UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Henrique da Cunha Buss

LINGUAGEM DE PROGRAMAÇÃO IPE:

Programação puramente funcional para APIs REST

Florianópolis, Santa Catarina – Brasil
2023

Henrique da Cunha Buss

LINGUAGEM DE PROGRAMAÇÃO IPE:
Programação puramente funcional para APIs REST

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador(a): Maicon Rafael Zatelli, Dr.

Florianópolis, Santa Catarina – Brasil
2023

Henrique da Cunha Buss

LINGUAGEM DE PROGRAMAÇÃO IPE: Programação puramente funcional para APIs REST

Este(a) Trabalho de Conclusão de Curso foi julgado adequado(a) para obtenção do Título de Bacharel em Ciências da Computação, e foi aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 14 de junho de 2023.

Renato Cislaghi, Dr.
Coordenador(a) do Programa de
Graduação em Ciências da Computação

Banca Examinadora:

Maicon Rafael Zatelli, Dr.
Orientador(a)
Universidade Federal de Santa
Catarina – UFSC

Prof. Rafael Santiago, Dr.
Universidade Federal de Santa Catarina –
UFSC

Prof. Alvaro Junio Franco, Dr.
Universidade Federal de Santa Catarina –
UFSC

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Márcio e Valquiria, por sempre investirem na minha educação, e por sempre me apoiarem e incentivarem a conquistar tudo o que conquistei até agora e ainda conquistarei, mesmo quando precisei de um empurrão a mais.

Ao meu irmão, Matheus, por servir de modelo e me apresentar à maioria das coisas que me interessam, e que formaram grande parte da minha personalidade. Coisas essas que culminaram na minha escolha por cursar Ciência da Computação e realizar este trabalho.

À minha namorada, Kétrin, por me apoiar e me incentivar a continuar, e sempre estar do meu lado quando precisei. Às minhas tias, tios, primos e avós, por cultivarem a cultura da universidade, e aos meus amigos, que sempre fizeram com que me sentisse orgulhoso de chegar onde cheguei.

Finalmente, ao meu orientador, Maicon, por me apresentar a programação funcional, e a todos os meus professores da graduação, que ajudaram a construir o amor que tenho pela área.

RESUMO

Grande parte dos produtos comerciais necessitam de um *backend*. Este trabalho apresenta Ipe, uma linguagem de programação feita voltada especificamente para desenvolver sistemas *backend*, com APIs REST, seguindo o paradigma funcional. Com as garantias do paradigma funcional e da tipagem estática, Ipe busca ser uma linguagem simples e fácil de se usar, ao mesmo tempo em que tenta capturar erros em tempo de compilação, diminuindo erros em tempo de execução.

Palavras-chaves: Linguagem de programação. Programação funcional.

ABSTRACT

Many commercial products need a *backend*. This work presents Ipe, a programming language made specifically to develop *backend* systems, following the functional style. With the guarantees of the functional paradigm and static typing, Ipe seeks to be a simple and easy to use language, while trying to capture all errors at compile time, avoiding errors at runtime.

Keywords: Programming language. Functional programming.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
HTTP	*Hypertext Transfer Protocol*
JSON	*Javascript Object Notation*
REST	*Representational State Transfer*
AST	*Abstract Syntax Tree*
CRUD	*Create, Read, Update, Delete*
IOT	*Internet of Things*
OO	*Orientação a Objetos*
EBNF	*Extended Backus-Naur Form*
CLI	*Command Line Interface*

LISTA DE SÍMBOLOS

|> *Pipe* direito

<| *Pipe* esquerdo

SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos, os desenvolvedores de *software* têm sido atraídos cada vez mais ao paradigma de programação funcional. Em contraste com o paradigma de orientação a objetos, o paradigma funcional visa simplificar o modelo mental dos programadores, focando em funções puras, imutabilidade, e composição de funções (Scalfani (2022)). Grandes linguagens de programação orientadas a objetos também estão indo na direção da programação funcional, como Java, JavaScript, C#, e Python. Embora esses conceitos possam criar programas mais robustos e mais fáceis de testar, pode ser difícil para desenvolvedores acostumados com outros paradigmas a aprenderem esse novo modelo mental (HINSEN, 2009).

O paradigma funcional não é algo concreto - é um conjunto de ideias que podem ser adotadas em partes. Enquanto existem linguagens que implementam algumas ideias da programação funcional (como as mencionadas acima), outras tentam implementar o máximo possível - comumente chamadas de *linguagens puras*, ou *linguagens puramente funcionais*. Não existe uma definição exata para o que é uma linguagem puramente funcional, mas elas geralmente têm alguns recursos, como *pattern matching*, funções puras (sem efeitos colaterais, como acesso ao sistema de arquivos, ou mostrar uma mensagem na tela), tipagem forte e estática, e inferência de tipos. Alguns exemplos de linguagens puramente funcionais são *Haskell* (HUDAK, 1989) e *Elm* (CZAPLICKI, 2012).

Linguagens de programação geralmente são criadas para um propósito específico. Por exemplo, *Javascript* foi criada para melhorar a experiência dos usuários na *web*. *Elm* foi criada para simplificar a criação de aplicativos *web*, utilizando programação funcional para diminuir erros em tempo de execução, e aumentar a produtividade dos desenvolvedores. Neste trabalho, vamos explorar algumas das linguagens de programação utilizadas para criar aplicativos *backend* e, inspirados por *Elm*, vamos criar *Ipe*[1], uma linguagem de programação puramente funcional, desenvolvida especificamente para desenvolver aplicativos *backend*, com o objetivo de simplificar o desenvolvimento e diminuir a quantidade de erros em tempo de execução, além de diminuir a barreira de entrada a linguagens puramente funcionais.

1.1 OBJETIVOS

Com a devida contextualização, os objetivos deste trabalho são:

---

[1] *Elm* significa Olmo (uma árvore) em inglês. Ipê é uma árvore brasileira. Para simplificar o uso em inglês, vamos usar o nome *Ipe*, sem acento.

1.1.1 Objetivo Geral

Desenvolver Ipe, uma linguagem de programação puramente funcional, com foco em desenvolvimento de aplicações *backend* para a *web*, e com objetivo de diminuir a barreira de entrada a linguagens funcionais.

1.1.2 Objetivos Específicos

Para cumprir o objetivo geral, precisamos cumprir os seguintes objetivos específicos:

1. Definir os requisitos para uma aplicação *backend* com uma API REST, para servir de modelo.
2. Implementar esses requisitos em algumas linguagens de programação, procurando por padrões, boas práticas, e dificuldades.
3. Definir a linguagem *Ipe*.
4. Criar um compilador para *Ipe*, que possibilite gerar código que possa ser executado.
5. Implementar os requisitos definidos no primeiro item usando *Ipe*, para comparar com as implementações descritas no item 2.
6. Analisar e discutir os resultados obtidos.

1.2 ESCOPO DO TRABALHO

Por questões de limitação de tempo, a aplicação modelo será bem simples, e não necessariamente representará uma aplicação real. Além disso, *Ipe* é apenas uma prova de conceito. Não é o objetivo deste trabalho criar uma linguagem de programação completa, com otimizações, ferramentas, e bibliotecas extensas. O objetivo é explorar as ideias da programação funcional no desenvolvimento *backend*. Por conta disso, a linguagem *Ipe* não deve ser usada no mundo real, em projetos reais. Dito isso, ainda queremos representar um caso de uso comum para sistemas *backend*, e é por isso que temos o objetivo de construir uma API REST em Ipe.

O compilador de *Ipe* deve funcionar em sistema operacional macOS, onde vai ser desenvolvido, e pode não funcionar em outros sistemas.

1.3 MÉTODO DE PESQUISA

Para cumprir os objetivos propostos, o desenvolvimento é dividido em 3 fases, que são explicadas a seguir:

1.3.1 Trabalhos Relacionados

Para termos noção do que outras linguagens fazem para o desenvolvimento de aplicações *backend*, vamos explorar algumas linguagens de programação ranqueadas como mais populares em índices como o índice TIOBE... (2022), e analisar algumas linguagens que inspiraram ou influenciaram no desenvolvimento de *Ipe*.

1.3.2 Implementação

Nesta fase, vamos de fato implementar *Ipe* - seu *parser*, compilador e tudo o que for necessário para executar programas escritos em *Ipe*. Vamos discutir decisões de sintaxe, funções padrões, estrutura de projetos.

Mostraremos como usar *Ipe*, descrevendo estruturas de controle, sistema de tipos, estruturas de dados, funções.

1.3.3 Uso em Aplicações

Vamos desenvolver a aplicação modelo utilizando *Ipe*, além de outras pequenas aplicações de exemplo. Vamos discutir as vantagens e desvantagens de usar *Ipe* para desenvolver aplicações *backend*, e comparar os resultados obtidos com *Ipe* e os resultados obtidos com linguagens orientadas a objetos, e também com outras linguagens funcionais.

1.3.4 Avaliação dos resultados

Após implementar a linguagem e a aplicação base, também conduziremos sessões de entrevista onde coletaremos *feedback* de outros desenvolvedores para obter métricas de avaliação. Nessas sessões de entrevista, desenvolvedores serão convidados a desenvolver a mesma aplicação base em Ipe. Ao final das entrevistas, os convidados irão responder perguntas sobre vantagens e desvantagens de Ipe em relação a outras linguagens, e poderemos ter métricas de comparação.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, discutiremos a fundamentação teórica necessária para compreender o trabalho. Começaremos discutindo APIs REST na Seção 2.1, que são o objetivo inicial para programas escritos em Ipe. Depois, na Seção 2.2, discutiremos o processo de compilação.

2.1 APIS E APIS REST

Uma API (*Application Programming Interface*) é um termo usado para representar a interface de comunicação entre programas de computador. Por exemplo, a coleção de métodos públicos de uma classe em Java pode ser chamada de API, pois é o meio de comunicação entre o resto do programa e a classe.

Mais comumente, uma API é a face de um serviço web, ativamente escutando por requisições HTTP e as respondendo (MASSE, 2011), de forma que clientes (páginas web, aplicativos celulares) possam se comunicar com o serviço.

REST (*Representational State Transfer*) é um estilo de arquitetura para modelar APIs, de forma que diferentes APIs tenham um formato padrão. Geralmente, APIs usam o formato JSON (*JavaScript Object Notation*) para representar dados.

O foco inicial de Ipe é desenvolver APIs REST.

2.2 ANÁLISE DE CÓDIGO FONTE

Ipe será uma linguagem de programação de alto nível. Isso significa que, em algum momento, código Ipe (linguagem fonte), deve ser transformado para algo que computadores entendam (linguagem alvo). Esse processo se chama compilação, e é feito por um programa chamado compilador. No caso deste trabalho, a linguagem alvo é Javascript, por ser uma linguagem com ampla adoção no mercado (STACK. . . , 2022). Quando a linguagem alvo também é uma linguagem de alto nível, como é o caso de Javascript, o processo de compilação é chamado de transpilação.

Compiladores geralmente funcionam com uma série de etapas. Primeiramente, o código fonte é analisado e transformado em uma *AST* (*Abstract Syntax Tree*), de forma a garantir que o código fonte é válido de acordo com a definição léxica e sintática da linguagem fonte. Este processo é chamado de *parsing*. Em seguida, a *AST* é analisada para fazer a checagem de tipos (ou *type checking*, em inglês). Depois disso, opcionalmente, são aplicadas transformações no código. Por exemplo, o compilador pode realizar otimizações, ou remover código não utilizado. Por fim, a AST e o código fonte são usados para produzir código na linguagem alvo. Vamos discutir cada uma dessas etapas em maiores detalhes nas seções a seguir.

A análise de código fonte é a primeira etapa do processo de compilação, e se

baseia na *gramática* da linguagem. A gramática de uma linguagem é uma lista de regras que dita o que é válido na linguagem. Em qualquer uma das etapas, se o compilador encontrar um erro, ele deve abortar o processo de compilação e emitir uma mensagem de erro. A gramática de Ipe é explicada no Capítulo 4, e está disponível em sua íntegra no Apêndice A.

A análise de código fonte geralmente é dividida em três etapas, descritas a seguir.

2.2.1 Parsing

O *parsing* é a primeira etapa da análise de código fonte. Nesta etapa, transformamos código em linguagem fonte para uma estrutura que o compilador entenda. Normalmente, o código é representado em formato de árvore. Nesta etapa da compilação estão as análises léxica e sintática.

2.2.1.1 Análise léxica

A análise léxica é encarregada de dividir o código fonte em *tokens*. Um token é uma sequência de caracteres que possui um significado. Por exemplo, a palavra *if* é um token, assim como o símbolo *+*. A análise léxica é responsável por identificar esses tokens, e gerar uma lista deles.

