by_vertex(struct polygon_edge *p1, struct polygon_edge *p2){
  if(p1 -> x1 == p1 -> x2 && p2 -> x1 == p2 -> x2){ // 2 vertical vertices
    return (p1 -> y1 <= p2 -> y1);
  }
  else if(p1 -> x1 == p1 -> x2){ // 1st vertical vertex
    float slope = (p2 -> y2 - p2 -> y1)/(p2 -> x2 - p2 -> x1);
    float b = p2 -> y1 - slope * p2 -> x1;
    if(p1 -> y1 <= p1 -> y2)
      return (p1 -> y2 <= (slope * p1 -> x2 + b));
    else
      return (p1 -> y1 <= (slope * p1 -> x1 + b));
  }
  else if(p2 -> x1 == p2 -> x2){ // 2nd vertical vertice
    float slope = (p1 -> y2 - p1 -> y1)/(p1 -> x2 - p1 -> x1);
    float b = p1 -> y1 - slope * p1 -> x1;
```

```
    if(p2 -> y1 <= p2 -> y2)
      return (slope * p2 -> x1 + b) <= p2 -> y1;
    else
      return (slope * p2 -> x2 + b) <= p2 -> y2;
  }
  if((p1 -> x1 <= p2 -> x1 && p1 -> x1 <= p2 -> x2) ||
     (p1 -> x2 <= p2 -> x1 && p1 -> x2 <= p2 -> x2)){ // p1 comes first
    float slope = (p1 -> y2 - p1 -> y1)/(p1 -> x2 - p1 -> x1);
    float b = p1 -> y1 - slope * p1 -> x1;
    if(p2 -> x1 <= p2 -> x2)
      return (slope * p2 -> x1 + b <= p2 -> y1);
    else
      return (slope * p2 -> x2 + b <= p2 -> y2);
  }
  else{ // p2 comes first
    float slope = (p2 -> y2 - p2 -> y1)/(p2 -> x2 - p2 -> x1);
    float b = p2 -> y1 - slope * p2 -> x1;
    if(p1 -> x1 <= p1 -> x2)
      return (p1 -> y1 <= slope * p1 -> x1 + b);
    else
      return (p1 -> y2 <= slope * p1 -> x2 + b);
  }
}
```

The equality test tor edges is simpler. We just need to compare the vertices. But taking into account that their order does not matter: the vertex $(0,1)--(1,0)$ is equal to $(1,0)--(0,1)$:

Section: Local Function Declaration (metafont.c) (continuation):

```
static bool eq_by_vertex(struct polygon_edge *, struct polygon_edge *);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static bool eq_by_vertex(struct polygon_edge *p1, struct polygon_edge *p2){
  return (p1 -> x1 == p2 -> x1 && p1 -> y1 == p2 -> y1 &&
          p1 -> x2 == p2 -> x2 && p1 -> y2 == p2 -> y2) ||
         (p1 -> x1 == p2 -> x2 && p1 -> y1 == p2 -> y2 &&
          p1 -> x2 == p2 -> x1 && p1 -> y2 == p2 -> y1);
}
```

To remove an edge from the tree, we use the code:

Section: Local Function Declaration (metafont.c) (continuation):

```
static struct polygon_edge *remove_polygon_edge(struct polygon_edge **,
                                                float, float, float, float,
                                              bool (*)(struct polygon_edge *,
                                                       struct polygon_edge *),
                                               bool (*)(struct polygon_edge *,
                                                       struct polygon_edge *));
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static struct polygon_edge *remove_polygon_edge(struct polygon_edge **tree,
                                                float x1, float y1, float x2,
                                                float y2,
                                              bool (*leq)(struct polygon_edge *p1,
```

```
                                                    struct polygon_edge *p2),
                                          bool (*eq)(struct polygon_edge *p1,
                                                    struct polygon_edge *p2)){
struct polygon_edge *current = *tree, searched;
searched.x1 = x1; searched.x2 = x2;
searched.y1 = y1; searched.y2 = y2;
if(current == NULL)
  return NULL;
if(eq(current, &searched)){
   if(current -> left == NULL && current -> right == NULL){
     *tree = NULL; // Removed node has no children
     return current;
   }
   else if(current -> right == NULL){
     *tree = current -> left; // Removed node has a single left child
     current -> left -> parent = current -> parent;
     return current;
   }
   else if(current -> left == NULL){
     *tree = current -> right; // Removed node has a single right child
     current -> right -> parent = current -> parent;
     return current;
   }
   else{ // Removed node has 2 children:
     // suc: removed node successor
     struct polygon_edge *suc = current -> right;
     while(suc -> left != NULL)
       suc = suc -> left;
     if(suc == current -> right)
       current -> right = suc -> right;
     else
       suc -> parent -> left = suc -> right;
     suc -> right = current -> right;
     suc -> left = current -> left;
     *tree = suc;
     return current;
   }
}
else{ // Not found yet: searching in current node children:
  if(leq(&searched, current)){
    if(current -> left != NULL)
      return remove_polygon_edge(&(current -> left),x1, y1, x2, y2, leq, eq);
    else
      return NULL;
  }
  else{
    if(current -> right != NULL)
      return remove_polygon_edge(&(current -> right),x1, y1, x2, y2,leq, eq);
    else
      return NULL;
  }
```

```
  }
}
```

Finally, we will need a function to get the edge immediately below some (x, y) coordinate. For this, we use the function below:

Section: Local Function Declaration (metafont.c) (continuation):

```
static struct polygon_edge *find_edge_below(struct polygon_edge *,
                                            float, float);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static struct polygon_edge *find_edge_below(struct polygon_edge *tree,
                                            float x, float y){
  float slope, b;
  if(tree == NULL)
    return NULL;
  slope = (tree -> y2 - tree -> y1)/(tree -> x2 - tree -> x1);
  b = tree -> y1 - slope * tree -> x1;
  if(slope * x + b <= y){ // Valid edge. Can we find a greater one?
    struct polygon_edge *candidate;
    candidate = find_edge_below(tree -> right, x, y);
    if(candidate != NULL)
      return candidate;
    else return tree;
  }
  // Not valid edge. Try to find one smaller:
  else return find_edge_below(tree -> left, x, y);
}
```

We must also be able to partition a polygon, passing the pointer to two of its vertices as argument. Such vertices must then be joined together and the polygon will be divided into two. To do this, we must clone both vertices, now having two copies of each. Both copies will be connected, each in a different way. Neighboring vertices will be updated consistently. We use two final pointers received as an argument to store the pointers to the newly created polygons:

Section: Local Function Declaration (metafont.c) (continuation):

```
static bool cut_polygon(struct polygon_vertex *v1, struct polygon_vertex *v2,
                        struct polygon_vertex **new1,
                        struct polygon_vertex **new2);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static bool cut_polygon(struct polygon_vertex *v1, struct polygon_vertex *v2,
                        struct polygon_vertex **new1,
                        struct polygon_vertex **new2){
  struct polygon_vertex *vv1, *vv2;
  bool turn1 = is_turning_left(v1 -> prev, v1, v1 -> next);
  bool turn2 = is_turning_left(v2 -> prev, v2, v2 -> next);
  vv1 = NEW_POLYGON_VERTEX();
  if(vv1 == NULL)
    return false;
  vv2 = NEW_POLYGON_VERTEX();
  if(vv2 == NULL)
    return false;
```

```
  // Remove flags:
  v1 -> flag &= (~0x3);
  v2 -> flag &= (~0x3);
  // Separate:
  memcpy(vv1, v1, sizeof(struct polygon_vertex));
  memcpy(vv2, v2, sizeof(struct polygon_vertex));
  // There are 2 ways to split the polygon. We discuss below why we need
  // the following test:
  if((is_turning_left(v2, v1, v1 -> next) != turn1 &&
     is_turning_left(vv1 -> prev, vv1, vv2) == turn1) ||
     (is_turning_left(v2 -> prev, v2, v1) != turn2 &&
     is_turning_left(vv1, vv2, vv2 -> next) == turn2)){
     v1 -> next -> prev = vv1;
     v1 -> next = v2;
     v2 -> prev -> next = vv2;
     v2 -> prev = v1;
     vv1 -> prev -> next = v1;
     vv1 -> prev = vv2;
     vv2 -> next -> prev = v2;
     vv2 -> next = vv1;
  }
  else{
    vv1 -> next -> prev = v1;
    vv1 -> next = vv2;
    vv2 -> prev -> next = v2;
    vv2 -> prev = vv1;
    v1 -> prev -> next = vv1;
    v1 -> prev = v2;
    v2 -> next -> prev = vv2;
    v2 -> next = v1;
  }
  // Find pointer for the new polygon:
  while(XMONOTONE_LEQ(vv1 -> prev, vv1))
    vv1 = vv1 -> prev;
  while(XMONOTONE_LEQ(vv1 -> next, vv1))
    vv1 = vv1 -> next;
  *new1 = vv1;
  while(XMONOTONE_LEQ(v1 -> prev, v1))
    v1 = v1 -> prev;
  while(XMONOTONE_LEQ(v1 -> next, v1))
    v1 = v1 -> next;
  *new2 = v1;
  return true;
}
```

In the code, the most mysterious part if when we test the direction in which we turn around follow the vertices where we make a cut and how this influences the way we cut. To understand why we do this, consider that if we are cutting a polygon, at least one of the vertices that we are connecting with a new diagonal has an internal angle greater than 180 degrees. If it has a smaller internal angle, the angle will remain small and this never will be a problem for us. If the internal angle is greater, then after cutting, either all angles become smaller than 180 degrees, or else one of the newly created polygons will keep having a greater internal angle. If so, further cuts must be made at that vertex until its angle is reduced. But as we

may have previously identified which vertices we need to cut, we may have stored pointers to these vertices. Because of this, in the function above, we must not modify the pointer to any vertex that will still need further cutting. If after the cut we discover that there is still big internal angles in some vertex, we must keep using the pointers `v1` and `v2` to represent them, not the new pointers `vv1` and `vv2`. And the way that we check for this, is checking if the turning direction changed in some vertex (which means that we had a big angle that becomes small).

Now we put all this together to create the triangulation for the general case, when we do not have a x-monotone polygon. First we order the vertices creating the doubly linked list between them, then initialize an empty binary tree to represent the straight imaginary line that will help us with the partition. We will also use a linked list containing the vertices where we will add new diagonals, cutting the polygon and partitioning it in other polygons. After the initialization, we position our imaginary straight vertical line in the leftmost vertex and from there, we will slide the line to the right. In the process we will discover all the diagonals that need to be added and later we will use them to partition the polygon.

We will try to avoid creating new data structures for this. Our list of diagonals that need to be added to cut our polygon in several parts will use the vertex structure to represent each new diagonal. The pointers `next` and `prev` will store which vertices each diagonal is joining. The pointer `succ` will connect each new diagonal in a linked list. Assuming that we have a pointer called `list_of_diagonals` that points to the beginning of the linked list and a pointer called `last_diagonal` pointing to the last diagonal inserted in the list, we could add new diagonals representing future cuts with the macro below:

Section: Local Macros (metafont.c) (continuation):

```
#define ADD_CUT(v1, v2) {                                          \
  if(last_diagonal == NULL){                                       \
    last_diagonal = list_of_diagonals = NEW_POLYGON_VERTEX();\
  } else{                                                          \
    last_diagonal -> succ = NEW_POLYGON_VERTEX();                  \
    last_diagonal = last_diagonal -> succ;                         \
  }                                                                \
  if(last_diagonal == NULL){                                       \
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0);                            \
    return false;                                                  \
  }                                                                \
  last_diagonal -> prev = v1;                                      \
  last_diagonal -> next = v2;                                      \
  last_diagonal -> succ = NULL;}
```

Later, when we use the diagonal list to partition out polygon, the newly created polygons also need to be stored. For this we will use the same tree structure that we used to represent our imaginary vertical line walking over the polygon. Each tree node will use the `helper` pointer to point to a unique newly created polygon. The other node attributes can be ignored. We will insert and store things in this tree sorting according with the pointer `helper`. For this, we will need the following comparison functions:

Section: Local Function Declaration (metafont.c) (continuation):

```
static bool leq_by_helper(struct polygon_edge *p1, struct polygon_edge *p2);
static bool eq_by_helper(struct polygon_edge *p1, struct polygon_edge *p2);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static bool leq_by_helper(struct polygon_edge *p1, struct polygon_edge *p2){
  return (p1 -> helper <= p2 -> helper);
}
static bool eq_by_helper(struct polygon_edge *p1, struct polygon_edge *p2){
  return (p1 -> helper == p2 -> helper);
}
```

And finally, the code that performs the described operations to triangulate non-monotonous concave polygons:

Section: Triangulation: Non-monotonous concave form:

```
{
  int i;
  float *last_triangle = triang;
  struct polygon_edge *list_of_subpolygons = NULL, *imaginary_line = NULL;
  struct polygon_vertex *current_vertex = poly, *last_vertex;
  struct polygon_vertex *list_of_diagonals = NULL, *last_diagonal = NULL;
  struct polygon_vertex **buffer;
  bool clockwise = is_turning_left(poly -> next, poly, poly -> prev);
  buffer = (struct polygon_vertex **)
            temporary_alloc(sizeof(struct polygon_vertex *) *
            number_of_vertices);
  if(buffer == NULL){
    RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
    if(temporary_free != NULL) temporary_free(triang);
    return false;
  }
  prepare_non_monotonous(poly, number_of_vertices);
  for(i = 0; i < number_of_vertices; i ++){
    // Here we create the list of new diagonals
          <Section to be inserted: Makes X-Monotonous: Adding Diagonals>
    current_vertex = current_vertex -> succ;
  }
  current_vertex = list_of_diagonals;
  // Partitioning:
  while(current_vertex != NULL){
    struct polygon_vertex *new1 = NULL, *new2 = NULL;
    cut_polygon(current_vertex -> next, current_vertex -> prev, &new1, &new2);
    insert_polygon_edge(&list_of_subpolygons, 0, 0, 0, 0, new1, leq_by_helper,
                        eq_by_helper);
    insert_polygon_edge(&list_of_subpolygons, 0, 0, 0, 0, new2, leq_by_helper,
                        eq_by_helper);
    last_vertex = current_vertex;
    current_vertex = current_vertex -> succ;
    DESTROY_POLYGON_VERTEX(last_vertex);
  }
  triangulate_polygon_tree(list_of_subpolygons, &last_triangle,
                           &number_of_triangles, buffer);
  //destroy_edge_tree(list_of_subpolygons);
  glGenBuffers(1, &(pen -> gl_vbo));
  glBindBuffer(GL_ARRAY_BUFFER, pen -> gl_vbo);
  glBufferData(GL_ARRAY_BUFFER, number_of_triangles * 3 * 2 *
               sizeof(float), triang, GL_STATIC_DRAW);
  pen -> indices = number_of_triangles * 3;
  if(temporary_free != NULL){
    temporary_free(buffer);
    temporary_free(triang);
  }
```

```
  return true;
}
```

After creating all partitions and storing all the new polygons in a binary tree, it is time to triangulate all them using the following function:

Section: Local Function Declaration (metafont.c) (continuation):

```
static void triangulate_polygon_tree(struct polygon_edge *tree,
                                     float **triangles,
                                     int *number_of_triangles,
                                     struct polygon_vertex **buffer);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static void triangulate_polygon_tree(struct polygon_edge *tree,
                                     float **triangles,
                                     int *number_of_triangles,
                                     struct polygon_vertex **buffer){
  if(tree == NULL)
    return;
  if(tree -> left != NULL)
    triangulate_polygon_tree(tree -> left, triangles, number_of_triangles,
                             buffer);
  if(tree -> right != NULL)
    triangulate_polygon_tree(tree -> right, triangles, number_of_triangles,
                             buffer);
  if(!is_xmonotone(tree -> helper))
    fprintf(stderr, "WARNING: This should not happen: non-monotonous pen!\n");
  triangulate_xmonotone_polygon(tree -> helper, triangles,
                                number_of_triangles, buffer);
  DESTROY_POLYGON_EDGE(tree);
  return;
}
```

Now all that's left to do is add diagonals so we can partition the polygon. To do this, we will move our imaginary vertical line making it pass through each vertex from left to right. What we will do in each case depends on the type of vertex that we find. If it is a begin vertex, we must register the lower edge connected to it as being being crossed by our imaginary straight line (we store it in the tree), and we make store as its "helper" the current begin vertex:

Section: Makes X-Monotonous: Adding Diagonals:

```
if(GET_VERTEX_TYPE(current_vertex) == TYPE_BEGIN_VERTEX){
  if(current_vertex -> prev -> y <= current_vertex -> next -> y)
    insert_polygon_edge(&imaginary_line, current_vertex -> x,
                        current_vertex -> y, current_vertex -> prev -> x,
                        current_vertex -> prev -> y,
                        current_vertex, leq_by_vertex, NULL);
  else
    insert_polygon_edge(&imaginary_line, current_vertex -> x,
                        current_vertex -> y, current_vertex -> next -> x,
                        current_vertex -> next -> y,
                        current_vertex, leq_by_vertex, NULL);
}
```

If we find an end vertex, then we check whether the upper edge connected to it has as "helper" a join

vertex. If so, we create a new diagonal between the current vertex and the join vertex. In any case, our imaginary line will no longer cross the found edge after this point and we must remove it from the tree.

Section: Makes X-Monotonous: Adding Diagonals (continuation):

```
else if(GET_VERTEX_TYPE(current_vertex) == TYPE_END_VERTEX){
  struct polygon_edge *removed;
  if(current_vertex -> prev -> y <= current_vertex -> next -> y)
    removed = remove_polygon_edge(&imaginary_line, current_vertex -> x,
                                  current_vertex -> y,
                                  current_vertex -> prev -> x,
                                  current_vertex -> prev -> y,
                                  leq_by_vertex, eq_by_vertex);
  else
    removed = remove_polygon_edge(&imaginary_line, current_vertex -> x,
                                  current_vertex -> y,
                                  current_vertex -> next -> x,
                                  current_vertex -> next -> y,
                                  leq_by_vertex, eq_by_vertex);
```

```
  if(GET_VERTEX_TYPE(removed -> helper) == TYPE_MERGE_VERTEX){
    ADD_CUT(current_vertex, removed -> helper);
  }
  DESTROY_POLYGON_EDGE(removed);
}
```

If we find a split vertex, then we search for the edge immediately below the current vertex. We do this searching for the edges in the tree. We create a new diagonal between the current vertex and the helper of the found edge. The current vertex then becomes the new helper for that edge. And we add to the tree the lower edge connected to the current vertex, as our imaginary line will then cross this edge:

Section: Makes X-Monotonous: Adding Diagonals (continuation):

```
else if(GET_VERTEX_TYPE(current_vertex) == TYPE_SPLIT_VERTEX){
  struct polygon_edge *below;
  below = find_edge_below(imaginary_line, current_vertex -> x,
                          current_vertex -> y);
  ADD_CUT(current_vertex, below -> helper);
  below -> helper = current_vertex;
  if(current_vertex -> prev -> y <= current_vertex -> next -> y)
    insert_polygon_edge(&imaginary_line, current_vertex -> x,
                        current_vertex -> y, current_vertex -> next -> x,
                        current_vertex -> next -> y,
                        current_vertex, leq_by_vertex, NULL);
  else
    insert_polygon_edge(&imaginary_line, current_vertex -> x,
                        current_vertex -> y, current_vertex -> prev -> x,
                        current_vertex -> prev -> y,
                        current_vertex, leq_by_vertex, NULL);
}
```

If we find a merge vertex, we get the upper edge connected to that vertex and check if its "helper" is another merge vertex. If so, we create a cut diagonal between this vertex and the current vertex. After this check, we remove that edge from our tree, as the imaginary vertical line will not cross it anymore. Then, we check which is the edge in our tree immediately below our current vertex. If this edge helper is another

merge vertex, we create another diagonal between it and the current vertex. Finally, we update the helper of the found vertex to point to the current vertex:

Section: Makes X-Monotonous: Adding Diagonals (continuation):

