WeaveFont: Linguagem de Desrição de Fontes Tipográficas

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 [KNUTH, 1989]. 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.*

Resumo: *Este artigo contém uma implementação de WeaveFont, uma linguagem criada para representar fontes tipográficas, ilustrações vetoriais e animações. A linguagem é fortemente baseada em METAFONT, uma linguagem de descrição criada por Donald Knuth em 1977 [KNUTH, 1989]. A sintaxe é muito similar para ambas as linguagens. mas WeaveFont foi criada para executar e produzir imagens em tempo real usando OpenGL. Por isso, mudanças foram feitas para tornar a linguagem mais rápida para tal propósito. Ao contrário de METAFONT, WeaveFont não suporta macros e é uma linguagem imperativa. Várias funcionalidades úteis que em METAFONT foram criadas usando macros, em WeaveFont são diretamente parte da linguagem.*

Índice

1. Introdução

A primeira linguagem de descrição para fontes tipográficas foi a METAFONT. Ela foi criada em 1984 por Donald Knuth e difere de outros formatos de fontes tipográficas por permitir a criação de fontes à partir da modificação de parâmetros definidos na descrição básica da fonte. Desta forma, o projetista de uma fonte não deve criar uma simples fonte tipográfica, mas uma meta-fonte, a qual pode por sua vez gerar muitas outras fontes tipográficas diferentes mediante a simples modificação dos parâmetros. Aqui também chamaremos de meta-fonte as fontes que definiremos, as quais não devem ser confundidas com a linguagem METAFONT.

A especificação original da linguagem METAFONT pode ser encontrada em [KNUTH, 1989]. Baseada nesta linguagem, definiremos aqui a linguagem WeaveFont, a qual tem objetivos similares. A diferença é que a linguagem a ser definida aqui deverá ser usada em tempo real para produzir imagens à partir da descrição vetorial.

O arquivo de código-fonte que está sendo definido neste artigo pode tanto ser usado sozinho para usar somente a linguagem WeaveFont, ou pode estar integrado ao subsistema do Motor de Jogos Weaver, do qual esta linguagem é um sub-projeto. Se estivermos usando este projeto dentro do motor de jogos, isso significa que temos definida a macro `WEAVER_ENGINE`. Neste caso, devemos definir a seguinte função:
(P)

Seção: Declaração de Função (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
```

Esta função serve para ler um arquivo (`source_filename`) e gerar com ele uma nova interface de usuário a ser usada pelo motor de jogos. No caso, o arquivo com código WeaveFont será interpretado como uma imagem vetorial ou uma animação vetorial.

E nós precisamos também inserir o cabeçalho Weaver de interface de usuário:

Seção: Inclui Cabeçalhos Gerais (metafont.h):

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

Tanto se estivermos executando este código de dentro do Motor Weaver ou de fora dele, uma operação que será necessária será ler uma fonte tipográfica, uma meta-fonte. Depois de lê-la, nós retornamos um ponteiro para uma estrutura com todas as informações relevantes sobre ela. Esta estrutura pode então ser usada para renderizar caracteres que estiverem definidos ali. Depois de renderizar o que queremos, precisaremos de uma segunda função para desalocar a estrutura. Em suma, iremos definir também as seguintes funções para criar e destruir uma nova meta-fonte que serão sempre exportadas:

Seção: Declaração de Função (metafont.h) (continuação):

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

Podemos em um mesmo projeto criar e manter muitas meta-fontes para renderizar caracteres ou imagens delas. Mas antes de começar a criar a primeira delas, será importante chamar a seguinte função de inicialização que especifica algumas das funções que devem ser usadas, e além disso nos

informa quantos pontos por polegada tem a nossa tela (DPI):

Seção: Declaração de Função (metafont.h) (continuação):

```
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);
```

As funções passadas como argumento para tal inicializador são respectivamente uma para fazer alocações temporárias de memória, outra para desalocar o que foi alocado com ela (pode ser NULL), outra para fazer alocações mais permanentes, outra para desalocar o que foi alocado com ela (pode ser NULL também), uma para gerar 64 bits aleatórios e por fim a medida de DPI. A função retorna verdadeiro se a inicialização foi bem-sucedida.

Após terminar o uso de nossas funções, deve-se chamar a função abaixo que finaliza e desaloca qualquer coisa que tenha sido gerada pela função de inicialização:

Seção: Declaração de Função (metafont.h) (continuação):

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

Caso algum erro tenha ocorrido durante a execução de um código WeaveFont que define uma meta-fonte, então o usuário poderá usar a seguinte função para imprimir na tela uma mensagem descrevendo o erro encontrado:

Seção: Declaração de Função (metafont.h) (continuação):

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

1.1. Programação Literária

Nossa API será escrita usando a técnica de Programação Literária, proposta por Knuth em [KNUTH, 1984]. Ela consiste em escrever um programa de computador explicando didaticamente em texto o que se está fazendo à medida que apresenta o código. Depois, o programa é compilado através de programas que extraem o código diretamente do texto didático. O código deve assim ser apresentado da forma que for mais adequada para a explicação no texto, não como for mais adequado para o computador.

Seguindo esta técnica, este documento não é uma simples documentação do nosso código. Ele é por si só o código. A parte que será extraída e compilada posteriormente pode ser identificada como sendo o código presente em fundo cinza. Geralmente começamos cada trecho de código com um título que a nomeia. Por exemplo, imediatamente antes desta subseção nós apresentamos uma série de declarações. E como pode-se deduzir pelo título delas, a maioria será posteriormente posicionada dentro de um arquivo chamado `metafont.h`.

Podemos apresentar aqui a estrutura do arquivo `metafont.h`:
(P)

Arquivo: `src/metafont.h`:

```
#ifndef __WEAVER_METAFONT
#define __WEAVER_METAFONT
#ifdef __cplusplus
extern "C" {
#endif
#include <stdbool.h> // Define tipo 'bool'
#include <stdlib.h>  // Define 'atof', 'abs'
#if defined(__linux__) || defined(BSD) || defined(__EMSCRIPTEN__)
#include <GLES3/gl3.h> // Nossa linguagem renderiza com OpenGL
#endif
#if !defined(_WIN32)
#include <sys/param.h> // Necessário no BSD, mas causa problema no Windows
#endif
```

```
        <Seção a ser Inserida: Inclui Cabeçalhos Gerais (metafont.h)>
          <Seção a ser Inserida: Declarações Gerais (metafont.h)>
          <Seção a ser Inserida: Estrutura de Dados (metafont.h)>
          <Seção a ser Inserida: Declaração de Função (metafont.h)>
#ifdef __cplusplus
}
#endif
#endif
```

O código acima mostra a burocracia padrão para definir um cabeçalho para nossa API em C. As duas primeiras linhas mais a última são macros que garantem que esse cabeçalho não será inserido mais de uma vez em uma mesma unidade de compilação. As linhas 3, 4, 5, assim como a penúltima, antepenúltima e a antes da antepenúltima tornam o cabeçalho adequado a ser inserido em código C++. Essas linhas apenas avisam que o que definirmos ali deve ser encarado como código C. Por isso o compilador está livre para fazer otimizações sabendo que não usaremos recursos da linguagem C++, como sobrecarga de operadores. Logo em seguida, inserimos um cabeçalho que nos permite declarar o tipo booleano. E tem também uma parte em vermelha. Note que uma delas é "Declaração de Função (metafont.h)", o mesmo nome apresentado no trecho de código mostrado quando descrevemos nossa API antes dessa subseção. Isso significa que aquele código visto antes será depois inserido ali. As outras partes em vermelho representam código que ainda iremos definir nas seções seguintes.

Caso queira observar o que irá no arquivo `metafont.c` associado a este cabeçaho, o código será este:

(P)

Arquivo: `src/metafont.c`:

```
#include "metafont.h"
          <Seção a ser Inserida: Cabeçalhos Locais (metafont.c)>
            <Seção a ser Inserida: Macros Locais (metafont.c)>
        <Seção a ser Inserida: Estrutura de Dados Locais (metafont.c)>
            <Seção a ser Inserida: Variáveis Locais (metafont.c)>
        <Seção a ser Inserida: Declaração de Função Local (metafont.c)>
          <Seção a ser Inserida: Funções Auxiliares Locais (metafont.c)>
        <Seção a ser Inserida: Definição de Funções da API (metafont.c)>
```

Todo o código que definiremos e explicaremos a seguir será posicionado nestes dois arquivos. Além deles, nenhum outro arquivo será criado.

2. Código Auxiliar Geral

O código apresentado nesta seção tem como característica ser usado em diferentes partes de nossa linguagem e também ser independente de estruturas de dados específicas deste projeto. Por causa disso, vamos defini-lo como uma introdução, antes do código mais específico.

2.1. Código de Álgebra Linear

Iremos usar com uma frequência muito grande matrizes 3×3, que serão usadas tipicamente para representar transformações lineares em um espaço vetorial de 3 dimensões. Tais matrizes serão simplesmente um array de 9 elementos, representando os elementos da matriz.

Os valores da matriz estarão dispostos da seguinte forma, de acordo como os armazenamos no array M:

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

Sendo assim, podemos inicializar uma matriz identidade com:

Seção: Macros Locais (metafont.c):

```
#define INITIALIZE_IDENTITY_MATRIX(I) {\
  int _i;\
```

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

Apesar de nossas matrizes lidarem com espaço vetorial de 3 dimensões, na prática todos os valores que usaremos estarão contidos no plano $\{(x, y, 1)\}$. tal que x e y são números reais. O motivo de trabalharmos no espaço de 3 dimensões é poder representar a translação (ou deslocamento) de pontos como transformações lineares, algo que só se torna possível no espaço 2D se o tratarmos como inscrito dentro de um espaço vetorial 3D.

Dado um vetor de 3 dimensões $(x, y, 1)$, podemos transformá-lo em um novo vetor $(x', y', 1)$ aplicando a transformação linear representada por uma matriz abaixo:

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

As novas coordenadas (x', y') do novo vetor podem ser calculadas com ajuda das macros abaixo:

Seção: Macros Locais (metafont.c) (continuação):

```
#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])
```

A multiplicação de matrizes tem a propriedade de associatividade abaixo:

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

Isso quer dizer que podemos acumular várias transformações em uma só matriz. Aplicar a transformaçao AB sobre um vetor é o mesmo que aplicar primeiro a transformação A e depois a transformação B. O código abaixo acumula duas transformações lineares, calculando AB e armazenando o resultado em A. Note que o código sempre assume que a última coluna de toda matriz é $(0, 0, 1)^T$:

Seção: Macros Locais (metafont.c) (continuação):

```
#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];\
}
```

Existem algumas transformações lineares que são muito mais comuns do que outras. Para realizá-las mais facilmente vamos criar macros especiais para elas. Por exemplo, rotacionar um vetor $(x, y, 1)$ em relação ao ponto $(0, 0, 1)$ um ângulo θ é feito multiplicando pela matriz:

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

Acumular a transformação acima sobre uma matriz é feito com a macro seguinte:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;\
}
```

Outra transformação relevante é ampliar ou reduzir vetores, os esticando ou comprimindo. Para fazer isso no eixo x, multiplicamos pela seguinte matriz:

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

O que é feito pela macro abaixo:

Seção: Macros Locais (metafont.c) (continuação):

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

Fazer isso no eixo y envolve a multiplicação:

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

E pra isso usamos a macro:

Seção: Macros Locais (metafont.c) (continuação):

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

E fazer isso nos dois eixos é feito simplesmente combinando ambas as operações:

Seção: Macros Locais (metafont.c) (continuação):

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

Uma translação, ou deslocamento é a única operação que exige que usemos três dimensões ao invés de duas. Deslocar um vetor uma distância a na horizontal e uma distância b na vertical é feito pela transformação:

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

E essa multiplicação de matrizes é feita usando a seguinte macro:

Seção: Macros Locais (metafont.c) (continuação):

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

Inclinar com intensidade s é uma operação feita pela multiplicação:

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

Que realizamos sobre outras matrizes com a macro abaixo:

Seção: Macros Locais (metafont.c) (continuação):

```
#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];\
}
```

A última transformação especial que iremos tratar aqui é a mudança de escala no plano complexo. Isso significa multiplicar os pontos por um par (s, t), interpretado como um número complexo. Isso obtém ao mesmo tempo tanto rotação como mudança de escala. É feito pela multiplicação de matriz:

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

Realizar esta multiplicação por outra matriz é feito com essa macro:

Seção: Macros Locais (metafont.c) (continuação):

```
#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. Resolvendo Sistemas de Equações Lineares

Resolver sistemas de equações lineares é o problema central da álgebra linear. São equações em que as incógnitas estão sempre sendo multiplicadas por constantes numéricas e somadas (sem haver duas incógnitas multiplicadas entre si). Por exemplo:

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

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

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

Podemos representar o sistema de equações acima na forma matricial, associando a primeira coluna a x, a segunda a y e a terceira a z:

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

Resolver o sistema é encontrar uma solução válida para x. No caso: $X = \begin{bmatrix}0\\ 0\\ 2\end{bmatrix}$

Uma função capaz de resolver este sistema pode ser a abaixo, que recebe um número de dimensões n, um vetor já alocado com os n^2 elementos de uma matriz, um vetor já alocado com n elementos do lado direito da equação, e um próximo vetor com n valores da solução, que serão preenchidos pela função:

Seção: Declaração de Função Local (metafont.c):

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

Antes de mostrar como resolver o sistema, é útil definir algumas novas macros para nos ajudar. Vamos assumir que toda matriz, independente do tamanho está alocada em um único vetor contínuo que concatena cada uma de suas linhas. Será muito comum que tenhamos que trocar a ordem das linhas de uma matriz para resolver o sistema de equações. Trocar linhas da matriz M, que representa o lado esquerdo de um sistema de equações, também significa trocar as mesmas linhas de b, a matriz representando o lado direito do sistema de equações.

A macro abaixo mostra o que acontece quando, em um sistema de n linhas, trocamos na matriz M e em b as linhas i e j:

Seção: Macros Locais (metafont.c) (continuação):

```
#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 que trocar as linhas não muda o resultado de um sistema de equações. Por exemplo, as três equações abaixo tem a mesma solução ($x = 0$, $y = 0$, $z = 2$) do sistema de equações do começo da Subseção:

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

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

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

Outra operação relevante é, dado um sistema com n linhas, com a matriz M e a matriz b, devemos combinar as linhas i e j, subtraindo da linha i o conteúdo da linha j multiplicado por q:

Seção: Macros Locais (metafont.c) (continuação):

```
#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]);}}
```

Realizar esta operação tampouco muda o resultado do sistema de equações. Por exemplo, no sistema logo acima, podemos subtrair a segunda linha pela primeira multiplicada por $1/6$ e a terceira pela primeira multiplicada por $1/3$. O resultado seria:

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

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

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

Embora o resultado não tenha mudado, conseguimos simplificar o sistema, deixando as duas últimas equações com apenas duas incógnitas. Com mais uma operação, é possível deixar a última equação com apenas uma incógnita e à partir daí a solução se torna trivial.

O que queremos então é realizar as operações das duas macros acima sucessivas vezes até obter uma matriz triangular com todos os elementos abaixo da diagonal principal nulos. Depois, obter a

solução se torna fácil: na última linha só há e podemos calcular diretamente o resultado. Sabendo o valor da última incógnita, resta apenas uma incógnita desconhecida na penúltima linha. E assim continuamos até chegar à primeira linha e não restar mais incógnitas desconhecidas.

Para construir a matriz triangular, temos uma iteração onde percorremos uma linha de cada vez. Na iteração i, nosso objetivo é tornar nulo todos os elementos da coluna i que estejam abaixo da linha i. Para isso, escolhemos a linha atual ou abaixo dela que tem o maior valor absoluto na coluna i e trocamos a posição dela com a linha atual. Isso não é estritamente necessário, mas escolher o maior valor possível minimiza os erros da operação. E então à partir disso escolhemos valores adequados para transformar a matriz e resolvê-la:

Seção: Funções Auxiliares Locais (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 pivô possível na coluna atual e troca de posição 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 pivô 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 solução à 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];
  }
}
```

Note que não preservamos o estado das matrizes passadas como argumento para a função. E que não fazemos qualquer checagem de erro. É responsabilidade de quem chamar esta função alocar vetores de tamanho correto e só passar sistemas que tenham uma solução (ou uma divisão por zero irá ocorrer e matar o programa).

Note that we do not preserve the state of the matrices passed as an argument for the function. And we don't do any error checking. AND responsibility of whoever calls this function to allocate vectors of size correct and only pass systems that have a solution (or a division by zero will occur and kill the program).

2.2. Mutexes e Seções Críticas

Um mutex é uma estrutura de dados abstrata usada para controlar acesso de múltiplos processos a um recurso em comum. Eles serão tratados de forma diferente dependendo do sistema operacional e ambiente. Devido à seu caráter não-portável, teremos que usar diferentes técnicas dependendo de onde o código será compilado.

No Linux e BSD, o Mutex é definido pela biblioteca `pthread` e segue a nomenclatura típica dela. No Web Assembly, o Emscripten possui uma implementação com API compatível, mas ela só irá funcionar se a página web que servir a aplicação passar os cabeçalhos certos, por isso só

iremos suportar threads nele se o usuário pedir definindo uma macro `W_ALWAYS_USE_THREADS`. No Windows há uma diferença entre um mutex, que pode ser compartilhado entre diferentes programas e uma "seção crítica", que só pode ser compartilhada entre threads de um mesmo programa. No nosso caso, devemos usar seções críticas que nos dão uma performance melhor.

Seção: Declarar um Mutex:

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

Isso significa que no Linux e BSD nós precisamos inserir o cabeçalho da biblioteca `pthread`. No Windows, basta o cabeçalho padrão do windows que já colocamos em `metafont.h`.

Seção: Inclui Cabeçalhos Gerais (metafont.h):

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

No nosso código vamos precisar inicializar cada Mutex que declararmos. Para isso, usaremos a seguinte macro:

Seção: Macros Locais (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
```

Para finalizar um Mutex quando não precisarmos mais, usamos a seguinte macro:

Seção: Macros Locais (metafont.c) (continuação):

```
#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
```

Tendo um Mutex, há duas operações que podemos fazer com ele. A primeira é requerer o uso do Mutex. Neste momento, se algum outro processo está usando ele, iremos esperar até que o Mutex esteja livre novamente:

Seção: Macros Locais (metafont.c) (continuação):

```
#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
```

E finalmente, depois de deixarmos de usar o recurso guardado pelo Mutex, podemos liberar ele com o código abaixo:

Seção: Macros Locais (metafont.c):

```
#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. Lidando com Erros

Boa parte das funções que são definidas aqui recebem como entrada uma estrutura da metafonte sendo renderizada e uma estrutura do contexto de execução. Além disso, elas constumam retornar verdadeiro caso a execução seja bem-sucedida e falsa caso contrário. Algumas funções podem seguir a convenção diferente de retornar um ponteiro se forem bem-sucedidas e retornar NULL caso contrário.

Caso um erro seja encontrado, devemos interromper imediatamente o nosso código e retornar falso. Se uma função invocada retornar falso indicando erro, a função que a chamou também deve interromper a execução. Além disso, devemos marcar dentro da estrutura da metafonte que um erro ocorreu, preenchendo informações adicionais que permitirão posteriormente que uma mensagem de diagnóstico seja impressa se o usuário assim desejar.

O ato de preencher as informações sobre o erro na estrutura da fonte será feita com a ajuda de macros. Por exemplo: `RAISE_NO_MEMORY`. Quase todas as macros de erros irão receber como entrada a estrutura da fonte (para preencherem ali dados sobre o erro), informações de contexto ou NULL (caso precisem ler informações adicionais sobre a execução) e opcionalmente o número da linha em que o erro foi encontrado. A depender do erro, mais informações podem ser colocadas na macro.

Para evitar ter que definirminuciosamente muitos tipos diferentes de erro no corpo do texto, tirqndo assim o foco dos algoritmos e da lógica sendo implementada, vamos definir os erros e suas macros apenas no Apêndice A.

3. Inicialização e Finalização

Primeiro vamos definir a função de inicialização. O que ela fará será armazenar em uma série de variáveis estáticas ques erão usadas pelas outras funções, além de nos informar a medida de densidade de pixels da tela em DPI. As variáveis que armazenarão tais informações são:

Seção: Variáveis Locais (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;
```

E a função de inicialização consiste em preencher tais variáveis:

Seção: Definição de Funções da API (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;
```

<Seção a ser Inserida: Inicialização WeaveFont>

```
  return true;
}
```

A função de finalização também existe e à medida que adicionarmos mais coisas à inicialização, poderemos precisar adicionar código correspondente à finalização:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
void _Wfinish_weavefont(void){
                <Seção a ser Inserida: Finalização WeaveFont>
}
```

Coisas como variáveis globais e informações que uma meta-fonte precisa à longo prazo serão sempre alocadas com a alocação permanente. Coisas como variáveis locais dentro de uma definição de como renderizar um caractere serão alocadas pela alocação temporária. Se o gerenciador de memória distinguir entre estes dois tipos de alocação (como ocorre no Motor Weaver), estas serão duas funções diferentes. Na maioria dos casos, será a mesma função, possivelmente um `malloc`. Caso em que ambas as funções de desalocação também serão um `free`. Já no Motor Weaver, não há funções de desalocação: a alocação permanente é desalocada quando o laço de jogo atual ("game loop") termina e a temporária é desalocada a cada frame.

Como esperamos que estas funções sejam chamadas apenas uma vez por programa, não precisamos usar mutex nelas.

4. Analisador Léxico

4.1. Tipos de Tokens

A primeira coisa a criar para uma linguagem é seu analisador léxico. Ele irá ler o código-fonte da linguagem presente em um arquivo e irágerar uma lista de "tokens", sendo cada token a unidade mais básica da linguagem, basicamente uma palavra.

WeaveFont reconhece o seguintes tipos de tokens: numéricos, strings, simbólicos de variável, simbólico de começo de loop, simbólico de fim de loop e simbólico genérico. Os simbólicos genéricos se subdividem em muitos outros tipos, um para cada palavra reservada da linguagem. Aqui é onde definimos os tipos principais e onde vamos mais adiante armazenar todos os subtipos genéricos:

Seção: Estrutura de Dados Locais (metafont.c):

```
enum { // Tipos de Tokens
  TYPE_NUMERIC = 1, TYPE_STRING, TYPE_SYMBOLIC, TYPE_FOR, TYPE_ENDFOR,
  // Os tipos básicos (numérico, string, varivel, começo e fim de
  // loop) estão acima. Os outros serão colocados logo abaixo:
        <Seção a ser Inserida: WeaveFont: Definição de Token Simbólico>
  // E um último tipo que não deve ser usado, exceto para indicar erro:
  TYPE_INVALID_TOKEN
};
```

Um token numérico será representado internamente como um número em ponto flutuante. Nisso iremos diferir do METAFONT que usava uma representação própria de números em ponto fixo. Esta escolha nos permitirá realizar operações sobre números mais rapidamente, graças ao suporte de hardware. Essa é a aparência de um token numérico:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
struct numeric_token{
  int type;   // Deve ser igual a 'TYPE_NUMERIC'
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  float value;
};
```

E isso é como representaremos tokens de strings:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
struct string_token{
  int type;   // Deve ser igual a 'TYPE_STRING'
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  char value[4];
  // Ponteiro para o glifo que a string representa (tratado na Seção 15.1)
  struct _glyph *glyph;
};
```

Nós iremos armazenar apenas os primeiros 4 bytes de cada string que encontrarmos, mesmo que no código-fonte a string seja maior. Isso será feito porque ao contrário do METAFONT original, a única utilidade para strings aqui será indicar qual caractere Unicode deve estar associado a cada glifo definido. Para isso nós precisamos de apenas 4 bytes.

Já no caso de tokens simbólicos de variáveis, precisamos armazenar um ponteiro para a variável em questão e o nome da variável:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
struct symbolic_token{
  int type;   // Deve ser igual a 'TYPE_SYMBOLIC'
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
  void *var;
  char *value;
};
```

Um token de começo de loop deve conter uma variável booleana que nos diz se ele já começou a ser executado, um ponteiro para um número de ponto flutuante (que controla a iteração do loop), um número em ponto flutuante de incremento (que iremos somar à variável de controle a cada iteração) e a condição de parada (um último número em ponto flutuante representando o valor que se ultrapassado pela variável de controle, interrompe o loop). Por fim, o token também irá conter um ponteiro para o fim do loop, para que seja fácil sair de dentro dele quando ele terminar:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

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

Há também alguns tokens simbólicos que precisam como informação adicional armazenar um ponteiro para outro token, seja ele qual for. É o caso do token que representa um `endfor` (fim de uma iteração): ele deve apontar para o começo da iteração de modo que seja eficiente voltar para o começo dela. Mas também ocorre com o `if`, `elseif`, `else` e `beginchar`, que devem conter um ponteiro para outro token de modo a facilitar o deslocamento pelo código durante a interpretação. Para tokens assim, usamos a seguinte estrutura de dados:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
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;
};
```

Já para os tokens simbólicos genéricos, toda a informação que precisamos sobre eles é o seu sub tipo. Podemos então representar eles com a estrutura abaixo. Quando não sabemos o tipo de um token, podemos também usar a estrutura abaixo, pois os campos que existem nela estão também presentes em todos os outros tipos de tokens:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
struct generic_token{
  int type;   // Um sub-tipo de token simbólico.
  struct generic_token *next;
#if defined(W_DEBUG_METAFONT)
  int line;
#endif
};
```

Precisamos apenas definir para eles um subtipo de token que indique o quê eles são:

Seção: WeaveFont: Definição de Token Simbólico:

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

Podemos definir posteriormente muitos outros tipos de tokens simbólicos reservados. Note que pelas regras da enumeração em C que usamos, qualquer token cujo tipo é um número igual ou maior que 3 é simbólico. E se o número for maior que 5, será um token simbólico genérico.

Todo token possui um ponteiro para um próximo token. Isso ocorre porque eles geralmente formarão uma lista encadeada de tokens.

Tokens são gerados pela função de alocação permanente. Precisamos mantê-los na memória, já que podemos ter que interpretá-los muitas vezes, cada vez que formos renderizar um novo caractere. Justamente por isso, não queremos armazenar neles mais do que é necessário. Por exemplo, o número de linha em que eles estão é útil para indicar onde um erro foi localizado no código-fonte. Esta é uma variável que certamente será usada por um projetista de fontes ou por quem está criando código WeaveFont. Mas para a maioria dos usuários, depurar erros não será algo que eles farão. Por causa disso, só armazenamos tal informação se estivermos em modo de depuração. E pelo mesmo motivo não nos preocupamos em armazenar strings muito grandes.

Para desalocar a memória ocupada por uma lista de tokens e apagá-los, pode-se usar a seguinte função:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. A Análise Léxica

Agora a função que fará a análise léxica. Ela receberá como argumento a estrutura da metafonte (cujos detalhes ainda vamos definir) e uma string com caminho para arquivo com código-fonte METAFONT. Ela produzirá uma lsita de tokens e armazenará no ponteiro indicado por seus dois últimos argumentos o primeiro e o último token lido. A função retornará verdadeiro se nenhum erro ocorrer, e falo caso contrário.

Seção: Funções Auxiliares Locais (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;
    }
                <Seção a ser Inserida: Análise Léxica: Regra 1>
                <Seção a ser Inserida: Análise Léxica: Regra 2>
                <Seção a ser Inserida: Análise Léxica: Regra 3>
                <Seção a ser Inserida: Análise Léxica: Regra 4>
                <Seção a ser Inserida: Análise Léxica: Regra 5>
                <Seção a ser Inserida: Análise Léxica: Regra 6>
    {
      // Nenhuma regra aplicada: erro.
      char unknown_char[5];
      memset(unknown_char, 0, 5);
      unknown_char[0] = c;
      if(!fgets(&(unknown_char[1]), 4, fp))
        unknown_char[1] = '\0';
      RAISE_ERROR_INVALID_CHAR(mf, NULL, line, unknown_char);
      return false;
    }
    free_token_list(first_token);
    *first_token = NULL;
    *last_token = NULL;
    return false;
  }
```

<Seção a ser Inserida: Análise Léxica: Detecção Final de Erros>

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

Como usamos o tipo `FILE`, precisamos inserir o cabeçalho de entrada e saída padrão para podermos abrir e ler arquivos:

Seção: Cabeçalhos Locais (metafont.c):

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

As regras para ler os tokens consiste em ler cada linha do código fonte aplicando as seguintes regras para cada novo caractere lido:

1) Se o próximo caractere é um espaço, tabulação ou um ponto que não é sucedido por um dígito decimal ou um novo ponto, ignore-o e siga em frente.

Seção: Análise Léxica: Regra 1:

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

Como estamos usando a função `isdigit`, devemos inserir o seguinte cabeçalho que a declara:

Seção: Cabeçalhos Locais (metafont.c):

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

2) Se o próximo caractere é o símbolo de porcentagem, ignore-o e ignore todos os outros caracteres seguintes na linha. Este símbolo marca o começo de comentários na linguagem.

Seção: Análise Léxica: Regra 2:

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

3) Se o próximo caractere é um dígito decimal ou um ponto seguido por um dígito decimal, então o próximo token será numérico. E é formado pela maior sequência possível de ser lida envolvendo dígitos decimais e opcionalmente um único ponto representando o ponto decimal.

Seção: Análise Léxica: Regra 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, line); // Vamos definir erros no Apêndice 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) Se o próximo caractere é uma aspa dupla, então o próximo token deve ser uma string e será composto pela sequência de caracteres até a próxima aspa dupla não-precedida por uma contrabarra, que deve estar na mesma linha. Se existir uma aspa dupla iniciando uma string, mas não existir outra para finalizar a string na mesma linha, um erro será gerado. Ao encontrar uma primeira contrabarra, ela é ignorada, mas o caractere a seguir é considerado sempre. Esta regra permite que representemos aspas duplas dentro de string (na forma `"\""`) e uma contrabara dentro de string (na forma `"\\"`).

Seção: Análise Léxica: Regra 4:

```
if(c == 34){ // 34: ASCII para aspas duplas
  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) Se o próximo caractere for um parêntese, ponto-e-vírgula ou uma vírgula, então o próximo token será um token simbólico formado por este único caractere.

Seção: Análise Léxica: Regra 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) Caso contrário, o próximo token será simbólico e formado pela sequência mais longa possível

formada somente por caracteres que pertencem a uma dentre 12 famílias:

Seção: Análise Léxica: Regra 6:

```
{
  char buffer[256];
  int i = 0;
  buffer[0] = '\0';
  // Buffer é lido de acordo com as subregras 6-a a 6-l
                <Seção a ser Inserida: Análise Léxica: Regra A>
                <Seção a ser Inserida: Análise Léxica: Regra B>
                <Seção a ser Inserida: Análise Léxica: Regra C>
                <Seção a ser Inserida: Análise Léxica: Regra D>
                <Seção a ser Inserida: Análise Léxica: Regra E>
                <Seção a ser Inserida: Análise Léxica: Regra F>
                <Seção a ser Inserida: Análise Léxica: Regra G>
                <Seção a ser Inserida: Análise Léxica: Regra H>
                 <Seção a ser Inserida: Análise Léxica: Regra I>
                 <Seção a ser Inserida: Análise Léxica: Regra J>
                 <Seção a ser Inserida: Análise Léxica: Regra K>
                 <Seção a ser Inserida: Análise Léxica: Regra L>
  // De acordo com conteúdo do buffer, gera o próximo token
    <Seção a ser Inserida: Análise Léxica: Gera Token de Controle de Fluxo>
     <Seção a ser Inserida: Análise Léxica: Gera Token Simbólico Reservado>
      <Seção a ser Inserida: Análise Léxica: Gera Token Simbólico Genérico>
}
```

a) A primeira família de letras são as letras de A a Z maiúsculas e minúsculas além do "underline" e dígitos. Um dígito não pode ser o primeiro caractere da sequência, ou ele seria interpretado como um token numérico.

Seção: Análise Léxica: Regra 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) A segunda família são os símbolos de maior, menor, igual, dois pontos e a barra vertical.

Seção: Análise Léxica: Regra 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) Acento agudo e grave.

Seção: Análise Léxica: Regra C:

```
else if(c == '`' || c == '\''){
```

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

d) Mais e menos.

Seção: Análise Léxica: Regra D:

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

e) Barra, barra invertida e símbolo de multiplicação.

Seção: Análise Léxica: Regra 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) Ponto de interrogação e exclamação.

Seção: Análise Léxica: Regra F:

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

g) Cerquilha, “e” comercial, arroba e cifrão.

Seção: Análise Léxica: Regra 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) Acento circunflexo e til.

Seção: Análise Léxica: Regra H:

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

i) Abrir colchetes.

Seção: Análise Léxica: Regra I:

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

j) Fechar colchetes.

Seção: Análise Léxica: Regra J:

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

k) Abrir e fechar chaves.

Seção: Análise Léxica: Regra K:

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

l) Ponto.

Seção: Análise Léxica: Regra L:

```
else if(c == '.'){
  do{
    buffer[i] = c;
```

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

Depois de ler todos os caracteres do próximo token, primeiro vemos se estamos diante de um token simbólico reservado. Estes são tokens simbólicos com o que seriam palavras-chave da linguagem. Podemos identificá-los porque iremos armazenar uma lista de palavras-chave reservadas da linguagem aqui:

Seção: Variáveis Locais (metafont.c) (continuação):

```
static char* list_of_keywords[] = {
            <Seção a ser Inserida: Lista de Palavras Reservadas>
  NULL};
```

Sendo assim, sabemos que temos um token que corresponde a uma palavra-chave reservada se ele estiver presente nesta lista terminada em NULL. Se for o caso, geramos um token especial com um tipo dependendo de sua posição na lista:

Seção: Análise Léxica: Gera Token Simbólico Reservado:

```
{
  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; // Palavras reservadas: após ';'
  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;
  }
}
```

Para usar a função `strcmp`, precisamos do seguinte cabeçalho:

Seção: Cabeçalhos Locais (metafont.c):

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

Os tokens que realizam controle de fluxo como `for`, `endfor`, `if`, `elseif`, `else` e `fi` também são tokens formados por palavras reservadas. Mas eles terao precedência sobre os outros

e a geração deles será um pouco diferente. Isso porque muitos destes tokens devem ter ponteiros inicializados que depois permitirão navegar mais facilmente pelo código caso eles mudem a ordem sequencial na qual os comandos da linguagem são interpretados.

No caso do `for`, quando o geramos também geramos o token `endfor` correspondente, fazendo com que o token `for` possua um ponteiro para `endfor` e o `endfor` um ponteiro para o token que vem antes do `for`. Isso ajudará nosso interpretador a navegar de forma mais rápida diante das operações de controle de fluxo. O token do `endfor` é então guardado em uma pilha para ser usado depois.

Seção: Análise Léxica: Gera Token de Controle de Fluxo:

```
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;
  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;
  new_token -> end = endfor_token;
  endfor_token -> type = TYPE_ENDFOR;
  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;
```

```
}
```

Quando finalmente encontrarmos no código um "endfor", podemos enfim usar o token armazenado na pilha ao invés de termos que gerar um novo. Se não existir tal token, devemos gerar um erro (que definiremos posteriormente):

Seção: Análise Léxica: Gera Token de Controle de Fluxo (continuação):

```
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;
}
```

Se lemos um token `if`, da mesma forma, criaremos um token `fi` provisório e o empilharemos. Mas este será um `fi` temporário, não o que será depois inserido na lista de tokens que forma o programa. Isso porque este token de finalização, ao contrário de um `fi` comum, terá um ponteiro para o `if` que o gerou, e este ponteiro será atualizado para o próximo `elseif` e `else` no mesmo nível de aninhamento. Este ponteiro nos ajudará a atualizar o último `if`, `elseif` e `else` para apontarem para o próximo token de controle de fluxo condicional depois deles.

Esta é então a criação de um `if`:

Seção: Análise Léxica: Gera Token de Controle de Fluxo:

```
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;
}
```

Se encontrarmos um `elseif` ou um `else` no mesmo nível de aninhamento, consultamos o ponteiro do `fi` empilhado, fazemos com que o token encontrado nele (que deve ser um `if` ou um `elseif`) aponte para o `elseif` ou `else` atual, criamos um novo token para o novo `elseif` ou `else` a ser inserido no programa e atualizamos o ponteiro do `fi` para que aponte para o token recém-criado:

Seção: Análise Léxica: Gera Token de Controle de Fluxo (continuação):

```
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;
}
```

Finalmente, ao encontrar um `fi`, devemos remover da pilha o token `fi` que contém um link para o último `if`, `elseif` ou `else`. Mas antes inicializando o ponteiro de ligação do token referenciado por tal link:

Seção: Análise Léxica: Gera Token de Controle de Fluxo (continuação):

```
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;
}
```

Outro token que consideramos como controle de fluxo é o `beginchar`, que para ser inicializado corretamente precisa ter um ponteiro para o `endchar` correspondente. Tratamos ele da mesma forma que fizemos com o `if` para garantir isso, criando um `endchar` provisório que será empilhado e conterá um ponteiro para o último `beginchar` criado. De fato, não é possível aninhar composições de `beginchar`, então só poderá haver no máximo um destes tokens empilhados:

Seção: Análise Léxica: Gera Token de Controle de Fluxo (continuação):

```
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){ // Sem 'beginchar' aninhados
      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;
}
```

E quando lemos um `endchar`, desempilhamos o token e inicializamos o ponteiro do último `beginchar` correspondente que está armazenado nele:

Seção: Análise Léxica: Gera Token de Controle de Fluxo (continuação):

```
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;
}
```

Caso aconteça de terminarmos a análise léxica com algum token de controle de fluxo empilhado, isso significa que um `for` ou `if` foi inicializado, mas não terminado. Neste caso, só acusamos um erro (que definiremos depois):

Seção: Análise Léxica: Detecção Final de Erros:

```
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;
}
```

Se não for um token simbólico reservado, e se existir algo copiado para o nosso buffer, então geramos um novo token simbólico genérico. Este token será então uma variável:

Seção: Análise Léxica: Gera Token Simbólico Genérico:

```
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;
  // Se tiver o nome de variável interna, podemos ajustar já seu ponteiro
  // para  o local em que seu conteúdo está:
```

<Seção a ser Inserida: Preenche Ponteiro para Variável Interna>

```
#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. Programas WeaveFont

5.1. Executando Programas

Quando avaliamos um programa WeaveFont, vamos precisar de duas extruturas adicionais. A primeira delas, a qual chamaremos de `struct metafont` irá conter as informações finais da meta-fonte tipográfica que foi lida e que terá tudo que for necessário para que tal fonte possa ser usada para renderizar cada um dos glifos da fonte. A segunda estrutura, a qual chamaremos de `struct context` representa o estado atual do analisador sintático e representa informações que devemos saber para poder continuar interpretando corretamente uma lista de tokens. Essa segunda estrutura pode ser descartada depois de terminarmos de ler os tokens de nossa fonte.

Em suma, para cada arquivo com código-fonte WeaveFont, iremos gerar um único `struct metafont` com informações da fonte. Mas cada vez que formos executar código deste arquivo (quando lemos ele pela primeira vez, e depois uma vez para cada caractere a ser renderizado), criaremos un novo `struct context`, que será descartado após a execução.

A primeira coisa que nosso analisador sintático precisa saber é que todo programa WeaveFont é uma sequência de instruções:

```
<Programa> -> <Lista de Instruções>
```

Na linguagem METAFONT era necessário indicar o fim de um programa com a instrução `end`. Mas para a linguagem WeaveFont isso não é necessário, pois ela assume que todo o código-fonte está contido em um único arquivo. Portanto, o fim do arquivo é o fim do código

A primeira função de análise sintática que definiremos é a que reconhece um programa inteiro. Ela recebe uma lista de tokens a ser executada, representada pelo primeiro e último elemento. Se a lista for nula, apenas retornamos: não fazer nada é a execução correta de um programa vazio. Se existir, passa para a próxima função que irá separar essa lista em instruções individuais e irá executar cada uma delas. Por fim, retornamos se a execução foi bem-sucedida ou não.

Seção: Funções Auxiliares Locais (metafont.c):

```
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;
  // Código adicional a ser definido futuramente:
            <Seção a ser Inserida: Após Execução de Programa>
  return true;
}
```

A estrutura de contexto ainda não será definida inteiramente. Iremos mostrar o conteúdo dela à medida que for necessário. Por hora, a única variável que mostraremos será a styring que armazena qual caractere UTF-8 está sendo renderizado no momento, ou que armazena a string vazia se não houver nenhum:

Seção: Estrutura de Dados (metafont.h):

```
struct context{
  char current_character[5];
            <Seção a ser Inserida: Atributos (struct context)>
};
```

Já a estrutura da meta-fonte, por hora vamos indicar três variáveis dentro dela. A primeira é um mutex, como descrito na Subseção 2.2. Depois temos uma string com o nome do arquivo de onde o código da fonte foi lida. A última indica se a fonte já terminou de carregar ou se ainda está em carregamento. Uma fonte em carregamento está executando código WeaveFont, mas ainda não terminou de processar todo o arquivo até o final pela primeira vez. Ela ainda não sabe quais caracteres ela pode ou não renderizar. Já se ela não está em modo de carregamento,ela já terminou de processar o arquivo com o código e está pronta para renderizar imagens.

Seção: Estrutura de Dados (metafont.h) (continuação):

```
struct metafont{
                  <Seção a ser Inserida: Declarar um Mutex>
  char *file;
  bool loading;
            <Seção a ser Inserida: Atributos (struct metafont)>
};
```

Ambas as estruturas de dados terão uma função que as inicializa e as finaliza (a função que finaliza a estrutura metafont, `_Wdestroy_metafont`, já foi declarada na Seção 1, quando listávamos o cabeçalho das funções que seriam exportadas para o usuário):

Seção: Declaração de Função Local (metafont.c) (continuação):

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

A estrutura `metafont` precisa armazenar mais informações, e também pode tanto ser alocada com a função permanente ou temporária de alocação. Já o contexto sempre será temporário, e por isso sempre será alocado e desalocado com as funções temporárias de memória.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
  // Copiando o nome do arquivo com o código
  size_t filename_size = strlen(filename) + 1;
  mf -> file = (char *) permanent_alloc(filename_size);
```

```
  memcpy(mf -> file, filename, filename_size);
  // Iniciamos no modo de carregamento
  mf -> loading = true;
  // Mais código a ser definido futuramente:
            <Seção a ser Inserida: Inicialização (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); // Vamos definir erros no Apêndice A
    return NULL;
  }
  cx -> current_character[0] = '\0';
  // A ser definido futuramente:
            <Seção a ser Inserida: Inicialização (struct context)>
  return cx;
}
```

Já a definição das funções que finalizam e desalocam as estruturas:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
void _Wdestroy_metafont(struct metafont *mf){
  if(permanent_free != NULL){
    permanent_free(mf -> file);
    // A ser definido no futuro:
              <Seção a ser Inserida: Finalização (struct metafont)>
    MUTEX_DESTROY(mf -> mutex);
    permanent_free(mf);
  }
}
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
void destroy_context(struct context *cx){
  if(temporary_free != NULL){
    // A ser definido no futuro:
              <Seção a ser Inserida: Finalização (struct context)>
    temporary_free(cx);
  }
}
```

Conforme foi comentado, a variável booleana `loading` dentro da `struct metafont` deve ser mudada para falso depois que o programa inteiro é avaliado e executado pela primeira vez. Nesta mesma Subseção mostramos o código que é executado para ler o programa pela primeira vez (função `eval_program`). Só precisamos então adicionar depois dele código para atualizar esta variável:

Seção: Após Execução de Programa:

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

5.2. Separando Lista de Instruções em Instruções Individuais

Uma lista de instruções é uma sequência vazia, ou uma sequência de instruções vazias, simples e compostas que podem se alternar em qualquer ordem:

```
<Lista de Instruções> -> <Vazio> |
```

```
<vazio> ; <Lista de Instruções> |
<Instrução Simples> ; <Lista de Instruções> |
<Instrução Composta> <Lista de Instruções>
```

Uma instrução simples e vazia sempre deve ser terminada por um ponto-e-vírgula. Já instruções compostas não precisam de ponto-e-vírgula como delimitador, pois todas elas possuem um conjunto de tokens de abertura no começo e um outro conjunto de tokens de fechamento que as delimita. Elas podem ser instruções bastante longas e podem conter dentro delas outras listas de instruções. Por exemplo, um `if` e todo o código que é executado condicionalmente dentro de seu corpo faz parte de uma mesma instrução composta.

Para realizar a análise sintática correta de nossos programas, nós devemos ser capazes de identificar cada uma das instruções individuais, mesmo quando não é tão simples encontrar como elas são delimitadas. Contudo, por enquanto, nesta Subseção nós iremos escrever código adotando uma heurística mais simples provisória: vamos assumir que todas as instruções são delimitadas ou por um ponto-e-vírgula ou pelo final da lista de tokens.

A função que fará isso é:

Seção: Declaração de Função Local (metafont.c):

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

O objetivo desta função é separar cada uma das instruções individuais e passar cada uma delas para uma próxima função que avalia uma única expressão. Para isso, ela assumirá que pode separar instruções meramente separando elas pelo ponto-e-vírgula. O seu código é:

Seção: Funções Auxiliares Locais (metafont.c):

```
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;
```

<Seção a ser Inserida: Antes de Avaliar Código>

```
  while(begin != NULL){
    // Este laço pula instruções vazias e posiciona 'begin' no começo da
    // próxima instrução não-vazia:
    while(begin != NULL && begin -> type == TYPE_SEMICOLON){
      if(begin != end_list)
        begin = begin -> next;
      else
        begin = NULL;
    }
    end = begin;
    // Este laço acha o próximo token antes de um ';'e posiciona 'end' nele:
    if(end != NULL){
      while(end != end_list && end -> next -> type != TYPE_SEMICOLON)
        end = end -> next;
    }
    // Se encontramos algo, avaliamos uma instrução individual:
    if(begin != NULL){
      if(!eval_statement(mf, cx, begin, &end))
        return false;
      // E após avaliar a instrução, posiciona 'begin' no ';'
      if(end != end_list)
        begin = end -> next;
      else
        begin = NULL;
```

```
    }
  }
```
<Seção a ser Inserida: Depois de Avaliar Código>
```
  return true;
}
```

A função acima basicamente percorre uma lista de tokens até o final. A invariante do laço `while` mais externo é que quando iniciamos ele, o ponteiro `begin` está sempre ou em um ponto-e-vírgula ou no começo da próxima instrução a ser executada. No primeiro caso, vamos avançando a posição desse ponteiro ignorando os pontos-e-vírgulas: elas são instruções vazias e portanto podem ser ignoradas. Depois que `begin` estiver no começo de instrução não-vazia, posicionamos então `end` no próximo token que preceder um ponto-e-vírgula. Assim, os dois tokens delimitam perfeitamente a instrução individual a ser executada, a qual passamos para a função `eval_statement`. Depois que esta função avalia a instrução, se não ocorreu erro, fazemos com que `begin` seja posicionado após o `end`, que precede um ponto-e-vírgula. Sendo assim, na próxima iteração o `begin` estará em um ponto-e-vírgula e o invariante do laço é mantido.

Ou ao menos, a função acima pode funcionar perfeitamente na ilusão de que é isso que ocorre. Isso porque o valor de `end` é passado por referência, não por valor para a função `eval_statement`. Isso quer dizer que o valor dele pode mudar. E isso de fato pode acontecer: quando uma instrução composta é executada, o ponteiro pode ser movido para um valor anterior ou posterior, dependendo de qual for a instrução e de como ela é executada. De qualquer forma, toda vez que o ponteiro for movido será para o token que precede a próxima instrução a ser executada. Desta forma, a função `eval_list_of_statements` fica bastante simples, com a ilusão de simplicidade mantida pelo trabalho da função `eval_statement`, que iremos definir nas próximas Seções.

6. Uma Instrução Composta: O Bloco Composto

6.1. Suportando Instruções Compostas

Vimos na Seção anterior que existem instruções compostas que podem ser formadas por muitas outras instruções. As regras gramaticais para elas é:

```
<Instrução Composta> -> <Bloco Composto> |
                        <Bloco Condicional> |
                        <Bloco de Iteração> |
                        <Declaração de Caractere>
<Bloco Composto> -> begingroup <Lista de Instruções> endgroup
<Bloco Condicional> -> if <Expressão Booleana> :
                         <Lista de Instruções>
                         <Alternativas>
                       fi
<Bloco de Iteração> -> for <Cabeçalho For>:
                         <Lista de Instruções>
                       endfor
<Definição de Caractere> -> beginchar <Descrição de Caractere>
                              <Lista de Instruções>
                            endchar
```

Pelas regras acima, as instruções compostas são qutro: podem ser blocos (começam com `begingroup` e terminam com `endgroup`), condicionais (começam com `if` e terminam com `fi`), iterações (comaçam com `for` e terminam em `endfor`) ou definição de caractere (começam com `beginchar` e terminam com `endchar`).

Como podem haver listas de instruções dentro de cada instrução composta, então várias instruções compostas podem também estar aninhadas, uma dentro da outra. Mas cada instrução composta deve ser iniciada e finalizada na ordem correta. Se o último começo de instrução composta que lemos foi um `begingroup`, mas pouco depois encontramos um `fi` ou um `endchar`, o programa não está correto.