2.2.1.2 Análise sintática

A análise sintática recebe como entrada a lista de tokens gerada pela análise léxica, e verifica se a sequência de tokens é válida de acordo com a gramática da linguagem. Nesse processo, os tokens são organizados em formato de árvore.

2.2.2 Análise semântica

A análise semântica usa os dados obtidos pelas etapas anteriores para verificar se o código fonte é válido de acordo com a semântica da linguagem. Por exemplo, é nessa etapa em que o compilador realiza a checagem de tipos, e verifica se todas as variáveis usadas no programa foram declaradas.

2.2.3 Transformações de código

A análise de código fonte gera dados sobre o programa fonte (tabelas e árvores que representam o programa). Durante o processo de compilação, é possível realizar transformações no código fonte, de forma a otimizar o programa, ou remover código não utilizado. Por simplicidade, Ipe não fará nenhuma otimização.

2.2.4 Geração de código

A etapa de geração de código é a etapa final do processo de compilação. Nessa etapa, o compilador gera o código alvo a partir dos dados gerados pelas etapas anteriores. No caso de Ipe, o código alvo é Javascript. Ou seja, vamos usar os dados obtidos nas etapas anteriores para traduzir o código fonte (escrito em Ipe) para código Javascript. Para isso, mapeamos cada instrução da linguagem Ipe para uma instrução equivalente em Javascript.

Como o objetivo da linguagem é ser uma linguagem de programação para *backend*, geraremos código compatível com o ambiente Node.js, por ser a plataforma mais popular para desenvolvimento de aplicações *backend* em Javascript (STACK. . . , 2022).

3 TRABALHOS RELACIONADOS

Neste capítulo, vamos explorar outras linguagens usadas para desenvolvimento *backend*, construindo uma aplicação simples em cada uma delas. O objetivo é comparar funcionalidades, padrões de projeto, sintaxe das linguagens, e outras características. Para selecionar as linguagens, vamos nos basear em índices de uso e satisfação de linguagens. Cada linguagem será apresentada em uma subseção, e os critérios de escolha das linguagens serão especificados na subseção que as apresenta.

3.1 APLICAÇÃO BASE

Para comparar as linguagens, vamos construir uma aplicação simples, que será usada como base para as comparações. A aplicação consiste em um sistema de cadastro de usuários (um simples *CRUD – Create, Read, Update, Delete*). A aplicação deve receber requisições HTTP para cadastrar, atualizar, deletar e listar usuários. Os dados devem ser enviados através do corpo da requisição, em formato JSON, seguindo a convenção REST. Cada usuário deve ter um id numérico (começando em 0, e aumentando em 1 a cada usuário cadastrado), nome e idade. A Tabela 1 apresenta os *endpoints* da aplicação, onde `/users/:id` significa uma rota que possui um número após `/users/`, como em `/users/1`. Este número equivale ao id numérico de um usuário.

Tabela 1 – *Endpoints* da aplicação base

Método HTTP	***Endpoint***	**Descrição**
GET	/users	Lista todos os usuários cadastrados
GET	/users/:id	Lista um usuário específico
POST	/users	Cria um novo usuário. O nome e a idade devem ser enviados no corpo da requisição
PUT	/users/:id	Atualiza um usuário específico. O nome e a idade devem ser enviados no corpo da requisição
DELETE	/users/:id	Deleta um usuário específico

3.2 LINGUAGENS RELACIONADAS

Nesta seção, vamos construir a aplicação base apresentada na Seção 3.1 em algumas linguagens de programação, a fim de comparar as linguagens.

Além de escolher uma linguagem, também precisaremos escolher *frameworks* para construir a aplicação. Para cada linguagem, vamos procurar por *frameworks* utilizando a pesquisa do GitHub, pesquisando **web framework language:*linguagem***, e escolhendo o resultado com maior número de estrelas.

3.2.1 Python

De acordo com TIOBE... (2022), Python é a linguagem de programação mais procurada no mundo, além de ter ganho o prêmio linguagem do ano nos anos 2007, 2010, 2018, 2020 e 2021 pelo mesmo índice.

Criada nos anos 80, é usada em diversas áreas, como ciência de dados, desenvolvimento web, desenvolvimento de jogos, entre outras. Python é uma linguagem de programação interpretada, de alto nível, de tipagem dinâmica e multi-paradigma (Van Rossum e Drake (2011)).

Os três maiores *frameworks* em termo de estrelas no GitHub são Django, Flask e FastAPI, nesta ordem. Como Django é um *framework fullstack* (ou seja, ele é um *framework* que abrange tanto o desenvolvimento *frontend* quanto o desenvolvimento *backend*, e não é usado para APIs REST), vamos usar o Flask. O Código 1 apresenta a aplicação base desenvolvida em Python com Flask.

Código 1 – Aplicação base em Python com Flask

```python
from flask import Flask, jsonify, request, Response

app = Flask(__name__)
users = {}
last_index = 0

@app.route('/users', methods=["GET"])
def index():
    return jsonify(users)

@app.route('/users/<int:user_id>', methods=["GET"])
def show(user_id):
    return jsonify(users[user_id])

@app.route('/users', methods=['POST'])
def create():
    global last_index

    body = request.get_json()
    last_index = last_index + 1
    users[last_index] = body

    return jsonify({'id': last_index})

@app.route('/users/<int:user_id>', methods=["PUT"])
def update(user_id):
```

```
27      body = request.get_json()
28      users[user_id] = body
29
30      return Response(status=200)
31
32  @app.route('/users/<int:user_id>', methods=["DELETE"])
33  def delete(user_id):
34      users.pop(user_id)
35
36      return Response(status=200)
```

Flask se baseia no uso de *decorators* para definir as diferentes rotas da aplicação. Isso faz com que seja rápido e fácil de escrever programas com Flask, mas também pode ser confuso para quem não está acostumado com o uso de *decorators* ou do *framework* em si.

3.2.2 C#

C# é a próxima linguagem comumente usada para aplicações *backend* depois de Python e Java (TIOBE... (2022)). C# foi escolhida ao invés de Java por serem duas linguagens bastante similares, e C# ser mais procurada, mais amada, e menos temida por desenvolvedores do que Java (STACK... (2022)).

C# foi desenvolvida por Anders Hejlsberg, na Microsoft, em 2000, com o intuito de ser uma linguagem simples, moderna, orientada a objetos e com segurança de tipos para a plataforma .NET (Hejlsberg *et al.* (2008)).

A plataforma .NET possui o *framework* ASP.NET, feito para desenvolver aplicativos web e serviços. Essa é a opção mais popular para desenvolvimento web com C#, e é a que vamos usar. O Código 2 apresenta parte da implementação da aplicação base em C# com ASP.NET, mostrando apenas a rota de atualizar um usuário.

Código 2 – Aplicação base em C# com ASP.NET

```
1  [Route("users")]
2  [ApiController]
3  public class UserController : ControllerBase
4  {
5      ...
6
7      [HttpPut("{id}")]
8      public async Task<IActionResult> UpdateUser(long id, UserInput
           input)
9      {
10         var user = await _context.FindAsync<User>(id);
```

```
11          if (user == null)
12          {
13              return NotFound();
14          }
15          user.Name = input.Name;
16          user.Age = input.Age;
17          _context.Entry(user).State = EntityState.Modified;
18          await _context.SaveChangesAsync();
19
20          return NoContent();
21      }
22
23      ...
24  }
```

Como C# é uma linguagem fortemente orientada a objetos, o *framework* ASP.NET faz forte uso classes, objetos e herança, além de também usar *decorators*. O corpo das requisições é obtido automaticamente como um parâmetro do método que responde à requisição, o que também pode ser confuso.

Por outro lado, o *framework* tem boas ferramentas de geração de código, que geram boa parte do código básico necessário para uma aplicação, o que pode agilizar o desenvolvimento inicial.

Um simples projeto necessita de vários arquivos de configuração (gerados automaticamente), que também podem ser confusos para quem não está acostumado com o *framework*.

3.2.3 Javascript

Javascript é a próxima linguagem comumente usada para aplicações *backend* depois de C# na tabela TIOBE... (2022), e é a linguagem alvo para Ipe.

Originalmente desenvolvida por Brendan Eich na Netscape em 1995 para adicionar interações em páginas web, Javascript é uma linguagem de script interpretada, de tipagem dinâmica e fracamente tipada, e orientada a objetos, com alguns aspectos de programação funcional. Javascript vem se tornando cada vez mais popular em outros meios, como o *backend*, dispositivos *Internet of Things* (*IOT*), microsserviços, aplicações desktop, entre outros. Por conta de sua popularidade, muitas bibliotecas, *frameworks* e ferramentas existem para auxiliar no desenvolvimento de aplicações Javascript. Notavelmente, existem esforços para adicionar tipagem estática, principalmente através do Typescript, que já é mais amado, mais procurado e menos temido do que o Javascript STACK... (2022).

Para desenvolver a aplicação base, usaremos a plataforma Node.js, que possibilita executar programas Javascript fora do navegador, e o *framework* Express. O

Código 3 apresenta a implementação da aplicação base em Javascript.

Código 3 – Aplicação base em Javascript com Express

```
1  const express = require("express");
2
3  const app = express();
4
5  const users = new Map();
6  let lastId = 0;
7
8  app.use(express.json());
9
10 app.get("/users", (req, res) => {
11   const allUsers = Array.from(users.values());
12
13   res.json(allUsers);
14 });
15
16 app.get("/users/:userId", (req, res) => {
17   const userId = parseInt(req.params.userId);
18   const user = users.get(userId);
19
20   res.json(user);
21 });
22
23 app.post("/users", (req, res) => {
24   const { name, age } = req.body;
25   lastId++;
26   users.set(lastId, { name, age, id: lastId });
27
28   res.json({ id: lastId });
29 });
30
31 app.put("/users/:userId", (req, res) => {
32   const userId = parseInt(req.params.userId);
33   const { name, age } = req.body;
34   users.set(userId, { name, age, id: userId });
35
36   res.json({ id: userId });
37 });
38
39 app.delete("/users/:userId", (req, res) => {
```

```
40   const userId = parseInt(req.params.userId);
41   users.delete(userId);
42
43   res.json({ id: userId });
44 });
45
46 app.listen(3000, (err) => {
47   if (err) {
48     console.log(err);
49   }
50
51   console.log("Server is running on port 3000");
52 });
```

O *framework* Express usa algumas ideias de programação funcional, como funções de alta ordem e funções anônimas, o que gera um código bastante legível e expansível. Além disso, tudo é bastante explícito, sem *decorators* ou geração de código, o que também melhora a legibilidade, com baixo custo em velocidade de desenvolvimento.

A abstração de rotas é feita através de funções, que recebem como parâmetros um objeto representando a requisição HTTP e outro representando a resposta. Uma abstração melhor seria receber apenas a requisição como parâmetro, e retornar uma resposta.

3.2.4 Elixir

Elixir é a segunda linguagem mais amada, mais procurada e menos odiada, perdendo apenas para Rust (STACK... (2022)). Rust não foi escolhido por que a abordagem dos *frameworks* em Rust são parecidas com as abordagens já vistas nas linguagens anteriores. Outro ponto para escolher Elixir é o fato de Elixir ser uma linguagem funcional, ou seja, uma linguagem diferente das outras já vistas, e similar a Ipe.

Elixir é uma linguagem de programação funcional, concorrente e de tipagem dinâmica, desenvolvida por José Valim em 2012. Elixir é compilada para a máquina virtual BEAM, que é concorrente e tolerante a falhas, e também é usada pela linguagem Erlang.

Para desenvolver a aplicação base, usaremos o *framework* Phoenix.

O *framework* Phoenix se aproveita bastante da geração automática de código e da funcionalidade de *macros* do Elixir, o que gera um código mais conciso, mas muito mais difícil de entender. A aplicação inteira foi gerada automaticamente, o que facilita e agiliza o início de um projeto. Porém, isso dificulta na manutenção e compreensão

do código.

Como a ferramenta gera bastante código, apenas algumas partes são apresentadas a seguir.

O Código 4 apresenta a definição das rotas da aplicação, que permite definir funções a serem executadas antes de cada rota, além do agrupamento de rotas.