```
else if(GET_VERTEX_TYPE(current_vertex) == TYPE_MERGE_VERTEX){
  struct polygon_edge *removed, *below;
  if(current_vertex -> prev -> y <= current_vertex -> next -> y)
    removed = remove_polygon_edge(&imaginary_line, current_vertex -> x,
                                  current_vertex -> y,
                                  current_vertex -> next -> x,
                                  current_vertex -> next -> y,
                                  leq_by_vertex, eq_by_vertex);
  else
    removed = remove_polygon_edge(&imaginary_line, current_vertex -> x,
                                  current_vertex -> y,
                                  current_vertex -> prev -> x,
                                  current_vertex -> prev -> y,
                                  leq_by_vertex, eq_by_vertex);
  if(GET_VERTEX_TYPE(removed -> helper) == TYPE_MERGE_VERTEX){
    ADD_CUT(current_vertex, removed -> helper);
  }
  DESTROY_POLYGON_EDGE(removed);
  below = find_edge_below(imaginary_line, current_vertex -> x,
                          current_vertex -> y);
  if(GET_VERTEX_TYPE(below -> helper) == TYPE_MERGE_VERTEX){
    ADD_CUT(current_vertex, below -> helper);
  }
  below -> helper = current_vertex;
}
```

The last case is when we have a regular vertex. What we do depends if the polygon interior is above or below our vertex. We check this according with the position of our neighbor vertices and knowing if we stored our vertices clockwise or counterclockwise.

If the polygon interior is above the vertex, we check if the previous edge has a "helper" which is a merge vertex. If so, we connect it with current vertex. Anyway, we remove the previous edge and add the next edge to the tree, using the current vertex as the helper.

If the polygon interior is below the vertex, we look for the edge immediately below. If its "helper" is a merge vertex, we connect it to the current vertex. Anyway, we set the current vertex as the new "helper" for that edge.

Section: Makes X-Monotonous: Adding Diagonals (continuation):

```
else{
  struct polygon_edge *removed;
  if((clockwise && // Checa se interior est acima:
     XMONOTONE_LEQ(current_vertex -> next, current_vertex -> prev)) ||
     (!clockwise &&
      XMONOTONE_LEQ(current_vertex -> prev, current_vertex -> next))){
    struct polygon_vertex *to_append;
    if(current_vertex -> prev -> x <= current_vertex -> next -> x){
      removed = remove_polygon_edge(&imaginary_line, current_vertex -> x,
                                    current_vertex -> y,
                                    current_vertex -> prev -> x,
                                    current_vertex -> prev -> y,
```

```
                                    leq_by_vertex, eq_by_vertex);
      to_append = current_vertex -> next;
    }
    else{
      removed = remove_polygon_edge(&imaginary_line, current_vertex -> x,
                                    current_vertex -> y,
                                    current_vertex -> next -> x,
                                    current_vertex -> next -> y,
                                    leq_by_vertex, eq_by_vertex);
      to_append = current_vertex -> prev;
    }
    if(GET_VERTEX_TYPE(removed -> helper) == TYPE_MERGE_VERTEX){
      ADD_CUT(current_vertex, removed -> helper);
    }
    DESTROY_POLYGON_EDGE(removed);
    insert_polygon_edge(&imaginary_line, current_vertex -> x,
                        current_vertex -> y, to_append -> x,
                        to_append -> y, current_vertex, leq_by_vertex, NULL);
  }
  else{ // The polygon interior is below:
    struct polygon_edge *below;
    below = find_edge_below(imaginary_line, current_vertex -> x,
                            current_vertex -> y);
    if(GET_VERTEX_TYPE(below -> helper) == TYPE_MERGE_VERTEX){
      ADD_CUT(current_vertex, below -> helper);
    }
    below -> helper = current_vertex;
  }
}
```

11.3. Reading the `pickup` Command

Now we will define what will happen when we read the `pickup` command.

1) First we read the next token. If it is a `nullpen`, we transform `currentpen` in a null pen, and if it is a `pencircle`, we transform in a circular pen. If it is a pen variable, we set `currentpen` to point to that given variable. In either case, we reset `currentpen` to an identity matrix. IN the other cases, or if the variable is not initialized, we return an error.

Section: Statement: Command:

```
else if(begin -> type == TYPE_PICKUP){
  struct generic_token *end_expression = *end;
  struct generic_token *next_token = begin -> next;
  if(begin == *end || (next_token -> type != TYPE_NULLPEN &&
                       next_token -> type != TYPE_SYMBOLIC &&
                       next_token -> type != TYPE_PENCIRCLE &&
                       next_token -> type != TYPE_PENSEMICIRCLE)){
    RAISE_ERROR_NO_PICKUP_PEN(mf, cx, OPTIONAL(next_token -> line));
    return false;
  }
  if(cx -> currentpen -> gl_vbo != 0)
    glDeleteBuffers(1, &(cx -> currentpen -> gl_vbo));
  if(next_token -> type == TYPE_NULLPEN){
    cx -> currentpen -> flags = FLAG_NULL;
```

```
    cx -> currentpen -> referenced = NULL;
    cx -> currentpen -> gl_vbo = 0;
  }
  else if(next_token -> type == TYPE_PENCIRCLE){
    cx -> currentpen -> flags = FLAG_CONVEX | FLAG_CIRCULAR;
    cx -> currentpen -> referenced = NULL;
    cx -> currentpen -> gl_vbo = 0;
  }
  else if(next_token -> type == TYPE_PENSEMICIRCLE){
    cx -> currentpen -> flags = FLAG_CONVEX | FLAG_SEMICIRCULAR;
    cx -> currentpen -> referenced = NULL;
    cx -> currentpen -> gl_vbo = 0;
  }
  else{
    struct symbolic_token *symbol = ((struct symbolic_token *) next_token);
    struct pen_variable *var = (struct pen_variable *) (symbol -> var);
    if(var == NULL){
      RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, OPTIONAL(next_token -> line),
                                      symbol);
      return false;
    }
    if(var -> type != TYPE_T_PEN){
      RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, OPTIONAL(next_token -> line),
                                      symbol, var -> type,
                                      TYPE_T_PEN);
      return false;
    }
    if(var -> format == NULL && var -> flags == false){
      RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, OPTIONAL(next_token -> line),
                                         symbol, TYPE_T_PEN);
      return false;
    }
    cx -> currentpen -> referenced = var;
  }
  INITIALIZE_IDENTITY_MATRIX(cx -> currentpen -> gl_matrix);
                <Section to be inserted: 'pickup' Command: Continue>
  return true;
}
```

After gettting the pen basis form, now we read the linear transforms specific to be applied to the `currentpen`. While we are not in the end of the expression, we read the next token, which should describe a linear transform like `shifted`, `rotated` or something like this. Based in the transform, we read and evaluate the following tokens as a numeric, pair or transform expression. And finally perform the given transform in the `currentpen` matrix. We keep doing this until we find no more new transforms to be performed.

Section: 'pickup' Command: Continue:

```
while(next_token != end_expression){
  struct generic_token *begin_subexpr, *end_subexpr;
  DECLARE_NESTING_CONTROL();
  next_token = next_token -> next;
  if(next_token == NULL){
```

```
  RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
  return false;
}
if(next_token == end_expression){
  if(next_token -> type == TYPE_ROTATED || next_token -> type == TYPE_SCALED ||
     next_token -> type == TYPE_SLANTED || next_token -> type == TYPE_XSCALED ||
     next_token -> type == TYPE_YSCALED){
    RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_NUMERIC);
  }
  else if(next_token -> type == TYPE_SHIFTED || next_token -> type == TYPE_ZSCALED){
    RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PAIR);
  }
  else if(next_token -> type == TYPE_TRANSFORMED){
    RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_TRANSFORM);
  }
  else{
    RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), next_token);
  }
  return false;
}
begin_subexpr = next_token -> next;
end_subexpr = begin_subexpr;
while(end_subexpr != end_expression){
  struct generic_token *next = end_subexpr -> next;
  COUNT_NESTING(end_subexpr);
  if(IS_NOT_NESTED() &&
     (next -> type == TYPE_ROTATED || next -> type == TYPE_SCALED ||
      next -> type == TYPE_SHIFTED || next -> type == TYPE_SLANTED ||
      next -> type == TYPE_XSCALED || next -> type == TYPE_YSCALED ||
      next -> type == TYPE_ZSCALED || next -> type == TYPE_TRANSFORMED))
    break;
  end_subexpr = next;
}
switch(next_token -> type){
  struct numeric_variable a;
  struct pair_variable p;
  struct transform_variable t;
case TYPE_ROTATED:
  if(!eval_numeric_expression(mf, cx, begin_subexpr, end_subexpr, &a))
    return false;
  TRANSFORM_ROTATE(cx -> currentpen -> gl_matrix, a.value * 0.0174533);
  break;
case TYPE_SCALED:
  if(!eval_numeric_expression(mf, cx, begin_subexpr, end_subexpr, &a))
    return false;
  TRANSFORM_SCALE(cx -> currentpen -> gl_matrix, a.value);
  // Curved pens need retriangulation if the size increase:
  if(cx -> currentpen -> referenced){
    struct pen_variable *v = cx -> currentpen -> referenced;
    if(v -> gl_vbo != 0 && a.value > 1.0 && !(v -> flags & FLAG_STRAIGHT)){
      glDeleteBuffers(1, &(v -> gl_vbo));
```

```
      v -> gl_vbo = 0;
    }
  }
  break;
case TYPE_SHIFTED:
  if(!eval_pair_expression(mf, cx, begin_subexpr, end_subexpr, &p))
    return false;
  TRANSFORM_SHIFT(cx -> currentpen -> gl_matrix, p.x, p.y);
  break;
case TYPE_SLANTED:
  if(!eval_numeric_expression(mf, cx, begin_subexpr, end_subexpr, &a))
    return false;
  TRANSFORM_SLANT(cx -> currentpen -> gl_matrix, a.value);
  // Slant non-circular curved pens always require retriangulation:
  if(cx -> currentpen -> referenced){
    struct pen_variable *v = cx -> currentpen -> referenced;
    if(v -> gl_vbo != 0 && a.value > 1.0 && !(v -> flags & FLAG_STRAIGHT) &&
       !(v -> flags & FLAG_CIRCULAR)){
      glDeleteBuffers(1, &(v -> gl_vbo));
      v -> gl_vbo = 0;
    }
  }
  break;
case TYPE_XSCALED:
  if(!eval_numeric_expression(mf, cx, begin_subexpr, end_subexpr, &a))
    return false;
  TRANSFORM_SCALE_X(cx -> currentpen -> gl_matrix, a.value);
  // Curved pens need retriangulation if the size increase:
  if(cx -> currentpen -> referenced){
    struct pen_variable *v = cx -> currentpen -> referenced;
    if(v -> gl_vbo != 0 && a.value > 1.0 && !(v -> flags & FLAG_STRAIGHT)){
      glDeleteBuffers(1, &(v -> gl_vbo));
      v -> gl_vbo = 0;
    }
  }
  break;
case TYPE_YSCALED:
  if(!eval_numeric_expression(mf, cx, begin_subexpr, end_subexpr, &a))
    return false;
  TRANSFORM_SCALE_Y(cx -> currentpen -> gl_matrix, a.value);
  // Curved pens need retriangulation if the size increase:
  if(cx -> currentpen -> referenced){
    struct pen_variable *v = cx -> currentpen -> referenced;
    if(v -> gl_vbo != 0 && a.value > 1.0 && !(v -> flags & FLAG_STRAIGHT)){
      glDeleteBuffers(1, &(v -> gl_vbo));
      v -> gl_vbo = 0;
    }
  }
  break;
case TYPE_ZSCALED:
  if(!eval_pair_expression(mf, cx, begin_subexpr, end_subexpr, &p))
```

```
      return false;
    TRANSFORM_SCALE_Z(cx -> currentpen -> gl_matrix, p.x, p.y);
    // Curved pens need retriangulation in this case:
    if(cx -> currentpen -> referenced){
      struct pen_variable *v = cx -> currentpen -> referenced;
      if(v -> gl_vbo != 0 && a.value > 1.0 && !(v -> flags & FLAG_STRAIGHT)){
        glDeleteBuffers(1, &(v -> gl_vbo));
        v -> gl_vbo = 0;
      }
    }
    break;
  case TYPE_TRANSFORMED:
    if(!eval_transform_expression(mf, cx, begin_subexpr, end_subexpr, &t))
      return false;
    MATRIX_MULTIPLICATION(cx -> currentpen -> gl_matrix, t.value);
    // Curved pens need retriangulation in this case:
    if(cx -> currentpen -> referenced){
      struct pen_variable *v = cx -> currentpen -> referenced;
      if(v -> gl_vbo != 0 && a.value > 1.0 && !(v -> flags & FLAG_STRAIGHT)){
        glDeleteBuffers(1, &(v -> gl_vbo));
        v -> gl_vbo = 0;
      }
    }
    break;
  default:
    RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(next_token -> line), next_token);
    return false;
  }
  next_token = end_subexpr;
}
```

After getting the new current pen shape, if it is a pointer for an existing pen or not, and its linear transform represented as a matrix, we just need to triangulate the current pen if needed. For this, we need to obtain the final transform matrix: if the current pen is a pointer for another pen, then the final transform matrix is obtained multiplicating the `currentpen` matrix by the pointed pen matrix. Otherwise, the final transform matrix is just the current pen matrix, After getting the final transform matrix, we just pass it to the triangulation function:

Section: 'pickup' Command: Continue (continuation):

```
{
  float final_transform_matrix[9];
  if(cx -> currentpen -> referenced == NULL){
    memcpy(final_transform_matrix, cx -> currentpen -> gl_matrix,
           9 * sizeof(float));
    if(!triangulate_pen(mf, cx, cx -> currentpen, final_transform_matrix))
      return false;
  }
  else{
    memcpy(final_transform_matrix,
           cx -> currentpen -> referenced -> gl_matrix, 9 * sizeof(float));
    MATRIX_MULTIPLICATION(final_transform_matrix,
                          cx -> currentpen -> gl_matrix);
```

```
    if(!triangulate_pen(mf, cx, cx -> currentpen -> referenced,
                        final_transform_matrix))
      return false;
  }
}
```

11.4. Operators `bot`, `top`, `lft`, `rt`

We defined most pair operators in Subsection 8.2, but there are still 4 missing operators that we will define here. Their grammar is:

```
<Pair Primary> -> bot <Pair Primary> | top <Pair Primary> |
                  lft <Pair Primary> | rt <Pair Primary>
```

This requires 4 new tokens for such operators:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_BOT,  // Symbolic token 'bot'
TYPE_TOP,  // Symbolic token 'top'
TYPE_LFT,  // Symbolic token 'lft'
TYPE_RT,   // Symbolic token 'rt'
```

And we also need to add the token names to the list of reserved keywords:

Section: List of Keywords (continuation):

```
"bot", "top", "lft", "rt",
```

These operators shift a pair for a distance that depends on the current pen size. If we have a point (x_0, y_0), then `bot (x0, y0)` represents (x_0, y_0) shifted down, such that it is now positioned below the pen when we center it in point (x_0, y_0). Assuming that the pen coordinates are centered on the origin, then its lower points are negative in axis y. Therefore, we can do it adding the initial point with the lowest y value in the pen:

Section: Pair Primary: Other Rules to Be Defined Later (continuation):

```
else if(begin -> type == TYPE_BOT){
  if(!eval_pair_primary(mf, cx, (struct generic_token *)
                                begin -> next,
                        end, result))
    return false;
  result -> y += cx -> pen_bot;
  return true;
}
```

The operator `top` shifts up a quantity equal to the biggest coordinate y in the current pen perimeter. This means that `top (x0, y0)` will be positioned such that it will be above the pen when the pen is placed in (x_0, y_0):

Section: Pair Primary: Other Rules to Be Defined Later (continuation):

```
else if(begin -> type == TYPE_TOP){
  if(!eval_pair_primary(mf, cx, (struct generic_token *)
                                begin -> next,
                        end, result))
    return false;
  result -> y += cx -> pen_top;
  return true;
}
```

The operator `lft` shifts left a quantity equal the smallest coordinate x in the current pen perimeter. Therefore, `lft (x0, y0)` will be positioned at left of the pen if the pen is centered in (x_0, y_0). As in a pen centered in the origin the leftmost points will have a negative x coordinate, we just need to add to the x coordinate of the point the leftmost x coordinate in the pen:

Section: Pair Primary: Other Rules to Be Defined Later (continuation):

```
else if(begin -> type == TYPE_LFT){
  if(!eval_pair_primary(mf, cx, (struct generic_token *)
                                begin -> next,
                        end, result))
    return false;
  result -> x += cx -> pen_lft;
  return true;
}
```

The last operator, `rt`, shifts right the pair (x_0, y_0) a quantity equivalent to the rightmost coordinate in the pen perimeter, such that the point coordinate will be placed at right of the pen when centered in (x_0, y_0):

Section: Pair Primary: Other Rules to Be Defined Later (continuation):

```
else if(begin -> type == TYPE_RT){
  if(!eval_pair_primary(mf, cx, (struct generic_token *)
                                begin -> next,
                        end, result))
    return false;
  result -> x += cx -> pen_rt;
  return true;
}
```

12. The `pickcolor` Command

The default color used by WeaveFont is defined by the global variables `color_r`, `color_g`, `color_b`, and `color_a`. However, when we are rendering a single glyph, we can change the color used in our renderings temporarily. This is because our struct context also has its own copy of the values of these variables:

Section: Attributes (struct context) (continuation):

```
  float color[4];
```

When the context structure is crerated, these variables are initialized with the same value stored in the corresponding global internal variable:

Section: Initialization (struct context) (continuation):

```
cx -> color[0] = mf ->internal_numeric_variables[INTERNAL_NUMERIC_R].value;
cx -> color[1] = mf ->internal_numeric_variables[INTERNAL_NUMERIC_G].value;
cx -> color[2] = mf ->internal_numeric_variables[INTERNAL_NUMERIC_B].value;
cx -> color[3] = mf ->internal_numeric_variables[INTERNAL_NUMERIC_A].value;
```

To change locally in the context the color values, we can use the command `pickcolor`, whose grammar is described below:

```
<Command> -> <'draw' Command> | <'erase' Command> | ...
<'draw' Command> -> draw <Path Expression>
```

Which requires a new reserved keyword:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_PICKCOLOR,  // Symbolic token 'pickcolor'
```

Section: List of Keywords (continuation):

```
"pickcolor",
```

The interpretation of the `pickcolor` command is intuitive: each numeric expression represents a value in the closed interval between 0 and 1 (rounded to the nearest value in the interval if it is not in it) and which is assigned respectively to the red, green, blue and alpha (transparency) channels.

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_PICKCOLOR){
  struct numeric_variable result;
  int i;
  DECLARE_NESTING_CONTROL();
  struct generic_token *begin_expr, *end_expr;
  begin_expr = begin -> next;
  if(begin_expr == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin_expr -> line));
    return false;
  }
  if(begin_expr -> type != TYPE_OPEN_PARENTHESIS){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(begin_expr -> line),
                               TYPE_OPEN_PARENTHESIS, begin_expr);
    return false;
  }
  begin_expr = begin_expr -> next;
  for(i = 0; i < 4; i ++){
    end_expr = begin_expr;
    if(end_expr == NULL || end_expr -> type == TYPE_COMMA ||
       end_expr -> type == TYPE_CLOSE_PARENTHESIS){
      RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(end_expr -> line),
                                     TYPE_T_NUMERIC);
      return false;
    }
    while(end_expr != NULL){
      COUNT_NESTING(end_expr);
      if(IS_NOT_NESTED() && end_expr -> next != NULL &&
         ((i < 3 && end_expr -> next -> type == TYPE_COMMA) ||
          (i == 3 && end_expr -> next -> type == TYPE_CLOSE_PARENTHESIS)))
        break;
      end_expr = end_expr -> next;
    }
    if(end_expr == NULL){
      RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                     TYPE_T_NUMERIC);
      return false;
    }
    if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &result))
      return false;
    cx -> color[i] = result.value;
    begin_expr = end_expr -> next -> next;
  }
```

```
  return true;
}
```

13. The monowidth Command

If a typeface is set to have a monospace version, where all characters must have exactly the same width, it is useful to use the **monowidth** command to be defined here to choose what this width is. If during the rendering of a glyph the internal numeric variable **monospace** is greater than zero, then the width of the character will be calculated according to the numeric expression defined by the **monowidth** command, instead of using the particular expression of that specific glyph.

