



Universidade do Estado do Rio de Janeiro

Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística

Roberto Carlos dos Santos

RecPy: pré-compilador para estudo da conversão de funções recursivas

Orientador: Professor Dr. Fabiano de Souza Oliveira

Rio de Janeiro
2021

Roberto Carlos dos Santos

RecPy: pré-compilador para estudo da conversão de funções recursivas

Monografia apresentada como requisito parcial para obtenção do título de Bacharel em Ciências da Computação, do Instituto de Matemática e Estatística da Universidade do Estado do Rio de Janeiro.

Orientador: Professor Dr. Fabiano de Souza Oliveira

Rio de Janeiro

2021

**CATALOGAÇÃO NA FONTE
UERJ/ REDE SIRIUS/ CB/C**

A ficha catalográfica deve ser preparada pela equipe da biblioteca. Não deve ser contada para fins de paginação.

Será elaborada após a normalização do trabalho.

NA VERSÃO IMPRESSA DEVERÁ CONSTAR NO VERSO DA FOLHA DE ROSTO.

Autorizo, apenas para fins acadêmicos e científicos, a reprodução total ou parcial desta monografia, desde que citada a fonte.

Assinatura

Data

Roberto Carlos dos Santos

RecPy: pré-compilador para estudo da conversão de funções recursivas

Monografia apresentada como requisito parcial para obtenção do título de Bacharel em Ciências da Computação, do Instituto de Matemática e Estatística da Universidade do Estado do Rio de Janeiro.

Aprovada em 10 de março de 2021.

Orientador: Professor Dr. Fabiano de Souza Oliveira

Instituto de Matemática e Estatística - UERJ

Banca Examinadora: _____

Prof. Dr. Paulo Eustáquio Duarte Pinto

Instituto de Matemática e Estatística - UERJ

Prof. Dr. Francisco Figueiredo Goytacaz Sant'anna

Instituto de Matemática e Estatística - UERJ

Rio de Janeiro

2021

DEDICATÓRIA

Tenho consciência de que o sacrifício do tempo de convívio familiar seja comum aos que se dedicam à pesquisa científica. Não obstante, entendo como indispensável valorizar a compreensão e o apoio recebidos de meus familiares. Penso na construção de um futuro melhor não só para minha família, mas para a sociedade. Desejo que este trabalho apresente os resultados esperados e possa contribuir, de algum modo, ainda que naturalmente limitado em pretensões e alcance, com os esforços evolutivos na área de Ciências da Computação.

Dedico-o também aos (às) professores(as) e aos(as) colegas e amigos que contribuíram para minha formação e aprimoramento. Faço menção especial a um grande incentivador, ao tempo em que dava meus primeiros passos na área de Tecnologia da Informação, há cerca de 30 anos: o saudoso amigo e pai de amigos Hugo Paulo de Araújo Cabral, exemplo de dedicação e boa vontade em compartilhar conhecimentos.

AGRADECIMENTOS

Agradeço a quem couber, no plano terreno ou espiritual, pela dádiva de estar vivo e com boa saúde para o estudo e para o trabalho.

Agradeço à minha mãe, Nice, por toda sua dedicação e seus esforços em propiciar a melhor formação moral e cultural que pode me oferecer. Seu exemplo de retidão e bom caráter, além do gosto pelos estudos, fizeram imensa diferença positiva em minha vida! Muito do que hoje tenho de virtudes é reflexo das boas e generosas sementes que plantou.

Agradeço à minha esposa, Andréa, pelos longos anos de convivência harmônica e feliz, pela maravilhosa prole que criamos, pelo constante apoio e tranquilidade que me passa.

Agradeço às minhas filhas, Mariana e Nicole, por darem à minha existência significado maior, por me trazerem mais felicidade, por me ensinarem a ser pessoa melhor.

Agradeço ao meu irmão José Carlos e à Marilena pelo apoio e incentivo em meus estudos, na juventude.

Agradeço a todos os professores que contribuíram com minha formação, tanto os atuais (Ciências da Computação na UERJ), quanto os de cursos passados (Direito, na UFRJ; Mecânica Técnica, no CEFET/RJ; ensino básico; Educação Física, na UFRJ; cursos técnicos e preparatórios). Sempre estudei em instituições públicas. Por isso mesmo, testemunho o imenso valor que o ensino público significa para as famílias de poucos recursos. Mais que isso, arrisco repetir uma obviedade: não existe nação que tenha evoluído sem valorizar o ensino, como um todo. Portanto, todo investimento nessa área, desde que bem empregado, tende a retornar multiplicado em benefícios para a sociedade.

