

Sprint 3



Laboratório de Compiladores - MC911

Grupo

(RA)

Nome

(155253)

Eric Krakauer

(155981)

José Pedro Nascimento

(156331)

Lucas Racoci

(156475)

Luiz Fernando Fonseca

(157055)

Rafael Gois

Pattern Matching

- Não é realmente necessário a linguagem
 - Trata-se de um facilitador sintático
 - É traduzido para estruturas condicionais
- Usado para facilitar a declaração de casos bases

Pattern Matching (Exemplo)

```
avg l = aux(l) (0) (0)
```

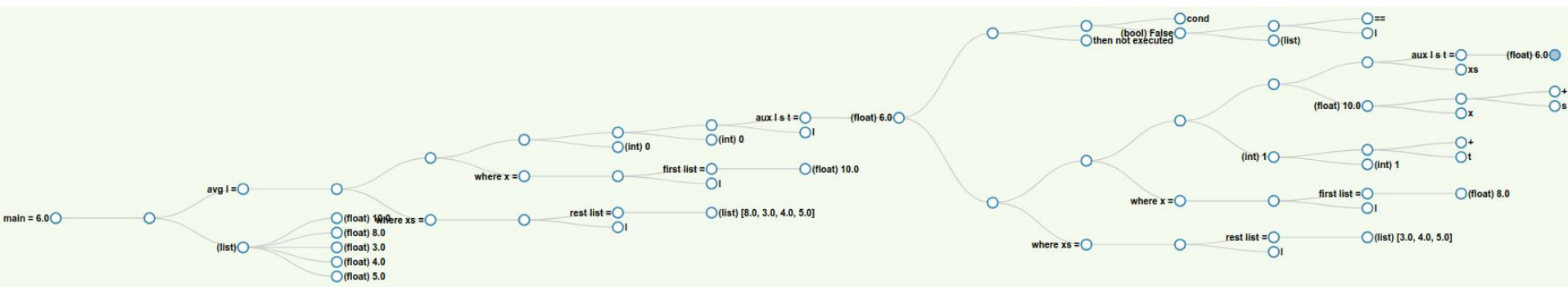
```
aux [] s t = s / t
```

```
aux l s t = aux(xs) (s+x) (t+1)
```

```
where x = first(l)
```

```
where xs = rest(l)
```

```
main = avg([10.0,8.0,3.0,4.0,5.0])
```



