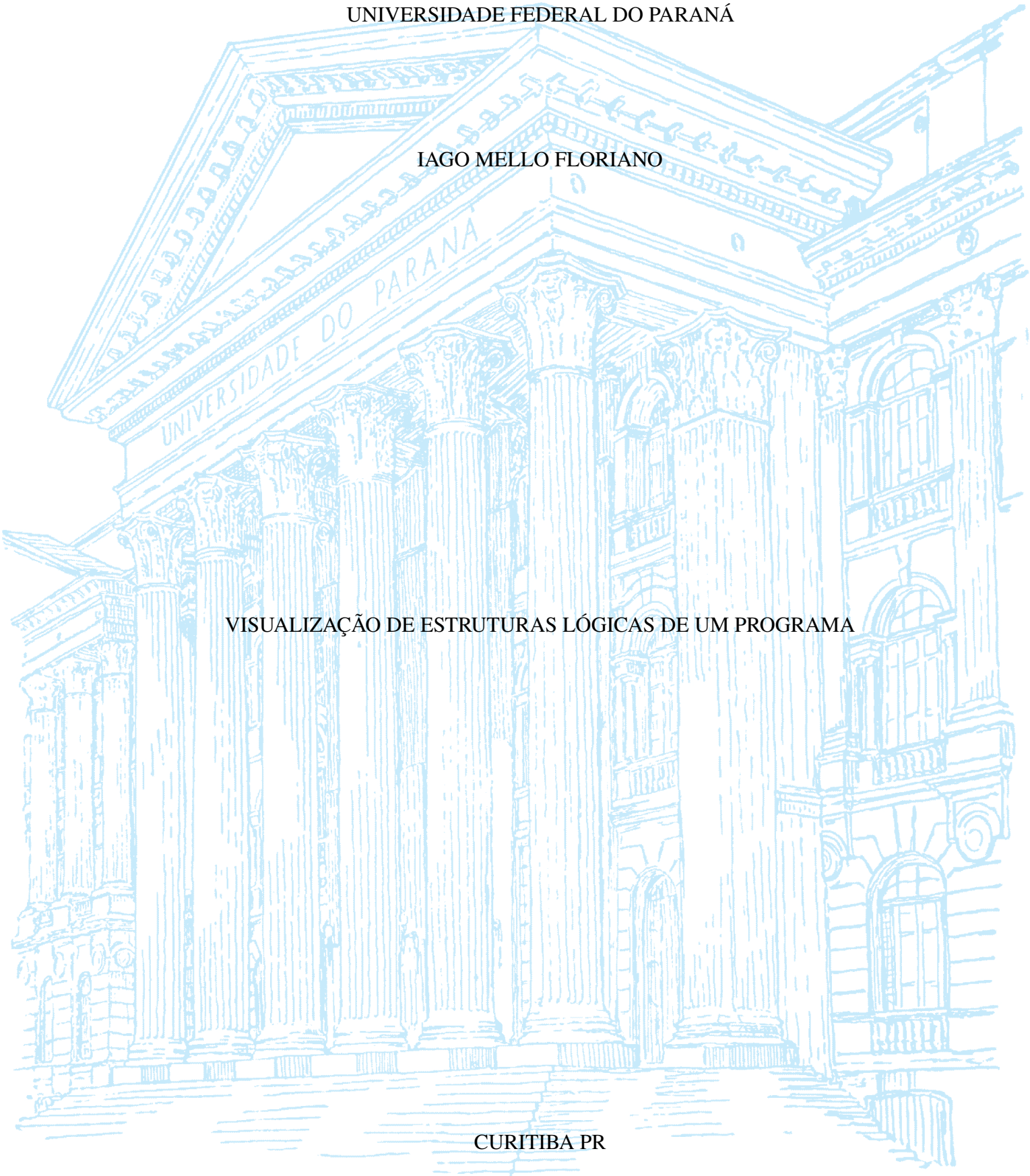
UNIVERSIDADE FEDERAL DO PARANÁ

IAGO MELLO FLORIANO



VISUALIZAÇÃO DE ESTRUTURAS LÓGICAS DE UM PROGRAMA

CURITIBA PR

2025

IAGO MELLO FLORIANO

VISUALIZAÇÃO DE ESTRUTURAS LÓGICAS DE UM PROGRAMA

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: Bruno Müller Junior.

CURITIBA PR

2025

Ficha de aprovação

Substituir o arquivo 0-iniciais/aprovacao.pdf pela ficha de aprovação fornecida pela secretaria do programa, em formato PDF A4.

ACKNOWLEDGEMENTS

Gostaria de expressar minha gratidão a minha família por estar sempre presente em minha vida, tanto nos momentos felizes quanto nos momentos difíceis. Gostaria de agradecer também ao professor Bruno Müller Junior pela sua orientação durante a produção desse trabalho.

RESUMO

O uso de ferramentas para visualização do fluxo de execução de um código de forma gráfica pode ser de ajuda para o entendimento sobre o funcionamento desse código. Verificar o funcionamento de um código é necessário para correção de trabalho em matérias que ensinam programação de computadores, portanto uma ferramenta gráfica pode auxiliar nesse processo. Esse trabalho desenvolve uma ferramenta que gera essa representação gráfica e explica as etapas usadas para gerar uma representação gráfica de um código. A ferramenta é dividida em dois programas: um compilador que gera uma representação intermediária do código, e o segundo usa essa representação para gerar a representação gráfica. A divisão em dois programas foi feita com o intuito de facilitar outros programas a gerar outras formas gráficas com o foco em outros aspectos do código.

Palavras-chave: Compiladores. Grafos. Representação Visual.

ABSTRACT

The use of tools for visualizing the flow of execution of a code in graphical form can help in the understanding of the way this code works. Verifying the way a code works is necessary for the grading of projects in courses that teach computer programming, for this reason a graphical tool can assist in this process. This work develops a tool that creates said graphical representation and explains the steps used to generate a graphical representation of a code. This tool is divided in two programs: one compiler that generates an intermediary representation of the code, and the second uses this representation to generate the graphical representation. The division in two programs was made with the goal to be easier to generate other graphical forms with the focus on different aspects of the code.

Keywords: Compilers. Graphs. Visual Representation.