Agradeço particularmente ao Professor Fabiano de Souza Oliveira não só pela orientação neste trabalho, mas também por todas as disciplinas que ministrou com peculiar excelência e dedicação, em especial nas áreas de Estruturas de Dados, Grafos e Algoritmos. Enfatizo, nesse agradecimento, a confiança depositada e a oportunidade de escrever sobre tema de tamanha relevância para a escrita de algoritmos eficientes e confiáveis.

Tive também a sorte de, nas matérias acima mencionadas, ter sido aluno de outro excelente Professor: Paulo Eustáquio Duarte Pinto. Aprendi muito com ambos! Das maratonas de programação de que participei levo a lembrança de momentos lúdicos e extremamente produtivos em termos de evolução técnica e cultural.

Continuo estes agradecimentos e lembranças em apêndice (Apêndice I, tópico 9).

RESUMO

SANTOS, Roberto Carlos dos. ***RecPy: pré-compilador para estudo da conversão de funções recursivas.*** 2021. 116 fls. Monografia em Trabalho de Conclusão de Curso (TCC) – Bacharelado em Ciências da Computação – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2020. Neste trabalho, tratam-se os conceitos básicos relacionados a funções recursivas. Descrevem-se as situações em que as conversões, entre si, de *algoritmos recursivos caudais*, *de recursivos não caudais* e *de iterativos* podem ser úteis ou necessárias no aspecto de legibilidade ou quanto à eficiência dos respectivos códigos. Apresenta-se o aplicativo *RecPy*, um pré-compilador destinado à automatização dessas conversões, nos tipos de funções recursivas reconhecidas. Incluem-se exemplos práticos de conversões realizadas no aplicativo, além de gráficos dos resultados obtidos.

Palavras-chave: Recursão. Recursão caudal. Recursão de cauda. Função recursiva. Iterativa. Eliminação de recursão. Otimização de recursão. Conversão de recursão. Pré-compilador. Algoritmos eficientes. Códigos concisos. *RecPy*.

ABSTRACT

SANTOS, Roberto Carlos dos. *RecPy: precompiler for studying the conversion of recursive functions.* 2021. 116 pgs. Monography in Course Conclusion Paper - Bachelor of Computer Science - Institute of Mathematics and Statistics, State University of Rio de Janeiro, Rio de Janeiro, 2020. This work treats the basic concepts related to recursive functions. It describes the situations in which the conversion, among itself, between caudal recursive algorithms, non-caudal recursives and iteratives may be usefull or necessary in respect to the legibility or the efficience of the related codes. It presents the application RecPy, a pre-compiler made for the automatization of these conversions, for the reconizable recursive function's types. It includes practical examples of conversions executed in the applicative, beyond graphical results obtained.

Keywords: Recursion. Caudal recursion. Tail recursion. Recursive function. Iterative. Elimination of recursion. Recursion optimization. Conversion from recursion. Pre-compiler. Efficient algorithms. Concise codes. RecPy.