Análise Sintática - Lexer

- Lexer é capaz de identificar novos tokens para:
 - Listas
 - Tuplas

Análise Sintática - Parser

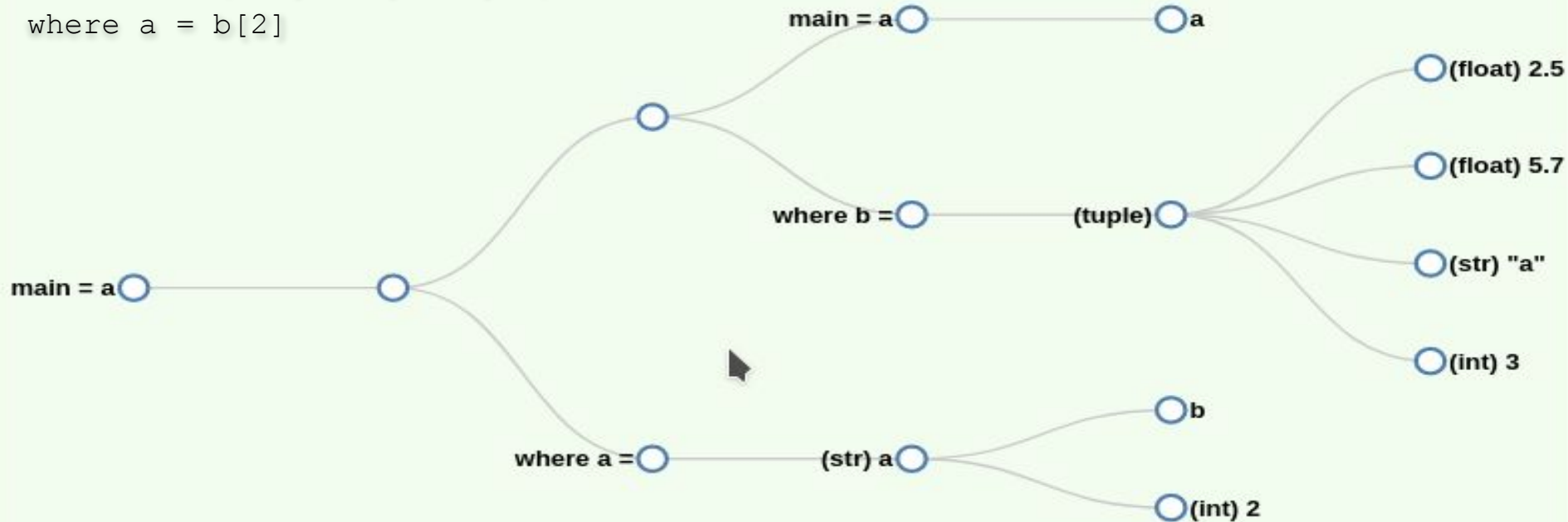
- É capaz de traduzir:
 - Expressões com where
 - String, Float, Struct
 - Novos tipos: Lista e Tupla
 - Pattern Matching
 - Lambdas

Parser: Novos tipos : Tupla

```
main = a
```

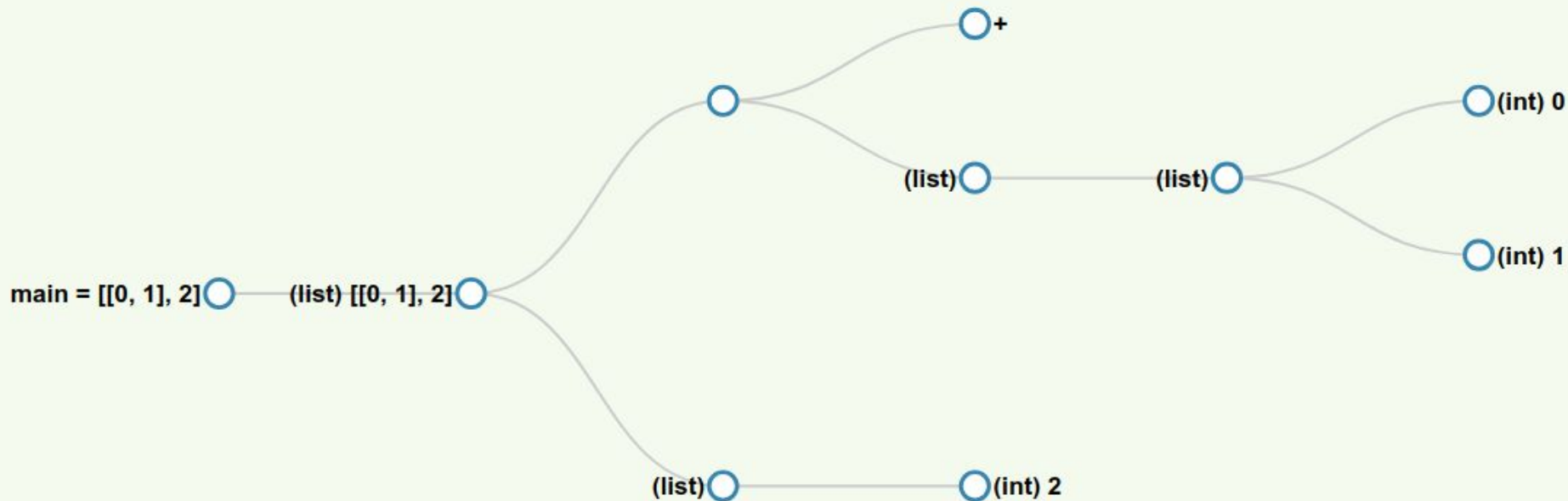
```
where b = (2.5, 5.7, "a", 3)
```

```
where a = b[2]
```