Lista de Figuras

Lista de Códigos

Sumário

1 INTRODUÇÃO

Em disciplinas que ensinam programação de computadores, é comum o professor pedir aos alunos que elaborem programas para avaliar o aprendizado dos alunos. Ao elaborar o enunciado o professor já tem ideia da estrutura lógica do programa que ele entende como correto, como, por exemplo, a quantidade de condicionais (if), a quantidade de laços (while/for) e como estes se conectam (por exemplo, ter uma condicional dentro de um laço).

Ao avaliar um trabalho, o professor busca pela estrutura lógica esperada. Quando encontra essa estrutura, então o professor avalia em mais detalhes se além da lógica a sequência de comandos está correta.

É mais simples reconhecer esta sequência lógica em programas pequenos onde são esperados, por exemplo, programas com até dois laços ou programas com uma condição em um laço. Já em programas maiores, por exemplo, aqueles com três ou mais laços e condicionais, o reconhecimento da estrutura lógica torna-se mais difícil e uma ferramenta gráfica que mostra a estrutura lógica de forma visual poderia ajudar. Este texto descreve a implementação de tal ferramenta gráfica.

Como será visto adiante, a ferramenta gera um arquivo intermediário com informações sobre o programa que podem ser usadas em outras análises, como, por exemplo, cópia de código, além da visualização gráfica em si. Portanto, a ferramenta pode ser dividida em dois programas:

1. um compilador Pascal adaptado para gerar um arquivo texto intermediário.
2. um renderizador executado em um navegador web que utiliza o arquivo intermediário para gerar a visualização.

Este texto está organizado da seguinte forma, no Capítulo 2 será feita uma revisão bibliográfica e uma revisão dos termos que serão usados ao longo do texto. No Capítulo 3 o projeto será descrito conceitualmente e no Capítulo 4 mostrará como é gerada a representação gráfica e como interpretá-la. Finalmente o Capítulo 5 trará uma conclusão sobre o projeto feito.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma revisão dos termos utilizados ao longo do texto assim como uma revisão bibliográfica. A Seção 2.1 revê o que é um compilador. A Seção 2.2 apresenta o formato json, formato utilizado como representação intermediária. A Seção 2.3 explica como funciona a renderização gráfica utilizando a tag "canvas" do HTML5. Por fim, a Seção 2.4 introduz alguns trabalhos relacionados que utilizam do mesmo algoritmo utilizado nesse trabalho para fazer uma representação gráfica.

2.1 O COMPILADOR

O processo de compilação é o processo de usar uma entrada para gerar um arquivo em outro formato (por exemplo, um código feito em uma linguagem de programação ser usado para gerar um código de máquina que pode ser executado). Esse processo pode ser descrito nas seguintes etapas: análise léxica, análise sintática, e geração da saída. Essas etapas são conceituais e não cronológicas, por isto é comum que compiladores executem parte da análise léxica e sintática de forma intercalada.

A análise léxica é o processo de separar o texto do arquivo de entrada em símbolos significativos para a linguagem. Esses símbolos podem ser tanto palavras chaves da linguagem, quanto caracteres com algum significado específico dependendo do contexto (por exemplo, parênteses, ponto e vírgula, o símbolo de atribuição do pascal), quanto identificadores de variáveis, funções, procedimentos, etc. Uma vez separados, esses símbolos se chamam tokens. No código 2.1 alguns tokens são: program, exemplo, ;, var, a, integer, begin, end. Para esse trabalho foi utilizada a ferramenta flex para a geração de tokens.

```
1 program exemplo (input, output);
2 var a, b, c: integer;
3 begin
4 end.
```

Código 2.1: Exemplo de um programa pascal

Cada linguagem tem uma estrutura da ordem de tokens, algo que pode ser representado com regras gramaticais. Cada regra é definida como uma sequência de tokens ou regras, sendo que é possível que para uma regra ser válida mais de um token seja aceito em uma posição. Regras definidas dessa forma são facilmente descritas na notação BNF, que representa uma gramática determinística livre de contexto. Um analisador sintático verifica se uma entrada condiz com todas as regras gramaticais definidas pela linguagem. Caso algo esteja escrito sem seguir as regras da gramática, o analisador sintático deve mostrar que isso é um erro durante sua execução.

É possível criar um analisador sintático sem auxílio de nenhum outro programa, porém uma forma mais conveniente seria utilizar um gerador de analisadores sintáticos, conhecido como compilador de compiladores, que recebe as regras gramaticais desejadas e gera um analisador sintático, e para esse trabalho foi utilizado o Bison e a gramática descrita em (Kowaltowski, 2018).

O trecho de Código 2.2 exemplifica como é a feita a definição em Bison da gramática de um laço em pascal. Neste exemplo, tokens são escritos com letras maiúsculas e regras são escritas com letras minúsculas. A definição da regra é a seguinte sequência: o token "while",

uma sequência de tokens que respeitam a regra “expressao”, o token “do”, e uma sequência de tokens que respeitam a regra “comando_sem_rotulo”.

```
comando_repetitivo: WHILE expressao DO comando_sem_rotulo ;
```

Código 2.2: Definição de um comando while em Bison.

Apenas a análise sintática não seria muito útil, pois o programa apenas informaria se um programa respeita ou não as regras de uma gramática. Portanto, é útil gerar uma saída com base na árvore da análise da entrada do programa. Em Bison é possível adicionar um código em C para ser executado durante a análise sintática, esse código será executado após ser identificado algum trecho da gramática (token ou regra). Esse código a ser executado deve ser escrito entre chaves após o token ou regra desejada. O Código 2.3 mostra como seria uma forma de imprimir qual parte de um laço foi lida; o trecho em 2.5 mostra a saída do resultado da saída da leitura do código 2.4. Esse código pode ser escrito para fazer o que for desejado, tendo apenas as limitações da linguagem C. Para esse trabalho foi feita uma adaptação do código escrito para a disciplina CI1211, código esse que gera instruções para a máquina virtual MEPA definida em (Kowaltowski, 2018). A adaptação foi feita para manter dados que seriam irrelevantes para a geração do código MEPA, mas que serão úteis para a visualização da estrutura do código e também para gerar uma representação dessa estrutura em Json.

```
comando_repetitivo: WHILE {
   // Código a ser executado após encontrar um token WHILE
   // em um comando_repetitivo
   printf("Foi lido o token WHILE\n");
  } expressao {
   // Código a ser executado após reconhecer completamente uma
   // expressão em um comando_repetitivo
   printf("Foi lido uma sequência de tokens");
   printf(" que segue a regra `expressao`\n");
  } DO {
   // Código a ser executado após encontrar um token DO em um
   // comando_repetitivo
   printf("Foi lido o token DO\n");
  } comando_sem_rotulo {
   // Código a ser executado após reconhecer completamente um
   // comando_sem_rotulo em um comando_repetitivo
   printf("Foi lido uma sequência de tokens");
   printf(" que segue a regra `comando_sem_rotulo`\n");
  };
  {
   // Código a ser executado após o reconhecimento de um
   // comando_repetitivo
   printf("Foi lido o token ;\n");
  }
```