LISTA DE FIGURAS E GRÁFICOS

Figura 1- Tela de apresentação do RecPy	39
Figura 2 - Explicação sobre o carregamento do arquivo fonte (*.py ou *.pyw)	42
Figura 3 - Exemplo de abertura de arquivo para tradução de função recursiva caudal (TR) para iterativa (IT).....	42
Figura 4 - Clique no botão Translate para realizar a tradução.....	43
Figura 5 - Salve o resultado em um arquivo de script ou copie-o para a área de transferência	43
Figura 6 - Exemplo de função a ser traduzida	48
Figura 7 - Exemplo de função já traduzida.....	49
Figura 8 - Exemplo de tokens gerados em janela de comando (CMD).....	52
Figura 9 - Exemplo de árvore sintática abstrata (AST) gerada em linha de comando	52
Figura 10 - Exemplo de bloco de função para teste do parser e do lexer do Antlr4	53
Figura 11 - Tokens gerados para a função fatmodNTR	59
Figura 12 - Tokens gerados para a função fatmodTR	59
Figura 13 - Tokens gerados para a função fatmodIT.....	59
Figura 14 - Tokens gerados para a função fatNTR	60
Figura 15 - Tokens gerados para a função fatTR	60
Figura 16 - Tokens gerados para a função fatIT	60
Figura 17 - Tokens gerados para a função fibTR	60
Figura 18 - Tokens gerados para a função fibIT	60
Figura 19 - Tokens gerados para a função invNTR.....	60
Figura 20 - Tokens gerados para a função invTR	60
Figura 21 - Tokens gerados para a função invIT.....	60
Figura 22 - Tokens gerados para a função mdcTR.....	61
Figura 23 - Tokens gerados para a função mdcIT	61
Figura 24 - Tokens gerados para a função ordNTR	61
Figura 25 - Tokens gerados para a função ordTR	61
Figura 26 - Tokens gerados para a função ordIT.....	61
Figura 27 - Tokens gerados para a função palTR.....	61
Figura 28 - Tokens gerados para a função palIT	61
Figura 29 - Tokens gerados para a função potNTR.....	61
Figura 30 - Tokens gerados para a função potTR	61

Figura 31 - Tokens gerados para a função potIT	62
Figura 32 - Tokens gerados para a função primNTR	62
Figura 33 - Tokens gerados para a função primIT	62
Figura 34 - Tokens gerados para a função sumVecNTR	62
Figura 35 - Tokens gerados para a função sumVecTR	62
Figura 36 - Tokens gerados para a função sumVecIT	62
Figura 37 - Tokens gerados para a função textNTR	62
Figura 38 - Tokens gerados para a função textTR	62
Figura 39 - Tokens gerados para a função textIT	62
Figura 40 - Árvore sintática abstrata gerada para o algoritmo fatmodNTR	63
Figura 41 - Árvore sintática abstrata gerada para o algoritmo fatmodTR	63
Figura 42 - Árvore sintática abstrata gerada para o algoritmo fatmodIT	63
Figura 43 - Árvore sintática abstrata gerada para o algoritmo fatNTR	63
Figura 44 - Árvore sintática abstrata gerada para o algoritmo fatTR	64
Figura 45 - Árvore sintática abstrata gerada para o algoritmo fatIT	64
Figura 46 - Árvore sintática abstrata gerada para o algoritmo fibTR	64
Figura 47 - Árvore sintática abstrata gerada para o algoritmo fibIT	64
Figura 48 - Árvore sintática abstrata gerada para o algoritmo invNTR	65
Figura 49 - Árvore sintática abstrata gerada para o algoritmo invTR	65
Figura 50 - Árvore sintática abstrata gerada para o algoritmo invIT	65
Figura 51 - Árvore sintática abstrata gerada para o algoritmo mdcTR	65
Figura 52 - Árvore sintática abstrata gerada para o algoritmo mdcIT	66
Figura 53 - Árvore sintática abstrata gerada para o algoritmo ordNTR	66
Figura 54 - Árvore sintática abstrata gerada para o algoritmo ordTR	66
Figura 55 - Árvore sintática abstrata gerada para o algoritmo ordIT	66
Figura 56 - Árvore sintática abstrata gerada para o algoritmo palTR	67
Figura 57 - Árvore sintática abstrata gerada para o algoritmo palIT	67
Figura 58 - Árvore sintática abstrata gerada para o algoritmo potNTR	67
Figura 59 - Árvore sintática abstrata gerada para o algoritmo potTR	67
Figura 60 - Árvore sintática abstrata gerada para o algoritmo potIT	67
Figura 61 - Árvore sintática abstrata gerada para o algoritmo primTR	68
Figura 62 - Árvore sintática abstrata gerada para o algoritmo primIT	68
Figura 63 - Árvore sintática abstrata gerada para o algoritmo sumVecNTR	68
Figura 64 - Árvore sintática abstrata gerada para o algoritmo sumVecTR	68