The syntax for **monowidth** command is:

```
<Command> -> ... | <'monowidth' Command> | ...
<'monowidth' Command> -> monowidth <Numeric Expression>
```

Which requires a new tokenn type and reserved keyword:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_MONOWIDTH,  // Symbolic token 'monowidth'
```

Section: List of Keywords (continuation):

```
"monowidth",
```

The numeric expression associated with the **monowidth** command is not executed at the time it is encountered. Instead, the pointer to the beginning and end of such an expression is simply stored by the following pointers in the font structure:

Section: Attributes (struct metafont) (continuation):

```
void *mono_expr_begin, *mono_expr_end;
```

The pointers are initialized as null:

Section: Inicializao (struct metafont) (continuation):

```
mf -> mono_expr_begin = mf -> mono_expr_end = NULL;
```

And interpret the command do not involve evaluating the numeric expression, but storing it in the pointer to evaluate later, when rendering glyphs in monospace mode:

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_MONOWIDTH){
  if(begin == *end){
    RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_NUMERIC);
    return false;
  }
  mf -> mono_expr_begin = begin -> next;
  mf -> mono_expr_end = *end;
  return true;
}
```

14. The draw and erase Command

Finally we can begin specifying two of the most important command in the language. The command that uses a pen to draw a path in a picture and the command that erases a path in a picture. The command syntax is:

```
<Command> -> <'draw' Command> | <'erase' Command> | ...
<'draw' Command> -> draw <Path Expression>
<'erase' Command> -> erase <Path Expression>
```

Which requires a new token type and a new reserved word:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_DRAW,  // Symbolic token 'draw'
TYPE_ERASE, // Symbolic token 'erase'
```

Section: List of Keywords (continuation):

```
"draw", "erase",
```

14.1. Preparing Framebuffer

To draw or erase with a pen in the picture `currentpicture`, we need a framebuffer associated with this picture. This is not much different than when we define several operations over picture variables. In several of them, we created a new framebuffer, stored the previous framebuffer, changed to the new framebuffer, rendered, erased the new framebuffer and changed to the old one. But for the `currentpicture` framebuffer, we will keep a single variable to store it, so that we do not need to create and destroy it all the time:

Section: Attributes (struct context) (continuation):

```
GLuint currentpicture_fb;
```

Which is initialized as zero:

Section: Initialization (struct context) (continuation):

```
cx -> currentpicture_fb = 0;
```

Before rendering to `currentpicture`, we should execute the following code to check if the framebuffer is initialized and if not, initializes it:

Section: Prepare 'currentpicture' for Drawing:

```
{
  if(cx -> currentpicture_fb == 0){
    int pic_width, pic_height;
    GLuint texture;
    pic_width = cx -> currentpicture -> width;
    pic_height = cx -> currentpicture -> height;
    texture = cx -> currentpicture -> texture;
    glGenFramebuffers(1, &(cx -> currentpicture_fb));
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, pic_width, pic_height, 0, GL_RGBA,
                 GL_UNSIGNED_BYTE, NULL);
    glBindFramebuffer(GL_FRAMEBUFFER, cx -> currentpicture_fb);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                           texture, 0);
    if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE){
      RAISE_ERROR_OPENGL_FRAMEBUFFER(mf, cx, 0);
      return false;
    }
  }
  else
    glBindFramebuffer(GL_FRAMEBUFFER, cx -> currentpicture_fb);
```

```
}
```

If for some reason we change the contents of `currentpicture`, we must remove the old framebuffer:

Section: Create new 'currentpicture':

```
{
  if(cx -> currentpicture_fb != 0){
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glBindTexture(GL_TEXTURE_2D, 0);
    glDeleteFramebuffers(1, &(cx -> currentpicture_fb));
  }
  cx -> currentpicture_fb = 0;
}
```

Before rendering to `currentpicture`, we do not store the old framebuffer. And after the rendering, we do not restore the old framebuffer. This is different than what de do when we render in other picture variabes using picture operators. We do it this way because rendering to `currentpicture` should be a much more common operation. Because of this, we will try to keep as the current framebuffer the framebuffer associated with `currentpicture`.

But before beginning to evaluate code, we store the previous framebuffer that the program was using before calling our code. It will be stored in the variable below:

Section: Local Variables (metafont.c) (continuation):

```
static GLint previous_fb;
```

And here is where we save the previous framebuffer before evaluating the code in `eval_list_of_expressions`. We also save the previous viewport, as we can change this value too when rendering a glyph:

Section: Before Evaluating Code:

```
GLint _viewport[4];
glGetIntegerv(GL_VIEWPORT, _viewport);
glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_fb);
```

And this restores the old framebuffer and viewport after code evaluation:

Section: After Evaluating Code:

```
glBindFramebuffer(GL_FRAMEBUFFER, previous_fb);
glViewport(_viewport[0], _viewport[1], _viewport[2], _viewport[3]);
```

14.2. Drawing Shaders

Drawing and erasing will require two new shaders (one for each operation) that we will define. The previous shaders were defined to perform operations between picture variables, and all them required a texture. But for our pen, instead of a texture, we will need to pass a drawing color. For now, the color will always be opaque black, but if this changes in the future, our shader code will be ready.

Our vertex shader code is presented below:

Section: Local Variables (metafont.c) (continuation):

```
static const char pen_vertex_shader[] =
  "#version 100\n"
  "attribute vec4 vertex_data;\n"
  "uniform mat3 model_view_matrix;\n"
  "void main(){\n"
  "  highp vec3 coord;\n"
  "  coord = vec3(vertex_data.xy, 1.0) * model_view_matrix;\n"
  "  gl_Position = vec4(coord.x, coord.y, 0.0, 1.0);\n"
  "}\n";
```

It is identical the previous vertex shader, except by the fact that this one does not receive nor try to extract values from a texture. The two new fragment shader is:

Section: Local Variables (metafont.c) (continuation):

```
static const char pen_erase_fragment_shader[] =
  "#version 100\n"
  "precision mediump float;\n"
  "uniform vec4 color;\n"
  "void main(){\n"
  "  gl_FragColor = vec4(1.0 - color.r, 1.0 - color.g, 1.0 - color.b, \n"
  "                      color.a);\n"
  "}\n";
static const char pen_fragment_shader[] =
  "#version 100\n"
  "precision mediump float;\n"
  "uniform vec4 color;\n"
  "void main(){\n"
  "  gl_FragColor = color;"
  "}\n";
static GLuint pen_program, pen_erase_program; // The program after compilation
static GLint pen_uniform_matrix, pen_erase_uniform_matrix; // Matrix
static GLint pen_uniform_color, pen_erase_uniform_color; // Color
```

During initialization, we compile these shaders and get the uniform locations:

Section: WeaveFont Initialization (continuation):

```
{
  pen_program = compile_shader_program(pen_vertex_shader, pen_fragment_shader);
  pen_uniform_matrix = glGetUniformLocation(pen_program, "model_view_matrix");
  pen_uniform_color = glGetUniformLocation(pen_program, "color");
  pen_erase_program = compile_shader_program(pen_vertex_shader,
                                             pen_erase_fragment_shader);
  pen_erase_uniform_matrix = glGetUniformLocation(pen_erase_program,
                                                  "model_view_matrix");
  pen_erase_uniform_color = glGetUniformLocation(pen_erase_program, "color");
}
```

And in the finalization we destroy these programs:

Section: WeaveFont Finalization (continuation):

```
glDeleteProgram(pen_program);
glDeleteProgram(pen_erase_program);
```

14.3. Drawing Paths

After finishing preparing the triangulation, the framebuffer and shaders, we finally can write code for drawing.

Section: Statement: Command:

```
else if(begin -> type == TYPE_DRAW){
  struct path_variable path;
  // Avaliar a expresso de caminho
  if(!eval_path_expression(mf, cx, begin -> next, *end, &path))
    return false;
  if(!drawing_commands(mf, cx, &path, 0))
```

```
    return false;
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &path, false);
  return true;
}
else if(begin -> type == TYPE_ERASE){
  struct path_variable path;
  // Avaliar a expresso de caminho
  if(!eval_path_expression(mf, cx, begin -> next, *end, &path))
    return false;
  if(!drawing_commands(mf, cx, &path, 1))
    return false;
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &path, false);
  return true;
}
```

The above code identifies a `draw` command and extract the path to be drawn. The code that performs the drawing is:

Section: Local Function Declaration (metafont.c) (continuation):

```
bool drawing_commands(struct metafont *mf, struct context *cx,
                      struct path_variable *path, unsigned int flags);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
#define ERASE_FLAG 1
bool drawing_commands(struct metafont *mf, struct context *cx,
                      struct path_variable *path, unsigned int flags){
  int i, j;
  float transform_matrix[9];
  struct pen_variable *currentpen = cx -> currentpen;
  struct picture_variable *currentpicture = cx -> currentpicture;
  // Prepare the pen, currentpicture and OpenGL parameters
  if(currentpen -> referenced != NULL){
    memcpy(transform_matrix, currentpen -> referenced -> gl_matrix,
           9 * sizeof(float));
    MATRIX_MULTIPLICATION(transform_matrix, currentpen -> gl_matrix);
    currentpen = currentpen -> referenced;
  }
  else
    memcpy(transform_matrix, currentpen -> gl_matrix, 9 * sizeof(float));
            <Section to be inserted: Prepare 'currentpicture' for Drawing>
  // Drawing loop
  for(i = 0; i < path -> length - 1; i ++){
    int distance = 0;
    float dx, dy, dt;
    dx = path -> points[i].point.u_x - path -> points[i].point.x;
    dy = path -> points[i].point.u_y - path -> points[i].point.y;
    distance += (int) ceil(sqrt(dx * dx + dy * dy));
    dx = path -> points[i].point.v_x - path -> points[i].point.u_x;
    dy = path -> points[i].point.v_y - path -> points[i].point.u_y;
    distance += (int) ceil(sqrt(dx * dx + dy * dy));
```

```
    dx = path -> points[(i+1) % (path -> length)].point.x - path -> points[i].point.v_x;
    dy = path -> points[(i+1) % (path -> length)].point.y - path -> points[i].point.v_y;
    distance += (int) ceil(sqrt(dx * dx + dy * dy));
    distance *= 1.4; // Multiplier obtained by tests to avoid leaps in the drawing
    dt = 1 / ((float) distance);
    for(j = 0; j <= distance; j ++){
      float t = dt * j;
      float x = (1-t)*(1-t)*(1-t) * path -> points[i].point.x +
                3*(1-t)*(1-t)*t * path -> points[i].point.u_x +
                3*(1-t)*t*t * path -> points[i].point.v_x +
                t*t*t * path -> points[(i + 1) % (path -> length)].point.x;
      float y = (1-t)*(1-t)*(1-t) * path -> points[i].point.y +
                3*(1-t)*(1-t)*t * path -> points[i].point.u_y +
                3*(1-t)*t*t * path -> points[i].point.v_y +
                t*t*t * path -> points[(i + 1) % (path -> length)].point.y;
      drawpoint(cx, currentpen, currentpicture, x, y, transform_matrix,
                flags & ERASE_FLAG);
    }
  }
  // When the path has a single point and the loop above do not run:
  if(path -> length == 1)
    drawpoint(cx, currentpen, currentpicture, path -> points[0].point.x,
              path -> points[0].point.y, transform_matrix, flags & ERASE_FLAG);
  return true;
}
```

The above code first gets the relevant variables (`currentpen`, `currentpicture`).

Then we get the correct pen that we should use, triangulate it and we initialize OpenGL parameters for this kind of draw, some of them we defined in previous Subsections. One of the two single places where we differentiate between commands `draw` and `erase` appears here. Depending on the command, we change the OpenGL blending function. The blending follows the same logic than we used when adding and subtracting images.

After this, we have the drawing loop, where the path is drawn, point by point using the function `drawpoint` that we still did not define. To get each individual point, we use the Bezier curve formula $z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2' + 3(1-t)t^2 z_3' + t^3 z_4$. And to know how many intermediate points we should draw between two extremity points, we add the distance between all four points that compose the curve: the two extremity points and two control points.

Now we just need to define `drawpoint`. Its header is:

Section: Local Function Declaration (metafont.c) (continuation):

```
void drawpoint(struct context *cx,
               struct pen_variable *pen, struct picture_variable *pic,
               float x, float y, float *matrix, bool erasing);
```

The pen knows all the coordinates of its triangulated vertices, and stores them in pixels. The picture stores its own width and height in pixels. But to render using OpenGL, we should convert this to OpenGL coordinates, where the destiny texture will always have width and height equal 2. This means that we need to create a new matrix to perform this last linear transformation.

Let h be the picture height in pixels and w its width. This means that 1 horizontal pixel is $2/w$ and 1 vertical pixel is $2/h$. Then, we just need to multiply the transform matrix stored in the pen by a diagonal matrix with $2/w$ and $2/h$ as the two first elements in the diagonal (the other diagonals will be 1).

Next, after performing the conversion for OpenGL coordinates, we should not use pixels anymore. But we need to shif each vertex by (x, y), the coordinate passed as argument for the `drawpoint` function. Their values are $2x/w$ and $2y/h$ in OpenGL coordinates. But we also need another shifting to make a correction

because we store the coordinates assuming that the origin is the lower left corner of the image, while OpenGL assumes that the origin is the center of the image.

However, in fact, we have another detail: it is not entirely true that we always assume that our origin is the left lower corner of the image. We could also assume that this origin is shifted vertically using an internal value d stored here:

Section: Attributes (struct context) (continuation):

```
int current_depth;
```

Its initial value is zero:

Section: Initialization (struct context) (continuation):

```
cx -> current_depth = 0;
```

This value d stores in pixels how much our axis x is shifted vertically. Therefore, in OpenGL coordinates, we should convert it to $2d/h$. If it is zero, then the origin indeed will be the lower left corner of the picture before converting to OpenGL origin. Otherwise, we shift the origin according with d value.

The multiplication that converts the pen transform matrix to OpenGL coordinates is given below:

$$\begin{bmatrix} a & b & 0 & 0 \\ c & d & 0 & 0 \\ 0 & 0 & 1 & 0 \\ e & f & 0 & 1 \end{bmatrix} \begin{bmatrix} 2/w & 0 & 0 & 0 \\ 0 & 2/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2x/w-1 & 2d/h+2y/w-1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2a/w & 2b/h & 0 & 0 \\ 2c/w & 2d/h & 0 & 0 \\ 0 & 0 & 1 & 0 \\ (2e+2x)/w-1 & (2f+2d+2y)/h-1 & 0 & 1 \end{bmatrix}$$

All this is done by the function `drawpoint` before drawing. After getting the correct matrix, we can render the result, using different shader programs if we are drawing or erasing:

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void drawpoint(struct context *cx,
               struct pen_variable *pen, struct picture_variable *pic,
               float x, float y, float *matrix, bool erasing){
  float gl_matrix[9];
  gl_matrix[0] = (2 * matrix[0]) /  pic -> width;  // 2a/w
  gl_matrix[1] = (2 * matrix[1]) /  pic -> height; // 2b/h
  gl_matrix[2] = 0.0;
  gl_matrix[3] = (2 * matrix[3]) /  pic -> width;  // 2c/w
  gl_matrix[4] = (2 * matrix[4]) /  pic -> height; // 2d/h
  gl_matrix[5] = 0.0;
  gl_matrix[6] = 2 * (matrix[6] + x) /  pic -> width - 1.0;
  gl_matrix[7] = 2 * (matrix[7] + cx -> current_depth + y) / pic -> height -
                 1.0;
  gl_matrix[8] = 1.0;
  glViewport(0, 0, pic -> width, pic -> height);
  // Se a caneta for quadrada, usamos a triangulao padro de quadrado.
  // Se no for, usamos a triangulao da prpria caneta.
  if(pen -> flags & FLAG_SQUARE)
    glBindBuffer(GL_ARRAY_BUFFER, pensquare_vbo);
  else
    glBindBuffer(GL_ARRAY_BUFFER, pen -> gl_vbo);
  glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 0, (void *) 0);
  if(erasing){
    glUseProgram(pen_erase_program);
    glUniformMatrix3fv(pen_erase_uniform_matrix, 1, true, gl_matrix);
    glUniform4f(pen_erase_uniform_color, 0.0, 0.0, 0.0, 1.0);
  }
  else{
```

```
    glUseProgram(pen_program);
    glUniformMatrix3fv(pen_uniform_matrix, 1, true, gl_matrix);
    glUniform4f(pen_uniform_color, cx -> color[0], cx -> color[1],
                cx -> color[2], cx -> color[3]);
  }
  glEnableVertexAttribArray(0);
  if(pen -> flags & FLAG_CONVEX)
    glDrawArrays(GL_TRIANGLE_FAN, 0, pen -> indices);
  else
    glDrawArrays(GL_TRIANGLES, 0, pen -> indices);
}
```

15. Compound Statement: Character Definition

Now we are in the part that is the main objective of the language: defining new characters for typographical fonts, or defining a new image to be used as static illustration or an animation.

The syntax for this is:

```
<Compound> -> <Character Definition>
<Character Definition> -> beginchar ( <String Token> ,
                                      <Numeric Expression> ,
                                      <Numeric Expresison> ,
                                      <Numeric Expression> )
                          <'beginchar' Body>
                          endchar
<'beginchar' Body> -> <Statement, except 'beginchar'>
<Statement, except 'beginchar'> -> <Simple> | <Compound, except 'beginchar'>
<Compound, except 'beginchar'> -> <Compound Block> | <Conditional>
```

The token `beginchar` begins a new character definition and `endchar` ends the definition. The string token is the character name. For typographical fonts, it must be the UTF-8 character that it is being represented. For animations and illustrations, the value will be ignored and can be any string. The numeric values after the string are teh character width, height and depth. The depth is its height below the baseline, for characters like "p" or "q", with parts extending below the line.

The language WeaveFont has two modes of operation: it can be loading or running. The mode changes how we deal with the code in a `beginchar` statement. If loading, the code inside the `beginchar` body will not be interpreted and executed. Instead, this code will be stored in an adequate place to be interpreted later, when we need to render the character. When running, then the language will interpret and execute the code because we need to render such character.

15.1 Unicode and UTF-8

But where should we store the code for each character when in the loading mode? For this, we will define the following structure that will hold information about a single glyph or image. It will store the code, the glyph dimensions, the OpenGL texture and an extra variable that says if we need to render the glyph, or we could just use the already rendered texture:

Section: General Declarations (metafont.h):

```
struct _glyph;
```

Section: Local Data Structures (metafont.c) (continuation):

```
struct _glyph{
  struct generic_token *begin, *end;
  int width, height, depth;
  int italic_correction;
```

```
  struct kerning *kern;
  GLuint texture;
  bool need_rendering, is_being_rendered;
};
#define INITIALIZE_GLYPH(a) {a.begin = NULL; \
                             a.end = NULL; \
                             a.width = 0; \
                             a.height = 0;\
                             a.depth = 0; \
                             a.italic_correction = 0; \
                             a.kern = NULL; \
                             a.texture = 0; \
                             a.need_rendering = true; \
                             a.is_being_rendered = false; \
                             }
```

The "kerning" present above a a linked list which stores the addictional spacing that should be placed before the next character, based on which is the next character.

Section: Local Data Structures (metafont.c) (continuation):

```
struct kerning{
  char next_char[5];
  float kern;
  struct kerning *next;
};
```

The list of all glyphs and a pointer to the first defined glyph together with its string encoding will be stored here:

Section: Attributes (struct metafont) (continuation):

```
struct _glyph *glyphs[332];
struct _glyph *first_glyph;
char first_glyph_symbol[5];
int number_of_glyphs;
```

The number 332 is enough to store all existing 327 Unicode blocks plus some other regions that currently are not in use, but could still be standardized in future years. Each block can store between 1 and thousands of different glyphs, typically, belonging to the same writting system. Initially all these blocks will be initialized as empty, but when we find new character definitions for glyphs belonging to them, we will need to allocate them:

Section: Initialization (struct metafont) (continuation):

```
memset(mf -> glyphs, 0, sizeof(struct _glyph *) * 332);
mf -> first_glyph = NULL;
memset(mf -> first_glyph_symbol, 0, 5);
mf -> number_of_glyphs = 0;
```

The function that allocates and returns the struct of a new or an existing glyph is:

Section: Local Function Declaration (metafont.c) (continuation):

```
static struct _glyph *get_glyph(struct metafont *mf, unsigned char *utf8,
                                bool create_if_not_exist);
```

The function first converts the UTF-8 representation to UTF-32, making the value identical to the Unicode code point. After this, we check a table where we store the biggest value stored in each block sequentially. We use the table to check which block doest the glyph belong, its position inside the block and

the block size. If the block does not exist and the last parameter is true, we allocate the block. In the end, we return the glyph associated with the UTF-8 character (the glyph can be unitialized). Or NULL if it was not possible to get glyph (the block does not exist and we asked to not allocate it, the UTF-8 code is invalid, the Unicode point does not exist or is not supported or we had no memory enough to allocate the block).