Para ler corretamente instruções compostas, devemos sevar em conta os tokens que as delimitam, definidos abaixo:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

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

E adicionamos eles à nossa lista de palavras reservadas:

Seção: Lista de Palavras Reservadas:

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

Os tokens que começam e terminam uma iteração são `TYPE_FOR` e `TYPE_ENDFOR`. Eles já foram definidos anteriormente, junto ao nosso analisador léxico. Tivemos que tratar eles antes porque os seus tokens precisam armazenar uma quantidade maior de informações e por isso precisam de uma inicialização mais complexa na etapa léxica.

E agora a função que irá avaliar uma instrução individual.

Seção: Declaração de Função Local (metafont.c) (continuação):

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

A função recebe o começo e o fim da lista de tokens que forma uma instrução individual. Para instruções simples, ambos os ponteiros estão corretos. Para instruções compostas, que não são delimitadas por ponto-e-vírgula como vimos na seção anterior, o ponteiro de começo está correto, mas o de final não. Cabe então à função `eval_statement` corrigir esse ponteiro final nestes casos.

Seção: Funções Auxiliares Locais (metafont.c):

```
bool eval_statement(struct metafont *mf, struct context *cx,
                    struct generic_token *begin, struct generic_token **end){
                  <Seção a ser Inserida: Instrução: Composta>
              <Seção a ser Inserida: Instrução: Declaração de Variável>
                  <Seção a ser Inserida: Instrução: Atribuição>
                  <Seção a ser Inserida: Instrução: Comando>
  // Se estamos aqui, não identificamos o tipo de instrução:
  RAISE_ERROR_UNKNOWN_STATEMENT(mf, cx, OPTIONAL(begin -> line));
  return false;
}
```

A função acima irá testar se está diante de uma instrução composta ou simples. Pode ser que nenhuma das opções seja identificada. Neste caso, devemos gerar um erro de instrução não reconhecida.

Agora vamos às instruções composta. A primeira coisa a lembrarmos é que será importante para a linguagem saber em qual nível de aninhamento estamos para saber o escopo de eventuais variáveis declaradas. Por causa disso, o contexto de nossa análise deve memorizar o nível de aninhamento atual. Cada início de instrução composta (`begingroup`, `if`, `beginchar`e `for`) aumenta esse nível em 1 e cada fim de instrução composta (`endgroup`, `fi`, `endchar` e `endfor`) diminui o nível em 1. Também devemos saber qual o token que criou o último nível de aninhamento caso estejamos aninhados em uma instrução composta e devemos armazenar uma cópia do token esperado que o finalizará em uma pilha. Esta cópia deve possuir um link para o começo da instrução composta caso seja necessário. Declaramos o nível de aninhamento atual e a pilha de tokens esperados de finalização:

Seção: Atributos (struct context):

```
  int nesting_level;
```

```
  struct linked_token *end_token_stack;
```

O nível de aninhamento deve ser inicializada como zero e a pilha de tokens de finalização deve ser vazia inicialmente:

Seção: Inicialização (struct context):

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

Para facilitar a escrita de código que lida com estes níveis de aninhamento, vamos criar duas funções auxiliares. Ambas recebem um `struct context` com o contexto de execução atual e um token. A primeira deve aumentar em 1 o nível de aninhamento dado um token que começa uma instrução composta e a segunda deve diminuir dado um token que finaliza um nível de aninhamento. A primeira deve retornar erro se receber um token incorreto que não inicia uma instrução composta e a segunda também vai gerar erro se o token passado não for o correto que finaliza o nível de aninhamento atual:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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);
```

A implementação da primeira função:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  }
  // Somente 4 tipos de tokens serão passados para esta função:
  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;
}
```

E a implementação da segunda função:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Note que se um erro ocorrer ao interpretar código quando estamos no meio de uma função composta, nós podemos interromper a execução antes de executarmos o `end_nesting_level`. Por causa disso, ao finalizar um contexto, devemos remover e finalizar qualquer token existente na pilha de tokens de finalização esperados:

Seção: Finalização (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. A Instrução `begingroup`...`endgroup`

Como exatamente uma instrução composta é avaliada? Primeiro lembre-se que a função `eval_list_of_statements` separa o código em partes usando o ponto-e-vírgula como delimitador. Em seguida, ele passa cada parte para a função `eval_statement`. Conforme comentado, esse comportamento é um pouco ingênuo, pois instruções compostas são instruções que podem possuir outras instruções e ponto-e-vírgula dentro delas. Considere a seguinte lista de tokens:

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

Nessa sequência, os três primeiros tokens seriam passados para serem avaliados. depois, seria o `[T3][T4]`. E por fim, o `[endgroup]`. Mas vamos fazer as coisas um pouco diferentes. Quando recebermos um `[begingroup][T5]`, devemos avaliar só ele iniciando um novo aninhamento. Os dois outros tokens recebidos serão devolvidos, para serem avaliados logo em seguida. Fazemos isso mudando a posição do ponteiro que marca o fim da instrução atual, conforme discutido na Subseção 6.1.

Quando recebermos o `endgroup`, faremos o mesmo: interpretaremos só ele, marcando o fim da nstrução composta e devolvemos o token que vem a seguir para ser interpretado depois.

O código para tratar então qualquer declaração que comece com `begingroup` é:

Seção: Instrução: Composta:

```
if(begin -> type == TYPE_BEGINGROUP){
  begin_nesting_level(mf, cx, begin);
  // Para que o começo da próxima instrução venha depois de 'begingroup':
  *end = begin;
  return true;
}
```

Da mesma forma, se recebermos uma função que começa com `endgroup`, nós tratamos só este primeiro token encerrando o aninhamento do bloco:

Seção: Instrução: Composta (continuação):

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

E se nós finalizarmos um programa sem finalizar uma instrução composta? Este tipo de erro geralmente é detectado durante a análise léxica. Exceto pelo caso em que começamos um `begingroup` e não terminamos. Para este caso, vamos adicionar a detecção de erro aqui:

Seção: Após Execução de Programa:

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

7. Declaração de Variáveis

A declaração de variáveis é a primeira das instruções simples que veremos. A sintaxe para declarar variáveis é:

```
<Instrução Simples> -> <Declaração de Variável> | ...
<Declaração de Variável> -> <Tipo> <Lista de Declaração>
<Tipo> -> boolean | string | path | pen | picture | transform | pair |
          numeric
<Lista de Declaração> -> <Tag> | <Tag> , <Lista de Declaração>
```

Uma "tag" é basicamente um token simbólico sem um significado pré-definido na linguagem. Por exemplo, "`tag`" é uma tag, mas "`begingroup`" não é.

Para interpretar a declaração de variáveis, vamos então introduzir no nosso analisador léxico os seguintes tokens especiais:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_T_BOOLEAN,              // O token simbólico 'boolean'
TYPE_T_PATH,                 // O token simbólico 'path'
TYPE_T_PEN,                  // O token simbólico 'pen'
TYPE_T_PICTURE,              // O token simbólico 'picture'
TYPE_T_TRANSFORM,            // O token simbólico 'transform'
TYPE_T_PAIR,                 // O token simbólico 'pair'
TYPE_T_NUMERIC,              // O token simbólico 'numeric'
```

E adicionamos todos à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

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

Quando uma variável é declarada, devemos fazer duas coisas:

1) Devemos alocar uma estrutura da memória para armazenar seu conteúdo. A variável não pode ter seu nome conflitando com algumas variáveis especiais da linguagem como: `w`, `h`, `d`, `currentpicture` ou `currentpen`. Se for o caso, geramos um erro.A variável é inicialmente preenchida com um valor padrão que depende de seu tipo. Ela não poderá ser usada antes de ser inicializada.

2) Devemos percorrer a lista de tokens e procurar por ocorrências desta variável mais adiante no mesmo nível de aninhamento, ou em aninhamento inferior. E devemos atualizar o ponteiro destes tokens para que apontem para a região de memória recém-alocada para esta variável.

Como cada variável possui diferentes informações e conteúdo, dependendo de seu tipo, o modo pelo qual cada uma delas é criada pode diferir. O que todas as variáveis tem em comum é a seguinte estrutura:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
// Variável genérica
struct variable{
  int type;
  struct variable *next;
};
```

Todas armazenam primeiro o seu tipo e depois um ponteiro para a próxima. Dependendo do tipo da variável haverá mais informações após o ponteiro.

Se estivermos diante de uma variável global, podemos querer preservar e armazenar o nome da variável para o caso de querermos depois modificar o seu valor dado o seu nome. Então vamos usar a seguinte estrutura que armazena o nome e a variável:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

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

E a estrutura de dados da meta-fonte possui ponteiros para as variáveis globais com nome armazenado e também para as outras variáveis armazenadas nela:

Seção: Atributos (struct metafont):

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

Tais ponteiros para variáveis são inicializados como NULL:

Seção: Inicialização (struct metafont):

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

Para desalocar a lista de variáveis armazenadas na meta-fonte, basta percorrer a lista encadeada delas. Algumas variáveis mais complexas precisarão de operações adicionais para serem removidas, mas trataremos delas posteriormente. Aqui desalocamos as variáveis cujos nomes nós não armazenamos nem preservamos:

Seção: Finalização (struct metafont):

```
if(permanent_free != NULL){
  struct variable *v = (struct variable *) (mf -> variables);
  struct variable *next;
  while(v != NULL){
    next = (struct variable *) (v -> next);
        <Seção a ser Inserida: Finaliza Variável 'v' em 'struct metafont'>
    permanent_free(v);
```

```
    v = next;
  }
}
```

Já desalocar a lista de variáveis globais com nomes preservados é praaticamente a mesma coisa, só precisamos desalocar também a estrutura onde guardamos o nome delas:

Seção: Finalização (struct metafont) (continuação):

```
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);
        <Seção a ser Inserida: Finaliza Variável 'v' em 'struct metafont'>
    permanent_free(v);
    permanent_free(named);
    named = next;
  }
}
```

No caso das variáveis que forem declaradas dentro da especificação de um caractere a ser renderizado, elas serão armazenadas no contexto, não na estrutura da fonte. Afinal, a duração delas será sempre temporária:

Seção: Atributos (struct context):

```
  struct variable *variables;
```

A lista de variáveis é inicializada como vazia:

Seção: Inicialização (struct context) (continuação):

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

E finalizamos ela assim como finalizamos a lista de variáveis globais:

Seção: Finalização (struct context) (continuação):

```
if(temporary_free != NULL){
  struct variable *v = (struct variable *) (cx -> variables);
  struct variable *next;
  while(v != NULL){
    next = (struct variable *) (v -> next);
        <Seção a ser Inserida: Finaliza Variável 'v' em 'struct context'>
    temporary_free(v);
    v = next;
  }
}
```

Agora vamos à interpretação da declaração de variável. Toda vez que uma instrução começa com um tipo de variável, significa que estamos diante de uma declaração de variável. Só temos então que obter o tipo e o nome de cada variável, que estará separada por vírgulas quando há mais de uma sendo declarada. E para cada nome encontrado, criaremos a variável:

Seção: Instrução: Declaração de Variável:

```
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;
    }
```

<Seção a ser Inserida: Insere Variável Declarada>

```
    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){ // Erro: Terminou com ','
      RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, OPTIONAL((*end) -> line));
      return false;
    }
    variable = (struct symbolic_token *) (variable -> next);
  } while(variable != NULL);
  return true;
}
```

Criar e inserir uma variável no fragmento de código acima significa:

Seção: Insere Variável Declarada:

```
{
  void *variable_pointer;
  if(mf -> loading){ // Variável deve ser guardada em '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{ // Variável guardada em 'struct context'
    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);
}
```

A função que insere uma nova variável sem o nome é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Ela aloca uma nova variável e a coloca no local indicado por `target`, que será a lista encadeada `variables` dentro da estrutura da meta-fonte ou do contexto. No primeiro caso, a variável deve ser alocada pela função de alocação permanente, e no segundo, pela alocação temporária. Por isso passamos como argumento a função de alocação que deve ser usada:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
              <Seção a ser Inserida: Inicialização de Nova Variável>
  }
  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;
}
```

Já inserir uma variável nova com o nome é feito com a seguinte função:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Ela funciona de maneira similar, apenas alocando a estrutura do nome e colocando a nova variável ali. Como só preservamos o nome de variáveis globais, então certamente usaremos a

alocação permanente para elas:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

E finalmente, a função que percorre uma lista de tokens procurando por tokens simbólicos com o mesmo nome da variável alocada e fazendo o ponteiro deles apontar para onde a variável está:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

A função percorre todos os tokens que vem depois do token com o nome da nova variável e só para quando percorre a lista inteira ou quando sai do nível de aninhamento atual para um onde a variável não existe mais (quando acha um `endgroup`, `fi` ou `endchar` encerrando o nível de aninhamento da variável):

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. Variáveis Numéricas

Uma variável numérica será armazenada na seguinte estrutura:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

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

Comparando uma variável numérica com uma variável genérica, a diferença é a variável de ponto-flutuante `value` que armazena o valor numérico. Durante a criação da variável, o valor é inicializado como NaN, que para nós representa uma variável com valor desconhecido:

Seção: Inicialização de Nova Variável:

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

O uso da macro `NAN` requer que usemos o cabeçalho matemático:

Seção: Cabeçalhos Locais (metafont.c) (continuação):

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

No caso das variáveis numéricas, nada a mais é necessário durante a finalização, já que não há nada muito complexo além de um número em ponto flutuante nelas.

Entretanto, além de variáveis declaradas pelo usuário, teremos também algumas variáveis internas numéricas próprias. Elas sempre estarão presentes e não precisam ser declaradas. Temos dez delas que são numéricas: `pt`, `cm`, `mm`, `color_r`, `color_g`, `color_b`, `color_a`, `monospace`, `w`, `h` e `d`. Basicamente as três primeiras irão armazenar quantos pixels correspondem a 1pt, 1cm e 1mm respectivamente. A próxima indica se estamos renderizando caracteres em monoespaço ou não. As próximas quatro vão armazenar a cor padrão com a qual renderizaremoz nossas imagens. Elas correspondem a um valor entre 0 e 1 correspondente aos canais das cores vermelho, verde, azul e alfa (transparência), respectivamente. As próximas três irão armazenar a largura, altura e profundidade de um novo glifo caso estejamos renderizando algum no momento. Nos demais casos, elas conterão zero.

As oito primeiras de tais variáveis numéricas internas serão armazenadas na estrutura da meta-fonte:

Seção: Atributos (struct metafont) (continuação):

```
struct numeric_variable *internal_numeric_variables;
```

As outras três serão declaradas na estrututra de contexto, pois elas são associadas ao contexto de qual caractere estamos renderizando (se estivermos renderizando um):

Seção: Atributos (struct context) (continuação):

```
struct numeric_variable *internal_numeric_variables;
```

Na inicialização alocamos espaço para as sete primeiras variáveis e as preenchemos:

Seção: Inicialização (struct metafont) (continuação):

```
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
// Cor padrão: (0, 0, 0, 1) (preto opaco)
((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;
// Por padrão, a fonte não começa sendo monoespaço:
((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;
  }
}
```

As outras três serão alocadas não na inicialização de uma nova fonte, mas na inicialização de um novo contexto:

Seção: Inicialização (struct context) (continuação):

```
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;
  }
```

```
}
```

Criamos as seguintes macros para acessar cada uma destas variáveis mais facilmente:

Seção: Macros Locais (metafont.c) (continuação):

```
#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
```

Para finalizar a estrutura metafont, devemos desalocar tais variáveis internas:

Seção: Finalização (struct metafont) (continuação):

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

Da mesma forma que desalocamos ela também ao desalocar um contexto:

Seção: Finalização (struct context) (continuação):

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

Quando o nosso analisador léxico gera um novo token simbólico que não é uma palavra reservada, ele já deve checar se este token tem o nome de uma variável interna. Se tiver, o ponteiro para ela já é preenchido corretamente:

Seção: Preenche Ponteiro para Variável Interna:

```
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. Variáveis de Pares

Uma variável de par serve para armazenar coordenadas de um ponto. Ela será armazenada na seguinte estrutura:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

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

A diferença é que elas possuem espaço para dois valores em ponto flutuante ao invés de um. Inicialmente vamos representar o primeiro deles como NaN, para representar uma variável não-inicializada:

Seção: Inicialização de Nova Variável (continuação):

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

7.3. Variáveis de Transformação

Se um variável de par é uma tupla de dois valores numéricos, uma variável de transformação é uma tupla com seis valores diferentes. Sejam os valores (a, b, c, d, e, f), o que eles representam é uma transformação linear na qual o par (x, y) é transformado em (x', y') da seguinte forma:

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

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

A ordem em que os elementos são representados pode parecer estranha, mas é feito assim por compatibilidade com a linguagem METAFONT original criada por Knuth.

Uma variável de transformação então simplesmente deve armazenar uma transformação linear na forma de uma matriz:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

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

```
};
```

Uma transformação com valores não-inicializados possui seu primeiro valor marcado como NAN:

Seção: Inicialização de Nova Variável (continuação):

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

Existe uma variável de transformação interna que sempre estará presente sem precisar ser declarada. É a variável `identity`, que representa a ausência de uma transformação. É uma transformação que não irá mudar o ponto. Ela é representada pela tupla $(0, 0, 1, 0, 0, 1)$ e sua transformação na forma matricial é representada pela multiplicação por uma matriz identidade.

As variáveis internas de transformação serão armazenadas na estrutura Metafont:

Seção: Atributos (struct metafont) (continuação):

```
struct transform_variable *internal_transform_variables;
```

Na inicialização o espaço para armazenar as variáveis internas será alocado e as variáveis internas são inicializadas:

Seção: Inicialização (struct metafont) (continuação):

```
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 memória suficiente
}
// A transformação 'identity':
mf -> internal_transform_variables[0].type = TYPE_T_TRANSFORM;
INITIALIZE_IDENTITY_MATRIX(mf -> internal_transform_variables[0].value);
```

Na finalização isso deve ser desalocado:

Seção: Finalização (struct metafont) (continuação):

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

A variável `identity` deve ser registrada como tendo a primeira posição:

Seção: Macros Locais (metafont.c) (continuação):

```
#define INTERNAL_TRANSFORM_IDENTITY 0
```

E durante a leitura do código-fonte, podemos já ajustar corretamente todas as referências para esta variável colocando o valor correto de seu endereço nos tokens lidos:

Seção: Preenche Ponteiro para Variável Interna (continuação):

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

7.4. Variáveis de Caminhos

Variáveis de caminho são os tipos de variável mais complexas em WeaveFont. Estas variáveis armazenam uma sequência de curvas onde a próxima curva começa onde a última termina. Cada curva pode estar representada de mais de uma forma diferente. Mas cada uma delas é uma Curva de Beziér cúbica. O caminho pode ser cíclico, quando a última curva da sequência termina no mesmo ponto em que o primeiro começa. Para armazenar o conteúdo de um caminho, precisamos alocar estruturas adicionais. Basicamente uma variável de caminho tem um array de "pontos", com cada ponto sendo um ponto da curva com informações adicionais relacionadas à ele, ou então

sendo um ponteiro para um subcaminho com vários outros pontos. Alguns destes pontos podem estar ainda com informações incompletas que precisam ser preenchidas. A variável que armazena isso é:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
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; // Array com 'length' elementos.
};
```

O número de pontos de uma variável de caminho é sempre maior ou igual ao de sua variável `length`, já que cada posição do array de `length` elementos contém ouumúnico ponto, ou um subcaminho de 1 ou mais pontos.

De fato, cada ponto do array pode estar na verdade em quatro diferentes formatos:

Seção: Macros Locais (metafont.c) (continuação):

```
#define UNINITIALIZED_FORMAT 0 // Não deve ser usado, exceto como inicialização
#define PROVISIONAL_FORMAT   1 // Informações sobre o ponto sendo coletadas
#define SUBPATH_FORMAT       2 // Ponteiro para muitos pontos de um subcaminho
#define FINAL_FORMAT         3 // Formato final, pronto pra ser usado
```

Vamos descrever agora como é o formato final que almejamos para cada uma de nossas curvas.

Uma Curva de Beziér cúbica é definida por dois pontos de extremidade (z_1 e z_4) e dois pontos de controle (z'_2 e z'_3). Os dois pontos de extremidade fazem parte da curva. Para obter todos os outros pontos que fazem parte da curva à partir de quatro pontos (z_1, z'_2, z'_3, z_4), usa-se o seguinte procedimento:

1) Obtenha o ponto intermediário z'_{12}, que fica na metade do caminho entre z_1 e z'_2, o ponto intermediário z'_{23}, que fica no meio do caminho entre z'_2 e z'_3 e o ponto intermediário z'_{34}, que fica no meio do caminho entre z'_3 e z_4.

2) Obtenha agora os dois pontos intermediários novos: z'_{123}, que fica no meio do caminho entre z'_{12} e z'_{23} e o ponto z'_{234} que fica entre os pontos z'_{23} e z'_{34}.

3) O novo ponto que faz parte da curva gerado é o ponto z_{1234}, que fica no meio do caminho entre z'_{123} e z'_{234}.

4) Gere os outros pontos da curva aplicando este procedimento recursivamente sobre $(z_1, z'_{12}, z'_{123}, z_{1234})$ e sobre $(z_{1234}, z'_{234}, z'_{34}, z_4)$.

É possível também obter uma fórmula para tais curvas. Dado dois pontos de extremidade e dois de controle, todos os pontos intermediários podem ser obtidos usando a fórmula abaixo, variando t entre 0 e 1:

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

A estrutura de dados que armazena a sequência de Curvas de Beziér nos diferentes formatos é:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
struct path_points{
  int format; // Qual o formato do ponto
  union{
    // Formato final: armazena ponto de extremidade e pontos de controle
    struct{
      float x, y; // Ponto de extremidade
      float u_x, u_y, v_x, v_y;   // Pontos de controle que o sucedem
    } point;
    // Formato de subcaminho: aponta para subcaminho
```

```
    struct path_variable *subpath;
    // Formato provisório: outras informações que não são pontos de controle
    struct{
      float x, y; // Ponto de extremidade
      float dir1_x, dir1_y, dir2_x, dir2_y; // Especificadores de direção
      float tension1, tension2;
      bool atleast1, atleast2;
    } prov;
  };
};
```

Basicamente seus pontos estarão apontados pelo ponteiro `points`, um vetor de estrutura que terá um número de elementos igual a `length`. Cada estrutura representará um ponto (x, y) ou então terá um ponteiro para outra variável de caminho que contém um subcaminho (sempre que `subpath` não for nulo, e nesta caso ignoraremos x e y). Essa outra variável de caminho pode conter recursivamente muitos pontos. Para armazenar a quantidade total de pontos que o caminho contém recursivamente, usamos `total_length`. Enquanto a variável `length` armazena apenas o tamanho do array `points`.

Os pontos de controle definidos por `u_x`, `u_y`, `v_x` e `v_y` são os pontos de controle entre o ponto ou subcaminho atual e o próximo, se houver. Caso seja um caminho cíclico, os pontos de controle do último ponto ou subcaminho determinam como ele será ligado ao primeiro ponto. Se não for cíclico, os pontos de controle do último ponto são ignorados.

O modo como representaremos uma variável de caminho não-inicializada é mantendo seu tamanho (`length`) como sendo igual a -1:

Seção: Inicialização de Nova Variável:

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

Quando vamos remover uma variável na estrutura da meta-fonte, se ela for do tipo caminho e estiver inicializada, precisamos remover a lista de pontos alocada com a função de desalocação permanente:

Seção: Finaliza Variável 'v' em '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);
}
```

Se a variável estiver na estrutura de contexto, fazemos o mesmo, mas usando uma função de desalocação para variáveis temporárias:

Seção: Finaliza Variável 'v' em '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);
}
```

Como um caminho pode conter subcaminhos e cada um deles também pode ter seus próprios subcaminhos, em tais casos usamos uma função recursiva para desalocar memória. Dada a função de desalocação e um ponteiro para uma variável de caminho, nós desalocamos todos os seus subcaminhos e depois a desalocamos. A declaração da função é:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_recursive_free(void (*free_func)(void *),
                         struct path_variable *path,
```

```
                          bool free_first_pointer);
```

E sua implementação é:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. Removendo Recursão de Variáveis de Caminho

As variáveis de caminho são as mais complexas da linguagem WeaveFont, e podem estar representadas de diferentes formas. Entretanto, muitas destas formas são provisórias e apresentam características indesejáveis. Por exemplo, embora um caminho possa armazenar seus subcaminhos de maneira recursiva por meio de vários ponteiros, muitas vezes isso é indesejável, pois nos impede de acessar rapidamente cada um dos seus pontos.

Depois de avaliar um caminho, antes de armazenar ele como variável, é interessante transformá-lo para o formato em que seja mais fácil manipulá-lo. O caminho inicialmente vem no formato que foi mais fácil montá-lo, dado a expressão de caminho que o gerou. Devemos então aproveitar que o caminho será armazenado para convertê-lo de um formato para outro.

A primeira destas conversões é remover a recursão dos caminhos. Isso fará com que sua `length` e `total_length` fiquem com o mesmo tamanho, eliminando possibilidade de confusões. Além disso, nenhum de seus pontos irá apontar para qualquer subcaminho: cada um deles será efetivamente um ponto.

Para isso, vamos definir uma função que fará uma cópia de um caminho. Mas ao fazer tal cópia, toda a recursão da origem será eliminada no destino. A função que fará isso tem seu cabeçalho mostrado abaixo, seguido do cabeçalho da função de cópia recursiva auxiliar que ela chamará:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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); // Precisa alocar destino?
void recursive_aux_copy(struct path_points **dst,
                        struct path_variable *origin, int *missing_directions,
                        struct path_points **previous_point);
```

A função usa a função de alocação indicada, que pode ser a permanente ou temporária. Em seguida, preenche as variáveis relevantes no destino e começa a copiar recursivamente. Este também é o momento em que aproveitamos a iteração para checar se há informações pendentes que precisam ser preenchidas no caminho. Mais especificamente a variável `number_of_missing_directions`:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Já a função recursiva que efetivamente copia os pontos:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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));
      // Se temos dois pontos iguais, não há especificadores de direção
      // desconhecidos entre eles
      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 direção inválidos:
        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. Tensão e Especificadores de Direção

Vimos que um ponto que faz parte de um caminho pode estar em um formato provisório, além de seu formato final e do seu formato de ponteiro para subcaminhos. Quando isso ocorre, ele possui uma quantidade bem grande de variáveis. Em especial, ele possui dois valores numéricos representando sua tensão e possui dois pares representando os dois próximos especificadores de direção.

O primeiro especificador de direção deve ser considerado como um vetor contendo o ângulo de virada da trajetória da curva no primeiro ponto de extremidade e o segundo é o mesmo para o próximo ponto de extremidade. O ângulo de virada pode ser visto como o ângulo das "rodas" de um veículo naquele ponto, caso o veículo estivesse fazendo a trajetória da curva. Podemos representar tais informações no código abaixo representando o trecho de um caminho:

```
{w0} z1 {w1} .. tension t1 and t2 {w2} z2 {w3}
```

Note que se $w_0 \neq w_1$ ou $w2 \neq w_2$, então haverá uma quebra na curva no respectivo ponto de extremidade, haverá uma "ponta", um ponto não-derivável.

Já a tensão deve representar o quão "solta" ou "presa" está uma curva. Uma curva com tensão maior terá um perímetro menor e tenderá a se aproximar de uma reta à medida que a tensão aumenta.

Tendo tais informações, como converter o ponto da curva do formato provisório para o formato

final? Usando como exemplo o segmento de curva acima, isso é feito com ajuda das fórmulas abaixo:

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

Não é o vetor de direção em si, mas o ângulo computado acima de tais vetores que é efetivamente usado para computar os pontos de controle do formato final de um ponto. A diferença entre o primeiero formato provisório e o segundo é que o segundo já calculou esses ângulos θ e ϕ e armazenou os resultados, descartando os vetores de direção.

Para calcular a função arg acima, vamos precisar do cabeçalho:

Seção: Cabeçalhos Locais (metafont.c) (continuação):

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

Os pontos de controle u e v são obtidos então pelas fórmulas:

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

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

Com a função f definida como:

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

Podemos então declarar a função que faz a conversão e também a função que computa f abaixo:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

A função f:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Se assumirmos que um caminho já passou pela normalização que removeu sua recursão, podemos facilmente transformar todos os seus pontos de formato provisório para o final usando a função:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
                    <Seção a ser Inserida: Ajuste de Tensão>
    }
  }
  if(p -> cyclic)
    memcpy(&(p -> points[p -> length - 1]), &(p -> points[0]),
           sizeof(struct path_points));
}
```

Uma última complicação que resta na conversão é ajustar a tensão. Se a variável booleana `atleast0` estiver como verdadeira, isso significa que a primeira tensão que tínhamos era um valor mínimo. Mas devemos aumentá-la se necessário, para o menor valor tal que o primeiro ponto de controle fique dentro de um triângulo delimitado pelos pontos de extremidade e do ponto em que as retas direcionadas pelos especificadores de direção se cruzam:

Seção: Ajuste de Tensão:

```
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));
```

A declaração da função que corrige a tensão e que usamos acima:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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);
```

E a implementação da função:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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; // // Não é um triângulo válido
 { // Descobre o terceiro vértice da coordenada
   // Primeiro compute lado do triângulo que vai de p0 ao vértice conhecido
   double known_side = hypot(p1_x - p0_x, p1_y - p0_y);
   double triangle_side = known_side * sin(internal_angle0) /
                          sin(internal_angle2);
   // Conhecendo o lado e o ângulo, computamos as coordenadas
   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);
 }
 {
       <Seção a ser Inserida: Checa se Ponto Está Dentro do Triângulo>
       <Seção a ser Inserida: Se Não Estiver, Ajusta Tensão para Ficar>
 }
}
```

A função que obtém o ângulo entre dois vetores:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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)));
}
```

Como checar se um ponto está dentro de um triângulo? Há várias formas, a que usaremos consiste em medir a área com sinal dos três novos triângulos formados pelo ponto e dois vértices do triângulo. A "área com sinal" é a área do triângulo, só que ela tem valor positivo ou negativo dependendo se passamos os vértices no sentido horário ou anti-horário. Em seguida, apenas verificamos se todas as áreas tem o mesmo sinal (positivo ou negativo). Se tiver, então o ponto certamente está fora. Caso contrário, o ponto está dentro do triângulo:

Seção: Checa se Ponto Está Dentro do Triângulo:

```
bool s1, s2, s3; // O sinal das áreas
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;
```

Uma vantagem deste método é que dependendo de qual área com sinal tem um sinal diferente das outras, isso determina qual dos lados do triângulo é mais próximo do ponto. Então, se ainda estivermos na função, checamos qual é o lado mais próximo, obtemos sua equação de reta e calculamos o ponto mais próximo na reta. Se o ponto estiver no lado do triângulo, este é o novo ponto do ponto de controle. Se não, o ponto de controle se torna a extremidade do triângulo mais próxima do ponto.

Seção: Se Não Estiver, Ajusta Tensão para Ficar:

```
{
  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 deve ser maior ou igual a x0
    double tmp;
    tmp = x1 ; x1 = x0; x0 = tmp;
    tmp = y1; y1 = y0; y0 = tmp;
  }
  if(x0 == x1){ // Reta vertical (equação da reta daria divisão por 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) // Reta horizontal
    *control_y = y0;
  else{ // Usar equação da reta
    // m0 x + b0 = y é a reta que contém o lado do triângulo
    double m0 = (y1 - y0) / (x1 - x0);
    double b0 = y1 - m0 * x1;
    // m1 x + b1 = y é a perpendicular ao lado do triângulo e passa pelo ponto
    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. Deduzindo Especificadores de Direção

Quando interpretamos um caminho de uma expressão ou trecho de código, e após remover a recursão dele, ele sempre virá com seus pontos ou no formato final, ou no formato provisório. Caso esteja no primeiro formato provisório, ele sempre virá com as variáveis de tensão e `atleast` corretamente preenchidas. Entretanto, ele pode vir sem um dos especificadores de direção ou mesmo sem ambos. Por isso cada variável de caminho armazena em `number_of_missing_directions` quantos especificadores de direção faltam.

Podem haver dois casos: podemos ter especificadores de direção em que o primeiro valor é NaN (Not-a-NUmber) e o segundo é um número, ou ambos podem ser NaN. Se ambos não tiverem valor numérico, o especificador de direção está ausente e é desconhecido. Se só o segundo for um número, então ele representa a "enrolamento" ("curl") da curva. Quanto maior o valor, mais os pontos de extremidade agem "enrolando" a curva, como se ela fôsse uma corda sendo enrolada em uma roldana. Uma parte maior da curva vai se aproximando de uma reta, com a mudança brusca de direção sendo reduzida a uma área menor. De qualquer forma, ambos os casos são tratados como faltando um especificador, só a regra de como calcular eles muda.

Para preencher os especificadores faltantes de um caminho, devemos dividir ele em diferentes segmentos. Cada caminho com pontos faltantes possui um ou mais segmentos em que existe um especificador de direção ou "curl" apenas na região de extremidade do segmento:

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

Este é o código da função que preenche os especificadores faltantes. Note que o que ela faz é primeiro checar se temos um caso em que praticamente todos os especificadores estão faltando. Neste caso, o nosso segmento é o caminho todo. Caso contrário, ela itera sobre cada segmento até que não hajam mais especificadores faltando:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  }
}
```

Os valores desconhecidos são obtidos resolvendo sistemas de equações lineares. Cada ponto que fica nas extremidades finais de um segmento constribuirá com 1 incógnita e uma equação para o sistema. Todos os outros pontos contribuirão com 3 incógnitas e 3 equações. Se o segmento é um caminho cíclico inteiro, então não há pontos nas extremidades e cada ponto contribui com 3 incógnitas e 3 equações. Isso significa que em caminhos não-cíclicos de n pontos, há $3(n-2)+2$ incógnitas, enquanto em caminhos cíclicos de n pontos teremos $3n$ incógnitas.

As incógnitas acrestentadas serão θ_0 para o primeiro ponto de segmento não-cíclico, ϕ_{n-1} para o último ponto de segmento não-cíclico, e para os demais pontos de índice i, temos as incógnitas θ_i, ψ_i e ϕ_i. A ordem de cada incógnita no sistema de equações é: $(\theta_0, \theta_1, \psi_1, \phi_1, \ldots, \theta_{n-2}, \psi_{n-2}, \phi_{n-2}, \phi_{n-1})$ quando temos mais de dois pontos. Quando temos apenas dois pontos, a ordem é simplesmente (θ_0, ϕ_1).

Quando estivermos preenchendo os valores de uma nova equação do sistema de equações em uma matriz M, que representa o lado esquerdo de um sistema de equações linear, as incógnitas que cada equação pode representar serão o θ atual e anterior, o ψ atual, o ϕ atual e o próximo. Nenhuma outra incógnita será referenciada. Desta forma, cada linha da matriz M terá no máximo cinco elementos não-nulos.

Mas qual a posição destes cinco elementos na matriz M? Vamos declarar cinco variáveis para manter o controle de tais índices:

Seção: Declaração de Variáveis de Incógnitas (Não-Cíclico):

```
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;
```

Os valores inicializados acima já funcionam no caso não-cíclico quando estamos no primeiro ponto e queremos inserir a primeira equação. Lembre-se que a posição das incógnitas em cada coluna de M é dada pela ordem: $(\theta_0, \theta_1, \psi, \phi_1, \ldots)$. Então quando estamos no ponto 0, não existe θ anterior e nem ψ ou ϕ atuais. Já a posição do atual θ_0 e ϕ_1 (que é o próximo ϕ) estão corretas.

Já no caso cíclico, em todos os pontos, sempre teremos definidos os valores de θ, ψ e ϕ. Portanto, inicicalizamos da seguinte forma:

Seção: Declaração de Variáveis de Incógnitas (Cíclico):

```
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;
```

Se o nosso sistema de equações tem "`size`" elementos, então o seguinte trecho de código abaixo muda a posição destes valores para a próxima equação, mas sem mudar o ponto atual. Isso ocorre quando não estamos nem na primeira e nem na última equação, quando um mesmo ponto pode incluir até três equações. Neste caso as incógnitas referenciadas são as mesmas, só precisamos passar para a próxima linha da matriz M.

Seção: Próxima Equação, Mesmo Ponto:

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

Já quando passamos de um ponto pra outro, não só queremos passar para a próxima equação, mas também precisamos atualizar as incógnitas que não serão mais as mesmas:

Seção: Próxima Equação, Próximo Ponto:

```
{
  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 ++;
}
```

Note que não precisamos tratar como caso especial quando passamos do primeiro ponto e equação para o próximo ponto. Em tais casos, o ψ_0, que não existe, está já inicializado como tendo índice -1. Pular para a próxima variável (de ψ_n para ψ_{n+1}) funcionasempre incrementando em 3 a posição. Então, ao passar para fora do primeiro ponto, o valor já fica ajustado corretamente como o índice 2, que realmente é a posição de ψ_1 no vetor de incógnitas x.

Levando em conta o que foi apresentado sobre o sistema de equações a ser resolvido em cada caso, no caso de segmento não-cíclico, a função que preenche as direções faltantes é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
```
<Seção a ser Inserida: Declaração de Variáveis de Incógnitas (Não-Cíclico)>
<Seção a ser Inserida: Checa caso degenerado>
```
  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; // Tamanho do sistema de equações
  // Alocação do sistema:
  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));
  // Iterando sobre cada ponto adicionando sistema de equações:
```
<Seção a ser Inserida: Equação: Ponto Inicial>
```
  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;
```
<Seção a ser Inserida: Equação: Caso Geral>
```
    }
  }
```
<Seção a ser Inserida: Equação: Último Ponto>
```
  solve_linear_system(size, M, b, x);
```
<Seção a ser Inserida: Preenche especificador: primeiro ponto>
```
  {
    int theta;
    for(theta = 1, i = 1; i < number_of_points - 1; i ++, theta += 3){
      // Acha direção em p->points[i] usando valor de x[theta]
```
<Seção a ser Inserida: Preenche especificador: ponto interno>
```
    }
  }
```
<Seção a ser Inserida: Preenche especificador: último ponto>
```
  if(temporary_free != NULL){
    temporary_free(M);
    temporary_free(x);
    temporary_free(b);
  }
  return true;
}
```

Já a função que preenche as direções faltantes no caso do caminho cíclico inteiro não ter direções é bastante similar. Ela apenas deve levar em conta que todos os pontos do caminho tem direções a serem descobertas e também irá usar uma fórmula diferente para calcular o tamanho do sistema de equações lineares, já que haverão três incógnitas por ponto:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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); // Tamanho do sistema de equações
     <Seção a ser Inserida: Declaração de Variáveis de Incógnitas (Cíclico)>
  // Alocação do sistema:
  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));
  // Iterando sobre cada ponto adicionando sistema de equações:
  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;
                  <Seção a ser Inserida: Equação: Caso Geral>
  }
  solve_linear_system(size, M, b, x);
  {
    int theta;
    for(theta = 0, i = 0; i < p -> length; i ++, theta += 3){
      // Acha direção em p->points[i] usando valor de x[theta]
            <Seção a ser Inserida: Preenche especificador: ponto interno>
    }
  }
  if(temporary_free != NULL){
    temporary_free(M);
    temporary_free(x);
    temporary_free(b);
```

```
  }
  return true;
}
```

Vamos ver agora quais são as equações que devemos inserir em cada caso. Vamos ao primeiro ponto de um segmento não-cíclico. Pelas regras de como expressões de caminho são interpretadas, temos que esse primeiro ponto de segmento, que podemos chamar de z_0 e que é sucedido por z_1 (aqui tratamos ambos como números complexos), ou já tem um especificador de direção conhecido (o qual chamamos de w_0) ou então ele possui um valor de "curl" associado a ele, o qual chamamos de γ.

Se ele já tem um especificador de direção, o valor de θ_0 dele já é conhecido e dado pela equação:

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

Se ao invés disso ele tiver apenas um valor de γ, então descobriremos depois θ_0 com ajuda da seguinte equação que adicionamos ao sistema:

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

Com as constantes k_0 e k_1 dependendo das tensões t_1 e t_2 do ponto inicial:

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

O código para adicionar equação em ambos os casos segue abaixo:

Seção: Equação: Ponto Inicial:

```
if(!isnan(p1 -> prov.dir1_x)){ // w0 conhecido
  double w1_x = p1 -> prov.dir1_x, w1_y = p1 -> prov.dir1_y;
  M[0] = 1.0; // Theta_0 é a primeira incógnita
  b[0] = carg((w1_x + w1_y * I) / (z2 - z1));
}
else{ // gamma conhecido
  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 ++;
}
```

No caso do ponto final, nós também teremos ou um especificador de direção w_n conhecido ou teremos um valor de "curl" conhecido com o valor γ_n.

Se a direção w_n for conhecida, nós já sabemos o falor de ϕ_n pela fórmula:

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

Já se tivermos apenas um valor de γ_n conhecido, então o valor de ϕ_n será descoberto depois com ajuda da equação:

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

Usando as constantes abaixo, que dependem da tensão t_1 e t_2 presente no ponto anterior:

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

O código para adicionar equação em ambos os casos segue abaixo.

Seção: Equação: Último Ponto:

```
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)){ // w_n conhecido
  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{ // gamma conhecido
  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;
}
```

Por fim, vamos às três equações que são geradas para os pontos interiores, que são todos os pontos de um segmento cíclico ou todos os pontos de um segmento não-cíclico que não são o primeiro nem o último.

Dado o ponto z_n precedido por z_{n-1} e sucedido por z_{n+1}, temos:

$$\psi_n = arg((z_{n+1} - z_n)/(z_n - z_{n-1}))$$
$$\theta_n + \psi_n + \phi_n = 0$$
$$k_0\theta_{k-1} + k_1\theta_k + k_2\phi_n + k_3\phi_{n+1} = 0$$

Assumindo o trecho de segmento:

$$z_0 \mathinner{..} tension\ t_0\ and\ t_1 \mathinner{..} z_1 \mathinner{..} tension\ t_2\ and\ t_3 \mathinner{..} z_2$$

As constantes k_0, k_1, k_2 e k_3 para o ponto z_1 são dadas por:

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

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

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

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

O código que armazena as três equações segue abaixo:

Seção: Equação: Caso Geral:

```
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;
          <Seção a ser Inserida: Próxima Equação, Mesmo Ponto>
M[current_theta] = M[current_psi] = M[current_phi] = 1.0;
b[number_of_equations] = 0.0;
          <Seção a ser Inserida: Próxima Equação, Mesmo Ponto>
{
  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;
          <Seção a ser Inserida: Próxima Equação, Próximo Ponto>
}
```

Mas o quê exatamente estas equações representam e de onde elas vieram? Basicamente usamos as mesmas regras e equações que a linguagem METAFONT usa para gerar suas curvas. Isso nos dá compatibilidade com a linguagem. O método usado e as equações foram desenvolvidas por John Hobby e publicadas em [HOBBY, 1986]. O que as regras significam é que o caminho tem o seu formato invariante diante de translação, rotação e mudança de escala. Além disso, a fórmula foi desenvolvida para que qualquer mudança em um ponto local tenha influência cada vez menor nos pontos seguintes (o impacto de mudanças cai exponencialmente) e para minimizar pontas e mudanças bruscas de direção na curva.

A equação que determina o formato da curva sempre tem solução desde que todas as tensões sejam maiores que 3/4 e os valores do "enrolamento" sejam sempre positivos ou nulos. Estas restrições de valores serão reforçadas na interpretação de expressões de caminho. Se encontrarmos um valor fora destes limites, iremos gerar um erro.

Existe, contudo, um último caso degenerado que podemos encontrar, em que a equação pode não ter solução. Ele ocorre quando tempos um segmento de apenas dois pontos e ambos possuem um "enrolamento" ao invés de um especificador de direção. Há dois casos possíveis neste cenário: eles geram um sistema de equações sem solução ou eles geram um sistema onde a solução é uma linha reta. Por causa disso, vamos assumir que sempre temos uma linha reta em tais casos:

Seção: Checa caso degenerado:

```
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;
```

```
}
```

Agora vamos começar a preencher os valores desconhecidos dos especificadores de direção, assumindo que já resolvemos o sistema de equações linear. No caso do primeiro ponto de um segmento não-cíclico, as equações nos revelaram o valor θ_0 associado a ele. Se ele já tinha um especificador de direção, não precisamos fazer nada. Se ele não tinha, o especificador w_0 é definido como:

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

E este é o código que coloca o primeiro especificador de direção se precisar:

Seção: Preenche especificador: primeiro ponto:

```
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 --;
}
```

Já no caso do último ponto, resolver o sistema de equações nos revelou ϕ_n. Se ele já possuía um especificador de direção, não é necessário fazer nada. Caso contrário, podemos deduzir seu especificador w_n com a fórmula:

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

O código que faz isso, se necessário, segue abaixo:

Seção: Preenche especificador: último ponto:

```
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 --;
}
```

Já para todos os demais pontos de índice k, nós temos tanto um θ_k como um ϕ_k e podemos usar tanto uma como a outra fórmula que o mesmo resultado é obtido. Mas a iteração que percorre os pontos para preencher os resultados itera sobre cada um dos valores existentes ajustando um índice `i` correspondendo ao número do ponto que estamos visitando e um índice `theta` com o valor do índice no vetor solução x que possui o θ associado ao ponto atual. Por causa disso, escolhemos usar a fórmula de θ_k:

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

Para os pontos internos, cada ponto possui na verdade dois especificadores de direção: um do lado direito (armazenado nele mesmo) e outro do lado esquerdo (armazenado no ponto anterior). Preenchemos ambos os valores com o código abaixo:

Seção: Preenche especificador: ponto interno:

```
{
  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. Normalização de Caminhos

Agora que definimos uma função que remove a recursão de caminhos (`recursive_copy_points`, vista na Subseção 7.4.1.), outra para garantir que todos os pontos terão especificadores de direção (`find_missing_directions`, vista na Subseção 7.4.3.) e uma outra que converte todos os pontos de um caminho do formato provisório para o final (`convert_to_final`, visto na Subseção 7.4.2.), é hora de combinar todas estas funções de modo a normalizar um caminho, deixando ele em seu formato final: o formato no qual ele já pode ser renderizado.

Fazer isso envolve chamar cada uma destas funções na ordem certa. Entretanto, existe uma última complicação que devemos tratar para caminhos cíclicos.

Poderíamos representar um caminho cíclico de duas formas: assumindo que é um caminho onde o primeiro e último ponto sempre são iguais ou assumindo que há uma ligação adicional entre o último e o primeiro ponto. No primeiro caso, o caminho cíclico abaixo armazenaria os pontos $(0,0)$, $(0,3)$, $(4,0)$ e $(0,0)$ totalizando quatro pontos, sendo que o último é igual ao primeiro. No segundo caso, armazenaríamos apenas os três pontos e presumiríamos que o último ponto está ligado ao primeiro, já que o caminho estaria marcado como cíclico.

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

Aparentemente, representar apenas três pontos é mais elegante: não desperdiçamos memória com um ponto adicional e como visto no código da subseção anterior, podemos iterar sobre caminhos cíclicos, começando à partir de qualquer ponto que códigos como o abaixo passariam exatamente uma vez por cada ponto:

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

Observe como no código acima, não importa qual o valor de `begin`, nós visitamos cada ponto uma só vez. Mas apenas se trabalharmos com o pressuposto de que não estamos repetindo o primeiro ponto uma vez adicional no fim do caminho cíclico. No código das Subseções anteriores, chegamos a usar um laço deste tipo para percorrer um segmento de caminho, e o código funciona mesmo quando o final do segmento de caminho cíclico está armazenado antes do começo.

Entretanto, há dois casos em que há vantagem em armazenar explicitamente uma cópia do

primeiro ponto no fim de um caminho cíclico. Primeiro, que e fizermos isso, a função que desenha o caminho usando uma caneta não precisa tratar de maneira diferente caminhos cíclicos e não-cíclicos. Nos dois casos, ela vai desenhando passando por cada ponto até chegar ao último. Segundo, e mais importante, considere o código abaixo que concatena dois caminhos:

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

A concatenação destrói a natureza cíclica do caminho. Se nós não armazenamos explicitamente um ponto $(0, 0)$ no fim do segundo caminho concatenado, como o resultado da operação deixa de ser cíclico, teremos que inserir um ponto $(0, 0)$ adicional no fim do caminho resultante. Para isso, potencialmente podemos ter que realocar o vetor em que os pontos são armazenados no segundo caminho.

Para evitar aumentar a complexidade do código de concatenação e de desenho, optaremos então por representar explicitamente uma cópia do primeiro ponto no fim de um caminho cíclico. Mas para os casos em que é melhor não ter esta cópia, podemos então realizar as operações:

1) Decremente a variável `length` do caminho.

2) Realize as operações assumindo que uma cópia do primeiro ponto não é colocada no fim.

3) Incremente novamente a variável `length` e copie para a última posição o primeiro ponto.

Desta forma, o código da Subseção anterior funcionará e conseguimos ficar com quase o melhor dos dois mundos. Apenas tendo que fazer a conversão acima em algumas partes. Por exemplo, dada a função que normaliza um caminho deixando-o em seu formato final:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Ela pode ser implementada da seguinte forma:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
  // Path se torna idêntico ao que era, mas sem recursão:
  memcpy(path, new, sizeof(struct path_variable));
  if(disalloc != NULL)
    disalloc(new);
  if(path -> number_of_missing_directions > 0){
    if(path -> cyclic){
      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 --;
      path -> length --;
```

```
    }
    if(!find_missing_directions(mf, cx, path))
      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. Variáveis de Caneta

Uma variável de caneta armazena a estrutura utilizada para desenhar formas no METAFONT. As canetas especificam o diâmetro e formato das linhas e pontos a serem desenhados. Elas são armazenadas na seguinte estrutura:

Seção: Cabeçalhos Locais (metafont.c) (continuação):

```
struct pen_variable;
```

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
struct pen_variable{
  int type; // Deve ser 'TYPE_T_PEN'
  void *next;
  bool permanent; // Alocação da variável é permanente ou temporária?
  struct path_variable *format; // O formato como caminho cíclico
  int flags;
  struct pen_variable *referenced; // Não-nulo só em 'currentpen'
  float gl_matrix[9]; // Matriz de transformação OpenGL
  // As variáveis abaixo serão manipuladas só na Seção 11.2
  GLuint gl_vbo; // Os vértices OpenGL após triangulação
  GLsizei indices; // Número de vértices presentes acima
  // Esta armazena o quão detalhada é a triangulação, necessária para
  // saber se devemos triangular novamente ou não em alguns casos:
  float triang_resolution;
};
```

Aqui vamos dar uma breve explicação do que são cada uma destas variáveis e por que elas são necessárias:

As variáveis `type`, `nesting_level` e `next` são comuns a todos os tipos de variáveis e não apresentam novidade. Servem para que saibamos qual tipo de dado está sendo armazenado na variável, qual o seu escopo e qual a próxima variável caso ela esteja em uma lista encadeada.