Figura 65 - Árvore sintática abstrata gerada para o algoritmo sumVecIT	68
Figura 66 - Árvore sintática abstrata gerada para o algoritmo textNTR	69
Figura 67 - Árvore sintática abstrata gerada para o algoritmo textTR	69
Figura 68 - Árvore sintática abstrata gerada para o algoritmo textIT	69
Figura 69 - Exemplo de gráfico resultante do experimento de fatorial recursivo não caudal – fatNTR	70
Figura 70 - Exemplo de gráfico resultante do experimento de fatorial recursivo caudal – fatTR	71
Figura 71 - Exemplo de gráfico resultante do experimento de fatorial iterativo - fatIT	71
Figura 72 - Exemplo de gráfico resultante do experimento de fatorial iterativo – fatmodNTR	72
Figura 73 - Exemplo de gráfico resultante do experimento de fatorial modificado recursivo caudal - fatmodTR	72
Figura 74 - Exemplo de gráfico resultante do experimento de fatorial modificado iterativo - fatmodIT	73
Figura 75 - Exemplo de gráfico resultante do experimento de Fibonacci recursivo caudal - fibTR.....	73
Figura 76 - Exemplo de gráfico resultante do experimento de Fibonacci iterativo - fibIT	74
Figura 77 - Exemplo de gráfico resultante do experimento de inversão de string recursivo não caudal – invNTR.....	74
Figura 78 - Exemplo de gráfico resultante do experimento de inversão de string recursivo caudal – invTR	75
Figura 79 - Exemplo de gráfico resultante do experimento de inversão de string iterativo – invIT	75
Figura 80 - Exemplo de gráfico resultante do experimento de MDC recursivo caudal – mdcTR	76
Figura 81 - Exemplo de gráfico resultante do experimento de MDC iterativo – mdcIT	76
Figura 82 - Exemplo de gráfico resultante do experimento de ordenação recursivo não caudal - ordNTR	77
Figura 83 - Exemplo de gráfico resultante do experimento de ordenação recursivo caudal – ordTR.....	77
Figura 84 - Exemplo de gráfico resultante do experimento de ordenação iterativo - ordIT	78
Figura 85 - Exemplo de gráfico resultante do experimento de palíndromo recursivo caudal - palTR	78

Figura 86 - Exemplo de gráfico resultante do experimento de palíndromo iterativo - palIT ...	79
Figura 87 - Exemplo de gráfico resultante do experimento de potenciação recursivo não caudal – potNTR.....	79
Figura 88 - Exemplo de gráfico resultante do experimento de potenciação recursivo caudal - potTR.....	80
Figura 89 - Exemplo de gráfico resultante do experimento de potenciação iterativo - potIT ..	80
Figura 90 - Exemplo de gráfico resultante do experimento números primos recursivo caudal - primTR	81
Figura 91 - Exemplo de gráfico resultante do experimento de números primos iterativo - primIT	81
Figura 92 - Exemplo de gráfico resultante do experimento soma itens de um vetor recursivo não caudal - sumVecNTR	82
Figura 93 - Exemplo de gráfico resultante do experimento soma de itens de um vetor recursivo caudal - sumVecTR	82
Figura 94 - Exemplo de gráfico resultante do experimento soma de itens de um vetor iterativo - sumVecIT	83
Figura 95 - Exemplo de gráfico resultante do experimento de concatenação de texto recursivo não caudal - textNTR.....	83
Figura 96 - Exemplo de gráfico resultante do experimento de concatenação de texto recursivo caudal – textTR.....	84
Figura 97 - Exemplo de gráfico resultante do experimento de concatenação de texto iterativo - textIT	84

LISTA DE QUADROS E TABELAS

Quadro 1 - Resumo de artigos relacionados à otimização de códigos recursivos ou à conversão de funções recursivas em iterativas	34
Quadro 2 - Exemplo de código a ser evitado (if ou while na mesma linha do comando subsequente)	46
Quadro 3 - Exemplo de códigos não reconhecíveis (mais de uma linha de if ou while)	46
Quadro 4 - Aviso quanto às linhas de return	47
Quadro 5 - Aviso quanto à utilização de parêntesis excessivos e desnecessários	47
Quadro 6 - Códigos dos blocos de funções testadas.....	58
Quadro 7 - Códigos de funções testadas (algoritmos em que não houve experimentos na versão NTR)	59
Quadro 8 - Síntese de resultados quanto ao sucesso no procedimento de conversão e de reconhecimento das funções testadas	85
Quadro 9 - Complexidade de tempo de execução dos algoritmos testados.....	87
<hr/>	
Tabela 1 - Resultados quanto ao valor máximo da variável N (quantidade de chamadas recursivas ou de iterações).....	86