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
static const uint32_t greatest_point[332] = {
  // Basic Multilingual Plane  (164 blocks)
0x7f, 0xff, 0x17f, 0x24f, 0x2af, 0x2ff, 0x36f, 0x3ff, 0x4ff, 0x52f,
0x58f, 0x5ff, 0x6ff, 0x74f, 0x7ff, 0x7bf, 0x7ff, 0x83f, 0x85f, 0x86f,
0x89f, 0x8ff, 0x97f, 0x9ff, 0xa7f, 0xaff, 0xb7f, 0xbff, 0xc7f, 0xcff,
0xd7f, 0xdff, 0xe7f, 0xeff, 0xfff, 0x109f, 0x10ff, 0x11ff, 0x137f, 0x139f,
0x13ff, 0x167f, 0x169f, 0x16ff, 0x171f, 0x173f, 0x175f, 0x177f, 0x17ff, 0x18af,
0x18ff, 0x194f, 0x197f, 0x19df, 0x19ff, 0x1a1f, 0x1aaf, 0x1aff, 0x1b7f, 0x1bbf,
0x1bff, 0x1c4f, 0x1c7f, 0x1c8f, 0x1cbf, 0x1ccf, 0x1cff, 0x1d7f, 0x1dbf, 0x1dff,
0x1eff, 0x1fff, 0x206f, 0x209f, 0x20cf, 0x20ff, 0x214f, 0x218f, 0x21ff, 0x22ff,
0x23ff, 0x243f, 0x245f, 0x24ff, 0x257f, 0x259f, 0x25ff, 0x26ff, 0x27bf, 0x27ef,
0x27ff, 0x28ff, 0x297f, 0x29ff, 0x2aff, 0x2bff, 0x2c5f, 0x2c7f, 0x2cff, 0x2d2f,
0x2d7f, 0x2ddf, 0x2dff, 0x2eff, 0x2fdf, 0x2fff, 0x303f, 0x309f, 0x30ff, 0x312f,
0x318f, 0x319f, 0x31bf, 0x31ef, 0x31ff, 0x32ff, 0x33ff, 0x4dbf, 0x4dff, 0x9fff,
0xa48f, 0xa4cf, 0xa4ff, 0xa63f, 0xa69f, 0xa69f, 0xa6ff, 0xa71f, 0xa7ff, 0xa82f,
0xa83f, 0xa87f, 0xa8df, 0xa8ff, 0xa92f, 0xa95f, 0xa97f, 0xa9df, 0xa9ff, 0xaa5f,
0xaa7f, 0xaadf, 0xaaff, 0xab2f, 0xab6f, 0xabbf, 0xabff, 0xd7af, 0xd7ff, 0xdb7f,
0xdbff, 0xdfff, 0xf8ff, 0xfaff, 0xfb4f, 0xfdff, 0xfe0f, 0xfe1f, 0xfe2f, 0xfe4f,
0xfe6f, 0xfeff, 0xffef, 0xffff,
  // Supplementary Multilingual Plane (151 blocks)
0x1007f, 0x100ff, 0x1013f, 0x1018f, 0x101cf, 0x101ff, 0x1029f, 0x102df,
0x102ff, 0x1032f, 0x1034f, 0x1037f, 0x1039f, 0x103df, 0x1044f, 0x1047f,
0x104af, 0x104ff, 0x1052f, 0x1056f, 0x105bf, 0x1077f, 0x107bf, 0x1083f,
0x1085f, 0x1087f, 0x108af, 0x108ff, 0x1091f, 0x1093f, 0x1099f, 0x109ff,
0x10a5f, 0x10a7f, 0x10a9f, 0x10aff, 0x10b3f, 0x10b5f, 0x10b7f, 0x10baf,
0x10c4f, 0x10cff, 0x10d3f, 0x10e7f, 0x10ebf, 0x10eff, 0x10f2f, 0x10f6f,
0x10faf, 0x10fdf, 0x10fff, 0x1107f, 0x110cf, 0x110ff, 0x1114f, 0x1117f,
0x111df, 0x111ff, 0x1124f, 0x112af, 0x112ff, 0x1137f, 0x1147f, 0x114df,
0x115ff, 0x1166f, 0x1167f, 0x116cf, 0x1174f, 0x1184f, 0x118ff, 0x1195f,
0x119ff, 0x11a4f, 0x11aaf, 0x11abf, 0x11aff, 0x11b5f, 0x11c6f, 0x11cbf,
0x11d5f, 0x11daf, 0x11eff, 0x11f5f, 0x11fbf, 0x11fff, 0x123ff, 0x1247f,
0x1254f, 0x12fff, 0x1342f, 0x1345f, 0x1467f, 0x16a3f, 0x16a6f, 0x16acf,
0x16aff, 0x16b8f, 0x16e9f, 0x16f9f, 0x16fff, 0x187ff, 0x18aff, 0x18cff,
0x18d7f, 0x1afff, 0x1b0ff, 0x1b12f, 0x1b16f, 0x1b2ff, 0x1bc9f, 0x1bcaf,
0x1cfcf, 0x1d0ff, 0x1d1ff, 0x1d24f, 0x1d2df, 0x1d2ff, 0x1d35f, 0x1d37f,
0x1d7ff, 0x1daaf, 0x1dfff, 0x1e02f, 0x1e08f, 0x1e14f, 0x1e2bf, 0x1e2ff,
0x1e4ff, 0x1e7ff, 0x1e8df, 0x1e95f, 0x1ecbf, 0x1ed4f, 0x1eeff, 0x1f02f,
0x1f09f, 0x1f0ff, 0x1f1ff, 0x1f2ff, 0x1f5ff, 0x1f64f, 0x1f67f, 0x1f6ff,
0x1f77f, 0x1f7ff, 0x1f8ff, 0x1f9ff, 0x1fa6f, 0x1faff, 0x1fbff,
  // Unused
  0x1ffff,
  // Supplementary Ideographic Plane
  0x2a6df, 0x2b73f, 0x2b81f, 0x2ceaf, 0x2ebef, 0x2fa1f,
  // Unused
  0x2ffff,
```

```
  // Tertiary Ideographic Plane
  0x3134f, 0x323af,
  // Unused
  0xdffff,
  // Supplementary Special-purpose Plane (unused middle region)
  0xe007f, 0xe00ff, 0xe01ef,
  // Unused
  0xeffff,
  // Supplementary Private Use Area-A
  0xfffff,
  // Supplementary Private Use Area-B
  0x10ffff
};
static struct _glyph *get_glyph(struct metafont *mf, unsigned char *c,
                                bool create_if_not_exist){
  uint32_t code_point;
  int block, block_size, index;
  // UTF-8 -> UTF-32
  if(c[0] < 128)
    code_point = c[0];
  else if(c[0] >= 192 && c[0] <= 223 && c[1] >= 128 && c[1] <= 191){
    code_point = c[1] - 128;
    code_point += (c[0] - 192) * 64;
  }
  else if(c[0] >= 224 && c[0] <= 239 && c[1] >= 128 && c[1] <= 191 &&
          c[2] >= 128 && c[2] <= 191){
    code_point = c[2] - 128;
    code_point += (c[1] - 128) * 64;
    code_point += (c[0] - 224) * 4096;
  }
  else if(c[0] >= 240 && c[0] <= 247 && c[1] >= 128 && c[1] <= 191 &&
          c[2] >= 128 && c[2] <= 191 && c[3] >= 128 && c[3] <= 191){
    code_point = c[3] - 128;
    code_point += (c[2] - 128) * 64;
    code_point += (c[1] - 128) * 4096;
    code_point += (c[0] - 240) * 262144;
  }
  else return NULL; // Invalid  UTF-8 string
  if(code_point > greatest_point[331])
    return NULL; // Unicode point not existent or not supported
  for(block = 0; code_point > greatest_point[block]; block ++);
  if(block == 0){
    block_size = greatest_point[block] + 1;
    index = code_point;
  }
  else{
    block_size = greatest_point[block] - greatest_point[block - 1];
    index = code_point - greatest_point[block - 1] - 1;
  }
  if(mf -> glyphs[block] == NULL){
    int i;
```

```
    if(!create_if_not_exist)
      return NULL;
    mf -> glyphs[block] = permanent_alloc(sizeof(struct _glyph) * block_size);
    if(mf -> glyphs[block] == NULL){
      RAISE_ERROR_NO_MEMORY(mf, NULL, 0);
      return NULL;
    }
    for(i = 0; i < block_size; i ++)
      INITIALIZE_GLYPH(mf -> glyphs[block][i]);
  }
  if(mf -> glyphs[block][index].begin == NULL && !create_if_not_exist)
    return NULL; // Allocated glyph, but does not exist
  return &(mf -> glyphs[block][index]);
}
```

But if we allocate glyphs in the above function, this means that during finalization, we need to iterate over all glyphs and deallocate them and their internal data:

Section: Finalization (struct metafont) (continuation):

```
{
  int block, block_size, index;
  // Walking over all blocks:
  for(block = 0; block < 332; block ++){
    if(mf -> glyphs[block] != NULL){
      // Get the block size
      if(block == 0)
        block_size = greatest_point[block] + 1;
      else
        block_size = greatest_point[block] - greatest_point[block - 1];
      // Walking over all glyphs:
      for(index = 0; index < block_size; index ++){
        struct kerning *kern = mf -> glyphs[block][index].kern;
        if(mf -> glyphs[block][index].texture != 0)
          glDeleteTextures(1, &(mf -> glyphs[block][index].texture));
        while(kern != NULL && permanent_free != NULL){
          struct kerning *to_be_erased;
          to_be_erased = kern;
          kern = kern -> next;
          permanent_free(to_be_erased);
        }
      }
      if(permanent_free != NULL)
        permanent_free(mf -> glyphs[block]);
    }
  }
}
```

When the language finds a `beginchar` token while in the loading mode, it will read the next string token and interpret it as a UTF-8 code. Based on tis value, the above function is executed to return the correct glyph and allocate it if necessary. If a glyph for that charactere already is fully initialized, an error will be raised: this would mean that the same glyph was defined twice. Otherwise, all the code between `beginchar` and `endchar` will be stored in the glyph and this will turn the glyph in a fully initialized one (but not a renderized one). The string token used to identify the glyph will also be update to point to the

newly created glyph using its internal variables:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_BEGINCHAR && mf -> loading){
  DECLARE_NESTING_CONTROL();
  struct _glyph *glyph;
  struct generic_token *t = begin -> next;
  if(t == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  if(begin == *end){
    RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  if(t -> type != TYPE_OPEN_PARENTHESIS){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(t -> line),
                               TYPE_OPEN_PARENTHESIS, t);
    return false;
  }
  if(t != *end)
    t = t -> next;
  if(t == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  if(t -> type != TYPE_STRING){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(t -> line),
                               TYPE_STRING, t);
    return false;
  }
  {
    struct string_token *str = (struct string_token *) t;
    glyph = get_glyph(mf, (unsigned char *) str -> value, true);
    if(glyph == NULL)
      return false;
    if(mf -> first_glyph == NULL){
      mf -> first_glyph = glyph;
      memcpy(mf -> first_glyph_symbol, (unsigned char *) str -> value, 4);
    }
    if(glyph -> begin != NULL){
      RAISE_ERROR_DUPLICATE_GLYPH(mf, cx, OPTIONAL(begin -> line), str -> value);
      return false;
    }
    glyph -> begin = begin;
    str -> glyph = glyph;
  }
  {
    int number_of_commas = 0;
    struct generic_token *prev = t;
    while(t != NULL && t != *end){
```

```
      COUNT_NESTING(t);
      if(IS_NOT_NESTED()){
        if(t -> type == TYPE_COMMA && prev -> type != TYPE_COMMA)
          number_of_commas ++;
      }
      prev = t;
      t = t -> next;
      if(IS_NOT_NESTED() && t -> type == TYPE_CLOSE_PARENTHESIS)
        break;
      if(t -> type == TYPE_SEMICOLON || t -> type == TYPE_ENDCHAR)
        break;
    }
    if(t == NULL){
      RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(prev -> line));
      return false;
    }
    if(t -> type != TYPE_CLOSE_PARENTHESIS){
      RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(t -> line),
                                 TYPE_CLOSE_PARENTHESIS, t);
      return false;
    }
    if(number_of_commas != 3){
      RAISE_ERROR_WRONG_NUMBER_OF_PARAMETERS(mf, cx, OPTIONAL(t -> line),
                                             TYPE_BEGINCHAR, 4,
                                             number_of_commas + 1);
      return false;
    }
    t = ((struct linked_token *) begin) -> link;
    glyph -> end = t;
    *end = t;
  }
  mf -> number_of_glyphs ++;
  return true;
}
```

If we are in loading mode, and we read a `endchar` token, this is always an error: it means that we have an `endchar` without a previous `beginchar`. Tokens that end character definition are treated in the same loop than `beginchar` when in loading mode:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_ENDCHAR && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

After, when not in loading mode, we will read all this code between `beginchar` and `endchar` again. And it will be important to know and memorize which glyph we are rendering. So we will create a variable to store this information in the context struct:

Section: Attributes (struct context) (continuation):

```
struct _glyph *current_glyph;
```

Finally, let's write code to handle `beginchar` when in running mode. First we read the string token in the statement. As we already executed this code in the loading mode, this string token points to some

corresponding glyph, which was already allocated and initialized. We just need to render it.

The next step will be getting the numeric values in the `beginchar` header, where we get information about the width, height and depth and use these values to reinitialize `currentpicture`. We will use the read values multiplied by 2, rendering the glyph with twice the size that we will use in the screen. This way, we get more precise results, rendering in the screen each pixel as a result multisampled from 4 other pixels. But we will not use this technique if the macro `W_WeaveFont_DISABLE_MULTISAMPLE` is defined. Finally, the current pen is modified to be equal a `nullpen`.

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_BEGINCHAR){
  DECLARE_NESTING_CONTROL();
  struct generic_token *t, *begin_expr, *end_expr;
  struct string_token *str;
  struct numeric_variable width, height, depth;
  begin_nesting_level(mf, cx, begin);
  // First we get the current glyph
  t = begin -> next;
  t = t -> next;
  str = (struct string_token *) t;
  if(str -> type != TYPE_STRING){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(begin -> line), TYPE_STRING,
                               (struct generic_token *) str);
    return false;
  }
  cx -> current_glyph = str -> glyph;
```

<Section to be inserted: **beginchar: Restart 'Kerning'**>

```
  memset(cx -> current_character, 0, 5);
  memcpy(cx -> current_character, str -> value, 4);
  // Reading the values in the header
  t = t -> next;
  t = t -> next;
  begin_expr = t;
  do{
    COUNT_NESTING(t);
    end_expr = t;
    t = t -> next;
  } while(!IS_NOT_NESTED() || t -> type != TYPE_COMMA);
  // Getting width: either monospace or normal:
  if(mf -> internal_numeric_variables[INTERNAL_NUMERIC_MONO].value > 0.0 &&
     mf -> mono_expr_begin != NULL){
    if(!eval_numeric_expression(mf, cx, mf -> mono_expr_begin, mf -> mono_expr_end,
                                &width))
      return false;
  }
  else if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &width))
    return false;
  t = t -> next;
  begin_expr = t;
  do{
    COUNT_NESTING(t);
    end_expr = t;
```

```
    t = t -> next;
  } while(!IS_NOT_NESTED() || t -> type != TYPE_COMMA);
  if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &height))
    return false;
  t = t -> next;
  begin_expr = t;
  do{
    COUNT_NESTING(t);
    end_expr = t;
    t = t -> next;
  } while(!IS_NOT_NESTED() || t -> type != TYPE_CLOSE_PARENTHESIS);
  if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &depth))
    return false;
  if(height.value + depth.value <= 0.0 || width.value <= 0){
    RAISE_ERROR_INVALID_DIMENSION_GLYPH(mf, cx, OPTIONAL(begin -> line),
                                        width.value,
                                        height.value + depth.value);
    return false;
  }
  *end = t;
  { // Initializing currentpicture = nullpicture(width, height + depth):
    unsigned char *data;
    struct numeric_variable *vars;
    size_t size;
    struct picture_variable *pic = cx -> currentpicture;
     if(pic -> texture != 0)
       glDeleteTextures(1, &(pic -> texture));
     vars = ((struct numeric_variable *) cx -> internal_numeric_variables);
#ifdef W_WeaveFont_DISABLE_MULTISAMPLE
    cx -> current_depth = round(depth.value);
    pic -> width = round(width.value);
    pic -> height = (round(height.value) + round(depth.value));
    vars[INTERNAL_NUMERIC_W].value = round(width.value);
    vars[INTERNAL_NUMERIC_H].value = round(height.value);
    vars[INTERNAL_NUMERIC_D].value = round(depth.value);
#else
    cx -> current_depth = 2 * round(depth.value);
    pic -> width = 2 * round(width.value);
    pic -> height = 2 * (round(height.value) + round(depth.value));
    vars[INTERNAL_NUMERIC_W].value = 2 * round(width.value);
    vars[INTERNAL_NUMERIC_H].value = 2 * round(height.value);
    vars[INTERNAL_NUMERIC_D].value = 2 * round(depth.value);
#endif
    size = pic -> width * pic -> height * 4;
    data = temporary_alloc(size);
    if(data == NULL){
      RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
      return false;
    }
    // Making a new white texture
    memset(data, 255, size);
```

```
    { // And turning it transparent
      size_t i;
      for(i = 3; i < size; i += 4)
        data[i] = 0;
    }
    glGenTextures(1, &(pic -> texture));
    glBindTexture(GL_TEXTURE_2D, pic -> texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, pic -> width, pic -> height, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, data);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glBindTexture(GL_TEXTURE_2D, 0);
    if(temporary_free != NULL)
      temporary_free(data);
                  <Section to be inserted: Create new 'currentpicture'>
  }
  return true;
}
```

Finally, when we read an `endchar` token not in the loading mode, this means that we finished to render a new glyph. Then we copy the rendered picture from `currentpicture` to the new glyph and get the new glyph dimensions reading the `currentpicture` size and also the current depth stored. If we are using multisampling, we adjust the final glyph size to half the stored value, so that it will always be rendered with half the actual texture size. Finally, we mark the glyph as already rendered and erase all `currentpicture` content:

Section: Statement: Compound (continuation):

```
else if(begin -> type == TYPE_ENDCHAR){
  struct picture_variable *currentpicture = cx -> currentpicture;
  if(!end_nesting_level(mf, cx, begin))
    return false;
  cx -> current_glyph -> texture = currentpicture -> texture;
  cx -> current_glyph -> width = round(currentpicture -> width);
  cx -> current_glyph -> depth = round(cx -> current_depth);
  cx -> current_glyph -> height = round(currentpicture -> height -
                                        cx -> current_depth);
  cx -> current_glyph -> texture = currentpicture -> texture;
#ifndef W_WeaveFont_DISABLE_MULTISAMPLE
  cx -> current_glyph -> width /= 2;
  cx -> current_glyph -> depth /= 2;
  cx -> current_glyph -> height /= 2;
#endif
  cx -> current_glyph -> need_rendering = false;
  currentpicture -> width = -1;
  currentpicture -> height = -1;
  currentpicture -> texture = 0;
  *end = begin;
  return true;
}
```

16. API Functions to Use the Fonts

A WeaveFont code can have its global variables changed and this will change the rendering of the next glyphs, without needing to change the source code or reload the code. To read and update numeric global variables, we will export the following functions:

Section: Function Declaration (metafont.h) (continuation):

```
bool _Wwrite_numeric_variable(struct metafont *mf, char *name, float value);
float _Wread_numeric_variable(struct metafont *mf, char *name);
```

This function will change the given global or internal variable assigning the given value to it and also will mark all glyphs as needing to be rendered again. If a numeric global variable with such name does not exist, the function returns false. Otherwise, it returns true:

Section: API Functions Definition (metafont.c) (continuation):

```
bool _Wwrite_numeric_variable(struct metafont *mf, char *name, float value){
  int i;
  bool updated = false;
  char *internal_vars[] = {"pt", "cm", "mm", "color_r", "color_g", "color_b",
                           "color_a", "monospace", NULL};
  for(i = 0; internal_vars[i] != NULL; i ++)
  if(!strcmp(name, internal_vars[i])){
    mf -> internal_numeric_variables[i].value = value;
    updated = true;
  }
  if(!updated){
    struct named_variable *var = (struct named_variable *) mf -> named_variables;
    while(var != NULL){
      if(!strcmp(name, var -> name)){
        struct numeric_variable *n = (struct numeric_variable *) var -> var;
        if(n -> type != TYPE_T_NUMERIC)
          return false;
        n -> value = value;
        updated = true;
      }
      var = var -> next;
    }
  }
  if(updated){
    int j;
    for(i = 0; i < 332; i ++){
      struct _glyph *g = mf -> glyphs[i];
      int size = ((i == 0)?(greatest_point[0] + 1):
                           (greatest_point[i] - greatest_point[i - 1]));
      if(g != NULL){
        for(j = 0; j < size; j ++)
          g[j].need_rendering = true;
      }
    }
    return true;
  }
  return false;
}
```

The code to read a gloabal numeric variable is similar. If such variable does not exist, we return NAN:

Section: API Functions Definition (metafont.c) (continuation):

```
float _Wread_numeric_variable(struct metafont *mf, char *name){
  struct named_variable *var = (struct named_variable *) mf -> named_variables;
  while(var != NULL){
    if(!strcmp(name, var -> name)){
      struct numeric_variable *n = (struct numeric_variable *) var -> var;
      if(n -> type != TYPE_T_NUMERIC)
        return NAN;
      return n -> value;
    }
    var = var -> next;
  }
  return NAN;
}
```

The function that reads a file with WeaveFont code and creates a new metafont is:

Section: API Functions Definition (metafont.c) (continuation):

```
struct metafont *_Wnew_metafont(char *filename){
  struct metafont *mf;
  struct context *cx;
  struct generic_token *first, *last;
  bool ret;
  mf = init_metafont(filename);
  lexer(mf, filename, &first, &last);
  cx = init_context(mf);
  ret = eval_program(mf, cx, first, last);
  destroy_context(cx);
  if(!ret){
    _Wdestroy_metafont(mf);
    return NULL;
  }
  return mf;
}
```

If we want to render a character and get information about its dimensions in pixels (width, height, depth, italic correction and kerning), we can execute the following function:

Section: Function Declaration (metafont.h) (continuation):