A variável `format` especifica o formato de nossa caneta como um caminho cíclico. No METAFONT original, era necessário que além de cíclico, o caminho seja convexo. Para nós, será necessário apenas que o caminho seja cíclico e simples. O que significa que o perímetro do caminho não pode ter ter intersecções e cruzar consigo mesmo. Caso isso não seja respeitado, o resultado será indefinido e não iremos garantir que as canetas gerem o resultado esperado.

Não importa o quão complexo seja o formato da caneta, para que ele possa ser desenhado na tela via OpenGL, ele deve ser convertido para um conjunto de triângulos em um processo chamado de triangulação. Detalhes sobre como triangular, ou mesmo se precisamos triangular serão definidos pela variável `flags`. As flags possíveis são:

Seção: Macros Locais (metafont.c) (continuação):

```
#define FLAG_CONVEX        1
```

```
#define FLAG_STRAIGHT      2
#define FLAG_CIRCULAR      4
#define FLAG_SEMICIRCULAR  8
#define FLAG_SQUARE       16
#define FLAG_NULL         32
// Mais flags serão definidas posteriormente, Subseção 11.2
```

A flag `FLAG_CONVEX` armazena se estamos diante de um formato convexo. Se esse for o caso, fazer a triangulação será um processo muito simples. Poderemos usar um algoritmo muito mais simples e rápido que fará tudo em O(n) quando precisarmos.

A flag `FLAG_STRAIGHT` armazena se o formato de nossa caneta é formado apenas por linhas retas, sem curvas. Se isso for verdade, então quer dizer que uma vez que ele seja triangulado, nunca mais precisaremos fazer isso novamente, mesmo que a caneta depois seja ampliada ou reduzida. Se ela é formada só por linhas retas, os triângulos são capazes de represerntar seu formato com perfeição, não estamos usando nenhum tipo de aproximação. Por isso, não precisamos gerar a triangulação novamente para poder mostrar mais detalhes quando, por exempolo, a caneta se torna maior.

A flag `FLAG_CIRCULAR` armazena se nós sabemos que a caneta tem um formato circular. Neste caso, seremos capazes de fazer a triangulação sem olhar para os pontos da variável de caminho representando o formato. De modo similar, a flag `FLAG_SEMICIRCULAR` checa se a caneta é um semi-círculo, o que também permite o mesmo.

A flag `FLAG_SQUARE` armazena se a caneta é uma caneta quadrada. Neste caso também não precisamos dos pontos da variável de caminho e podemos usar uma triangulação já feita antes sem precisar triangular novamente.

A flag `FLAG_NULL` armazena se esta é uma caneta nula. Neste caso ela nunca precisa ser triangulada, pois ela representa um único ponto sem largura nem altura.

A variável `referenced` só poderá ter um valor não-nulo quando estivermos nos referindo à caneta 'currentpen'. Neste caso, a 'currentpen' atuará como um ponteiro para outra caneta definida. Devemos considerar então que 'currentpen' tem o formato e triangulação da caneta apontada, mas a sua matriz de transformação (descrita abaixo) na verdade deverá ser considerada como a matriz da caneta apontada multiplicada pela matriz de `currentpen`.

A variável `gl_matrix` armazena a matriz de transformação OpenGL. Como as canetas possuem um formato bem-definido e não serão depois subdivididos ou concatenados como podem ser os subcaminhos, então podemos representar eventuais transformações lineares nela por meio dessa matriz ao invés de ter que computar novos valores para seus pontos.

As demais variáveis serão usadas mais na Seção 11.2 (sobre triangulação). A `gl_vbo` referenciará os vértices da caneta após ela ser triangulada para poder ser desenhada. Se a caneta não foi triangulada, seu valor será zero. Tais vértices, se existirem, estarão armazenados na memória da placa de vídeo. Por fim, `triang_resolution` será uma medida interna do quão detalhada é a triangulação. Ela ajudará a determinar se precisamos criar uma triangulação mais detalhada, ou se a que temos já é a suficiente.

Uma variável de caneta declarada, mas ainda não inicializada terá seu formato igual ao ponteiro nulo, será não-circular e como ela ainda não foi triangulada, terá seu ID de vértices OpenGL igual a zero.

Seção: Inicialização de Nova Variável (continuação):

```
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;
}
```

O que significa que ao remover uma variável global e ela for de caminho, devemos fazer a desalocação do formato e jogar fora os vértices:

Seção: Finaliza Variável 'v' em 'struct metafont':

```
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));
}
```

Se a variável for local ao invés de global, nós fazemos o mesmo usando uma função diferente de desalocação.

Seção: Finaliza Variável 'v' em 'struct context' (continuação):

```
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));
          <Seção a ser Inserida: Finalizando Caneta Local 'pen'>
}
```

Além das variáveis de caneta definidas pelo usuário, iremos suportar a existência de duas outras variáveis interna de caneta. Uma delas será chamada de `currentpen` e irá representar a caneta atual com a qual iremos escrever e desenhar. A outra será uma caneta de formato perfeitamente quadrado que estará já inicializada à disposição do usuário e que deixamos já alocada para fins de melhor performance.

A variável `currentpen` será especial: ela será como uma variável global. Mas o seu estado entre diferentes execuções de código WeaveFont não será salvo: cada vez que executamos novamente um determinado código, é como se esta variável tivesse sido novamente reinicializada. Além disso, iremos armazenar informações sobre ela que não armazenamos para as outras canetas: a sua maior e menor coordenada x, assim como a sua maior e menor coordenada y. Essa variável não ficará junto com as outras: ela não estará guardada na estrutura `metafont` que representa a fonte, mas na estrutura que representa o nosso contexto de execução:

Seção: Atributos (struct context) (continuação):

```
struct pen_variable *currentpen;
```

Já a variável de caneta quadrada, que será chamada de `pensquare` será tratada como uma variável interna típica:

Seção: Atributos (struct metafont) (continuação):

```
struct pen_variable *internal_pen_variables;
```

A variável `currentpen` é inicializada como uma caneta vazia:

Seção: Inicialização (struct context) (continuação):

```
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);
```

Já variáveis internas precisam ter o seu espaço alocado. E a caneta quadrada será inicializada como uma caneta quadrada de lado 1:

Seção: Inicialização (struct metafont) (continuação):

```
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);
```

As seguintes macros nos permitirão acessar mais fácil a posição da variável interna:

Seção: Macros Locais (metafont.c) (continuação):

```
#define INTERNAL_PEN_PENSQUARE  0
```

Para finalizar a estrutura metafont, devemos desalocar as variáveis internas:

Seção: Finalização (struct metafont) (continuação):

```
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);
}
```

E também devemos desalocar a `currentpen` após encerrar um contexto de execução:

Seção: Finalização (struct context) (continuação):

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

E ao ler um código-fonte, devemos ajustar corretamente os ponteiros das variáveis que apontam para tais variáveis internas (não fazemos isso com a `currentpen` pois ela será tratada de maneira diferente):

Seção: Preenche Ponteiro para Variável Interna:

```
else if(!strcmp(buffer, "pensquare"))
```

```
  new_token -> var =
          &(mf -> internal_pen_variables[INTERNAL_PEN_PENSQUARE]);
```

A caneta '`currentpen`' é especial por ser a única caneta que pode ao invés de ter um conteúdo próprio, referenciar o conteúdo de outra caneta como um ponteiro. Mas o que acontece se desalocamos uma caneta que era uma variável local, mas que era referenciada pela `currentepen`? Fácil. Neste caso, a `currentpen` volta a ter seu valor padrão e ser uma caneta nula:

Seção: Finalizando Caneta Local '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. Variáveis de Imagens

Uma variável de imagem armazena uma imagem renderizada, possivelmente por meio de pincéis e comandos de desenho do Weaver Metafont. Ao contrário do METAFONT original, o Weaver Metafont requer que cada imagem tenha um tamanho bem-definido. Todas elas terão uma altura e largura. E além disso terão um índice representando uma textura OpenGL, onde estará a iamgem renderizada:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

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

Uma variável de imagem que foi declarada, mas ainda não inicializada terá altura e largura negativa, e seu índice de textura será zero:

Seção: Inicialização de Nova Variável (continuação):

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

Ao remover uma variável alocada permanentemente na estrutura da meta-fonte, se ela for de imagem, pedir via OpenGL que a textura seja destruída caso exista:

Seção: Finaliza Variável 'v' em 'struct metafont':

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

Se a variável for local ao invés de global, nós fazemos o mesmo:

Seção: Finaliza Variável 'v' em 'struct context' (continuação):

```
if(v -> type == TYPE_T_PICTURE){
  struct picture_variable *pic = (struct picture_variable *) v;
  if(pic -> texture != 0)
```

```
    glDeleteTextures(1, &(pic -> texture));
}
```

Existirá uma variável interna de imagem no Weaver Metafont chamada de `currentpicture`. Mas assim como a `currentpen`, esta variável será especial: será reinicializada em cada contexto de execução. Será uma imagem nova cada vez que um novo caractere será renderizado.

Seção: Atributos (struct context) (continuação):

```
struct picture_variable *currentpicture;
```

E na inicialização de cada execução, deixamos a variável `currentpicture` declarada, mas deixada como não-inicializada:

Seção: Inicialização (struct context) (continuação):

```
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;
```

No momento em que chegar a hora de desalocar o contexto de execução, devemos eliminar a textura dela também caso ela tenha sido inicializada:

Seção: Finalização (struct context) (continuação):

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

7.7. Variáveis Booleanas

A mais simples de todas as variáveis, uma variável booleana armazena somente verdadeiro ou falso. O formato de sua estrutura é:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

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

O valor `value` armazenado pela variável será 0 se for falso ou 1 se for verdadeiro. No caso de ser uma variável não-inicializada, armazenaremos -1:

Seção: Inicialização de Nova Variável (continuação):

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

8. Atribuições

Atribuições são como fazemos com que as variáveis sejam inicializadas, como modificamos seus valores e como armazenamos nelas o resultado de expressões.

A sintaxe de uma atribuição é:

```
<Instrução Simples> -> <Declaração de Variável> | <Atribuição> | ...
```

```
<Atribuição> -> <Variável> = <Lado Direito> |
                <Variável> := <Lado Direito>
<Lado Direito> -> <Expressão> | <Atribuição>
```

Isso significa que podemos realizar atribuições encadeadas, por exemplo, todas as variáveis abaixo, se forem numéricas, passarão a valer 5:

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

Se no começo de uma declaração nós encontramos uma variável, então certamente estamos diante de uma atribuição.

No contexto de uma atribuição, os tokens "=" e ":=" são equivalentes:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_EQUAL,             // O token simbólico '='
TYPE_ASSIGNMENT,        // O token simbólico ':='
```

Ambos os tipos de atribuição vão para a lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

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

Agora que temos este novo tipo de token, o código para avaliar uma atribuição segue em seguida. O que ele faz é percorrer a atribuição detectando todas as variáveis que deverão passar pela atribuição. Enquanto faz isso, o código checa se todas as variáveis estão declaradas, se todas possuem o mesmo tipo, se tentamos atribuir para algo que não é uma variável, um símbolo de atribuição está faltando ou se está faltando uma expressão depois do último símbolo de atribuição. Em qualquer um destes casos, um erro é gerado.

A parte que ainda não está sendo mostrada no código abaixo, mas será definida nas próximas subseções é como avaliar a expressão após o último símbolo de atribuição e como efetivamente fazemos a atribuição. Isso porque o modo de fazer isso depende do tipo das variáveis. Dependendo do tipo delas, esperaremos encontrar um tipo diferente de expressão, e realizar o armazenamento de tais variáveis também será diferente. Estes detalhes veremos nas próximas subseções.

Seção: Instrução: Atribuição:

```
else if(begin -> type ==  TYPE_SYMBOLIC){
  struct symbolic_token *var = (struct symbolic_token *) begin;
  struct generic_token *begin_expression;
  int type = 0; // Tipo das variáveis na atribuição
  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;
    }
    // Variáveis como 'h', 'w', 'd', 'currentpen' e
    // 'currentpicture' são especiais:
    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(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;
```

<Seção a ser Inserida: Atribuição de Variável Numérica>
<Seção a ser Inserida: Atribuição de Variável de Par>
<Seção a ser Inserida: Atribuição de Variável de Transformação>
<Seção a ser Inserida: Atribuição de Variável de Caminho>
<Seção a ser Inserida: Atribuição de Variável de Caneta>
<Seção a ser Inserida: Atribuição de Variável de Imagem>
<Seção a ser Inserida: Atribuição de Variável Booleana>

```
  return true;
}
```

O código de atribuição acima será executado sempre que o primeiro token de uma instrução for um nome de variável. Isso só ocorre em um programa válido quando estamos prestes a fazer uma atribuição.

8.1. Atribuições e Expressões Numéricas

Como então realizar a atribuição de variáveis numéricas após checarmos que não há erros no lado esquerdo a instrução de atribuição? Usamos o código abaixo:

Seção: Atribuição de Variável Numérica:

```
if(type == TYPE_T_NUMERIC){
```

```
  int i;
  struct numeric_variable result;
  // Obtém o valor da expressão do lado direito da atribuição:
  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 ++){
    // A atribuição em si:
    ((struct numeric_variable *) (var -> var)) -> value = result.value;
    // Obtendo a próxima variável a atribuir:
    var = (struct symbolic_token *) (var -> next);
    var = (struct symbolic_token *) (var -> next);
  }
}
```

Agora passemos para definir como interpretar expressões numéricas.

8.1.1. Soma e Subtração: Normal e Pitagórica

As regras iniciais de expressões numéricas seguem abaixo:

```
<Expressão Numérica> -> <Terciário Numérico>
<Terciário Numérico> -> <Secundário Numérico> |
                        <Terciário Numérico> <T-Op> <Secundário Numérico>
<T-Op> -> + | - | ++ | +-+
```

Os símbolos de `+` e `-` corespondem à adição e subtração. O símbolo `++` corresponde à soma pitagórica:

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

Isso pode ser facilmente calculado em C usando a função da biblioteca padrão matemática `hypot`.

Já o símbolo `+-+` corresponde à "subtração pitagórica" definida abaixo:

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

Na subtração pitagórica, nós vamos usar a última definição envolvendo a multiplicação de duas raízes quadradas em nossa implementação, pois é a forma de cálculo que minimiza erros e presença de overflows e underflows.

Os quatro operadores acima serão aqueles com a menor precedência. Tais operações de soma e subtração serão feitas apenas depois que todas as outras operações forem feitas.

Vamos definir os tokens de tais operadores:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_SUM,                   // O token simbólico '+'
TYPE_SUBTRACT,              // O token simbólico '-'
TYPE_PYTHAGOREAN_SUM,       // O token simbólico '++'
TYPE_PYTHAGOREAN_SUBTRACT,  // O token simbólico '+-+'
```

Entretanto, identificar corretamente os operadores de uma expressão terciária envolve levar em conta delimitadores como "[", "]", "{" e "}", além dos parênteses que já foram definidos quando introduzimos o analisador léxico. Por isso, adicionaeremos também tokens para os delimitadores que ainda não foram definidos:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_OPEN_BRACKETS,         // O token simbólico '['
TYPE_CLOSE_BRACKETS,        // O token simbólico ']'
TYPE_OPEN_BRACES,           // O token simbólico '{'
```

```
TYPE_CLOSE_BRACES,          // O token simbólico '}'
```

E adicionamos todos na lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

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

Vamos agora à avaliação das expressões. A função que as avalia para expressões numéricas é:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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);
```

As expressões numéricas começam nas expressões terciárias numéricas onde aparecem essas somas e subtrações. Pelas regras de sintaxe, o que temos que fazer é percorrer toda a expressão até chegar ao último símbolo de soma ou subtração que não estiver delimitado por parênteses, colchetes ou chaves.

Para ajudar com isso, as seguintes macros vão declarar variáveis para armazenar o aninhamento de delimitadores como parênteses e chaves, vão checar se o aninhamento está aumentando ou diminuindo e verificar se estamos dentro de um aninhamento ou não:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;
```

Depois de percorrermos uma expressão, podemos querer gerar um erro caso tenhamos aberto um parênteses, um colchetes ou chaves que não foi fechado ou se fechamos um que não foi aberto. Para isso, a seguinte macro pode ser usada:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

Note que a checagem de erros acima não detecta todos os erros possíveis envolvendo delimitadores. Por exemplo, podemos ter um "`({)}`" que está errado, mas que não seria detectado. Tais tipos de erro serão detectados futuramente. No momento só estamos nos preocupando com delimitadores porque temos que identificar quando um operador está fora dele para ser considerado um operador de expressão terciária. E somente erros que nos impeçam de identificar isso corretamente serão tratados.

Em uma expressão terciária, tudo que estiver do lado esquerdo do operador terciário mais à direita deve ser avaliado recursivamente como outra expressão terciária numérica. Já o que estiver à direita será interpretado como uma expressão secundária numérica. Por fim, se nenhum símbolo de soma e subtração for encontrada, a expressão inteira será avaliada como uma expressão secundária numérica.

Contudo, existem exceções, casos nos quais não devemos considerar os tokens `+` e `-` como soma e subtração. Por exemplo:

```
a = +1;
```

Neste caso, o token de `+` é só um operador unário que não muda o sinal do elemento que o sucede. Se fôsse um `-`, ele inverteria o sinal do elemento a seguir. Não está ocorrendo soma ou subtração alguma.

Este caso ocorre quando o `+` ou `-` estiverem bem no começo da expressão numérica, ou então caso sejam precedidos por vírgula, abrir de colchetes, símbolo de multiplicação, divisão, outro operador terciário ou então um dos tokens seguintes que veremos mais adiante: `length`, `sqrt`, `sind`, `cosd`, `log`, `exp`, `floor`, `uniformdeviate`, `rotated`, `shifted`, `slanted`, `xscaled`, `yscaled`, `zscaled`, `of`, `point`, `precontrol`, `postcontrol` ou `uniformdeviate`.

Se nós temos um token anterior (`prev`) e um atual (`cur`), podemos checar se ele representa uma adição ou subtração válida com a macro abaixo:

Seção: Macros Locais (metafont.c) (continuação):

```
#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))
```

O código abaixo usa tais macros e interpreta as expressões numéricas terciárias e trata a execução de operadores terciários, identificando-os corretamente com ajuda da macro:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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{ // Encontra último operador terciário: '+', '-', '++' ou '+-+'
    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){ // Se nós temos operador terciário:
    begin_secondary = (struct generic_token *) (last_sum -> next);
    if(!eval_numeric_expression(mf, cx, begin, end_tertiary, &a))
      return false;
    if(!eval_numeric_secondary(mf, cx, begin_secondary, end, &b))
      return false;
    if(last_sum -> type == TYPE_SUM) // Avalia '+':
      result -> value = a.value + b.value;
    else if(last_sum -> type == TYPE_SUBTRACT) // Avalia '-':
      result -> value = a.value - b.value;
```

```
    else if(last_sum -> type == TYPE_PYTHAGOREAN_SUM) // Avalia '++':
      result -> value = hypotf(a.value, b.value);
    else if(last_sum -> type == TYPE_PYTHAGOREAN_SUBTRACT){ // Avalia '+-+':
      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 // Sem operador terciário:
    return eval_numeric_secondary(mf, cx, begin, end, result);
}
```

8.1.2. Multiplicação e Divisão

As regras para lidar com expressões secundárias são:

```
<Secundário Numérico> -> <Primário Numérico> |
                         <Secundário Numérico> <S-Op> <Primário Numérico>
<S-Op> -> * | /
```

Os operadores `*` e `/` são respectivamente a multiplicação e divisão.

Vamos adicionar estes operadores à lista de palavras-chave reservadas e definir seus tipos:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_MULTIPLICATION,        // O token simbólico '*'
TYPE_DIVISION,              // O token simbólico '/'
```

Seção: Lista de Palavras Reservadas (continuação):

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

A função que irá avaliar expressões secundárias é:

Vamos agora à avaliação das expressões. A função que as avalia para expressões numéricas é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E a sua definição é muito semelhante à da função que avalia expressões terciárias, exceto que ela lida com o cálculo de multiplicação e divisão.

Entretanto, deve-se levar em conta que o token de divisão só deve ser tratado neste estágio se aquilo que estiver antes e depois dele não forem dois tokens numéricos. See forem, estamos diante de uma fração e elas serão tratadas com um nível de precedência ainda maior. Por outro lado, se já consideramos o token retrasado como uma fração, então neste caso sim, o símbolo será de divisão, mesmo que esteja cercado de tokens numéricos. Assim o código `1/3/1/3` é interpretado como a divisão de duas frações (1/3)/(1/3):

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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{ // Acha último operador secundário '*' ou '/'
    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) // Separador de fração
         last_fraction = p;
       else{ // Multiplicação ou divisão válida
         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){ // Se nós temos operador secundário:
    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) // Avalia '*':
      result -> value = a.value * b.value;
    else if(last_mul -> type == TYPE_DIVISION){ // Avalia '/':
      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 // Nenhum operador secundário:
    return eval_numeric_primary(mf, cx, begin, end, result);
}
```

No código acima, fazemos novamente uma checagem por erros de uso de delimitadores, mesmo que uma checagem já tenha sido feita ao avaliar a expressão terciária. Isso é necessário porque, por exemplo, o seguinte código incorreto não produziria nenhum erro durante a avaliação de expressão terciária que foi definida na Subseção anterior, mas encontraria o erro de uso incorreto na avaliação da expressão secundária acima:

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

8.1.3. Módulo, Seno, Cosseno, Exponenciais, Piso e Aleatórios Uniformes

As regras de expressões numéricas primárias são:

```
<Primário Numérico> -> <Átomo Numérico> |
                       length <Primário Numérico> | (...) |
                       <Operador Numérico> <Primário Numérico>
<Operador Numérico> -> sqrt | sind | cosd | log | exp | floor |
                       uniformdeviate |
                       <Operador de Multiplicador Escalar>
<Operador de Multiplicador  Escalar> -> + | - |
                       <Token Numérico Primário antes de var>
<Token Numérico Primário> -> <Token Numérico> / <Token Numérico> |
                             <Token Numérico não sucedido por ‘‘/ num’’>
```

O operador `length`, quando sucedido por um primário numérico significa que queremos o módulo do valor. Ele pode ter outros significados se sucedido por outro tipo de expressão primária.

Os novos operadores que aparecem aqui precisam ser adicionados à lista de tokens:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

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

Seção: Lista de Palavras Reservadas (continuação):

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

A função que interpreta uma expressão numéria primária será esta:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Podemos decidir qual regra deve ser aplicada na avaliação da expressão numérica primária por meio de sete regras de interpretação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

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

<Seção a ser Inserida: Primário Numérico: Regra 1>

<Seção a ser Inserida: Primário Numérico: Regra 2>

<Seção a ser Inserida: Primário Numérico: Regra 3>

<Seção a ser Inserida: Primário Numérico: Operadores Adicionais>

<Seção a ser Inserida: Primário Numérico: Regra 4>

```
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
```

```
                                        TYPE_T_NUMERIC);
  return false;
}
```

As regras que usamos para reconhecer a expressão são:

1) Se a expressão for composta por um único token, se começar com "(" e terminar com ")" ou se for composta por três tokens, um numérico, um / e outro numérico, então toda a expressão é um átomo numérico:

Seção: Primário Numérico: Regra 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) Se encontramos o operador `length`, apenas checamos se a expressão depois dele é numérica, e se for, calculamos o seu módulo. Se for de outro tipo, será definido depois como iremos calcular. Isso assume que temos uma função que identifica o tipo de uma expressão.

Seção: Primário Numérico: Regra 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;
  }
                    <Seção a ser Inserida: Avalia 'length'>
  else{
    RAISE_ERROR_UNSUPORTED_LENGTH_OPERAND(mf, cx, OPTIONAL(begin -> line),
                                          expr_type);
    return false;
  }
}
```

Note que identificar se estamos ou não em uma expressão numérica primária envolve usar a função `get_primatry_expression_type`. Esta função é definida na Subseção 8.8. Mas por hora, só é necessário saber que ela recebe o começo e fim de uma expressão e à partir daí retorna o tipo da expressão. Se o tipo for numérico, o código acima mostra como calculamos o módulo. Se a expressão depois de `length` não for numérica, ela será avaliada por outras regras que definiremos depois (nas Subseções 8.2.4 e 8.4.5).

3) Se encontrarmos um operador numérico, temos então um operador numérico seguido de um primário numérico.

O primeiro dos operadores numéricos é o de raíz quadrada:

Seção: Primário Numérico: Regra 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;
}
```

Em seguida, temos o `sind`, que interpreta o próximo número em graus ("degrees", por isso a letra "d" no fim) e calcula seu seno:

Seção: Primário Numérico: Regra 3 (continuação):

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

O cálculo do cosseno:

Seção: Primário Numérico: Regra 3 (continuação):

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

Em seguida calculamos o logaritmo na base e:

Seção: Primário Numérico: Regra 3 (continuação):

```
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;
}
```

Assim é como calculamos a exponencial correspondente à $expx = e^x$:

Seção: Primário Numérico: Regra 3 (continuação):

```
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;
}
```

O piso de um valor:

Seção: Primário Numérico: Regra 3 (continuação):

```
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;
}
```

Com relação ao operador `uniformdeviate`, o que ele faz é gerar um número uniforme e aleatório entre 0 e o valor passado para o operador. O modo que usaremos para fazer isso é gerar um número aleatório em ponto flutuante entre 0 e 1, para em seguida multiplicá-lo pelo operando.

Gerar um número em ponto flutuante entre 0 e 1 seguindo uma distribuição próxima à uniforme, basicamente podemos gerar um inteiro aleatório de 64 bits e multiplicá-lo por 2^{-64}. Nem todos os valores em ponto flutuante podem ser gerados desta forma, estaremos ignorando valores menores que 2^{-64} e o arredondamento fará com que alguns números mais próximos de 1 sejam mais comuns, embora a densidade destes números mais comuns também seja menor. Entretanto, o resultado será suficientemente próximo para nossos propósitos:

Seção: Primário Numérico: Regra 3 (continuação):

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

O próximo operador numérico é o símbolo de `+`. Este símbolo significa uma multiplicação por 1, e pode ser ignorado:

Seção: Primário Numérico: Regra 3 (continuação):

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

Já se tivermos um símbolo de `-`, então isso significa uma multiplicação por -1:

Seção: Primário Numérico: Regra 3 (continuação):

```
else if(begin -> type == TYPE_SUBTRACT){
  if(!eval_numeric_primary(mf, cx, begin -> next, end, result))
    return false;
  result -> value *= -1;
  return true;
}
```

4) Nos demais casos, temos uma multiplicação escalar em que o escalar é um token numérico primário, que não é sucedido por `+`, `-` ou por outro token numérico. Para lidar com isso, teremos que identificar o começo e o fim do token numérico primário. Pelas regras ele é um único token numérico, ou então três tokens (dois tokens numéricos separados pelo token `/` representando uma

fração). Depois de realizar a separação, a primeira parte é multiplicada pela segunda (que é interpretada como primário numérico):

Seção: Primário Numérico: Regra 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. Números Isolados e Valores Aleatórios Normais

As regras finais para as expressões numéricas que iremos tratar são:

```
<Átomo Numérico> -> <Variável Numérica> |
                    <Token Numérico Primário> |
                    normaldeviate |
                    ( <Expressão Numérica> )
```

O único token novo a ser tratado é `normaldeviate`:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_NORMALDEVIATE,  // O token simbólico 'normaldeviate'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"normaldeviate",
```

Este operador serve para gerar um novo número aleatório com uma distribuição normal com média 0 e desvio padrão 1.

A função que interpreta átomos numéricos é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Vamos decidir qual regra aplicar para interpretar o átomo primeiro com base nele ser um único token ou não. E depois, aplicamos diferentes regras em cada caso:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
                <Seção a ser Inserida: Átomo Numérico: Regra 1>
                <Seção a ser Inserida: Átomo Numérico: Regra 2>
                <Seção a ser Inserida: Átomo Numérico: Regra 3>
  }
  else{
                <Seção a ser Inserida: Átomo Numérico: Regra 4>
                <Seção a ser Inserida: Átomo Numérico: Regra 5>
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_NUMERIC);
  return false;
}
```

1) Se temos um único token e ele é um token numérico, basta retornar o seu valor:

Seção: Átomo Numérico: Regra 1:

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

2) Se temos um único token e ele é uma variável, retornamos o conteúdo da variável. Mas temos que checar se ela foi declarada, se é numérica e se foi inicializada. Também tratamos de maneira especial as variáveis `w`, `h` e `d`:

Seção: Átomo Numérico: Regra 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) Por fim, se temos um único token e ele é `normaldeviate`, então temos que gerar um número aleatório de uma distribuição normal. Dado que temos uma função que gera números aleatórios, podemos fazer isso procedendo da seguinte forma:

a) Geramos dois números aleatórios e uniformes entre -1 e 1, os quais chamaremos de u e v. Isso pode ser feito obtendo 64 bits, multiplicando os 63 primeiros por 2^{-63} e usando o último bit para definir o sinal. O resultado é suficientemente próximo de uniforme para nossos propósitos.

b) Se $u^2 + v^2 \geq 1$, isso significa que eles correspondem a pontos fora de um círculo de raio 1. Neste caso, os descartamos e voltamos ao passo 1. Também voltamos ao passo um se ambos forem zero, pois neste caso o método não funcionaria.

c) Isso permite gerar dois valores que terão uma distribuição uniforme:

$$x_0 = u\sqrt{-2\ln(u^2 + v^2)/(u^2 + v^2)}$$

$$x_1 = v\sqrt{-2\ln(u^2 + v^2)/(u^2 + v^2)}$$

Um dos valores pode ser retornado. O outro pode ser armazenado na estrutura Metafont:

Seção: Atributos (struct metafont) (continuação):

```
  bool have_stored_normaldeviate;
  float normaldeviate;
```

Inicialmente a estrutura não tem nenhum valor destes armazenado, vamos armazenar nela só depois de usarmos o método acima:

Seção: Inicialização (struct metafont) (continuação):

```
mf -> have_stored_normaldeviate = false;
```

Então, quando precisarmos gerar um valor aleatório de uma distribuição normal, sempre verificamos se existe um valor pré-obtido, e se não nós geramos os dois valores:

Seção: Átomo Numérico: Regra 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) Agora os casos em que o átomo numérico tem mais de um token. O primeiro caso é quanto o primeiro token é "(" e o último é ")". Neste caso, a parte interna aos parênteses é interpretada como uma expressão numérica:

Seção: Átomo Numérico: Regra 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;
}
```

Se apenas encontramos um parênteses vazio, isso é um erro. O usuário parece ter simplesmente esquecido de colocar uma expressão dentro dele.

5) Finalmente, o caso em que o átomo numérico é uma fração composta por um token numérico, / e outro token numérico. O resultado deve ser obtido dividindo ambos os tokens numéricos:

Seção: Átomo Numérico: Regra 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. Atribuições e Expressões de Pares

Para realizar a atribuição de pares a variáveis do tipo certo, usamos o código abaixo:

Seção: Atribuição de Variável de Par:

```
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);
  }
}
```

Vamos agora à avaliação de expressões de pares.

8.2.1. Soma e Subtração

As regras gramaticais para as expressões de pares começam com:

```
<Expressão de Par> -> <Terciário de Par>
<Terciário de Par> -> <Secundário de Par> |
                      <Terciário de Par> <PT-Op> <Secundário de Par>
<PT-Op> -> + | -
```

A soma e subtração é tratada exatamente como se espera de uma soma e subtração de vetores.

A função que avalia expressões de pares é declarada aqui:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

O método de avaliar as expressões de pares terciárias não é diferente do que já fizemos com as expressões numéricas. Apenas temos menos operadores terciários aqui.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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{ // Encontra último operador terciário de '+' ou '-'
    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){ // Avalia '+'
      result -> x = a.x + b.x;
      result -> y = a.y + b.y;
    }
    else if(last_sum -> type == TYPE_SUBTRACT){ // Avalia '-'
      result -> x = a.x - b.x;
      result -> y = a.y - b.y;
    }
    return true;
  }
  else // Nenhum operador terciário:
    return eval_pair_secondary(mf, cx, begin, end, result);
}
```

8.2.2. Transformações, Multiplicação e Divisão Escalar

A gramática para expressões de pares secundárias é:

```
<Secundário de Par> -> <Primário de Par> |
                       <Par Secundário><Mul ou Div><Numérico Primário> |
                       <Secundário Numérico> * <Primário de Par> |
                       <Secundário de Par><Transformador>
<Mul ou Div> -> * | /
<Transformador> -> rotated <Primário Numérico> |
                   scaled <Primário Numérico> |
                   shifted <Par Primário> |
                   slanted <Numérico Primário> |
                   xscaled <Numérico Primário> |
                   yscaled <Numérico Primário> |
                   zscaled <Par Primário> | (...)
```

As regras acima sobre transformadores estão incompletas, pois veremos ainda na Subseção 8.3.1 que há mais um tipo de transformador a ser definido.

Por hora, vamos adicionar as sete novas palavras-chave representando transformadores conhecidos:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_ROTATED,  // O token simbólico 'rotated'
TYPE_SCALED,   // O token simbólico 'scaled'
TYPE_SHIFTED,  // O token simbólico 'shifted'
TYPE_SLANTED,  // O token simbólico 'slanted'
TYPE_XSCALED,  // O token simbólico 'xscaled'
TYPE_YSCALED,  // O token simbólico 'yscaled'
TYPE_ZSCALED,  // O token simbólico 'zscaled'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"rotated", "scaled", "shifted", "slanted", "xscaled", "yscaled",
"zscaled",
```

A declaração da função que avaliará expressões secundárias de pares:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

```
                                struct pair_variable *result);
```

Interpretar uma expressão secundária aqui é similar ao que fizemos para as expressões numéricas. Também devemos percorrer a lista de tokens até achar a operação secundária mais à direita, ignorando aquilo que está aninhado entre parênteses e colchetes. Deve-se seguir as mesmas regras para determinar se o / é mesmo uma divisão ou se é uma fração. Mas como aqui temos um total de nove operadores secundários, incluindo os transformadores. Devido à quantidade, vamos mostrar separadamente cada um deles ao invés de colocá-los todos neste bloco de código abaixo:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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{ // Encontra o operador secundário mais à direita
    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 ||
              // 'transformed' será definido na Subseção 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) // Separador de fração
         last_fraction = p;
       else{ //  Divisão ou operadores válidos
         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;
      <Seção a ser Inserida: Operador Secundário de Pares: Multiplicação>
        <Seção a ser Inserida: Operador Secundário de Pares: Divisão>
        <Seção a ser Inserida: Operador Secundário de Pares: Rotação>
         <Seção a ser Inserida: Operador Secundário de Pares: Escala>
      <Seção a ser Inserida: Operador Secundário de Pares: Deslocamento>
       <Seção a ser Inserida: Operador Secundário de Pares: Inclinação>
        <Seção a ser Inserida: Operador Secundário de Pares: X-Escala>
        <Seção a ser Inserida: Operador Secundário de Pares: Y-Escala>
```

<Seção a ser Inserida: Operador Secundário de Pares: Z-Escala>
<Seção a ser Inserida: Operador Secundário de Pares: Outros>

```
  }
  else
    return eval_pair_primary(mf, cx, begin, end, result);
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PAIR);
  return false;
}
```

O primeiro operador é o de multiplicação. Note que podemos ter dois tipos de multiplicação: um par por um numérico ou um numérico por um par. Para identificar qual dos dois tipos se aplica, devemos examinar o tipo da expressão à direita do operador.

Seção: Operador Secundário de Pares: Multiplicação:

```
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;
  }
}
```

Se temos uma divisão, então sempre será um par dividido por um numérico. Devemos gerar erro em caso de divisão por zero:

Seção: Operador Secundário de Pares: Divisão:

```
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;
}
```

Se temos uma rotação, rotacionamos nosso par no sentido anti-horário em relação à origem, interpretando o ângulo em graus, não radianos:

Seção: Operador Secundário de Pares: Rotação:

```
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 grau = 0,0174533 radianos
  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;
}
```

Se temos uma mudança de escala, isso é sinônimo a uma multiplicação:

Seção: Operador Secundário de Pares: Escala:

```
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;
}
```

Um deslocamento é igual a uma soma, mas tem uma ordem de precedência maior:

Seção: Operador Secundário de Pares: Deslocamento:

```
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;
}
```

O operador de inclinação desloca mais à direita um ponto quanto mais acima do eixo x ele está e mais à esquerda quanto mais abaixo do eixo x ele está:

Seção: Operador Secundário de Pares: Inclinação:

```
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;
}
```

Mudança de escala no eixo x multiplica um escalar apenas pelo primeiro valor do par:

Seção: Operador Secundário de Pares: X-Escala:

```
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;
}
```

Assim como no eixo y multiplica o escalar só pelo segundo valor do par:

Seção: Operador Secundário de Pares: Y-Escala:

```
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;
}
```

Por fim, a mudança de escala z înterpreta dois pares como números complexos e os multiplica:

$$(a + bi)(c + di) = ac + (ad)i + (cb)i + (bd)i^2 = (ac - bd) + (cb + ad)i$$

Seção: Operador Secundário de Pares: Z-Escala:

```
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. Valores Intermediários de Pares, Literais e Variáveis

As regras gramaticais finais para a expressão de pares é:

```
<Primário de Par> -> <Variável de Par> |
                     ( <Expressão Numérica> , <Expressão Numérica> ) |
                     ( <Expressão de Par> ) |
                     (...) |
                     <Átomo Numérico>[<Expressão de Par,
                                      <Expressão de Par]  |
                     <Operador de Multiplicação Escalar><Primário de Par>
```

Temos uma parte omitida na regra gramatical acima porque há alguns operadores primários que vamos definir só nas Subseções 8.4.6 e 11.4.

Uma novidade é a construção do tipo $a[b, c]$ onde b e c são pares. Ela representa um valor intermediário entre os pontos. Basicamente é avaliado como $a(b+c)$, de modo que $.5[b, c]$ representa o meio do caminho entre os dois pontos.

As outras regras são análogas às que já vimos na gramática de expressões numéricas.

A função que avaliará expressões de pares primárias é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E cada uma das cinco regras gramaticais acima será testada separadamente para sabermos qual regra devemos aplicar ao encontrar uma expressão primária:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
                <Seção a ser Inserida: Primário de Par: Regra 1>
  }
  else if(begin -> type == TYPE_OPEN_PARENTHESIS &&
          end -> type == TYPE_CLOSE_PARENTHESIS){
                <Seção a ser Inserida: Primário de Par: Regra 2>
                <Seção a ser Inserida: Primário de Par: Regra 3>
  }
    <Seção a ser Inserida: Primário de Par: Outras Regras a Definir Depois>
                <Seção a ser Inserida: Primário de Par: Regra 4>
                <Seção a ser Inserida: Primário de Par: Regra 5>
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PAIR);
  return false;
}
```

A primeira regra será aplicada quando temos um único token na expressão. O único caso em que isso ocorre é quando a expressão é uma variável de par:

Seção: Primário de Par: Regra 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;
```

Se a expressão é delimitada por parênteses, nós tanto podemos estar diante de uma representação literal de um par (a, b) como podemos estar diante de parênteses com uma expressão de par completa dentro dele como em $(pair1 + (a, b))$. Podemos diferenciar os dois casos pela vírgula interna no parênteses, a qual está no mesmo escopo que a e b, não deve estar dentro de outros parênteses e colchetes.

Seção: Primário de Par: Regra 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;
  }
```

No código acima nós identificamos se estamos diante de um literal pela presença da vírgula e nós identificamos isso marcando na variável booleana `literal`. Portanto, se esta variável não for verdadeira, imediatamente após o último `if` acima, executamos o `else` como a próxima regra:

Seção: Primário de Par: Regra 3:

```
else
  return eval_pair_expression(mf, cx, begin_a, end_a, result);
```

Se o último token for um `]`, então temos uma construção do tipo $a[B, C]$ onde a é numérico enquanto B e C são pares. Nossa tarefa é separar essas três partes a, B e C, interpretá-las e retornar $B + a(C - B)$:

Seção: Primário de Par: Regra 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: do começo da expressão até antes de '['
    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: começa após '['
  end_b = begin_b;
  while(end_b != end){ // b: Termina antes de ','
    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: Começa após ','
  end_c = begin_c;
  while(end_c != end){ // c: termina no penúltimo 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)) // Avalia 'a'
    return false;
  if(!eval_pair_expression(mf, cx, begin_b, end_b, &b)) // Avalia 'b'
    return false;
  if(!eval_pair_expression(mf, cx, begin_c, end_c, &c)) // Avalia 'c'
    return false;
  result -> x = b.x + a.value * (c.x - b.x); // resultado = b +a(c-b)
  result -> y = b.y + a.value * (c.y - b.y);
  return true;
}
```

A última regra é quando temos um operador de multiplicação escalar e um par. O operador pode ser um token de `+`, `-`, um único token numérico, ou então uma fração. Temos um exemplo de cada um dos quatro casos abaixo:

```
pair a, b, c, d;
a = +(1, 2);
b = -(1, 2);
c = 2a;
d = 1/2b;
```

O código que trata tais expressões é:

Seção: Primário de Par: Regra 5:

```
else{
  if(begin -> type == TYPE_SUM) // Um '+' unário antes da expressão
    return eval_pair_primary(mf, cx, begin -> next, end, result);
  else if(begin -> type == TYPE_SUBTRACT){ // Um '-' unário
    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){ // Número/fração antes de expressão
    struct generic_token *tok;
    float value = ((struct numeric_token *) begin) -> value;
    tok = begin -> next;
    if(tok -> type == TYPE_DIVISION){ // É uma fração
      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. Pares em Expressões Numéricas

Subexpressões numéricas podem aparecer dentro de expressões de pares. Por exemplo, em $a[b, c]$ onde a é um átomo numérico. Da mesma forma, subexpressões de pares podem aparecer em expressões numéricas. Mas não definimos os casos quando definimos as expressões numéricas porque então não tínhamos ainda definido como avaliar expressões de pares. Existem quatro operadores numéricos primários que envolvem avaliar pares:

```
<Primário Numérico> -> length <Par Primário> | xpart <Par Primário> |
                       ypart <Par Primário> | angle <Par Primário>
```

Isso requer definir os seguintes novos tipos de tokens:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_XPART,  // O token simbólico 'xpart'
TYPE_YPART,  // O token simbólico 'ypart'
TYPE_ANGLE,  // O token simbólico 'angle'
```

Correspondentes às seguintes palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"xpart", "ypart", "angle",
```

O primeiro caso, que não precisou de um token novo, `length` já era usado para obter o módulo de números e havíamos avisado que ele poderia ser usado para outros tipos. No caso de pares, ele mede a norma euclideana deles:

Seção: Avalia '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;
}
```

O operador `xpart` simplesmente retorna o primeiro valor de um par. Entretanto, devemos ter um cuidado adicional: este operador pode ser usado em outro contexto como ainda será visto na Subseção 8.3.3. Então devemos primeiro nos certificar que o que vem depois do operador é um par. Em caso afirmativo, retornamos seu primeiro valor:

Seção: Primário Numérico: Operadores Adicionais:

```
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{
        <Seção a ser Inserida: Primário Numérico: X-Part em Não-Par>
  }
}
```

Enquanto o operador `ypart` retorna o segundo valor de um par. Neste caso também devemos ter o mesmo cuidado de verificar se o operando que temos depois é mesmo um par ou de algum outro tipo:

Seção: Primário Numérico: Operadores Adicionais (continuação):

```
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{
        <Seção a ser Inserida: Primário Numérico: Y-Part em Não-Par>
  }
}
```

Por fim, o último operador, `angle` retorna o ângulo de um par. É o ângulo do segmento que conecta a origem ao par em relação ao segmento que conecta a origem à $(1, 0)$. Um erro deve ser gerado se tentar medir o ângulo de $(0, 0)$:

Seção: Primário Numérico: Operadores Adicionais:

```
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; // Radianos para graus
  return true;
}
```

8.3. Atribuições e Expressões de Transformação

Um dos momentos em que esperamos avaliar uma expressão de transformação é quando formos fazer uma atribuição para uma variável de transformação. Neste caso, devemos avaliar a expressão

e realizar a atribuição para todas as variáveis que estiverem do lado esquerdo da expressão de atribuição:

Seção: Atribuição de Variável de Transformação:

```
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);
  }
}
```

Vamos agora ver como avaliar as expressões de transformação.

8.3.1. Transformações sobre Transformadores

A regra gramatical para avaliar uma expressão de transformação começam como:

```
<Expressão de Transformação> -> <Terciário de Transformação>
<Terciário de Transformação> -> <Secundário de Transformação>
<Secundário de Transformação> -> <Secundário de Transformação> <Transformador> |
                                 <Primário de Transformação>
<Transformador> -> rotated <Primário Numérico> |
                   scaled <Primário Numérico> |
                   shifted <Par Primário> |
                   slanted <Numérico Primário> |
                   xscaled <Numérico Primário> |
                   yscaled <Numérico Primário> |
                   zscaled <Par Primário>
                   transformed <Primário de Transformação>
```

Temos aqui um novo tipo de token, representando o último tipo de transformador que não havíamos definido na subseção anterior sobre expressões de pares:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_TRANSFORMED,  // O token simbólico 'transformed'
```

Ele corresponde à seguinte palavra reservada:

Seção: Lista de Palavras Reservadas (continuação):

```
"transformed",
```

Com relação às regras gramaticais, elas dizem que não existem no momento operadores terciários de transformação. De qualquer forma, vamos definir a função que avalia expressões, e avalia operadores terciários apenas como uma função que chama o avaliador de operadores secundários. Tanto para manter a uniformidade das funções que avaliam expressões como para deixar essa função pronta para ser modificada caso versões futuras da linguagem venham a trazer operadores terciários de transformação.

A declaração da função:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

```
                                struct transform_variable *result);
```

E sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
}
```

Já a função que avalia expressões secundárias de transformação é esta:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

A função funciona percorrendo a lista de tokens da expressão, ignorando tokens dentro de parênteses, colchetes e chaves. Toda vez que encontra o último operador secundário, ele é armazenado em uma variável. Depois de percorrer todos os tokens, se não achamos um operador secundário, nós passamos a expressão inteira para o avaliador de expressões primárias. Se encontramos um, tudo que vem antes dele é avaliado como expressão secundária e o resultado passa pelo operador, que será operador transformador.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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{ // Encontra operador de transformação mais à direita:
    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){ // Existe um operador secundário:
    if(end_secondary == NULL){ // Mas não tem nada antes dele:
      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;
      <Seção a ser Inserida: Aplica Operador Secundário de Transformação>
  }
  else // Nenhum operador secundário:
    return eval_transform_primary(mf, cx, begin, end, result);
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_TRANSFORM);
  return false;
}
```

Vamos agora aplicar os diferentes transformadores sobre as nossas variáveis e resultados de transformação. O primeiro tipo de transformador é a rotação, que aplicamos sobre um transformador com o código abaixo:

Seção: Aplica Operador Secundário de Transformação:

```
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 grau = 0,0174533 radianos
  angle_radian = theta.value * 0.0174533;
  TRANSFORM_ROTATE(result -> value, angle_radian);
  return true;
}
```

Já mudar a escala de um transformador, ampliando ele s vezes é feito pelo código:

Seção: Aplica Operador Secundário de Transformação (continuação):

```
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;
}
```

A próxima transformação é realizar uma translação, ou deslocamento (x, y) sobre uma transformação.

Seção: Aplica Operador Secundário de Transformação (continuação):

```
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;
}
```

Agora vamos tratar o transformador que inclina a nossa transformação uma quantia s.