LISTA DE ABREVIATURAS E SIGLAS

AST	<i>Abstract Sintax Tree</i> (Árvore Sintática Abstrata)
CPS	<i>Continuation Passing Style</i> (Estilo de Continuação de Passagem)
IME	Instituto de Matemática e Estatística
IT	<i>Iteractive</i> (função Iterativa)
NTR	<i>Non Tail Recursion</i> (função Recursiva Não-Caudal)
PTC	<i>Proper Tail Call</i> (Chamada de Cauda Adequada ou Chamada Final Adequada)
PTR	<i>Proper Tail Recursion</i> (Adequada Recursão de Cauda)
TCC	Trabalho de Conclusão de Curso (esta dissertação)
TCE	<i>Tail Call Elimination</i> (Eliminação da Chamada de Cauda)
TCO	<i>Tail Call Optimization</i> (Otimização de Chamada Caudal)
TR	<i>Tail Recursion</i> (função Recursiva Caudal)
TRE	<i>Tail Recursion Elimination</i> (Eliminação da Recursão de Cauda)
TRF	<i>Tail Recursive Function</i> (Função Recursiva de Cauda ou Função Recursiva Final)
UERJ	Universidade do Estado do Rio de Janeiro

SUMÁRIO

INTRODUÇÃO	15
1. APRESENTAÇÃO.....	15
2. CONCEITOS BÁSICOS.....	16
2.1. Recursão	16
2.2. Recursão não caudal (Non Tail Recursive – NTR).....	16
2.3. Chamada de cauda recursiva ou Recursão Caudal (<i>Tail Recursive Call – TR</i>)	16
2.4. Eliminação da chamada de cauda (<i>tail call elimination</i>) pela transformação em função iterativa	16
2.5. Indução matemática.....	17
2.6. Algoritmos de divisão e conquista	17
3. OBJETIVOS	18
3.1. Objetivo geral	18
3.2. Objetivos específicos.....	18
3.3. Motivação e justificativa do trabalho.....	18
3.4. Levantamento de hipóteses	19
REVISÃO DA LITERATURA CIENTÍFICA	20
CONVERSÃO AUTOMÁTICA DE FUNÇÕES RECURSIVAS: APLICATIVO RECPY	35
4. INTRODUÇÃO	35
4.1. A série de artigos de Moertel.....	36
4.2. Vantagens e desvantagens da utilização de algoritmos recursivos. Quando e como utilizá-los e quando evitá-los	37
5. PRÉ-COMPILADOR RECPY, PARA CONVERSÃO DE FUNÇÕES RECURSIVAS	39
5.1. Apresentação do <i>RecPy</i>	39
5.2. Escopo de conversões possíveis no <i>RecPy</i>	41
5.3. Como utilizar o <i>RecPy</i>	41
5.4. Formatos de funções recursivas reconhecíveis no <i>RecPy</i>	44
5.5. Requisitos, restrições, observações e recomendações no reconhecimento de funções .	45
5.6. Desenvolvimento do aplicativo <i>RecPy</i>	51
6. EXPERIMENTOS REALIZADOS	55
6.1. Introdução	55
6.2. Apresentação da metodologia	55
6.3. Procedimentos e experimentos realizados	56

6.4.	Códigos dos exemplos de algoritmos conversíveis no pré-compilador <i>RecPy</i>	57
6.5.	Tokens gerados para os algoritmos testados.....	59
6.6.	Árvores sintáticas abstratas (ASTs) geradas para os algoritmos testados.....	63
6.7.	Apresentação dos gráficos dos experimentos realizados em algoritmos conversíveis no <i>RecPy</i>	70
6.8.	Síntese dos Resultados.....	85
CONCLUSÃO	88
SUGESTÃO DE TRABALHOS FUTUROS	90
ANEXOS	91
7.	ANEXO I - BREVE HISTÓRICO DA RECURSÃO.....	91
7.1.	Introdução	91
7.2.	λ - <i>Calculus</i> (cálculo lambda).....	91
7.3.	Tese de Church-Turing	92
8.	ANEXO II - Resumo da revisão bibliográfica.....	94
APÊNDICES	96
9.	APÊNDICE I – Agradecimentos e menções (continuação).....	96
REFERÊNCIAS	101
10.	Mecanismos de busca e páginas de consulta utilizados para a elaboração deste TCC	101
11.	Leituras recomendadas na Internet	103
12.	Índice de artigos científicos de referência	104
BIBLIOGRAFIA	115
13.	Índice de obras referenciadas	115

INTRODUÇÃO

A genialidade depende de um por cento de inspiração e de noventa e nove por cento de transpiração.