Código 4 – Definição das rotas da aplicação em Elixir com Phoenix

```
pipeline :api do
  plug(:accepts, ["json"])
end

scope "/users", UserCrudWeb do
  pipe_through(:api)
  get("/", UserController, :index)
  get("/:id", UserController, :show)
  post("/", UserController, :create)
  put("/:id", UserController, :update)
  delete("/:id", UserController, :delete)
end
```

O Código 5 apresenta a implementação do método de criação de usuários. O uso do operador |> (lê-se *pipe*) permite a composição de funções, o que gera um código mais conciso, e descreve uma sequência de transformações de dados, um dos pilares da programação funcional. Também podemos ver que a função recebe um mapa como parâmetro, que representa o corpo da requisição HTTP.

Código 5 – Função de criação de usuário em Elixir com Phoenix

```
def create(conn, %{"user" => user_params}) do
  with {:ok, %User{} = user} <- Accounts.create_user(user_params) do
    conn
    |> put_status(:created)
    |> put_resp_header("location", Routes.user_path(conn, :show,
   user))
    |> render("show.json", user: user)
  end
end
```

3.2.5 Haskell

Haskell é a linguagem número 50 na tabela TIOBE... (2022), e foi escolhida por ser considerada a principal linguagem funcional hoje em dia.

Haskell é uma linguagem de programação de propósito geral, puramente funcio-

nal, que inclui funções de alta ordem, tipos polimórficos, inferência de tipos e tipagem estática (Hudak (1989)).

O *framework* escolhido foi o IHP, que, similarmente ao Phoenix do Elixir, usa ferramentas de geração de código para gerar a maior parte do código necessário de uma aplicação. Ao invés de usar *macros*, o IHP usa tipos polimórficos e outros recursos da linguagem para diminuir repetição de código. Haskell é famosa por ser difícil de aprender e usar, e o uso extensivo de recursos avançados da linguagem tornam o código gerado pelo IHP difícil de entender.

O Código 6 apresenta a implementação do método de criação de usuários. Em Haskell, também é comum vermos a estrutura de composição de funções (com o operador `|>`), assim como vimos em Elixir.

Código 6 – Função de criação de usuário em Haskell com IHP

```
1  action CreateUserAction = do
2      let user = newRecord @User
3      user
4          |> buildUser
5          |> ifValid \case
6              Left user -> render NewView { .. }
7              Right user -> do
8                  user <- user |> createRecord
9                  setSuccessMessage "User created"
10                 redirectTo UsersAction
```

3.3 CONCLUSÕES DOS EXPERIMENTOS

Agora que experimentamos diferentes linguagens, com diferentes paradigmas, podemos recolher aprendizados e conclusões sobre o que foi visto. Nesta seção, discutiremos alguns pontos vistos como importantes para a linguagem Ipe. Além de analisar cada ponto nas linguagens vistas anteriormente, vamos discutir as lições aprendidas e usadas para o desenvolvimento de Ipe.

A Tabela 2 apresenta uma comparação entre as principais características das linguagens vistas, além do que teremos em Ipe. As subseções seguintes discutem alguns aspectos gerais das linguagens em maiores detalhes.

Podemos ver na Tabela 2 que Javascript possui um alto nível de temor (38,54%). Isso provavelmente se dá pelo fato do seu sistema de tipos ser dinâmico e fraco, pois, na mesma pesquisa, o nível de temor de Typescript é 12 pontos percentuais menor (26,54%). Haskell é extremamente temida, mas ao mesmo tempo, é a 16ª linguagem mais desejada. Estima-se que isso seja culpa da grande complexidade da linguagem, e sua curva de aprendizado íngreme – muitas pessoas querem conhecer a programação

Tabela 2 – Principais características das linguagens analisadas e de Ipe

Linguagem	**Paradigma principal**	**Tipagem**	**Nível de temor (STACK. . . , 2022)**
Python	OO	Dinâmica	32,66%
C#	OO	Estática	36,61%
Javascript	OO com protótipos e elementos funcionais	Dinâmica	38,54%
Elixir	Funcional	Dinâmica	24,54%
Haskell	Puramente funcional	Estática	43,56%
Ipe	Puramente funcional	Estática	–

funcional, mas encontram grande dificuldade em aprender Haskell. Elixir está apenas uma posição acima de Haskell no índice de linguagens mais desejadas. Porém, como é uma linguagem funcional com uma barreira de entrada menor, é a segunda linguagem mais amada e menos temida. Ipe almeja ser uma linguagem puramente funcional, com tipagem forte e estática, como Haskell, mas com uma curva de aprendizado mais suave, como Elixir, Javascript e Python. Desta maneira, desenvolvedores poderão conhecer a programação funcional fortemente tipada com uma barreira de entrada menor. Afinal, um dos objetivos de Ipe é ajudar a ensinar programação funcional com um bom sistema de tipos.

3.3.1 Simplicidade e funcionalidades

Linguagens mais maduras ou mais antigas tendem a ter várias funcionalidades, e cada funcionalidade pode ter mais de uma forma de ser usada. Por exemplo, em Javascript uma variável pode ser definida com `var`, `let` ou `const`, e funções podem ser criadas com a palavra reservada `function` ou em formato de *arrow function*. Isso pode ser confuso para quem está aprendendo a linguagem, além de gerar divisões entre os usuários da linguagem.

Outro exemplo de alta complexidade é Haskell, que possui inúmeras extensões de linguagem, que servem para melhorar recursos da linguagem, ou adicionar novos recursos. Pode ser difícil mudar de uma base de código que usa certas extensões de linguagem para outra base de código que não as usa. Mesmo sendo a mesma linguagem, a forma de usá-la pode ser extremamente diferente em cada base de código.

Como Ipe é uma linguagem específica para aplicações *backend*, podemos montar uma linguagem mais simples, com menos funcionalidades. Assim, a curva de aprendizado deve ser menor, e a linguagem deve ser mais fácil de usar. Além disso, código Ipe deve ser padronizado – cada tarefa deve ter uma única forma de ser feita –, de forma que não leve muito tempo para entender um projeto novo, vindo de outro projeto. Neste sentido, podemos utilizar de alguns pontos da filosofia do Python:

- *Explicit is better than implicit.*
- *There should be one– and preferably only one –obvious way to do it.*

(PETERS, 2010)
Que, em tradução livre, significam:

- *Explícito é melhor do que implícito.*
- *Deveria existir uma– e de preferência apenas uma –maneira óbvia de se fazer algo.*

Também notamos que o uso de *macros*, *decorators* e tipos polimórficos dificulta a compreensão do código, pois gera mais código implícito. Ipe tem o objetivo de ser uma linguagem extremamente explícita – mesmo que isso custe mais linhas de código para usuários da linguagem.

3.3.2 Operadores

A composição de funções através de *pipelines* com o operador `|>` produz código conciso que expressa muito bem a ideia de transformações sucessivas de dados, um dos focos da programação funcional. Podemos nos tentar a criar operadores arbitrários, que podem ser usados para resolver problemas específicos (por exemplo, um operador `.+` que faça soma de matrizes). Porém, isso pode afetar bastante a legibilidade do código, gerando símbolos imprevisíveis, específicos a uma pequena parte de uma única aplicação. Haskell permite a criação de operadores arbitrários, o que, em muitos casos, produz código extremamente difícil de se entender. Em Ipe, iremos evitar a criação de operadores arbitrários, dando preferência a funções nomeadas. Os únicos operadores serão definidos pela linguagem em si, e serão os mesmos em todos os programas Ipe. Todos os operadores disponíveis em Ipe estão listados na Tabela 3.

3.3.3 Ferramentas externas

Em quase todos os exemplos que analisamos, foi necessário utilizar ferramentas externas para auxiliar no desenvolvimento do programa. O principal exemplo são as ferramentas de geração de código, presentes no ASP.NET, Phoenix e IHP. Enquanto essas ferramentas podem ser extremamente úteis, o objetivo de Ipe é ser extremamente simples de usar. Portanto, a única ferramenta que precisará ser instalada para rodar programas Ipe será o compilador da linguagem.

Ao desenvolver as aplicações, também foi notável as ferramentas de auxílio em desenvolvimento nos editores de código. Linguagens com tipagem dinâmica têm maior dificuldade em sugerir compleções de código, e ferramentas como *IntelliSense*. A exceção é Javascript, pois os editores usam informações de tipo do *Typescript* por

baixo do panos.

No futuro, o compilador de Ipe pode cumprir o papel de algumas dessas ferramentas, como formatadores automáticos e *linters*, além de disponibilizar dados que podem ser usados por outras ferramentas, como editores de código.

4 ESPECIFICAÇÃO DA LINGUAGEM

Este capítulo tem o objetivo de definir as características da linguagem Ipe, como sintaxe e semântica. As características serão demonstradas principalmente através de código Ipe, para exemplificar o uso real, seguidos por sua definição sintática. Para definir a sintaxe de Ipe, usaremos a notação EBNF (*Extended Backus-Naur Form*), como descrito em (WIRTH, 1996). Por simplicidade, algumas definições são omitidas neste capítulo. Além disso, como iremos fazer o nosso próprio *parser*– ao invés de usar um gerador automático –a gramática não será otimizada– as ferramentas que usaremos para construir o *parser* fazem isso por nós –, e alguns detalhes não serão apresentados, como espaços em branco e comentários de bloco entre tokens. O Apêndice A contém a definição sintática completa da linguagem.

4.1 COMENTÁRIOS

Ipe define três tipos de comentários, que são ignorados pelo compilador. O primeiro tipo é o comentário de linha, que começa com `//`, e vai até o final da linha. O segundo tipo é o comentário de bloco, que começa com `/*` e termina com `*/`, podendo conter múltiplas linhas, ou terminar na mesma linha. O último tipo de comentário se chama comentário de documentação, e é similar ao comentário de bloco, mas começa com `/|*`, e deve ser colocado na linha acima de uma definição, ou no começo do arquivo, logo após a declaração de módulo. Comentários de documentação têm a finalidade de documentar código, e serão usados principalmente por futuras ferramentas de análise de código. Por exemplo, ao manter o cursor do mouse sobre uma função em um editor de texto, o editor pode mostrar o comentário de documentação. No Código 7 são demonstrados alguns exemplos de comentários. No Código 8 são apresentados os símbolos não terminais que definem comentários.

Código 7 – Exemplos de comentários

```
1 // Comentário de linha
2
3 /* Comentário de bloco
4    Que pode ter várias linhas
5 */
6
7 /|* Comentário de documentação
8     Aqui, podemos descrever uma função ou um módulo
9 */
```

Código 8 – Definição de comentários em EBNF

```
line comment   ::=  "//", {? qualquer caractere ?} - "\n", "\n";
block comment  ::=  "/*", {? qualquer caractere ?} - "*/", "*/";
doc comment    ::=  "/*|", {? qualquer caractere ?} - "*/", "*/";
```

4.2 MÓDULOS

Programas escritos em Ipe são organizados em módulos. Cada módulo representa um arquivo, que começa com letra maiúscula e termina com a extensão `.ipe` (por exemplo, `Main.ipe`). Dentro de cada módulo, podem existir várias definições (mas ao menos uma), e cada módulo pode exportar várias funções (cada módulo deve exportar ao menos uma definição). Falaremos mais sobre definições na Seção 4.3. Para declarar um módulo, a primeira linha do arquivo deve ser no formato `module NomeDoModulo exports <lista de definições exportadas>`. Módulos dentro de pastas devem conter os nomes das pastas, separados por pontos. Por exemplo, o arquivo `Pasta1/Pasta2/Modulo.ipe` deve conter a linha `module Pasta1.Pasta2.Modulo exports <lista de definições exportadas>`. A seguir, demonstramos alguns tipos de definição de módulo.

Código 9 – Exemplo de módulos

```
1 module Main exports [ main ]
```

No Código 9, o módulo `Main` reside no arquivo `Main.ipe`, e exporta a definição chamada `main`.

Código 10 – Exemplo de módulo em subpasta que exporta várias definições

```
1 module Pasta1.Pasta2.Modulo exports
2     [ funcaoA
3     , tipoA
4     , funcaoB
5     , funcaoC
6     ]
```

No Código 10, o módulo `Pasta1.Pasta2.Modulo` reside no arquivo `Modulo.ipe`, que está dentro da pasta `Pasta2`, que está dentro da pasta `Pasta1`. O módulo exporta as definições `funcaoA`, `funcaoB`, `funcaoC` e `tipoA`. A lista de exportações pode percorrer múltiplas linhas, e a ordem em que as definições aparecem não importa.

Código 11 – Exemplo de módulo com comentário de documentação

```
1 module ModuloComComentario exports [ funcaoA ]
2
3 /|* Comentário de documentação
```

```
4     Aqui podemos descrever o módulo e suas definições
5 */
```

No Código 11, o módulo `ModuloComComentario` possui um comentário de documentação, que pode ser usado por ferramentas para melhorar a exploração do código.