Seção: Aplica Operador Secundário de Transformação (continuação):

```
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;
}
```

A mudança de escala somente no eixo x estica ou comprime a transformação neste eixo. Isso é obtido pelo código abaixo:

Seção: Aplica Operador Secundário de Transformação (continuação):

```
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;
}
```

Podemos também fazer a mudança de escala somente no eixo y usando o fator de mudança s:

Seção: Aplica Operador Secundário de Transformação (continuação):

```
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;
}
```

Por fim, há a mudança de escala no plano complexo, que envolve multiplicar os pontos por um par (s, t), interpretado como um número complexo. Isso obtém ao mesmo tempo tanto rotação como mudança de escala. Esse transformador é implementado pelo código abaixo:

Seção: Aplica Operador Secundário de Transformação (continuação):

```
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;
}
```

Por fim, o último tipo de transformador. Quando temos duas transformações e as multiplicamos para combiná-las em uma só. Isso é feito simplesmente multiplicando as matrizes que representam a transformação:

Seção: Aplica Operador Secundário de Transformação (continuação):

```
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 Expressões Primárias de Transformação: Literais e Variáveis

Continuando a gramática das expressões de transformadores, as próximas regras são:

```
<Primário de Transformação> -> <Variável de Transformação> |
```

```
                                    ( <Terciário de Transformação> ) |
                                    ( <Expressão Numérica> , <Expressão Numérica> ,
                                      <Expressão Numérica> , <Expressão Numérica> ,
                                      <Expressão Numérica> , <Expressão Numérica> )
```

No primeiro caso, temos só um token simbólico com uma variável de transformação. Nos dois outros casos, o primeiro Token é o abrir de parênteses. No primeiro, há só uma expressão entre parênteses que avaliará para um transformador. No segundo, temos seis expressões numéricas que formarão os valores de nosso transformador.

A função que avaliará as expressões primárias de transformação é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E a sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){ // É uma variável?
            <Seção a ser Inserida: Transformação Primária: Variável>
  }
  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 && p != end);
    if(has_comma){
              <Seção a ser Inserida: Transformação Primária: Literal>
    }
    else{
            <Seção a ser Inserida: Transformação Primária: Parênteses>
    }
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
```

```
                                    TYPE_T_TRANSFORM);
  return false;
}
```

O código acima identifica se estamos diante de um dos três casos de expressão primária de transformador: temos uma variável, um literal ou uma subexpressão delimitada por parênteses. A variável é mais fácil de identificar, pois é o único caso em que há um único token simbólico. Tanto o literal como a subexpressão são delimitados por parênteses. O modo de diferenciar as duas coisas é pela presença de vírgula, que encontramos no literal, mas não na sub-expressão.

O caso mais simples é quando estamos só lendo uma variável:

Seção: Transformação Primária: Variável:

```
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, 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;
```

O próximo caso que trataremos é quando temos uma subexpressão entre parênteses. Devemos apenas avaliar ela após descartar os parênteses delimitadores:

Seção: Transformação Primária: Parênteses:

```
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);
```

E finalmente, quando temos um literal de transformação formado por seis expressões numéricas separadas por vírgulas:

Seção: Transformação Primária: 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]; // Cada um dos 6 valores será lido aqui
for(i = 0; i < 6; i ++){
  p = begin_numeric_expr;
  do{
    if(p != end){ // Vamos ignorar o último ')'
      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;
}
// Armazenando na ordem correta dentro da matriz:
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;
```

O código acima tem dois laços. O primeiro tem como invariante de que no começo de cada uma de suas iterações, `p` e `begin_numeric_expr` apontam para o token marcando o começo da próxima expressão numérica no literal que ainda não foi avaliada. O segundo loop tem como tarefa fazer então com que `end_numeric_expr` passe a marcar corretamente o fim dessa expressão numérica, delimitada pela próxima vírgula ou pelo fechamento de parênteses final do literal. Depois da execução do loop mais interno, já temos a expressão delimitada, avaliamos ela e armazenamos seu valor. Repetimos isso seis vezes para ler o literal de transformação inteiro.

8.3.3. Transformações em Expressões Numéricas

Transformações podem aparecer dentro de expressões primárias numéricas. Existem as seguintes regras gramaticais adicionais que as prevêem:

```
<Primário Numérico> -> <Parte de Transformação><Primário de Transformação>
<Parte de Transformação> -> xpart | ypart | xxpart | xypart | yxpart | yypart
```

Os seis operadores acima servem para extrair cada um dos seis valores numéricos que compõe uma transformação. A ordem em que aparecem acima é a mesma ordem na qual eles extraem o elemento da transformação.

Como `xpart` e `ypart` já havia aparecido quando definíamos as expressões de pares, resta definirmos tokens para os outros quatro operadores que são novos:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_XXPART,  // O token simbólico xxpart'
TYPE_XYPART,  // O token simbólico xypart'
TYPE_YXPART,  // O token simbólico yxpart'
TYPE_YYPART,  // O token simbólico yypart'
```

E as palavras reservadas associadas a cada token:

Seção: Lista de Palavras Reservadas (continuação):

```
"xxpart", "xypart", "yxpart", "yypart",
```

Estes quatro novos operadores são tratados na função que avalia expressões numéricas primárias:

Seção: Primário Numérico: Operadores Adicionais (continuação):

```
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;
}
```

Mas e quanto ao `xpart` e `ypart`? Em princípio, eles já foram tratados para o caso em que extraimos valores de pares. Precisamos apenas incrementar a definição já feita na Subseção 8.2.4. para tratar o caso em que devemos aplicá-la quando não temos um par. Fazendo isso no caso do `xpart`:

Seção: Primário Numérico: X-Part em Não-Par:

```
struct transform_variable t;
if(!eval_transform_primary(mf, cx, begin -> next, end, &t))
  return false;
result -> value = t.value[6];
return true;
```

O mesmo deve ser feito no operador `ypart`:

Seção: Primário Numérico: Y-Part em Não-Par:

```
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. Transformações em Expressões de Pares

Transformações também aparecem nas regras gramaticais de expressões de pares. Existe uma expressão secundária que obtém uma expressão secundária de par do lado esquerdo e um primário de transformação do lado direito:

```
<Secundário de Par> -> <Secundário de Par><Transformador>
<Transformador> -> (...) | transformed <Primário de Transformação>
```

O que este operador faz é aplicar a transformação linear sobre o par. Vamos implementá-lo com o código adicional abaixo que irá se somar ao código que implementa transformações sobre pares da Subseção 8.2.2.

Seção: Operador Secundário de Pares: Outros:

```
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. Atribuições e Expressões de Caminhos

Para realizar a atribuição de caminhos a variáveis do tipo certo, usamos o código abaixo:

Seção: Atribuição de Variável de Caminho:

```
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' começa na variável atribuída
  for(i = 0; i < number_of_variables; i ++){
    struct path_variable *dst = (struct path_variable *) var -> var;
    if(dst -> permanent){ // Como alocar e desalocar
      alloc = permanent_alloc;
      disalloc = permanent_free;
    }
    else{
      alloc = temporary_alloc;
      disalloc = temporary_free;
    }
    // Se o destino não está vazio, desaloca ele:
    if(dst -> length != -1 && disalloc != NULL)
      path_recursive_free(disalloc, dst, false);
    // Copia a origem para destino
    if(!recursive_copy_points(mf, NULL, alloc, &dst, &result, false))
      return false;
    var = (struct symbolic_token *) (var -> next); // 'var' aponta para '='
    var = (struct symbolic_token *) (var -> next); // 'var' é próxima variável
  }
  if(temporary_free != NULL)
    path_recursive_free(temporary_free, &result, false);
}
```

Veremos agora na próxima Subseção as regras gramaticais para as expressões de caminhos:

8.4.1. Junção de Caminhos

A gramática para a expressão de caminhos começa com:

```
<Expressão de Caminho> -> <Expressão de Par> | <Terciário de Caminho> |
                          <Sub-expressão de Caminho><Especificador de Direção> |
                          <Sub-expressão de Caminho><Junção de Caminho> cycle
<Junção de Caminho> -> <Especificador de Direção><Junção Básica>
                       <Especificador de Direção>
<Junção Básica> -> & | .. | .. <Tensão> .. | .. <Controles> .. | --
<Tensão> -> tension <Quantidade de Tensão> |
```

```
            tension <Quantidade de Tensão> and <Quantidade de Tensão>
<Quantidade de Tensão> -> <Primário Numérico> | atleast <Primário Numérico>
<Controles> -> controls <Par Primário> |
               controls <Par Primário> and <Par Primário>
<Especificação de Direção> -> Vazio |
                              { <Expressão de Par } |
                              { <Expressão Numérica> , <Expressão Numérica> } |
                              { curl <Expressão Numérica> }
<Sub-expressão de Caminho> -> <Expressão de Caminho> |
                              <Sub-expressão de Caminho><Junção de Caminho>
                              <Terciário de Caminho>
```

Tudo isso requer que registremos os seguintes novos tipos de tokens:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_CYCLE,          // O token simbólico 'cycle'
TYPE_AMPERSAND,      // O token simbólico '&'
TYPE_JOIN,           // O token simbólico '..'
TYPE_TENSION,        // O token simbólico 'tension'
TYPE_AND,            // O token simbólico 'and'
TYPE_ATLEAST,        // O token simbólico 'atleast'
TYPE_CONTROLS,       // O token simbólico 'controls'
TYPE_CURL,           // O token simbólico 'curl'
TYPE_STRAIGHT_JOIN,  // O token simbólico '--'
```

E para cada um deles adicionemos sua string na lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"cycle", "&", "..", "tension", "and", "atleast", "controls", "curl", "--",
```

O que as regras gramaticais revelam é que toda expressão de caminho é formado pela junção de vários outros sub-caminhos. Se não existir nenhuma junção, então nosso caminho é um par, que é considerado um caminho de um único ponto.

Uma das coisas que temos a fazer então é contar quantas junções existem em uma expressão de caminho. Realizar a contagem pode ser feito com a seguinte função auxiliar:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
int count_path_joins(struct generic_token *begin, struct generic_token *end);
```

A função funciona contando o número de "&", "–" e também contando os ".." quando eles aparecem sozinhos ou quando eles aparecem pela segunda vez dentro de uma junção que especifica pontos de controle ou tensão. As junções dentro de sub-expressões dentro de parênteses e delimitadores não são contadas:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Contar o número de junções será importante para que saibamos a quantidade de pontos que devemos alocar inicialmente para um caminho. De posse de como obter tal informação, podemos começar a tratar as expressões. A variável que avalia expressão de caminho tem como cabeçalho:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

O número de pontos de um caminho é inicialmente o número de junções mais um:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
    <Seção a ser Inserida: Expressão de Caminho: Quando Não Há Junção>
  <Seção a ser Inserida: Expressão de Caminho: Alocar Variável de Caminho>
      <Seção a ser Inserida: Expressão de Caminho: Itera sobre Junções>
  // Caminhos são retornados normalizados, como visto na Subseção 7.4.4:
  return normalize_path(mf, cx, result);
}
```

E se o número de junções for zero? Neste caso, podemos estar diante de uma expressão de par como abaixo:

```
path p;
p = (2, 3);
```

Ou de uma expressão terciária de caminho onde as junções estão dentro de parênteses ou outros delimitadores como em:

```
path p;
p = ((2, 3)..(1, 2));
```

De qualquer forma, pode haver um ou mais especificador de direção depois da expressão, o qual precisa ser ignorado. No primeiro caso porque especificadores de direção não tem efeito algum sobre pontos isolados. No segundo porque a sub-expressão seria normalizada e já viria com todos os pontos de controle ao terminarmos de avaliar os parênteses. Então o especificador de direção novamente não teria efeito.

Podemos checar se temos especificadores de direção checando se o último token é um "}". Se for o caso, mudamos a posição do fim da expressão para imediatamente antes do primeiro especificador de direção, e consideramos somente esta parte como a que deve ser avaliada. Entretanto, temos que

fazer uma validação para garantir que o especificador adicional está de acordo com a gramática:

Seção: Expressão de Caminho: Quando Não Há Junção:

```
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));
    // O especificador é ignorado, mas precisa ser validado:
    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);
}
```

Se existir uma ou mais junções, então caberá a nós criar uma nova variável de caminho alocando as estruturas necessárias para ela. O que temos a fazer é alocar nela um número de pontos igual à variável `expected_length`:

Seção: Expressão de Caminho: Alocar Variável de Caminho:

```
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; // Inicialização
result -> permanent = false;
result -> number_of_points = 0;
result -> cyclic = false;
```

Agora temos que interpretar todas as junções. Temos que iterar sobre cada junção presente na expressão, as quais possuem o formato:

$$z_1 djez_2$$

Então vamos usar variáveis para indicar o começo e o fim de cada parte à medida que iteramos. Vamos criar um laço onde iremos iterar sobre cada uma das junções no formato acima para interpretá-las e vamos marcar o token de começo e de fim de cada parte:

Seção: Expressão de Caminho: Itera sobre Junções:

```
{
  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;
```
<Seção a ser Inserida: Expressão de Caminho: Prepara Valor Inicial de z1>
```
  while(end_z1 != end || result -> length < expected_length){
```
<Seção a ser Inserida: Expressão de Caminho: Separa Tokens da Junção>
<Seção a ser Inserida: Expressão de Caminho: Interpreta Extremidades da Junção>
<Seção a ser Inserida: Expressão de Caminho: Interpreta Direção se Existir>
<Seção a ser Inserida: Expressão de Caminho: Interpreta Junção>
```
    begin_z1 = begin_z2;
    end_z1 = end_z2;
  }
```
<Seção a ser Inserida: Expressão de Caminho: Após Ler as Junções>
```
}
```

Para delimitar o primeiro elemento z_1, na primeira vez quando ainda estamos no começo da expressão, basta avançarmos os tokens até acharmos o primeiro "", "..", "" ou "&" fora de qualquer parênteses ou colchetes. Nas demais vezes, o elemento z_1 deve começar como sendo igual àquilo que era o elemento z_2 na iteração anterior, o que já é garantido pelo código acima.

Seção: Expressão de Caminho: Prepara Valor Inicial de z1:

```
{
  DECLARE_NESTING_CONTROL();
  int next_type;
  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));
}
```

Agora vamos delimitar o primeiro especificador de direção. Primeiro lemos o próximo token para ver se é um "{". Se não for, não há o primeiro especificador de direção. Se for, então delimitamos ele até acharmos o próximo "}":

Seção: Expressão de Caminho: Separa Tokens da Junção:

```
begin_d = end_z1 -> next;
if(begin_d -> type != TYPE_OPEN_BRACES){ // Nenhum especificador
  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));
}
```

Agora vamos delimitar a junção. Se ela for um "&" ou um "–", ela corresponde a um único token. Se for um "..", devemos checar o próximo token para identificar se temos um "`controls`" ou "`tension`". Em caso afirmativo, teremos que delimitar a junção englobando do primeiro ".." até o segundo "..". Em caso negativo, a junção é somente um token "..".

Seção: Expressão de Caminho: Separa Tokens da Junção (continuação):

```
if(end_d == NULL)
  begin_j = end_z1 -> next; // Começa depois de z1
else
  begin_j = end_d -> next; // Ou depois de especificador de direção
end_j = begin_j; // Começamos assumindo que é só um token
if(begin_j == end){ // Mas é preciso ter algo depois dele
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
}
if(begin_j -> type == TYPE_JOIN && // Testando se é mais de um 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){ // Se for, o final é no próximo '..'
    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;
}
```

O próximo delimitador de direção pode existir ou não dependendo de termos um token "{" após a junção:

Seção: Expressão de Caminho: Separa Tokens da Junção (continuação):

```
begin_e = end_j -> next;
if(begin_e -> type != TYPE_OPEN_BRACES){ // Sem especificador
  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));
}
```

E por fim, o último elemento da junção é o próximo ponto da junção, que seria uma expressão de caminho terciária. Este próximo elemento tem seu fim delimitado pelo fim da expressão, por um token de abrir chaves, ou pelo começo da próxima junção:

Seção: Expressão de Caminho: Separa Tokens da Junção (continuação):

```
{
  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));
}
```

Uma vez que tenhamos identificado cada parte da junção, temos que interpretar cada uma delas. Primeiro devemos identificar os pontos das extremidades da junção e copiá-las para o resultado da expressão. Para isso primeiro interpretaremos z_0, a primeira extremidade de junção

e armazenaremos ela na posição correta do caminho que estamos interpretando.

Seção: Expressão de Caminho: Prepara Valor Inicial de z1 (continuação):

```
{
  struct path_variable z1;
  if(!eval_path_tertiary(mf, cx, begin_z1, end_z1, &z1))
    return false;
  // Se z1 é um único ponto:
  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{ // Se z1 é um subcaminho:
    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);
}
```

No início da iteração quando estamos passando pelos pontos a serem interpretados, sempre teremos uma variável z_1 que é um ponto ou sub-caminho que começará sendo a primeira parte do caminho sendo unida às outras com uma junção e que em cada iteração será o próximo valor. E teremos também uma variável z_2 que é a que sucede z_1 e é o valor que na iteração seguinte será assumido pelo z_1.

O código abaixo interpreta o z_2. Para este caso, devemos levar em conta que podemos estar diante de um token `cycle`. Se for o caso, devemos marcar o caminho interpretado como cíclico e devemos copiar o ponto inicial para esta posição. Caso contrário, concluímos que o caminho não é cíclico e interpretamos o ponto z_2 de maneira similar a como interpretamos o primeiro ponto z_0:

Seção: Expressão de Caminho: Interpreta Extremidades da Junção (continuação):

```
if(begin_z2 == end_z2 && begin_z2 -> type == TYPE_CYCLE){ // Lemos 'cycle'
  struct path_points *p = result -> points;
  // O primeiro ponto pode ser interno a sub-caminho:
  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 é 1 ponto:
    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 é um sub-caminho:
    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);
}
```

Note que uma construção como a abaixo é perfeitamente legal:

```
path p;
p = (0, 0) .. (1, 2) .. cycle .. (2, 4) .. cycle .. cycle;
```

Cada ponto que não é um `cycle` faz o interpretador concluir que não está diante de um caminho cíclico, e cada `cycle` faz ele chegar à conclusão oposta. Então é somente o último elemento da junção que determina se o caminho será tratado como cíclico ou não. Nos demais casos, este token é apenas um sinônimo para o primeiro ponto.

O código que escrevemos até então, garante que todos os pontos de extremidade serão corretamente preenchidos. Mas nós não devemos preencher somente eles, mas também a informação de como eles são unidos, dados pontos de controle, tensão e especificadores de direção. Mas para preencher esta informação, precisamos de um ponteiro que mostre quais são os valores exatos dos pontos de extremidade que estamos unindo (pontos z_1 e z_2) além do ponto anterior se houver (ponto z_0). Aparentemente, seria só consultar `result -> points[result -> length - 2]` e `result -> points[result -> length - 1]` para obter z_1 e z_2. Entretanto, estes valores podem não ser pontos isolados, eles podem ser sub-caminhos. Neste caso, para achar os verdadeiros pontos de extremidade a serem unidos, temos que percorrer o sub-caminho:

Seção: Expressão de Caminho: Interpreta Extremidades da Junção (continuação):

```
// z0 é o ponto anterior, z1 é o atual e z2 é o próximo
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]);
}
```

Agora temos que interpretar os dois especificadores de direção. Eles podem ser vazios (equivalente a ter uma direção $(0, 0)$), podem ser um par especificando um vetor de direção ou então podem ser um valor numérico de "enrolamento" (`curl`).

A função que lê ambos os especificadores é chamada para isso e preenche corretamente eles caso estejam sendo explicitamente escritos:

Seção: Expressão de Caminho: Interpreta Direção se Existir:

```
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;
```

A função que avalia os especificadores de direção é declarada aqui:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool eval_direction_specifier(struct metafont *mf, struct context *cx,
                              struct generic_token *begin,
                              struct generic_token *end, float *w_x,
                              float *w_y);
```

E ela funciona checando quatro casos diferentes: quando não há especificador nenhum, quando é um especificador de "enrolamento" ("`curl`"), quando é um de direção no formato de dois números ou quando é um de direção no formato de um par:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
bool eval_direction_specifier(struct metafont *mf, struct context *cx,
                              struct generic_token *begin,
                              struct generic_token *end, float *w_x,
                              float *w_y){
  // Detectar erro de especificador vazio:
  if(begin != NULL && begin -> next == end){
```

```
    RAISE_ERROR_EMPTY_DELIMITER(mf, cx, OPTIONAL(begin -> line), '{');
    return false;
  }
```

<Seção a ser Inserida: Especificador de Direção: Caso 1>
<Seção a ser Inserida: Especificador de Direção: Caso 2>
<Seção a ser Inserida: Especificador de Direção: Caso 3>
<Seção a ser Inserida: Especificador de Direção: Caso 4>

```
  return false;
}
```

Quando um especificador não existe, não modificamos nem armazenamos nada, apenas retornamos a função:

Seção: Especificador de Direção: Caso 1:

```
if(begin == NULL || end == NULL)
  return true;
```

Vamos checar agora se temos o caso "$\{curl\gamma\}$" onde γ é uma expressão numérica. Neste caso, nós terminamos armazenando γ na coordenada y do especificador de direção e deixamos a coordenada x como "Not-a-Number":

Seção: Especificador de Direção: Caso 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;
}
```

Agora devemos checar se temos um caso do tipo "$\{a, b\}$" onde a e b são números. Para isso, tentaremos delimitar os tokens de a e b procurando pela vírgula. Mas somente se a vírgula for encontrada que iremos interpretar isso como o terceiro caso:

Seção: Especificador de Direção: Caso 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;
}
```

E o terceiro caso, onde teremos um elemento $\{a\}$, onde a é uma expressão de par. Para tratar este caso, observe que no caso anterior nós já delimitamos a. Se a vírgula não foi encontrada, de qualquer forma temos seus tokens delimitados. Então podemos interpretá-los como uma expressão de par:

Seção: Especificador de Direção: Caso 4:

```
else{ // Se no caso anterior não achamos a vírgula
  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;
}
```

Com o código que escrevemos até agora estamos preenchendo todos os especificadores de direção explícitos que encontramos à medida que avaliamos a expressão de caminho. Contudo, quando nenhum especificador explícito é colocado, existem algumas regras que nos permitem deduzir valores implícitos para eles.

Primeiro que quando temos uma junção de concatenação (`&`) ,de linha reta (`--`), ou quando temos dois pontos idênticos em sequência, o especificador de direção no segmento atual é irrelevante. No primeiro caso, porque duas extremidades concatenadas terão o mesmo ponto e por isso não há direção alguma entre eles. No segundo caso, o segmento será armazenado diretamente no formato final como uma linha reta sem precisar de especificadores de direção. No terceiro caso, simplesmente temos um ponto redundante sendo escrito mais de uma vez em sequência e não existe nenhum espaço intermediário entre eles. Contudo, devemos colocar um `curl 1` implícito antes e depois dos pontos em tais casos. Basicamente transformamos as expressões abaixo:

```
z0 .. z1 & z2 .. z3
z0 .. z1 -- z2 .. z3
z0 .. z1 .. z1 .. z3
```

no código:

```
z0 .. {curl 1}z1 & z2{curl 1} .. z3
z0 .. {curl 1}z1 -- z2{curl 1} .. z3
z0 .. {curl 1}z1 .. z1{curl 1} .. z3
```

e isso é feito pelo código abaixo:

Seção: Expressão de Caminho: Interpreta Direção se Existir (continuação):

```
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;
  }
  // Se z1 == z2, colocamos pontos de controles explícitos caso eles não existam:
  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;
  }
}
```

Outra dedução implícita de especificadores de direção ocorre quando o ponto anterior ou o próximo ponto estão no formato final com pontos de controle explícitos. A transformação que ocorre é convertendo as expressões abaixo:

```
z0 .. controls u and v .. z1 .. z2 .. z3
z0 ..  z1 .. z2 .. controls u and v ..z3
```

Fazendo elas serem interpretadas como:

```
z0 .. controls u and v .. z1{z1-v} .. z2 ..z3
z0 ..  z1 .. {u-z2}z2 .. controls u and v ..z3
```

Este caso é tratado no momento em que interpretamos os pontos de controle da junção quando eles são explícitos:

Seção: Expressão de Caminho: Leu Pontos de Controle Explícitos:

```
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;
  }
}
```

Especificadores de direção são também implicitamente copiados de um lado de um ponto para

o outro. Isso significa que as expressões abaixo:

```
z0 .. {w1}z1 .. z2 ..z3
z0 .. z1{w1} .. z2 ..z3
```

Seriam interpretadas como:

```
z0 .. {w1}z1{w1} .. z2 ..z3
z0 .. {w1}z1{w1} .. z2 ..z3
```

E isso é feito pelo código:

Seção: Expressão de Caminho: Interpreta Direção se Existir (continuação):

```
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;
}
```

Agora finalmente devemos interpretar a junção em si. A junção pode ter as seguintes formas:

```
z1 & z2
z1 -- z2
z1 .. z2
z1 .. controls u and v .. z2
z1 .. tension a and b .. z2
```

Se nossa junção for um token “&”, então estamos lidando com a concatenação de dois pontos ou (mais provavelmente) sub-caminhos. Neste caso, devemos checar se os dois pontos que estão sendo unidos ocupam a mesma posição (consideramos a mesma posição como sendo uma distância menor que 0,00002). Se não estiverem, um erro deve ser gerado. Se estiverem, juntaremos ambos os caminhos removendo a primeira cópia do ponto onde está sendo feita a junção.

Seção: Expressão de Caminho: Interpreta Junção:

```
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;
  }
  // Isso fará o primeiro ponto de extremidade ser sobrescrito
  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;
}
```

O próximo caso de junção será quando os pontos serão ligados por uma linha reta. Neste caso, o nosso ponto terá seu formato modificado para a forma final e os pontos de controle junto com os pontos de extremidade estarão no mesmo segmento de reta:

Seção: Expressão de Caminho: Interpreta Junção (continuação):

```
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);
}
```

Se a junção for simples, sem qualquer informação, ela será interpretada como equivalente a tendo uma tensão igual a 1. Então as duas expressões abaixo são equivalentes:

```
z1 .. z2
z1 .. tension 1 and 1 .. z2
```

Preencher a tensão neste caso é feito pelo código abaixo:

Seção: Expressão de Caminho: Interpreta Junção (continuação):

```
else if(begin_j == end_j && begin_j -> type == TYPE_JOIN){
  z1_point -> prov.tension1 = 1.0;
  z1_point -> prov.tension2 = 1.0;
}
```

O próximo caso é quando tempos pontos de controle explícitos. Podem haver dois formatos diferentes:

```
z1 .. controls u and v .. z2
z1 .. controls u .. z2
```

O segundo caso é equivalente a ter ambos os pontos de controle com o mesmo valor. O modo como diferenciamos ambos os casos é devido ao token `and`:

Seção: Expressão de Caminho: Interpreta Junção (continuação):

```
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;
  }
```

<Seção a ser Inserida: **Expressão de Caminho: Leu Pontos de Controle Explícitos**>

```
}
```

E finalmente, a junção pode ter o formato de descrição de tensão. Há as seguintes possibilidades:

```
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
```

O código que interpreta a junção dada uma descrição de tensão é:

Seção: Expressão de Caminho: Interpreta Junção (continuação):

```
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;
  }
}
```

Conforme visto na Subseção 7.4.3, o valor de tensão não pode ser menor que 3/4, ou a equação que nos permite deduzir o formato da curva pode não ter solução.

Uma vez que todas as tensões, pontos de controle e direções foram coletadas, existem mais algumas operações que precisamos fazer após percorrer todas as junções. Para estas operações finais, vamos mudar o nosso ponteiro do ponto z_0 para apontar para o primeiro ponto do caminho. Já os ponteiros para os pontos z_1 e z_2, não iremos mudar: depois de passarmos pela última iteração z_1 é o penúltimo ponto e z_2 é o último:

Seção: Expressão de Caminho: Após Ler as Junções:

```
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]);
}
```

Agora após termos iterado sobre todas as junções, é possível que ainda não tenhamos de interpretar tudo o que há na expressão. Isso porquê pode haver um último especificador de direção do lado direito do último ponto. Caso ele exista, ele precisa ser validado. Em um caminho não-cíclico, ele será copiado para o lado esquerdo daquele ponto caso não haja especificador de direção. Em um caminho cíclico, isso também pode acontecer, mas o especificador também terá efeito no ponto em que ele está:

Seção: Expressão de Caminho: Após Ler as Junções:

```
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;
  }
}
```

Agora vamos aplicar algumas regras adicionais para detectar especificadores de direção implícitos, que só podem ser feitas depois que lemos a expressão inteira.

Se estamos com um caminho que não é cíclico, mas o começo ou o fim do caminho não tem especificador de direção, devemos marcar o especificador inicial e/ou final como `curl 1`. Em outras palavras, as duas expressões abaixo devem ser vistas como equivalentes:

```
p = z1 .. z2 .. z3;
p = z1{curl 1} .. z2 .. {curl 1}z3;
```

Além disso, o último ponto não tem nada o sucedendo. O seu formato deve ser ajustado para o formato final, com os pontos de controle sendo iguais ao do próprio ponto:

Seção: Expressão de Caminho: Após Ler as Junções:

```
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;
  }
}
```

Após avaliarmos todas as expressões, podemos garantir que as coordenadas do primeiro e último ponto são as mesmas. Entretanto, pode acontecer que eles estejam em formato diferente. Se o primeiro ponto está em seu formato final e o último não, então o último deve passar a ficar na forma final também, copiando os pontos de controle:

Seção: Expressão de Caminho: Após Ler as Junções:

```
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;
    }
  }
}
```

Se ambos os pontos estiverem no formato provisório, copiamos todos os especificadores de direção que não estiverem vazios de um para o outro. Caso haja algum conflito, como no exemplo abaixo, as informações do primeiro ponto terão prioridade sobre o último:

```
% Exemplo de conflito:
p = (0, 0){1, 2} .. (1, 3) .. cycle{1, 3};
% Assim como no METAFONT, isso é interpretado como:
p = (0, 0){1, 2} .. (1, 3) .. {1, 3}cycle{1, 2};
```

E este é o código que deixa ambos os pontos consistentes:

Seção: Expressão de Caminho: Após Ler as Junções:

```
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;
}
```

Isso finaliza a construção de uma curva dados seus pontos e sub-caminhos lidos. Agora iremos estabelecer como ler os pontos e sub-caminhos.

8.4.2. Expressões Terciárias de Caminhos

A gramática para expressões de caminho terciárias é:

```
<Terciário de Caminho> -> <Terciário de Par> | <Secundário de Caminho>
```

E é só isso. Então, para interpretar uma expressão terciária de caminho, o que faremos é percorrer ela e verificar se encontramos uma expressão terciária de par. Se encontrarmos, devemos interpretar a expressão inteira como um terciário de par. Caso contrário, a interpretaremos como um secundário de caminho. Se interpretarmos tudo como um par, devemos também depois converter o resultado de um par para um caminho com um único ponto.

De qualquer forma, depois de interpretar a expressão e obter um caminho como resultado, nós retornamos o resultado.

A declaração da função é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E a sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. Expressões Secundárias de Caminhos: Transformadores

A gramática para expressões secundárias de caminho é:

```
<Secundário de Caminho> -> <Secundário de Par> | <Primário de Caminho> |
                           <Secundário de Caminho><Transformador>
```

Os transformadores são os mesmos que já foram apresentados na expressão secundária de par, mas incluindo o operador `transformed` introduzido na seção sobre transformações. A primeira coisa que devemos testar é se temos um transformador no fim e se temos uma operação secundária de par (multiplicação e divisão). Depois de percorrer toda a expressão, se houver o tranformador, aplicamos a terceira regra da gramática acima. Se houver uma multiplicação e divisão, mas não um transformador, aplicamos a segunda regra. E caso não haja nada disso, aplicamos a primeira.

Transformar um caminho via rotação, escala e outras modificações envolve aplicar tais modificações sobre cada ponto de extremidade e cada ponto de controle.

A declaração da função que fará isso é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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) // Separador de fração
         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;
```

<Seção a ser Inserida: Transformação de Caminho: Rotação>
<Seção a ser Inserida: Transformação de Caminho: Escala>
<Seção a ser Inserida: Transformação de Caminho: Deslocamento>
<Seção a ser Inserida: Transformação de Caminho: Inclinação>
<Seção a ser Inserida: Transformação de Caminho: X-Escala>
<Seção a ser Inserida: Transformação de Caminho: Y-Escala>
<Seção a ser Inserida: Transformação de Caminho: Z-Escala>
<Seção a ser Inserida: Transformação de Caminho: Transformação Genérica>

```
  }
  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);
}
```

No código acima, quando verificamos se temos um transformador na expressão, sempre armazenamos o último transformador encontrado na variável `transform_op` e o último token antes do transformador em `before_transform`. Isso nos permite depois poder dividir a expressão em partes para poder interpretá-la e obter o caminho sobre o qual devemos aplicar o transformador.

Podemos assumir que toda vez que aplicamos um transformador, o faremos sobre um caminho já normalizado, isto é, sem recursões e com todos os pontos no formato final. Isso ocorre porque a normalização acontece no momento em que a função `eval_path_expression` retorna. E a transformação no código acima sempre ocorrerá sobre o resultado de expressões primárias de caminho. Nas expressões primárias teremos variáveis (que para terem sido armazenadas, foram retornadas por um `eval_path_expression`) ou então sub-expressões (que são interpretadas recursivamente por um `eval_path_expression`). Felizmente, isso limitará a complexidade do código que lida com tais operações.

Se temos uma rotação, para interpretar um transformador de rotação depois de termos obtido o caminho a ser rotacionado na variável `result`, podemos então aplicar o código abaixo:

Seção: Transformação de Caminho: Rotação:

```
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; // Converter de graus para radianos
  sin_theta = sin(theta);
  cos_theta = cos(theta);
  path_rotate(result, sin_theta, cos_theta);
  return true;
}
```

E a função que efetivamente faz a rotação:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_rotate(struct path_variable *p, double sin_theta,
                 double cos_theta);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  }
}
```

Agora vamos ao operador que muda a escala de um caminho. Ele deve interpretar um valor numérico e depois multiplicar cada ponto do caminho por tal valor numérico:

Seção: Transformação de Caminho: Escala:

```
else 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;
}
```

A função que itera sobre os pontos do caminho fazendo a mudança de escala:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_xyscale(struct path_variable *p, float x, float y);
```

Note que a função é mais ampla e pode ser usada para aumentar de maneira não-proporcional o tamanho de uma curva no eixo x e y, apesar de aqui só a estarmis usando para esticá-la a mesma quantidade na horizontal e vertical. A sua implementação é:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  }
}
```

A próxima transformação é uma simples translação ou deslocamento. Recebemos um par e este par determina o quanto cada ponto deve ser deslocado no eixo s e y:

Seção: Transformação de Caminho: Deslocamento:

```
else 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;
}
```

E a função recursiva que faz o deslocamento:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_shift(struct path_variable *p, float x, float y);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  }
}
```

Vamos agora à inclinação, o transformador que empurra os pontos mais à direita se estiverem acima do 0 no eixo y e mais à esquerda se estiverem mais abaixo da origem no eixo y:

Seção: Transformação de Caminho: Inclinação:

```
else 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;
}
```

E a função que faz a inclinação em si:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_slant(struct path_variable *p, float s);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  }
}
```

A próxima transformação muda o tamanho horizontal do caminho, mas preserva o tamanho vertical. Devemos ler um valor numérico e ele determina o quanto o caminho deve ser esticado horizontalmente:

Seção: Transformação de Caminho: X-Escala:

```
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;
}
```

Aqui estamos usando uma função já definida, então não precisamos defini-la novamente. Além de transformar esticando horizontalmente, vamos agora esticar verticalmente:

Seção: Transformação de Caminho: Y-Escala:

```
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;
}
```

A penúltima transformação é a mudança de escala Z que lê um par, o interpreta como número complexo, e também interpreta cada ponto do caminho como número complexo fazendo a multiplicação:

$$(a+bi)(c+di) = ac + (ad)i + (cb)i + (bd)i^2 = (ac-bd) + (cb+ad)i$$

Interpretamos e realizamos a transformação da forma abaixo, lembrando também que esta é a última transformação possível:

Seção: Transformação de Caminho: Z-Escala:

```
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;
}
```

E a função que fará isso:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_zscale(struct path_variable *p, float x, float y);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
```

```
  }
}
```

E o último tipo de transformador é a transformação genérica, onde teremos um transformador que irá indicar como iremos mudar todos os pontos do caminho. Tratamos este caso da seguinte forma:

Seção: Transformação de Caminho: Transformação Genérica:

```
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;
}
```

Para isso usamos a seguinte função:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_transform(struct path_variable *p, float *M);
```

A função é definida como:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. Expressões Primárias: Variáveis, Reversos e Subcaminhos

A gramática para expressões primárias de caminho é:

```
<Primário de Caminho> -> <Primário de Par> | <Variável de Caminho> |
                         ( <Expressão de Caminho> ) |
                         reverse <Primário de Caminho> |
                         subpath <Expressão de Par> of <Primário de Caminho> |
                         ...
```

As regras estão incompletas porque um operador primário adicional relacionado à variáveis de caneta será definido na Subseção 8.5.4.

Por hora devemos registrar tokens novos: `reverse`, `subpath` e `of`:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_REVERSE,        // O token simbólico 'reverse'
TYPE_SUBPATH,        // O token simbólico 'subpath'
TYPE_OF,             // O token simbólico 'of'
```

E para cada um deles adicionemos sua string na lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"reverse", "subpath", "of",
```

O que as regras gramaticais dizem é que no fim de toda expressão de caminho, terminaremos encontrando no fim uma variável de caminho, um parênteses, ou algum destes operadores primários novos. Se não, em todos os outros casos, temos uma expressão primária de parque nos dará um par. Devemos então testar se estamos nestes casos, e se não estivermos, basta interpretar como par.

A função que irá interpretar expressões primárias é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
                <Seção a ser Inserida: Caminho Primário: Reverso>
  }
  else if(begin -> type == TYPE_SUBPATH){
              <Seção a ser Inserida: Caminho Primário: Subcaminho>
  }
  else if(begin == end && begin -> type == TYPE_SYMBOLIC){
                <Seção a ser Inserida: Caminho Primário: Variável>
  }
  else if(begin -> type == TYPE_OPEN_PARENTHESIS &&
          end -> type == TYPE_CLOSE_PARENTHESIS){
              <Seção a ser Inserida: Caminho Primário: Parênteses>
  }
         <Seção a ser Inserida: Caminho Primário: Outras Expressões>
  { // Se ainda não retornou, é um par primário
              <Seção a ser Inserida: Caminho Primário: Par Primário>
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line), TYPE_T_PATH);
  return false;
}
```

Vamos ao primeiro caso: devemos calcular o reverso de um caminho. Para isso, devemos inverter a ordem na qual aparecem os pontos de extremidade e mover os pontos de controle para suas novas posições. Toda vez que esta operação é feita, será sempre sobre uma variável de caminho já normalizada, sem recursão e com todos os pontos no formato final.

Seção: Caminho Primário: Reverso:

```
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;
```

E a função que realiza a reversão:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool reverse_path(struct metafont *mf, struct context *cx,
                  struct path_variable *dst,
                  struct path_variable *origin);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

O próximo passo é calcular um subcaminho. O subcaminho gera um novo caminho que é parte de um caminho maior. Por exemplo:

`subpath (0, 2) of p1 -- p2 -- p3;`

O código acima avalia para o novo caminho `p1 -- p2`, assumindo que `p1` e `p2` são um único ponto cada um.

No METAFONT original, é possível calcular sub-caminhos entre pontos em posições não-inteiras. Por exemplo, calcular o sub-caminho do ponto 0,5 até 1,8. O METAFONT obteria um ponto intermediário entre os pontos de controle 0 e 1 e entre 1 e 2 conforme pedido. Novos pontos de extremidade e de controle adequados seriam gerados. Aqui por hora iremos suportar só uma versão mais simples de sub-caminhos. Somente sub-caminhos inteiros serão suportados.

No caso de caminhos não-cíclicos, se tentarmos especificar um sub-caminho com uma posição menos que zero, tal índice será tratado como zero. E se tentarmos especificar uma posição maior que seu índice máximo, será tratada como igual ao seu índice máximo.

No caso de caminhos cíclicos, índices negativos são contados caminhando no ciclo na direção oposta do caminho. E também não há um índice máximo permitido. Desta forma, você pode criar um subcaminho de um caminho cíclico que é maior que o caminho original. Mas a natureza cíclica do caminho será sempre perdida após a operação.

Seção: Caminho Primário: Subcaminho:

```
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;
  // Ignora último ponto igual primeiro:
  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));
  // Ajusta últimos pontos de controle se perdemos o caráter cíclico:
  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;
}
```

O próximo caso a tratar é quando estamos diante de uma variável. Neste caso, devemos determinar seu tipo. Pode ser um par ou um caminho. Em seguida, alocamos o tamanho certo para nosso resultado e copiamos o conteúdo da variável para ele:

Seção: Caminho Primário: Variável:

```
{
  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,
                                    var -> type, TYPE_T_PATH);
    return false;
  }
}
```

O próximo caso a ser tratado é quando a expressão primária de caminho começa e termina com parênteses. E não estamos diante de um par. Neste caso, devemos avaliar a expressão entre parênteses como uma nova expressão de caminho. Mas para isso temos antes que checar que não há uma vírgula dentro dela, caso em que estamos diante de um par:

Seção: Caminho Primário: Parênteses:

```
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);
}
```

E finalmente, o último caso de expressãso primária é quando temos uma expressão primária de par. Neste caso, alocamos o único ponto de nosso caminho, avaliamos a expressão de par e passamos o resultado para o ponto alocado:

Seção: Caminho Primário: Par Primário:

```
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. Caminhos em Expressões Numéricas

Existe a expressão numérica primária com o operador `length` que retorna o número de pontos de extremidade de um caminho menos um. A sua sintaxe é:

```
<Primário Numérico> -> length <Caminho Primário>
```

Não precisamos aqui de nenhum tipo de token novo. A expressão `length` já era usada para obter o módulo de números e de pares. Para o caso deste operador obter um caminho, sua implementação é ainda mais simples:

Seção: Avalia 'length' (continuação):

```
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. Caminhos em Expressões de Pares

Dado um caminho, nós podemos extrair pares dele. O par pode ser um dos pontos de extremidade ou algum dos pontos de controle. A sintaxe para isso é:

```
<Primário de Par> -> point <Expressão Numérica> of <Primário de Caminho> |
                     precontrol <Expressão Numérica> of <Primário de Caminho> |
                     postcontrol <Expressão Numérica> of <Primário de Caminho>
```

Isso requer adicionarmos os seguintes novos tipos de tokens:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_POINT,             // O token simbólico 'point'
TYPE_PRECONTROL,        // O token simbólico 'precontrol'
TYPE_POSTCONTROL,       // O token simbólico 'postcontrol'
```

E os adicionamos à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"point", "precontrol", "postcontrol",
```

Todos estes operadores requerem que nós encontremos determinado ponto na posição n de um caminho. Para isso, será útil uma função auxiliar que irá receber uma variável de caminho e um número e irá retornar um ponteiro para um dos pontos desta variável. Essa função irá primeiro verificar se temos um caminho onde o tamanho total é igual ao tamanho (`length == total_length`). Se for o caso, assumiremos estar diante de um caminho sem subcaminhos recursivos e retornar o ponto correto será fácil e rápido. Caso contrário, vamos chamar uma função recursiva auxiliaar para nos responder qual é o ponto correto:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
struct path_points *get_point(struct path_variable *v, int n);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
  }
}
```

Essa função auxiliar recursiva para obter o ponto correto no caso de caminhos com subcaminhos recursivos irá percorrer os pontos do caminho na ordem, contando quantos são encontrados. E irá retornar o ponto quando chegar ao correto. Para isso ela precisa saber qual a variável de caminho, qual o índice n do ponto e por fim, quantos pontos já foram percorridos:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
struct path_points *_get_point(struct path_variable *v, int n, int *count);
```

A definição desta função será:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

No caso do primeiro operador, `point`, ele nos retorna o ponto de extremidade indicado. Se o caminho não for cíclico, os pontos de índice menor que zero serão iguais ao de índice zero (o primeiro) e os pontos de índice maior ou igual ao tamanho serão iguais ao último ponto. Se o caminho for cíclico, o índice é contado seguindo o ciclo. No METAFONT original, índices não-positivos eram permitidos, mas aqui eles serão convertidos para inteiros.

O operador `postcontrol` é semelhante, mas ele obtém o primeiro ponto de controle imediatamente após o ponto cujo índice é indicado. E o `precontrol` obtém o ponto de controle imediatamente antes do ponto indicado.

Seção: Primário de Par: Outras Regras a Definir Depois:

```
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){ // Código termina após 'of'
    RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
    return false;
  }
  if(of == end){ // Expressão termina após '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. Atribuições e Expressões de Caneta

Para realizar a atribuição de canetas a variáveis do tipo certo, usamos o código abaixo:

Seção: Atribuição de Variável de Caneta:

```
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));
```

```
}
```

A declaração da função que faz a atribuição:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool assign_pen_variable(struct metafont *mf, struct context *cx,
                         struct pen_variable *target,
                         struct pen_variable *source);
```

E a sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  // Se estamos atribuindo para 'currentpen', devemos retriangular a caneta.
  // Conforme descreveremos na Seção 11.2.
  if(target == cx -> currentpen)
    triangulate_pen(mf, cx, target, target -> gl_matrix);
  return true;
}
```

8.5.1. Expressão Terciária de Caneta

Por enquanto não existe nenhum operador terciário de caneta:

```
<Expressão de Caneta> -> <Terciário de Caneta>
<Terciário de Caneta> -> <Secundário de Caneta>
```

Então a função que avalia expressão terciária de canetas apenas repassa a expressão para a

expressão secundária (mas podemos adicionar algumas linhas para detectar erros comuns):

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. Transformadores de Caneta

A sintaxe para expressões secundárias de caneta é:

```
<Secundário de Caneta> -> <Primário de Caneta> |
                          <Secundário de Caneta> <Transformador>
<Transformador> -> rotated <Primário Numérico> |
                   scaled <Primário Numérico> |
                   shifted <Par Primário> |
                   slanted <Numérico Primário> |
                   xscaled <Numérico Primário> |
                   yscaled <Numérico Primário> |
                   zscaled <Par Primário> |
                   transformed <Primário de Transformação>
```

Transformadores não são coisas novas, eles já eram usados em expressões de pares, transformações e de caminho. Mas aplicá-los sobre canetas é ligeiramente diferente, pois ao invés de modificar cada um de seus pontos, nós modificamos apenas sua matriz de transformação OpenGL.

A declaração da função que avaliará expressões secundárias de caneta é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Sua implementação consiste em percorrer os tokens da expressão até achar o último transformador, se houver. Se não houver, é só repassar a avaliação para a função que avalia expressões primárias. Se houver, então avaliamos como expressão secundária o que vem antes e então realizamos a transformação adequada:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
            <Seção a ser Inserida: Secundário de Caneta: Rotação>
  }
  else if(last_transformer -> type == TYPE_SCALED){
             <Seção a ser Inserida: Secundário de Caneta: Escala>
  }
  else if(last_transformer -> type == TYPE_SHIFTED){
          <Seção a ser Inserida: Secundário de Caneta: Deslocamento>
  }
  else if(last_transformer -> type == TYPE_SLANTED){
           <Seção a ser Inserida: Secundário de Caneta: Inclinação>
  }
  else if(last_transformer -> type == TYPE_XSCALED){
            <Seção a ser Inserida: Secundário de Caneta: X-Escala>
  }
  else if(last_transformer -> type == TYPE_YSCALED){
            <Seção a ser Inserida: Secundário de Caneta: Y-Escala>
  }
  else if(last_transformer -> type == TYPE_ZSCALED){
            <Seção a ser Inserida: Secundário de Caneta: Z-Escala>
  }
  else if(last_transformer -> type == TYPE_TRANSFORMED){
     <Seção a ser Inserida: Secundário de Caneta: Transformação Genérica>
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
```

```
                                        TYPE_T_PEN);
    return false;
  }
}
```

Primeiro cuidaremos da rotação em um ângulo θ. Primeiro fazemos a conversão entre o ângulo medido em graus que o WeaveFONT usa e a medida em radianos esperada pela biblioteca padrão C. Rotacionar significa aplicar a transformação de rotação sobre a matriz de transformação da caneta:

Seção: Secundário de Caneta: Rotação:

```
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;
```

Para fazer a operação de escala também aplicamos tal transformação sobre a matriz da caneta:

Seção: Secundário de Caneta: Escala:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SCALE(pen -> gl_matrix, a.value);
// Canetas curvas devem ser retrianguladas se aumentam de tamanho:
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;
```

A transformação de deslocamento que desloca a caneta na horizontal e vertical é tratada abaixo:

Seção: Secundário de Caneta: Deslocamento:

```
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;
```

O próximo transformador é a inclinação.