Thomas Alva Edison (atribuída a)

1. APRESENTAÇÃO

Inicialmente idealizado para abranger apenas aspectos teóricos sobre a transformação sistemática de funções recursivas, o escopo deste trabalho foi ampliado para o desenvolvimento de um aplicativo – denominado RecPy – que realizasse a conversão de funções recursivas caudais em suas respectivas modalidades iterativas. Uma vez alcançado sucesso nesse objetivo, o escopo do trabalho foi mais uma vez ampliado, para tratar também de outros tipos de conversões automatizadas: de recursiva não caudal para recursiva caudal; de recursiva não caudal para iterativa; de iterativa para recursiva caudal.

Mantém-se o conteúdo teórico essencial; apresenta-se revisão da literatura científica; indicam-se as vantagens e desvantagens da utilização de algoritmos recursivos; aborda-se o desenvolvimento do aplicativo RecPy, inclusive em aspectos de interesse ao estudo da disciplina de Compiladores, ainda que esse não seja o objetivo deste trabalho; realizam-se experimentos; ilustram-se os resultados desses experimentos com gráficos; conclui-se com a exposição dos resultados.

2. CONCEITOS BÁSICOS

2.1. Recursão

A função recursiva pode ser conceituada como aquela que contém uma ou várias chamadas a si mesma.

As funções recursivas podem ser de dois tipos: recursiva caudal (TR) ou recursiva não caudal (NTR). Apresentaremos, abaixo, exemplos dos dois tipos.

2.2. Recursão não caudal (Non Tail Recursive – NTR)

Funções recursivas não caudais são aquelas que, para retornarem seus resultados, dependem de uma operação posterior à chamada da função.

Exemplo de função recursiva não caudal:

```
def fatNTR(n):
    if n < 2:
        return 1
    return n * fatNTR(n - 1)
```

2.3. Chamada de cauda recursiva ou Recursão Caudal (*Tail Recursive Call – TR*)

Uma função recursiva caudal é aquela em que a função retorna seu resultado sem ser necessário nada além. Como os resultados podem ser entregues imediatamente, sem guardar espaço de memória para operações intermediárias, ganha-se eficiência, pois não há necessidade de se manter, como nas NTR, um quadro de pilha para cada chamada recursiva.

O código NTR exemplificado no Tópico 2.2 foi traduzido assim para TR pelo *RecPy*:

```
def fatTR(n, acc=1):
    if n < 2:
        return acc
    return fatTR(n - 1, n * acc)
```

2.4. Eliminação da chamada de cauda (*tail call elimination*) pela transformação em função iterativa

O RecPy elimina a chamada de cauda transformando-a em um bloco em loop, ou seja, em uma função iterativa.

O código TR exemplificado no Tópico 2.2. foi traduzido assim para IT pelo RecPy:

```
def fatIT(n, acc=1):
    while not n < 2:
        n,acc = n - 1, n * acc
    return acc
```

2.5. Indução matemática

Uma das principais aplicações da indução matemática é provar a validade de fórmulas conjecturadas ou deduzidas. (Santos et al., 2013)^[B20], (Roberts, 1986)^[B19] apresentam textos esclarecedores sobre o assunto.

Importante ressaltar a correlação existente entre os algoritmos recursivos e a noção de indução matemática. Um dos passos mais importantes para se elaborar bons algoritmos recursivos é o estudo de indução matemática. (Graham, 1993)^[B10]

2.6. Algoritmos de divisão e conquista

A estratégia de divisão e conquista resolve um problema seguindo-se estes passos (Dasgupta et al., 2006^[B6]):

1. Quebrando-o em subproblemas que são eles próprios instâncias menores do mesmo tipo de problema;
2. Recursivamente resolvendo esses subproblemas; e
3. Combinando suas respostas apropriadamente.