A vantagem de organizar o código em módulos é que módulos podem importar outros módulos, através da palavra-chave `import`. Isso é demonstrado no Código 12.

Código 12 – Exemplo de importação de módulos

```
1 module Main exports [ main ]
2
3 /|* Os comentários de documentação vêm antes das importações
4 */
5
6 import ModuloA
7 import Pasta1.Pasta2.ModuloB as ModuloB
```

No Código 12, também podemos ver que é possível importar um módulo com um apelido, usando a palavra-chave `as`. Isso é útil quando um módulo possui um nome muito longo, ou quando outro nome faz mais sentido no contexto do módulo atual. Quando um módulo é importado com apelido, seu nome original não pode mais ser usado para se referir às definições daquele módulo.

O Código 13 mostra a definição de módulos em EBNF.

Código 13 – Definição de módulos em EBNF

```
upper identifier  ::=  uppercase letter, {letter | digit | "_"};
module name       ::=  {upper identifier, "."}, upper identifier;
exported item     ::=  letter, {letter | digit | "_"};
export list       ::=  "[", {exported item}-, "]";
module decl       ::=  "module", module name, "exports", export list;
import decl       ::=  "import", module name, [ "as", module name ];
module            ::=  module decl, [doc comment], {import decl};
```

4.3 DEFINIÇÕES

Nesta seção, discutiremos os dois tipos de definições em Ipe: tipos e funções. Definições são itens definidos a nível de módulo, ou seja, podem ser exportadas para serem usadas em outros módulos.

4.3.1 Tipos

Ipe possui um sistema de tipos muito parecido com o de Elm, e tem o objetivo de ser o mais simples possível, sem abandonar expressividade. Por isso, existem apenas três tipos primitivos: `Number`, `String` e `Never`, que representam números (inteiros e reais), cadeias de caracteres, e valores que nunca podem ser construídos respectivamente. Números podem ter um ponto, para separar as casas decimais. `String`s são definidas entre aspas simples.

Além disso, existe o tipo `Record`, que representa uma coleção de pares chave-valor pré-definidos. `Records` são construídos entre chaves, e cada par chave-valor é separado por vírgula. As chaves e seus valores são separados por dois pontos. Para acessar um elemento de um `Record`, basta usar a sintaxe `record.chave`.

Em Ipe, todos os valores e todas as definições são tipados (possuem um tipo). Para demonstrar os tipos dos valores, usamos o caractere `:`, seguido pelo tipo. O Código 14 demonstra a anotação de tipos, enquanto o Código 15 mostra as regras em EBNF para declaração dos tipos primitivos (`Number`, `String` e `Record`).

Código 14 – Exemplo de anotação de tipos

```
5 : Number
3.14 : Number
'abc' : String
{ a: 1, b: 'abc' } : { a : Number, b : String }
```

Código 15 – Definição dos tipos primitivos em Ipe em EBNF. A regra `expression` está descrita no Apêndice A

```
number      ::=  ["-"], {digit}-, [".", {digit}-];
string      ::=  "'", {? qualquer caractere ?} - ("\n"| "'"), "'";
field decl  ::=  lower identifier, ":", expression;
record      ::=  "{", [field decl, {",", field decl}], "}";
```

A verdadeira força do sistema de tipos de Ipe está na definição de novos tipos. Ipe permite três modalidades de novos tipos:

- Apelidos: são usados para dar um novo nome a algum outro tipo qualquer. São úteis para abreviar tipos compridos, como `Record`s com vários campos. Para definir um novo apelido, usamos `type alias`, como no Código 16.
- Uniões: são usados para definir novos tipos, e são definidos com `type union`. Discutiremos mais sobre uniões no Código 17.
- Opacos: são iguais às uniões, mas suas variantes não são exportadas para outros módulos, ou seja, tipos opacos só podem ser construídos e manipulados no

módulo onde foram definidos. Tipos opacos são úteis para definir uma interface específica ao redor de tipos, pois seus módulos podem exportar funções que manipulam os tipos de forma específica, e é impossível que eles sejam manipulados de qualquer outra maneira. São definidos com `type opaque`, como mostra o Código 18.

Código 16 – Exemplo de apelido de tipo

```
1 type alias Pessoa = { nome : String, idade : Number }
```

Um apelido de tipo serve simplesmente para abreviar um tipo. Outras definições poderiam se referir ao tipo `{ nome: String, idade : Number }` simplesmente como `Pessoa`.

Código 17 – Exemplo de união de tipos

```
1 type union Permissao =
2     | ConvidarNovosUsuarios
3     | RemoverUsuarios
4
5 type union Usuario =
6     | Admin { nome : String, permissoes : List Permissao }
7     | UsuarioCadastrado { nome : String, email : String }
8     | UsuarioAnonimo
```

Podemos ver que as uniões de tipos são definidas por uma coleção de uma ou mais etiquetas (*tags*, em inglês), e cada etiqueta pode ter um `Record` como argumento. Veremos mais como utilizar uniões de tipos na Seção 4.4.

Código 18 – Exemplo de tipos opacos

```
1 type opaque Permissao =
2     | ConvidarNovosUsuarios
3     | RemoverUsuarios
4
5 type opaque Usuario =
6     | Admin { nome : String, permissoes : List Permissao }
7     | UsuarioCadastrado { nome : String, email : String }
8     | UsuarioAnonimo
```

Tipos opacos são definidos e funcionam exatamente como uniões de tipos, mas suas etiquetas não são exportadas para outros módulos.

O nome de um tipo é sempre escrito com a primeira letra maiúscula, para facilitar a distinção entre tipos e funções.

4.3.1.1 Parâmetros de tipos

Ipe permite definir tipos que recebem outros tipos como parâmetro. É assim que estruturas como `Lists` e `Dicts` funcionam – elas recebem parâmetros de tipo para que possam ser genéricas. Todos os tipos mencionados acima (apelidos, uniões e tipos opacos) podem receber parâmetros de tipo. Os nomes dos parâmetros de tipos podem ser qualquer identificador que comece com uma letra minúscula. O Código 19 mostra como definir um tipo com parâmetro.

Código 19 – Exemplo de como definir um tipo com parâmetro

```
1 type opaque Pilha elemento =
2     | Pilha { itens : List elemento }
3
4 type alias PilhaDeNumeros = Pilha Number
```

O tipo `Pilha` recebe um parâmetro de tipo chamado `elemento`. Dentro da definição de `Pilha`, podemos utilizar `elemento` para nos referirmos ao tipo que foi passado como parâmetro. Com esse mecanismo, podemos definir tipos genéricos. O tipo `PilhaDeNumeros` representa uma pilha especificamente de números. O Código 20 mostra a gramática para a definição de apelidos, uniões e tipos opacos.

Código 20 – Definição de novos tipos em EBNF

```
custom type        ::=  {upper identifier, "."}, upper identifier, {any type};
any type           ::=  custom type
                     |  lower identifier
                     |  record annotation;
record ann field   ::=  lower identifier, ": ", any type;
record annotation  ::=  "{", [record ann field, {",", record ann field}] "}";
alias              ::=  upper identifier, parameters, "=", any type;
union              ::=  upper identifier, parameters, "=", {constructor}-;
constructor        ::=  "|", upper identifier, [record annotation];
new type           ::=  "type alias", alias
                     |  "type union", union
                     |  "type opaque", union;
```

4.3.2 Funções

Enquanto tipos são usados para modelar modelos e domínios, funções são usadas para manipular esses modelos e domínios. Juntos, funções e tipos são a base da programação funcional. Diferentemente de outras linguagens, Ipe possui uma única forma de definir funções, como mostra o Código 21.

Código 21 – Exemplo de como definir uma função

```
1 \argumento1 argumento2 ->
2     soma = argumento1 + argumento2;
3     dobro = soma * 2;
4     dobro * 4
```

Na linha 1, definimos os argumentos da função (`argumento1` e `argumento2`). Argumentos podem receber qualquer identificador que comece com uma letra minúscula. Nas linhas 2 e 3, definimos duas variáveis locais (`soma` e `dobro`). Podemos incluir quantas declarações quisermos no corpo da função, mas as variáveis declaradas devem ter nomes únicos (não podem repetir o nome de outra variável). A expressão da última linha define o retorno da função (`dobro * 4`). Todas as linhas do corpo de uma função, com exceção da última, devem ser terminadas em ponto e vírgula (;).

Esta sintaxe de definição de funções serve tanto para funções anônimas quanto para funções nomeadas. Para definir uma função nomeada, basta dar um nome à função, como mostra o Código 22.

Código 22 – Exemplo de como definir uma função nomeada

```
1 somaEMultiplicaPor8 : Number -> Number -> Number
2 somaEMultiplicaPor8 =
3     \argumento1 argumento2 ->
4         soma = argumento1 + argumento2;
5         dobro = soma * 2;
6         dobro * 4
```

O Código 22 também mostra como definir o tipo de uma função. Basta repetir o nome da função, seguido dos tipos dos argumentos e do tipo de retorno, separados por `->`. O último tipo é o tipo de retorno.

Incluir assinaturas de funções é opcional, pois Ipe consegue inferir os tipos de qualquer programa válido. Porém, adicionar as assinaturas de tipo pode ajudar a produzir melhores erros de compilação, e também pode ajudar a documentar o código.

O último tipo de definição é o de constantes, que, na verdade, são apenas funções que recebem zero argumentos. O Código 23 mostra como definir uma constante.

Código 23 – Exemplo de como definir uma constante

```
1 resposta : Number
2 resposta = 42
```

Para executar uma função, passamos os argumentos, separados por espaço (sem parênteses e sem vírgulas), como mostra o Código 24.

Código 24 – Exemplo de como chamar uma função

```
numero96 : Number
numero96 = somaEMultiplicaPor8 4 8
```

Podemos usar parênteses para definir expressões maiores, como mostra o Código 25.

Código 25 – Exemplo de como chamar uma função com uma expressões maiores

```
numero96 : Number
numero96 = somaEMultiplicaPor8 (2 + 2) ((1 + 1) * 4)
```

Também é possível passar funções como argumento, como demonstrado no Código 26.

Código 26 – Exemplo de como passar uma função como argumento

```
type List element =
    | Empty
    | Node { head : element, tail : List element }

map : (element -> mappedElement) -> List element -> List
   mappedElement
map =
    \mapFn list ->
        match list with
            | Empty -> Empty
            | Node node ->
                Node { head = mapFn node.head, tail = map mapFn
                   node.tail }

soma2 : Number -> Number
soma2 =
    \x -> x + 2

soma2EmLista : List Number -> List Number
soma2EmLista =
    \lista -> map soma2 lista

soma2EmListaAnonimo : List Number -> List Number
soma2EmListaAnonimo =
    \lista -> map (\x -> x + 2) lista
```

O Código 27 mostra a definição sintática de funções em Ipe. Podemos ver que funções são apenas estruturas que recebem argumentos, e retornam uma expressão.

Código 27 – Definição de funções em EBNF, onde `tl` significa *top level*

```
fn args         ::=  {lowercase identifier};
function        ::=  "\", fn args, "->", {attribution}, expression;
tl annotation   ::=  lowercase identifier, ":", {any type, "->"}, any type;
attribution     ::=  lowercase identifier, "=", expression, ";";
tl left         ::=  [doc comment], [tl annotation], lowercase identifier;
tl value        ::=  tl left, "=", expression;
```

4.4 PATTERN MATCHING

Pattern matching é o que possibilita que usemos todos os poderes dos tipos que definimos na Subseção 4.3.1. Com *pattern matching*, podemos extrair dados de tipos complexos. Diferente da maioria das linguagens, Ipe não possui instruções `if`. Em vez disso, podemos usar *pattern matching*. O Código 28 mostra como podemos fazer o mesmo que uma instrução `if` usando a instrução `match ... with`.

Código 28 – Substituto de if usando *pattern matching*

```
1  type union Bool =
2      | True
3      | False
4
5  greet : Bool -> String
6  greet =
7      \isHuman ->
8          match isHuman with
9              | True -> "Hello, human!"
10             | False -> "Beep boop"
```

O Código 29 mostra uma versão mais idiomática, que usa um tipo específico para o contexto da aplicação.

Código 29 – Maneira idiomática da função greet

```
1  type union Being =
2      | Human { name : String }
3      | Robot
4
5  greet : Being -> String
6  greet =
7      \being ->
8          match being with
9              | Human human -> String.concat [ "Hello, ", human.name ,
                   "!" ]
```

```
10              | Robot -> "Beep boop"
```

Podemos usar a instrução `match ... with` com qualquer tipo de dado! Por exemplo, podemos usar *pattern matching* com `Numbers`, como mostra o Código 30.