Parser: Novos tipos : Lista

```
main = [[0,1]] + [2]
```



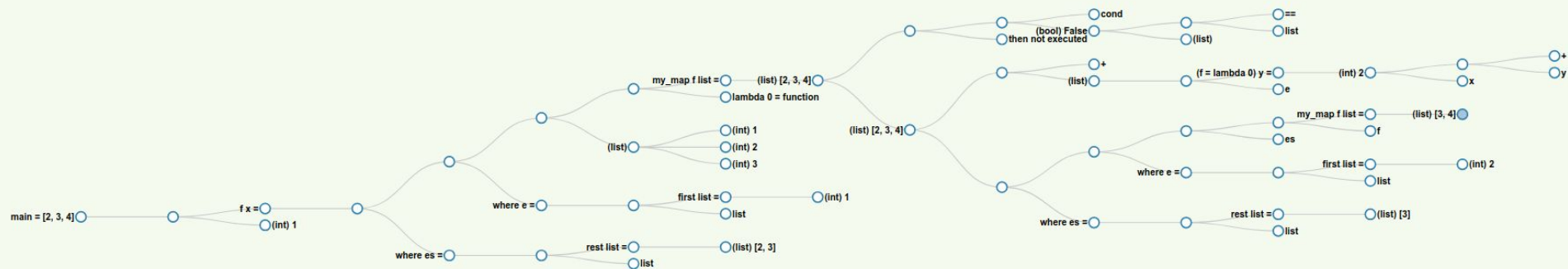
Programação Funcional

- Funções que operam sobre os elementos de uma lista:
 - Map: retorna lista modificada
 - Filter: retorna sub-lista
 - Fold: retorna combinação dos elementos

Programação Funcional (Exemplo)

```
main = f(1)
f x = my_map(\ y -> y + x) ([1,2,3])

my_map f [] = []
my_map f list = [f(e)] + my_map(f) (es)
  where e = first(list)
  where es = rest(list)
```