Algoritmos clássicos que se utilizam dessa técnica: busca binária, Mergesort, multiplicação de matrizes, transformação rápida de Fourier.

Sobre a íntima ligação entre recorrências e o paradigma da divisão e conquista, (Cormen, 2009)^[B5] menciona que recorrências andam de mãos dadas com o paradigma da divisão e conquista porque elas nos dão um modo natural de caracterizar os tempos de execução de algoritmos de divisão e conquista. Uma recorrência é uma equação ou inequação que descreve uma função em termos de seus valores nas entradas menores.

3. OBJETIVOS

3.1. Objetivo geral

Este trabalho tem como objetivo geral oferecer à comunidade científica e acadêmica aplicação destinada ao estudo sistemático de técnicas de conversão de algoritmos recursivos caudais, recursivos não caudais e iterativos. O pré-compilador para Python, denominado *RecPy*, é apresentado para cumprir essa finalidade.

3.2. Objetivos específicos

Abordar os seguintes temas:

- a. Vantagens e desvantagens da utilização de algoritmos recursivos;
- b. Comparação, em termos de eficiência, de algoritmos recursivos caudais, recursivos não caudais e não recursivos (iterativos);
- c. Problemas que podem ocorrer nos algoritmos recursivos, em especial, o estouro de pilha (*stack overflow*) e a perda de eficiência;
- d. Apresentação de exemplos de algoritmos conversíveis no pré-compilador *RecPy*;
- e. Abrangência do reconhecimento de algoritmos e restrições sintáticas impostas na linguagem reconhecida pelo *RecPy*.
- f. Descrição do desenvolvimento do projeto do pré-compilador *RecPy*.

3.3. Motivação e justificativa do trabalho

O estudo das funções recursivas é tema que, além de sua enorme importância em Ciências da Computação, envolve níveis de complexidade em geral bastante elevados. Este trabalho busca oferecer à comunidade científica, técnica e acadêmica algumas colaborações no sentido de facilitar o ensino e o aprendizado de algumas técnicas sistemáticas de conversão entre tipos de funções recursivas. Em especial, o aplicativo *RecPy* oferece a possibilidade de automatização dessas conversões para os padrões que reconhece nesta primeira versão.

3.4. Levantamento de hipóteses

As hipóteses que serão tratadas neste trabalho são especialmente:

- 1) Algumas linguagens, dentre as de uso mais frequente, como Python, por exemplo, apresentam problemas com códigos recursivos, ou seja, apresentam estouro de pilha em recursões mais profundas.
- 2) Há benefícios, em termos de escalabilidade na quantidade de iterações, quando se transforma uma função recursiva em iterativa.
- 3) Existem vantagens das rotinas recursivas caudais em relação às não caudais, no sentido de facilitar a otimização dos códigos.
- 4) Há viabilidade e benefícios, seja em termos de ganhos de eficiência, ou de facilitação de estudos, na utilização de pré-compilador para a tradução automática de rotinas recursivas.

REVISÃO DA LITERATURA CIENTÍFICA

Apresentamos o cotejo entre as características deste trabalho e as da literatura científica pesquisada, relacionada à conversão de funções recursivas.

A maior vantagem de uma abordagem eminentemente prática, sem descuidar de aspectos teóricos, é oferecer ao estudo uma dinâmica estimulante à realização de novos experimentos.

Outro diferencial deste trabalho em relação à maioria dos pesquisados é que não se restringe à temática de otimização de funções recursivas. Trata, também, de questões de interesse no estudo da disciplina de Compiladores.

O quadro a seguir, baseado nos resumos, introduções, conclusões dos próprios autores e/ou em leituras de outras partes dos textos pesquisados, identifica os aspectos relacionados ao presente trabalho.