```
bool _Wrender_glyph(struct metafont *mf, char *glyph,
                    char *next_glyph, GLuint *texture,
                    int *width, int *height, int *depth,
                    int *italcorr, int *kerning);
```

And the implementation:

Section: API Functions Definition (metafont.c) (continuation):

```
bool _Wrender_glyph(struct metafont *mf, char *glyph,
                    char *next_glyph, GLuint *texture,
                    int *width, int *height, int *depth,
                    int *italcorr, int *kerning){
  struct _glyph *current;
  struct kerning *k;
```

```
  struct context *cx = NULL;
  MUTEX_WAIT(mf -> mutex);
  current = get_glyph(mf, (unsigned char *) glyph, false);
  if(current == NULL || current -> begin == NULL || current -> end == NULL){
    MUTEX_SIGNAL(mf -> mutex);
    return false;
  }
  if(current -> need_rendering){
    current -> is_being_rendered = true;
    cx = init_context(mf);
    if(cx  == NULL){
      MUTEX_SIGNAL(mf -> mutex);
      return false;
    }
    if(!eval_list_of_statements(mf, cx, current -> begin, current -> end)){
      destroy_context(cx);
      MUTEX_SIGNAL(mf -> mutex);
      return false;
    }
    current -> is_being_rendered = false;
  }
  *texture = current -> texture;
  *width = current -> width;
  *height = current -> height;
  *depth = current -> depth;
  *italcorr = current -> italic_correction;
  k = current -> kern;
  *kerning = 0;
  while(k != NULL && next_glyph != NULL){
    if(!strcmp(k -> next_char, next_glyph)){
      *kerning = k -> kern;
      break;
    }
    k = k -> next;
  }
  if(cx != NULL)
    destroy_context(cx);
  MUTEX_SIGNAL(mf -> mutex);
  return true;
}
```

17. The `shipit` Command

In the original METAFONT language, in order for any character to be rendered, it was necessary to use the macro `shipit` or the command `shipout` passing an image as a parameter. In the WeaveFont language, this is not necessary, since we have the compound command `beginchar` recognized as a primitive of the language. In the original METAFONT, `beginchar` was not a primitive, but rather a macro, and this prevented rendering from occurring automatically.

But unlike METAFONT, WeaveFont renders in real time. Because of this, there may be differences in how we want to treat the rendered image. Typically, when we render a character, we want to cache its image so that we can use it multiple times without having to reinterpret its code. Rendering text would be much slower if we had to reinterpret WeaveFont code for each new letter to produce the drawing.

But there are times when we want to render something temporarily. For example, if we have produced a typeface in which each letter is unique, its parameters are randomly modified each time we produce a new character. This way, even if the same character appears several times, it will be slightly different each time it appears.

We can also use WeaveFont to create vector animations. One way to do this is to render the same character multiple times. If it is designed to be an animation, each run will correspond to a new frame.

What determines whether a glyph needs to be rendered or not is its `need_rendering` attribute. This attribute is always initialized to true and also becomes true if a global variable changes. It becomes false only if we encounter an `endchar` character. Let's then use another command other than `endchar` that will also end the rendering, but it will not set the `need_rendering` attribute to false. Other than that, it will behave exactly the same as a `endchar`, except that it can be nested inside other compound commands such as a `if`. We will choose the `shipit` command for this, which has the following grammar:

```
<Command> -> ... |<'shipit' Command> | ...
<'shipit' Command> -> shipit
```

Let's add `shipit` to our list of tokens and keywords:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_SHIPIT,  // Symbolic token 'shipit'
```

Section: List of Keywords (continuation):

```
"shipit",
```

If we find this command in loading mode, this means that it is outside a `beginchar` command and this is an error:

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_SHIPIT && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

When we find such command when not in loading mode, we act exactly as if we found a correct `endchar`, except that we do not change the flag `need_rendering`. We also set the ennd token to point to the correspondent `endchar` token, which we get from the `beginchar` currently active:

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_SHIPIT){
  struct picture_variable *currentpicture;
  if(begin -> next == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  if(begin -> next -> type != TYPE_SEMICOLON){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(begin -> line), TYPE_SEMICOLON,
                               begin -> next);
    return false;
  }
  currentpicture = cx -> currentpicture;
  cx -> current_glyph -> texture = currentpicture -> texture;
  cx -> current_glyph -> width = round(currentpicture -> width);
  cx -> current_glyph -> depth = round(cx -> current_depth);
  cx -> current_glyph -> height = round(currentpicture -> height -
                                        cx -> current_depth);
```

```
  cx -> current_glyph -> texture = currentpicture -> texture;
#ifndef W_WeaveFont_DISABLE_MULTISAMPLE
  cx -> current_glyph -> width /= 2;
  cx -> current_glyph -> depth /= 2;
  cx -> current_glyph -> height /= 2;
#endif
  currentpicture -> width = -1;
  currentpicture -> height = -1;
  currentpicture -> texture = 0;
  { //  Return nesting level as before last 'beginchar':
    struct linked_token *aux = cx -> end_token_stack;
    while(aux != NULL && aux -> type != TYPE_ENDCHAR){
      if(aux -> type == TYPE_ENDFOR){ // Running loops must be stopped:
        struct begin_loop_token *loop = (struct begin_loop_token *) aux -> link;
        loop -> running = false;
      }
      end_nesting_level(mf, cx, (struct generic_token *) aux);
      aux = cx -> end_token_stack;
    }
    *end = ((struct linked_token *) (aux -> link)) -> link;
    end_nesting_level(mf, cx, (struct generic_token *) aux);
  }
  return true;
}
```

Something relevant about the `shipit` command is that it becomes the programmer's responsibility to erase the texture produced by it with `glDeleteTextures`. Either the programmer must know if the produced glyph internally used `shipit`, or she must checks if the glyph whose texture was produced is still marked as needing to be rendered or not.

18. The `renderchar` Command

Sometimes a glyph is composed by other glyph. For example, the glyph "$" can be seen as the glyph "S" with an additional vertical dash. In some writting systems, such compositions are even more common. For example, in chinese logograms, we can find characters that are radicals that create other characters. And the result also can be part of other more complex character. For example, the logograms below are: "rn" (person), "d" (big), "tiān" (sky), "W" (a common surname), and "y" (pleasure):

This is not adequate in all cases, but sometimes a character can be directly extended by another character, which makes easier and faster to create typographical fonts with several characters. For this, we offer the command `renderchar`, whose grammar is described below:

```
<Command> -> ... |<'renderchar' Command> | ...
<'renderchar' Command> -> renderchar <String> between <Pair Expression>
                          and <Pair Expression>
```

Which requires support for the `between` token and keyword:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_RENDERCHAR, // Symbolic token 'renderchar'
```

```
TYPE_BETWEEN,    // Symbolic token 'between'
```

Section: List of Keywords (continuation):

```
"renderchar", "between",
```

If we find this command in the loading mode, this is an error. In loading mode we still do not know which characters our font defines:

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_RENDERCHAR && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

If we are not in loading mode, this command will: (1) get the asked glyph, rendering it if still not rendered, and (2) will draw the obtained glyph, using the two pair expressions to delimit the area where we will draw in our current picture:

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_RENDERCHAR){
  struct _glyph *glyph;
  struct string_token *str;
  struct pair_variable p1, p2;
                  <Section to be inserted: Renderchar: Get Glyph>
                 <Section to be inserted: Renderchar: Render Glyph>
  return true;
}
```

First we need to check that there is a string after `renderchar`. If not, this is an error. If it exists, otherwise, we use it to get the needed glyph:

Section: Renderchar: Get Glyph:

```
{
  str = (struct string_token *) (begin -> next);
  if(str == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  if(str -> type != TYPE_STRING){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(begin -> line), TYPE_STRING,
                               (struct generic_token *) str);
    return false;
  }
  if(str -> glyph != NULL)
    glyph = str -> glyph;
  else{
    glyph = get_glyph(mf, (unsigned char *) (str -> value), false);
    if(glyph == NULL){
      RAISE_ERROR_UNKNOWN_GLYPH_DEPENDENCY(mf, cx, OPTIONAL(str -> line),
                                           str -> value);
      return false;
    }
    str -> glyph = glyph;
  }
```

```
}
```

It may be that the rendering of a first character depends on a second, which in turn depends on a third and fourth. This is not a problem. What we cannot allow to happen is that the rendering of a character depends on its own rendering. Recursion should not be allowed in `renderchar`. We detect this in the code below:

Section: Renderchar: Get Glyph (continuation):

```
if(glyph -> is_being_rendered){
  RAISE_ERROR_RECURSIVE_RENDERCHAR(mf, cx, OPTIONAL(str -> line),
                                   str -> value);
  return false;
}
```

Finally, if the character invoked by `renderchar` has not yet been rendered, we render it:

Section: Renderchar: Get Glyph (continuation):

```
if(glyph -> need_rendering){
  struct context *new_cx;
  glyph -> is_being_rendered = true;
  new_cx = init_context(mf);
  if(new_cx == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(str -> line));
    return false;
  }
  if(!eval_list_of_statements(mf, new_cx, glyph -> begin, glyph -> end)){
    destroy_context(new_cx);
    return false;
  }
  glyph -> is_being_rendered = false;
  destroy_context(new_cx);
}
```

Now to render the glyph, we must identify where exactly in the texture of the current image we should place it. To do this, we must obtain the pair expressions: the first one delimited by `between` and `and` and the second one delimited between this same `and` and the end of expression:

Section: Renderchar: Render Glyph:

```
{
  DECLARE_NESTING_CONTROL();
  struct generic_token *begin_expr, *end_expr;
  begin_expr = str -> next;
  if(begin_expr == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(str -> line));
    return false;
  }
  if(begin_expr -> type != TYPE_BETWEEN){
    RAISE_ERROR_EXPECTED_FOUND(mf, NULL, OPTIONAL(begin -> line),
                               TYPE_BETWEEN, begin_expr);
    if(glyph -> need_rendering) // Rendered with 'shipit'
      glDeleteTextures(1, &(glyph -> texture));
    return false;
  }
  begin_expr = begin_expr -> next;
  if(begin_expr -> type == TYPE_AND || begin_expr -> next == NULL){
```

```
  RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin_expr -> line),
                                 TYPE_T_PAIR);
  if(glyph -> need_rendering) // Rendered with 'shipit'
    glDeleteTextures(1, &(glyph -> texture));
  return false;
}
end_expr = begin_expr;
while(end_expr != NULL && end_expr -> next != NULL){
  COUNT_NESTING(end_expr);
  if(IS_NOT_NESTED() && end_expr -> next -> type == TYPE_AND)
    break;
  end_expr = end_expr -> next;
}
if(end_expr == NULL || end_expr -> next == NULL){
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_expr -> line),
                                 TYPE_T_PAIR);
  if(glyph -> need_rendering) // Rendered with 'shipit'
    glDeleteTextures(1, &(glyph -> texture));
  return false;
}
if(!eval_pair_expression(mf, cx, begin_expr, end_expr, &p1)){
  if(glyph -> need_rendering) // Rendered with 'shipit'
    glDeleteTextures(1, &(glyph -> texture));
  return false;
}
begin_expr = end_expr -> next -> next;
if(begin_expr -> type == TYPE_SEMICOLON || begin_expr -> next == NULL){
  RAISE_ERROR_MISSING_EXPRESSION(mf, cx, OPTIONAL(begin_expr -> line),
                                 TYPE_T_PAIR);
  if(glyph -> need_rendering) // Rendered with 'shipit'
    glDeleteTextures(1, &(glyph -> texture));
  return false;
}
end_expr = begin_expr;
while(end_expr != NULL && end_expr -> next != NULL){
  COUNT_NESTING(end_expr);
  if(IS_NOT_NESTED() && end_expr -> next -> type == TYPE_SEMICOLON)
    break;
  end_expr = end_expr -> next;
}
if(end_expr == NULL || end_expr -> next == NULL){
  printf("FALHA AQUI\n");
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin_expr -> line),
                                 TYPE_T_PAIR);
  if(glyph -> need_rendering) // Rendered with 'shipit'
    glDeleteTextures(1, &(glyph -> texture));
  return false;
}
if(!eval_pair_expression(mf, cx, begin_expr, end_expr, &p2)){
  if(glyph -> need_rendering) // Rendered with 'shipit'
    glDeleteTextures(1, &(glyph -> texture));
```

```
    return false;
  }
}
```

If the image is not going to undergo any transformation, we expect the first pair to be the coordinates of the lower left corner from where we will render the glyph and the second pair to be the coordinates of the upper right corner. This would imply that both coordinates of the first pair must necessarily be smaller than those of the second.

If this is not true, we will assume that the image must undergo a transformation. If the x coordinate of the first pair is greater than that of the second, then the image must be mirrored horizontally. If the y coordinate of the first pair is greater, then the image will be mirrored vertically. This works by simply multiplying by -1 the relevant positions of the matrix that we will use in the rendering:

Section: Renderchar: Render Glyph (continuation):

```
{
  float gl_matrix[9];
  float width, height, depth;
  float current_width, current_height, current_depth;
  current_width = cx -> internal_numeric_variables[INTERNAL_NUMERIC_W].value;
  current_height = cx -> internal_numeric_variables[INTERNAL_NUMERIC_H].value +
                cx -> internal_numeric_variables[INTERNAL_NUMERIC_D].value;
  current_depth = cx -> internal_numeric_variables[INTERNAL_NUMERIC_D].value;
  INITIALIZE_IDENTITY_MATRIX(gl_matrix);
  if(p1.x > p2.x){
    width = p1.x - p2.x;
    gl_matrix[0] = - (width / current_width);
    gl_matrix[6] = ((p1.x + p2.x) / current_width) - 1.0;
  }
  else{
    width = p2.x - p1.x;
    gl_matrix[0] = (width / current_width);
    gl_matrix[6] = ((p1.x + p2.x) / current_width) - 1.0;
  }
  if(p1.y > p2.y){
    height = p1.y - p2.y;
    depth = glyph -> depth * (height / (glyph -> height));
    gl_matrix[4] = - (height + depth) / current_height;
    gl_matrix[7] = ((p1.y + p2.y) / 2) -
                   (glyph -> depth / 2) * (height/glyph -> height) - (current_height)/2;
    gl_matrix[7] = 2 * (gl_matrix[7] / current_height);
    gl_matrix[7] += 2 * (current_depth / current_height);
  }
  else{
    height = p2.y - p1.y;
    depth = glyph -> depth * (height / (glyph -> height));
    gl_matrix[4] = (height + depth) / current_height;
    gl_matrix[7] = ((p1.y + p2.y) / 2) -
                   (glyph -> depth / 2) * (height/glyph -> height) - (current_height)/2;
    gl_matrix[7] = 2 * (gl_matrix[7] / current_height);
    gl_matrix[7] += 2 * (current_depth / current_height);
  }
```

<Section to be inserted: Prepare 'currentpicture' for Drawing>

```
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glBlendEquation(GL_FUNC_ADD);
  glColorMask(true, true, true, true);
  glViewport(0, 0, current_width, current_height);
  glBindBuffer(GL_ARRAY_BUFFER, vbo);
  glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void *) 0);
  glEnableVertexAttribArray(0);
  glUseProgram(program);
  glUniformMatrix3fv(uniform_matrix, 1, true, gl_matrix);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, glyph -> texture);
  glUniform1i(uniform_texture, 0);
  glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
  glBindTexture(GL_TEXTURE_2D, 0);
  glDisable(GL_BLEND);
  if(glyph -> need_rendering) // Rendered with 'shipit'
    glDeleteTextures(1, &(glyph -> texture));
}
```

19. The `kerning` Command

It is often necessary to adjust the spacing between two rendered glyphs that are to be placed side by side. For example, in sequences such as "VA", the spacing of the "A" is often adjusted further to the left so that the lower left tip of this letter starts slightly before the upper right tip of the letter "V", producing a more aesthetically pleasing result.

In WeaveFont, these adjustments are specified using the `kerning` command, which has the following syntax:

```
<Command> -> ... |<'kerning' Command> | ...
<'kerning' Command> ->  kerning ( <String> , <Numeric Expression> )
```

The command requires one more reserved word:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_KERNING,  // The 'kerning' token
```

Section: List of Keywords (continuation):

```
"kerning",
```

The `kerning` command can be used only inside a glyph definition (between `beginchar` and `endchar`), otherwise an error is raised, as shown below.

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_KERNING && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

The command indicates with a numeric expression what additional spacing should be placed when the next character is given by the string passed as argument to the command. A negative numeric value means an additional negative spacing, which shifts the next character to the left. A positive numeric value shifts it to the right. The code below interprets this:

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_KERNING){
  struct generic_token *str, *begin_expr, *end_expr;
  struct numeric_variable numeric_result;
  str = begin -> next;
  if(str == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  else if(str -> type != TYPE_OPEN_PARENTHESIS){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(str -> line),
                               TYPE_OPEN_PARENTHESIS, str);
    return false;
  }
  str = str -> next;
  if(str == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  else if(str -> type != TYPE_STRING){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(str -> line),
                               TYPE_STRING, str);
    return false;
  }
  begin_expr = str -> next;
  if(begin_expr == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(str -> line));
    return false;
  }
  else if(begin_expr -> type != TYPE_COMMA){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(begin_expr -> line),
                               TYPE_COMMA, begin_expr);
    return false;
  }
  begin_expr = begin_expr -> next;
  if(begin_expr == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(str -> line));
    return false;
  }
  end_expr = begin_expr;
  while(end_expr -> next != NULL && end_expr -> next -> next != NULL &&
        end_expr -> next -> type != TYPE_SEMICOLON &&
        end_expr -> next -> next -> type != TYPE_SEMICOLON)
    end_expr = end_expr -> next;
  if(end_expr -> next == NULL || end_expr -> next -> next == NULL){
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin_expr -> line));
    return false;
  }
  if(end_expr -> next -> type == TYPE_SEMICOLON){
    RAISE_ERROR_EXPECTED_FOUND(mf, cx, OPTIONAL(begin_expr -> line),
                               TYPE_CLOSE_PARENTHESIS, end_expr -> next);
    return false;
```

```
  }
  if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &numeric_result))
    return false;
                    <Section to be inserted: Stores Kerning>
}
```

The above code checks that the command follows the grammar rules and interpret the numeric expression to get the kerning value. The code below now stores the read value in the current glyph:

Section: Stores Kerning:

```
{
  struct kerning *new_kerning, *existing_kerning;
  struct string_token *next_char = (struct string_token *) str;
  existing_kerning = cx -> current_glyph -> kern;
  while(existing_kerning != NULL){
    if(!strncmp(next_char -> value, existing_kerning -> next_char, 4)){
      existing_kerning -> kern = numeric_result.value;
      return true;
    }
    existing_kerning = existing_kerning -> next;
  }
  new_kerning = (struct kerning *) permanent_alloc(sizeof(struct kerning));
  if(new_kerning == NULL){
    RAISE_ERROR_NO_MEMORY(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  memcpy(new_kerning -> next_char, next_char -> value, 4);
  new_kerning -> next_char[4] = '\0';
  new_kerning -> kern = numeric_result.value;
  new_kerning -> next = cx -> current_glyph -> kern;
  cx -> current_glyph -> kern = new_kerning;
  return true;
}
```

Finally, after we read a `beginchar` token, but before interpreting the glyph code, we should fill with zeroes all the stored kernings in the glyph. The new rendering should produce new values for them. If not, then this means that the kerning changed between different renderings and some kerning values are not necessary anymore:

Section: beginchar: Restart 'Kerning':

```
{
  struct kerning *existing_kerning = cx -> current_glyph -> kern;
  while(existing_kerning != NULL){
    existing_kerning -> kern = 0.0;
    existing_kerning = existing_kerning -> next;
  }
}
```

20. The `debug` Command

The `debug` command can have two different behaviors, depending on whether we are in debug mode or not. If we are, then it reads the expression that is passed to it, extracts the result and prints details about it on standard output. If we are not in debug mode, the expression that follows it is simply ignored and the command does nothing.

The command syntax is:

```
<Command> -> ... |<'debug' Command> | ...
<'debug' Command> ->  debug <Expression>
```

For this, we add `debug` as a reserved keyword:

Section: WeaveFont: Symbolic Token Definition (continuation):

```
TYPE_DEBUG,  // O token simblico 'debug'
```

Section: List of Keywords (continuation):

```
"debug",
```

The `debug` command is not designed to be very efficient. It is an occasional aid to debug code that has problems, and should be removed when the debugging process is finished. Below is the structure of the command. It works by first trying to identify the type of the expression that follows it and, depending on the type, it executes the appropriate functions to interpret and obtain the result:

Section: Statement: Command (continuation):