Código 30 – Pattern matching com números

```
1 fibonacci : Number -> Number
2 fibonacci =
3     \n ->
4         match n with
5             | 0 -> 0
6             | 1 -> 1
7             | _ -> fibonacci (n - 1) + fibonacci (n - 2)
```

Pattern matching em Ipe é exaustivo, ou seja, todas as variações de um tipo devem ser tratadas. No Código 30, precisaríamos tratar todos os números possíveis. A primeira linha (de cima para baixo) que resultar em um *match* é a linha executada. Ipe também possibilita o uso de _ para fazer *match* em qualquer valor de um tipo, o que é útil para tipos com infinitas etiquetas, como `Number` e `String`, mas também pode ser usado em tipos com etiquetas finitas. O Código 31 mostra a definição sintática de *pattern matching* em Ipe.

Código 31 – Definição de *pattern matching* em EBNF

```
type destruct  ::=  {upper identifier, "."}, upper identifier, {pattern};
pattern        ::=  "_"
                  |  number
                  |  string
                  |  type destruct;
match case     ::=  "|", pattern, "->", {attribution}, expression;
match          ::=  "match", expression, "with", {match case}-;
```

4.5 OPERADORES

A Tabela 3 apresenta todos os operadores disponíveis na linguagem Ipe. Embora os operadores tenham uma assinatura de tipo como qualquer outra função, eles são usados de maneira infixa. Ou seja, o operador `+` possui a assinatura `Number -> Number -> Number`, mas o primeiro argumento fica à esquerda do símbolo `+`. Poderíamos definir uma função `soma` com a mesma assinatura, e usar como `soma 1 2`. Com os operadores, o primeiro argumento vai à esquerda do operador, e o segundo à direita, como em `1 + 2`.

Tabela 3 – Operadores em Ipe

Operador	Assinatura	Descrição
+	`Number -> Number -> Number`	Soma dois números
-	`Number -> Number -> Number`	Subtrai dois números
/	`Number -> Number -> Number`	Divide dois números
*	`Number -> Number -> Number`	Multiplica dois números
^	`Number -> Number -> Number`	Eleva o primeiro número ao segundo
|>	`a -> (a -> b) -> b`	Aplica um argumento a uma função
<|	`(a -> b) -> a -> b`	Aplica uma função a um argumento

Podemos ver que Ipe possui um baixo número de operadores. Isso é proposital, pois queremos que a linguagem seja o mais simples possível. Isso também é uma consequência de não termos o tipo booleano na linguagem.

Da tabela, talvez os operadores com maior destaque são |> e <|, pois estes operadores ajudam a criar séries de transformações, que são muito comuns em programação funcional. O Código 32 ajuda a ilustrar o uso deles.

Código 32 – Exemplos de uso dos operadores de pipe

```
1  comPipeDireito : Number -> Number -> Number -> Number
2  comPipeDireito =
3      \n1 n2 n3 -> operacao3 n3 |> operacao2 n2 |> operacao1 n1
4
5  // é equivalente a
6
7  comPipeEsquerdo : Number -> Number -> Number -> Number
8  comPipeEsquerdo =
9      \n1 n2 n3 -> operacao1 n1 <| operacao2 n2 <| operacao3 n3
10
11 // é equivalente a
12
13 semPipe : Number -> Number -> Number -> Number
14 semPipe =
15     \n1 n2 n3 -> operacao1 n1 (operacao2 n2 (operacao3 n3))
```

O Código 33 mostra a definição de expressões em Ipe, que combinam operadores e chamadas de função para construir novos valores. Para facilitar a compreensão da gramática, ela é apresentada com recursão à esquerda.

Código 33 – Definição de expressões em EBNF

```
expression      ::=  expression, "|>", expression
                 |   expression, "<|", expression
                 |   function
                 |   match
                 |   exponentiation;
exponentiation  ::=  exponentiation, "^", term
                 |   term;
term            ::=  term, ("+"| "−"), factor
                 |   factor;
factor          ::=  factor, ("/"| "*"), primary
                 |   primary;
primary         ::=  number
                 |   string
                 |   record
                 |   list
                 |   variable name, {primary}
                 |   "(", expression, ")";
list            ::=  "[", [expression, {",", expression}], "]";
variable name   ::=  {uppercase identifier, "."}, record access;
record access   ::=  lowercase identifier, ".", record access
                 |   lowercase identifier;
```

5 CONSTRUÇÃO DO COMPILADOR

O compilador de Ipe, escrito em Haskell, é dividido em 4 partes: análise sintática e léxica, transformação, checagem de tipos, e emissão de código. Discutiremos cada uma dessas partes nas seções seguintes. Finalmente, na Seção 5.5, falaremos sobre o processo de compilação como um todo: como os arquivos são escaneados, e como as diferentes partes do processo se comunicam.

5.1 ANÁLISE SINTÁTICA E LÉXICA

A etapa de análise sintática e léxica é a primeira etapa do compilador Ipe. Esta etapa também é conhecida como a etapa de *parsing*, e ela é responsável por transformar texto que representa um programa Ipe em um formato de árvore, que representa a gramática de Ipe. Durante esta etapa, o compilador verifica se todos os símbolos de entrada são válidos, ao mesmo tempo em que verifica se a ordem dos símbolos também é válida. Caso haja algum erro sintático ou léxico no texto analisado, o processo de compilação para, e o erro é reportado ao usuário.

5.2 TRANSFORMAÇÃO DE CÓDIGO

A etapa de transformação de código é responsável por passar pela árvore gerada na etapa de *parsing* e normalizar o código e sua estrutura. Na versão inicial de Ipe, essa etapa só é responsável por adicionar os construtores de uniões de tipos na lista de exportação de cada módulo, mas outras transformações poderiam ser adicionadas no futuro, como algumas otimizações, substituições de variáveis por constantes, ou qualquer simplificação que torne o trabalho das próximas etapas mais fácil (desde que o funcionamento do código gerado não seja afetado). Como essa etapa apenas transforma um programa válido em outro programa válido, ela não gera erros.

5.3 CHECAGEM DE TIPOS

A checagem de tipos é a etapa que garante que um programa Ipe é seguro de se executar. Esta etapa é responsável por assegurar que todas as funções invocadas recebam o número de argumentos corretos, e que estes argumentos sejam do tipo correto, além de verificar a exaustividade de *pattern matching* e a unicidade de *pattern matching* em valores. O sistema de tipos de Ipe é baseado no sistema de tipos de Hindley-Milner (DAMAS; MILNER, 1982), e usa uma adaptação do algoritmo W (MIKKOLA, 2018), com adição de uniões de tipos, tipos opacos, *records* e listas.

O algoritmo de checagem de tipos se baseia no conceito de unificação de tipos. A unificação de tipos é o processo de encontrar uma substituição para variáveis de tipos que torna dois tipos iguais. Por exemplo, o tipo `a -> b` pode ser unificado com

o tipo `Number -> String` se a variável `a` for substituída por `Number`, e a variável `b` for substituída por `String`, assim como o tipo `Number` pode ser unificado com o tipo `Number`. No caso de dois tipos não poderem ser unificados (por exemplo, se tentarmos unificar `Number` e `String`), o compilador deve reportar um erro e abortar o processo de compilação. O algoritmo original é estendido, para termos suporte a uniões de tipos, tipos opacos, *records* e listas, além de outros tipos de expressão, como definição de funções, *pattern matching*, e outros tipos de operadores binários, como os operadores de *pipe*. Seguindo este processo, o compilador Ipe é capaz de inferir os tipos de qualquer programa Ipe válido, de modo que anotações de tipo são opcionais. Porém, é recomendável que anotações de tipo sejam usadas, para gerar melhores mensagens de erro, e para melhor documentação do programa. Caso esta etapa seja bem sucedida, nenhuma saída é gerada, e o processo de compilação continua.

5.4 EMISSÃO DE CÓDIGO

Por fim, depois de analisar todo o código, a estrutura em árvore que representa o programa em Ipe é transformada em uma estrutura em árvore que representa um programa em Javascript, e esta árvore é transformada em texto. Cada módulo Ipe gera um arquivo Javascript com o mesmo nome, e com a extensão `.ipe.js`. Por exemplo, o módulo `Root.ipe` gera o arquivo `Root.ipe.js`. Como sabemos das etapas anteriores que o código é válido, não existe a possibilidade de gerar erros nesta etapa.

5.5 PROCESSO DE COMPILAÇÃO

O processo de compilação é iniciado chamando um aplicativo de linha de comando (*CLI*), onde o usuário deve indicar o arquivo de entrada do programa, que deve expor uma função chamada `main`. O Código 34 mostra um exemplo de chamada do compilador, passando `Root.ipe` como arquivo de entrada, e especificando o diretório `output` como diretório de saída (diretório onde os arquivos Javascript deverão ser gerados).

Código 34 – Chamada do aplicativo de linha de comando

```
1 $ ipe build Root.ipe -o output
```

Ao executar o comando acima, o compilador irá fazer o *parsing* do módulo `Root.ipe` e de todos os módulos que ele importa, e aplicar a etapa de transformação de código em cada um, gerando uma árvore de sintaxe abstrata para cada módulo. Em seguida, o compilador começa a aplicar a etapa de checagem de tipos, utilizando as árvores de sintaxe abstrata geradas anteriormente. A checagem de tipos começa pelo módulo de entrada (no exemplo, `Root.ipe`). Para cada módulo importado pelo arquivo sendo analisado, o compilador verifica todos os módulos que estão sendo importados:

- Se o módulo importado já foi analisado, o compilador continua a checagem de tipos do módulo atual.
- Se o módulo importado ainda não foi analisado, o compilador analisa o módulo importado, e depois continua com a checagem de tipos do módulo atual.

Os resultados da checagem de tipos são armazenados em uma tabela de símbolos. Ao terminar a checagem de tipos de um módulo, apenas suas definições exportadas são mantidas na tabela de símbolos global. Ao terminar a checagem de tipos de um módulo, o compilador já o transforma em uma representação textual em Javascript. Depois de checar os tipos de todos os módulos importados (direta ou indiretamente) pelo módulo de entrada, o compilador gera a estrutura de pastas e escreve os arquivos com o conteúdo em Javascript de cada módulo. Deste modo, apenas módulos que aparecem na cadeia de dependências do módulo de entrada são analisados e transformados em Javascript. Módulos que não são usados não são analisados, e portanto não são transformados em Javascript.

Por fim, o compilador baixa as bibliotecas padrão (descritas na Seção 6.1), definidas em Javascript, para que o programa possa ser executado. Para executar o programa, basta usar algum *runtime* Javascript, como Node, Deno ou Bun. O código Javascript de algumas bibliotecas padrão utiliza funções oferecidas pelo Bun, portanto é recomendável utilizar Bun para executar programas Ipe.

6 PROGRAMAS EM IPE

Para facilitar o desenvolvimento de aplicações Ipe, todos os programas contam com um conjunto de bibliotecas padrão, também conhecidas como *Prelude*, que são apresentadas na Seção 6.1. Ademais, para manter a pureza dos programas Ipe, a linguagem disponibiliza um ambiente que faz a gerência de efeitos colaterais, como acesso a disco ou rede. Esse ambiente é apresentado na Seção 6.2.

6.1 BIBLIOTECAS PADRÃO

Ipe oferece módulos para facilitar o uso de seus tipos primitivos (`Number` e `String`), além de módulos para manipulação de listas e dicionários, e representação de dados opcionais. Essas bibliotecas podem utilizar código Javascript (a linguagem alvo da compilação) para utilizar otimizações e acessar dados que não seriam possíveis de acessar em Ipe. O módulo mais importante para o desenvolvimento de aplicativos backend que respondem a chamadas HTTP é o módulo `Http` (descrito na Subseção 6.1.9), que possui funções que são responsáveis por montar a estrutura básica de uma aplicação web, como a da Seção 3.1.

6.1.1 Number

Este módulo contém algumas funções para manipulação de números, como arrendondamento. Além disso, como Ipe não possui operadores lógicos (em especial == e !=), este módulo oferece funções e tipos para comparar números, como mostra o Código 35.

Código 35 – Comparação de números em Ipe

```
1 type union CompareResult =
2     | Smaller
3     | Equal
4     | Greater
5
6 compare : Number -> Number -> CompareResult
```

6.1.2 String

O módulo de `String` contém funções para manipulação de strings, como concatenação, busca de substrings, e conversão para maiúsculas e minúsculas.