Código 2.3: Definição do comando while em Bison com códigos.

```
while (i <= 10) do
begin
  j := 0;
  while (j <= 10) do
  begin
    writeln(i*j);
    j := j + 1;
  end;
  i := i + 1;
```

```
10 end;
```

Código 2.4: Código Pascal de dois comandos while aninhados.

```
1  Foi lido o token WHILE
2  Foi lido uma sequência de tokens que segue a regra `expressao`
3  Foi lido o token DO
4  Foi lido o token WHILE
5  Foi lido uma sequência de tokens que segue a regra `expressao`
6  Foi lido o token DO
7  Foi lido uma sequência de tokens que segue a regra `comando_sem_rotulo`
8  Foi lido o token ;
9  Foi lido uma sequência de tokens que segue a regra `comando_sem_rotulo`
10 Foi lido o token ;
```

Código 2.5: Saída gera ao rodar o código 2.3 no código 2.4.

Note que, como existe um laço dentro de um laço no código pascal, antes de terminar a leitura dos tokens que seguem a regra de "comando_sem_rotulo" do laço externo, o laço interno foi lido completamente. Por causa disso as linhas 1 a 3 e linhas 9 e 10 são referentes ao laço externo e as linhas 4 a 8 são referentes ao laço interno.

2.2 JSON

O padrão Json foi criado para a representação do tipo objeto da linguagem Javascript, tipo esse que é uma coleção de pares chave-valor. A chave sendo uma sequência de caracteres e o valor podendo ser um número (inteiro ou com casas decimais), uma sequência de caracteres, um vetor de valores, ou um objeto. O formato Json é uma apenas uma forma legível por humanos para se exibir um objeto, sendo que na representação Json sequências de caracteres devem ter aspas duplas antes e depois, vetores devem ter colchetes antes e depois, e objetos devem ter chaves antes e depois. O código 2.6 exemplifica como é o formato Json, sendo uma lista na forma "chave":valor. A escolha do formato Json para esse projeto foi dada principalmente por ser um formato muito utilizado, e por ser legível por humanos, esses dois fatores auxiliam no objetivo principal do trabalho: facilitar a visualização da estrutura do programa.

```
1  {
2    "nome": "Aluno da Silva",
3    "idade": 20,
4    "informacoes_academicas": {
5      "status": "matriculado",
6      "curso": "Ciência da Computação",
7      "IRA": 0.8753,
8      "GRR": 20191234,
9      "materias_cursando": [
10       "CI1234",
11       "CI1235",
12       "CI1236",
13       "CI1237"
14     ]
15   }
16 }
```

Código 2.6: Exemplo de um arquivo Json para representar a situação de um aluno na universidade.

Além do formato ser de fácil leitura, pelo valor de uma chave poder ser um objeto, também permite representar fluxos de execução sem muitos problemas. Sendo que uma forma de

fazer essa representação, e a forma usada nesse projeto, é representar cada comando do fluxo com um objeto com as informações relevantes para aquele comando, e comandos que alteram o fluxo de execução do programa tendo filhos de acordo com como esse controle é feito. No trabalho foi definido que cada comando tem algumas chaves conhecidas para informações necessárias para aquele vértice, e chaves que serão criadas durante a execução do compilador serão filhos desse comando. O programa inteiro é um objeto com chaves para três objetos: variáveis, sub-rotinas (procedimentos e funções), e comandos. Esses três objetos foram decididos por representarem a estrutura de como um programa é escrito em Pascal. Cada variável tem uma chave sendo seu identificador e uma chave que guarda a informação sobre seu tipo no programa, por exemplo, número inteiro. Procedimentos e funções compartilham o mesmo objeto na representação por serem muito similares apenas com a diferença que procedimentos não têm retorno e funções têm um retorno. Cada procedimento e função tem como chave seu identificador, e as informações sobre a linha exata que declara esse procedimento ou função, ambos também têm informações sobre seus parâmetros (tipo de passagem, e tipo), variáveis e comandos, porém funções também têm a informação sobre seu tipo de retorno. Os objetos para comandos estão presentes em funções, procedimentos e no objeto do programa, sendo que sua chave é um nome dado com base no comando + um ID. Cada comando tem uma chave com a linha de como foi chamado esse comando. Os Códigos 2.7 e 2.8 são um programa pascal simples e sua representação em Json respectivamente:

```
1  program exemplo (input, output);
2  var varGlobal: integer;
3     varGlobal2: integer;
4
5  function funcao(var parReferencia: integer; parCopia: integer): integer;
6  begin
7    while (parReferencia < parCopia) do
8    begin
9      parReferencia := parReferencia +1;
10   end;
11   funcao := parCopia;
12 end;
13
14 begin
15   varGlobal := 2;
16   varGlobal2 := funcao(varGlobal, varGlobal + 2);
17 end.
```

Código 2.7: Exemplo de um código Pascal.

```
1  {
2    "variaveis": {
3      "varGlobal": {
4        "tipo": "integer"
5      },
6      "varGlobal2": {
7        "tipo": "integer"
8      }
9    },
10   "procedimentos": {
11     "funcao": {
12       "linha": "function funcao(var parReferencia :integer;parCopia:integer):
             integer;",
13       "tipo_retorno": "integer",
14       "parametros": {
```

```
15        "parReferencia": {
16          "tipo_passagem": "referencia",
17          "tipo": "integer"
18        },
19        "parCopia": {
20          "tipo_passagem": "valor",
21          "tipo": "integer"
22        }
23      },
24      "comandos": {
25        "While000": {
26          "linha": "while (parReferencia < parCopia) do",
27          "Atribuicao001": {
28            "linha": "parReferencia := parReferencia + 1"
29          }
30        },
31        "Atribuicao002": {
32          "linha": "funcao := parCopia"
33        }
34      }
35    }
36  },
37  "comandos": {
38    "Atribuicao003": {
39      "linha": "varGlobal := 2"
40    },
41    "Atribuicao004": {
42      "linha": "varGlobal2 := funcao(varGlobal, varGlobal + 2)"
43    }
44  }
45 }
```

Código 2.8: Json que representa o código 2.7.