```
else if(begin -> type == TYPE_DEBUG){
#if defined(W_DEBUG_METAFONT)
  struct generic_token *begin_expr, *end_expr;
  int type;
  if(begin == *end)
    return true;
  begin_expr = begin -> next;
  end_expr = *end;
  type = get_tertiary_expression_type(mf, cx, begin_expr, end_expr);
  switch(type){
  case TYPE_T_NUMERIC:
                  <Section to be inserted: Debug: Numeric Expression>
    break;
  case TYPE_T_PAIR:
                    <Section to be inserted: Debug: Pair Expression>
    break;
  case TYPE_T_TRANSFORM:
                 <Section to be inserted: Debug: Transform Expression>
    break;
  case TYPE_T_PATH:
                    <Section to be inserted: Debug: Path Expression>
    break;
  case TYPE_T_PEN:
                    <Section to be inserted: Debug: Pen Expression>
    break;
  case TYPE_T_PICTURE:
                  <Section to be inserted: Debug: Picture Expression>
    break;
  case TYPE_T_BOOLEAN:
                  <Section to be inserted: Debug: Boolean Expression>
    break;
  default:
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   type);
    return false;
```

```
  }
#else
  return true;
#endif
}
```

In the case of a numeric expression, this is how we interpret it and generate the debug message:

Section: Debug: Numeric Expression:

```
{
  struct numeric_variable result;
  if(!eval_numeric_expression(mf, cx, begin_expr, end_expr, &result))
    return false;
  printf("DEBUG: %s: %d: Numeric Expression: %f\n", mf -> file,
         begin_expr -> line, result.value);
  return true;
}
```

In the case of a Pair Expression:

Section: Debug: Pair Expression:

```
{
  struct pair_variable result;
  if(!eval_pair_expression(mf, cx, begin_expr, end_expr, &result))
    return false;
  printf("DEBUG: %s: %d: Pair Expression: (%f, %f)\n", mf -> file,
         begin_expr -> line, result.x, result.y);
  return true;
}
```

In the case of a Transform Expression:

Section: Debug: Transform Expression:

```
{
  struct transform_variable result;
  if(!eval_transform_expression(mf, cx, begin_expr, end_expr, &result))
    return false;
  printf("DEBUG: %s: %d: Pair Expression: "
         "(%f, %f, %f, %f, %f, %f, %f, %f, %f)\n", mf -> file,
         begin_expr -> line, result.value[0], result.value[1], result.value[2],
         result.value[3], result.value[4], result.value[5], result.value[6],
         result.value[7], result.value[8]);
  return true;
}
```

In the case of Path Expression:

Section: Debug: Path Expression:

```
{
  struct path_variable result;
  int j;
  if(!eval_path_expression(mf, cx, begin_expr, end_expr, &result))
    return false;
  printf("DEBUG: %s: %d: Path Expression: ", mf -> file, begin_expr -> line);
  for(j = 0; j < result.length; j ++){
    if(j == result.length - 1 && result.cyclic)
      printf("cycle");
```

```
    else
      printf("(%f, %f)", result.points[j].point.x, result.points[j].point.y);
    if(j < result.length - 1){
      printf(" .. controls (%f, %f) and (%f, %f) .. ",
             result.points[j].point.u_x, result.points[j].point.u_y,
             result.points[j].point.v_x, result.points[j].point.v_y);
    }
  }
  printf("\n");
  return true;
}
```

In case of Pen Expression:

Section: Debug: Pen Expression:

```
{
  struct pen_variable result;
  struct path_variable *format;
  int j;
  if(!eval_pen_expression(mf, cx, begin_expr, end_expr, &result))
    return false;
  printf("DEBUG: %s: %d: Pen Expression: ", mf -> file, begin_expr -> line);
  switch(result.flags){
  case FLAG_CIRCULAR:
    printf("pencricle");
    break;
  case FLAG_SEMICIRCULAR:
    printf("pensemicircle");
    break;
  case FLAG_SQUARE:
    printf("pensquare");
    break;
  case FLAG_NULL:
    printf("nullpen");
    break;
  default:
    printf("makepen(");
    format = result.format;
    for(j = 0; j < format -> length; j ++){
      if(j == format -> length - 1 && format -> cyclic)
        printf("cycle");
      else
        printf("(%f, %f)", format -> points[j].point.x,
                           format -> points[j].point.y);
      if(j < format -> length - 1){
        printf(" .. controls (%f, %f) and (%f, %f) .. ",
               format -> points[j].point.u_x, format -> points[j].point.u_y,
               format -> points[j].point.v_x, format -> points[j].point.v_y);
      }
    }
    printf(")");
  }
```

```
  printf(" transformed (%f, %f, %f, %f, %f, %f)\n",
         result.gl_matrix[6], result.gl_matrix[7], result.gl_matrix[0],
         result.gl_matrix[3], result.gl_matrix[1], result.gl_matrix[4]);
  return true;
}
```

So far, we have sought to create a string representation for each type of variable that is compatible with a literal, or an expression needed to create a copy of that variable. Unfortunately, this cannot be done with images, because except in the case of `nullpicture`, there are no literals or expressions capable of recreating an image. Images are typically manipulated by commands rather than expressions. Nevertheless, we have produced a representation of an expression that generates an image of the same width and height.

Section: Debug: Picture Expression:

```
{
  struct picture_variable result;
  if(!eval_picture_expression(mf, cx, begin_expr, end_expr, &result))
    return false;
  printf("DEBUG: %s: %d: Picture Expression: ", mf -> file, begin_expr -> line);
  printf("nullpicture(%d, %d)\n", result.width, result.height);
  return true;
}
```

And finally, in the case of Boolean Expression:

Section: Debug: Boolean Expression:

```
{
  struct boolean_variable result;
  if(!eval_boolean_expression(mf, cx, begin_expr, end_expr, &result))
    return false;
  printf("DEBUG: %s: %d: Boolean Expression: ", mf -> file, begin_expr -> line);
  if(result.value)
    printf("true\n");
  else
    printf("false\n");
  return true;
}
```

21. Integrating WeaveFont and Weaver Game Engine

The Weaver game engine has a module responsible for loading new interfaces, which are visual elements to be presented to the user. The idea is that the engine can automatically create visual elements from a file name where a texture can be seen. This is done by invoking the function:

```
W.new_interface("textura.mf", NULL, 100.0, 100.0, 0.0, 100.0, 100.0);
```

The above function must read the file `textura.mf`, extract the texture that is inside it and produce a new structure of type `struct user_interface` that represents something that will immediately appear on the screen.

But how does the game engine know how to interpret the file `textura.mf`? To do so, it must be initialized by passing it the pointer to a function that interprets such texture. In our case, we will assume that such texture is a WeaveFont source code.

We also must define af function with the header below:

```
void extract(void *(*permanent_alloc)(size_t),
             void (*permanent_free)(void *),
             void *(*temporary_alloc)(size_t),
```

```
                void (*temporary_free)(void *),
                void (*before_loading_interface)(void),
                void (*after_loading_interface)(void),
                char *source_filename, struct user_interface *i)
```

The first four parameters are just functions for the allocation and deallocation functions that we must use when we interpret the file. The next two functions are just pointers to functions that must be executed before and after we interpret the texture. Next comes the path of the file where the code to be interpreted is located. And finally, a pointer to the interface where we must place the texture. It is possible to define animated textures.

We define such function as:

Section: API Functions Definition (metafont.c) (continuation):

```
#if defined(WEAVER_ENGINE)
          <Section to be inserted: Functions to be Stored in User Interface>
void _Wmetafont_loading(void *(*p_alloc)(size_t),
                        void (*p_free)(void *),
                        void *(*t_alloc)(size_t),
                        void (*t_free)(void *),
                        void (*before_loading_interface)(void),
                        void (*after_loading_interface)(void),
                        char *source_filename,
                        struct user_interface *target){
  struct metafont *mf;
  float size;
  int texture_width, texture_height, texture_depth, ignore_this;
  if(p_alloc != permanent_alloc || p_free != permanent_free ||
     t_alloc != temporary_alloc || t_free != temporary_free){
    fprintf(stderr, "Weaver Interface Metafont Submodule: ERROR: Inconsistent "
                    "usage of memory allocators.\n");
    exit(1);
  }
  if(before_loading_interface != NULL)
      before_loading_interface();
  mf = _Wnew_metafont(source_filename);
  if(mf == NULL)
    return;
  if(mf -> err){
    _Wprint_metafont_error(mf);
    return;
  }
  // Computing interface height in pixels to a value in pt.
  // If supported by the texture, this defines its size.
  size = 72.0 * (target -> height) / dpi;
  _Wwrite_numeric_variable(mf, "size", size);
  // Computing width:
  size = 72.0 * (target -> width) / dpi;
  _Wwrite_numeric_variable(mf, "hsize", size);
  // The texture is the first glyph in mf -> first_glyph_symbol
  target -> _texture1 = (GLuint *) permanent_alloc(sizeof(GLuint));
  if(target -> _texture1 == NULL)
    return;
  _Wrender_glyph(mf, mf -> first_glyph_symbol, NULL, target -> _texture1,
```

```
                &texture_width, &texture_height,
                &texture_depth, &ignore_this, &ignore_this);
  if(mf -> err){
    _Wprint_metafont_error(mf);
    _Wdestroy_metafont(mf);
    return;
  }
  if(mf -> first_glyph -> need_rendering){ // Animated texture
    float frame_duration;
    target -> animate = true;
    target -> frame_duration = (unsigned *) permanent_alloc(sizeof(unsigned));
    if(target -> frame_duration == NULL){
      _Wdestroy_metafont(mf);
      return;
    }
    frame_duration = (unsigned) _Wread_numeric_variable(mf, "frame_duration");
    if(isnan(frame_duration))
      target -> frame_duration[0] = 0;
    else
      target -> frame_duration[0] = (unsigned) frame_duration;
    target -> max_repetition = -1;
  }
  target -> width = texture_width;
  target -> height = texture_height + texture_depth;
  target -> _internal_data = (void *) mf;
  target -> _free_internal_data = _interface_free_metafont;
  target -> _reload_texture = _reload_texture;
  target -> _loaded_texture = true;
  if(after_loading_interface != NULL)
    after_loading_interface();
}
#endif
```

The above function creates a new user interface from WeaveFont code, where the code will define the texture that the interface will have. A pointer to the metafont structure is also placed in the interface, as well as a pointer to the finalization and reloading function.

That function creates a new user interface from WeaveFont code which describes the texture. A pointer to the metafont struct is also stored together with a pointer to the function that disallocate it and another function that can reload the texture.

The disallocation function should receive a generic pointer and should do any work necessary to close and deallocate any resources placed in the interface. In this case, we will need to execute `_Wdestroy_metafont` on the stored metafont structure:

Section: Functions to be Stored in User Interface:

```
void _interface_free_metafont(void *data){
  _Wdestroy_metafont((struct metafont *) data);
}
```

The function that reloads textures is executed every time the user interface changes size, or every frame it is animated. What it will do is reload the texture by reinterpreting the WeaveFont code:

Section: Functions to be Stored in User Interface (continuation):

```
void _reload_texture(struct user_interface *target){
  struct metafont *mf = (struct metafont *) (target -> _internal_data);
```

```
  int texture_width, texture_height, texture_depth, ignore_this;
  // Computando a altura da interface, dada em pixels, para pontos tipogrficos.
  // Se for suportado pela textura, usamos isso para determinar seu tamanho.
  float size = 72.0 * (target -> height) / dpi;
  _Wwrite_numeric_variable(mf, "size", size);
  // Computando a largura:
  size = 72.0 * (target -> width) / dpi;
  _Wwrite_numeric_variable(mf, "hsize", size);
  glDeleteTextures(1, target -> _texture1);
  _Wrender_glyph(mf, mf -> first_glyph_symbol, NULL, target -> _texture1,
                 &texture_width, &texture_height,
                 &texture_depth, &ignore_this, &ignore_this);
  if(mf -> err){
    _Wprint_metafont_error(mf);
    _Wdestroy_metafont(mf);
    target -> _internal_data = NULL;
    return;
  }
  if(!(mf -> first_glyph -> need_rendering))
    target -> animate = false;
  target -> width = texture_width;
  target -> height = texture_height + texture_depth;
}
```

Appendix A: Handling Errors

When there are errors in a WeaveFont source code, no image or typographical meta-font will be created. However, the programmer should have means to discover what was wrong to solve the problem. For this, we will add in the struct for each meta-font additional data for error handling and diagnostic:

Section: Attributes (struct metafont) (continuation):

```
  int err, errno_line; // Error code and error line
  char errno_character[5]; // Character being rendered during error
  char errno_str[32];  // String with additional info
  int errno_int; // Integer with additional info
```

The idea is that if no error is raised, all the above variables will be zero or NULL. Therefore, during initialization, we set these values to zero:

Section: Initialization (struct metafont) (continuation):

```
mf -> err = mf -> errno_line = 0;
mf -> errno_character[0] = '\0';
memset(mf -> errno_str, 0, 32);
mf -> errno_int = 0;
```

As long as the first error is raised, the value of `errno` will change to reflect the found error nature. If we are counting the lines in our source code (which is true if the macro `W_DEBUG_METAFONT` is defined), then the line where the error happened also will be stored in `errno_line`. If more information must be passed, depending on error nature, we can use the other variables to store some additional string or an integer.

Only the first error found in a meta-font source code will be stored. If there are more errors, they will be ignored. All functions that execute WeaveFont source code return a boolean value. If no errors were found they return true; and if an error was found they return false.

The different errors will be stored here:

Section: Local Data Structures (metafont.c) (continuation):

```
enum { // Types of error
  ERROR_NO_ERROR = 0,
  // All the different types of error that will be defined:
                    <Section to be inserted: Types of Error>
  // And a last one that we hope that won't be used: unknown error
  ERROR_UNKNOWN
};
```

There are several different things that could go wrong causing errors. Our memory allocation could fail when we create a new token, context or any other auxiliary structure. Or our lexer found an unsupported character in the source code. Or we began defining a string with an opening double quote, but did not close it until the end of line.

As it is hard for an user dealing with all different errors when reading the `errno` variable and other information stored about it, a function will be exported to print in the screen a message with error dignastic:

Section: API Functions Definition (metafont.c) (continuation):

```
void _Wprint_metafont_error(struct metafont *mf){
  char line_number[8];
  // First we try to create a string with the error line number.
  // But if we have no line number, we keep the string empty:
  if(mf -> errno_line == 0)
    line_number[0] = '\0';
  else
    sprintf(line_number, "%d:", mf -> errno_line);
  switch(mf -> err){
  case ERROR_NO_ERROR:
    fprintf(stderr, "%s:%s No errors.\n", mf -> file, line_number);
    break;
                    <Section to be inserted: Print Error Message>
  default:
    fprintf(stderr, "%s:%s Unknown error.\n", mf -> file, line_number);
  }
  if(mf -> errno_character[0] != '\0')
    fprintf(stderr, " (while rendering '%s')\n", mf -> errno_character);
  else
    fprintf(stderr, "\n");
}
```

Each subsection in this appendix will list and define a different type of error, how it is raised by its corresponding macro, its error messafe and some examples of what could cause it.

Several different errors will use the macro below as part of the definition for their own macros. The macro below stores generic information important for all types of error: the error code, the line where it was found, which character (glyph) was being rendered at the moment, if any:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_GENERIC_ERROR(mf, cx, line, error_code) {\
  struct context *_cx = cx;\
  if(!mf -> err){\
    mf -> err = error_code;\
    mf -> errno_line = line;\
    if(cx != NULL && _cx -> current_character[0] != '\0')\
      memcpy(mf -> errno_character, _cx -> current_character, 5);}}
```

As not always we will have information about the line where the error happened (this information is

stored only in debug mode), this macro will help us to consider this information optional:

Section: Local Macros (metafont.c) (continuation):

```
#if defined(W_DEBUG_METAFONT)
#define OPTIONAL(x) x
#else
#define OPTIONAL(x) 0
#endif
```

When we print error diagnostics, it will also be important to have access to a function that, given a token, shows us its representation in string form. For most tokens, this is not difficult. Note that in `metafont.c`, we defined a list of reserved words identified by "`@<List of Reserved Words@>`". The order in which each reserved word was inserted in this list is the same order in which we defined the tokens. Except that the first reserved word we defined was "`begingroup`", when we had already defined some previous tokens. This means that if the token type is greater or equal than `TYPE_BEGINGROUP`, we simply subtract this value from it and thus obtain the index corresponding to its reserved word.

If we have a token id between `TYPE_FOR` and `TYPE_SEMICOLON`, then the token will not be listed in the list of keywords, but anyway it would have a single string that identifies all tokens with same id. Finally, the reamining tokens with lesser codes are variables, strings and numbers. Representing these tokens as strings requires reading their values stored in its structure.

Using these rules, we can create an auxiliary function that produces token names from the token id number, and we can use this auxiliary function to help us defining a more general rule.

To handle the simplest cases, receiving as arguments a token and a string to be filled (maximum size: 32), the functions would work like this:

Section: Local Function Declaration (metafont.c) (continuation):

```
void token_to_string(struct generic_token *tok, char *dst);
void tokenid_to_string(int token_id, char *dst);
```

Section: Auxiliary Local Functions (metafont.c) (continuation):

```
void token_to_string(struct generic_token *tok, char *dst){
          <Section to be inserted: token“to“string: More Complex Cases>
  tokenid_to_string(tok -> type, dst);
  return;
}
void tokenid_to_string(int token_id, char *dst){
  switch(token_id){
    case TYPE_NUMERIC:
      strcpy(dst, "<Numeric>");
    break;
    case TYPE_STRING:
      strcpy(dst, "<String>");
    break;
    case TYPE_SYMBOLIC:
      strcpy(dst, "<Variable>");
    break;
    case TYPE_FOR:
      strcpy(dst, "for");
    break;
    case TYPE_ENDFOR:
      strcpy(dst, "endfor");
    break;
    case TYPE_OPEN_PARENTHESIS:
```

```
      dst[0] = '('; dst[1] = '\0';
    break;
    case TYPE_CLOSE_PARENTHESIS:
      dst[0] = ')'; dst[1] = '\0';
    break;
    case TYPE_COMMA:
      dst[0] = ','; dst[1] = '\0';
    break;
    case TYPE_SEMICOLON:
      dst[0] = ';'; dst[1] = '\0';
    break;
    default:
      if(token_id >= TYPE_BEGINGROUP){
        size_t len = strlen(list_of_keywords[token_id - TYPE_BEGINGROUP]);
        memcpy(dst, list_of_keywords[token_id - TYPE_BEGINGROUP], len + 1);
        return;
      }
  }
}
```

In the case of numeric tokens, we convert them to string using the standard library function `snprintf`:

Section: token_to_string: More Complex Cases:

```
if(tok -> type == TYPE_NUMERIC){
  snprintf(dst, 32, "%g", ((struct numeric_token *) tok) -> value);
  return;
}
```

In case of strings we copy their values, delimiting it with double quotes:

Section: token_to_string: More Complex Cases (continuation):

```
if(tok -> type == TYPE_STRING){
  dst[0] = '"';
  memcpy(&dst[1], ((struct string_token *) tok) -> value, 29);
  strncat(dst, "\"", 2);
  return;
}
```

Symbolic tokens here are variable names. Dealing with them is like dealing with strings, but without delimiting with double quotes:

Section: token_to_string: More Complex Cases (continuation):

```
if(tok -> type == TYPE_SYMBOLIC){
  struct symbolic_token *symb = (struct symbolic_token *) tok;
  size_t size = strlen(symb -> value);
  if(size <= 31)
    memcpy(dst, ((struct symbolic_token *) tok) -> value, size);
  else{
    memcpy(dst, ((struct symbolic_token *) tok) -> value, 28);
    dst[28] = dst[29] = dst[30] = '.';
  }
  dst[31] = '\0';
  return;
}
```

Everything we have described above is useful for an error message to be printed by the _Wprint_metafont_error command. But what if the user does not want to print the error and wants to diagnose it manually, generating his own custom error handler? In this case, she should read this Appendix to learn how errors are handled and how their information is added. Knowing this, she can inspect on his own in `mf -> error` whether any error occurred and what type it was. She can force certain errors to be ignored so that the code can be re-executed by changing the variable that stores whether an error occurred to false. And she can write custom messages for each type of error.