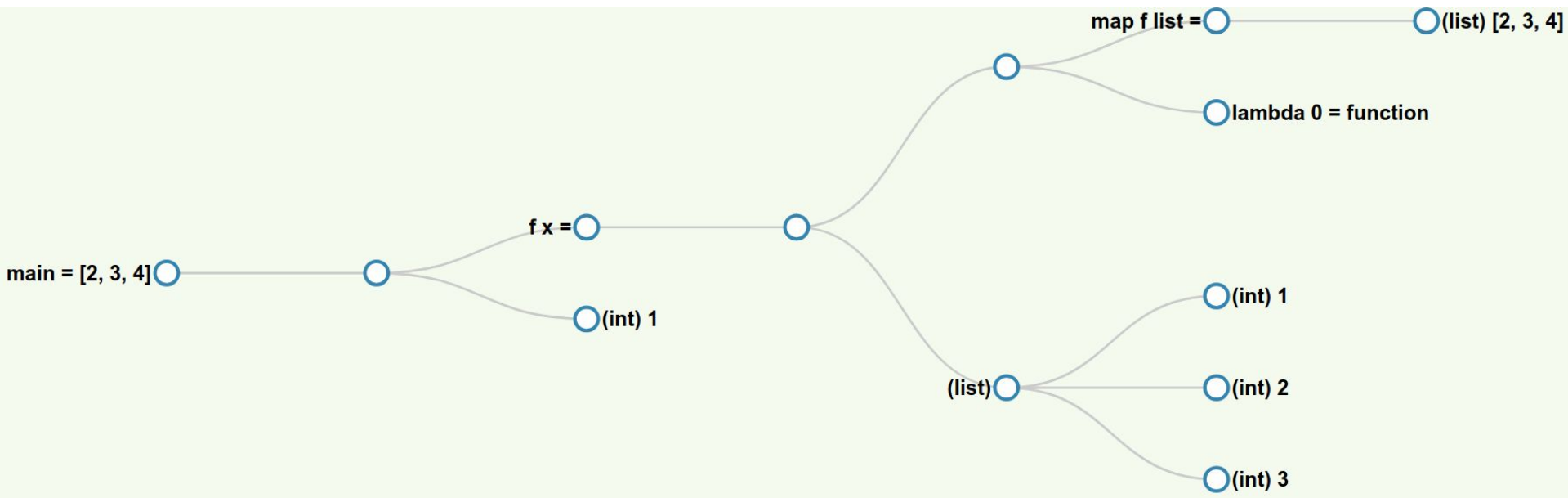
Lambdas

- Função anônima
- Na nossa linguagem é possível declarar funções com escopos aninhados
- Permite passar funções como argumento de uma maneira mais visual

Lambda (Exemplo)

```
main = f(1)
```

```
f x = map(\y -> y + x) ([1,2,3])
```



Cliente Web

- O cliente web mostra 3 novas otimizações (memoização, constant propagation e constant folding).
- Modificações do CSS, HTML e JavaScript usando as funções de D3, aproveitando a base da redução eta vinda do sprint anterior.
- [Sintaxe de Listas]

Otimização

- Foram feitas 3 novas otimizações: Constant Folding, Constant Propagation e Memoização.
- Memoização é uma técnica de otimização que consiste no cache do resultado de uma função baseada nos parâmetros de entrada.
- Assim, não é necessário calcular resultados frequentes no programa, o que economiza tempo de processamento.
- No entanto, o gasto de memória é maior pelo fato de armazenar uma tabela.

Otimização de Memoização

- Evita recálculo em chamadas recursivas
- Visualmente evita a repetição de sub-árvores
- Implementação: Dicionário: Argumentos → Resposta

Otimização de Memoização (Exemplo)

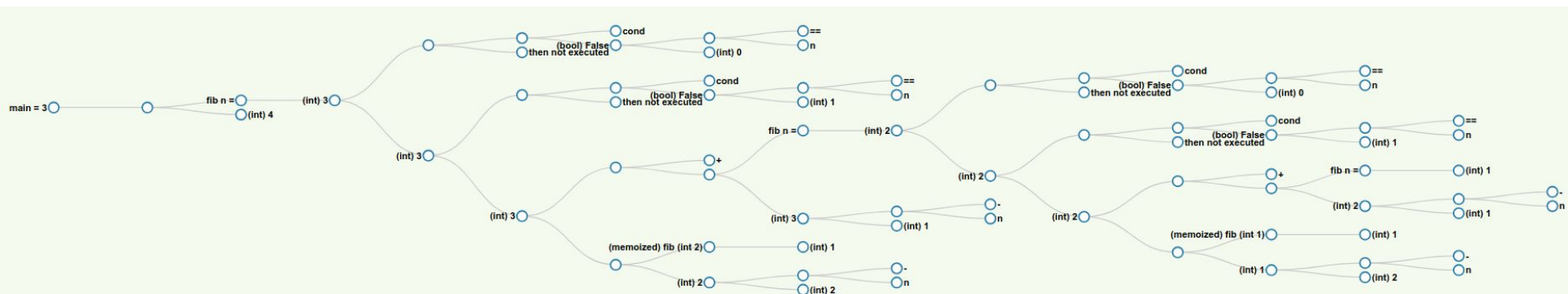
```
fib 0 = 1
```

$$\text{fib } 1 = 1$$

```
fib n = fib (n - 1) + fib (n - 2)
```

```
main = fib(4)
```