2.3 O RENDERIZADOR

O programa renderizador é responsável por gerar a representação gráfica do fluxo de execução do programa analisado utilizando como entrada um documento Json. Para proporcionar independência de plataforma, este trabalho renderiza imagens em navegadores web, mais especificamente na tag canvas de um documento HTML. Essa tag permite desenhar retas e curvas através de uma biblioteca desenvolvida em Javascript ou Typescript.

Para representar o fluxo visualmente decidiu-se interpretá-lo como um tipo de grafo denominado “hierarquia”. Uma hierarquia é um grafo onde os vértices são colocados em camadas horizontais e as arestas ligam os vértices da camada n com os vértices das camadas $n + 1$ e $n - 1$. A vantagem desta abordagem é que existem algoritmos que melhoram a aparência do grafo ao trocar a posição dos vértices e minimizar a quantidade de cruzamentos entre as arestas. Dentre os algoritmos para desenhar hierarquias, este trabalho utilizou o algoritmo de (Sugiyama et al., 1981). Esse algoritmo foi escolhido por ser bem estudado, por conseguir resolver problemas mais complexos que o necessário para esse projeto e pela representação do grafo como uma hierarquia poder ser usada para representar o fluxo de um programa.

2.4 REVISÃO BIBLIOGRÁFICA

Assim como dito na Seção 2.3, o algortimo de (Sugiyama et al., 1981) já foi bem estudado. Um exemplo é em (Forster, 2005) que é feita uma possível melhoria para um dos processos do algoritmo de (Sugiyama et al., 1981). Essa melhoria se dá na etapa de reduzir o número de cruzamento de arestas da representação do grafo, a melhoria é dada ao adicionar a possibilidade de levar em conta restrições na ordem dos vértices nesse processo de reduzir o número de cruzamento de arestas. Existem também trabalhos que utilizam o algoritmo de (Sugiyama et al., 1981) para fazer representações gráficas diversas. Por exemplo em (Schwikowski et al., 2000) em que o algoritmo é usado para visualizar uma rede de interações entre proteínas em levedura. Também é usado em (Wongsuphasawat et al., 2018) para fazer a visualização do fluxo de dados em modelos de aprendizagem profunda.

3 CONCEITUAL

Este capítulo apresenta uma visão conceitual de cada uma das etapas envolvidas no processo para a geração da imagem pelo renderizador. A Seção 3.1 explica as funcionalidades da linguagem pascal suportadas pelo compilador. A Seção 3.2 explica como o compilador gera o arquivo Json a partir de um arquivo em Pascal. Por fim, a Seção 3.3 explica como o renderizador gera o grafo do fluxo de execução a partir do Json.

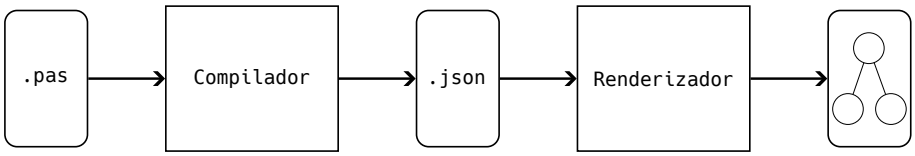


Figura 3.1: Fluxo do funcionamento da ferramenta desenvolvida.

Imagem que mostra cada uma das etapas do projeto, um arquivo feito na linguagem Pascal é usado de entrada para o compilador. Esse compilador gera um arquivo Json e o renderizador usa esse arquivo json para exibir uma árvore.

3.1 LINGUAGEM PASCAL

Pascal é uma linguagem de programação procedural desenvolvida por Niklaus Wirth no final da década de 1960 e publicada em 1970. Foi criado para promover uma programação estruturada e boas práticas de programação, sendo muito utilizada para o ensino de programação de computadores.

O algoritmo 3.1 é um exemplo de um programa na linguagem pascal, um programa com nome de euclides que tem as variáveis *a*, *b*, e *resultado*, com as sub-rotinas *gcd* e *passo*. Cada sub-rotina tem seus parâmetros e suas variáveis.

```
1  program euclides(input, output);
2  var a, b, resultado: integer;
3
4    function gcd(a, b: integer):integer;
5    var r:integer;
6
7      procedure passo;
8      begin
9        r:=a mod b;
10       a:=b;
11       b:=r;
12     end;
13   begin
14     while(a mod b <> 0) do
15       begin
16         passo;
17       end;
18     gcd := b;
19   end;
20
21 begin
22   read(a);
23   read(b);
```

```
24   resultado := gcd(a, b);
25   write(resultado);
26 end.
```

Código 3.1: Exemplo de programa escrito em Pascal

Como este trabalho tem um escopo de monografia de conclusão de curso, não haveria tempo para tratar todas as funcionalidades da linguagem. Por esta razão, este trabalho trata do seguinte subconjunto da linguagem Pascal:

- Declaração e atribuição a variáveis dos tipos: número inteiro (integer), booleano (boolean), caractere (char), e número real (real);
- Soma e subtração com inteiros e com números reais;
- Multiplicação e divisão inteira;
- Expressões numéricas e booleanas;
- Comandos condicionais do tipo if then else;
- Laços do tipo while do;
- Declaração e uso de funções e procedimentos respeitando níveis léxicos;
- Atribuição de expressões a variáveis e a retornos de funções;
- Passagem de parâmetros por valor e referência para funções e procedimentos;

3.2 O COMPILADOR

Conforme apresentado na Seção 2.1, um compilador é um programa que traduz um arquivo especificado em uma linguagem livre de contexto determinística para uma saída que normalmente é o código de um arquivo executável. Neste trabalho a saída do compilador é um arquivo Json, que contém as informações relevantes da estrutura do programa, por exemplo, comandos de repetição e comandos alternativos, símbolos (por exemplo, nomes de variáveis e procedimentos), tipos e escopo destes símbolos.