For this second task, it will need to have the list of possible errors (each one will be described in a subsection below) and it must also have a list of possible tokens to identify some information stored about some errors. We will export all this information in a specific header that only needs to be inserted into a program when the user wants to create custom error handlers:

File: src/metafont_error.h:

```
#ifndef __WEAVER_METAFONT_ERROR
#define __WEAVER_METAFONT_ERROR
#ifdef __cplusplus
extern "C" {
#endif
enum { // Types of error
  ERROR_NO_ERROR = 0,
  // All the different errors that will be defined in this APpendix
                    <Section to be inserted: Types of Error>
  // And a last one that we should not use for unknown errors
  ERROR_UNKNOWN
};
enum { // Types of tokens
  TYPE_NUMERIC = 1, TYPE_STRING, TYPE_SYMBOLIC, TYPE_FOR, TYPE_ENDFOR,
  // The basic types can be seen above. All others are inserted below:
          <Section to be inserted: WeaveFont: Symbolic Token Definition>
  // And a last one that should not be used:
  TYPE_INVALID_TOKEN
};
#ifdef __cplusplus
}
#endif
#endif
```

A.1. ERROR_DISCONTINUOUS_PATH

Section: Types of Error (continuation):

```
ERROR_DISCONTINUOUS_PATH,
```

This error occurs when we try to concatenate two paths, but we try to join them through points that are too far apart. In this case, concatenation is impossible and the error is generated. We must pass four numeric values to the error generation: the coordinates of the first point and the coordinates of the second to warn the user that they are different points and cannot be merged.

We will treat the four coordinate values passed as single-precision floating-point numbers. This means that we need 16 bytes to store all of them. Since this is a very particular case of this error, and it is not so common that we need to store so many numeric values in other errors, we can store them sequentially in the buffer that we normally use to store error message strings, since it has a space of 32 bytes:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_DISCONTINUOUS_PATH(mf, cx, line, x1, y1, x2, y2) {\
  if(!mf -> err){\
```

```
    float *buffer = (float *) (mf -> errno_str);\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_DISCONTINUOUS_PATH);\
    buffer[0] = (float) (x1);\
    buffer[1] = (float) (y1);\
    buffer[2] = (float) (x2);\
    buffer[3] = (float) (y2);}}
```

The error message to warn the user:

Section: Print Error Message (continuation):

```
case ERROR_DISCONTINUOUS_PATH:
  float *buffer = (float *) (mf -> errno_str);
  fprintf(stderr,
          "%s:%s Concatenating endpoint '(%g, %g)' with endpoint '(%g, %g)':"
          " discontinuous path.",
          mf -> file, line_number, buffer[0], buffer[1], buffer[2], buffer[3]);
  break;
```

A.2. ERROR_DIVISION_BY_ZERO

Section: Types of Error (continuation):

```
ERROR_DIVISION_BY_ZERO,
```

When we find this error, we use the default procedure, without storing any other information:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_DIVISION_BY_ZERO(mf, cx, line) {\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_DIVISION_BY_ZERO);}
```

And this is our diagnostic message:

Section: Print Error Message (continuation):

```
case ERROR_DIVISION_BY_ZERO:
  fprintf(stderr, "%s:%s Division by zero.", mf -> file, line_number);
  break;
```

A.3. ERROR_DUPLICATE_GLYPH

Section: Types of Error (continuation):

```
ERROR_DUPLICATE_GLYPH,
```

This error occurs when a glyph is defined twice with the `beginchar` command. When this occurs, we must store which glyph is having a redundant definition.

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_DUPLICATE_GLYPH(mf, cx, line, glyph) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_DUPLICATE_GLYPH);\
    memcpy(mf -> errno_str, glyph, 4);\
    mf -> errno_str[4] = '\0';}}
```

And this is the error message presented to the user:

Section: Print Error Message (continuation):

```
case ERROR_DUPLICATE_GLYPH:
  fprintf(stderr,
          "%s:%s Glyph '%s' is being defined twice.",
```

```
          mf -> file, line_number, mf -> errno_str);
  break;
```

A.4. ERROR_EMPTY_DELIMITER

Section: Types of Error (continuation):

```
ERROR_EMPTY_DELIMITER,
```

This error is generated if we encounter empty delimiters. For example:

```
{}
()
[]
```

In the original METAFONT language, an empty delimiter `[]` was allowed in variable declarations, to declare that a given variable could store several different values, using numbers to represent the internal position of each value (they are associative arrays where the keys are always numbers, not necessarily integers). But WeaveFont does not support this, so any empty delimiter is treated as an error.

When generating this error, we must inform what type of empty delimiter we encountered. It can be parentheses, square brackets or curly braces:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_EMPTY_DELIMITER(mf, cx, line, delimiter) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_EMPTY_DELIMITER);\
    mf -> errno_int = delimiter;}}
```

The error message will be customized according to which delimiter was found to be empty:

Section: Print Error Message (continuation):

```
case ERROR_EMPTY_DELIMITER:
  fprintf(stderr, "%s:%s Unexpected empty delimiter '%c%c'.",
          mf -> file, line_number, mf -> errno_int,
          ((mf -> errno_int == '(')?(')'):
            ((mf -> errno_int == '[')?(']'):('}'))));
  break;
```

A.5. ERROR_EXPECTED_FOUND

Section: Types of Error (continuation):

```
ERROR_EXPECTED_FOUND,
```

This type of error occurs when we expected to find one type of token in an instruction, but ended up finding another type.

If this happens, we should store as a number the id of the expected token and as a string the token that we found instead:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_EXPECTED_FOUND(mf, cx, line, expected, found) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_EXPECTED_FOUND);\
    mf -> errno_int = expected;\
    token_to_string(found, mf -> errno_str);}}
```

With this information, we can produce the informative error message:

Section: Print Error Message (continuation):

```
case ERROR_EXPECTED_FOUND:
  char expected_name[32];
  memset(expected_name, 0, 32);
  tokenid_to_string(mf -> errno_int, expected_name);
  fprintf(stderr, "%s:%s Expected '%s' token. Found '%s' instead.",
          mf -> file, line_number, expected_name, mf -> errno_str);
  break;
```

A.6. ERROR_FAILED_OPENING_FILE

Section: Types of Error (continuation):

```
ERROR_FAILED_OPENING_FILE,
```

This error occurs when we try to read a file with source code that does not exist or cannot be read for some reason. To correctly report an error diagnostic message when we detect this type of error, we must store the name of the file we tried to open and the value of the `errno` variable:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_FAILED_OPENING_FILE(mf, cx, line, str) {\
  if(!mf -> err){\
    size_t _len = strlen(str) + 1;\
    if(_len > 31) _len = 31;\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_FAILED_OPENING_FILE);\
    mf -> errno_int = errno;\
    memcpy(mf -> errno_str, str, _len);\
    mf -> errno_str[31] = '\0';}}
```

And when printing the error message we use this information to provide a diagnosis with the help of the `strerror` function applied to the saved value of `errno`:

Section: Print Error Message:

```
case ERROR_FAILED_OPENING_FILE:
  fprintf(stderr, "%s:%s Failed opening file \"%s\": %s.", mf -> file,
          line_number, mf -> errno_str, strerror(mf -> errno_int));
  break;
```

But this means that we need to include the header with `errno` support:

Section: Local Headers (metafont.c) (continuation):

```
#include <errno.h>
```

A.7. ERROR_INCOMPLETE_SOURCE

Section: Types of Error:

```
ERROR_INCOMPLETE_SOURCE,
```

This error occurs whenever we are interpreting a simple statement, but the source code ends in the middle. It indicates that the WeaveFont program is incomplete, or the user has inadvertently added some additional characters to the end of the source code.

We use the standard procedures when dealing with this error:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, line) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INCOMPLETE_SOURCE);}}
```

And this is the error message for this error:

Section: Print Error Message:

```
case ERROR_INCOMPLETE_SOURCE:
  fprintf(stderr,
          "%s:%s Incomplete code. WeaveFont source code ended in middle of statement.",
          mf -> file, line_number);
  break;
```

A.8. ERROR_INCOMPLETE_STATEMENT

Section: Types of Error:

```
ERROR_INCOMPLETE_STATEMENT,
```

This error happens when we are reading a statement, bu we find the end of the expression (a semicolon) before the statement is properly finished. This may happen if the user wrongly insert a semicolon in the middle of some statement, or if some reserved keyword in incorrectly placed in the statement.

When we find this errror, we invoke the default error handling:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, line) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INCOMPLETE_STATEMENT);}}
```

And our error message:

Section: Print Error Message:

```
case ERROR_INCOMPLETE_STATEMENT:
  fprintf(stderr,
          "%s:%s Incomplete statement. You ended the statement with ';' before "
          "fully defining it.",
          mf -> file, line_number);
  break;
```

A.9. ERROR_INVALID_CHAR

Section: Types of Error:

```
ERROR_INVALID_CHAR,
```

The WeaveFont lexical analyzer works by following a series of rules depending on the type of character it encounters. This is what allows it to interpret `2cm` as a two-token construction equivalent to `2*cm`, and to interpret `cm2` as being formed by a single token.

For this, it classifies the different types of characters into families and types. But not all characters belong to a family and are used in WeaveFont code. For example, characters like “” have no classification and cannot be used unless they are part of a string. Therefore, they cannot be part of variable names or anything like that. WeaveFont restricts the use of non-ASCII characters in its language. The only characters that can be used outside of strings and comments are:

```
.%0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
"(),;_><=:|'\+-/*?!#&@$^~[]{}
```

If an unsupported character if found in any part of the code, this error will be raised. Due to the error nature, the only part where this error can be raised is in our lexer code.

When the error is raised, we must store which unsupported character was found:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_INVALID_CHAR(mf, cx, line, str) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INVALID_CHAR);\
```

```
    memcpy(mf -> errno_str, str, 4);\
    mf -> errno_str[4] = '\0';}}
```

We should print a corresponding error message telling us which character was found that caused the problem. This requires interpreting the UTF-8 encoding to find out whether we have a character that occupies one, two, three or four bytes. Knowing this, we can print the character in the error message and also indicate its numerical representation:

Section: Print Error Message:

```
case ERROR_INVALID_CHAR:
{
  uint32_t code_point;
  if((unsigned char) mf -> errno_str[0] < 128){
    code_point = (unsigned char) mf -> errno_str[0];
    mf -> errno_str[1] = '\0';
  }
  else if((unsigned char) mf -> errno_str[0] >= 192 &&
          (unsigned char) mf -> errno_str[0] <= 223 &&
          (unsigned char) mf -> errno_str[1] >= 128 &&
          (unsigned char) mf -> errno_str[1] <= 191){
    code_point = (unsigned char) mf -> errno_str[1] - 128;
    code_point += ((unsigned char) mf -> errno_str[0] - 192) * 64;
    mf -> errno_str[2] = '\0';
  }
  else if((unsigned char) mf -> errno_str[0] >= 224 &&
          (unsigned char) mf -> errno_str[0] <= 239 &&
          (unsigned char) mf -> errno_str[1] >= 128 &&
          (unsigned char) mf -> errno_str[1] <= 191 &&
          (unsigned char) mf -> errno_str[2] >= 128 &&
          (unsigned char) mf -> errno_str[2] <= 191){
    code_point = (unsigned char) mf -> errno_str[2] - 128;
    code_point += ((unsigned char) mf -> errno_str[1] - 128) * 64;
    code_point += ((unsigned char) mf -> errno_str[0] - 224) * 4096;
    mf -> errno_str[3] = '\0';
  }
  else if((unsigned char) mf -> errno_str[0] >= 240 &&
          (unsigned char) mf -> errno_str[0] <= 247 &&
          (unsigned char) mf -> errno_str[1] >= 128 &&
          (unsigned char) mf -> errno_str[1] <= 191 &&
          (unsigned char) mf -> errno_str[2] >= 128 &&
          (unsigned char) mf -> errno_str[2] <= 191 &&
          (unsigned char) mf -> errno_str[3] >= 128 &&
          (unsigned char) mf -> errno_str[3] <= 191){
    code_point = (unsigned char) mf -> errno_str[3] - 128;
    code_point += ((unsigned char) mf -> errno_str[2] - 128) * 64;
    code_point += ((unsigned char) mf -> errno_str[1] - 128) * 4096;
    code_point += ((unsigned char) mf -> errno_str[0] - 240) * 262144;
    mf -> errno_str[4] = '\0';
  }
  else{
    fprintf(stderr, "%s:%s Invalid UTF-8 character in source code: '%s'.",
            mf -> file, line_number, mf -> errno_str);
```

```
    break;
  }
  fprintf(stderr, "%s:%s Unsupported UTF-8 character in source code: '%s' (U+%06X).",
          mf -> file, line_number, mf -> errno_str, code_point);
  break;
}
```

A.10. ERROR_INVALID_COMPARISON

Section: Types of Error (continuation):

```
ERROR_INVALID_COMPARISON,
```

This error occurs when the user uses Boolean comparison in path, pen or expressions. Such types cannot be compared with each other. When such an error is generated, we store the invalid expression type that was used in the comparison and also the comparison operator.

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_INVALID_COMPARISON(mf, cx, line, operator, type) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INVALID_COMPARISON);\
    token_to_string(operator, mf -> errno_str);\
    mf -> errno_int = type;}}
```

And the message for diagnostic:

Section: Print Error Message (continuation):

```
case ERROR_INVALID_COMPARISON:
  char expr_type[32];
  tokenid_to_string(mf -> errno_int, expr_type);
  fprintf(stderr,
          "%s:%s Tried to use '%s' to compare an "
          "expression of type '%s', but such type is not comparable.",
          mf -> file, line_number, mf -> errno_str, expr_type);
  break;
```

A.11. ERROR_INVALID_DIMENSION_GLYPH

Section: Types of Error (continuation):

```
ERROR_INVALID_DIMENSION_GLYPH,
```

This error occurs when we are going to render a character, and we are faced with something that will have a non-positive width, or if we add the depth to the height, we also obtain a non-positive value. Since it is impossible to generate a texture with a dimension equal to zero or negative, we generate this error and store the dimensions of the glyph that we were trying to generate:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_INVALID_DIMENSION_GLYPH(mf, cx, line, width, height) {\
  if(!mf -> err){\
    int *buffer = (int *) (mf -> errno_str);\
    buffer[0] = (int) width;\
    buffer[1] = (int) height;\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INVALID_DIMENSION_GLYPH);}}
```

The error will then roduce this diagnostic message:

Section: Print Error Message (continuation):

```
case ERROR_INVALID_DIMENSION_GLYPH:
  int *size_buffer = (int *) (mf -> errno_str);
  fprintf(stderr,
          "%s:%s Glyph with size %dx%d. Expected positive values for ",
          mf -> file, line_number, size_buffer[0], size_buffer[1]);
  if(size_buffer[0] <= 0 && size_buffer[1] <= 0)
    fprintf(stderr, "both width and height+depth.");
  else if(size_buffer[0] <= 0)
    fprintf(stderr, "width.");
  else if(size_buffer[1] <= 0)
    fprintf(stderr, "height+depth.");
  break;
```

A.12. ERROR_INVALID_NAME

Section: Types of Error (continuation):

```
ERROR_INVALID_NAME,
```

This error occurs when the user tries to give a variable an unsupported name because it is the name of a reserved keyword, it is the name of a special variable which cannot be overridden, or the name corresponds to a token which is not symbolic. When this error is invoked, the unsupported token that was used as the variable name and its type must be passed. With the token, we store a string with its name:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_INVALID_NAME(mf, cx, line, tok, type) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INVALID_NAME);\
    token_to_string(tok, mf -> errno_str);\
    mf -> errno_int = type;}}
```

With the token name and its type, we can produce an error message. The message should indicate the name of the token that could not be used as a variable name, and should indicate the reason why this occurred. The reason will be written in an additional string:

Section: Print Error Message (continuation):

```
case ERROR_INVALID_NAME:
  char reason[128];
  if(mf -> errno_int == TYPE_NUMERIC || mf -> errno_int == TYPE_STRING){
    char type[16];
    tokenid_to_string(mf -> errno_int, type);
    sprintf(reason, "it is a %s", type);
  }
  else if(mf -> errno_int == TYPE_SYMBOLIC){
    sprintf(reason,
     "it's a special variable which cannot be shadowed by other declarations");
  }
  else
    sprintf(reason, "it is a reserved keyword");
  fprintf(stderr, "%s:%s You can not use '%s' as a variable name: %s.",
          mf -> file, line_number, mf -> errno_str, reason);
  break;
```

A.13. ERROR_INVALID_TENSION

Section: Types of Error (continuation):

```
ERROR_INVALID_TENSION,
```

When we read the specification of a curve, one of the parameters controlled by user is the segment tension, making the curve that connects two endpoints more rigid, close to a straight line, or looser, making a greater deviation until reaching the destination.

The way in which tension is calculated has the restriction that the value of the tension parameter cannot be less than 0.75, otherwise, we can obtain a set of equations for the curves that will not have a solution.

If a tension value lower than this is found, this error is generated. The macro that generates the error stores in `errno_int` whether the tension with an invalid value was the first (0) or the second (1) in the segment. And it stores in the 32-byte `errno_str` buffer the five floating point values (totaling 20 bytes, which fit easily in the buffer) representing the coordinates of the two endpoints that have the invalid tension between them, and the invalid tension value itself:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_INVALID_TENSION(mf, cx, line, value, position, x1, y1, x2, y2) {\
  if(!mf -> err){\
    float *buffer = (float *) (mf -> errno_str);\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INVALID_TENSION);\
    buffer[0] = (float) (x1);\
    buffer[1] = (float) (y1);\
    buffer[2] = (float) (x2);\
    buffer[3] = (float) (y2);\
    buffer[4] = (float) (value);\
    mf -> errno_int = position;}}
```

The error message sent to the user:

Section: Print Error Message (continuation):

```
case ERROR_INVALID_TENSION:
  float *buf = (float *) (mf -> errno_str);
  fprintf(stderr,
          "%s:%s Between path points (%g, %g) and (%g, %g) we found %s tension value '%g'
smaller than minimal allowed '0.75'.",
          mf -> file, line_number, buf[0], buf[1], buf[2], buf[3],
          (mf -> errno_int == 0)?"first":"second", buf[4]);
  break;
```

A.14. ERROR_MISSING_EXPRESSION

Section: Types of Error (continuation):

```
ERROR_MISSING_EXPRESSION,
```

This error occurs whenever we expected to find an expression of a certain type, but instead, nothing is found. Or, something that is not that expression is found. For example, if we create an assignment with nothing on the right side of the expression.

In the face of such an error, we need to store the type of expression that we expected to find, but did not find:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_MISSING_EXPRESSION(mf, cx, line, type) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_MISSING_EXPRESSION);\
    tokenid_to_string(type, mf -> errno_str);}}
```

And here we print the error message for the user:

Section: Print Error Message (continuation):

```
case ERROR_MISSING_EXPRESSION:
  fprintf(stderr, "%s:%s Missing '%s' expression.",
          mf -> file, line_number, mf -> errno_str);
  break;
```

A.15. ERROR_MISSING_TOKEN

Section: Types of Error (continuation):

```
ERROR_MISSING_TOKEN,
```

This error occurs when the user starts some compound commands (such as `if`, for example) without ending them with their respective closing (`fi`, in this case). When the error is generated, we should use as the error line the place where the opening part is, and the type of expected closing token as last argument:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_MISSING_TOKEN(mf, cx, line, tok) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_MISSING_TOKEN);\
    tokenid_to_string(tok, mf -> errno_str);}}
```

The error message to warn the user about what happened:

Section: Print Error Message (continuation):

```
case ERROR_MISSING_TOKEN:
  fprintf(stderr, "%s:%s Missing matching token '%s'.", mf -> file, line_number,
          mf -> errno_str);
  break;
```

A.16. ERROR_NEGATIVE_LOGARITHM

Section: Types of Error (continuation):

```
ERROR_NEGATIVE_LOGARITHM,
```

This error is raised when we try to compute the logarithm of a negative value. When the error is raised, we convert the negative number to a string representation:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_NEGATIVE_LOGARITHM(mf, cx, line, number) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NEGATIVE_LOGARITHM);\
    snprintf(mf -> errno_str, 31, "%g", number);\
    mf -> errno_str[31] = '\0';}}
```

And warn the user about the problem:

Section: Print Error Message (continuation):

```
case ERROR_NEGATIVE_LOGARITHM:
  fprintf(stderr, "%s:%s Tried to compute logarithm of negative value '%s'.",
          mf -> file, line_number, mf -> errno_str);
  break;
```

A.17. ERROR_NEGATIVE_SQUARE_ROOT

Section: Types of Error (continuation):

```
ERROR_NEGATIVE_SQUARE_ROOT,
```

This error occurs whenever you try to calculate the square root of a negative number. When this occurs, you must store a string representation of the negative number that caused the problem:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_NEGATIVE_SQUARE_ROOT(mf, cx, line, number) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NEGATIVE_SQUARE_ROOT);\
    snprintf(mf -> errno_str, 31, "%g", number);\
    mf -> errno_str[31] = '\0';}}
```

This is the error message for the user:

Section: Print Error Message (continuation):

```
case ERROR_NEGATIVE_SQUARE_ROOT:
  fprintf(stderr, "%s:%s Tried to compute square root of negative value '%s'.",
          mf -> file, line_number, mf -> errno_str);
  break;
```

A.18. ERROR_NESTED_BEGINCHAR

Section: Types of Error (continuation):

```
ERROR_NESTED_BEGINCHAR,
```

This error is raised when we have a `beginchar` statement nested another `beginchar` statement. This cannot happen and have no valid semantic meaning.

THis is a very simple error, so we do not do nothing special besides the default error handling:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_NESTED_BEGINCHAR(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NESTED_BEGINCHAR);}
```

And this is our error message when finding the error:

Section: Print Error Message:

```
case ERROR_NESTED_BEGINCHAR:
  fprintf(stderr,
          "%s:%s You cannot nest 'beginchar' statements.", mf -> file, line_number);
  break;
```

A.19. ERROR_NO_MEMORY

Section: Types of Error (continuation):

```
ERROR_NO_MEMORY,
```

The library defined here to interpret WeaveFont code gets during initialization allocation functions, which could be the standard `malloc` function or any other custom allocator. If we get NULL after running the allocator, this means that we could not get more memory. And in this case, we raise this error.