- Executado no programa para melhor visualização

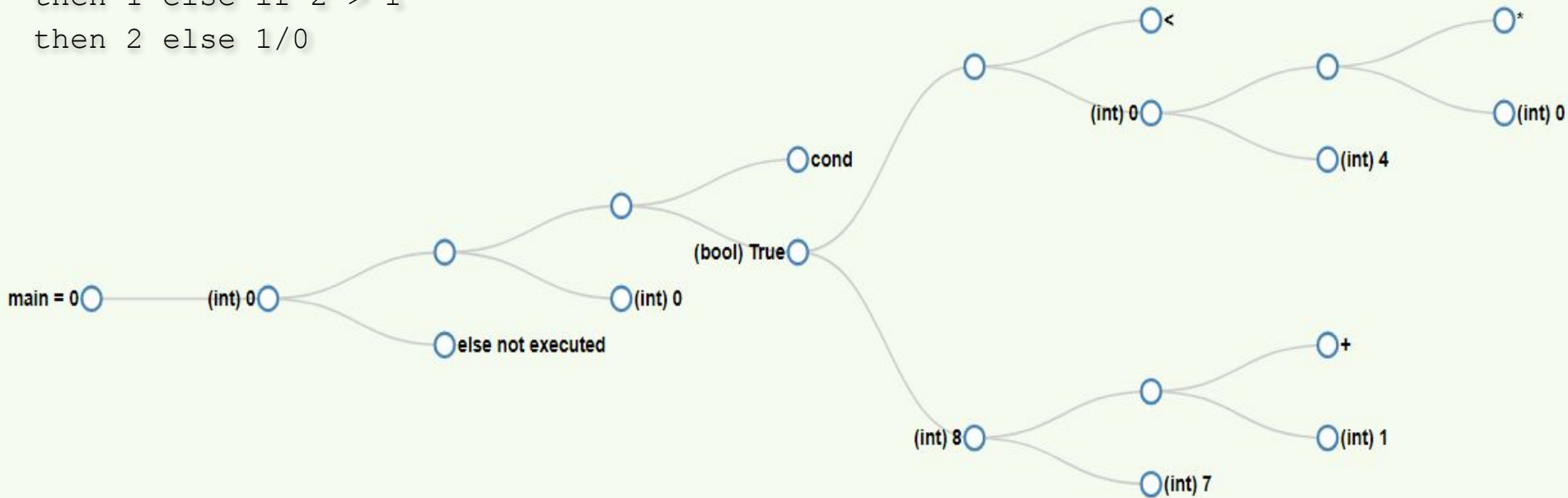


Otimização

- Constant Folding executa previamente as operações entre constantes para substituir pelo resultado final no código compilado.
- Na construção do Json que representa a árvore, verificamos as operações entre elementos constantes da árvore e já executamos.




Otimização de Constant Folding (Desligada)

```
main = if 0 * 4 < 1 + 7  
then 0 else if 1 == 1  
then 1 else if 2 > 1  
then 2 else 1/0
```



Otimização de Constant Folding (Ligada)

```
main = if 0 * 4 < 1 + 7  
then 0 else if 1 == 1  
then 1 else if 2 > 1  
then 2 else 1/0
```