Este compilador utiliza uma tabela de símbolos para analisar a compatibilidade de operações aritméticas. Nesta tabela, símbolos são acrescidos e removidos conforme o seu escopo. Além da tabela de símbolos, o compilador utiliza uma tabela de comandos que inclui informações sobre alguns comandos, em especial comando de repetição e alternativas. Diferente da tabela de símbolos, a tabela de comandos não tem elementos removidos ao longo do processo de compilação. Essa segunda tabela é a base para geração do arquivo de saída do compilador que está no formato Json.

O lado esquerdo da figura 3.2 apresenta um programa pascal que tem um comando *while* com um comando *if* dentro dele. O lado direito apresenta o arquivo Json que representa seu fluxo.

```
program exemplo (input, output);
var
  n: integer;

begin
  read(n);
  while(n <> 1) do
  begin
    if (n mod 2 = 0) then
      n := n div 2
    else
      n := (n*3) + 1;
    write(n);
  end;
end.
```

```
{
  "variaveis": {
    "n": {
      "tipo": "integer"
    }
  },
  "comandos": {
    "Leitura000": {
      "linha": "read(n);"
    },
    "While001": {
      "linha": "while (n <> 1) do",
      "If002": {
        "linha": "if (n mod 2 = 0) then",
        "Then003": {
          "linha": "Then",
          "Atribuicao004": {
            "linha": "n := n div 2"
          }
        },
        "Else005": {
          "linha": "Else",
          "Atribuicao006": {
            "linha": "n := (n * 3) + 1"
          }
        }
      },
      "Escrita007": {
        "linha": "write(n);"
      }
    }
  }
}
```

Figura 3.2: Um código Pascal e sua representação no formato Json.

3.3 O RENDERIZADOR

O programa renderizador é o responsável por desenhar o fluxo de comandos no formato de hierarquia. A entrada do programa do renderizador é o arquivo Json gerado pelo compilador.

O lado esquerdo da Figura 3.3 apresenta o arquivo json usado de exemplo na seção anterior. O lado direito da figura apresenta o grafo que representa o fluxo de execução daquele programa, como gerado pelo renderizador. Os vértices do grafo indicam comandos como atribuições, chamadas de procedimentos, while, if, etc. As arestas do grafo representam o fluxo de execução entre comandos.

O renderizador coloca os vértices em camadas e também coloca alguns vértices que não são exibidos para impedir que haja cruzamento de arestas, algo que deixaria menos claro o fluxo da exibição dos comandos. Posteriormente os vértices têm suas posições horizontais alteradas visando minimizar o comprimento das arestas.

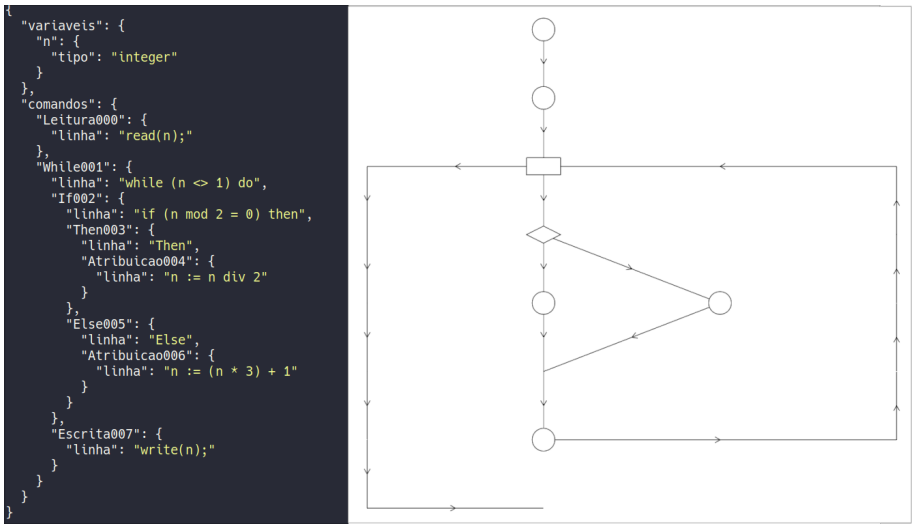


Figura 3.3: Representação visual a partir de um arquivo Json.

4 RENDERIZADOR

Este capítulo apresenta como gerar uma renderização com a interface criada, além de explicar como interpretar essa renderização. A Seção 4.1 explica como utilizar a interface web criada e a funcionalidade de cada componente da interface. Em seguida, a Seção 4.2 explica como entender o que cada componente da renderização gráfica significa no contexto do entendimento do fluxo de um código. A Seção 4.3 explica como foi utilizada parte do algoritmo em (Sugiyama et al., 1981) para fazer a renderização da forma desejada. E finalmente a Seção 4.4 explica como é feita a renderização do resultado do algoritmo de Sugiyama.

4.1 A INTERFACE WEB

Antes mesmo de se utilizar a interface web, é necessário ter um Json gerado pelo compilador para poder ser feita a renderização de um programa. O compilador deve ser executado em uma linha de comando com o comando ./compilador <entrada> onde <entrada> é o caminho para o programa pascal que se deseja ser compilar. Então, o compilador gera um arquivo <entrada>.json. Tendo um arquivo Json, é possível dá-lo como entrada para a página web.

A página web consiste em 3 seções. A primeira seção contém apenas o botão para o usuário selecionar o arquivo Json que deseja visualizar. Já a segunda seção consiste em informações sobre o programa, tendo 3 elementos. Uma linha com o nome da sub-rotina que está sendo exibida, uma linha que mostra qual linha do programa um vértice representa; essa linha é atualizada à medida que é passado o mouse por cima dos vértice na exibição da sub-rotina, e por fim, a tag canvas que mostra a exibição da sub-rotina como uma árvore. Por fim, a última seção é usada para o controle de qual sub-rotina é exibida na seção anterior. Essa última seção contém um botão para cada sub-rotina acessível pelo escopo da sub-rotina em exibição e um botão para exibir a sub-rotina que a tem em seu escopo.