6.1.3 Maybe e Result

Como Ipe não possui nenhum tipo de exceção ou `null`, os módulos `Maybe` e `Result` são utilizados para representar valores opcionais e resultados de funções que podem falhar, respectivamente. O Código 36 mostra a definição destes tipos. Este módulo também possui funções para manipular esses tipos, como as funções `Maybe.map` e `Result.map`, que servem para manipular o conteúdo de um `Maybe` ou `Result`, sem precisar de uma expressão `match`.

Código 36 – Definição de `Maybe` e `Result`

```
type union Maybe content =
    | Nothing
    | Just content

type union Result error ok =
    | Error error
    | Ok ok
```

6.1.4 List

O módulo `List` define funções para lidar com uma das principais estruturas de dado de Ipe. Internamente, listas são as mesmas listas usadas em Javascript, e podem ser criadas da mesma maneira: com colchetes. O Código 37 mostra exemplos de uso de listas, e a implementação da função `map`, que modifica o tipo dos elementos de uma lista.

Código 37 – Parte do módulo de listas de Ipe

```
split : List element -> Maybe { head : element, tail : List element }
// A implementação de split é feita em Javascript
// split tenta separar o primeiro elemento da lista do resto da lista

insertLeft : element -> List element -> List element
// A implementação de insertLeft é feita em Javascript
// insertLeft adiciona um elemento no início da lista

map : (element -> newElement) -> List element -> List newElement
map =
    \mapFn list ->
        match split list with
            | Nothing -> []
            | Just parts ->
                insertLeft (mapFn parts.head) (map mapFn parts.tail)
```

```
16
17 empty : List element
18 empty = []
19
20 singleton : element -> List element
21 singleton =
22     \element -> [ element ]
```

6.1.5 Dict

Dicionários são coleções de chave-valor dinâmicos. O módulo `Dict` define o tipo `Dict` e funções para manipular dicionários. Por questões de performance e simplicidade, dicionários são implementados em Javascript.

6.1.6 Subscription

O tipo `Subscription` é usado pelo ambiente de execução (discutido na Seção 6.2) para sinalizar eventos que ocorrem no sistema e o programa pode escolher ser notificado para tomar alguma ação.

6.1.7 Promise

Promises (ou promessas) são funções que solicitam ações (tarefas) no mundo exterior para o ambiente de execução, e usam as *Promises* nativas do Javascript. Esse módulo disponibiliza funções para manipular tarefas, como usar o resultado de uma como entrada de outra. O Código 38 mostra um exemplo de sequenciamento de tarefas.

Código 38 – Exemplo de sequenciamento de tarefas

```
1 uploadFile : String -> String -> Promise String
2 uploadFile =
3     \fileId destinationUrl ->
4         Db.getFilePathById fileId
5         |> Promise.andThen File.read
6         |> Promise.andThen (\fileContent -> Http.post destinationUrl
             { body = fileContent })
```

A função acima recebe um identificador de arquivo e uma URL de destino, e lê o caminho do arquivo no banco de dados, lê o arquivo, e envia o conteúdo para a URL de destino. A função `Promise.andThen` é usada para encadear as tarefas, passando o resultado de uma para a outra. Este código é apenas um exemplo, pois na realidade, teríamos que lidar com erros (usando o tipo `Result`), e não apenas com o caso de sucesso.

Para realizar os efeitos colaterais de uma `Promise`, usamos a função `Promise.perform : (Result error ok -> event) -> Promise error ok -> Effect event`, que converte uma `Promise` em um `Effect`, que pode ser enviado ao *runtime*.

6.1.8 JSON

Ipe tem três módulos relacionados a JSON: `Json`, `Json.Encode` e `Json.Decode`. O módulo `JSON` define o tipo `Json`, que representa um valor JSON. O módulo `Json.Encode` define funções para transformar valores de Ipe em valores JSON, e o módulo `Json.Decode` define funções para transformar valores JSON em valores de Ipe. O Código 39 mostra um exemplo de uso desses módulos.

Código 39 – Exemplo de uso dos módulos de JSON

```
type alias User =
    { name : String
    , age : Int
    }

encodeUser : User -> Json.Json
encodeUser =
    \user ->
        Json.Encode.object
           [ { field = "name", value = Json.Encode.string user.name }
           , { field = "age", value = Json.Encode.int user.age }
           ]

userDecoder : Json.Decode.Decoder User
userDecoder =
        Json.Decode.object (\name age -> { name = name, age = age })
        |> Json.Decode.field "name" Json.Decode.string
        |> Json.Decode.field "age" Json.Decode.int
```

6.1.9 Http

O módulo `Http` define funções para tratar requisições HTTP que o servidor recebe. A função `Http.createApp` é a principal função deste módulo, e ela é usada para criar um servidor HTTP capaz de manter um estado (chamado de **contexto**) entre respostas. Esta função recebe configurações para o servidor (como a porta em que ele deve escutar), uma função para construir o contexto inicial, e uma função que recebe o contexto atual e uma requisição HTTP, e retorna uma resposta HTTP e um novo contexto. Com esta função, também é possível registrar `Subscription`s. O Capítulo 7 mostra exemplos de uso deste módulo.

6.2 RUNTIME E ARQUITETURA

Programas Ipe são desenvolvidos seguindo uma certa arquitetura, também inspirada pela arquitetura Elm. Programas Ipe são representados por três partes principais:

- *Model*: representa o estado do programa. É um tipo definido pelo programador – tipicamente um `Record`. Também é necessário definir uma função `init` que define o estado inicial do programa. Também é possível realizar *Effects* na função de `init`.
- *update*: função que recebe um evento e o estado atual do programa (`Model`). Eventos também são representados por um tipo definido pelo programador, e tipicamente são uniões de tipos. A função `update` possui a anotação `update : event -> model -> { model : model, effect : Effect event }`, ou seja, recebe um evento e o estado atual do programa, e retorna o novo estado, com possíveis efeitos colaterais.
- *subscribe*: função que possibilita o programa a se inscrever em eventos do sistema, como o recebimento de requisições HTTP. Para tal, a função recebe como argumento o `Model` atual, e retorna uma `Subscription`.

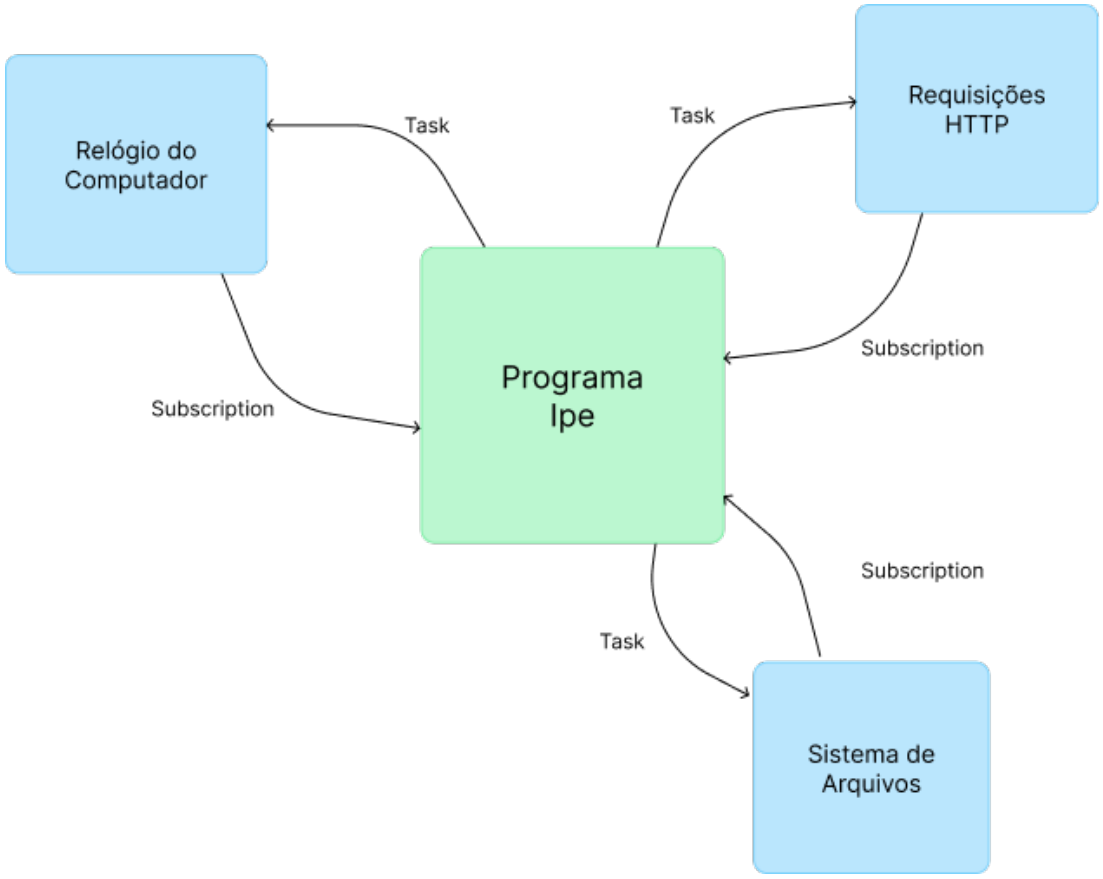
O *Runtime* (ambiente de execução, em português) é o responsável por manter o estado do programa, e executar as funções `update` e `subscribe`, além de executar os efeitos colaterais retornados pela função de `update`. O *Runtime* é implementado em Javascript, e ele converte `Promises` em código Javascript, e retorna o resultado para o programa Ipe. Similarmente, quando um programa Ipe adiciona uma nova `Subscription`, o *Runtime* é responsável por gerar Javascript que avise o programa Ipe quando necessário. A Figura 1 mostra uma visão geral do *Runtime* se comunicando com outras partes do computador.

Para dizer ao *Runtime* quais funções utilizar como funções de inicialização, *update* e *subscribe*, programas Ipe devem exportar uma função `main` do módulo apontado como módulo de entrada (geralmente `Root.ipe`). O Código 40 mostra um exemplo de programa completo em Ipe, que imprime na tela uma mensagem a cada minuto, parando depois de imprimir 10 vezes.

Código 40 – Exemplo de programa completo em Ipe, que imprime na tela uma mensagem a cada minuto

```
1 module Root exports [main]
2
3 type alias Model = { messagesPrinted : Number }
4
5 type union Event =
6     | PrintMessage
```

Figura 1 – Arquitetura do *Runtime* Ipe




```
7       | PrintedMessage (Result Console.Error String)
8
9   main : { init : { model : Model, effect : Effect Event }
10          , update : Event -> Model -> { model : Model, effect : Effect
                Event }
11          , subscribe : Model -> Subscription Event
12          }
13  main =
14      { init = { model = { messagesPrinted = 0 }, effect = Effect.none
            }
15      , update = update
16      , subscribe = subscribe
17      }
18
19  update : Event -> Model -> { model : Model, effect : Effect Event }
20  update =
21      \event model ->
22          match event with
23              | PrintMessage ->
```

```
24                  { model = model
25                  , effect =
26                      String.fromInt model.messagesPrinted
27                      |> Console.log
28                      |> Promise.perform PrintedMessage
29                  }
30              | PrintedMessage ->
31                  { model =
32                      { messagesPrinted = model.messagesPrinted + 1 }
33                  , effect = Effect.none
34                  }
35
36  subscribe : Model -> Subscription Event
37  subscribe =
38      \model ->
39          match Number.compare model.messagesPrinted 10 with
40              | Number.Smaller -> Time.every 60000 PrintMessage
41              | Number.Equal -> Subscription.none
42              | Number.Greater -> Subscription.none
```