Seção: Secundário de Caneta: Inclinação:

```
struct numeric_variable a;
if(!eval_numeric_primary(mf, cx, last_transformer -> next, end, &a))
  return false;
TRANSFORM_SLANT(pen -> gl_matrix, a.value);
// Inclinar canetas curvas não-circulares requer retriangulação:
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;
```

A transformação de escala no eixo x é igual à transformação de escala, mas a caneta é esticada ou comprimida somente no eixo horizontal:

Seção: Secundário de Caneta: X-Escala:

```
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);
// Canetas curvas devem ser retrianguladas se aumentam de tamanho:
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;
```

Da mesma forma, mudar a escala somente no eixo y é feito no código abaixo:

Seção: Secundário de Caneta: Y-Escala:

```
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);
// Canetas curvas devem ser retrianguladas se aumentam de tamanho:
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;
```

A mudança de escala z înterpreta dois pares como números complexos e os multiplica:

$$(x + yi)(a + bi) = ax + (bx)i + (ay)i + (by)i^2 = (ax - by) + (bx + ay)i$$

Podemos então realizar a operação acima usando o código abaixo:

Seção: Secundário de Caneta: Z-Escala:

```
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);
// Canetas curvas devem ser retrianguladas neste caso:
if(pen -> gl_vbo != 0 && !(pen -> flags & FLAG_STRAIGHT)){
  glDeleteBuffers(1, &(pen -> gl_vbo));
  pen -> gl_vbo = 0;
  pen -> indices = 0;
}
return true;
```

Por fim, a última transformação é a genérica que envolve aplicar a transformação linear que está armazenada em um transformador:

Seção: Secundário de Caneta: Transformação Genérica:

```
struct transform_variable t;
if(!eval_transform_primary(mf, cx, last_transformer -> next, end, &t))
  return false;
MATRIX_MULTIPLICATION(pen -> gl_matrix, t.value);
// Canetas curvas devem ser retrianguladas neste caso:
```

```
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. Expressões Primárias: Caneta Nula, Circular, Arbitrária e Variáveis

A gramática das expressões primárias de caminho é definida como:

```
<Primário de Caneta> -> <Variável de Caneta> |
                        nullpen | ( <Expressão de Caneta> ) |
                        pencircle | pensemicircle |
                        makepen <Primário de Caminho>
```

Isso requer registrar quatro novas palavras reservadas:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_NULLPEN,        // O token simbólico 'nullpen'
TYPE_PENCIRCLE,      // O token simbólico 'pencircle'
TYPE_PENSEMICIRCLE,  // O token simbólico 'pensemicircle'
TYPE_MAKEPEN,        // O token simbólico 'makepen'
```

E adicionamos uma string com o nome delas na lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"nullpen", "pencircle", "pensemicircle", "makepen",
```

A função que irá interpretar expressões primárias de caneta é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
                <Seção a ser Inserida: Caneta Primária: Variável>
    }
    else if(begin -> type == TYPE_NULLPEN){
              <Seção a ser Inserida: Caneta Primária: Caneta Nula>
    }
    else if(begin -> type == TYPE_PENCIRCLE){
            <Seção a ser Inserida: Caneta Primária: Caneta Circular>
    }
    else if(begin -> type == TYPE_PENSEMICIRCLE){
          <Seção a ser Inserida: Caneta Primária: Caneta Semicircular>
    }
```

```
  }
  else{
    if(begin -> type == TYPE_OPEN_PARENTHESIS &&
      end -> type == TYPE_CLOSE_PARENTHESIS){
                <Seção a ser Inserida: Caneta Primária: Parênteses>
    }
    else if(begin -> type == TYPE_MAKEPEN){
          <Seção a ser Inserida: Caneta Primária: Formato Personalizado>
    }
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PEN);
  return false;
}
```

Se a expressão avaliar para uma variável, devemos copiar o conteúdo da variável. Se avaliar para uma variável que aponta para outra (o caso do `currentpen`), devemos ao invés disso copiar o conteúdo da variável referenciada. Mas depois disso devemos multiplicar a matriz da variável que aponta pela matriz obtida copiada.

Seção: Caneta Primária: Variável:

```
struct symbolic_token *v = (struct symbolic_token *) begin;
struct pen_variable *content = v -> var, *to_copy = v -> var;
if(content == NULL){
  if(!strcmp(v -> value, "currentpen"))
    content = cx -> currentpen;
  else{
    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,
                                  ((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;
```

Se a expressão avaliar para uma caneta nula, devemos apenas gerar uma nova caneta com a flag de `FLAG_NULL` ativa. Esta caneta nunca será triangulada e nunca produzirá qualquer desenho:

Seção: Caneta Primária: Caneta Nula:

```
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;
```

Se a expressão avaliar para uma caneta circular, geraremos uma nova caneta com as flags de ser circular e convexa. Também inicializamos a matriz de transformação como sendo a matriz identidade:

Seção: Caneta Primária: Caneta Circular:

```
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;
```

Se a caneta for semicircular, o código é bastante semelhante:

Seção: Caneta Primária: Caneta Semicircular:

```
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;
```

Avaliar parênteses envolve avaliar como expressão de caneta os tokens do que vem após o abrir parênteses até o que vem imediatamente antes do fechar parênteses:

Seção: Caneta Primária: Parênteses:

```
struct generic_token *t = begin -> next;
DECLARE_NESTING_CONTROL();
if(begin -> next == end){ // Parênteses vazio: '()':
  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);
```

O último caso de expressão primária de caneta é quando o usuário escolhe um formato personalizado. Neste caso devemos avaliar a expressão de caminho indicada e com ela criamos a nova caneta com tal formato:

Seção: Caneta Primária: Formato Personalizado:

```
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); // Função a ser implememententada
result -> referenced = NULL;
result -> gl_vbo = 0;
result -> indices = 0;
INITIALIZE_IDENTITY_MATRIX(result -> gl_matrix);
return true;
```

O que falta implementar para que o código de criação de canetas personalizadas funcione é a função que percorra os pontos de uma variável de caminho e retorne quais flags teria uma caneta com aquele formato.

Seção: Declaração de Função Local (metafont.c) (continuação):

```
int read_flags(struct path_variable *path);
```

Já sabemos que ela nunca seria circular, pois círculos perfeitos não podem ser expressos como curvas de Beziér. Iremos então verificar se ela é reta ou convexa percorrendo os seus pontos:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
      // Ela não é reta se os pontos de extremidade e de controle não estão
      // na mesma reta. Eles estão na mesma reta se a área de seu triângulo
      // é igual a zero (ou suficientemente próxima):
      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){
      // Ela é convexa somente se o componente z do produto vetorial dos
      // vetores formados pelos pontos de extremidade e controle na ordem
      // de desenho são todos ou não-positivos ou não-negativos.
      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. Canetas em Expressões de Caminho

Expressões primárias de caneta podem aparecer também quando avaliamos uma expressão primária de caminho. Isso permite que possamos extrair um caminho à partir do formato de uma caneta. A gramática de tal operação é:

```
<Primário de Caminho> -> makepath <Primário de Caneta>
```

Temos que registrar um novo token para o “makepath”:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_MAKEPATH,       // O token simbólico 'makepath'
```

E adicionamos a string com seu nome à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"makepath",
```

O operador “makepath” deve avaliar tudo o que vem depois dele como sendo uma expressão primária da caneta. Em seguida, o resultado da avaliação é um caminho com o formato da caneta. Contudo, devemos lembrar que nem sempre armazenamos o formato de uma caneta explicitamente. Caso tenhamos uma `nullpen`, um `circlepen` ou se estivermos diante de um quadrado, então precisamos gerar um caminho novo, já que não temos caminho nenhum:

Seção: Caminho Primário: Outras Expressões:

```
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 único (0, 0)
                  <Seção a ser Inserida: 'makepath': 'nullpen'>
  }
  else if(tmp.flags & FLAG_CIRCULAR){ // circlepen: Cria aproximação de círculo
                  <Seção a ser Inserida: 'makepath': 'pencircle'>
  }
  else if(tmp.flags & FLAG_SEMICIRCULAR){
                <Seção a ser Inserida: 'makepath': 'pensemicircle'>
  }
  else if(tmp.flags & FLAG_SQUARE){ // Caneta quadrada
                  <Seção a ser Inserida: 'makepath': 'pensquare'>
  }
  else{ // Caneta com formato personalizado
            <Seção a ser Inserida: 'makepath': Formato Personalizado>
  }
       <Seção a ser Inserida: 'makepath': Realiza Transformação Linear>
  return true;
}
```

Assim como na linguagem METAFONT original, o `nullpen` significa um único ponto na posição $(0, 0)$ que nunca é desenhado:

Seção: '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;
```

Caso estejamos diante de um caminho circular, usaremos como representação do caminho os mesmos valores que o METAFONT original usa para representar o caminho na macro `fullcircle`. O resultado será algo próximo a um círculo.

Seção: '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;
```

Um semicírculo é a mesma coisa, mas usamos só metade dos pontos acima:

Seção: '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;
```

Se acaneta for quadrada, geramos seu formato verdadeiro preenchendo manualmente seus pontos de extremidade e de controle:

Seção: '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;
```

Por fim, se estivermos diante de uma caneta com formato personalizado, então nós já temos uma representação de seu formato na forma de um caminho e nós só temos que copiá-la:

Seção: '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);
}
```

E finalmente, uma vez que tenhamos o nosso caminho, devemos aplicar todas as transformações de rotação, translação e demais transformações lineares que estiverem registradas na matriz de transformação OpenGL:

Seção: 'makepath': Realiza Transformação Linear:

```
{
  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. Atribuições e Expressões de Imagens

Para realizar a atribuição de imagens a variáveis do tipo certo, usamos o código abaixo. O que ele faz é avaliar uma expressão de imagem e coletar o resultado. Em seguida, ele precisa fazer com que todas as variáveis da atribuição recebam uma cópia da imagem avaliada. Se existe somente uma variável a ser atribuída, nós apenas copiamos o identificador textura já obtida e desenhada na avaliação. Se houver mais de uma variável, a primeira delas recebe uma cópia direta do identificador da imagem já obtida. Para as demais, precisaremos usar comandos OpenGL para copiar a textura avaliada para o destino.

Seção: Atribuição de Variável de Imagem:

```
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;
      // Se atribuindo para 'currentpicture', precisamos de código adicional
      if(pic == cx -> currentpicture){
        // O código abaixo será definido na Subseção 14.1:
                  <Seção a ser Inserida: Gera nova '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);
  }
}
```

A função `assign_picture_variable` acima precisa gerar uma nova textura na variável de destino (apagando a textura já existente se for o caso) e copiar o conteúdo da textura de origem no destino. Isso significa que nós precisaremos renderizar o conteúdo de uma textura na outra.

Para renderizar algo, primeiro precisaremos de vértices. Como tudo que vamos renderizar são sempre imagens e texturas retangulares, os únicos vértices que precisaremos renderizar são:

Seção: Variáveis Locais (metafont.c) (continuação):

```
static const float square[20] = {
  -1.0, -1.0, //Primeiro vértice
  0.0, 0.0, // Coordenada da textura
  1.0, -1.0, // Segundo vértice
  1.0, 0.0, // Textura
  1.0, 1.0, // Terceiro vértice
  1.0, 1.0, // Textura
  -1.0, 1.0, // Quarto vértice
  0.0, 1.0}; // Textura
static GLuint vbo; // OpenGL Vertex Buffer Object
```

Associamos acima a cada vértice uma coordenada de textura. Desta forma conseguiremos enviar para a placa de vídeo ambas as informações de uma só vez. Note que o que definimos acima é um quadrado de lado 2 centralizado na origem. Este é o tamanho padrão para que ele seja renderizado em todo o espaço disponível nas convenções OpenGL. Nós também definimos o quadrado definindo seus vértices no sentido anti-horário, o que faz com que por convenção ele seja renderizado de frente para a câmera. Isso é necessário para que o quadrado sempre seja renderizado, mesmo que otimizações estejam ativas fazendo com que o lado de trás dos polígonos não sejam desenhados.

Na inicialização do Weaver Metafont nós enviamos estes vértices para a placa de vídeo:

Seção: Inicialização WeaveFont:

```
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
// Enviando os vértices para a placa de vídeo:
glBufferData(GL_ARRAY_BUFFER, sizeof(square), square, GL_STATIC_DRAW);
```

Na finalização nós removemos os vértices da placa de vídeo:

Seção: Finalização WeaveFont:

```
glDeleteBuffers(1, &vbo);
```

Além de vértices, precisamos de um shader de vértice que nos indique como renderizar tais

vértices. O programa deve ser versátil o bastante para receber como argumento uma matriz com transformações lineares a serem aplicadas a cada vértice. Os atributos do programa são a coordenada do vértice e da textura. O valor uniforme recebido é a matriz para transformação linear. E o valor de retorno a ser passado para o shader de fragmento é a posição final da coordenada depois da transformação, além da coordenada de textura.

Seção: Variáveis Locais (metafont.c) (continuação):

```
static const char vertex_shader[] =
  "#version 100\n"
  "attribute vec4 vertex_data;\n"
  "uniform mat3 model_view_matrix;\n"
  "varying highp 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";
```

Já o código do shader de fragmento recebe a saída do shader de vértice acima além de uma textura:

Seção: Variáveis Locais (metafont.c) (continuação):

```
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"
  "  gl_FragColor = texture2D(texture1, texture_coordinate);"
  "}\n";
static GLuint program; // Armazenará o programa após compilar os shaders acima
GLint uniform_matrix; // Armazenará a posição da matriz de modelo visualização
acima
GLint uniform_texture; // A posição da variável de textura no código acima
```

Estes shaders simples precisam ser compilados na inicialização. Vamos criar uma função auxiliar local para nos ajudar na compilação:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
GLuint compile_shader_program(const char *vertex_shader_source,
                              const char *fragment_shader_source);
```

Esta função irá retornar o identificador do programa compilado à partir dos códigos-fontes dos shaders passados como argumento. Se algo der errado, irá retornar zero.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
GLuint compile_shader_program(const char *vertex_shader_source,
                              const char *fragment_shader_source){
  GLuint vertex, fragment, prog;
  GLint status = GL_TRUE;
  // Criando shader de vértice e de fragmento
  vertex = glCreateShader(GL_VERTEX_SHADER);
  fragment = glCreateShader(GL_FRAGMENT_SHADER);
  // Passando o código-fonte a cada um deles
  glShaderSource(vertex, 1, &vertex_shader_source, NULL);
  glShaderSource(fragment, 1, &fragment_shader_source, NULL);
  // Compilando shader de vértice:
```

```
  glCompileShader(vertex);
  glGetShaderiv(vertex, GL_COMPILE_STATUS, &status);
  if(status == GL_FALSE){
    fprintf(stderr,
            "ERROR: Weaver Metafont vertex shader compilation failed!\n");
    return 0;
  }
  // Compilando shader de fragmento:
  glCompileShader(fragment);
  glGetShaderiv(fragment, GL_COMPILE_STATUS, &status);
  if(status == GL_FALSE){
    fprintf(stderr,
            "ERROR: Weaver Metafont fragment shader compilation failed!\n");
    return 0;
  }
  // Criando programa:
  prog = glCreateProgram();
  // Ligando o programa:
  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 0;
  }
  // Finalização:
  glDeleteShader(vertex);
  glDeleteShader(fragment);
  return prog;
}
```

E podemos usar esta função para inicializar nosso programa de shader padrão:

Seção: Inicialização WeaveFont (continuação):

```
{
  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");
}
```

Na finalização iremos destruir o programa compilado acima:

Seção: Finalização WeaveFont (continuação):

```
glDeleteProgram(program);
```

Para a parte de renderizar o conteúdo de uma textura para dentro de outra, devemos criar um framebuffer com uma nova textura associada a ele. A função abaixo cria tanto um novo framebuffer como uma nova textura, deixando-os ligados e ativos. Tudo o que for renderizado em seguida será renderizado na nova textura, e não na tela.

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool get_new_framebuffer(GLuint *new_framebuffer, GLuint *new_texture,
```

```
                          int width, int height);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Outro código que vamos usar muito e por isso é importante isolar em uma função é para renderizar uma imagem usando o framebuffer atual e o programa de shader padrão que fizemos:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
    // Limpando o destino para branco transparente antes de renderizar:
    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");
    }
  }
}
```

Agora vamos declarar a função que atribui o conteúdo de uma variável de imagem à outra, gerando uma nova textura e copiando o conteúdo das texturas:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool assign_picture_variable(struct metafont *mf, struct context *cx,
                             struct picture_variable *target,
                             struct picture_variable *source);
```

E a sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
```

```
  // Finalizando
  glBindTexture(GL_TEXTURE_2D, 0);
  glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
  glDeleteFramebuffers(1, &temporary_framebuffer);
  if(target == cx -> currentpicture){
    // O código abaixo será definido na Subseção 14.1:
                <Seção a ser Inserida: Gera nova 'currentpicture'>
  }
  return true;
}
```

O único erro que pode ocorrer acima é um erro no OpenGL, caso não seja possível criar um framebuffer. Isso é algo que definitivamente não deve acontecer.

8.6.1. Expressões Terciárias de Imagem: Soma e Subtração

A operação terciária de imagem é a soma e a subtração:

```
<Expressão de Imagem> -> <Terciário de Imagem>
<Terciário de Imagem> -> <Terciário de Imagem><Mais ou Menos><Secundário de
Imagem>
<Mais ou Menos> -> + | -
```

Imagens então podem ser somadas ou subtraídas. O resultado de $p_1 + p_2$ representa uma nova imagem composta por todos os pixels da primeira imagem mais os pixels da segunda imagem. Já $p_1 - p_2$ são os pixels da primeira menos os pixels da segunda. A imagem resultante sempre terá a largura igual ao maior valor em ambas as imagens e terá a altura igual ao maior valor em ambas as imagens. A soma dos seus pixels sempre ocorrerá com ambas as imagens centralizadas uma na outra.

A função que avalia expressões terciárias de imagem:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

A implementação da função consiste em identificar o operador + ou o operador − que estiver mais ao fim da expressão. Se ele não existe, então devemos avaliar tudo como uma expressão secundária. Se ele existe, o lado à esquerda dele será avaliado como expressão terciária e o lado à direita como expressão secundária. E só então fazemos a soma ou subtração.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
      <Seção a ser Inserida: Imagem: Avalia Expressão Secundária em '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;
      <Seção a ser Inserida: Imagem: Avalia Expressão Secundária em 'sec'>
        <Seção a ser Inserida: Expressão de Imagem: Soma ou Subtração>
    if(a.texture != 0)
      glDeleteTextures(1, &(a.texture));
    if(b.texture != 0)
      glDeleteTextures(1, &(b.texture));
    return true;
  }
}
```

Para somar ou subtrair as imagens, devemos primeiro obter qual deve ser o tamanho da imagem de destino. Ela terá a maior altura e maior largura dentre as imagens sendo somadas. Em seguida, a criamos e a inicializamos como sendo uma imagem transparente. Tanto na soma como na subtração, desenhamos o primeiro operando no destino. Em seguida, dependendo se estamos realizando soma ou subtração, usamos o OpenGL para ajustar como as cores serão misturadas. No caso da soma, ajustamos para que seja feita uma mistura de cores de modo que a nova cor (y_R, y_G, y_B, y_A) seja multiplicada pelo seu valor alfa e somada à cor de destino (x_R, x_G, x_B, x_A) multiplicada pelo complemento do alfa da nova cor:

$$(1 - y_A)(x_R, x_G, x_B, x_A) + y_A(y_R, y_G, y_B, y_A)$$

O caso da subtração é mais complexo e para entendê-la, devemos entender as regras de como representamos nossas texturas com imagens. Por padrão, assumimos que uma imagem inicializada vazia com um dado tamanho é toda preenchida por uma cor branca totalmente transparente (representada pelo vetor RGBA $(1, 1, 1, 0)$). Já quando escrevemos algo em uma imagem usando canetas ou outro método, usamos uma tinta preta totalmente opaca (representada pelo vetor RGBA $(0, 0, 0, 1)$). Para nós a cor branca é o zero, a ausência de tinta e a cor preta é a presença de tinta. O branco é o elemento neutro da subtração, remover o branco não deve mudar nada. Já remover o preto remove toda a tinta. Isso é o oposto da representação do branco como 1 e o preto como 0 usada no OpenGL. Por causa disso, a equação de quando subtraímos uma cor existente (x_R, x_G, x_B, x_A) por uma nova cor (y_R, y_G, y_B, y_A) é:

$$(max(x_R, 1 - y_R), max(x_G, 1 - y_G), max(x_B, 1 - y_B), x_A - y_A)$$

Infelizmente não é possível expressar a equação acima usando apenas o recurso de ajuste da equação de combinação de cores. O OpenGL ES 3.0 e OpenGL 4 suporta o uso da função max, mas quando a usa, ele ignora os fatores configurados. O que significa que é possível calcular

$max(x_R, y_R)$, mas não $max(x_R, 1 - y_R)$. Essa inversão precisa ser feita então a nível de shader. O shader de fragmento que inverte uma cor deixando o alfa intacto é:

Seção: Variáveis Locais (metafont.c) (continuação):

```
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; // O programa acima compilado
static GLint uniform_inv_texture; // A posição da textura acima
static GLint uniform_inv_matrix; // A posição da matriz do programa acima
```

Um novo programa de shader deve ser então compilado na inicialização para podermos inverter os valores RGB de imagens:

Seção: Inicialização WeaveFont (continuação):

```
{
  inv_program = compile_shader_program(vertex_shader, fragment_shader_inverse);
  uniform_inv_matrix = glGetUniformLocation(inv_program, "model_view_matrix");
  uniform_inv_texture = glGetUniformLocation(inv_program, "texture1");
}
```

E na finalização nós destruimos este programa:

Seção: Finalização WeaveFont (continuação):

```
glDeleteProgram(inv_program);
```

E usando esse novo programa shader para ajudar a inverter valores RGB, podemos enfim computar a equação de mistura de cores no caso da subtração:

Seção: Expressão de Imagem: Soma ou Subtração:

```
// Alocando e declarando dados, gerando imagem vazia inicial
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);
// Renderizando imagem '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);
// Renderizando imagem '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);
```

```
  // Os fatores a serem usados na mistura
  glBlendFunc(GL_ONE, GL_ONE);
  // Função 'max' para RGB e subtração para o canal alfa:
  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{ // Soma
  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);
}
// Finalizando
glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
glBindTexture(GL_TEXTURE_2D, 0);
glDeleteFramebuffers(1, &temporary_framebuffer);
```

8.6.2. Expressões Secundárias de Imagem: Transformadores

Uma expressão secundária de imagem tem a seguinte sintaxe:

```
<Secundário de Imagem> -> <Primário de Imagem> |
                          <Secundário de Imagem><Transformador>
<Transformador> -> rotated <Primário Numérico> |
                   scaled <Primário Numérico> |
                   shifted <Par Primário> |
                   slanted <Numérico Primário> |
                   xscaled <Numérico Primário> |
                   yscaled <Numérico Primário> |
                   zscaled <Par Primário> |
                   transformed <Primário de Transformação>
```

Os mesmos transformadores que podem ser usados em pares, transformações, caminhos e canetas podem também ser usados em imagens. Mas o modo como transformamos as imagens é diferente. Assim como canetas, as transformações lineares sobre imagens são armazenadas na forma de uma matriz que acumula todas as transformações. Desta forma, podemos realizar as transformações de uma só vez somente depois que o usuário especificar todas as que serão necessárias. Mas ao contrário de canetas, as variáveis de imagem não possuem uma matriz própria armazenada na variável.

As matrizes são recursos que irão existir somente enquanto estivermos avaliando expressões secundárias de imagem. E serão efetivamente usadas para transformar a imagem somente ao terminarmos a avaliação de uma expressão tercisária. Desta forma, no código abaixo:

```
a = img totated 45 slanted 0.2 zscaled(2, 3)
```

Ao invés de realizarmos três transformações diferentes, nós apenas acumularemos as transformações em uma matriz. Já quando não houver mais transformação e estivermos terminando de interpretar a expressão terciária, é quando usaremos a matriz para efetivamente transformar a imagem em sua forma final. Isso significa que a função que avaliará as expressões secundárias

de imagem deverá ser diferente das demais. Ela irá receber dois argumentos adicionais: uma matriz pré-inicializada como a matriz identidade e um ponteiro para uma variável booleana pré-inicializada como falso. Esta variável será mudada para verdadeira somente se a matriz passada for modificada.

Seção: Imagem: Avalia Expressão Secundária em '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;
  }
}
```

A declaração da função que avalia expressões secundárias de imagem é:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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);
```

E a sua implementação consiste primeiramente em percorrermos toda a expressão até achar o último token com um operador secundário de transformação. Se não achamos nada, é só avaliar tudo como expressão primária. Se achamos um, avaliamos tudo antes dele como expressão secundária e aplicamos a transformação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
              <Seção a ser Inserida: Secundário de Imagem: Rotação>
    }
    else if(last_transformer -> type == TYPE_SCALED){
               <Seção a ser Inserida: Secundário de Imagem: Escala>
    }
    else if(last_transformer -> type == TYPE_SHIFTED){
           <Seção a ser Inserida: Secundário de Imagem: Deslocamento>
    }
    else if(last_transformer -> type == TYPE_SLANTED){
             <Seção a ser Inserida: Secundário de Imagem: Inclinação>
    }
    else if(last_transformer -> type == TYPE_XSCALED){
              <Seção a ser Inserida: Secundário de Imagem: X-Escala>
    }
    else if(last_transformer -> type == TYPE_YSCALED){
              <Seção a ser Inserida: Secundário de Imagem: Y-Escala>
    }
    else if(last_transformer -> type == TYPE_ZSCALED){
              <Seção a ser Inserida: Secundário de Imagem: Z-Escala>
    }
    else if(last_transformer -> type == TYPE_TRANSFORMED){
      <Seção a ser Inserida: Secundário de Imagem: Transformação Genérica>
    }
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   TYPE_T_PICTURE);
    return false;
  }
}
```

Avaliar cada um dos diferentes tipos de transformadores significa modificar a matriz multiplicando ela pela matriz que representa a transformação. Exatamente como fizemos com as canetas.

Esta é a mudança de escala:

Seção: Secundário de Imagem: Escala:

```
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;
```

Este é o código que realiza a rotação:

Seção: Secundário de Imagem: Rotação:

```
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;
```

O código que desloca a imagem uma coordenada (x, y) segue abaixo. Para imagens, nós representamos o deslocamento em pixels, então antes de aplicá-lo à matriz, devemos convertê-lo para coordenadas OpenGL que dependem do tamanho da imagem:

Seção: Secundário de Imagem: Deslocamento:

```
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;
```

O código que inclina uma imagem:

Seção: Secundário de Imagem: Inclinação:

```
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;
```

O código que muda a escala somente no eixo x:

Seção: Secundário de Imagem: X-Escala:

```
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;
```

O código que muda a escala somente no eixo y:

Seção: Secundário de Imagem: Y-Escala:

```
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;
```

Mudança de escala usando o plano complexo e multiplicaçãop por número complexo:

Seção: Secundário de Imagem: Z-Escala:

```
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;
```

Finalmente, a transformação genérica que aplica a transformação linear armazenada em um transformador:

Seção: Secundário de Imagem: Transformação Genérica:

```
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;
```

Agora resta definir como iremos aplicar as transformações lineares sobre uma imagem uma vez que tenhamos a matriz, a variável de imagem de origem e a de destino. Isso é feito com ajuda da seguinte função auxiliar:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool apply_image_transformation(struct metafont *mf,
                                struct picture_variable *dst,
                                struct picture_variable *origin,
                                float *matrix);
```

Aplicar a transformação da imagem envolve usar a matriz de transformação obtida para transformar a imagem de origem gerando assim a nova imagem de destino. Para isso aplicamos as seguintes etapas:

1) Devemos descobrir qual deve ser o tamanho em pixels da imagem de destino. Para isso, multiplicamos as coordenadas da origem pela matriz. Mas neste caso, medidmos tais coordenadas por pixels, não pelas coordenadas OpenGL. Se a imagem de origem é um quadrado de 5 pixels de lado, um de seus vértices será $(-5/2, -5/2)$ e outro será $(5/2, 5/2)$. O resultado da transformação serão as coordenadas dos vértices medidos por pixels, à partir das quais obtemos o tamanho em pixels. Nesta etapa, podemos ignorar a translação presente na matriz.

2) Para levar em conta a translação, devemos somar o dobro da distância em píxels da translação ao tamanho da imagem. Lembre-se que a translação não muda o centro da imagem. Então, ao deslocar $(1, 1)$ uma imagem com um píxel preto 1×1, geramos nova imagem de 3×3 com o píxel preto no canto. O centro da imagem continua sendo a posição em que o píxel preto estava antes do deslocamento.

3) A mudança de tamanho do desenho da imagem original será obtida simplesmente renderizando o desenho transformado em uma imagem de destino ou menor. Então devemos remover da matriz de transformação a mudança de tamanho e escala que estiver presente. Também devemos ajustar levando em conta isso qualquer translação sendo feita. Isso é feito aplicando um valor calculado de correção para as posições x e y da matriz.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  // Para calcular o tamanho final da imagem, armazenaremos as maiores e
  // menores coordenadas que encontrarmos nas extremidades da imagem
  // nos dois eixos após aplicar a matriz de transformação. (Etapa 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;
  }
  // Ajuste de tamanho da imagem final devido ao deslocamento (Etapa 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]);
  // Ajustando escala e translação da imagem final levando em conta novo tamanho
  // do destino (Etapa 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;
  }
  // Gerando textura inicial, framebuffer
  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;
  }
  // Renderizando:
  render_picture(origin, matrix, dst -> width, dst -> height, true);
  // Finalização
```

```
  glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
  glDeleteFramebuffers(1, &temporary_framebuffer);
  return true;
}
```

8.6.3. Expressões Primárias: Inversores, Identidade e Imagens Vazias

Segundo a gramática da linguagem, a sintaxe para expressões primárias de imagens é:

```
<Primário de Imagem> -> <Variável de Imagem> |
                        nullpicture <Primário de Par> |
                        ( <Expressão de Imagem> ) |
                        <Mais ou Menos> <Primário de Imagem> |
                        subpicture <Primário de Par> and <Primário de Par> of
                                   <Primário de Imagem>
```

Isso requer registrar “nullpicture” e “subpicture” como um token:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_NULLPICTURE,        // O token simbólico 'nullpicture'
TYPE_SUBPICTURE,         // O token simbólico 'subpicture'
```

E requer que essas strings seja adicionada à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"nullpicture", "subpicture",
```

A função que irá interpretar expressões primárias de imagem é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E sua implementação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){
                <Seção a ser Inserida: Imagem Primária: Variável>
    }
  }
  else{
    if(begin -> type == TYPE_OPEN_PARENTHESIS &&
      end -> type == TYPE_CLOSE_PARENTHESIS){
              <Seção a ser Inserida: Imagem Primária: Parênteses>
    }
    else if(begin -> type == TYPE_NULLPICTURE){
          <Seção a ser Inserida: Imagem Primária: Imagem em Branco>
    }
    else if(begin -> type == TYPE_SUM){
              <Seção a ser Inserida: Imagem Primária: Identidade>
    }
    else if(begin -> type == TYPE_SUBTRACT){
```

```
                <Seção a ser Inserida: Imagem Primária: Inverso>
    }
    else if(begin -> type == TYPE_SUBPICTURE){
                <Seção a ser Inserida: Imagem Primária: Subimagem>
    }
  }
  RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                 TYPE_T_PICTURE);
  return false;
}
```

O primeiro caso de expressão primária de imagem é ler uma imagem de uma variável. Isso envolve copiar o conteúdo da variável armazenada no resultado da avaliação.

Seção: Imagem Primária: Variável:

```
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;
}
// Preparar renderização:
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);
// Renderiza:
render_picture(content, identity_matrix, result -> width, result -> height,
true);
// Finaliza:
glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
glDeleteFramebuffers(1, &temporary_framebuffer);
return true;
```

Lidar com os parênteses envolve percorrer os tokens até encontrar o último token antes do fechar de parênteses. E avaliar como uma nova expressão de imagem tudo aquilo que estiver entre o parênteses inicial e o final:

Seção: Imagem Primária: Parênteses:

```
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);
```

A próxima expressão primária é a criação de uma imagem em branco com um dado tamanho. Isso ocorre quando lemos um token `nullpicture` seguido de um par primário. Criar a imagem vazia envolve primeiro interpretar o par primário que contém o seu tamanho e em seguida criar uma textura vazia com tal tamanho.

Seção: Imagem Primária: Imagem em Branco:

```
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;
}
// Pintando a nova textura de branco
memset(data, 255, result -> width * result -> height * 4);
{ // E deixando ela totalmente transparente:
  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;
```

E agora o antepenúltimo caso de imagem primária: quando vem um token `+` antes de uma imagem. Neste caso o operador não faz nada, é um operador de identidade para imagens. Então

só precisamos ignorar ele e avaliar os tokens restantes:

Seção: Imagem Primária: Identidade:

```
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);
```

Para o operador `-` que é usado para inverter uma imagem, nós iremos novamente renderizar ativando a equação de mistura de cores, mas fazendo com que a nova cor seja obtida usando $(1, 1, 1, 1)$ como a cor atural e subtraindo disso a nova cor. Isso é obtido tratando os fatores de mistura como 1 (GL_ONE) e usando a subtração como operador (GL_FUNC_SUBTRACT):

Seção: Imagem Primária: Inverso:

```
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;
// Preparar renderização:
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);
// Inicializar nova textura com branco opaco (1, 1, 1, 1)
glClearColor(1.0, 1.0, 1.0, 1.0);
glClear(GL_COLOR_BUFFER_BIT);
// Ajustar equação de mistura pra subtrair ela do branco opaco:
glEnable(GL_BLEND);
glBlendFuncSeparate(GL_ONE, GL_ONE, GL_ONE, GL_ONE);
glBlendEquationSeparate(GL_FUNC_REVERSE_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT);
// Renderiza:
render_picture(&p, identity_matrix, result -> width, result -> height, false);
// Finaliza:
glDisable(GL_BLEND);
glDeleteTextures(1, &(p.texture));
glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
glDeleteFramebuffers(1, &temporary_framebuffer);
return true;
```

FInalmente, a expressão primária de computar sub-imagens. Esta expressão começa com um token `subpicture`, recebe um par primário (o offset em pixels), um token `and`, um segundo par (com o tamanho da sub-imagem a ser extraída), o token `of` e uma expressão primária de imagem (de onde a sub-imagem deve ser extraída).

A primeira parte da avaliação desta expressão envolve extrair os pares e a imagem das subex-

pressões. A segunda parte é executar o comando de gerar uma sub-imagem:

Seção: Imagem Primária: Subimagem:

```
struct pair_variable pair_offset, subpicture_size;
struct picture_variable original_picture;
```

<Seção a ser Inserida: Subimagem: Extrai Subexpressões>

<Seção a ser Inserida: Subimagem: Extrai Subimagem>

```
return false;
```

Extrair as subexpressões envolve delimitar o começo e fim de cada uma delas. Para isso percorremos a expressão como um todo buscando pelos tokens de auxílio `of` e `and`. Armazenamos também um estado para saber quantas subexpressões já delimitamos. Isso nos ajuda a detectar se temos uma expressão mal-formada onde os tokens de auxílio aparecem, mas na ordem errada ou sem expressões entre eles:

Seção: Subimagem: Extrai Subexpressões:

```
{
  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;
}
```

Extrair a sub-imagem requer gerar uma nova textura com o tamanho indicado pelo segundo par e renderizar a imagem lida nesta nova textura, de acordo com as informações presentes nos pares sobre tamanho e offset:

Seção: Subimagem: Extrai Subimagem:

```
{
  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;
  // Inicializar nova textura com branco transparente (1, 1, 1, 0)
  glClearColor(1.0, 1.0, 1.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT);
  // Ajustando tamanho da renderização na textura
  render_matrix[0] = original_picture.width / subpicture_size.x;
  render_matrix[4] = original_picture.height / subpicture_size.y;
  // Passando o offset para a matriz de renderização
  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;
  // Renderiza:
  render_picture(&original_picture, render_matrix, result -> width, result ->
height,
                 false);

  // Finaliza:
  glDisable(GL_BLEND);
  glDeleteTextures(1, &(original_picture.texture));
  glBindFramebuffer(GL_FRAMEBUFFER, previous_framebuffer);
  glDeleteFramebuffers(1, &temporary_framebuffer);
  return true;
}
```

8.6.4. Imagens em Expressões Numéricas

Há casos nos quais precisamos avaliar uma expressão de imagem dentro de uma expressão

numérica. Há três expressões primárias numéricas adicionais onde devemos avaliar uma imagem:

```
<Primário Numérico> -> totalweight <Primário de Imagem> |
                       width <Primário de Imagem> |
                       height <Primário de Imagem>
```

Os dois últimos operadores apenas retornam respectivamente a largura e altura da imagem. Já o operador `totalweight` avalia uma expressão de imagem, lê a imagem obtida e retorna a soma do "peso" de cada pixel. O peso de um pixel branco transparente deve ser zero e o de um pixel preto opaco deve ser 1. Já para calcular valores intermediários, converteremos um pixel colorido para um tom de cinza. Para isso obteremos uma média ponderada entre os valores entre 0 e 1 deles, dando um peso maior para o verde e menor para o azul. Isso porque a visão humana é mais sensível ao verde que ao azul. O resultado desta média sera o peso do píxel colorido. Depois de obter o valor, multiplicamos pela transparência.

Primeiro vamos adicionar novos tipos de tokens para esses operadores:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_TOTALWEIGHT,        // O token simbólico 'totalweight'
TYPE_WIDTH,              // O token simbólico 'width'
TYPE_HEIGHT,             // O token simbólico 'height'
```

E adicionamos seus nomes à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"totalweight", "width", "height",
```

Para o primeiro operador, o modo pelo qual iremos obter os pixels de uma textura será renderizando ela para um framebuffer e lendo usando `glReadPixels` para obter o conteúdo do framebuffer:

Seção: Primário Numérico: Operadores Adicionais (continuação):

```
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;
  }
```

```
  // Renderiza:
  render_picture(&p, identity_matrix, p.width, p.height, true);
  // Ler dados do framebuffer:
  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;
  }
  // Finalização:
  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;
}
```

Para os operadores `width` e `height`, tudo é mais simples. Basta obter a imagem e retornar respectivamente a largura e altura. No caso da largura:

Seção: Primário Numérico: Operadores Adicionais (continuação):

```
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;
}
```

E no caso da altura:

Seção: Primário Numérico: Operadores Adicionais (continuação):

```
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. Atribuições e Expressões Booleanas

Como em todos os outros tipos de atribuições e expressões, começamos definindo o código que faz a atribuição de variáveis após avaliar a expressão:

Seção: Atribuição de Variável Booleana:

```
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);
  }
}
```

Podendo atribuir o resultado de expressões às variáveis, vamos agora à parte de escrever código para avaliar as expressões booleanas.

8.7.1. Comparações

Comparações são feitas usando relações. Elas permitem avaliar se duas variáveis ou expressões nos dão o mesmo resultado, um resultado diferente, bem como avaliar a relação de ordem entre seus resultados. As regras gramaticais para comparar valores são:

```
<Expressão Booleana> -> <Terciário Booleano> |
                        <Expressão Numérica> <Relação> <Terciário Numérico> |
                        <Expressão de Par> <Relação> <Terciário de Par>     |
                        <Expressão Booleana> <Relação> <Terciário Booleano> |
                        <Expressão de Transformação> <Relação>
                                              <Terciário de Transformação>
<Relação> -> < | <= | > | >= | = | <>
```

Isso significa que as comparações usando estas relações são expressões quarternárias. Elas tem uma precedência ainda menor que as expressões terciárias booleanas. Para outros tipos de expressão, nós consideramos uma expressão terciária como sendo sinônimo para a expressão daquele tipo. Mas para expressões booleanas, expressões terciárias são um tipo interno das expressões mais gerais.

As relações mostradas acima requerem que novos tipos de tokens sejam considerados:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_LT,                    // O token simbólico '<'
TYPE_LEQ,                   // O token simbólico '<='
TYPE_GT,                    // O token simbólico '>'
```

```
TYPE_GEQ,                    // O token simbólico '>='
TYPE_NEQ,                    // O token simbólico '<>'
```

E adicionamos seus nomes à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"<", "<=", ">", ">=", "<>",
```

A declaração da função que avaliará expressões booleanas é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

Esta função deverá checar se temos um dos operadores de relações booleanas. Se não tivermos, a expressão deve ser avaliada como terciária. Se tivermos, devemos identificar a relação mais à direita e avaliar ela após avaliar as duas outras sub-expressões. Mas antes de fazer isso, precisamos saber qual o tipo destas expressões. Para isso, vamos assumir que temos uma função que dada uma expressão terciária, retorna seu tipo. Ela será chamada `get_tertiary_expression_type`. Iremos definir ela na Subseção 8.8. Mas por hora, vamos simplesmente assumir que ela existe.

A implementação desta função é:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. A Operação OR

O operador booleano OR é a única operação booleana terciária. A gramática das expressões booleanas terciárias é:

```
<Terciário Booleano> -> <Terciário Booleano> or <Secundário Booleano> |
                        <Secundário Booleano>
```

Para implementarmos esta expressão, precisamos então definir `or` como um novo tipo de token simbólico:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_OR,                    // O token simbólico 'or'
```

O qual deve ser adicionado à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"or",
```

A declaração da função que tratará expressões terciárias é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E a implementação desta função segue o modelo esperado. Ela percorre a lista de tokens até achar o último `or` não-aninhado em parênteses e outros delimitadores e aplica o operador sobre o resultado das subexpressões que o cercam. Se não existir um operador `or`, toda a expressão é tradada como uma expressão secundária:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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_secondary(mf, cx, begin, before_last_operator, &a))
      return false;
    if(!eval_boolean_tertiary(mf, cx, last_operator -> next, end, &b))
      return false;
```

```
    result -> value =  (a.value || b.value);
    return true;
  }
}
```

8.7.3. A Operação AND

O operador booleano AND é a única operação booleana secundária. A gramática das expressões booleanas secundárias é:

```
<Secundário Booleano> -> <Secundário Booleano> and <Primário Booleano> |
                         <Primário Booleano>
```

O token `and` já foi definido previamente. Ele é também usado para descrever caminhos, quando há dois pontos de controle diferentes em uma curva.

A declaração da função que tratará expressões secundárias booleanas é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

E a implementação desta função é praticamente igual ao do operador OR, apenas mudando qual operador estamos procurando e a operação que aplicamos sobre o resultado das subexpressões que delimitam o AND. E, caso não haja um operador AND não-aninhado na expressão, nós tratamos ela inteira como uma expressão booleana primária:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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. Expressões Primárias Booleanas: Literais e Predicados Simples

A gramática das expressões primárias booleanas é:

```
<Primário Booleano> -> <Variável Booleana> | true | false |
                       cycle <Caminho Primário> | odd <Primário Numérico> |
                       not <Primário Booleano> |
                       ( <Expressão Booleana> )
```

A maioria destas expressões é auto-explictiva. Uma variável booleana é avaliada para seja qual for o seu valor armazenado. Os valores `true` e `false` representam verdadeiro e falso. O `not` é o operador NOT de negação. Parênteses podem ser usados para mudar a ordem de avaliação de operadores. O token `odd` serve para checar se o número à seguir é ímpar após arredondá-lo para o inteiro mais próximo. E `cycle` serve para checar se um caminho é cíclico ou não.

Os seguintes novos tipos de token não foram ainda definidos:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_TRUE,                    // O token simbólico 'true'
TYPE_FALSE,                   // O token simbólico 'false'
TYPE_ODD,                     // O token simbólico 'odd'
TYPE_NOT,                     // O token simbólico 'not'
```

E o nome de cada um deles deve ser adicionado à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"true", "false", "odd", "not",
```

A função que avaliará expressões booleanas primárias é:

Seção: Declaração de Função Local (metafont.c) (continuação):

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

No caso de expressões booleanas primárias, é perfeitamente possível identificar qual regra gramatical seguir após observar o primeiro token encontrado. Então a função apenas observa o primeiro token, e depois disso ela decide o que fazer:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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: // Variável
      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: // Verdadeiro
    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;
  }
}
```

8.8. Identificando Tipos de Expressões

Muitas vezes temos uma expressão e devemos identificar qual o tipo da expressão para tratá-la corretamente. Por exemplo, quando temos uma expressão booleana comparando qual valor é maior, o que estamos comparando pode ser uma expressão numérica, de par, de transformação ou outra expressão booleana. Em uma expressão numérica, podemos usar o operador `length` sobre outros números ou sobre pares. Uma multiplicação pode ser entre números ou entre um escalar e um vetor. Os operadores `xpart` funcionam tanto em pares como em transformadores.

Note que nenhuma das vezes em que precisamos identificar o tipo de uma expressão espera-se que encontremos uma imagem ou uma caneta. Por causa disso, a prioridade de identificar corretamente tais expressões é menor. Contudo, saber identificá-las permite que mensagens de erro sejam mais explicativas, pois poderemos avisar que encontramos uma expressão de um tipo, quando esperávamos de outro.

Identificar o tipo de uma expressão pode ser complexo e trabalhoso. Felizmente, os casos mais comuns tentem a ser simples e serão avaliados rapidamente. De qualquer forma, vamos usar funções diferentes para poder identificar expressões primárias, secundárias e terciárias:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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);
```

Para identificar expressões primárias, primeiro podemos tratar os casos mais simples (a expressão é composta por um único token) e depois lidar com os casos mais complexos. As regras que nos permitem identificar são:

1) Uma variável é uma expressão com o tipo da variável. Precisamos apenas tratar de forma diferente apenas algumas variáveis especiais cuja existência é temporária.

2) São expressões booleanas as que tem como único token `true` ou `false`. Ou que começam com `cycle`, `odd` e `not`.

3) São expressões de pares os que começam com `point`, `precontrol` e `postcontrol`. Ou se dentro da expressão tivermos um `[`. E também algo que começa com token numérico e tem parênteses ou o começo de um primário de par um pouco depois (`-3(8, 4)`).

4) É expressão numérica um único token numérico, o token `normaldeviate`, um token numérico seguido por `/` e toda expressão que começa com `length`, `xpart`, `ypart`, `xxpart`, `xypart`, `yxpart`, `yypart`, `angle`, `sqrt`, `sind`, `cosd`, `log`, `mexp`, `floor`, `uniformdeviate`.

5) Se temos um `+` ou `-`, descobrimos o tipo avaliando o tipo do restante da expressão.

6) Se o primeiro token é `subpath`, `makepath` ou `reverse`, é uma expressão de caminho.

7) Se o primeiro token for `nullpen`, `pencircle`, `pensemicircle` ou `makepen`, então é uma expressão de caneta.

8) Se o primeiro token for `nullpicture` ou `subpicture`, então é uma expressão de imagem.

9) Se temos parênteses e dentro deles temos uma vírgula, é um par. Se houverem mais vírgulas, é uma transformação. Se não houverem vírgulas, devemos avaliar o tipo da expressão terciária interna.

10) Nos demais casos, o tipo é desconhecido e um erro deve ter ocorrido.

Na nossa implementação, para testar mais rapidamente, primeiro checamos se temos ou não um caso simples composto por um único token. Todos os casos acima são tratados separadamente de acordo com isso:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){ // Caso 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) // Caso 2
      return TYPE_T_BOOLEAN;
    if(begin_expr -> type == TYPE_NUMERIC ||
       begin_expr -> type == TYPE_NORMALDEVIATE) // Caso 4
      return TYPE_T_NUMERIC;
    if(begin_expr -> type == TYPE_NULLPEN || // Caso 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) // Caso 2
    return TYPE_T_BOOLEAN;
  // Expressão de par (caso 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;
  // Expressão numérica (Caso 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;
  // Caso 5:
  if(begin_expr -> type == TYPE_SUM || begin_expr -> type == TYPE_SUBTRACT)
    return get_primary_expression_type(mf, cx, begin_expr -> next, end_expr);
  // Token numérico, pode ser expressão numérica ou de par:
  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;
  }
  // Caso 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) // Caso 7
    return TYPE_T_PEN;
  if(begin_expr -> type == TYPE_NULLPICTURE || // Caso 8
     begin_expr -> type == TYPE_SUBPICTURE)
    return TYPE_T_PICTURE;
  // Pode haver expressão de par que começa com parênteses e não termina:
  // (1+1)[p1, p2] e também: 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;
  }
}
```

Agora vamos às expressões secundárias. As regras que usaremos são:

1) Se temos um único token, avaliamos tudo como expressão primária.

2) Se temos `and`, a expressão é booleana.

3) Se temos um transformador (`transformed`, `rotated`, `scaled`, `shifted`, `slanted`, `xscaled`, `yscaled`, `zscaled`), ignoramos ele e o que há depois dele, voltando a avaliar o restante da expressão como expressão secundária. Pode ser um transformador, par, caminho, caneta ou imagem.

4) Se temos multiplicações e divisões, devemos checar a que está mais à direita. Se um dos operandos for um par, é um par. Caso contrário, é expressão numérica. Lembrando que a regra para a divisão é um pouco complexa. Um token `/` só é divisão quando não é delimitada por dois tokens numéricos. Neste caso, temos uma fração, não uma divisão, e não há aí um operador secundário. A exceção é quando o token anterior já faz parte de uma fração. Assim, 1/3 é fração ao invésd e divisão, e não há nenhum operador secundário aí. Já 1/3/1/3 contém um operador secundário de divisão, dividindo duas frações de 1/3.

5) Nos demais casos, não pudemos identificar e o tipo deve ser avaliado como se a expressão fôsse primária.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
}
```

Por fim, as expressões terciárias. Neste caso, as regras seguidas são:

1) Se temos um único token, avaliamos tudo como expressão primária.

2) Se temos um `or` ou uma relação (`<`, `<=`, `>`, `>=`, `=`, `<>`), então é uma expressão booleana. Tecnicamente as relações não são expressões terciárias. A ordem de precedência delas seria de expressões quarternárias. Mas podemos considerar todos esses operadores aqui, já que não estamos de fato avaliando as expressões, onde seguir regras de precedência é fundamental.

3) Se temos tokens de junção de caminho como `&`, `..` ou `--`, então também não é uma expressão terciária, mas uma expressão de caminho que tem precedência menor ainda. De qualquer forma, podemos retornar a informação de que temos uma expressão de caminho. O mesmo ocorre se encontrarmos o token "{", que é usado apenas na especificação de direção em caminhos.

4) Se temos soma e subtração pitagóricos (`++`, `+-+`), então é expressão numérica.

5) Se temos soma ou subtração, o tipo da expressão é o mesmo de qualquer um de seus operandos.