4.2 A REPRESENTAÇÃO

Para representar as estruturas do fluxo de um código foi decidido formato geométrico diferente para cada vértice do grafo com base no que esse comando representa do código. A leitura do fluxo é feita de cima para baixo, com comandos que não alteram o fluxo sendo representados por círculos. Comandos *if* são representados por um losango, como na figura 4.2. A aresta mais à esquerda sendo o fluxo tomado caso a expressão do comando seja verdadeira, e a aresta mais à direita sendo o fluxo caso contrário. Comandos *while* são representados por um retângulo, como na figura 4.3. Seu fluxo é dado por executar a aresta para baixo do vértice, enquanto sua expressão for verdadeira; quando a expressão for falsa o fluxo segue pela aresta diretamente à esquerda do vértice do while.

4.3 O ALGORITMO DE SUGIYAMA

Como foi explicado na Seção 2.3, o problema de gerar representações visuais de grafos já foi bem estudado, e esse projeto utiliza o algoritmo de (Sugiyama et al., 1981) para o posicionamento dos vértices. O algoritmo é usado para gerar uma representação visual de um grafo de forma hierárquica. Para isso, o algoritmo é dividido em três etapas, a primeira gera a hierarquia do grafo, a segunda reduz o número de cruzamento de arestas do grafo ao alterar a ordem dos

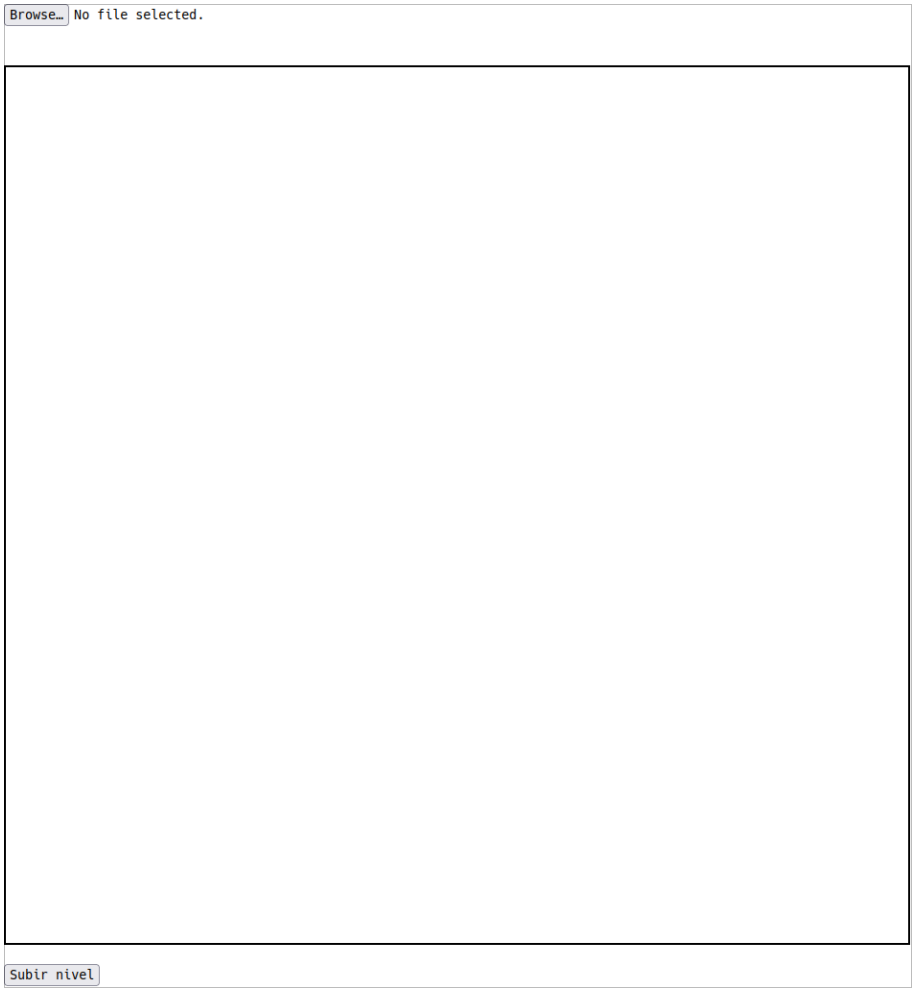


Figura 4.1: Figura do renderizador como descrito na Seção 4.1.

vértices em cada nível, e a terceira altera o posicionamento horizontal dos vértices em cada nível para reduzir o comprimento das arestas do grafo. Isso é feito sem alterar a ordem dos vértices. As duas primeiras etapas não são necessárias para os grafos gerados por este trabalho, pois é possível criar um grafo com a hierarquia desejada e sem cruzamentos a partir do arquivo Json gerado, por isso apenas a terceira etapa é usada.

A geração da hierarquia do grafo é dada pela ordem de execução de cada um dos comandos, tendo comandos executados antes em níveis visualmente mais acima de comandos executados depois. Para a redução do número de cruzamento de arestas o algoritmo (Sugiyama et al., 1981) altera a ordem dos vértices em um nível do grafo. Para manter a consistência da estrutura de cada tipo de comando não foi usada essa etapa do algoritmo. Pelos testes executados utilizando apenas as construções da linguagem Pascal que o compilador engloba, foi possível remover cruzamentos de arestas ao adicionar vértices ao grafo para controlar o caminho de arestas, vértices esses que não são exibidos na representação final, portanto são chamados de vértices fantasma. Essa construção sem cruzamento de arestas não seria possível na presença de comandos como *goto* ou *break*. Na presença desse comandos seria necessário utilizar o algoritmo para redução de arestas apresentado em (Forster, 2005) para se manter a ordem desejada dos vértices e reduzir o número de cruzamento de arestas.