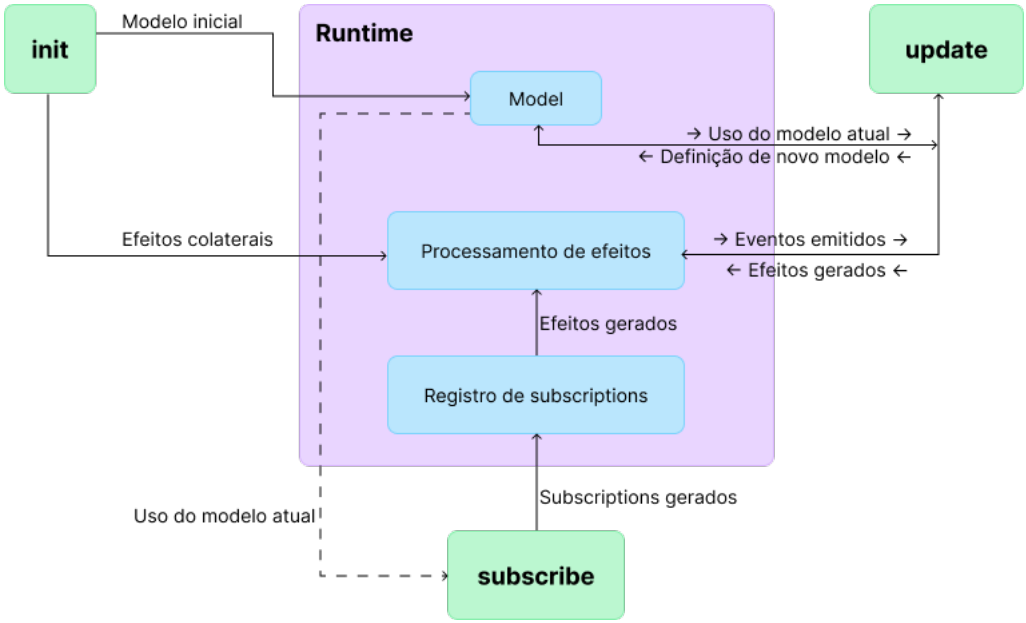
Os passos que o *Runtime* toma para executar o Código 40 são:

1. Ler a função `main` para saber quais são as funções `init`, `update` e `subscribe`.
2. Executa a função `init`, e define o estado inicial do programa como o `Record { messagesPrinted = 0 }`. Caso a função `init` retornasse um efeito colateral, o *Runtime* executaria o efeito colateral.
3. Usa o modelo inicial para executar a função `subscribe`, e começa a esperar por eventos.
4. Quando um evento chega, o *Runtime* executa a função `update` com o evento e o modelo atual. Efeitos (retornados pelas funções `init` e `update`) também produzem eventos, que também são tratados na função `update`. Após cada chamada a `update`, a função `subscribe` é chamada novamente.
5. Após imprimir 10 vezes na tela, a função `subscribe` retorna `Subscription.none`, e o programa para de esperar por eventos. Quando não existem mais efeitos sendo processados, e a função `subscribe` retorna `Subscription.none`, o programa termina.

A Figura 2 mostra mais detalhes de como as funções `init`, `update` e `subscribe` se integram com o *Runtime*.

Podemos ver que, após a definição do modelo inicial em `init`, as funções `update`

Figura 2 – Arquitetura detalhada do *Runtime* Ipe



e `subscribe` leem o modelo, e respondem a eventos.

Embora este seja o fluxo principal de execução, alguns módulos do *Prelude* de Ipe podem oferecer funções auxiliares, que ajudam a definir o comportamento do programa, de maneira mais apropriada de acordo com uma certa necessidade ou contexto. Por exemplo, com `Http.createApp`, o modelo passa a se chamar contexto, a função `init` se chama `createContext`, e a função `update` é substituída pela função `handleRequest`, de forma que algumas partes do funcionamento do servidor são omitidas do programador final, como a `Subscription` que escuta por requisições HTTP, ou o efeito que envia respostas HTTP.

7 RESULTADOS OBTIDOS

Neste capítulo, será apresentada a implementação em Ipe do sistema descrito na Seção 3.1. Depois, iremos comparar alguns detalhes da implementação com as implementações em outras linguagens, a fim de mostrar as vantagens e desvantagens de cada uma.

7.1 IMPLEMENTAÇÃO DA APLICAÇÃO BASE EM IPE

A implementação em Ipe usa várias das bibliotecas padrão de Ipe, apresentadas na Seção 6.1, com maior foco nas bibliotecas de `JSON` e `Http`, o que é comum em *API*s *REST*, por ser o principal meio de comunicação com o mundo exterior. Esta seção será dividida em partes menores, cada uma descrevendo uma parte da implementação. Por simplicidade, todo o código que será apresentado está em um único arquivo, `Root.ipe`. Para aplicações maiores, recomenda-se que o código seja dividido em vários arquivos diferentes. Durante a apresentação do código, partes que poderiam ser definidas em outros arquivos serão destacadas. Nos códigos apresentados, não serão mostradas as importações de módulos, que são feitas no início do arquivo, e são necessárias para que as funções e tipos dos módulos importados possam ser usados.

7.1.1 Descrição do contexto

Como a maior parte dos programas em Ipe, começamos definindo um tipo para o nosso modelo (ou contexto, quando usando a biblioteca `Http`). Para a aplicação base, o contexto deve saber o último identificador usado para um usuário, e um dicionário de usuários, como mostra o Código 41.

Código 41 – Definição do contexto

```
1 type alias User =
2   { name : String
3   , age : Number
4   }
5
6 type alias Context =
7   { lastId : Number
8   , users : Dict.Dict Number User
9   }
```

A definição do tipo `User` poderia ser feita em outro módulo, `User.ipe`, para que todas as suas funcionalidades (que veremos nas seções seguintes), fiquem agrupadas em um único módulo. Para maior segurança, o tipo `User` poderia ser definido como um tipo opaco neste outro arquivo. Deste modo, poderíamos controlar todas as maneiras

de criar e acessar usuários, evitando que usuários inválidos sejam criados, como, por exemplo, usuários com idade menor do que 0, ou cujo nome é uma `String` vazia.

7.1.2 Funções de transformação de dados

Como a grande vantagem de usar Ipe é sua tipagem estática, e a expressividade de seus tipos, é ideal que os tipos recebidos de fora da aplicação (como em requisições *HTTP*) sejam convertidos para tipos internos da aplicação o mais rápido possível. Similarmente, quando for necessário enviar dados para fora da aplicação, é ideal que os tipos internos sejam convertidos para tipos externos o mais tarde possível, apenas quando necessário. Para isso, usamos funções de transformação de dados, que convertem os tipos internos para tipos externos, e vice-versa. No caso da aplicação base, o único tipo externo é JSON, mas Ipe é versátil o suficiente para lidar com outros tipos de dados no futuro, sem muita alteração no código de usuário. O Código 42 mostra como transformar um JSON em um usuário.

Código 42 – Função que transforma JSON em um usuário

```
1 userDecoder : Json.Decode.Decoder User
2 userDecoder =
3   Json.Decode.map2 (\name age -> { age = age, name = name })
4     (Json.Decode.field 'name' Json.Decode.string )
5     (Json.Decode.field 'age' Json.Decode.number)
```

Decodificadores (ou *Decoders*, em inglês) são funções responsáveis por transformar tipos externos (como JSON) em tipos internos (como `User`). Eles descrevem como traduzir os dados para algo que Ipe entenda. No caso do Código 42, o decodificador `userDecoder` recebe um JSON e, se o JSON tiver um campo (*field*, em inglês) `name` e um campo `age`, que sejam uma `String` e um `Number`, respectivamente, ele usa esses valores para construir um `Record` com o mesmo formato de `User`. É importante ressaltar que esta função apenas descreve como transformar um JSON em um usuário, mas não faz a transformação em si. Veremos que a função `Json.Decode.parseJson` é responsável por isso. A função `userDecoder` poderia ser definida no arquivo `User.ipe`.

Código 43 – Transformando um usuário em JSON

```
1 encodeUser : { key: Number, value: User } -> Json.Value
2 encodeUser = \input ->
3   Dict.empty {}
4   |> Dict.insert 'id' (Json.Encode.number input.key)
5   |> Dict.insert 'name' (Json.Encode.string input.value.name)
6   |> Dict.insert 'age' (Json.Encode.number input.value.age)
7   |> Json.Encode.object
8
```

```
encodeUsers : Dict.Dict Number User -> Json.Value
encodeUsers = \users ->
  users
  |> Dict.toList
  |> Json.Encode.list encodeUser
```

O Código 43 mostra como transformar usuários em objetos JSON, usando codificadores (ou `encoders`, em inglês). Para facilitar a implementação da rota que retorna vários usuários, definimos também a função `encodeUsers`, que transforma um dicionário de usuários (que é o que temos no contexto) em um JSON. Na implementação de `encodeUser`, usamos a função `Json.Encode.object`, que simplesmente transforma um dicionário em um objeto JSON. Estas funções também poderiam ser definidas no arquivo `User.ipe`.

É importante notar que a representação de um usuário em JSON (usado para se comunicar com o mundo externo) não necessariamente reflete a representação interna de um usuário, o que é desejável. Diferentes linguagens de programação têm diferentes maneiras de representar dados, e é importante que isto não afete a comunicação entre elas. Por exemplo, em Python, é comum usar `snake_case`, enquanto em Ipe, é comum usar `camelCase` para nomes de variáveis. Com a estratégia das bibliotecas de JSON de Ipe, podemos usar o nome que quisermos para as variáveis internas, e apenas mapeá-las para o nome correto quando for necessário enviar dados para fora da aplicação. Também podemos aplicar processamentos nos dados antes de ingeri-los para dentro de uma aplicação Ipe, como, por exemplo, converter uma `String` para um `type union`.

7.1.3 Configuração da aplicação

Agora que já definimos o contexto e as funções de transformação de dados, podemos definir a aplicação em si, utilizando a função `main`, que chamará a função `Http.createApp` para iniciar um servidor que escuta chamadas `HTTP`. O Código 44 mostra a configuração inicial da aplicação, como a porta em que o servidor deve escutar, e como construir o contexto inicial. Exploraremos o conteúdo escondido pelas reticências na Subseção 7.1.4.

Código 44 – Configuração inicial da aplicação

```
main: {} -> {}
main = \_ -> Http.createApp
    { port = 3000
    , createContext = \_ -> Promise.succeed
        { lastId = 0, users = Dict.empty {} }
    , handleRequest = \context request -> ...
    }
```

Este código diz que o servidor deve escutar na porta 3000, e que o contexto inicial deve ser um dicionário vazio de usuários, e inicializar o `lastId` com 0. Este `Record` segue a definição de `Context` do Código 41.

7.1.4 Rotas

Ipe não possui nenhum mecanismo de roteamento embutido, mas é bastante simples implementar um, simplesmente usando *pattern matching*. Por conveniência, Ipe disponibiliza o *endpoint* da URL requisitada como uma lista de `String`s, onde cada entrada da lista é um pedaço do *endpoint*. Por exemplo, a URL `/users/1` seria representada como `["users", "1"]`. Além disso, o método da requisição é representado por uma união de tipos, como podemos ver no Código 45, que substitui as reticências do Código 44.

Código 45 – Roteamento da aplicação base

```
1  match request.endpoint with
2  | ['users'] ->
3    match request.method with
4      | Http.Get ->
5        Promise.succeed
6          { response =
7            context.users
8            |> encodeUsers
9            |> Http.jsonResponse
10         , newContext = context
11         }
12
13     | Http.Post ->
14       match Json.Decode.parseJson userDecoder request.body with
15         | Ok user ->
16           Promise.succeed
17             { response =
18               Dict.empty {}
19               |> Dict.insert 'id' (context.lastId + 1)
20               |> Json.Encode.object
21               |> Http.jsonResponse
22             , newContext =
23               { lastId = context.lastId + 1
24               , users = Dict.insert (context.lastId + 1) user
                     context.users
25               }
26             }
```

```
27
28          | Err error ->
29            Promise.succeed
30              { response = Http.jsonResponse (Json.Encode.string error)
31              , newContext = context
32              }
33
34      | _ -> Promise.succeed { response = Http.jsonResponse
           (Json.Encode.string '404'), newContext = context }
35
36  | ['users', stringId] ->
37    match Number.fromString stringId with
38      | Nothing -> Promise.succeed { response = Http.jsonResponse
           (Json.Encode.string 'error: invalid ID'), newContext =
           context }
39      | Just id ->
40        match Dict.get id context.users with
41          | Nothing -> Promise.succeed { response = Http.jsonResponse
               (Json.Encode.string 'error: user not found'), newContext
               = context }
42          | Just user ->
43            match request.method with
44              | Http.Get ->
45                Promise.succeed
46                  { response =
47                    encodeUser { key = id, value = user }
48                    |> Http.jsonResponse
49                  , newContext = context
50                  }
51
52              | Http.Put ->
53                match Json.Decode.parseJson userDecoder request.body
                     with
54                  | Ok user ->
55                    Promise.succeed
56                      { response =
57                        Dict.empty {}
58                        |> Dict.insert 'id' id
59                        |> Json.Encode.object
60                        |> Http.jsonResponse
61                      , newContext =
62                        { lastId = context.lastId
```

```
63                      , users = Dict.insert id user context.users
64                      }
65                  }
66
67              | Err error ->
68                Promise.succeed
69                  { response = Http.jsonResponse
                        (Json.Encode.string error)
70                  , newContext = context
71                  }
72
73
74          | Http.Delete ->
75            Promise.succeed
76              { response = Http.jsonResponse (Json.Encode.string
                    id)
77              , newContext =
78                { lastId = context.lastId
79                , users = Dict.remove id context.users
80                }
81              }
82
83          | _ -> Promise.succeed { response = Http.jsonResponse
                (Json.Encode.string id), newContext = context }
84
85 | _ -> Promise.succeed { response = Http.jsonResponse
     (Json.Encode.string '404'), newContext = context }
```

A estratégia de roteamento é fazer `pattern matching` com o *endpoint* da requisição, e então fazer `pattern matching` com o método da requisição. Embora seja uma solução mais verbosa do que as outras linguagens analisadas, tudo o que está acontecendo é bastante explícito, a fim de ser mais fácil de compreender. Além disso, o programador é forçado a validar todos os dados vindos do mundo exterior, o que é uma boa prática de programação. No exemplo acima, o programador é obrigado a validar que o ID do usuário é um número, e que o usuário existe no contexto.