6) Nos demais casos, tentamos novamente avaliando a expressão como secundária.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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; // 1o aninhamento '{': especifica direção (Caso 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. Declaração Composta: Declaração Condicional

Uma declaração condicional é um `if`, uma estrutura que garante que certos trechos de código sejam executados somente se certas condições forem verdadeiras. A gramática completa para expressões deste tipo é:

```
<Bloco Condicional> -> if <Expressão Booleana> :
                         <Lista de Instruções>
                         <Alternativas>
                       fi
<Alternativas> -> <Vazio> |
                  elseif <Expressão Booleana>:
                    <Lista de Instruções>
                  <Alternativas> |
                  else: <Lista de Instruções>
```

Declararemos então os seguintes novos tipos de tokens:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_ELSEIF,  // O token simbólico 'elseif'
TYPE_ELSE,    // O token simbólico 'else'
```

```
TYPE_COLON,   // O token simbólico ':'
```

Os quais devem ser adicionados à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"elseif", "else", ":",
```

Na Seção 6.2 nós vimos a primeira das declarações compostas. A que começa com `begingroup` e termina com `endgroup`. A declaração condicional usando `if` também é uma declaração composta do tipo, que contém dentro de si várias outras declarações.

Lembre-se que código é primeiro avaliado na função `eval_list_of_statement`, qe separa o código usando ponto-e-vírgula como delimitador e passa para a função `eval_statement` tratar cada parte. Esta função deve interpretar o código que recebe, mas tem também a liberdade de mudar a posição do ponteiro que delimita o fim da expressão que avaliou, o qual é usado para depois escolher os próximos tokens a serem lidos. É nessa segunda função que o código que trata um `if` é localizado.

Assim como no caso do `begingroup` da Seção 6.2, quando encontramos tokens a serem avaliados que começam com `if`, o que fazemos é iniciar um novo nível de aninhamento e depois disso corrigimos a posição do ponteiro do último token avaliado para que os próximos tokens a serem avaliados sejam os corretos. Mas aqui, para determinar a posição do ponteiro do próximo token temos que executar uma lógica mais complexa, avaliando expressões booleanas.

Basicamente, diante de um `if`, primeiro avaliamos a expressão booleana diante dele. Se ela for verdadeira, posicionamos o ponteiro para o próximo dois pontos (":") encontrado e assumiremos que todos os outros `elseif` e `else` não devem ser executados. Se não, posicionamos o ponteiro para o endereço armazenado no próprio token `if` e que foi inicializado durante a análise léxica, quando os tokens foram gerados.

No caso de um `elseif`, novamente avaliamos sua expressão booleana e nos comportamos exatamente como no caso do `if`. Já no caso de um `else`, sempre devemos colocar o ponteiro diante de seu corpo. E se não encontrarmos nem um `else`, e nem um `elseif` onde a expressão booleana é verdadeira, então pulamos a declaração condicional, colocando o ponteiro de fim no próprio `fi`.

O código que trata então um `if` é:

Seção: Instrução: Composta (continuação):

```
else if(begin -> type == TYPE_IF){
  struct generic_token *begin_bool, *end_bool;
  struct boolean_variable b;
  // Delimitando a expressão booleana
  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;
  }
  // Avaliando a expressão booleana:
  if(!eval_boolean_expression(mf, cx, begin_bool, end_bool, &b))
    return false;
  if(b.value == 1){ // Verdadeiro: código no 'if' deve ser avaliado
    *end = end_bool -> next;
    return true;
```

```
  }
  else{ // Falso: pulamos o código no 'if'
    struct generic_token *t = ((struct linked_token *) begin) -> link;
    while(t != NULL){
      if(t -> type == TYPE_FI){
        // Achamos o 'fi' correspondente ao nosso 'if'
        *end = t;
        return true;
      }
      else if(t -> type == TYPE_ELSE){
        // Achamos o 'else' correspondente ao nosso '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){
        // Achamos um 'elseif' correspondente ao nosos '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){ // Verdadeiro: código no 'elseif' deve ser avaliado
          *end = end_bool -> next;
          return true;
        }
      }
      t = ((struct linked_token *) t) -> next;
    }
  }
  // Isso nunca deve ocorrer: o lexer deve detectar se há 'if' sem 'fi':
  RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
  return false;
}
```

A responsabilidade de encontrar o código certo a ser executado dependendo da condicional é do código mostrado mais acima. Mas nós podemos também encontrar um `elseif` ou um `else` no começo de uma declaração a ser avaliada. Quando isso ocorre, ou indica que é um erro de sintaxe, quando um destes tokens está solto, sem um `if` correspondenteo, ou então significa que estamos dentro de um `if`, onde já terminamos de executar o código correto e encontramos um delimitador

onde novas expressões booleanas podems ser testadas. Independente da expressão que estiver lá, nós já terminamos de executar o nosso `if`. Então devemos ignorar tudo que vem a seguir até achar o `fi` que encerra o `if` em que estamos:

Seção: Instrução: Composta (continuação):

```
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){
      // Achamos o 'fi' correspondente ao nosso 'if'
      *end = t;
      return true;
    }
    t = ((struct linked_token *) t) -> link;
  }
  // Isso nunca deve ocorrer: o lexer deve detectar se há 'if' sem 'fi':
  RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, OPTIONAL(begin -> line));
  return false;
}
```

Finalmente, nós podemos também encontrar um `fi` no começo de declaração a ser avaliada. Neste caso, nós apenas a ignoramos e seguimos adiante para interpretar o que vem depois dela:

Seção: Instrução: Composta (continuação):

```
else if(begin -> type == TYPE_FI){
  *end = begin;
  return true;
}
```

10. Declaração Composta: Iterações

Uma iteração é um comando composto `for` que faz com que o código interno a ele possa ser executado entre zero e um número arbitrário de vezes. A gramática para este tipo de comando é:

```
<Bloco de Iteração> -> for <Cabeçalho For>:
                         <Lista de Instruções>
                       endfor
<Cabeçalho For> -> <Variável Numérica>  = <Expressão Numérica><Progressão> |
                   <Variável Numérica> := <Expressão Numérica><Progressão>
<Progressão> -> step <Expressão Numérica> until <Expressão Numérica>
```

Por exemplo, a seguinte iteração é válida:

```
numeric i;
for i = 5 step 10 until 50:
  draw (i, 0);
endfor
```

Na iteração acima o comando de desenho `draw` é aplicado sobre os pontos $(5, 0)$, $(15, 0)$, $(25, 0)$, $(35, 0)$, $(45, 0)$. O valor de `i` começa como sendo 5 e a cada iteração tal valor é incrementado em 10. No último incremento, o valor se torna igual a 55, o que é maior que 50, o valor colocado como o limite da iteração. Isso faz com que a iteração e o comando de desenho interno pare de ser executado.

Vamos precisar então dos novos tipos de tokens:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_STEP,  // O token simbólico 'step'
```

```
TYPE_UNTIL, // O token simbólico 'until'
```

Os quais são adicionados à lista de palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"step", "until",
```

Executar um código de loop envolve uma inicialização (que deve ser executado caso o loop ainda não tenha começado) e o código de teste da condição, que incrementa a variável de controle e checa se o loop deve continuar a executar ou encerrar:

Seção: Instrução: Composta (continuação):

```
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);
                <Seção a ser Inserida: Interação: Prepara Iteração>
  }
                <Seção a ser Inserida: Interação: Checa Condição>
}
```

Para preparar a iteração, devemos ler o cabeçalho, que é composto por uma variável numérica e duas expressões numéricas delimitadas por palavras reservadas `step` e `until`. A variável numérica deve ser encontrada imediatamente após o `for`:

Seção: Interação: Prepara Iteração:

```
{
  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);
}
```

Depois da variável, devemos obter um token "`=`" ou "`:=`":

Seção: Interação: Prepara Iteração (continuação):

```
{
  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;
  }
}
```

Podemos então delimitar e extrair o resultado da primeira expressão numérica que começa logo após o símbolo de atribuição e vai até antes do token `step`. O valor lido deve inicializar a variável de controle da iteração. Uma expressão numérica não pode conter um token `step`, então não precisamos nos preocupar com aninhamentos de parênteses e expressões ao avaliar esta parte:

Seção: Interação: Prepara Iteração (continuação):

```
{
  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;
}
```

A próxima coisa a fazer será avaliar a expressão após o `step` e antes do `until` para saber o quanto o laço precisa ser incrementado. Note, contudo, que o incremento não deve ser feito na inicialização, mas sim a cada vez que formos checar novamente a condição da iteração. Como a expressão que controla este incremento pode ter o seu resultado modificado sempre precisaremos reavaliar ela.

Seção: Interação: Checa Condição:

```
{
  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;
}
```

Por fim, temos que obter a última expressão: aquela que determina a condição de parada da iteração. Ela é delimitada pelo token "`until`" e pelo token "`:`":

Seção: Interação: Checa Condição:

```
{
  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)){
    // Interromper loop:
    for_token -> running = false;
    *end = (struct generic_token *) for_token -> end;
    if(!end_nesting_level(mf, cx, *end))
      return false;
    return true;
  }
  else{
```

```
    // Continuar loop:
    *end = end_expr -> next;
    return true;
  }
}
```

Já quando encontramos um token `endfor`, nosso comportamento deve ser configurar para que o próximo comando executado deva ser novamente o `for` correspondente. Para isso, nós apenas ajustamos o ponteiro de final do nosso comando para o ponteiro antes do `for`, que foi armazenado no token `endfor` durante a criação dos tokens na análise léxica.

Seção: Instrução: Composta (continuação):

```
else if(begin -> type == TYPE_ENDFOR){
  struct linked_token *endfor_token = (struct linked_token *) begin;
  *end = endfor_token -> link;
  return true;
}
```

11. O Comando `pickup`

Agora podemos começar a definir o nosso primeiro comando. A sintaxe do comando `pickup` é dada por:

```
<Comando> -> <Comando 'pickup'>
<Comando 'pickup'> -> pickup <Caneta a Ser Pega> <Transformadores Opcionais>
<Caneta a Ser Pega> -> nullpen | pencircle | pensemicircle |
                       <Variável de Caneta>
<Transformadores Opcionais> -> <Vazio> |
                               <Transformador> <Transformadores Opcionais>
```

O comando requer que criemos um novo token e palavra reservada:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_PICKUP,   // O token simbólico 'pickup'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"pickup",
```

O que este comando faz é armazenar um novo valor na variável `currentpen`, a qual usaremos para fazer desenhos (conforme definiremos na próxima Seção).

11.1. Pontos de Extremidade de Caneta

Uma coisa que devemos fazer ao recebermos uma caneta com o comando `pickup` é armazenar 4 valores internamente. Dois deles para armazenar o menor e maior valor na coordenada x que esta caneta gera ao ser renderizada e também o menor e maior valor na coordenada y. Armazenar tais valores será útil para que possamos ajustar melhor os pontos que vamos desenhar. Por exemplo, podemos querer desenhar com a caneta o mais próximo possível do canto inferior esquerdo da imagem, sem que a caneta saia para fora da imagem. Tal tipo de controle requer que saibamos o tamanho da caneta. Por isso vamos ter que armazenar internamente os valores `pen_lft`, `pen_rt`, `pen_top` e `pen_bot`. Estes valores ficarão armazenados em nosso contexto de execução:

Seção: Atributos (struct context) (continuação):

```
float pen_lft, pen_rt, pen_top, pen_bot;
```

Estes valores são inicializados como zero porque a `currentpen` começa como uma `nullpen` em cada contexto de execução:

Seção: Inicialização (struct context) (continuação):

```
cx -> pen_lft = cx -> pen_rt = cx -> pen_top = cx -> pen_bot = 0.0;
```

Uma operação muito comum que faremos nesta Seção será, ao gerar novos vértices que irão compor o formato da caneta, verificar se geramos um ponto mais à esquerda que `pen_lft`, mais à direita que `pen_rt`, acima de `pen_top` ou abaixo de `pen_bot`. Em tais casos devemos atualizar a informação sobre os pontos de extremidade da caneta.

Para reduzir o tamanho do código, a macro abaixo vai representar o código no qual declaramos as variáveis que armazenam a coordenada mais à esquerda, direito, acima ou abaixo de um vértice que geramos:

Seção: Macros Locais (metafont.c) (continuação):

```
#define DECLARE_PEN_EXTREMITIES() float _max_x = -INFINITY, _min_x = INFINITY,\
                                  _max_y = -INFINITY, _min_y = INFINITY;
```

Diante de um novo ponto, vamos usar a macro abaixo para obter sua coordenada (x, y), multiplicar por uma amtriz de transformação e assim checar se temos algo que deve ser armazenado como um dos menores ou maiores pontos da caneta no eixo x ou eixo y:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;\
  }
// Se a matriz é a identidade, podemos usar esta macro:
#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;\
  }
```

Depois de checarmos todos os pontos de uma caneta, podemos então atualizar seus pontos de extremidade:

Seção: Macros Locais (metafont.c) (continuação):

```
#define UPDATE_PEN_EXTREMITIES() {\
  cx -> pen_lft = _min_x;\
  cx -> pen_rt = _max_x;\
  cx -> pen_top = _max_y;\
  cx -> pen_bot = _min_y;\
}
```

As macros anteriores presumem que iremos iterar sobre cada um dos pontos que compõe o perímetro de uma caneta. Fazendo isso, podemos chamar `CHECK_PEN_EXTREMITIES` em todos os pontos e assim obtemos os que forem maiores no eixo x e y. Nos casos em que não queremos iterar sobre todos os pontos, vamos criar também algumas funções para deduzi-los para nós. Por exemplo, caso tenhamos usado o comando `pickup` sobre `pencircle` ou `pensemicircle` e a nossa caneta tenha passado por uma transformação linear representada por uma matriz. A função abaixo pode ser usada para obter os pontos de extremidade para esta caneta:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void pencircular_extremity_points(struct context *cx,
                                  float *matrix, bool fullcircle);
```

Um `pencircle` ou `pensemicircle` padrão, sem nenhuma transformação é uma caneta de raio 1/2. O que quer dizer que toda coordenada (x, y) atende à seguinte fórmula:

$$x^2 + y^2 = 0.25$$

Ou, de maneira equivalente:

$$x = \pm\sqrt{0.25 - y^2}$$

$$y = \pm\sqrt{0.25 - x^2}$$

No caso do semicírculo as mesmas fórmulas são válidas, exceto que temos a restrição adicional que $y \geq 0$.

Dada uma transformação linear definida pela matriz M, obtemos os novos valores de x e y seguindo a fórmula abaixo:

$$f_x(x) = M_{11}x \pm M_{21}\sqrt{0.25 - x^2} + M_{31}$$

$$f_y(y) = M_{22}y \pm M_{12}\sqrt{0.25 - y^2} + M_{32}(círculo)$$

$$f_y(y) = M_{22}y + M_{12}\sqrt{0.25 - y^2} + M_{32}(semicírculo)$$

Quando M_{11} é igual à 0, os pontos mais extremos no eixo x serão a transformação linear dada por M de $(-0.5, 0)$ e $(0.5, 0)$. Se M_{21} for igual à zero, os pontos mais extremos serão dados por $(0, -0.5)$ e $(0, 0.5)$ transformados por M. A mesma lógica se aplica aos extremos no eixo y quando M_{22} ou M_{12} é igual à zero. Em todos estes casos, o valor depende somente de maximizar ou a coordenada x ou y.

Para todos os demais casos, achar o ponto de máximo ou de mínimo destas funções requer calcular a derivada das fórmulas acima e igualamos à zero:

$$f'_x(x) = M_{11} \pm (M_{21}x)/(\sqrt{0.25 - x^2}) = 0$$

$$f'_y(y) = M_{22} \pm (M_{12}y)/(\sqrt{0.25 - y^2}) = 0(círculo)$$

$$f'_y(y) = M_{22} + (M_{12}y)/(\sqrt{0.25 - y^2}) = 0(semicírculo)$$

Como não há uma fórmula que nos dê a solução de maneira direta, temos que usar métodos iterativos como o Método de Newton. para achar os valores x e y que tornam as fórmulas acima corretas. Para poder usar o médodo de Newton, começamos calculando a derivada de segunda ordem destas fórmulas:

$$f''_x(x) = \pm(M_{21}\sqrt{0.25 - x^2} + (M_{21}x^2)/(\sqrt{0.25 - x^2}))$$

$$f''_y(y) = \pm(M_{12}\sqrt{0.25 - x^2} + (M_{12}x^2)/(\sqrt{0.25 - x^2}))(círculo)$$

$$f''_y(y) = +(M_{12}\sqrt{0.25 - x^2} + (M_{12}x^2)/(\sqrt{0.25 - x^2}))(semicírculo)$$

Pelo Método de Newton, uma vez que tenhamos uma estimativa x_n e y_n para o zero da função f'_x e f'_y, podemos obter uma estimativa melhor calculando:

$$x_{n+1} = x_n - (f'_x(x_n)/f''_x(x_n))$$

$$y_{n+1} = y_n - (f'_y(y_n)/f''_y(y_n))$$

O problema do Método de Newton é que como nossa função f_x e f_y está definida somente entre -0,5 e +0,5 (ou entre 0 e 0,5 no caso de f_y para semicírculos), se o valor que estamos buscando

estiver muito próximo destas extremidades, ele pode acabar nos empurrando para fora do Domínio da função. Caso isso ocorra, teremos que recorrer a outro método. No caso, usaremos o mais lento Método da Bissecção. Note que $f'_x(-0,5)$ e $f'_x(+0,5)$ tem sinais opostos sempre, assumindo que um zero da função exista. Sendo assim, este método começa com este intervalo e depois checa o sinal de f_x no meio das duas extremidades. Baseado no sinal ele reduz o tamanho do intervalo até reduzi-lo à zero e encontrar o valor.

Após revisar toda a teoria, podemos enfim implementar a função que obtém os pontos de extremidade de uma caneta redonda:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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 computa eixo x, i=1 computa eixo 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{
      // Método de newton
      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){
          // Convergência falhou, usar método da Bissecção
          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();
}
```

Também podemos querer achar os pontos de extremidade não para um círculo ou semicírculo, mas para um caminho definido como curvas de Bézier:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void path_extremity_points(struct context *cx,
                           struct path_variable *p, float *matrix);
```

Curiosamente, neste caso as fórmulas são mais simples. Uma curva de Bézier cúbica como as que usamos tem a seguinte fórmula, com t variando entre 0 e 1, com z_1 e z_4 sendo pontos de extremidade e com z_2 e z_3 sendo pontos de controle:

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2 + 3(1-t)t^2 z_3 + t^3 z_4$$

A derivada da função é:

$$z'(t) = (-3z_1 + 9z_2 - 9z_3 + 3z_4)t^2 + (6z_1 - 12z_2 + 6z_3)t + (-3z_1 + 3z_2)$$

Descobrir em quais valores isso é igual à zero requer meramente usar a fórmula de Bháskara.

Achar os pontos de extremidade então será feito iterando sobre os pontos do caminho e computando os zeros de $z'(t)$. Se acharmos um zero entre 0 e 1, checamos o valor para ver se precisamos armazená-lo. Também checamos cada um dos pontos de extremidade. Os pontos z_1, z_2, z_3, z_4 que usamos são aqueles que estão armazenados após passar pela transformação linear da matriz correspondente:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
    // Fórmula de Bháskara (eixo 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);
      }
    }
    // Fórmula de Bháskara (eixo 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. Triangulação

Vamos agora lidar com o código de triangulação. As placas de vídeo e o OpenGL trabalha com triângulos. Nós devemos então triangular um pincel antes de podermos representá-lo graficamente. Devemos representá-lo como uma lista de triângulos. Isso chegou a ser mencionado na Subseção 7.5 sobre variáveis de caneta, mas o processo de triangulação não foi definido lá.

Para recapitular, estas são as variáveis relevantes para a triangulação que estão definidas dentro de um `struct pen_variable`:

* `struct path_variable *format`: Possui um caminho cíclico com o formato da caneta.

* `GLuint gl_vbo`: Referência para os vértices armazenados na placa de vídeo. Será 0 se a caneta ainda não foi triangulada.

* `float triang_resolution`: Uma medida interna do nível de detalhamento de nossa triangulação. Relevante para saber se devemos retriangular uma caneta curva ou circular para podermos mostrar mais detalhes dela quando seu tamanho muda.

* `int flags`: Possui informação se o caminho acima define uma forma convexa ou côncava, um polígono ou uma figura com curvas, se é um quadrado, um círculo ou um polígono nulo sem lados. Além disso, nesta seção vamos definir e usar os novos valores desta flag:

Seção: Macros Locais (metafont.c) (continuação):

```
#define FLAG_ORIENTATION      64
#define FLAG_COUNTERCLOCKWISE 128
```

A segunda flag acima será usada para indicar se estamos armazenando osvértices do formato da caneta no sentido anti-horário (se estiver ativa) ou horário (se não estiver ativa). Entretanto, só vamos considerar o valor válido e inicializado se a flag anterior acima estiver ativa.

Vamos usar a seguinte função para obter tal informação de uma caneta:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool is_pen_counterclockwise(struct pen_variable *pen);
```

A função funciona checando se temos tal informação armazenada na flag e a retornando. Se não, ela busca descobrir. Ela faz isso encontrando o vértice com a menor coordenada y (e o maior x se houver empate). Seja A este vértice, P o vértice anterior e N o próximo. O sinal do resultado do produto escalar de AP e AN determina a orientação:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
  }
}
```

A função que fará a triangulação é:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool triangulate_pen(struct metafont *mf, struct context *cx,
                     struct pen_variable *pen, float *transform_matrix);
```

E a sua implementação é:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
bool triangulate_pen(struct metafont *mf, struct context *cx,
                     struct pen_variable *pen, float *transform_matrix){
                  <Seção a ser Inserida: Triangulação: Caneta Nula>
                <Seção a ser Inserida: Triangulação: Caneta Quadrada>
                <Seção a ser Inserida: Triangulação: Polígono Convexo>
              <Seção a ser Inserida: Triangulação: Círculo e Semicírculo>
              <Seção a ser Inserida: Triangulação: Forma Curva Convexa>
                  <Seção a ser Inserida: Triangulação: Forma Côncava>
}
```

O caso mais simples é se tentarmos triangular uma caneta nula, definida pelo comando `nullpen`. Essas canetas nunca serão trianguladas, pois elas não fazem desenho algum. Consideramos elas como sendo somente o ponto $(0, 0)$. Embora ela possa ser deslocada com uma matriz de

transformação. Este caso mais simples é tratado abaixo:

Seção: Triangulação: Caneta Nula:

```
if((pen -> flags & FLAG_NULL)){
  pen -> indices = 0;
  DECLARE_PEN_EXTREMITIES();
  CHECK_PEN_EXTREMITIES(0, 0, transform_matrix);
  UPDATE_PEN_EXTREMITIES();
  return true;
}
```

O próximo caso é quando temos uma caneta quadrada. Este tipo de caneta não precisa ser triangulada porque durante a inicialização nós já fazemos uma só triangulação que será usada por toda e qualquer caneta quadrada. Seus vértices serão armazenados aqui:

Seção: Variáveis Locais (metafont.c) (continuação):

```
static GLuint pensquare_vbo;
```

E o código de triangulação que será usado na inicialização:

Seção: Inicialização WeaveFont (continuação):

```
{
  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);
}
```

Na finalização podemos remover os vértices da placa de vídeo:

Seção: Finalização WeaveFont (continuação):

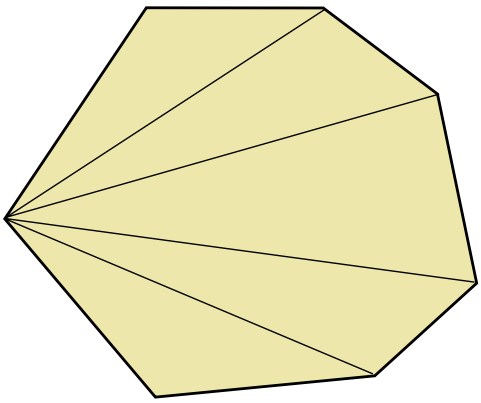
```
glDeleteBuffers(1, &pensquare_vbo);
```

Sendo assim, quando temos que triangular uma caneta quadrada, nós não a triangulamos. Mesmo assim, devemos obter os seus pontos de extremidades:

Seção: Triangulação: Caneta Quadrada:

```
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;
}
```

O próximo caso mais simples é quando temos um polígono convexo. Neste caso, podemos usar o algoritmo mais simples. Escolhemos um único vértice como pivô e fazemos uma triangulação gerando triângulos usando o pivô e cada um dos vértices adjacentes que temos. É o método de triangulação em leque:



Para isso basta passarmos os vértices da caneta na ordem anti-horária para a placa de vídeo usando o OpenGL. Depois, quando chegar a hora de desenhar, é só pedir para o OpenGL usar a triangulação em leque:

Seção: Triangulação: Polígono Convexo:

```
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;
}
```

Vamos agora passar para o próximo caso de triangulação: o círculo e também o semicírculo. Uma caneta circular é gerada com a expressão `pencircle` ou `pensemicircle` no caso de semicírculos. Para este tipo de caneta, devemos gerar seus vértices na hora para fazer a triangulação. Com saber quantos vértices gerar? Se o círculo tiver um único pixel de diâmetro, 4 vértices já seriam o suficiente para aproximá-lo e ninguém notaria estar diante de um quadrado ao invés de um círculo. Mas quanto maior ele fica, maior o número de vértices que devemos usar. O mesmo vale para o semicírculo: o número de vértices que ele precisa cai pela metade em relação ao círculo, mas quanto maior ele é, maior o número de vértices necessários.

Devemos levar em conta a fórmula do arco de circunferência $L = \theta r$. Aqui θ é um ângulo interno do círculo e r seu raio. O valor de L é o comprimento de arco definido por tais valores. Devemos então fazer com que cada arco que desenhamos tenha um tamanho de 1 pixel, fazendo com que o desenho sempre se pareça com um círculo. Para isso o ângulo interno entre cada um deles em radianos deve ser $1/r$. Como o ângulo total é 2π para círculos e π para semicírculos, então devemos representar respectivamente $2\pi r$ e πr vértices diferentes no perímetro de tais canetas.

A triangulação que usaremos será a mesma triangulação em leque anterior. Mas vamos escolher como pivô o centro do círculo. Para semicírculos também escolheremos o ponto análogo como pivô. Desta forma, como temos um pivô, vamos armazenar $2\pi r + 1$ vértices para círculos e $\pi r + 1$ para semicírculos.

Já o raio da caneta circular é escolhido como sendo metade do maior lado do quadrado de lado 1 que passa pela mesma transformação linear definida pela matriz de transformação do círculo. Como um círculo de diâmetro 1 é inscrito em um quadrado de lado 1, então este valor nos dá um limite superior adequado para representar o raio da caneta circular em cada ponto de seu perímetro, mesmo depois de uma transformação linear possivelmente transformar ela em uma elipse ou semielipse.

Esse maior número de diâmetro do quadrado no qual o círculo está inscrito é também como mediremos a resolução da triangulação de uma caneta circular. Se a caneta circular já estiver triangulada e tivermos que desenhar um com resolução menor, não é necessário triangular novamente. Mas se tivermos que desenhar um com resoluçao maior, temos que triangulá-la novamente, mesmo que ele já tenha sido triangulado.

O código para a triangulação da caneta circular é:

Seção: Triangulação: Círculo e Semicírculo:

```
if((pen -> flags & FLAG_CIRCULAR) || (pen -> flags & FLAG_SEMICIRCULAR)){
  float radius;
  GLsizei size;
  // Obter pontos de extremidade:
  pencircular_extremity_points(cx, transform_matrix,
                               (pen -> flags & FLAG_CIRCULAR));
  // Checando resolução (raio):
  {
    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;
  }
  // Só retriangula se raio for maior que resolução já triangulada:
  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; // Centro do círculo
    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;
}
```

Vamos tratar agora o último caso restante de caneta convexa: quando temos uma caneta cujo formato tem curvas, mas ela é convexa. Neste caso, assim como no círculo, devemos levar em conta seu tamanho para decidir quantos vértices devemos usar para aproximar corretamente a curva. Mas agora temos que fazer isso iterando sobre cada par adjacente de pontos de extremidade, levando em conta os pontos de controle.

Para cada par de pontos (A, D) de pontos de extremidade, com os pontos de controle (B, C), o número de vértices que geraremos será dado pela soma das distâncias AB, BC e CD. A estimativa gerará valores razoáveis, exceto em casos nos quais os pontos de controle aparecem em ordens problemáticas, nas quais a curva acabaria cruzando sobre si mesma. Como não suportamos formas não-simples para canetas, este é um caso indefinido e não nos preocuparemos com ele.

Esse cálculo de distância, somando a distância dos pontos ao iterar sobre toda a curva é também como representaremos a resolução da triangulação de uma caneta curva. Se a soma

das distâncias for menor do que a resolução da triangulação, então não precisamos triangular novamente, pois a caneta já está triangulada em um nível de detalhe aceitável.

Podemos também economizar o número de vértices quando a curva na verdade é uma linha reta entre A e D. Em tais casos, só é necessário ter vértices para os pontos de extremidade.

Para gerar os pontos, após determinar o número de vértices, podemos gerar cada um deles, modificando o valor de t na fórmula abaixo, variando entre 0 e 1 passando por um valor intermediário para cada vértice a mais a ser gerado:

$$P(t) = (1-t)^3 A + 3(1-t)^2 tB + 3(1-t)t^2 C + t^3 D$$

Naturalmente, devemos também percorrer os pontos em um sentido anti-horário, o que não necessariamente é a orientação na qual eles estão armazenados. Outra coisa a lembrar é que ao medir a distância entre os pontos, nós devemos levar em conta a transformação linear armazenada na matriz de transformação. Mas ao renderizar os vértices, isso não deve ser levado em conta, já que tal transformação será aplicada pela placa de vídeo ao renderizar a caneta.

O código para triangular uma caneta neste caso é:

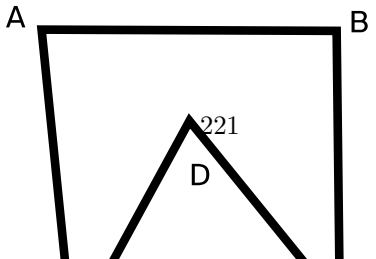
Seção: Triangulação: Forma Curva Convexa:

```
if((pen -> flags & FLAG_CONVEX)){
  bool counterclockwise = is_pen_counterclockwise(pen);
  int i, number_of_vertices = 1;
  // Obtém pontos de extremidade:
  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;
}
```

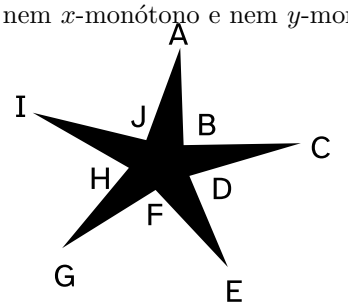
Por último, vamos à mais complexa das triangulações: quando temos uma forma côncava. Não podemos triangulá-la de maneira tão simples à partir de um pivô à partir do qual geramos todos os triângulos. Considere o polígono côncavo abaixo, por exemplo:



Se nós escolhermos o vértice E como pivô, poderíamos acabar enviando para a triangulação um triângulo EDC, que mesmo que seja composto por vértice do polígono, não representa uma região que fica dentro do polígono e portanto, não deve ser triangulado.

Para triangular polígonos côncavos, usaremos os algoritmos descritos em [DE BERG, 2000]. Primeiro vamos considerar uma caso particular que é mais simples: o polígono é x-monótono. Isso ocorre quando partindo do vértice mais à esquerda, vamos para o vértice mais à direita tanto no sentido horário, como no sentido anti-horário e tanto em um caso como em outro, as coordenadas x dos vértices em que estamos nunca diminui. Ela pode apenas ficar igual ou aumentar conforme vamos de um vértice para o outro. Por exemplo, o polígono acima é x-monótono. Mas se ele fosse rotacionado 90 graus, ele não seria mais x-monótono, mas sim y-monótono seguindo uma definição equivalente para o eixo y.

Já o polígono abaixo, não é nem x-monótono e nem y-monótono:



Para triangular um polígono x-monótono, podemos assumir que nós primeiro verificamos que ele é x-monótono percorrendo seus vértices do mais à esquerda para o mais à direita, tanto pegando o caminho de cima como o de baixo. E que no processo nós marcamos cada vértice como pertencendo ao caminho de cima e/ ou ao de baixo. O vértice inicial e o final pertencem aos dois caminhos. No processo nós também ordenamos cada vértice de acordo com a sua coordenada x. Caso dois vértices tenham a mesma coordenada x, nós desempatamos a ordenação de acordo com a coordenada y. Por exemplo, no caso do primeiro polígono côncavo que representamos nas imagens acima, a ordem de seus vértices seria (A, E, D, B, C). Onde (A, B, C) é o caminho de cima e (A, E, D, C) é o caminho de baixo.

Uma vez que os vértices estejam ordenados e marcados, podemos percorrer eles na ordem em que os deixamos. Cada vez que lemos um novo vértice, tentamos criar uma diagonal dele com os vértices anteriores, formando um novo triângulo. Quando isso não é possível, armazenamos o vértice em uma pilha. Quando encontramos um vértice de cima, quando todos os vértices em nossa pilha são do caminho de baixo, podemos criar diagonal entre todos eles, e podemos esvaziar a pilha. Se lemos um vértice do caminho de baixo, quando só há vértices do caminho de cima empilhados, o mesmo ocorre. Já se lemos um vértice do mesmo caminho, podemos tentar criar uma nova diagonal gerando um triângulo, somente se essa diagonal passa por dentro do polígono (o que é fácil de checar se sabemos se cada vértice está na parte de cima ou de baixo do polígono). Se não for possível criar a diagonal neste caso, nós deixamos os vértices acumularem na pilha e seguimos o algoritmo, sabendo que uma hora acharemos um vértice do caminho oposto que permitirá a triangulação.

Por exemplo, no caso de nosso polígono x-monótono cujos vértices ordenados são (A, E, D, B, C), podemos começar lendo (A, E, D). Como infelizmente, D e E são ambos da parte de baixo do polígono, a triangulação não é automática. Verificamos então se podemos criar uma diagonal DA, formando o triângulo DAE. Por sorte podemos, pois a diagonal DA passa por dentro do polígono (afinal, DE são da parte de baixo e o vértice E está abaixo dos dois pontos da diagonal DA). Após gerar o triângulo DAE, podemos nos livrar de E e mantemos agora em nossa pilha (A, D). Lemos o próximo vértice B. Como B é da parte de cima e D da parte de baixo, uma diagonal DB pode ser criada. Isso produz o triângulo ADB. Podemos então remover o vértice A e ficamos somente com (D, B, C), que é o triângulo final da triangulação.

Quando nossa caneta não é formada só por segmentos retos e possui curvas, podemos convertê-la antes para um polígono com um número suficientemente alto de arestas para assim poder aplicar o método. Durante a conversão podemos identificar qual o vértice mais à esquerda para poder depois testar se é um polígono x-monótono. Mas primeiro temos que definir uma estrutura para armazenar uma lista encadeada de vértices:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
#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; // Vai armazenar se é um vértice de cima ou de baixo
  float x, y;
```

```
  struct polygon_vertex *prev, *next;
      <Seção a ser Inserida: struct polygon“vertex: Variáveis Adicionais>
};
```

Para ordenar os vértices em ordem de suas coordenadas x, desempatando vértices que estão na mesma coordenada x usando a coordenada y, podemos usar a seguinte macro:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
#define XMONOTONE_LEQ(v1, v2) ((v1->x==v2->x)?(v1->y<=v2->y):(v1->x<=v2->x))
```

Caso tenhamos uma lista duplamente encadeada de vértices e queiramos remover todos os elementos dela, podemos usar a função:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void destroy_vertex_linked_list(struct polygon_vertex *poly);
```

Que é implementada abaixo:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
  }
}
```

A lista duplamente encadeada de vértices será gerada à partir de uma caneta por meio da seguinte função:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
struct polygon_vertex *polygon_from_pen(struct metafont *mf,
                                        struct pen_variable *,
                                        float *transform_matrix,
                                        int *number_of_vertices);
```

A implementação da função:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  // Obter o caminho à partir da caneta:
  struct path_variable *path = p -> format;
  if(path == NULL){
    RAISE_GENERIC_ERROR(mf, NULL, 0, ERROR_UNKNOWN);
    return NULL; // Isso não deve acontecer
  }
  // Primeiro vértice:
  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 sobre pontos de extremidade do caminho:
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){ // Em caso de linha reta, precisamos só de 1 vértice
    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{ // Em caso de curva, vamos inserir vários vértices
    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){ // Vértice anterior é desnecessário?
        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){
                  // Se for desnecessário:
          last -> x = x;
          last -> y = y;
          continue;
        }
      }
      // Criar novo vértice:
      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;
}
```

A função anterior funciona primeiro extraindo o caminho do formato da caneta e gerando o primeiro vértice à partir do primeiro ponto. Em seguida, se tivermos uma linha reta até o próximo ponto de extremidade, nós apenas adicionamos mais um vértice até aquele ponto. Caso contrário, estimamos o número de vértices necessários para aproximar a curva por meio da distância do caminho reto do primeiro ponto de extremidade até o próximo, passando pelos pontos de controle. Geramos então esta quantidade de vértices, mas toda vez que detectamos que um novo vértice, junto com os dois anteriores estão na mesma reta, sobrescrevemos o vértice anterior com o atual, já que o anterior é redundante. O número de operações que a função acima executa para inserir vértices é assintoticamente equivalente ao perímetro da caneta (em pixeis) para canetas curvas e ao número de vértices para canetas poligonais.

A função que checa se um polígono é x-convexo é declarada abaixo:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool is_xmonotone(struct polygon_vertex *poly);
```

Ela assume que um polígono é já representado como um ponteiro para seu vértice mais à esquerda. A função então tentará identificar se o polígono está no sentido horário ou anti-horário. Baseado nisso ele percorre os demais vértices ajustando sua flag para `FLAG_UPPER` ou `FLAG_LOWER`. Caso cheguemos em um vértice que viola o que se espera de um polígono x-monótono, interrompemos e retornamos falso. As flags preenchidas neste caso devem ser simplesmente ignoradas. Segue a implementação da função:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Quando triangulamos um polígono côncavo, não iremos usar as otimizações em que assumimos que todos os triângulos compartilham um mesmo ponto (o que nos permite gastar menos espaço para armazenar os triângulos) tal omo fizemos para triangular círculos, quadrados, semicírculos e polígonos convexos. Ao invés disso, cada triângulo é inteiramente representado por três coordenadas de seu vértice. Assumimos que teremos alocado um vetor com $3n$ elementos e durante a triangulação iremos preenchê-lo. E para isso, precisaremos sempre de um índice t ($0 \leq t < 3n$) para nos indicar em qual posição do vetor podemos colocar o próximo triângulo.

A função que triangula polígonos x-monótonos tem o objetivo de preencher este vetor. Ela assume que o vetor foi alocado com espaço suficiente, então não faz nenhuma checagem. Ela implementa o algoritmo sobre o polígono P que descrevemos mais informalmente acima, mas que é descrito de maneira mais formal abaixo:

1. Ordene todos os vértices do polígono em ordem crescente de acordo com sua coordenada x. Seja u_1, ..., u_n a sequência ordenada.
2. Inicialize uma pilha S e empilhe nela u_1 e u_2.
3. Faça $j \leftarrow 3$ variar até $n - 1$:
4. **Se** u_j e o vértice no topo de S estão em caminhos diferentes (cima e baixo):
5. Desempilhe todos os vértices de S.
6. Crie diagonal de u_j a cada vértice desempilhado, exceto o último.
7. Empilhe u_{j-1} e u_j.
8. **Senão**:
9. Desempilhe um vértice de S.
10. Desempilhe de S enquanto diagonal de vértice desempilhado a u_j for inscrita em P.
11. Crie diagonal u_j até cada vértice desempilhado.
12. Devolva último vértice desempilhado a S.
13. Empilhe u_j em S.
14. Adicione diagonais de u_n a todo vértice que ainda estiver na pilha.

Uma diagonal é uma aresta adicional colocada no polígono. Após preencher todas as diagonais, o polígono estará triangulado. Na prática, cada diagonal permite que adicionemos um novo triângulo e permite que removamos um vértice de nosso polígono.

A primeira etapa do algoritmo acima é ordenar os vértices de modo crescente, da menor coordenada x até a maior. Como é um polígono x-monótono, podemos fazer isso em $O(n)$ apenas percorrendo os vértices à partir do vértice inicial e usando o caminho de cima e o de baixo. Ao invés de criar uma nova estrutura para a lista encadeada ou vetor de vértices ordenados, nós apenas vamos adicionar neles um ponteiro adicional para que cada um deles aponte para o próximo vértice na lita ordenada:

Seção: struct polygon_vertex: Variáveis Adicionais:

```
  struct polygon_vertex *succ;
```

Nós inicializamos este ponteiro executando a seguinte função de ordenação de vértices:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

E finalmente a função de triangulação:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static bool triangulate_xmonotone_polygon(struct polygon_vertex *p,
                                          float **triangles,
                                          int *number_of_triangles,
                                          struct polygon_vertex **stack);
```

A função recebe um polígono p a ser triangulado, um ponteiro para onde os triângulos gerados devem ser armazenados e um ponteiro para o total de triângulos já gerados, que deve ter seu conteúdo atualizado à medida que mais triângulos são gerados. O último ponteiro é um buffer alocado que assumiremos que terá espaço suficiente para armazenar todos os vértices do polígono que quisermos. A implementação do algoritmo é:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){ // Triângulo
    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: Ordena todos os vértices do polígono:
  order_vertices_on_xmonotone_polygon(p);
  // 2: Inicializa pilha 'S' e empilha 'u_1' e 'u_2':
  stack[0] = p;
  p = p -> succ;
  stack[1] = p;
  stack_size = 2;
  // 3: Iterando 'j' de 3 até 'n-1':
  while(p != NULL && p -> succ != NULL && p -> succ -> succ != NULL){
    p = p -> succ;
    // 4: Se 'u_j' e próximo vértice estão em caminhos diferentes:
    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 e 6: Desempilha tudo de 'S' e cria diagonal de 'u_j' nos vértices:
      for(d = 1; d < stack_size; d ++){
        ADD_DIAGONAL(data, p, stack[d]);
      }
      // 7: Empilha 'u_{j-1}' e 'u_j':
      stack[0] = stack[stack_size - 1];
      stack[1] = p;
      stack_size = 2;
    } // 8: Senão (se 'u_j' e próximo vértice são do mesmo caminho):
    else{
      // 9 e 10: Desempilhe e repita enquanto criar diagonal for possível:
      int d = stack_size - 2, last_popped_vertex = stack_size - 1;
      // 11: Crie diagonal entre 'u_j' e cada vértice desempilhado:
      while(d>=0 && IS_DIAGONAL_INSIDE(stack[d], p)){
        ADD_DIAGONAL(data, p, stack[d]);
        last_popped_vertex = d;
        d --;
      }
      // 12 e 13: Empilhe de volta último vértice desempilhado e 'u_j':
      stack[last_popped_vertex + 1] = p;
      stack_size = last_popped_vertex + 2;
    }
  }
  {
    int d;
    if(p -> succ != NULL)
      p = p -> succ;
    // 14: Adicione diagonal de 'u_n' a cada vértice na pilha menos 1o e último
    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;
}
```

No código acima nós usamos a macro ADD_DIAGONAL para inserir uma diagonal. Isto é, nós geramos um triângulo estabelecendo uma nova aresta entre dois vértices anteriormente não conectados, mas que tinham um vizinho em comum. Ao fazer isso criamos um triângulo. Uma das principais tarefas desta macro é estabelecer qual é o vizinho em comum de ambos os vértices. Uma vez que isso seja feito, nós podemos gerar o triangulo e podemos remover um dos vértices do polígono:

Seção: Macros Locais (metafont.c) (continuação):

```
#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 (%d): This should not happen
(%p<-v1->%p)(%p<-v2->%p)!\n", __LINE__, v1->prev, v1->next,v2->prev,v2->next);
```

A macro que destrói vértices já foi escrita. Mas a macro que insere triângulos ainda não. A tarefa dela será inserir um triângulo no sentido anti-horário para que ele seja renderizado corretamente. Depois de fazer isso, deve-se atualizar o ponteiro do local onde inserimos o triângulo.

Seção: Macros Locais (metafont.c) (continuação):

```
#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;
```

Por fim, precisamos da macro que testa se uma diagonal está dentro de um polígono ou não, assumindo que ambos os vértices da diagonal fazem parte ou do caminho de cima ou do de baixo do polígono (se eles pertencerem a caminhos diferentes, então já saberíamos que a diagonal está dentro do polígono e não seria necessário testar). A macro que faz isso é:

Seção: Macros Locais (metafont.c) (continuação):

```
#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)))
```

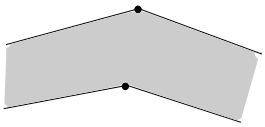
Tendo todas as macros e funções para polígonos x-monótonos, podemos começar a tratar alguns casos de polígonos côncavos. Primeiro transformamos a caneta em um polígono, baseados no número de vértices verificamos se ela precisa ser triangulada novamente, caso ela já tenha sido no passado, e em caso afirmativo, alocamos espaço para os vértices e a triangulamos. Finalmente, depois disso, usamos as funções correspondentes no OpenGL para enviar os triângulos para a placa de vídeo.

Seção: Triangulação: Forma Côncava:

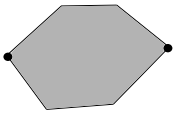
```
if(!(pen -> flags & FLAG_CONVEX)){
  struct polygon_vertex *poly;
  float *triang, *last_triang;
  int number_of_vertices = 0, number_of_triangles = 0;
  poly = polygon_from_pen(mf, pen, transform_matrix, &number_of_vertices);
  if(poly == NULL)
    return false;
  if(pen -> gl_vbo != 0){
    if(number_of_vertices <= pen -> triang_resolution){
      destroy_vertex_linked_list(poly);
      return true; // Já triangulado satisfatoriamente
    }
    else
      glDeleteBuffers(1, &(pen -> gl_vbo)); // Precisa retriangular
  }
  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{ // Esta parte trata o caso geral (não é x-monótono):
       <Seção a ser Inserida: Triangulação: Forma Não-Monótona Côncava>
  }
}
return false;
```

Mas e quando o nosso polígono não é x-monótono? Neste caso, a solução mais eficiente conhecida envolve particionar ele de modo que cada uma das partições será x-monótono. Em seguida, particionamos separadamente cada partição. Para analisar como fazer isso, precisamos catalogar os diferentes tipos de vértices que um polígono pode ter.

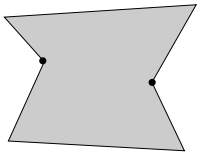
O primeiro tipo é o vértice regular: é um vértice que fica entre o anterior e o próximo no eixo x. Espera-se que a maior parte dos vértices de um polígono x-monótono seja regular (exceto pelo primeiro e último). A imagem abaixo mostra dois vértices regulares. O primeiro deles tem um ângulo interno menor que 180 graus e o mais abaixo tem um ângulo interno maior que 180 graus.



Os dois próximos tipos de vértice são o de começo e de fim. Um vértice de começo tem ambos os vizinhos mais à direita que ele, e seu ângulo interno é menor que 180 graus. Um vértice de fim tem ambos os seus vizinhos à esquerda e seu ângulo interno é menor que 180 graus. Espera-se que um polígono x-monótono tenha só um vértice de começo e um de fim. A imagem abaixo mostra ambos os tipos de vértice destacados:



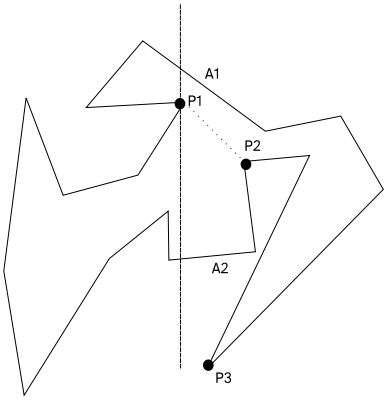
Por fim, os dois últimos tipos de vértice são o de junção e separação. Um vértice de junção tem ambos os seus vizinhos à esquerda, mas seu ângulo interno é maior que 180 graus. Enquanto um vértice de separação tem ambos os seus vizinhos à direita, mas seu ângulo interno é maior que 180 graus. A figura abaixo mostra um polígono onde seu vértice de junção (à esquerda) e de separação (à direita) estão destacados:



Os vértices de junção e de separação são a fonte de não monotonicidade. Todo polígono sem vértices deste tipo é um polígono x-monótono. Desta forma, se conseguirmos remover todos os vértices deste tipo, teremos um polígono x-monótono. Para isso podemos ligar cada vértice de junção a algum outro vértice à sua direita para particionar o polígono. E cada vértice de separação pode ser ligado à outro vértice à esquerda. Se fizermos isso sem criar uma intersecção com uma aresta prévia, conseguiremos remover todos os vértices deste tipo.

O modo pela qual ligamos um vértice de junção e separação à outro envolve percorrer os vértices ordenados mais à esquerda ou à direita, dependendo do tipo de vértice. Devemos saber também quais são as duas arestas que tem uma intersecção com a reta vertical que cruza o vértice que é fonte de não monotonicidade. Ao percorrer os vértices ordenados, devemos escolher o próximo vértice que fica entre as duas arestas. Se não houver nenhum, acabaremos escolhendo um vértice que pertence à tais arestas e fazemos a ligação, particionando o polígono criando uma nova aresta nesta região.