A terceira etapa do algoritmo (Sugiyama et al., 1981) pode ser feita de duas formas. A primeira tem seu custo computacional inviável e trazendo resultados o melhor possível dado o objetivo de minimizar o comprimento de arestas no grafo. A figura 4.4 tem à sua esquerda

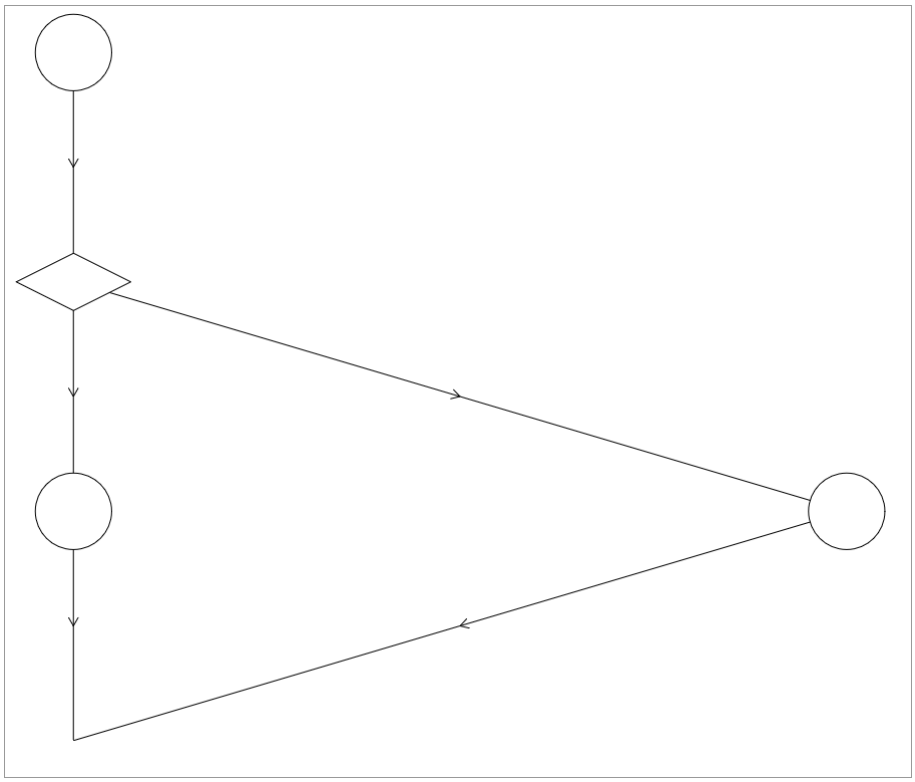


Figura 4.2: Representação visual de um comando if.

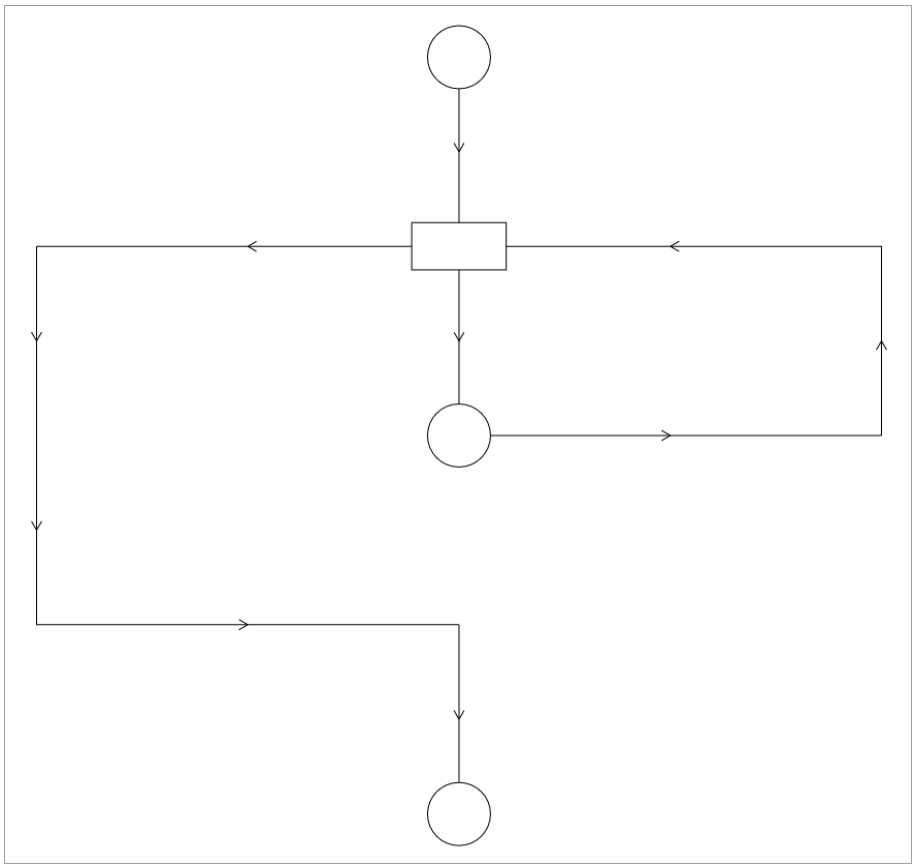


Figura 4.3: Representação visual de um comando while.

um grafo já no formato de hierarquia, e à direita tendo o objetivo de visualização a ser gerada. A segunda forma é feita iterativamente trazendo melhorias no posicionamento horizontal dos vértices de cada nível a cada iteração sem alterar a ordem que esses vértices estão representados. Cada iteração é feita ao analisar os vértices de dois níveis do grafo; apenas vértices de um dos níveis do grafo são alterados de posição. Cada vértice tem sua posição horizontal alterada de modo a ficar o mais próximo possível da média aritmética da posição horizontal dos vértices que têm uma aresta com o vértice analisado. A figura 4.5 tem na sua esquerda um nível antes de sua melhoria e na sua direita a movimentação feita pelos vértices após serem analisados os níveis representados na imagem.

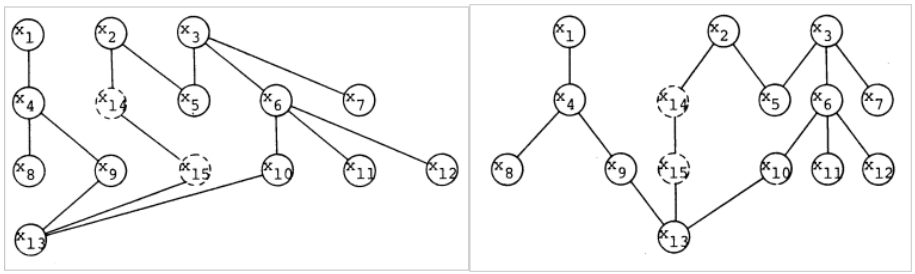


Figura 4.4: O objetivo do algoritmo Sugiyama.