Com esta estratégia de roteamento, também é bastante simples adicionar a funcionalidade de *middleware*, que é uma função que é executada antes de um grupo qualquer de rotas (por exemplo, todas as rotas de usuário). Como tudo é baseado em funções e *pattern matching*, basta chamar a função desejada no *pattern matching* desejado. Esta estratégia também possibilita que o programador divida a lógica de roteamento de qualquer maneira que quiser.

7.1.5 Comparação com outras linguagens

Dentre todas as implementações, Ipe foi a linguagem que necessitou de mais linhas escritas (um total de 132, contando com espaços em branco), embora linguagens que usam *frameworks* com geração de código (C#, Elixir e Haskell) para iniciar novos projetos tenham vários arquivos gerados, resultando em um número de linhas de código total muito maior, o que pode ser intimidante para programadores que não conhecem a linguagem ou o *framework*.

Ipe também não possui nenhuma função ou decorador específico para definir rotas da aplicação, o que pode resultar em código mais verboso, como podemos ver no Código 45. Porém, esta estratégia de roteamento é bastante simples de entender, e é bastante flexível, permitindo que o programador divida a lógica de roteamento da maneira que quiser. Além disso, em rotas com parâmetros (como `/users/:id`), todas as outras linguagens dependem em definir uma *string* com esse parâmetro, e depois recebê-lo através de uma variável. Esta abordagem é bastante suscetível a erros de nomeação, ou aceitar menos ou mais parâmetros do que o esperado. Em Ipe, um simples *pattern matching* é responsável por perceber que a variável existe na URL, e capturá-la em uma variável.

É muito mais simples iniciar um projeto em Ipe do que em qualquer uma das linguagens analisadas. Todas as outras linguagens requerem a instalação do compilador ou interpretador da linguagem, além do processo de escolha de *framework*, e a instalação do *framework* escolhido, ou de sua ferramenta de geração de código. Para iniciar um projeto de API em Ipe, basta instalar seu compilador e um ambiente de execução Javascript, e escrever um arquivo `.ipe`, sem a necessidade de qualquer configuração adicional.

Além disso, por conta de seu sistema de tipos, é muito difícil que um programa encontre um erro que o faça parar de funcionar em tempo de execução, pois Ipe obriga o programador em pensar em casos de erro. Isto é bastante diferente de linguagens sem tipagem estática (Python, Javascript e Elixir), e também é melhor do que Haskell, que, embora sua ótima tipagem estática, ainda permite que o programador ignore casos de entrada inválida. E, diferente de todas as outras linguagens, Ipe não possui o conceito de erros, fazendo com que estes casos sejam lidados explicitamente, e impedindo a chance de um erro não ser pego (geralmente com blocos `try/catch`).

Desta forma, embora Ipe possa ser mais verbosa e devagar de se escrever do que outras linguagens, ela é mais segura, e mais simples de se iniciar um projeto. Além disso, Ipe é capaz de prover mais garantias de funcionamento, o que é desejável para aplicações *backend*, que comumente têm a necessidade de funcionar indefinidamente. Ainda, assim como Elm, aplicações Ipe têm a tendência de seguirem uma estrutura geral parecida com outras aplicações Ipe, o que facilita a compreensão de código escrito por outros programadores.

8 CONSIDERAÇÕES FINAIS

Este trabalho apresenta Ipe, uma linguagem de programação puramente funcional. Ipe visa usar características de linguagens e *frameworks* de desenvolvimento de aplicações *backend* modernos. Vimos algumas dessas características na Subseção 3.2.3 e Subseção 3.2.1, quando vimos aplicações *backend* em Javascript e Python, respectivamente. Também vimos o exemplo de uma aplicação em Haskell, na Subseção 3.2.5, e vimos que, embora o código seja bastante resiliente, ele é extremamente complexo e difícil de compreender. Ipe tem o objetivo de simplificar o uso da programação funcional, sem comprometer a expressividade garantida por linguagens funcionais e diminuindo a barreira de entrada no mundo funcional, seguindo o exemplo de Elm.

No Capítulo 4, definimos as características e a sintaxe da linguagem Ipe: vimos definições de módulos, tipos e funções. Também discutimos sobre a arquitetura e o *runtime* Ipe no Capítulo 6, que possibilitam programas compostos apenas de funções puras, sem efeitos colaterais, mas que sejam úteis no mundo real.

No Capítulo 7, colocamos em prática o que foi definido nos capítulos anteriores, após implementar um compilador para Ipe, usando a linguagem Haskell. Construímos a aplicação base, definida na Seção 3.1, e vimos que, embora Ipe precise de mais linhas de código, a implementação é bastante simples e fácil de compreender, além de ter várias garantias, por conta de seu forte sistema de tipos e gerenciamento de sistemas colaterais.

Deste modo, concluímos todos os objetivos delineados na Seção 1.1, e temos uma nova linguagem puramente funcional, que serve para definir APIs REST, com fortes garantias de tipos. Como trabalho futuro, podemos expandir os módulos padrão de Ipe, além de adicionar otimizações ao compilador, dando um foco maior à performance da linguagem. Também poderíamos adicionar um sistema que permita a interação entre Ipe e Javascript, para que programadores Ipe possam criar suas próprias bibliotecas padrão.

REFERÊNCIAS

CZAPLICKI, Evan. Elm: Concurrent frp for functional guis. **Senior thesis, Harvard University**, v. 30, 2012. Citado na p. 17.

DAMAS, Luis; MILNER, Robin. Principal Type-Schemes for Functional Programs. *In:* PROCEEDINGS of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Albuquerque, New Mexico: Association for Computing Machinery, 1982. (POPL '82), p. 207–212. DOI: 10.1145/582153.582176. Disponível em: <https://doi.org/10.1145/582153.582176>. Citado na p. 46.

HEJLSBERG, Anders *et al.* **The C# programming language**. [*S.l.*]: Pearson Education, 2008. Citado na p. 25.

HINSEN, Konrad. The Promises of Functional Programming. **Computing in Science & Engineering**, v. 11, n. 4, p. 86–90, 2009. DOI: 10.1109/MCSE.2009.129. Citado na p. 17.

HUDAK, Paul. Conception, Evolution, and Application of Functional Programming Languages. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 21, n. 3, p. 359–411, set. 1989. ISSN 0360-0300. DOI: 10.1145/72551.72554. Disponível em: <https://doi.org/10.1145/72551.72554>. Citado nas pp. 17, 30.

MASSE, Mark. **REST API design rulebook: designing consistent RESTful web service interfaces**. [*S.l.*]: " O'Reilly Media, Inc.", 2011. Citado na p. 20.

MIKKOLA, Jeremy. Understanding Algorithm W, mar. 2018. Disponível em: <https://jeremymikkola.com/posts/2018_03_25_understanding_algorithm_w.html>. Acesso em: 2023. Citado na p. 46.

PETERS, Tim. The zen of python. *In:* PRO Python. [*S.l.*]: Springer, 2010. P. 301–302. Citado na p. 32.

SCALFANI, Charles. **WHY FUNCTIONAL PROGRAMMING SHOULD BE THE FUTURE OF SOFTWARE DEVELOPMENT**. [*S.l.*], out. 2022. Disponível em: <https://spectrum.ieee.org/functional-programming>. Acesso em: 29 nov. 2022. Citado na p. 17.

STACK Overflow Developer Survey 2022. [*S.l.*], mai. 2022. Disponível em: <https://survey.stackoverflow.co/2022>. Acesso em: 24 nov. 2022. Citado nas pp. 20, 22, 25, 26, 28, 31.

TIOBE Index for November 2022. [*S.l.*], nov. 2022. Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em: 24 nov. 2022. Citado nas pp. 19, 24–26, 29.

VAN ROSSUM, G.; DRAKE, F.L. **An Introduction to Python**. [*S.l.*]: Network Theory Limited, 2011. (A Python manual). ISBN 9781906966133. Disponível em: <https://books.google.com.br/books?id=8-qkuAAACAAJ>. Citado na p. 24.

WIRTH, Niklaus. Extended backus-naur form (ebnf). **Iso/Iec**, v. 14977, n. 2996, p. 2–21, 1996. Citado nas pp. 34, 68.

Apêndices

APÊNDICE A – DESCRIÇÃO COMPLETA DA SINTAXE DE IPE EM EBNF

Neste apêndice, apresentamos a sintaxe completa de Ipe usando a notação EBNF, de acordo com (WIRTH, 1996) (como iremos fazer o nosso próprio *parser*– ao invés de usar um gerador automático –a gramática não será otimizada– as ferramentas que usaremos para construir o *parser* fazem isso por nós –, e alguns detalhes não serão apresentados, como espaços em branco e comentários de bloco entre tokens). O símbolo inicial da gramática é `file`. Discutimos mais sobre a sintaxe de Ipe no Capítulo 4.

```
file              ::=  module, {new type | tl value}-;
line comment      ::=  "//", {? qualquer caractere ?} - "\n", "\n";
block comment     ::=  "/*", {? qualquer caractere ?} - "*/", "*/";
doc comment       ::=  "/|*", {? qualquer caractere ?} - "*/", "*/";
upper identifier  ::=  uppercase letter, {letter | digit | "_"};
lower identifier  ::=  lowercase letter, {letter | digit | "_"};
number            ::=  ["−"], {digit}-, [".", {digit}-];
string            ::=  "'", {? qualquer caractere ?} - ("\n"| "'"), "'";
module name       ::=  {upper identifier, "."}, upper identifier;
exported item     ::=  letter, {letter | digit | "_"};
export list       ::=  "[", {exported item}-, "]";
module decl       ::=  "module", module name, "exports", export list;
import decl       ::=  "import", module name, [ "as", module name ];
module            ::=  module decl, [doc comment], {import decl};
field decl        ::=  lower identifier, ": ", expression;
record            ::=  "{", [field decl, {",", field decl}], "}";
custom type       ::=  {upper identifier, "."}, upper identifier, {any type};
any type          ::=  custom type
                    |  lower identifier
                    |  record annotation;
record ann field  ::=  lower identifier, ": ", any type;
record annotation ::=  "{", [record ann field, {",", record ann field}] "}";
alias             ::=  upper identifier, parameters, "=", any type;
union             ::=  upper identifier, parameters, "=", {constructor}-;
constructor       ::=  "|", upper identifier, [record annotation];
new type          ::=  "type alias", alias
                    |  "type union", union
                    |  "type opaque", union;
```

```
fn args         ::= {lowercase identifier};
function        ::= "\", fn args, "->", {attribution}, expression;
tl annotation   ::= lowercase identifier, ": ", {any type, "->"}, any type;
attribution     ::= lowercase identifier, "=", expression, ";";
tl left         ::= [doc comment], [tl annotation], lowercase identifier;
tl value        ::= tl left "=", expression;
type destruct   ::= {upper identifier, "."}, upper identifier, {pattern};
pattern         ::= "_"
                  | number
                  | string
                  | type destruct;
match case      ::= "|", pattern, "->", {attribution}, expression;
match           ::= "match", expression, "with", {match case}-;
expression      ::= expression, "|>", expression
                  | expression, "<|", expression
                  | function
                  | match
                  | exponentiation;
exponentiation  ::= exponentiation, "^", term
                  | term;
term            ::= term, ("+"| "-"), factor
                  | factor;
factor          ::= factor, ("/"| "*"), primary
                  | primary;
primary         ::= number
                  | string
                  | record
                  | list
                  | variable name, {primary}
                  | "(", expression, ")";
list            ::= "[", [expression, {",", expression}], "]";
variable name   ::= {uppercase identifier, "."}, record access;
record access   ::= lowercase identifier, ".", record access
                  | lowercase identifier;
```