Por exemplo: na imagem abaixo, ao percorrer os vértices da esquerda para a direita, encontramos o vértice P1, que é uma fonte de não monotonicidade. Ele é um vértice de junção. Traçamos uma reta vertical imaginária tracejada, que mostra que este vértice está entre a aresta A1 e A2. Se houvesse outra aresta mais externa (caso o vértice P3 se estendesse mais à esquerda), ela seria ignorada. Então, percorremos quais são os próximos vértices à direita. O primeiro que encontramos é P3, que é ignorado por não estar entre as arestas A1 e A2. Depois encontramos P2 que está entre ambas as arestas e por isso escolhemos P1 e P2 como os vértices onde iremos "cortar" nosso polígono particionando-o:



Isso significa que precisamos definir algumas flags novas para representar os diferentes tipos de vértices que podemos ter:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
#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)
```

Agora devemos escrever o código que preenche os tipos de vértice e também ordena os vértices de acordo com sua coordenada x. Nós temos um ponteiro em cada vértice para ligá-los em uma lista encadeada ordenada. Mas agora, como temos um polígono não monótono, e não conseguimos gerar uma ordenação apenas adicionando novos vértices no fim da lista encadeada, precisaremos usar uma lista duplamente encadeada. Isso requer um ponteiro adicional para o antecessor de cada vértice na lista duplamente encadeada:

Seção: struct polygon_vertex: Variáveis Adicionais:

```
  struct polygon_vertex *pred;
```

Enfim, para gerarmos a nossa lista ordenada duplamente encadeada, o que faremos é primeiro percorrer todos os vértices na ordem em que ele são desenhados. Enquanto fazemos isso, podemos classificar os seus tipos e também ligá-los inicialmente de forma desordenada formando a lista duplamente encadeada. Depois disso, fazemos a ordenação usando um "merge sort". Fazendo isso, conseguimos ordenar e organizar os vértices em $O(n \log n)$.

A função que será responsável por unir duas listas duplamente encadeadas e ordenadas em uma só, mantendo a ordenação segue abaixo. Ela é o cerne do algoritmo "merge sort":

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

E o “merge sort” em si segue abaixo:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  }
}
```

Agora vamos criar em si a lista duplamente encadeada. Ao fazer isso, devemos preencher também as flags de cada vértice. O que inclui o tipo de vértice ou se é um vértice da parte inferior ou superior do polígono. Lembre-se que sabemos que o primeiro vértice não é um vértice de junção nem de separação. Então sabemos que seu ângulo interno não é maior que 180 graus. Se seguirmos o percurso deste vértice, passando pelas arestas no sentido horário ou anti horário, caso estejamos em um vértice x-monótono, então iremos apenas fazer curvas ou só para a direita, ou só para a esquerda. Assim que chegarmos em um vértice de separação ou junção, com um ângulo interno maior que 180 graus, é então que finalmente faremos uma curva para uma direção diferente da inicial. Sendo assim, podemos identificar se estamos em um vértice de junção ou separação apenas observando a direção na qual fazemos uma curva. Se todas as nossas curvas em vértices regulares e de começo e fim forem para a direita, é porque estamos percorrendo em sentido anti-horário. E se for para a esquerda, estamos em sentido horário.

Usando esta lógica podemos então criar uma iteração que percorre todos os vértices identificando seu tipo e criando alista duplamente encadeada. Depois disso, ordenamos tudo com um "merge sort". E finalmente, depois de ordenar, já saberemos qual é o vértice mais à esquerda e o mais à direita:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void prepare_non_monotonous(struct polygon_vertex *p, int number_of_vertices);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
    // Vértice de começo ou de separação:
    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{ // Vértice de fim ou de junção
      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);
}
```

Mas como podemo saber se estamos virando para a direita ou para a esquerda quando percorremos um caminho definido por três vértices? Para isso podemos representar tal caminho por dois vetores e calcular o produto vetorial e obter o componente z. Se o resultado for positivo, caminhar pelos vértices significa fazer uma virada para a esquerda. Se for negativo, estamos virando para a direita. Então, a função abaixo pode ser usada para nos dar a resposta:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static bool is_turning_left(struct polygon_vertex *p1,
                            struct polygon_vertex *p2,
                            struct polygon_vertex *p3);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
}
```

Agora volte à última imagem que mostramos de um polígono côncavo. Havíamos comentado que o modo de particionar ele, separando-o em polígono x-monótonos envolve imaginar uma reta vertical que percorre a região ocupada pelo polígono, da esquerda para a direita. Por meio desta reta, saberíamos quais as arestas que existem na mesma coordenada x de cada vértice, e assim poderíamos saber quando podemos fazer uma partição ligando dois vértices diferentes.

A nossa reta vertical será representada por duas informações diferentes: por qual vértice ela está passando neste exato momento (podemos simular ela percorrer a área do polígono da esquerda para a direita fazendo ela saltar de um vértice para o outro, seguindo a ordem dos vértices da esquerda para a direita na lista duplamente encadeada) e quais as arestas que ela está cruzando em um dado momento. Além disso, cada aresta que está abaixo de um vértice deve armazenar como informação adicional qual o seu "ajudante", isto é, o vértice à esquerda de nossa reta imaginária mais próximo dela tal que o segmento que conecta tal vértice à tal aresta fica localizado internamente ao polígono. À medida que a reta vertical imaginária percorre seu caminho da esquerda para a direita, o "ajudante" de cada aresta pode mudar. Armazenar os "ajudantes" é importante, pois eles são candidatos possíveis que devemos considerar quando vamos fazer uma nova partição.

A lista de arestas, onde cada uma potencialmente pode querer armazenar um "ajudante", é mais eficientemente representada como uma árvore binária. Ela permite que possamos buscar arestas relevantes em $O(\log n)$, o que ajuda a tratar melhor polígonos patológicos com um número muito grande de arestas que cruzam uma mesma coordenada no eixo x. Os nós na árvore são ordenados de acordo com sua posição no eixo y (usando o eixo x para desempatar). Cada nó de nossa árvore é:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
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))
```

Para inserir uma nova aresta nesta árvore, podemos usar a função abaixo. Vamos escrever a função de maneira que o usuário informe as funções que serão usadas para comparar as arestas para saber se uma é menor ou igual à outra. Desta forma, poderemos usar a função tanto para construir árvores ordenadas de acordo com o primeiro vértice, de acordo com seu ponteiro para o "ajudante", ou de acordo com qualquer outra informação que seja mais relevante. O usuário também pode passar opcionalmente uma função para comparar a igualdade de duas arestas, mas somente se ele quiser que a árvore represente um conjunto onde não há valores repetidos:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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 *));
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Mas como devemos ordenar os elementos na nossa árvore? Precisaremos de uma função que estabeleça uma relação de ordem para usar as funções acima:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static bool leq_by_vertex(struct polygon_edge *, struct polygon_edge *);
```

Para esta relação de ordem, considere que um dos casos de uso desta árvore será armazenar quais as arestas de um polígono que estão sendo cruzadas por uma reta vertical em um dado momento. E que uma busca que precisaremos fazer será determinar qual a aresta imediatamente abaixo de um dado ponto. Por causa disso, fará sentido armazenar as arestas levando em conta a coordenada y na qual elas então sendo cruzadas pela reta vertical. Como as arestas nunca cruzam entre si, e como de qualquer forma vamos remover as arestas assim que não estivermos cruzando elas, podemos então criar uma relação de ordem entre elas se baseando em quais estão acima ou abaixo umas das outras. Para isso, primeiro identificamos qual aresta começou antes (tem a menor coordenada x em um de seus vértices) e obtemos a equação da reta que estende esta aresta. Com base na equação de reta, podemos verificar se o ponto mais à esquerda da segunda aresta está abaixo ou acima da reta obtida.

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
static bool leq_by_vertex(struct polygon_edge *p1, struct polygon_edge *p2){
  if(p1 -> x1 == p1 -> x2 && p2 -> x1 == p2 -> x2){ // 2 vértices verticais
    return (p1 -> y1 <= p2 -> y1);
  }
  else if(p1 -> x1 == p1 -> x2){ // primeiro vértice vertical
    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){ // segundo vértice vertical
    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;
  }
  else if((p1 -> x1 <= p2 -> x1 && p1 -> x1 <= p2 -> x2) ||
          (p1 -> x2 <= p2 -> x1 && p1 -> x2 <= p2 -> x2)){ //p1 vem antes de p2
    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 vem antes do p1
    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);
  }
}
```

Já o teste de igualdade entre arestas é mais simples. Basta comparar os vértices, mas lembrar que a ordem deles não é relevante. Pois a aresta $(0,1) - -(1,0)$ é igual a $(1,0) - -(0,1)$:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static bool eq_by_vertex(struct polygon_edge *, struct polygon_edge *);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
}
```

Para remover uma aresta da árvore, usamos o seguinte código:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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 *));
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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; // Nó a ser removido não tem filhos
       return current;
     }
     else if(current -> right == NULL){
       *tree = current -> left; // Nó a ser removido tem só 1 filho esquerdo
       current -> left -> parent = current -> parent;
       return current;
     }
     else if(current -> left == NULL){
       *tree = current -> right; // Nó a ser removido tem só 1 filho direito
       current -> right -> parent = current -> parent;
       return current;
     }
     else{ // Remoção quando alvo tem 2 filhos
       // suc: sucessor do nodo removido
       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{ // Ainda não achou: procurar alvo nos filhos:
    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;
    }
  }
}
```

Finalmente, vamos precisar de uma função de busca para encontrar o vértice que está imediatamente abaixo de uma coordenada (x, y). Para isso, usamos a função abaixo:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static struct polygon_edge *find_edge_below(struct polygon_edge *,
                                            float, float);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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){ // Esta aresta é válida. Tem uma maior?
    struct polygon_edge *candidate;
    candidate = find_edge_below(tree -> right, x, y);
    if(candidate != NULL)
      return candidate;
    else return tree;
  }
  // Esta aresta não é válida, precisamos de uma menor:
  else return find_edge_below(tree -> left, x, y);
}
```

Devemos também sermos capazes de particionar um polígono, passando dois ponteiros para seus vértices como argumento. Tais vértices deverão então ser unidos e o polígono será dividido em dois. Para isso, devemos clonar ambos os vértices, passando a ter duas cópias de cada. Ambas as cópias serão conectadas, cada um de uma forma diferente. Os vértices vizinhos serão atualizados de maneira consistente. Usamos os dois últimos ponteiros recebidos como argumento para armazenar os dois novos polígonos gerado à partir da partição:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static bool cut_polygon(struct polygon_vertex *v1, struct polygon_vertex *v2,
                        struct polygon_vertex **new1,
                        struct polygon_vertex **new2);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
  // Remover flags:
  v1 -> flag &= (~0x3);
  v2 -> flag &= (~0x3);
  // Separar:
  memcpy(vv1, v1, sizeof(struct polygon_vertex));
  memcpy(vv2, v2, sizeof(struct polygon_vertex));
  // Há duas formas de separar. Discutimos logo abaixo o por quê
```

```
  // de termos que fazer este teste:
  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;
  }
  // Achar ponteiro do novo polígono:
  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;
}
```

No código acima há uma parte que talvez seja misteriosa. O por quê de tomarmos um cuidado testando a direção na qual fazemos uma curva ao acompanhar os vértices onde fazemos um corte e o por quê disso influenciar a forma pela qual cortamos. Considere que se estamos fazendo um corte no polígono, pelo menos um dos vértices que estamos conectando tem um ângulo interno maior que 180 graus. Se ele tiver um ângulo interno menor, então isso não mudará depois do corte. Já se o ângulo interno for maior, então após o corte, ou todos os ângulos se tornam menores que 180 graus, ou então um dos polígonos gerados continua tendo um ângulo interno maior. Se for o caso, mais cortes deverão ser feitos naquele vértice até reduzir o seu ângulo o suficiente. Mas possivelmente teremos identificado previamente quais os cortes que devemos fazer e teremos armazenado ponteiros para os vértices em que cortes estão para serem feitos. Desta forma, é importante que na função acima, ao fazer um corte, não modifiquemos o ponteiro de qualquer vértice que ainda precisará de mais cortes futuros. Se após o corte ainda houverem ângulos internos muitos grandes, eles devem ser associados ao mesmo ponteiro `v1` e `v2` que tínhamos, não aos novos ponteiros `vv1` e `vv2`. E o modo pelo qual verificamos isso é observando se a direção de cada vértice mudou (o que significa que um ângulo grande se tornou um pequeno).

Agora vamos juntar tudo isso para criar a triangulação para o caso geral, quando não temos um polígono x-monótono. A triangulação irá primeiro ordenar os vértices criando a lista duplamente

encadeada entre eles. Depois irá inicializar uma árvore vazia para representar a reta imaginária que nos ajudará na partição. Também usaremos uma lista encadeada para armazenar os vértices nos quais iremos fazer cortes que particionarão nosso polígono. Depois da inicialização, percorreremos todo o polígono com nossa reta imaginária da esquerda para a direita e armazenaremos na lista todos os cortes que iremos fazer. Só depois nós iremos fazer os cortes fazendo o particionamento e depois disso vamos triangular cada novo polígono obtido.

Para não precisar criar novas estruturas de dados, a nossa lista de diagonais (representando cortes a serem feitos para particionar o polígono) será representada por um vértice para cada diagonal. Os ponteiros `next` e `prev` irão apontar para os vértices que deverão ser unidos no corte. O ponteiro `succ` irá conectar os cortes formando uma lista encadeada simples. Assumindo que temos um ponteiro `list_of_diagonals` que aponta para esta lista encadeada e `last_diagonal` que aponta para a última diagonal inserida, podemos adicionar diagonais com a macro abaixo:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}
```

À medida que realizamos todos os cortes, vamos inserindo em uma árvore os novos polígonos. Podemos usar a mesma estrutura da árvore que criamos para armazenar a reta imaginária que percorre o polígono. Mas desta vez nossa árvore irá representar um conjunto de novos polígonos gerados. Somente o atributo "helper" será relevante: ele irá conter o ponteiro para o polígono. Devemos inserir e ordenar na árvore usando somente este atributo. Para isso, vamos definir as seguintes funções de comparação:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
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);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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);
}
```

E finalmente o código que realiza as operações descritas para triangular polígonos não monótonos e côncavos:

Seção: Triangulação: Forma Não-Monótona Côncava:

```
{
  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 ++){
    // Aqui preenchemos a lista de cortes a serem feitos
        <Seção a ser Inserida: Torna X-Monótono: Adicionando Diagonal>
    current_vertex = current_vertex -> succ;
  }
  current_vertex = list_of_diagonals;
  // Realiza a partição:
  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;
}
```

Quando tivermos terminado todas as partições e todos os novos polígonos gerados estiverem armazenados em uma estrutura de árvore binária, então podemos triangular todos eles percorrendo a árvore, e desalocando todos os elementos à medida que fazemos isso:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static void triangulate_polygon_tree(struct polygon_edge *tree,
                                     float **triangles,
                                     int *number_of_triangles,
                                     struct polygon_vertex **buffer);
```

```
void print_tree(struct polygon_edge *tree){
  printf(" (%f %f)--(%f %f)\n", tree -> x1, tree -> y1, tree -> x2, tree -> y2);
  if(tree -> left != NULL)
    print_tree(tree -> left);
  else printf("left nil\n");
  if(tree -> right != NULL)
    print_tree(tree -> right);
  else printf("right nil\n");
}
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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;
}
```

Agora tudo o que resta fazer é adicionar diagonais para podermos particionar o polígono. Para isso, iremos mover a nossa reta vertical imaginária fazendo ela passar por cada vértice da esquerda para a direita. O que iremos fazer em cada caso, depende do tipo de vértice que temos. Se for um vértice de começo, devemos registrar que a aresta de baixo ligada à ele está sendo cruzada por nossa reta (armazenamos ela na árvore), e fazemos com que o ponteiro "ajudante" desta aresta seja o vértice de começo atual:

Seção: Torna X-Monótono: Adicionando Diagonal:

```
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);
  }
}
```

Se encontrarmos um vértice de fim, então verificamos se a aresta superior conectada à ele tem como "ajudante" um vértice de junção. Se for o caso, criamos uma nova diagonal entre o vértice

atual e o vértice de junção encontrado. De qualquer forma, a nossa reta imaginária deixará de cruzar a aresta inferior à este vértice e devemos removê-la da árvore.

Seção: Torna X-Monótono: Adicionando Diagonal (continuação):

```
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);
}
```

Se encontramos um vértice de separação, procuramos qual é a aresta que está imediatamente abaixo deste vértice fazendo uma busca na nossa árvore de arestas. Criamos então uma diagonal de corte conectando o vértice atual até o "ajudante" da aresta que encontramos. O vértice atual se torna então o novo "ajudante" da aresta que encontramos. Em seguida, adicionamos a aresta mais abaixo que se conecta ao vértice atual, pois à partir de agora nossa reta vertical imaginária irá passar por ela:

Seção: Torna X-Monótono: Adicionando Diagonal (continuação):

```
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);
  }
}
```

Se encontramos um vértice de junção, primeiro obtemos a aresta superior conectada à ele e checamos se ela tem como "ajudante" um outro vértice de junção. Se for o caso, criamos uma

diagonal entre o vértice atual e este "ajudante". Independente disso, apagamos tal vértice da árvore, pois nossa reta imaginária não estará mais sobre ele. Procuramos então outra aresta na árvore que esteja imediatamente abaixo do vértice atual. Se o "ajudante" desta aresta também for um vértice de junção, criamos também uma diagonal entre ele e o vértice atual. E atualizamos o "ajudante" da aresta encontrada para ser o vértice atual.

Seção: Torna X-Monótono: Adicionando Diagonal (continuação):

```
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;
}
```

E o último caso é o tratamento de vértices regulares. A nossa ação vai depender se o interior do polígono está acima ou abaixo do vértice. Verificamos isso checando a posição dos vértices vizinhos na coordenada x e também a informação se os vértices estão no sentido horário ou anti-horário.

Se o interior do polígono estiver acima do vértice, checamos se a aresta anterior tem como "ajudante" um vértice de junção. Se for o caso, criamos uma diagonal de corte até ele. De qualquer forma, removemos da árvore a aresta anterior e colocamos a nova aresta conectada ao vértice atual.

Se o interior do polígono estiver abaixo do vértice, devemos procurar a aresta imediatamente abaixo do vértice atual. Se seu "ajudante" for vértice de junção, conectamos ele ao vértice atual criando nova diagonal. Independente de termos feito isso, em seguida marcamos o "ajudante" deste vértice para ser o vértice atual.

Seção: Torna X-Monótono: Adicionando Diagonal (continuação):

```
else{
  struct polygon_edge *removed;
  if((clockwise && // Checa se interior está 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{ // Interior do polígono está abaixo do vértice:
    struct polygon_edge *below;
    below = find_edge_below(imaginary_line, current_vertex -> x,
                            current_vertex -> y);
    if(below == NULL){
      print_tree(imaginary_line);
    }
    if(GET_VERTEX_TYPE(below -> helper) == TYPE_MERGE_VERTEX){
      ADD_CUT(current_vertex, below -> helper);
    }
    below -> helper = current_vertex;
  }
}
```

11.3. Interpretando o Comando 'pickup'

Agora vamos ao que efetivamente ocorre quando lemos o comando `pickup`.

1) Primeiro lemos o próximo token. Se for `nullpen`, fazemos com que `currentpen` se torne uma caneta nula, se for um `pencircle` se torna uma caneta circular e se for um `pensemicircle`, aí será a caneta semicircular. Se for uma variável de caneta, fazemos com que ela aponte para essa variável. Em ambos os casos, reiniciamos a matriz de transformação de `currentpen` para a de uma identidade. Nos demais casos, ou se a variável não estiver inicializada, nós retornamos um erro.

Seção: Instrução: Comando:

```
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);
```

<Seção a ser Inserida: Comando 'pickup': Continuação>

```
  return true;
}
```

Depois de saber a forma base da caneta, é hora de ler as transformações lineares específicas que vamos aplicar à `currentpen`. Enquanto não estivermos no fim da expressão a ser interpretada, lemos o próximo token. Ele indicará qual transformação linear devemos aplicar à matriz de transformação de `nullpen`. Baseado nele, lemos uma expressão numérica, de par ou de transformação que descreve a transformação em questão. Fazemos isso até chegar ao fim da expressão, quando não houver mais transformações a serem feitas.

Seção: Comando 'pickup': Continuação:

```
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;
}
```

Depois de termos obtido tanto o formato da caneta nova, se ela é um ponteiro e qual a sua transformação linear, enfim iremos à etapa de triangular a caneta. Para isso, se a caneta atual ter sido ajustada para um ponteiro, sua transformação é a multiplicação da matriz armazenada na `currentpen` pela matriz da caneta para a qual ela aponta. Ou, se não for um ponteiro, é apenas a matriz armazenada em `currentpen`. Após obter a matriz final de transformação, a passamos para a função que fará a triangulação:

Seção: Comando 'pickup': Continuação (continuação):

```
{
  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. Os Operadores `bot`, `top`, `lft`, `rt`

Embora tenhamos definido quase todos os operadores de pares na Subseção 8.2, existem quatro operadores primários que deixamos para definir aqui. A gramática deles é:

```
<Primário de Par> -> bot <Primário de Par> | top <Primário de Par> |
                     lft <Primário de Par> | rt <Primário de Par>
```

Isso requer adicionar 4 novos tokens para tais operadores:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_BOT,  // O token simbólico 'bot'
TYPE_TOP,  // O token simbólico 'top'
TYPE_LFT,  // O token simbólico 'lft'
TYPE_RT,   // O token simbólico 'rt'
```

E também adicionar o nome dos tokens como palavras reservadas:

Seção: Lista de Palavras Reservadas (continuação):

```
"bot", "top", "lft", "rt",
```

O que estes operadores fazem é deslocar um par uma distância que depende do tamanho da caneta atual que estamos usando. Se temos um ponto (x_0, y_0), então `bot (x0, y0)` representa (x_0, y_0) deslocado para baixo, de modo que fique embaixo da caneta quando ela é centralizada no ponto (x_0, y_0). Assumindo que uma caneta padrão é centralizada na origem, o menor ponto y dela será um valor negativo. Então isso é obtido somando o ponto inicial com o ponto mais abaixo da caneta:

Seção: Primário de Par: Outras Regras a Definir Depois (continuação):

```
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;
}
```

O operador `top` desloca o par para cima de modo que `top (x0, y0)` passa a estar posicionado acima da caneta se esta for posicionada em (x_0, y_o).

Seção: Primário de Par: Outras Regras a Definir Depois (continuação):

```
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;
```

```
}
```

O operador `lft` desloca o ponto à esquerda uma quantidade igual à menor coordenada x da caneta. Assim, `lft (x0, y0)` passa a estar posicionado à esquerda da caneta se esta for posicionada em (x_0, y_o). Como em uma caneta centrada na origem temos que seu ponto mais à esquerda é negativo, basta somarmos o valor x_0 com o menor valor no eixo x do perímetro da caneta:

Seção: Primário de Par: Outras Regras a Definir Depois (continuação):

```
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;
}
```

Por fim, o operador `rt` desloca o par (x_0, y_0) para a direita uma quantidade equivalente à maior coordenada x da caneta atual, de modo que se as coordenadas do ponto ficarão à direita de uma caneta centralizada em (x_0, y_0):

Seção: Primário de Par: Outras Regras a Definir Depois (continuação):

```
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. O Comando `pickcolor`

A cor padrão usada por WeaveFont é definida pelas variáveis globais `color_r`, `color_g`, `color_b` e `color_a`. Entretanto, quando estamos renderizando um único glifo, podemos mudar a cor usada em nossos desenhos temporariamente. Isso porque a nossa estrutura de contexto `struct context` também possui sua própria cópia dos valores destas variáveis:

Seção: Atributos (struct context) (continuação):

```
  float color[4];
```

Quando a estrutura de contexto é criada, o valor das cores é copiado da variável global:

Seção: Inicialização (struct context) (continuação):

```
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;
```

Mas os valores destas cores pode ser mudado dentro do contexto. Para isso podemos usar o comando `pickcolor` cuaj gramática é descrita abaixo:

```
<Comando> -> <Comando 'pickcolor'> | ...
<Comando 'pickcolor'> -> pickcolor ( <Expressão Numérica> ,
                                     <Expressão Numérica> ,
                                     <Expressão Numérica> ,
                                     <Expressão Numérica> )
```

O qual requer uma palavra reservada nova:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_PICKCOLOR,   // O token simbólico 'pickcolor'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"pickcolor",
```

A interpretação do comando `pickcolor` é intuitiva: cada expressão numérica representa um valor no intervalo fechado entre 0 e 1 (arredondado para o valor mais próximo do intervalo se não estiver nele) e que é atribuido respectivamente aos canais vermelho, verde, azul e alfa (transparência).

Seção: Instrução: Comando (continuação):

```
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. O Comando `monowidth`

Caso uma fonte tipográfica seja configurada para ter uma versão monoespaço, onde todos os caracteres devem ter exatamente a mesma largura, será útil usar o comando `monowidth` a ser definido aqui para escolher qual é esta largura. Se durante a renderização de um glifo a variável numérica interna `monospace` for maior que zero, então a largura do caractere será calculado de acordo com a expressão numérica definida pelo comando `monowidth`, ao invés de usar a expressão particular daquele glifo específico.

A sintaxe para o comando `monowidth` é:

```
<Comando> -> ... | <Comando 'monowidth'> | ..
<Comando 'monowidth'> -> monowidth <Expressão Numérica>
```

O que requer um novo token e palavra reservada:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_MONOWIDTH,   // O token simbólico 'monowidth'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"monowidth",
```

A expressão numérica associada ao comando `monowidth` não é executada no momento em que é encontrada. Ao invés disso, o ponteiro para o início e fim de tal expressão é apenar armazenado pelos seguintes ponteiros na estrutura da fonte tipográfica:

Seção: Atributos (struct metafont) (continuação):

```
void *mono_expr_begin, *mono_expr_end;
```

Os ponteiros são inicializados como nulos:

Seção: Inicialização (struct metafont) (continuação):

```
mf -> mono_expr_begin = mf -> mono_expr_end = NULL;
```

Interpretar o comando então não envolve interpretar a expressão numérica, mas armazená-la para ser interpretada na hora de preparar um novo glifo para renderizar:

Seção: Instrução: Comando (continuação):

```
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. O Comando `draw` e `erase`

Finalmente podemos começar a especificar dois dos mais importantes comandos da linguagem. O comando que usa uma caneta para caminhos em uma imagem ou para apagar eles. A sintaxe do comando de desenho e de apagar é:

```
<Comando> -> <Comando 'draw'> | <Comando 'erase'> | ...
<Comando 'draw'> -> draw <Expressão de Caminho>
<Comando 'erase'> -> erase <Expressão de Caminho>
```

Os quais requerem dois tokens e palavras reservada novas:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_DRAW,   // O token simbólico 'draw'
TYPE_ERASE,  // O token simbólico 'erase'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"draw", "erase",
```

14.1. Preparando o Framebuffer

Para desenhar ou apagar com uma caneta na imagem `currentpicture`, nós precisamos de um framebuffer associado com esta imagem. Isso não é muito diferente do que quando definimos as operações sobre variáveis do tipo imagem. Em boa parte das operações, nós criamos um framebuffer novo, salvamos o framebuffer anterior, mudamos para o novo, renderizamos, apagamos o framebuffer novo e voltamos ao antigo. Para a `currentpicture`, ao invés disso, vamos manter uma variável armazenando seu framebuffer para não termos que criá-lo e destruí-lo o tempo todo:

Seção: Atributos (struct context) (continuação):

```
GLuint currentpicture_fb;
```

A qual é inicializada como zero:

Seção: Inicialização (struct context) (continuação):

```
cx -> currentpicture_fb = 0;
```

Antes de renderizar para `currentpicture`, devemos executar o código abaixo que checa se o framebuffer está inicializado, e o inicializa se não estiver:

Seção: Prepara 'currentpicture' para Desenho:

```
{
  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);
}
```

Se por algum motivo nós mudarmos a nossa `currentpicture` para outra imagem, devemos

remover o framebuffer anterior:

Seção: Gera nova 'currentpicture':

```
{
  if(cx -> currentpicture_fb != 0){
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glBindTexture(GL_TEXTURE_2D, 0);
    glDeleteFramebuffers(1, &(cx -> currentpicture_fb));
  }
  cx -> currentpicture_fb = 0;
}
```

Antes de renderizar para nossa `currentpicture`, nós não armazenamos o framebuffer anterior em uma variável. E depois de renderizar, nós não restauramos o framebuffer anterior. Ao contrário do que fazíamos em outras operações de imagens. Isso ocorre porque enquanto estivermos interpretando nosso código, renderizar na `currentpicture` é uma operação muito mais comum que renderizar em outros lugares. Por causa disso, vamos sempre preferencialmente manter o nosso framebuffer atual para o da `currentpicture`.

Antes de começar a avaliar código, é quando nós salvamos o framebuffer anterior. Para isso usamos a variável abaixo:

Seção: Variáveis Locais (metafont.c) (continuação):

```
static GLint previous_fb;
```

E abaixo temos onde salvamos o framebuffer anterior antes de começar a executar qualquer código em `eval_list_of_expressions`. Salvamos também as dimensões do "viewport", já que podemos mudar tais valores ao renderizar uma imagem:

Seção: Antes de Avaliar Código:

```
GLint _viewport[4];
glGetIntegerv(GL_VIEWPORT, _viewport);
glGetIntegerv(GL_DRAW_FRAMEBUFFER_BINDING, &previous_fb);
```

E aqui nós restauramos o framebuffer e o "viewport" de antes de começarmos a interpretar código:

Seção: Depois de Avaliar Código:

```
glBindFramebuffer(GL_FRAMEBUFFER, previous_fb);
glViewport(_viewport[0], _viewport[1], _viewport[2], _viewport[3]);
```

14.2. Shaders de Desenho

Desenhar e apagar com uma caneta vai exigir dois novos shaders (um para cada operação) que precisamos definir. Os shaders que definimos anteriormente era para fazermos operações envolvendo variáveis de imagem, e eles exigiam que enviássemos uma textura. Já o shader para desenhar e apagar não requer uma textura, ao invés disso ele requer a cor que devemos usar para desenhar. Por hora vamos deixar que a cor sempre seja preto opaco, mas isso pode mudar no futuro.

O código de nosso shader de vértice será então:

Seção: Variáveis Locais (metafont.c) (continuação):

```
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";
```

Ele é idêntico ao que já definimos, mas não precisará receber e extrair coordenadas de textura. Já os dois novos shaders de fragmento:

Seção: Variáveis Locais (metafont.c) (continuação):

```
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; // O programa após compilar
static GLint pen_uniform_matrix, pen_erase_uniform_matrix; // Matriz
static GLint pen_uniform_color, pen_erase_uniform_color; // Cor
```

Na inicialização nós compilamos estes dois shaders e obtemos a localização de suas variáveis uniformes:

Seção: Inicialização WeaveFont (continuação):

```
{
  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");
}
```

E na finalização nós destruimos estes programas:

Seção: Finalização WeaveFont (continuação):

```
glDeleteProgram(pen_program);
glDeleteProgram(pen_erase_program);
```

14.3. Desenhando Caminhos

Terminada a preparação da triangulação, do framebuffer e do shader usado, podemos escrever agora códigos que fazem o desenho.

Seção: Instrução: Comando:

```
else if(begin -> type == TYPE_DRAW){
  struct path_variable path;
  // Avaliar a expressão 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 expressão 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;
}
```

O código acima detecta um comando `draw` e obtém o caminho a ser desenhado. A função que efetivamente realiza o desenho é:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
bool drawing_commands(struct metafont *mf, struct context *cx,
                      struct path_variable *path, unsigned int flags);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
#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;
  // Preparar a caneta, a imagem de destino e parâmetros OpenGL
  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));
        <Seção a ser Inserida: Prepara 'currentpicture' para Desenho>
  // Loop de desenho
  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; // Multiplicador obtido experimentalmente para evitar saltos
    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);
    }
  }
  // Quando é um único ponto e o loop acima não foi executado:
  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;
}
```

O código acima primeiro obtém as variáveis relevantes ao comando (`currentpen`, `currentpicture`).

Em seguida, obtemos qual caneta que devemos usar, a triangulamos e inicializamos os parâmetros OpenGL específicos deste tipo de desenho, que definimos nas Subseções anteriores. Um dos dois únicos lugares em que tratamos de forma diferente o `draw` e `erase` aparece nesta parte. Dependendo do comando, nós mudamos a operação de como misturamos os pixels novos desenhados com os pixels já existentes na imagem de destino. As escolhas feitas são análogas às que fizemos quando definimos a soma e subtração de imagens.

Depois temos o loop no qual iremos desenhar ponto-a-ponto o nosso caminho usando a função que ainda será definida `drawpoint`. Para obter casa um dos pontos, usamos a fórmula das curvas de Beziér cúbicas $z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2' + 3(1-t)t^2 z_3' + t^3 z_4$. E para saber quantos pontos intermediários devem ser desenhados entre dois pontos de extremidade, nós somamos a distância em pixels de todos os pontos envolvidos em cada curva: os dois pontos de extremidade e os de controle.

Resta apenas definir a função `drawpoint`. O seu cabeçalho é:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void drawpoint(struct context *cx,
               struct pen_variable *pen, struct picture_variable *pic,
               float x, float y, float *matrix, bool erasing);
```

A caneta possui todas as coordenadas de seus vértices triangulados usando o número de pixels como coordenada. A imagem tem sua altura e largura conhecida. Ao renderizar usando OpenGL, devemos usar as coordenadas padrão, onde o centro da imagem é a origem e a imagem tem sempre 2 de altura e de largura. Isso significa que devemos fazer uma última transformação linear para converter a caneta na renderização.

Seja h a altura da imagem em pixels e w sua largura. Isso significa que 1 pixel horizontal é $2/w$ e um vertical é $2/h$. Então, basta multiplicarmos a matriz de transformação pela matriz diagonal com $2/w$ como primeiro valor, $2/h$ como segundo valor na diagonal e com o resto da diagonal sendo 1.

O próximo passo seria depois de converter de pixels para coordenadas OpenGL, ajustar a matriz de transformação para deslocar cada vértice da caneta x e y pixels, o que daria $2x/w$

e $2y/h$ nas coordenadas OpenGL. E aí fazemos outro deslocamento para corrigirmos o fato do OpenGL tratar o centro da imagem como origem, enquanto nós tratamos o canto inferior esquerdo da imagem como origem.

Contudo, existe uma última complicação: na verdade nem sempre tratamos o canto inferior esquerdo da imagem como origem. Existe um tipo de valor interno d que pode mudar verticalmente a posição do eixo x que usamos para medir nossas coordenadas. Um valor interno de profundidade. Ele é armazenado aqui:

Seção: Atributos (struct context) (continuação):

```
int current_depth;
```

E seu valor inicial é zero:

Seção: Inicialização (struct context) (continuação):

```
cx -> current_depth = 0;
```

Este valor d mostra o quão deslocado para cima está o eixo x em quando medimos em pixels (e ele vira $2d/h$ ao ser convertido nas coordenadas OpenGL). Se ele é zero, então o eixo x será o limite inferior do espaço da imagem e a origem o canto inferior esquerdo da imagem. Se for um valor positivo, ele estará deslocado d pixels para cima.

Para fazer todos este ajustes, devemos então multiplicar a matriz de transformação da caneta pela seguinte nova matriz:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix} \begin{bmatrix} 2/w & 0 & 0 \\ 0 & 2/h & 0 \\ 2x/w-1 & 2d/h+2y/w-1 & 1 \end{bmatrix} = \begin{bmatrix} 2a/w & 2b/h & 0 \\ 2c/w & 2d/h & 0 \\ (2e+2x)/w-1 & (2f+2d+2y)/h-1 & 1 \end{bmatrix}$$

E isso é feito internamente pela função `drawpoint` imediatamente antes de desenhar. Depois de ajustar a matriz, só temos que renderizar o resultado, e aqui também devemos usar programas de shaders diferentes se estamos desenhando ou apagando:

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
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 triangulação padrão de quadrado.
  // Se não for, usamos a triangulação da própria 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. Declaração Composta: Declaração de Caractere

Vamos agora à parte que é o prinicpal objetivo da linguagem: definir novos caracteres para fontes tipográficas, ou então definir uma imagem que será usada como ilustração, ou animação.

A sintaxe para isso é:

```
<Composta> -> <Declaração de Caractere>
<Declaração de Caractere> -> beginchar ( <Token de String> ,
                                         <Expressão Numérica> ,
                                         <Expressão Numérica> ,
                                         <Expressão Numérica> )
                             <Corpo do 'beginchar'>
                             endchar
<Corpo do 'beginchar'> -> <Declaração, exceto 'beginchar'>
<Declaração, exceto 'beginchar'> -> <Simples> | <Composta, exceto 'beginchar'>
<Composta, exceto 'beginchar'> -> <Bloco Composto> | <Bloco Condicional>
```

O token `beginchar` começa a definição de um novo caractere e o token `endchar` termina. O token de string é o nome do caractere definido. Para fontes tipográficas, deve ser a representação em UTF-8 do caractere. Para animações e ilustrações, pode ser qualquer nome, o token terminará sendo ignorado. Os valores numéricos após o token de string representam respectivamente a largura, altura e profundidade do caractere (profundidade sendo a altura dele abaixo da linha de base, para caracteres como "p" com partes dele ficando abaixo da linha).

A linguagem WeaveFont tem dois modos de operação: ela pode estar carregando ou executando. O modo determina o comportamento dela ao encontrar o token `beginchar`. No modo de carregamento, ela copia o código que representa o caractere para uma posição adequada, para que se necessário, o código seja executado para renderizar o caractere. O código dentro do corpo de `beginchar` nem chega a ser executado e interpretado. No modo de execução, aí sim este código será executado para que o caracteres seja renderizado.

15.1 Unicode e UTF-8

Mas onde devemos armazenar o código de um caractere quando estamos no modo de carregamento? Vamos definir a seguinte estrutura que armazena o código de um caractere. Ela armazena o código, as dimensões do caractere, a textura OpenGL onde ele foi renderizado e um atributo booleano que diz se podemos usar essa textura ou se devemos renderizar novamente o caractere:

Seção: Declarações Gerais (metafont.h):

```
struct _glyph;
```

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
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; \
                             }
```

O "kerning" que é armazenado será uma lista encadeada listando quanto de espaço adicional deve ser colocado antes do próximo caractere baseado em qual é o próximo caractere.

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
struct kerning{
  char next_char[5];
  float kern;
  struct kerning *next;
};
```

A lista de todos os glifos possíveis ficará na estrutura global da linguagem, assim como um ponteiro para o primeiro glifo definido e sua representação em string:

Seção: Atributos (struct metafont) (continuação):

```
struct _glyph *glyphs[332];
struct _glyph *first_glyph;
char first_glyph_symbol[5];
int number_of_glyphs;
```

O número 332 comporta a quantidade de diferentes blocos Unicode existentes (327) mais algumas regiões adicionais que não são usadas pelo Unicode no momento, mas podem vir a ser no futuro. Cada bloco potencialmente armazena de 1 a milhares de diferentes glifos relacionados entre si, tipicamente pertencentes a um mesmo sistema de escrita. Inicialmente, todos estes blocos serão inicializados como vazios. Mas à medida que encontrarmos caracteres pertencentes à eles, teremos que alocar cada um deles:

Seção: Inicialização (struct metafont) (continuação):

```
memset(mf -> glyphs, 0, sizeof(struct _glyph *) * 332);
mf -> first_glyph = NULL;
memset(mf -> first_glyph_symbol, 0, 5);
mf -> number_of_glyphs = 0;
```

A função que irá alocar e retornar a estrutura de um novo glifo, ou de um glifo existente é:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
static struct _glyph *get_glyph(struct metafont *mf, unsigned char *utf8,
                                bool create_if_not_exist);
```

A função começa convertendo a representação UTF-8 para UTF-32, tornando-a idêntica ao seu número Unicode. Em seguida, consultamos uma tabela como maior valor existente em cada um dos blocos Unicode para saber a qual bloco o caractere pertence, qual a sua posição dentro do bloco e também o tamanho do bloco. Se o bloco não existir e o último parâmetro da função for

verdadeiro, alocamos o bloco. No fim, retornamos o glifo representando o caractere (que pode ter sido inicializado ou não). Ou NULL se não forpossível fazer isso (o bloco não existe e foi pedido para não alocá-lo, o símbolo UTF-8 é inválido, o código Unicode não existe ou não é suportado ou não há memória suficiente para alocar o bloco).

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
static const uint32_t greatest_point[332] = {
  // Plano Multilingual Básico (164 blocos)
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,
  // Plano Multilingual Suplementar (151 blocos)
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,
  // Não Usado
  0x1ffff,
  // Plano Ideográfico Suplementar
  0x2a6df, 0x2b73f, 0x2b81f, 0x2ceaf, 0x2ebef, 0x2fa1f,
  // Não Usado
  0x2ffff,
  // Plano Ideográfico Terciário
  0x3134f, 0x323af,
  // Não Usado
```

```
  0xdffff,
  // Plano de Propósito Geral (a região do meio é não usada)
  0xe007f, 0xe00ff, 0xe01ef,
  // Não Usado
  0xeffff,
  // Área de Uso Privado Suplementar A
  0xfffff,
  // Área de Uso Privado Suplementar 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; // String UTF-8 inválida
  if(code_point > greatest_point[331])
    return NULL; // Código Unicode não-existente ou não-suportado
  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; // Glifo alocado, mas não existente
  return &(mf -> glyphs[block][index]);
}
```

Note que umavez que glifos podem ser alocados, isso significa que na finalização, teremos que desalocar eles. Também temos que remover a lista encadeada de kerning quando ela existir:

Seção: Finalização (struct metafont) (continuação):

```
{
  int block, block_size, index;
  // Percorre todos os blocos:
  for(block = 0; block < 332; block ++){
    if(mf -> glyphs[block] != NULL){
      // Obtém tamanho do bloco
      if(block == 0)
        block_size = greatest_point[block] + 1;
      else
        block_size = greatest_point[block] - greatest_point[block - 1];
      // Percorre todos os glifos:
      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]);
    }
  }
}
```

O comportamento da linguagem, quando está em modo de carregamento e lê o token `beginchar` será ler o token de string e interpretá-lo como um caractere codificado em UTF-8. Baseado nele, a função acima é executada para obter a estrutura do glifo correspondente, e alocá-la se necessário. Se já existir um glifo totalmente inicializado para tal caractere, um erro será gerado: isso significa que o mesmo caractere foi definido mais de uma vez. Se não, todo o código que vai do token `beginchar` até o `endchar` será armazenadono glifo e ele será considerado completamente inicializado (mas não renderizado). O token de string que especificou o glifo também será atualizado para que seu ponteiro interno aponte para o novo glifo criado:

Seção: Instrução: Composta (continuação):

```
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;
}
```

Já se estivermos em modo de carregamento e lermos um token `endchar`, isso sempre será um erro: significa que o código possui um `endchar`, mas não um `beginchar`. Já que os tokens de `endchar` são tratados na mesma iteração em que tratamos o `beginchar`:

Seção: Instrução: Composta (continuação):

```
else if(begin -> type == TYPE_ENDCHAR && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

Depois, quando estivermos em modo de execução, e não em modo de carregamento, iremos ler todo esse código entre `beginchar` e `endchar` novamente. E será importante memorizar qual o glifo que estamos tratando anquele instante. Por causa disso, armazenaremos no contexto qual glifo dvemos renderizar:

Seção: Atributos (struct context) (continuação):

```
struct _glyph *current_glyph;
```

Finalmente, vamos tratar o `beginchar` quando estamos em modo de execução. Primeiro lemos o token de string que vem em sequência. Como já havíamos executado este código no modo de carregamento, o token de string possui um ponteiro para um glifo já alocado e totalmente inicializado. Precisamos apenas renderizá-lo.

O próximo passo será obter os valores no cabeçalho do `beginchar`, lendo os valores numéricos de largura, altura e profundidade e usando tais valores para gerar uma nova `currentpicture`. A imagem renderizada terá o dobro dos valores indicados para que possamos obter tons de cinza realizando uma multi-amostragem, gerando cada pixel à partir de 4 outros pixels de amostragem. Exceto se a macro `W_WeaveFont_DISABLE_MULTISAMPLE` estiver definida, caso em que não usaremos a técnica. Por fim, a caneta atual é modificada para ser uma caneta nula.

Seção: Instrução: Composta (continuação):

```
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);
// Primeiro obtemos o glifo atual:
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;
```

<Seção a ser Inserida: beginchar: Reinicia 'Kerning'>

```
memset(cx -> current_character, 0, 5);
memcpy(cx -> current_character, str -> value, 4);
// Lendo os valores no cabeçalho
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);
// Obtendo largura: se monoespaço ou 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;
  { // Inicializando 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;
    }
    // Pintando a nova textura de branco
    memset(data, 255, size);
    { // E deixando ela totalmente transparente:
      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);
```
<Seção a ser Inserida: Gera nova 'currentpicture'>
```
  }
  return true;
}
```

Finalmente, quando lemos um `endchar` em modo de execução, significa que terminamos de renderizar um novo glifo. Devemos então copiar a imagem renderizada de `currentícture` para o glifo. E se estivermos usando multi-amostragem, devemos ajustatr o tamanho do glifo para a metade, para ser renderizado na tela com metade do tamanho que tem na memória. Por fim, devemos marcar o glifo como já renderizado e apagar o conteúdo de `currentpicture`:

Seção: Instrução: Composta (continuação):

```
else if(begin -> type == TYPE_ENDCHAR){
  struct picture_variable *currentpicture = cx -> currentpicture;
  if(!end_nesting_level(mf, cx, begin))
    return false;
  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. Funções de API para Usar as Fontes

Um código WeaveFont pode ter suas variáveis globais modificadas e assim atualizar a renderização de glifos e imagens sem que o código precise ser reescrito e reinterpretado. Para escrever e ler variáveis globais numéricas, usamos as seguintes funções que serão exportadas:

Seção: Declaração de Função (metafont.h) (continuação):

```
bool _Wwrite_numeric_variable(struct metafont *mf, char *name, float value);
float _Wread_numeric_variable(struct metafont *mf, char *name);
```

O que a primeira função fará será mudar a variável global ou interna indicada por tal nome e irá marcar todos os glifos como necessitando ser renderizados novamente. Se não existir variável numérica com tal nome, retornamos falso. Se não, retornamos verdadeiro:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
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;
}
```

O código para lermos uma variável numérica será similar. Se a variável numérica global com tal nome não existir, retornamos NAN:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
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;
}
```

Já a função que lê um arquivo com código WeaveFont e carrega uma nova meta-fonte é:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
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;
}
```

Finalmente, se quisermos renderizar um caractere e obter informações sobre suas dimensões em pixels (largura, altura, profundidade, correção itálica e "kerning"), executamos a seguinte função:

Seção: Declaração de Função (metafont.h) (continuação):

```
bool _Wrender_glyph(struct metafont *mf, char *glyph,
                    char *next_glyph, GLuint *texture,
                    int *width, int *height, int *depth,
                    int *italcorr, int *kerning);
```

E a implementação da função:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
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. O Comando `shipit`

Na linguagem METAFONT original, para que qualquer caractere fosse renderizado, era necessário usar a macro `shipit` ou o comando `shipout` passando uma imagem como parâmetro. Na linguagem WeaveFont, isso não é necesário, pois temos o comando composto `beginchar` reconhecido como uma primitiva da linguagem. No METAFONT original, o `beginchar` não era uma primitiva, mas sim uma macro, e isso impedia que a renderização pudesse ocorrer de maneira automática.

Mas ao contrário da linguagem METAFONT, o WeaveFont produz renderizações em tempo real. E por causa disso, pode haver diferenças em como queremos tratar a imagem renderizada. Tipicamente, ao produzirmos um caractere, vamos querer armazenar em cache sua imagem para que possamos usá-la várias vezes, sem que precisemos interpretar novamente o seu código. Renderizar um texto seria muito mais lento se a cada nova letra tivéssemos que interpretar novamente código WeaveFont para produzir novamente o desenho.

Mas há ocasiões em que queremos renderizar algo temporariamente. Por exemplo, caso tenhamos produzido uma fonte tipográfica em que cada letra é única, tendo seus parâmetros modificados aleatoriamente cada vez que produzimos um novo caractere. Desta forma, mesmo que um mesmo caractere apareça várias vezes, ele será ligeiramente diferente cada vez que aparece.

Podemos também usar WeaveFont para criar animações vetoriais. Uma forma de fazer isso é renderizar um mesmo caractere várias vezes. Se ele foi projetado para ser uma animação, cada execução irá corresponder a um novo quadro.

O que determina se um glifo precisa ser renderizado ou não é o seu atributo `need_rendering`. Este atributo é sempre inicializado como verdadeiro e também se torna verdadeiro se uma variável global muda. Ele se torna falso somente se encontramos um caractere `endchar`. Vamos então usar um outro comando diferente de `endchar` que também vai encerrar a renderização, mas ele não irá ajustar como falso o atributo `need_rendering`. Fora isso, ele se comportará exatamente da mesma forma como um `endchar`, exceto por ele poder estar aninhado dentro de outros comandos compostos como um `if`. Escolheremos o comando `shipit` para isso, que tem a seguinte gramática:

```
<Comando> -> ... |<Comando 'shipit'> | ...
<Comando 'shipit'> -> shipit
```

Vamos adicionar então o token correspondente e a palavra reservada para o `shipit`:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_SHIPIT,   // O token simbólico 'shipit'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"shipit",
```

Se encontrarmos esse comando em modo de carregamento, isso significa que ele está foram de um `beginchar` e isso é um erro:

Seção: Instrução: Comando:

```
else if(begin -> type == TYPE_SHIPIT && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

Ao encontrar tal comando, se não estivermos em modo de carregamento, encerramos o desenho atual como se tivéssemos encontrado um `endchar` válido, exceto que não mudamos a sua flag `need_rendering`. Ajustamos também o ponteiro de finalização para o próximo `endchar`, que obtemos consultando o seu ponteiro no `beginchar` ativo:

Seção: Instrução: Comando:

```
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;
  { // Voltar o nível de aninhamento para anterior ao 'beginchar':
    struct linked_token *aux = cx -> end_token_stack;
    while(aux != NULL && aux -> type != TYPE_ENDCHAR){
      if(aux -> type == TYPE_ENDFOR){ // Loop deve ser desativado:
        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;
}
```

Algo relevante sobre o comando `shipit` é que torna-se responsabilidade do programador apagar da placa de vídeo a textura produzida por ele, e que deve sr retornada por uma função como `_Wrender_glyph`. Ou o programador deve saber se o glifo que ele acabou de renderizar usa o `shipit`, ou então ele deve conferir após a renderização o valor do atributo `need_rendering` para saber como tratar a textura.

18. O Comando `renderchar`

Muitas vezes um glifo tipográfico é formado pela junção de outros glifos. Por exemplo, o glifo "$" pode ser visto como um glifo "S" com um traço vertical adicionado. Em alguns sistemas de escrita, tais composições são ainda mais comuns. Por exemplo, em logogramas chineses podemos encontrar caracteres que são radicais que formam outros caracteres. Os quais, por sua vez, podem ser a base de vários outros também. Um exemplo são os logogramas abaixo que são: "rén" (pessoa), "dá" (grande), "tiān" (céu), "Wù" (um sobrenome comum), e "yú" (prazer):

人大天吴娱

Nem sempre é adequado, mas algumas vezes um caractere pode ser diretamente estendido de outro existente, o que torna mais fácil e rápido definir uma fonte composta por muitos glifos. Para isso, fornecemos o comando `renderchar`, cuja gramática pode ser vista abaixo:

```
<Comando> -> ... |<Comando 'renderchar'> | ...
<Comando 'renderchar'> -> renderchar <String> between <Expressão de Par>
                          and <Expressão de Par>
```

O que requer suporte para o token e palavra reservada `between`:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_RENDERCHAR,  // O token simbólico 'renderchar'
TYPE_BETWEEN,     // O token simbólico 'between'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"renderchar", "between",
```

Se este comando for encontrado em modo de carregamento, isso é um erro. Quando estamos neste modo, não sabemos ainda se o caractere em questão será definido ou não:

Seção: Instrução: Comando:

```
else if(begin -> type == TYPE_RENDERCHAR && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

Já se não estivermos em modo de carregamento, este comando irá: (1) obter o glifo pedido, renderizando-o caso ele ainda não tenha sido renderizado, e (2) desenhará o glifo na textura do caractere atual, usando as expressões de pares para delimitar onde desenhá-lo:

Seção: Instrução: Comando:

```
else if(begin -> type == TYPE_RENDERCHAR){
  struct _glyph *glyph;
  struct string_token *str;
  struct pair_variable p1, p2;
```

<Seção a ser Inserida: Renderchar: Obter Glifo>

<Seção a ser Inserida: Renderchar: Renderizar Glifo>

```
  return true;
}
```

Primeiro nos certificamos que existe uma string após o `renderchar`. Se não existir, é um erro. Se existir, usamos ela para obter o glifo a ser desenhado:

Seção: Renderchar: Obter Glifo:

```
{
  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;
  }
}
```

Pode ser que a renderização de um primeiro caractere dependa de um segunda, que por sua vez dependa de um terceiro e quarto. Isso não é um problema. O que não podemos permitir que aconteça é que a renderização de um caractere dependa de sua própria renderização. Recursividade não deve ser permitida no `renderchar`. Detectamos isso no código abaixo:

Seção: Renderchar: Obter Glifo (continuação):

```
if(glyph -> is_being_rendered){
  RAISE_ERROR_RECURSIVE_RENDERCHAR(mf, cx, OPTIONAL(str -> line),
                                   str -> value);
  return false;
}
```

Finalmente, se o caractere invocado pelo `renderchar` não foi ainda renderizado, nós o renderizamos:

Seção: Renderchar: Obter Glifo (continuação):

```
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);
}
```

Agora para renderizar o glifo, devemos identificar onde exatamente na textura da imagem atual devemos colocá-lo. Para isso, devemos obter a primeira expressão de par, delimitada por `between` e `and` e armazenar seu resultado:

Seção: Renderchar: Renderizar Glifo:

```
{
  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){
    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;
  }
}
```

Se a imagem não for sofrer nenhuma transformação, esperamos que o primeiro par seja a coordenada do canto inferior esquerdo de onde renderizaremos o glifo e o segundo par seja a coordenada do canto superior direito. Isso implicaria que necessariamente ambas as coordenadas do primeiro par devem ser menores que as do segundo.

Se isso não for verdade, assumiremos que a imagem deve passar por uma transformação. Se a coordenada x do primeiro par for maior que a do segundo, a imagem deverá ser espelhada horizontalmente. Se a coordenada y do primeiro par for maior, então a imagem será espelhada verticalmente. Isso funciona simplesmente multiplicando por -1 as posições relevantes da matriz que usaremos na renderização:

Seção: Renderchar: Renderizar Glifo (continuação):

```
{
  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);
  }
```

<Seção a ser Inserida: Prepara 'currentpicture' para Desenho>

```
  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. O Comando `kerning`

Frequentemente é preciso ajustar o espaçamento entre dois glifos renderizados que serão colocados lado-a-lado. Por exemplo, em sequências como "VA", o espaçamento do "A" costuma ser ajustado mais para a esquerda de modo que a ponta inferior esquerda desta letra comece um pouco antes da ponta superior direita da letra "V", produzindo um resultado estético mais agradável.