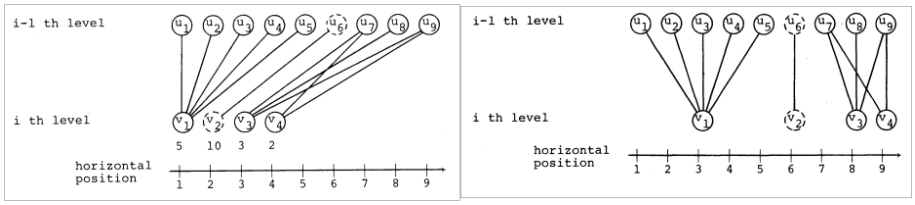


Figura 4.5: Melhoramento em uma iteração do algoritmo Sugiyama.

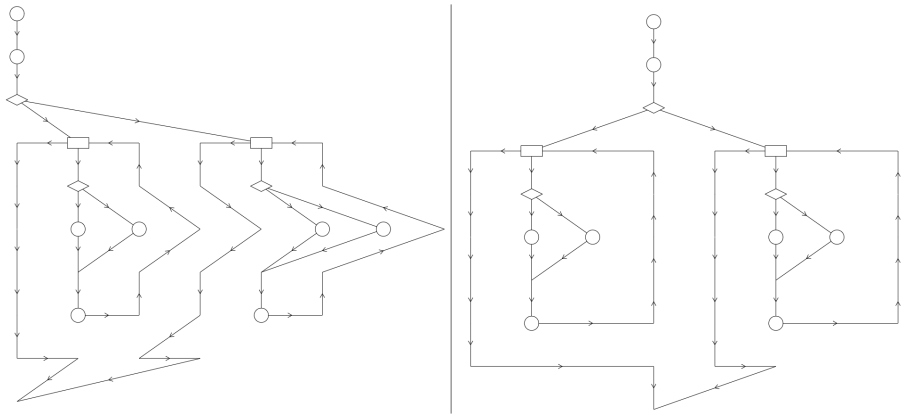


Figura 4.6: Efeito do algoritmo de Sugiyama na representação gráfica feita.

4.4 RENDERIZAÇÃO

Ao final do algoritmo de (Sugiyama et al., 1981) cada vértice terá as informações necessárias para sua renderização, que são: posição horizontal, posição vertical, forma (losango, retângulo, círculo, ou nada), e as arestas incidentes àquele vértice. Para fazer a renderização em si, criou-se

uma biblioteca em Typescript para fazer a renderização de losangos, círculos, retângulos, e arestas. A renderização é feita linha a linha, a renderização de cada linha é feita vértice a vértice, a renderização de cada vértice é feita pela renderização da sua forma em sua posição, e em seguia a renderização das arestas incidentes ao vértice, as arestas são feitas a partir da posição salva do vértice de origem em direção a posição salva no vértice de destino. Ao final desse processo a tag canvas do HTML terá o desenho completo do fluxo desejado.

5 CONCLUSÃO

Na correção de trabalhos de disciplinas que ensinam programação de computadores, o professor já tem ideia da estrutura lógica de um programa que ele entende como correto para aquele trabalho. Ao avaliar o trabalho, o professor busca pela estrutura lógica esperada, e então pela sequência de comandos. O reconhecimento dessa estrutura se torna mais difícil à medida que os programas ficam maiores, e uma ferramenta gráfica que mostra essa estrutura lógica de forma visual poderia ajudar. Esse texto descreveu a implementação de tal ferramenta.

No Capítulo 2 foram apresentados termos e conceitos necessários para o desenvolvimento da ferramenta. A Seção 2.1 apresentou o funcionamento de um compilador. A Seção 2.2 mostrou uma forma de representar a estrutura lógica de um programa de forma textual, seguindo o formato Json. A Seção 2.3 apresentou o que é necessário para gerar a representação visual a partir da representação textual gerada.

O Capítulo 3 explicou o que é feito em cada etapa do projeto, com cada seção apresentando uma etapa. A Seção 3.1 explicou qual a entrada da ferramenta criada, um programa pascal. A Seção 3.2 explicou como é gerado o arquivo Json usado pelo renderizador. Por fim, a seção 3.3 explicou como o renderizador gera a representação visual a partir do Json gerado pelo compilador.

O Capítulo 4 apresentou como usar a ferramenta usada e como é gerada a representação desejada para um fluxo. A Seção 4.1 explicou a interface criada. A Seção 4.2 explicou o que cada parte de uma representação significa. A seção 4.3 explicou como são calculadas as posições dos vértices na representação. Por fim, a Seção 4.4 explica como a renderização da representação é feita em si.

A partir do que foi desenvolvido pode se sugerir os seguintes trabalhos futuros: estender as funcionalidade do compilador, e adicionar informações à renderização. O compilador desenvolvido não trata todas as construções da linguagem Pascal, ao tratar todas as construções será necessário atualizar o renderizador para representar as novas construções que alteram o fluxo. Além disso o renderizador pode ser melhorado para trazer mais informações nas representações, por exemplo, substituir o vértice de uma chamada de sub-rotina pela representação do fluxo desssa sub-rotina.

Referências Bibliográficas

Forster, M. (2005). A fast and simple heuristic for constrained two-level crossing reduction. In Pach, J., editor, *Graph Drawing*, pages 206–216, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kowaltowski, T. (2018). Implementação de linguagens de programação. `https://www.ic.unicamp.br/~tomasz/ilp`. Acessado em 21/06/2025.

Schwikowski, B., Uetz, P., and Fields, S. (2000). A network of protein–protein interactions in yeast. *Nature Biotechnology*, 18(12):1257–1261.

Sugiyama, K., Tagawa, S., and Toda, M. (1981). Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125.

Wongsuphasawat, K., Smilkov, D., Wexler, J., Wilson, J., Mané, D., Fritz, D., Krishnan, D., Viégas, F. B., and Wattenberg, M. (2018). Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):1–12.