This error could happen in almost any part of the code. It is not caused by any syntax error in the language. It can be caused by external factors. But it also could be caused by memory leaks due to poor or questionable WeaveFont code. For example:

```
numeric i;
path p;
p = (0, 0);
```

```
for i = 0 step 0 until 1:
  p = p & p;
endfor
```

The above code creates an infinite loop (perhaps because a logical error in the code?) and in each iteration, it increases a path, adding another extremity point. No matter how much memory you have, at some point the loop above will exhaust it all.

When we detect this error, we use the macro below to raise it. The macro just runs the code to store the default information needed by any error:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_NO_MEMORY(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NO_MEMORY);}
```

This is the error message that we will print if the user asks what went wrong:

Section: Print Error Message:

```
case ERROR_NO_MEMORY:
  fprintf(stderr, "%s:%s Not enough memory for allocation.", mf -> file,
          line_number);
  break;
```

A.20. ERROR_NO_PICKUP_PEN

Section: Types of Error (continuation):

```
ERROR_NO_PICKUP_PEN,
```

This error occurs when we use a `pickup` command but do not pass it a valid pen. Perhaps the user has finished the command without passing the pen, perhaps passed something that is not a pen, or perhaps mistakenly thought that a pen expression could be passed to the command (when only pen variables or pen literals actually stored in memory can be passed).

We handle this error using the standard macro:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_NO_PICKUP_PEN(mf, cx, line) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NO_PICKUP_PEN);}}
```

And this is our error message:

Section: Print Error Message:

```
case ERROR_NO_PICKUP_PEN:
  tokenid_to_string(mf -> errno_int, mf -> errno_str);
  fprintf(stderr, "%s:%s After a 'pickup' command, you should use either a "
                  "'nullpen', 'pencircle', 'pensemicircle' or a pen variable.",
                  mf -> file, line_number);
  break;
```

A.21. ERROR_NONCYCLICAL_PEN

Section: Types of Error (continuation):

```
ERROR_NONCYCLICAL_PEN,
```

This error occurs when we want to create a new pen with a custom format using `makepen`. To do this, we must pass a path as the operand of this command, which must be cyclic. If it is not cyclic, this error is generated:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_NONCYCLICAL_PEN(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NONCYCLICAL_PEN);}
```

The error message to the user:

Section: Print Error Message (continuation):

```
case ERROR_NONCYCLICAL_PEN:
  fprintf(stderr, "%s:%s Tried to create a pen from non-cyclical path.",
          mf -> file, line_number);
  break;
```

A.22. ERROR_NULL_VECTOR_ANGLE

Section: Types of Error (continuation):

```
ERROR_NULL_VECTOR_ANGLE,
```

This error occurs when we try to measure the angle of vector $(0, 0)$ with the x-axis. However, since this is the null vector, there is no angle and using this operation over the null vector results in this error being raised.

This error is handled in the standard way:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_NULL_VECTOR_ANGLE(mf, cx, line) {\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NULL_VECTOR_ANGLE);}
```

And this is the error message that warns the user:

Section: Print Error Message (continuation):

```
case ERROR_NULL_VECTOR_ANGLE:
  fprintf(stderr, "%s:%s You cannot use 'angle' operator in a null vector '(0, 0)'.",
          mf -> file, line_number);
  break;
```

A.23. ERROR_OPENGL_FRAMEBUFFER

Section: Types of Error (continuation):

```
ERROR_OPENGL_FRAMEBUFFER,
```

This error occurs when a framebuffer cannot be created internally due to errors in OpenGL. Ideally, this should not happen, but if it does, it may be caused by incorrect OpenGL configuration by the user.

To help debug this type of error, we store the state of the current framebuffer in OpenGL, where the error likely occurred:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_OPENGL_FRAMEBUFFER(mf, cx, line) {\
  if(!mf -> err){\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_OPENGL_FRAMEBUFFER);\
  mf -> errno_int = glCheckFramebufferStatus(GL_FRAMEBUFFER);}}
```

The error message depends on the framebuffer status:

Section: Print Error Message (continuation):

```
case ERROR_OPENGL_FRAMEBUFFER:
  fprintf(stderr, "%s:%s OpenGL error. Couldn't create framebuffer for image.",
          mf -> file, line_number);
  switch(mf -> errno_int){
    case GL_FRAMEBUFFER_UNDEFINED:
```

```
      printf(" Default framebuffer not defined.");
      break;
    case GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT:
      printf(" The framebuffer had incomplete attachment.");
      break;
    case GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT:
      printf(" The framebuffer had no image attached to it.");
      break;
    case GL_FRAMEBUFFER_UNSUPPORTED:
      printf(" Depth and stencil attachments are not the same renderbuffer, "
             "or internal image format is not supported by this implementation.");
      break;
    case GL_FRAMEBUFFER_INCOMPLETE_MULTISAMPLE:
      printf("  GL_RENDERBUFFER_SAMPLES is not the same for all attached "
             " renderbuffers or attached images are a mix of renderbuffers "
             "and textures while the value of GL_RENDERBUFFER_SAMPLES "
             "is not zero. ");
      break;
    default:
      printf(" Unknown error.");
      break;
  }
  break;
```

A.24. ERROR_RECURSIVE_RENDERCHAR

Section: Types of Error (continuation):

```
ERROR_RECURSIVE_RENDERCHAR,
```

This error occurs when the use of the `renderchar` command on two or more glyphs generates a circular dependency. For example, to render the glyph “a”, we must first render “b”. But to render “b”, we must render “a”. No type of recursion of this type is allowed, even if the user takes care to create a logic that prevents it from being infinite. When a recursive definition of this type is encountered, this error is generated. And we inform which glyph we need to render as a dependency of the current one, such that the current one is a dependency of it.

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_RECURSIVE_RENDERCHAR(mf, cx, line, glyph) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_RECURSIVE_RENDERCHAR);\
    memcpy(mf -> errno_str, glyph, 4);\
    mf -> errno_str[4] = '\0';}}
```

This is the error message to be print in such cases:

Section: Print Error Message (continuation):

```
case ERROR_RECURSIVE_RENDERCHAR:
  fprintf(stderr, "%s:%s Recursive 'renderchar' detected. Glyph '%s' depends on "
          "current glyph, but current glyph depends on '%s'.", mf -> file,
          line_number, mf -> errno_str, mf -> errno_str);
  break;
```

A.25. ERROR_UNBALANCED_ENDING_TOKEN

Section: Types of Error (continuation):

```
ERROR_UNBALANCED_ENDING_TOKEN,
```

This is an error, usually caught during lexical analysis, where a token that ends a compound statement (`elseif`, `endif`, `endfor`, `endchar`) is found in a context where such a token was not supposed to be found. Either the beginning part of the compound statement was not found , or there is more than one unbalanced compound statement.

If we find this error, we store the token id number from our unexpected closing token:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNBALANCED_ENDING_TOKEN(mf, cx, line, tok) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNCLOSED_STRING);\
    mf -> errno_int = tok;}}
```

And we print the following error message to war the user:

Section: Print Error Message:

```
case ERROR_UNBALANCED_ENDING_TOKEN:
  tokenid_to_string(mf -> errno_int, mf -> errno_str);
  fprintf(stderr, "%s:%s Unexpected token \"%s\" found. You are trying to "
          "close a compund command that were never opened or you have "
          "two or more unbalanced compound statements.", mf -> file,
          line_number, mf -> errno_str);
  break;
```

A.26. ERROR_UNCLOSED_DELIMITER

Section: Types of Error (continuation):

```
ERROR_UNCLOSED_DELIMITER,
```

This error occurs when a delimiter such as parentheses, brackets or curly braces has been opened but not closed. Since each of these delimiters is represented by a character, when generating the error, the character corresponding to the delimiter that was not closed must be passed as the last argument:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNCLOSED_DELIMITER(mf, cx, line, delimiter) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNCLOSED_DELIMITER);\
    mf -> errno_int = delimiter;}}
```

And here we print the error message:

Section: Print Error Message (continuation):

```
case ERROR_UNCLOSED_DELIMITER:
  fprintf(stderr, "%s:%s Delimiter '%c' was not closed.",
          mf -> file, line_number, mf -> errno_int);
  break;
```

A.27. ERROR_UNCLOSED_STRING

Section: Types of Error (continuation):

```
ERROR_UNCLOSED_STRING,
```

This is a lexical error that is generated during token creation, not during code execution. It occurs when code begins a string with double quotes but does not end it. If a line break is found within a string, this also constitutes an unterminated string.

When we encounter such an error, we copy what would be the unterminated string so that we can present it if the user requests an error message:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNCLOSED_STRING(mf, cx, line, str) {\
  if(!mf -> err){\
    size_t _len = strlen(str);\
    if(_len > 32) _len = 32;\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNCLOSED_STRING);\
    memcpy(mf -> errno_str, str, _len);\
    mf -> errno_str[31] = '\0';}}
```

The error message in this case only reports the incomplete string found and its position, in the hope that this will be enough for the user to fix the problem:

Section: Print Error Message:

```
case ERROR_UNCLOSED_STRING:
  fprintf(stderr, "%s:%s Unclosed string \"%s\".", mf -> file,
          line_number, mf -> errno_str);
  break;
```

A.28. ERROR_UNDECLARED_VARIABLE

Section: Types of Error (continuation):

```
ERROR_UNDECLARED_VARIABLE,
```

This error happens if the user uses an undeclared variable. When this is raised, we must pass as argument the symbolic token which was undeclared:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNDECLARED_VARIABLE(mf, cx, line, tok) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNDECLARED_VARIABLE);\
    token_to_string((struct generic_token *) tok, mf -> errno_str);}}
```

And when we print the error message, we say which variable name was being used undeclared:

Section: Print Error Message (continuation):

```
case ERROR_UNDECLARED_VARIABLE:
  fprintf(stderr, "%s:%s Variable '%s' was not declared.", mf -> file,
          line_number, mf -> errno_str);
  break;
```

A.29. ERROR_UNEXPECTED_TOKEN

Section: Types of Error (continuation):

```
ERROR_UNEXPECTED_TOKEN,
```

Unexpected token errors mean that we read a token that we did not expect. We were expecting something else (we do not know what), but not this specific token that we encountered. This also occurs when we use a `endfor` without having opened a `for` at the current nesting level. Or when we mistakenly use a reserved word in a context in which it was not expected.

If this error occurs, we must store in a string the unexpected token we found:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, line, tok) {\
  if(!mf -> err){\
```

```
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNEXPECTED_TOKEN);\
    token_to_string(tok, mf -> errno_str);}}
```

And using it, we can produce an error message:

Section: Print Error Message (continuation):

```
case ERROR_UNEXPECTED_TOKEN:
  fprintf(stderr, "%s:%s We found '%s' token in a context where such "
          "token makes no sense.", mf -> file, line_number, mf -> errno_str);
  break;
```

A.30. ERROR_UNINITIALIZED_VARIABLE

Section: Types of Error (continuation):

```
ERROR_UNINITIALIZED_VARIABLE,
```

This error is raised when we try to read the value of somo uninitialized variable in an expression. When this error occurs, we have to pass as argument the variable token, as well as its type. With this, we store the name of the variable as a string and its type as a numeric value:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNINITIALIZED_VARIABLE(mf, cx, line, var_token, var_type) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNINITIALIZED_VARIABLE);\
    token_to_string((struct generic_token *) (var_token), mf -> errno_str);\
    mf -> errno_int = var_type;}}
```

And this allow us to generate an error message:

Section: Print Error Message (continuation):

```
case ERROR_UNINITIALIZED_VARIABLE:
  char var_type[32];
  tokenid_to_string(mf -> errno_int, var_type);
  fprintf(stderr, "%s:%s Uninitialized %s variable '%s'.",
          mf -> file, line_number, var_type, mf -> errno_str);
  break;
```

A.31. ERROR_UNKNOWN_GLYPH_DEPENDENCY

Section: Types of Error (continuation):

```
ERROR_UNKNOWN_GLYPH_DEPENDENCY,
```

This error occurs when a glyph has a rendering dependency on another glyph, which in turn has not been defined. It occurs when we use the `renderchar` command, passing it an undefined glyph. When faced with this error, we need to store as a string the name of the non-existent glyph that we found.

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNKNOWN_GLYPH_DEPENDENCY(mf, cx, line, glyph) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNKNOWN_GLYPH_DEPENDENCY);\
    memcpy(mf -> errno_str, glyph, 4);\
    mf -> errno_str[4] = '\0';}}
```

And with this, we produce the error message:

Section: Print Error Message (continuation):

```
case ERROR_UNKNOWN_GLYPH_DEPENDENCY:
```

```
  fprintf(stderr,
          "%s:%s Command 'renderchar' created dependency of undefined glyph '%s'.",
          mf -> file, line_number, mf -> errno_str);
  break;
```

A.32. ERROR_UNKNOWN_EXPRESSION

Section: Types of Error (continuation):

```
ERROR_UNKNOWN_EXPRESSION,
```

This error occurs when we interpret an expression of a certain type (numeric, pair, transformation, path, image or boolean), but we encounter a sequence of tokens that were not predicted by any grammatical rule.

Generating this error requires storing the type of expression we were parsing when we encountered something completely unknown:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, line, type) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNKNOWN_EXPRESSION);\
    mf -> errno_int = type;}}
```

And the error message for diagnostic:

Section: Print Error Message (continuation):

```
case ERROR_UNKNOWN_EXPRESSION:
  fprintf(stderr, "%s:%s Unknown %s expression.", mf -> file,
          line_number, list_of_keywords[mf -> errno_int - 10]);
  break;
```

A.33. ERROR_UNKNOWN_STATEMENT

Section: Types of Error (continuation):

```
ERROR_UNKNOWN_STATEMENT,
```

This is an error generated when we read a statement in the language, but we are not able to identify its type. We do not know if it is a compound statement, variable declaration, assignment or command.

The error is too cryptic for us to be able to help the user much more or point out what is wrong, since we are unable to identify what the user intended to express. As we do not have much information about this error, it is generated in a very generic way:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNKNOWN_STATEMENT(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNKNOWN_STATEMENT);}
```

And this is the print message if the user asks what went wrong:

Section: Print Error Message (continuation):

```
case ERROR_UNKNOWN_STATEMENT:
  fprintf(stderr, "%s:%s Unknown statement. Perhaps you misspelled some "
          "operator, forgot an assignment or placed a ';' in the wrong "
          "place.", mf -> file, line_number);
  break;
```

A.34. ERROR_UNOPENED_DELIMITER

Section: Types of Error (continuation):

```
ERROR_UNOPENED_DELIMITER,
```

Esta mensagem de erro ocorre quando encontramos o fechamento de um delimitador (fechar parnteses, colchetes ou chaves) quando tal delimitador no foi aberto.

Quando este erro gerado, devemos informar o caractere do delimitador sendo fechado que encontramos e que gerou o problema:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNOPENED_DELIMITER(mf, cx, line, delimiter) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNOPENED_DELIMITER);\
    mf -> errno_int = delimiter;}}
```

Esta a mensagem de diagnstico deste erro:

Section: Print Error Message (continuation):

```
case ERROR_UNOPENED_DELIMITER:
  fprintf(stderr, "%s:%s Delimiter '%c' was not previously opened.",
          mf -> file, line_number, mf -> errno_int);
  break;
```

A.35. ERROR_UNSUPORTED_LENGTH_OPERAND

Section: Types of Error (continuation):

```
ERROR_UNSUPORTED_LENGTH_OPERAND,
```

This error always occurs after we use the 'length' operator. Such operator acts upon either on a number (to calculate its modulus), on a pair (to calculate its Euclidean norm) or on a path (to calculate the number of curves that form the path). This means that after finding such an operator, we need to identify whether we are dealing with a number, a pair or a path. If we are dealing with something else or an expression that we cannot identify, then this error is generated.

When generating such an error, we pass as the last argument the type of expression that we found instead of the one we expected. If the type is unknown, we will pass the number -1 in this argument:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_UNSUPORTED_LENGTH_OPERAND(mf, cx, line, type) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNSUPORTED_LENGTH_OPERAND);\
    mf -> errno_int = type;}}
```

And this is how we produce the error message:

Section: Print Error Message (continuation):

```
case ERROR_UNSUPORTED_LENGTH_OPERAND:
  if(mf -> errno_int == -1)
    fprintf(stderr, "%s:%s Operator 'length' expects a numeric, pair or path "
            "expression as operand. Instead, we found an unknown expression.",
            mf -> file, line_number);
  else{
    tokenid_to_string(mf -> errno_int, mf -> errno_str);
    fprintf(stderr, "%s:%s Operator 'length' expects a numeric, pair or path "
                    "expression as operand. Instead, we found a %s expression.",
                    mf -> file, line_number, mf -> errno_str);
  }
  break;
```

A.36. ERROR_WRONG_NUMBER_OF_PARAMETERS

Section: Types of Error (continuation):

```
ERROR_WRONG_NUMBER_OF_PARAMETERS,
```

Algumas instrues na linguagem WeaveFont requerem que umm certo nmero de parmetros seja passado. Por exemplo, `beginchar(a, b, c, d)` requer quatro parmetros, aqui representados por `a`, `b`, `c` e `d`. Se um nmero menor ou maior for passado, este erro gerado.

Quando geramos este erro, precisamos indicar qual instruo `s` causou o problema (armazenaremos o nome dela como string), o nmero de argumentos `e` que espervamos e o valor `f` que o quantos obtivemos ao invs disso (ambos os valores inteiros so combinados e armazenados juntos em `errno_int`):

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_WRONG_NUMBER_OF_PARAMETERS(mf, cx, line, s, e, f) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_WRONG_NUMBER_OF_PARAMETERS);\
    tokenid_to_string(s, mf -> errno_str);\
    mf -> errno_int = (e << 8) + f;}}
```

Com estas informaes, podemos gerar a devida mensagem de erro:

Section: Print Error Message (continuation):

```
case ERROR_WRONG_NUMBER_OF_PARAMETERS:
  fprintf(stderr,
          "%s:%s Statement '%s' expected %d parameters, but %d were given.",
          mf -> file, line_number, mf -> errno_str, mf -> errno_int >> 8,
          mf -> errno_int & 255);
  break;
```

A.37. ERROR_WRONG_VARIABLE_TYPE

Section: Types of Error (continuation):

```
ERROR_WRONG_VARIABLE_TYPE,
```

This error occurs when we use a variable of one type when it was expected to be of another type. It occurs when we are going to perform an assignment between variables of different types or when a variable of one type occurs within an expression that required a variable of a different type. When this error is encountered, we must store the name of the wrong variable, its type and also the type that was expected in its place. Since in the C language an integer variable has at least 16 bits, we use the first 8 bits to store the type of the variable found and the last 8 to store the expected type:

Section: Local Macros (metafont.c) (continuation):

```
#define RAISE_ERROR_WRONG_VARIABLE_TYPE(mf, cx, line, tok, type, expected) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_WRONG_VARIABLE_TYPE);\
    token_to_string((struct generic_token *) tok, mf -> errno_str);\
    mf -> errno_int = (type << 8) + expected;}}
```

Printing the error message requires converting the types to strings so that we can write them in the diagnostic message:

Section: Print Error Message (continuation):

```
case ERROR_WRONG_VARIABLE_TYPE:
  char expected[32], found[32];
  tokenid_to_string(mf -> errno_int & 255, expected);
  tokenid_to_string(mf -> errno_int >> 8, found);
```

```
fprintf(stderr, "%s:%s Variable '%s' is a '%s' variable, but we expected a "
        "'%s' variable.", mf -> file, line_number, mf -> errno_str,
        found, expected);
break;
```

References

De Berg, M. (2000) "Computational geometry: algorithms and applications". Springer Science & Business Media.

Knuth, D. E. (1984) "Literate Programming", The Computer Journal, Volume 27, Issue 2, Pages 97–111.

Hobby, J. D. (1986), "Smooth, easy to compute interpolating splines", Discrete & computational geometry, 1(2), 123-140

Knuth, D. E. (1989) "The METAFONT book", Addison-Wesley Longsman Publishing Co., Inc