Em WeaveFont, estes ajustes de espaçamento são especificados com ajuda do comando `kerning`, cuja sintaxe é mostrada abaixo:

```
<Comando> -> ... |<Comando 'kerning'> | ...
<Comando 'kerning'> ->  kerning ( <String> , <Expressão Numérica> )
```

O que requer uma palavra reservada a mais:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_KERNING,  // O token simbólico 'kerning'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"kerning",
```

O comando `kerning` só pode ser usado dentro dadefinição de um glifo (entre `beginchar` e `endchar`), caso contrário, um erro é gerado (como mostrado abaixo).

Seção: Instrução: Comando:

```
else if(begin -> type == TYPE_KERNING && mf -> loading){
  RAISE_ERROR_UNEXPECTED_TOKEN(mf, cx, OPTIONAL(begin -> line), begin);
  return false;
}
```

O comando indica com uma expressão numérica qual o espaçamento adicional a ser colocado quando o próximo caractere é dado pela string passada como argumento para o comando. Um valor numérico negativo significa um espaçamento negativo adicional, o que desloca o próximo caractere para a esquerda. Um valor numérico positivo desloca para a direita. O código abaixo interpreta isso e adiciona o valor de kerning ao glifo sendo gerado:

Seção: Instrução: Comando:

```
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;
```

<Seção a ser Inserida: Armazena Kerning>

```
}
```

O código acima se certifica de que o comando obedece as regras gramaticais definidas para ele e interpreta a expressão numérica para obter o valor de "kerning". O código abaixo é o que efetivamente armazena o valor lido no glifo atual:

Seção: Armazena 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;
}
```

E finalmente, quando começamos a ler o código de um glifo após encontrarmos um `begin-char`, devemos zerar todos os valores de "kerning" que ele tenha antes. Pois se o glifo será renderizado novamente, ele deve produzir valores novos de espaçamento. E se não produzir, assumimos

que o espaçamento adicional será zero:

Seção: beginchar: Reinicia 'Kerning':

```
{
  struct kerning *existing_kerning = cx -> current_glyph -> kern;
  while(existing_kerning != NULL){
    existing_kerning -> kern = 0.0;
    existing_kerning = existing_kerning -> next;
  }
}
```

20. O Comando debug

O comando debug pode ter dois comportamentos diferentes, dependendo se estamos em modo de depuração ou não. Se estivermos, então ele lê a expressão que é passada para ele, extrai o resultado e imprime na saída padrão o resultado obtido. Se não estivermos em modo de depuração, a expressão que o segue é simplesmente ignorada e o comando não faz nada.

A sintaxe gramatical do comando é:

```
<Comando> -> ... |<Comando 'debug'> | ...
<Comando 'debug'> ->  debug <Expressão ou String Vazia>
```

E para isso, adicionamos debug como palavra reservada:

Seção: WeaveFont: Definição de Token Simbólico (continuação):

```
TYPE_DEBUG,  // O token simbólico 'debug'
```

Seção: Lista de Palavras Reservadas (continuação):

```
"debug",
```

O comando debug não é feito para ser muito eficiente. Ele é uma ajuda ocasional para depurar código que apresenta problemas, e deve ser removido quando o processo de depuração termina. Segue abaixo a estrutura do comando. Ele funciona primeiro buscando identificar qual o tipo da expressão que o segue e, dependendo do tipo, ele executa as funções adequadas para interpretar e obter o resultado:

Seção: Instrução: Comando:

```
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:
              <Seção a ser Inserida: Debug: Expressão Numérica>
    break;
  case TYPE_T_PAIR:
               <Seção a ser Inserida: Debug: Expressão de Par>
    break;
  case TYPE_T_TRANSFORM:
          <Seção a ser Inserida: Debug: Expressão de Transformação>
    break;
  case TYPE_T_PATH:
             <Seção a ser Inserida: Debug: Expressão de Caminho>
```

```
    break;
  case TYPE_T_PEN:
              <Seção a ser Inserida: Debug: Expressão de Caneta>
    break;
  case TYPE_T_PICTURE:
              <Seção a ser Inserida: Debug: Expressão de Imagem>
    break;
  case TYPE_T_BOOLEAN:
              <Seção a ser Inserida: Debug: Expressão Booleana>
    break;
  default:
    RAISE_ERROR_UNKNOWN_EXPRESSION(mf, cx, OPTIONAL(begin -> line),
                                   type);
    return false;
  }
#else
  return true;
#endif
}
```

No caso de uma expressão numérica, assim é como a interpretamos e geramos a mensage de depuração:

Seção: Debug: Expressão Numérica:

```
{
  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;
}
```

No caso de uma expressão de par:

Seção: Debug: Expressão de Par:

```
{
  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;
}
```

No caso de uma expressão de transformação:

Seção: Debug: Expressão de Transformação:

```
{
  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;
```

```
}
```

No caso de um Caminho:

Seção: Debug: Expressão de Caminho:

```
{
  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;
}
```

No caso de uma Caneta:

Seção: Debug: Expressão de Caneta:

```
{
  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;
}
```

Até então, buscamos criar uma representação em string para cada tipo de variável que fôsse compatível com um literal, ou uma expressão necessária para criar uma cópia daquela variável. Infelizemente isso não pode ser feito com imagens, pois exceno no caso de `nullpicture`, não existem literais ou expressões capazes de recriar uma imagem. Pois imagens são tipicamente manipuladas por comandos ao invés de expressões. Ainda assim, produzimos uma representação de uma expressão que gera uma imagem de mesma largura e altura.

Seção: Debug: Expressão de Imagem:

```
{
  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;
}
```

Já no caso de uma expressão Booleana, o valor só pode ser verdadeiro ou falso:

Seção: Debug: Expressão Booleana:

```
{
  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. Integração entre WeaveFont e Motor de Jogos Weaver

O motor de jogos Weaver possui um módulo responsável por carregar novas interfaces, que são elementos visuais a serem apresentados ao usuário. A ideia é que o motor possa criar automaticamente elementos visuais à partir de um nome de arquivo onde pode-se ver uma textura. Isso é feito por meio da invocação da função:

```
W.new_interface("textura.mf", NULL, 100.0, 100.0, 0.0, 100.0, 100.0);
```

A função acima deve ler o arquivo `textura.mf`, extrair a textura que está dentro dele e produzir uma nova estrutura de tipo `struct user_interface` que representa algo que imediatamente aparecerá na tela.

Mas como o motor de jogos sabe como interpretar o arquivo `textura.mf`? Para isso, ele deve ser inicializado passando para ele o ponteiro de uma função que interpreta tal textura. No nosso caso, vamos assumir que a tal textura é um código-fonte WeaveFont.

Para isso, se estivermos usando o motor de jogos Weaver, devemos inserir o cabeçalho que contém a descrição das interfaces de usuário:

Seção: Cabeçalhos Locais (metafont.c) (continuação):

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

Devemos também definir uma nova função com o seguinte cabeçalho:

```
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)
```

Os quatro primeiros parâmetros são apenas funções para as funções de alocação e desalocação que devemos usar quando interpretamos o arquivo. As duas próximas funções são apenas ponteiros para funções que devem ser executadas antes e depois de interpretarmos a textura. Em seguida, vem o caminho do arquivo onde está o código a ser interpretado. E por fim, umponteiro para a interface onde devemos colocar a textura lida. É possível definir texturas animadas.

Definimos tal função como:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
#if defined(WEAVER_ENGINE)
 <Seção a ser Inserida: Funções a Serem Colocadas na Interface de Usuário>
void _Wmetafont_loading(void *(*p_alloc)(size_t), //Não usado
                        void (*p_free)(void *), // Não usado
                        void *(*t_alloc)(size_t), // Não usado
                        void (*t_free)(void *), // Não usado
                        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){
    if(after_loading_interface != NULL)
      after_loading_interface();
    return;
  }
  if(mf -> err){
```

```
    _Wprint_metafont_error(mf);
    if(after_loading_interface != NULL)
      after_loading_interface();
    return;
  }
  // Computando a altura da interface, dada em pixels, para pontos tipográficos.
  // Se for suportado pela textura, usamos isso para determinar seu tamanho.
  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);
  // A textura  é o primeiro glifo definido em mf -> first_glyph_symbol
  target -> _texture1 = (GLuint *) permanent_alloc(sizeof(GLuint));
  if(target -> _texture1 == NULL){
    if(after_loading_interface != NULL)
      after_loading_interface();
    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);
    if(after_loading_interface != NULL)
      after_loading_interface();
    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
```

A função acima cria uma nova interface de usuário à partir de um código WeaveFont, onde o código irá definir a textura que a interface terá. Um ponteiro para a estrutura metafont é também colocado na interface, e também um ponteiro para função de finalização e para recarregar a textura.

A função de finalização deve receber um ponteiro genérico e deve fazer qualquer trabalho necessário para encerrar e desalocar qualquer recurso que tenhamos colocado na interface. No caso, para nós será necessário executar `_Wdestroy_metafont` na estrutura metafont armazenada:

Seção: Funções a Serem Colocadas na Interface de Usuário:

```
void _interface_free_metafont(void *data){
  _Wdestroy_metafont((struct metafont *) data);
}
```

Já a função que recarrega texturas é executada toda vez que a interface de usuário muda de tamanho, ou a cada frame que ela for animada. O que ela fará será recarregar a textura reinterpretando o código WeaveFont:

Seção: Funções a Serem Colocadas na Interface de Usuário (continuação):

```
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 tipográficos.
  // 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;
}
```

Apêndice A: Tratamento de Erros

Quando erros existirem em um código-fonte WeaveFont, nenhuma imagem ou meta-fonte tipográfica pode ser gerada. Entretanto, o programador deve ter alguma forma de descobrir o quê deu errado para que o problema possa ser ajustado. Por causa disso, vamos adicionar à estrutura de cada meta-fonte variáveis que, em caso de erro, conterão um diagnóstico do quê aconteceu de errado:

Seção: Atributos (struct metafont) (continuação):

```
  int err, errno_line; // Código de erro, linnha em que teve erro
  char errno_character[5]; // Caractere sendo renderizado durante erro
  char errno_str[32]; // Informações adicionais na forma de string
```

```
  int errno_int; // INformações adicionais na forma de inteiro
```

A ideia é que caso nenhum erro aconteça, todas as variáveis acima serão igual a zero ou nulo. Então, quando inicializamos uma meta-fonte, precisamos inicializar estas variáveis com tais valores que indicam ausência de erro:

Seção: Inicialização (struct metafont) (continuação):

```
mf -> err = mf -> errno_line = 0;
mf -> errno_character[0] = '\0';
memset(mf -> errno_str, 0, 32);
mf -> errno_int = 0;
```

A ideia é que tão logo o primeiro erro seja encontrado, o valor de `err` deve mudar para algo que indique a natureza do erro encontrado. Caso estejamos contando o número de linhas no código (ou seja, quando a macro `W_DEBUG_METAFONT` estiver definida), então a linha onde o erro ocorreu deve ser armazenada em `errno_line`. E se mais informação precisam ser indicadas para que o erro possa ser descrito com mais precisão, então usaremos o ponteiro logo abaixo e o faremos apontar para uma string relevante. Ou armazenaresmo algum número que explique o erro em `errno_int`.

Somente o primeiro erro encontrado em cada meta-fonte deve ser armazenado. Se houverem mais, eles devem ser ignorados. Todas as funções que executam código WeaveFont devem retornar um valor booleano. Caso nenhum erro tenha ocorrido, retrona-se verdadeiro. Já em caso de erros, retorna-se falso.

Os diferentes tipos de erros serão armazenados aqui:

Seção: Estrutura de Dados Locais (metafont.c) (continuação):

```
enum { // Tipos de Erros
  ERROR_NO_ERROR = 0,
  // Os diferentes tipos que serão definidos:
                    <Seção a ser Inserida: Tipos de Erros>
  // E um último que esperamos nunca usar: indica erro desconhecido
  ERROR_UNKNOWN
};
```

Há inúmeras coisas que podem dar errado e provocar erros. Podemos ficar sem memória quando estamos alocando um token, um contexto ou qualquer outra estrutura auxiliar da qual precisamos. Ou podemos ter encontrado um símbolo completamente não-suportado em nosso analisador léxico. Ou ainda podemos ter começado uma string e não terminado.

Pode ser difícil para um usuário ter que lembrar todos os tipos de erros para analisar corretamente como sua informação é armazenada. Por causa disso, uma função será exportada para imprimir na tela uma mensagem diagnosticando o erro encontrado:

Seção: Definição de Funções da API (metafont.c) (continuação):

```
void _Wprint_metafont_error(struct metafont *mf){
  char line_number[8];
  // Primeiro tentamos gerar uma string com o número de linha onde teve erro.
  // Mas se não temos um número de linha, só deixamos a string vazia:
  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.", mf -> file, line_number);
    break;
                <Seção a ser Inserida: Imprime Mensagem de Erro>
  default:
    fprintf(stderr, "%s:%s Unknown error.", mf -> file, line_number);
```

```
  }
  if(mf -> errno_character[0] != '\0'){
    fprintf(stderr, " (while rendering '%s')\n", mf -> errno_character);
  }
  else
    fprintf(stderr, "\n");
}
```

Cada subseção deste Apêndice irá listar e definir um diferente tipo de erro, como definimos a sua macro que o gera, a sua mensagem de erro, e exemplos de código que podem causá-lo.

Muitos deles usarão a macro genérica abaixo como parte da definição de suas macros. A macro abaixo apenas armazena informações genéricas que são relevantes para todos os tipos de erro: o código de erro, a linha em que ele ocorreu, e qual caractere (glifo) estava sendo renderizado no momento, se é que algo estava sendo renderizado:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

Como nem sempre sabemos a linha de código em que o erro está (só temos esta informação em modo de depuração), a macro abaixo vai nos ajudar a tratar tal informação como opcional:

Seção: Macros Locais (metafont.c) (continuação):

```
#if defined(W_DEBUG_METAFONT)
#define OPTIONAL(x) x
#else
#define OPTIONAL(x) 0
#endif
```

Quando formos imprimir diagnósticos de erro, também será importante termos acesso a uma função que, dado um token, nos mostra a sua representação na forma de string. Para a maioria dos tokens, isso não é algo difícil. Observe que em `metafont.c`, definimos uma lista de palavras reservadas identificada por "<`@Lista de Palavras Reservadas@`>". A ordem em que cada palavra reservada foi inserida nesta lista é a mesma ordem na qual fomos definindo os tokens. Exceto que a primeira palavra reservada que definimos foi "`begingroup`", quando já tínhamos definido alguns tokens anteriores. Isso significa se o tipo de token for maior ou igual que `TYPE_BEGINGROUP`, basta subtrair este menor valor dele e assim obtemos o índice correspondente à sua palavra reservada.

Se o token for entre `TYPE_FOR` e `TYPE_SEMICOLON`, então ele não estará listado como palavra reservada. De qualquer forma, o token terá um nome único baseado somente em seu tipo. Por fim, tokens cuja identificação é menor, serão variáveis, strings e números. Descobrir a representação de tais tokens requer consultar a sua estrutura.

Por meio destas regra, podemos fazer uma função auxiliar que produz uma string com o nome de um token dado o número representando o tipo de token e podemos usar tal função auxiliar para gerar nossa função mais geral.

O tratamento dos casos mais simples, recebendo como argumento um token e uma string a ser preenchida (tamanho máximo: 32) funcionaria assim:

Seção: Declaração de Função Local (metafont.c) (continuação):

```
void token_to_string(struct generic_token *tok, char *dst);
void tokenid_to_string(int token_id, char *dst);
```

Seção: Funções Auxiliares Locais (metafont.c) (continuação):

```
void token_to_string(struct generic_token *tok, char *dst){
        <Seção a ser Inserida: token“to“string: Casos Mais Complexos>
  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;
      }
  }
}
```

No caso de tokens numéricos, convertemos eles para string usando a função da biblioteca padrão `snprintf`:

Seção: token_to_string: Casos Mais Complexos:

```
if(tok -> type == TYPE_NUMERIC){
  snprintf(dst, 32, "%g", ((struct numeric_token *) tok) -> value);
  return;
}
```

No caso de strings, copiamos o valor delas, adicionando as aspas duplas:

Seção: token_to_string: Casos Mais Complexos (continuação):

```
if(tok -> type == TYPE_STRING){
  dst[0] = '"';
  memcpy(&dst[1], ((struct string_token *) tok) -> value, 29);
  strncat(dst, "\"", 2);
  return;
}
```

No caso dos tokens simbólicos, eles são nomes de variáveis. Lidar com eles não é muito diferente de lidar com strings. Mas não usamos aspas neles:

Seção: token_to_string: Casos Mais Complexos (continuação):

```
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;
}
```

Tudo o que descrevemos acima serve para que uma mensagem de erro possa ser impressa pelo comando `_Wprint_metafont_error`. Mas e se o usuário não quiser imprimir o erro e queira diagnosticar ele manualmente, gerando o seu tratamento de erros customizado? Neste caso, ele deve ler este Apêndice para saber como os erros são tratados e como suas informações são adicionadas. Sabendo disso, ele pode inspecionar por conta própria em `mf -> error` se algum erro ocorreu e qual o tipo dele. Ele pode forçar certos erros a serem ignorados para que tente-se executar mais uma vez o código, mudando para falso a variável que armazena se um erro ocorreu. E ele pode escrever mensagens customizadas para cada tipo de erro.

Para esta segunda tarefa, ele precisará ter a lista de erros possíveis (cada um será descrito em uma subseção a seguir) e deve também ter uma lista de tokens possíveis para identificar algumas informações armazenadas sobre alguns erros. Vamos exportar todas estas informações em um cabeçalho próprio que só precisa ser inserido em um programa quando o usuário quer criar tratamentos de erros customizados:

(P)

Arquivo: `src/metafont_error.h`:

```
#ifndef __WEAVER_METAFONT_ERROR
#define __WEAVER_METAFONT_ERROR
#ifdef __cplusplus
extern "C" {
#endif
enum { // Tipos de Erros
  ERROR_NO_ERROR = 0,
  // Os diferentes tipos que serão definidos:
                    <Seção a ser Inserida: Tipos de Erros>
  // E um último que esperamos nunca usar: indica erro desconhecido
  ERROR_UNKNOWN
};
enum { // Tipos de Tokens
  TYPE_NUMERIC = 1, TYPE_STRING, TYPE_SYMBOLIC, TYPE_FOR, TYPE_ENDFOR,
  // Os tipos básicos (numérico, string, varivel, começo e fim de
  // loop) estão acima. Os outros serão colocados logo abaixo:
```

<Seção a ser Inserida: WeaveFont: Definição de Token Simbólico>

```
  // E um último tipo que não deve ser usado, exceto para indicar erro:
  TYPE_INVALID_TOKEN
};
#ifdef __cplusplus
}
#endif
#endif
```

A.1. ERROR_DISCONTINUOUS_PATH

Seção: Tipos de Erros (continuação):

```
ERROR_DISCONTINUOUS_PATH,
```

Este erro ocorre quando tentamos concatenar dois caminhos, mas tentamos uni-los por meio de pontos que estão muito distantes entre si. Neste caso, a concatenação é impossível e o erro é gerado. Devemos passar para a geração de erro quatro valores numéricos: as coordenadas do primeiro ponto e as coordenadas do segundo para avisar o usuário que são pontos diferentes e não podem ser fundidos.

Vamos tratar como números em ponto flutuante de precisão simples os quatro valores de coordenadas passados. O que significa que precisamos de 16 bytes para armazenar todos eles. Como é um caso bastante particular deste erro, e não é tão comum assim que precisemos armazenar tantos valores numéricos nos demais erros, podemos armazená-los sequancialmente no buffer que normalmente usamos para armazenar strings de mensagem de erro, já que ele tem um espaço de 32 bytes:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

A mensagem de erro para avisar o usuário:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_DIVISION_BY_ZERO,
```

Ao encontrar este erro, apenas invocamos o procedimento padrão, sem precisar passar qualquer outra informação.

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_DIVISION_BY_ZERO(mf, cx, line) {\
```

```
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_DIVISION_BY_ZERO);}
```

E esta é a mensagem de diagnóstico:

Seção: Imprime Mensagem de Erro (continuação):

```
case ERROR_DIVISION_BY_ZERO:
  fprintf(stderr, "%s:%s Division by zero.", mf -> file, line_number);
  break;
```

A.3. ERROR DUPLICATE GLYPH

Seção: Tipos de Erros (continuação):

```
ERROR_DUPLICATE_GLYPH,
```

Este erro ocorre quando um glifo é definido duas vezes com o comando `beginchar`. Quando ele ocorre, devemos armazenar quaç é o glifo que está tendo uma definição redundante.

Seção: Macros Locais (metafont.c) (continuação):

```
#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';}}
```

E esta é a mensagem de erro apresentada para o usuário:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_EMPTY_DELIMITER,
```

Este erro é gerado caso encontramos delimitadores vazios. Por exemplo:

```
{}
()
[]
```

Na linguagem METAFONT original, um delimitador vazio `[]` era permitido na declaração de variáveis, para declarar que uma variável específica podia armazenar vários valores diferentes, usando números para representar a posição interna de cada valor (são arrays associativos, ou dicionários, onde o índice sempre é um número que pode ser não-inteiro). Mas WeaveFont não suporta isso, então qualquer delimitador vazio é tratado como erro.

Ao gerar este erro devemos informar qual o tipo de delimitador vazio que encontramos. Pode ser parênteses, colchetes ou chaves:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

A mensagem de erro vai ser personalizada de acordo com qual o delimitador foi encontrado vazio:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_EXPECTED_FOUND,
```

Este tipo de erro ocorre quando esperávamos encontrar um tipo de token em uma instrução, mas acabamos encontrando um outro tipo.

Se isso ocorre, devemos armazenar como número a identificação do token que esperávamos encontrar e como string a representação do token encontrado:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

E com isso podemos produzir uma mensagem de erro informativa:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_FAILED_OPENING_FILE,
```

Este erro ocorre quando tentamos ler um arquivo com código-fonte que não existe ou não pode ser lido por algum motivo. Para informar de maneira correta uma mensagem de diagnóstico de erro, quando detectamos este tipo de erro, devemos armazenar o nome do arquivo que tentamos abrir e o valor da variável `errno`: '

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_FAILED_OPENING_FILE(mf, cx, line, str) {\
  if(!mf -> err){\
    size_t _len = strlen(str) + 1;\
    if(_len > 32) _len = 32;\
    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';}}
```

E ao imprimir a mensagem de erro usamos estas informações para fornecer um diagnóstico com ajuda da função `strerror` aplicada sobre o valor salvo do `errno`:

Seção: Imprime Mensagem de Erro:

```
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;
```

Mas isso significa que precisamos incluir o cabeçalho com suporte à variável `errno`:

Seção: Cabeçalhos Locais (metafont.c) (continuação):

```
#include <errno.h>
```

A.7. ERROR_INCOMPLETE_SOURCE

Seção: Tipos de Erros:

```
ERROR_INCOMPLETE_SOURCE,
```

Este erro ocorre toda vez que estamos interpretando uma instrução simples, mas o código-fonte termina no meio dela. Indica que o programa WeaveFont está incompleto, ou o usuário inadvertidamente colocou por engano alguns caracteres adicionais no final do código-fonte.

Tratamos este erro da forma padrão:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_INCOMPLETE_SOURCE(mf, cx, line) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INCOMPLETE_SOURCE);}}
```

E esta é a mensagem de erro para este erro:

Seção: Imprime Mensagem de Erro:

```
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

Seção: Tipos de Erros:

```
ERROR_INCOMPLETE_STATEMENT,
```

Este erro ocorre quando estamos processando uma instrução, mas encontramos o final dela (um ponto-e-vírgula) antes dela ter sido realmente finalizada. Isso pode indicar que um ponto-e-vírgula foi inserido incorretamente no meio da expressão, ou que alguma palavra reservada foi usada incorretamente.

Diante deste erro, invocamos o tratamento de erros padrão:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_INCOMPLETE_STATEMENT(mf, cx, line) {\
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_INCOMPLETE_STATEMENT);}}
```

E a nossa mensagem de erro:

Seção: Imprime Mensagem de Erro:

```
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

Seção: Tipos de Erros:

```
ERROR_INVALID_CHAR,
```

O analizador léxico WeaveFont funciona seguindo uma série de regras dependendo do tipo de caractere que encontra. Isso é o que permite a ele interpretar `2cm` como uma construção de tois tokens equivalente a `2*cm`, e interpretar `cm2` como sendo formada por um token só.

Para isso, ele classifica os diferentes tipos de caracteres em famílias e tipos. Mas nem todos os caracteres pertencem a alguma família e são usados em código WeaveFont. Por exemplo, caracteres como "ç" não possuem qualquer classificação e não podem ser usados, a menos que façam parte de uma string. Sendo assim, eles não podem fazer parte de nomes de variáveis ou coisas assim. WeaveFont restringe o uso de caracteres que não percencem ao conjunto ASCII em sua linguagem. Os únicos caracteres que podem ser usados fora de strings e comentários são:

```
.%0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
"(),;_><=:|'\+-/*?!#&@$^~[]{}
```

Se um caractere não suportado for encontrado em qualquer parte do código, este erro será gerado. Devido à natureza deste erro, o único lugar em que encontramos este erro é no nosso analizador léxico.

Quando o erro é gerado, devemos armazenar qual foi o caractere não-suportado que encontramos. Para isso, passamos os próximos 4 bytes à partir de onde tal caractere foi encontrado. Isso porque um caractere UTF-8 tem no máximo 4 bytes. O que fazemos então é alocar 5 bytes para a string dedicada à informações sobre erros e copiamos os bytes lidos para lá, colocando uma marcação de fim da string na última posição:

Seção: Macros Locais (metafont.c) (continuação):

```
#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';}}
```

Devemos imprimir uma mensagem de erro correspondente dizendo qual o caractere encontrado que causou problemas. Isso requer interpretar a codificação UTF-8 para descobrir se temos um caractere que ocupa um, dois, três ou quatro bytes. Sabendo disso, podemos imprimir o caractere na mensagem de erro e também indicar sua representação numérica:

Seção: Imprime Mensagem de Erro:

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_INVALID_COMPARISON,
```

Este erro ocorre quando o usuário usa operadores booleanos de comparação em expressões de tipo caminho, caneta ou imagem. Tais tipos não podem ser comparados entre si. Quando tal erro é gerado, nós armazenamos qual o tipo de expressão inválido foi usado na comparação e também qual foi o operador de comparação.

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

E a mensagem de diagnóstico:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_INVALID_DIMENSION_GLYPH,
```

Este erro ocorre quando vamos renderizar um caractere, e estamos diante de algo que terá largura não-positiva, ou estão, se somarmos a profundidade com a altura, obtemos também um valor não-positivo. Como é impossível gerar uma textura com uma dimensão igual a zero ou negativa, geramos tal erro e armazenamos as dimensões do glifo que tentamos gerar:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

E o erro gera esta mensagem de diagnóstico:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_INVALID_NAME,
```

Este erro ocorre quando o usuário tenta dar para uma variável um nome não-suportado porque é o nome de uma palavra reservada, é o nome de uma variável especial que não pode ser sobrescrita ou então o nome corresponde a um token que não é simbólico. Quando este erro é invocado, deve-se passar qual o token não suportado que foi usado como nome de variável e também o seu tipo. De posse do token, armazenamos uma string com o seu nome:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

De posse do nome do token e de seu tipo, podemos produzir uma mensagem de erro. A

mensagem deve indicar o nome do token que não pôde ser usado como nome de variável, e deve indicar o motivo do por quê isso ocorreu. O motivo será escrito em uma string adicional:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_INVALID_TENSION,
```

Quando lemos a especificação de uma curva, um dos parâmetros que o usuário pode controlar é a tensão em um segmento, deixando a curva que liga duas extremidades mais rígida, próxima de uma reta, ou mais frouxa, fazendo um desvio maior até chegar no destino.

O modo pelo qual a tensão é calculada tem a restrição de que o valor do parâmetro tensão não pode ser menor que 0.75, do contrário, podemos obter um conjunto de equações para as curvas que não pode ter solução.

Caso um valor de tensão menor que este é encontrado, este erro é gerado. A macro que o gera o erro armazena em `errno_int` se a tensão com valor inválido foi a primeira (0) ou a segunda (1) no segmento. E armazena no buffer `errno_str` de 32 bytes os cinco valores em ponto flutuante simples (totalizando 20 bytes, eles cabem tranquilamente no buffer) representando as coordenadas dos dois pontos de extremidade que possuem entre si a tensão inválida, e o próprio valor da tensão inválida:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

A mensagem de erro para avisar o usuário:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_MISSING_EXPRESSION,
```

Este erro ocorre sempre que esperava-se encontrar uma expressão de determinado tipo, mas ao invés disso, nada é encontrado. Ou então algo que não é tal expressão é encontrada. Por exemplo, se criamos uma atribuição sem nada do lado direito da expressão.

Diante de tal erro, precisamos armazenar o tipo de expressão que esperávemos encontrar, mas não encontramos:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

E aqui imprimimos a mensagem com o dignóstico do erro:

Seção: Imprime Mensagem de Erro (continuação):

```
case ERROR_MISSING_EXPRESSION:
  fprintf(stderr, "%s:%s Missing '%s' expression.",
          mf -> file, line_number, mf -> errno_str);
  break;
```

A.15. ERROR_MISSING_TOKEN

Seção: Tipos de Erros (continuação):

```
ERROR_MISSING_TOKEN,
```

Este erro ocorre quando o usuário inicial alguns comandos compostos (como `if`, por exemplo) sem encerrá-los com o seu respectivo fechamento (`fi`, neste caso). Quando o erro é gerado, devemos usar como a linha de erro o ponto em que está a abertura do comando que não foi fechado, e o tipo de token no último argumento é o fechamento que estávemos esperando:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

E a mensagem de erro que mostramos para o usuário é:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_NEGATIVE_LOGARITHM,
```

Este erro é causado toda vez que tentamos calcular o logaritmo de um valor negativo. Quando o erro é gerado, nós convertemos o número negativo que causou problemas para uma string:

Seção: Macros Locais (metafont.c) (continuação):

```
#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';}}
```

E avisamos o usuário do problema:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_NEGATIVE_SQUARE_ROOT,
```

Este erro ocorre toda vez que tenta-se calcular a raíz quadrada de um número negativo. Quando isso ocorre, devemos armazenar uma representação em string do número negativo que causou os problemas:

Seção: Macros Locais (metafont.c) (continuação):

```
#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';}}
```

E esta é a mensagem impressa para o usuário:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_NESTED_BEGINCHAR,
```

Este tipo de erro ocorre quando temos uma instrução `beginchar` aninhada dentro de outra. Isso não pode ocorrer e semanticamente não faz sentido algum.

Este é um erro bastante simples, então não fazemos nada além do procedimento mínimo de quando um erro é encontrado:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_NESTED_BEGINCHAR(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NESTED_BEGINCHAR);}
```

E esta é a mensagem de erro quando encontramos tal problema:

Seção: Imprime Mensagem de Erro (continuação):

```
case ERROR_NESTED_BEGINCHAR:
  fprintf(stderr,
          "%s:%s You cannot nest 'beginchar' statements.", mf -> file,
line_number);
  break;
```

A.19. ERROR_NO_MEMORY

Seção: Tipos de Erros (continuação):

```
ERROR_NO_MEMORY,
```

A biblioteca que implementamos aqui que interpreta código WeaveFont recebe em sua inicialização funções de alocação, que podem ser o `malloc` da biblioteca padrão, ou podem ser alocadores personalizados. Caso o alocador seja executado e retorne NULL, isso significa que não conseguimos memória suficiente. É quando geramos este erro.

Ele pode ocorrer em praticamente qualquer parte do código. Ele não é causado porqualquer erro de sintaxe na linguagem. Sua causa pode ser devido a fatores externos. Contudo, eles também podem ser sintomas de vazamento de memória devido à práticas de programação questionáveis no programa WeaveFont. Por exemplo:

```
numeric i;
path p;
p = (0, 0);
for i = 0 step 0 until 1:
  p = p & p;
endfor
```

O código acima cria um laço infinito (talvez devido a um erro lógico do usuário?) e a cada iteração aumenta o tamanho de um caminho, adicionando mais um ponto de extremidade nele. Por mais memória que se tenha, uma hora o laço acima irá exaurir toda ela.

Quando detectamos tal erro, executamos a macro que o gera e que funciona apenas armazenando as informações padrão que todo erro precisa passar:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_NO_MEMORY(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NO_MEMORY);}
```

Esta é a mensagem de erro que imprimimos caso o usuário pergunte o quê aconteceu.

Seção: Imprime Mensagem de Erro:

```
case ERROR_NO_MEMORY:
  fprintf(stderr, "%s:%s Not enough memory for allocation.", mf -> file,
          line_number);
  break;
```

A.20. ERROR_NO_PICKUP_PEN

Seção: Tipos de Erros (continuação):

```
ERROR_NO_PICKUP_PEN,
```

Este erro ocorre quando usamos um comando `pickup`, mas não passamos para ele uma caneta válida. Talvez o usuário tenha terminado o comando sem passar a caneta, talvez tenha passado algo que não é uma caneta, ou então, erroneamente achou que uma expressão de caneta poderia ser passada para o comando (quando somente variáveis de caneta ou literais de caneta realmente armazenados em memória podem ser passados).

Este tipo de erro é gerado de forma genérica:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_NO_PICKUP_PEN(mf, cx, line) {\
```

```
  if(!mf -> err){\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NO_PICKUP_PEN);}}
```

E esta é a nossa mensagem de erro:

Seção: Imprime Mensagem de Erro:

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_NONCYCLICAL_PEN,
```

Este erro ocorre quando queremos criar uma nova caneta com formato personalizado usando `makepen`. Para isso, devemos passar como operando deste comando um caminho, que deve ser cíclico. Caso ele não seja cíclico, este erro é gerado:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_NONCYCLICAL_PEN(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NONCYCLICAL_PEN);}
```

A mensagem de erro para o usuário:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_NULL_VECTOR_ANGLE,
```

Este erro ocorre quando tentamos medir o ângulo que o vetor $(0, 0)$ forma com o eixo x. Contudo, como é o vetor nulo, não existe ângulo algum sendo formado e usar esta operação sobre o vetor nulo resulta no erro.

Este erro é tratado da maneira padrão:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_NULL_VECTOR_ANGLE(mf, cx, line) {\
    RAISE_GENERIC_ERROR(mf, cx, line, ERROR_NULL_VECTOR_ANGLE);}
```

E a mensagem de erro impressa na tela para informar o usuário:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_OPENGL_FRAMEBUFFER,
```

Este erro ocorre quando internamente um framebuffer não pode ser criado por erros no OpenGL. Isso idealmente não deve acontecer, mas se acontecer pode estar sendo causado por configuração incorreta do OpenGL feita ou modificada pelo usuário.

Para ajudar na depuração deste tipo de erro, armazenamos o estado do framebuffer atual no OpenGL, onde o erro provavelmente ocorreu:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

E a mensagem de erro para o usuário vai depender do estado do framebuffer:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_RECURSIVE_RENDERCHAR,
```

Este erro ocorre quando o uso do comando `renderchar` em dois ou mais glifos gera uma depenndência circular. Por exemplo, para renderizar o glifo "a", devemos primeiro renderizar "b". Mas para renderizar o "b", devemos renderizar "a". Nenhum tipo de recursão deste tipo é permitida, mesmo que o usuário tome cuidado de criar uma lógica que a impeça de ser infinita e nunca terminar. Quando uma definição recursiva deste tipo é encontrada, este erro é gerado. E

informamos qual o glifo que estamos precisando renderizar como dependência do atual, tal que o atual é dependência dele.

Seção: Macros Locais (metafont.c) (continuação):

```
#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';}}
```

E esta é a mensagem de erro impressa para tais erros:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNBALANCED_ENDING_TOKEN,
```

Este é um erro, geralmente capturado durante a análise léxica, onde um token, usado para finalizar uma instrução composta (`elseif`, `endif`, `endfor`, `endchar`) é encontrado em um contecto em que tal token não era para ser obtido. Ou a instrução composta nunca foi iniciada, ou então há mais de uma expressão composta e elas não estão balanceadas.

Ao encontrar tal erro, apenas armazenamos o número do token de finalização que obtivemos:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

E imprimimos a seguinte mensagem de erro para informar o usuário do problema encontrado:

Seção: Imprime Mensagem de Erro:

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNCLOSED_DELIMITER,
```

Este erro ocorre quando um delimitador como parênteses, colchetes ou chaves foi aberto, mas não foi fechado. Como cada um destes delimitadores é representado por um caractere, ao gerar o erro, deve-se passar como último argumento o caractere correspondente ao delimitador que não foi encerrado:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

E aqui imprimimos a mensagem de erro:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNCLOSED_STRING,
```

Este é um erro léxico que é gerado durante a criação de tokens, não durante a execução do código. Ele ocorre quando o código começa uma string com as aspas duplas, mas não a termina. Caso uma quebra de linha seja encontrada dentro de uma string, isso também configura uma string não terminada.

Quando encontramos tal erro, nós copiamos o que seria a string não-terminada para que possamos apresentá-la se o usuário pedir uma mensagem de erro:

Seção: Macros Locais (metafont.c) (continuação):

```
#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';}}
```

A mensagem de erro neste caso apenas informa a string incompleta encontrada e sua posição, na esperança de que isso seja o suficiente para que o usuário possa arrumar o problema:

Seção: Imprime Mensagem de Erro:

```
case ERROR_UNCLOSED_STRING:
  fprintf(stderr, "%s:%s Unclosed string \"%s\".", mf -> file,
          line_number, mf -> errno_str);
  break;
```

A.28. ERROR_UNDECLARED_VARIABLE

Seção: Tipos de Erros (continuação):

```
ERROR_UNDECLARED_VARIABLE,
```

Este erro ocorre toda vez que alguém tenta usar uma variável que não foi declarada. Quando ele é gerado, devemos passar o nome da variável como argumento:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

E ao imprimir a mensagem de erro, indicamos o nome da variável não-declarada:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNEXPECTED_TOKEN,
```

Erros de token inesperados significam que lemos um token que não esperávamos. Estávamos esperando alguma outra coisa (não sabemos o quê), mas não este token específico que encontramos. Isso ocorre também quando usamos um `endfor` sem que tenhamos aberto um `for` no nível de aninhamento atual. Ou quando usa-se de maneira errônea uma palavra reservada em contexto no qual ela não era esperada.

Se este erro ocorre, devemos armazenar em uma string qual o token inesperado que encontramos:

Seção: Macros Locais (metafont.c) (continuação):

```
#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);}}
```

E com isso podemos gerar uma mensagem de erro:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNINITIALIZED_VARIABLE,
```

Este erro ocorre quando tentamos acessar o valor de uma variável não-inicializada em uma expressão. Quando este erro ocorre, temos que passar como argumento o token da variável, e também o seu tipo. Com isso, armazenamos o nome da variável como string e o seu tipo como valor numérico:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

E isso nos permite gerar a mensagem de erro:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNKNOWN_GLYPH_DEPENDENCY,
```

Este erro ocorre quando um glifo tem como dependência de renderização algum outro glifo, que por sua vez, não foi definido. Ocorre quando usamos os comando `renderchar` passando para ele unm glifo não-definido. Diante deste erro, precisamos armazenar como string qual o nome do glifo inexistente que encontramos como dependência.

Seção: Macros Locais (metafont.c) (continuação):

```
#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';}}
```

E com isso produzimos a mensagem de erro:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNKNOWN_EXPRESSION,
```

Este erro ocorre quando estávamos interpretando uma expressão de determinado tipo (numérica, de par, de transformação, de caminho, de imagem ou booleana), mas encontramos uma sequência de tokens que não estavam previstos por qualquer regra gramatical.

Gerar esse erro requer armazenar o tipo de expressão que analizávamos quando encontramos algo completamente desconhecido:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

E a mensagem de diagnóstico é:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNKNOWN_STATEMENT,
```

Este é um erro gerado quando vamos ler uma instrução na linguagem, mas não somos capazes de identificar o seu tipo. Não sabemos se é uma instrução composta, declaração de variável, atribuição ou comando.

O erro é críptico demais para que possamos ajudar muito mais o usuário ou apontar o quê está errado. Pois não somos capazes de identificar o quê ele queria expressar. Como este é um erro sobre o qual não temos muitas informações, ele é gerado de maneira bastante genérica:

Seção: Macros Locais (metafont.c) (continuação):

```
#define RAISE_ERROR_UNKNOWN_STATEMENT(mf, cx, line) {\
  RAISE_GENERIC_ERROR(mf, cx, line, ERROR_UNKNOWN_STATEMENT);}
```

E geramos uma mensagem impressa caso o usuário pergunte o quê deu errado:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNOPENED_DELIMITER,
```

Esta mensagem de erro ocorre quando encontramos o fechamento de um delimitador (fechar parênteses, colchetes ou chaves) quando tal delimitador não foi aberto.

Quando este erro é gerado, devemos informar o caractere do delimitador sendo fechado que encontramos e que gerou o problema:

Seção: Macros Locais (metafont.c) (continuação):

```
#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 é a mensagem de diagnóstico deste erro:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_UNSUPORTED_LENGTH_OPERAND,
```

Este erro ocorre sempre depois que usamos o operador 'length'. Tal operador atua sobre um número (para calcular seu módulo), sobre um par (para calcular sua norma euclideana) ou sobre um caminho (para calcular o número de curvas que forma o caminho). Isso significa que depois de encontrarmos tal operador, precisamos identificar se estamos diante de um número, par ou caminho. Se estivermos diante de alguma outra coisa ou de uma expressão que não conseguimnos identificar, então este erro é gerado.

Ao gerar tal erro, passamos como último argumento o tipo de expressão que encontramos ao invés das que esperávamos. Caso o tipo seja desconhecido, passaremos o número -1 neste argumento:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

E assim produzimos a mensagem de erro:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_WRONG_NUMBER_OF_PARAMETERS,
```

Some instructions in the WeaveFont language require a certain number of parameters to be passed. For example, `beginchar(a, b, c, d)` requires four parameters, here represented by `a`, `b`, `c`, and `d`. If a smaller or larger number is passed, this error is generated.

When we generate this error, we need to indicate which instruction `s` caused the problem (we will store its name as a string), the number of arguments `e` we expected, and the value `f` which is how many we got instead (both integer values are combined and stored together in `errno_int`):

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

With these information, we can generate the correct errror message:

Seção: Imprime Mensagem de Erro (continuação):

```
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

Seção: Tipos de Erros (continuação):

```
ERROR_WRONG_VARIABLE_TYPE,
```

Este erro ocorre quando usamos uma variável de um tipo quando esperava-se que a variável fôsse de outro tipo. Ele ocorre quando formos realizar uma atribuição entre variáveis de tipos diferentes ou então quando uma variável de um tipo ocorre dentro de uma expressão que requeria uma variável de tipo diferente. Quando este erro é encontrado, devemos armazenar o nome da variável com problema, o seu tipo e também qual era o tipo que esperávaos no lugar. Como

na linguagem C uma variável inteira tem no mínimo 16 bits, usamos os 8 primeiros bits para armazenar o tipo da variável encontrada e os 8 últimos para armazenar o tipo esperado:

Seção: Macros Locais (metafont.c) (continuação):

```
#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;}}
```

Para imprimir a mensagem de erro, apenas temos que converter os tipos para strings de modo a formular a mensagem de erro:

Seção: Imprime Mensagem de Erro (continuação):

```
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;
```

Referências

De Berg, M. (2000) "Computational geometry: algorithms and applications". Springer Science & Business Media.

Knuth, D. E. (1984) "Literate Programming", The Computer Journal, Volume 27, Edição 2, Páginas 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