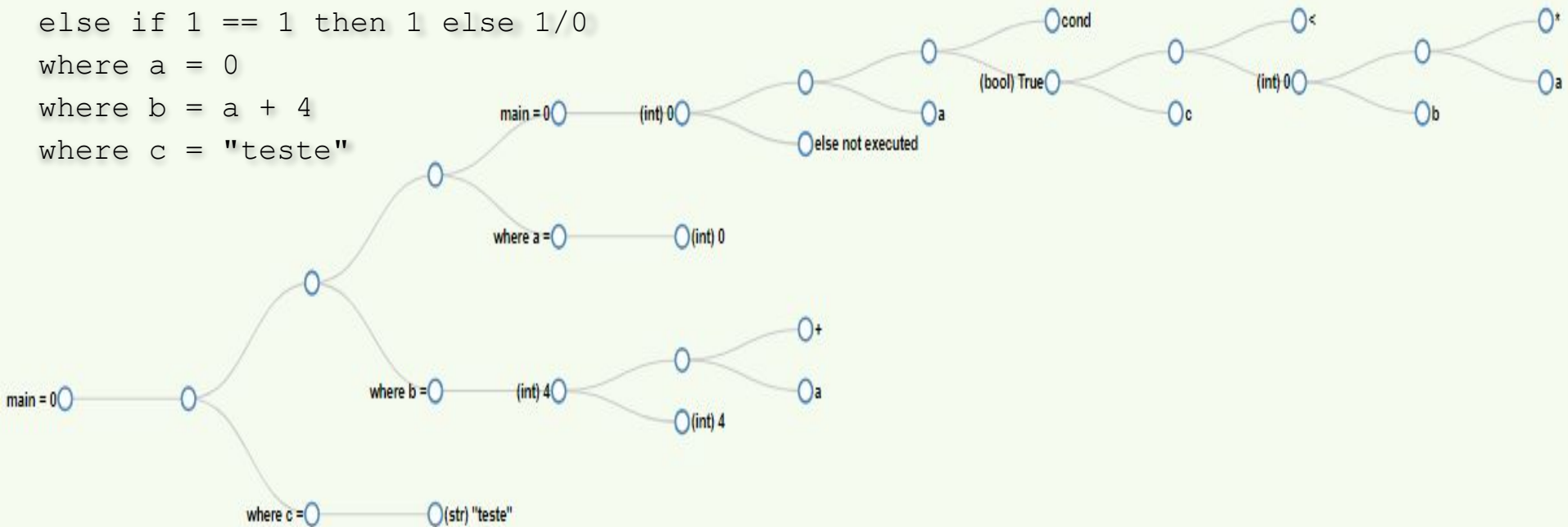
main = 0  ————— (int) 0  —————  (int) 0

Otimização

- Constant Propagation substitui todas as ocorrências da variável que tem um valor constante pelo seu valor em si.
- Nos aproveitamos do design pattern visitor para percorrer a árvore, remover a estrutura das variáveis e substituir os valores.

Otimização de Constant Propagation(Desligada)

```
main = if a * b < c then a  
      else if 1 == 1 then 1 else 1/0  
where a = 0  
      where b = a + 4  
      where c = "teste"
```



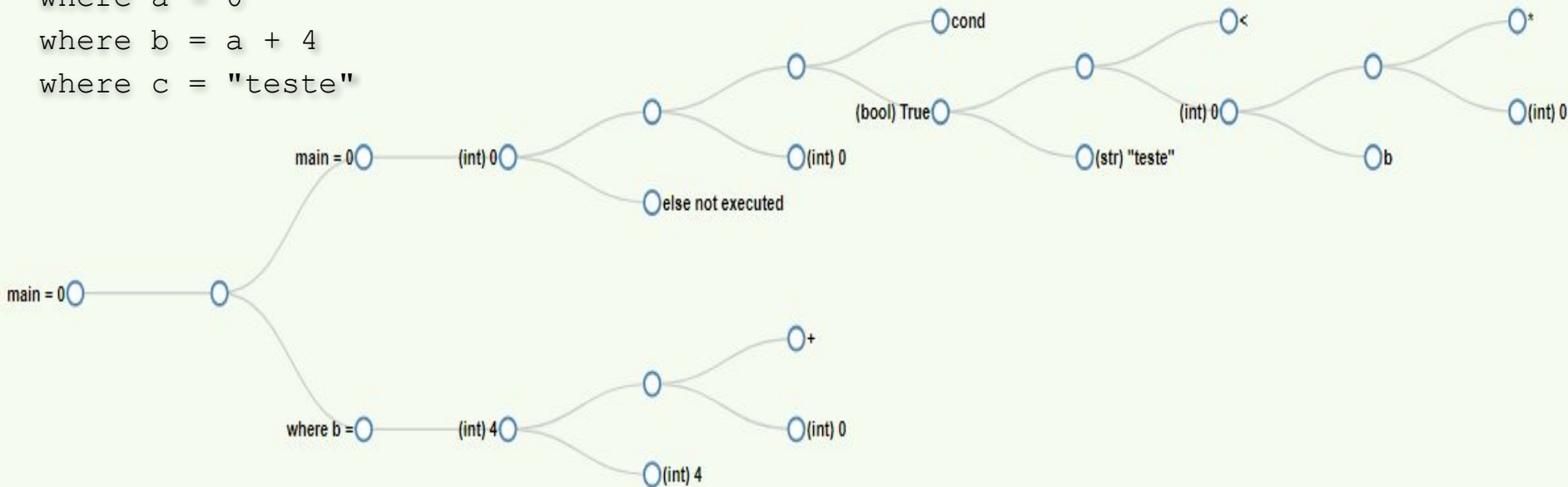
Otimização de Constant Propagation(Ligada)

```
main = if a * b < c then a  
      else if 1 == 1 then 1 else 1/0
```

where a = 0




where b = a + 4

where c = "teste"



Otimizações de Constant Propagation e Constant Folding (Ligadas)

```
main = if a * b < c then a  
      else if 1 == 1 then 1 else 1/0  
where a = 0  
where b = a + 4  
where c = "teste"
```

main = 0  ————— (int) 0  —————  (int) 0

Todas as otimizações Ligadas

fib 0 = 0

fib 1 = 1

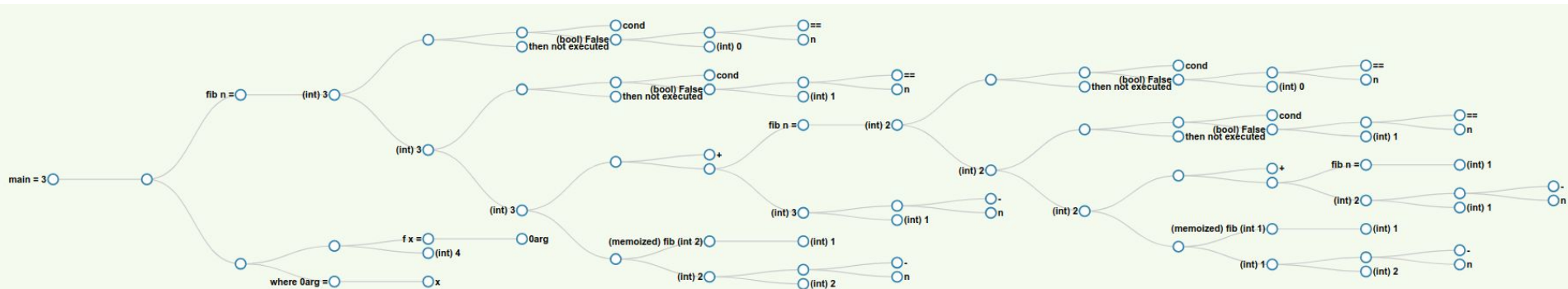
fib n = fib(n-1) + fib(n-2)

g x = x

f x = g(x)

main = fib(f(a + a)) where a = 2

- ☒ Constant Folding
- ☒ Memoization
- ☒ Constant Propagation
- ☒ Otimização Eta



Docker

- Configurar mais testes para o docker.
 - Testar funcionalidades implementadas no Sprint 3:
 - Pattern Matching
 - Lambdas
 - Otimização
 - Memoização
 - Constant Propagation
 - Constant Folding

Overview do projeto

- Tokens: definição de funções, chamada de funções, definição da função main, operador condicional if-else, números naturais, booleanos, operadores aritméticos, operadores lógicos e comentários
- Tradução para Estrutura de Árvore
- Cliente Web e Servidor
- Docker
- Testes

Overview do projeto

- Definição de inteiro, ponto flutuante, string, declaração de variáveis com where, declaração de struct
- Escopos de funções
- Interpretação de código
- Análise semântica e detecção de erros
- Otimização: Redução eta
- Front-end: adaptação para redução eta
- Testes