Referência	Resumo do tema de cada trabalho relacionado	Principal(is) pontos de similaridade	Principais pontos de divergência
Steele e Sussman (1975)^[3]	Apresenta uma implementação de um interpretador para uma linguagem parecida com LISP chamada Scheme. Tal linguagem baseia-se em Lambda-Cálculo (Church), mas extendida para efeitos laterais, multiprocessamento e sincronização de processos. Faz uma revisão de lambda-cálculo. Trata de programas recursivos. Aborda o Continuation Passing Recursion (CPR).	Apresenta técnicas de otimização de funções recursivas; Apresenta códigos de exemplo.	Foco na linguagem Scheme; Não apresenta interface de conversão.
Steele e Sussman (1976)^[4]	Demonstra como modelar construções comuns de programação, em termos de uma linguagem de ordem aplicativa similar à LISP. São apresentados algoritmos de recursão simples, iteração, declarações e expressões compostas, goto e atribuição, “continuation-passing”, expressões	Apresenta técnicas de otimização de funções recursivas; Apresenta códigos de exemplo.	Foco na linguagem Scheme; Apresenta a técnica de “continuation-passing” (CPS);

	de escape, variáveis fluidas, chamadas pelo nome, chamadas por necessidade e chamada por referência.		Não apresenta interface de conversão.
Sickel (1978) ^[6]	Apresenta algoritmo que reescreve em forma eficiente algumas funções recursivas que contêm chamadas redundantes (exemplo: função de Fibonacci). Apresenta também algumas classificações e definições de funções recursivas (linear x não-linear; redundante x não-redundante; primitiva x composta). Exibe também um método para eliminação de recursão.	Apresenta técnicas de otimização de funções recursivas.	Não apresenta interface de conversão; Trata de funções recursivas que contêm chamadas redundantes.
Bird (1980) ^[8]	Examina um número de estratégias gerais para introduzir tabulação em programas recursivos. Explora, por meio de programas de exemplos, as vantagens de introduzir a tabulação como uma técnica de eliminação de recursão.	Apresenta técnicas de otimização de funções recursivas; Apresenta códigos de exemplo.	Não apresenta interface de conversão; Trata de técnicas de tabulação para programas recursivos.
Kleene (1981) ^[9]	Aborda a teoria das funções recursivas, inclusive pelo aspecto histórico. Trata do primeiro teorema de recursão; de um esquema para definições recursivas; de recursão primitiva; algoritmos recursivos; da teoria geral de “recursiveness” de Herbrand-Gödel; de lambda-definibilidade; da Computabilidade de Turing; da “recursiveness” parcial; da tese de Church; do teorema de Church e do teorema de Gödel. Aborda também os cinco axiomas de Peano, influenciados, de sua vez, pelo simples sistema de infinito de Dedekind.	Aborda a teoria de funções recursivas.	Ênfase em aspectos teóricos e históricos do tema; Não apresenta interface de conversão.
Rosser (1982) ^[11]	Trata não somente do lambda-cálculo, mas também de seu relacionado próximo: o cálculo	Aborda a teoria de funções	Ênfase em aspectos teóricos e

	combinatório. Inclui uma curta e simples prova do teorema de Church-Rosser, que não fora ainda publicado e que apareceu em uma impressão limitada em agosto de 1982. Fala sobre as origens do Lambda-cálculo.	recursivas.	históricos do tema;
Boiten (1991)^[21]	Várias descrições de basicamente uma técnica de transformação, nomeadamente <i>acumulação</i> são comparadas. Sua base, nomeadamente a associatividade e a existência de um elemento neutro inerentes a um monoide são identificadas. O artigo apresenta algoritmos de transformações, de factorial, de reversão rápida, de recursão de cauda, de implementação de listas.	A técnica de utilização de uma variável de acumulação também é utilizada no RecPy.	Artigo bastante específico a essa técnica de conversão; Não apresenta interface de conversão.
Baker (1992)^[22]	No que pertine aos objetivos deste trabalho, o artigo trata da recursão de cauda. Sobre ela, menciona que é uma otimização Lisp que foi elevada no dialeto Scheme a um requisito.	Apresenta técnicas de otimização de funções recursivas.	Aprofunda questões relacionadas à alocação e gerenciamento de memória de pilha;
			Bastante focado em Lisp e Scheme;
Kelsey (1993)^[28] [27]	Este artigo examina vários métodos diferentes de implementação de recursão de cauda adequada em um interpretador baseado em pilha, incluindo a passagem de argumentos em um heap, a cópia de argumentos para chamadas recursivas de cauda e coleta de lixo da pilha.	Apresenta técnicas de otimização de funções recursivas.	Bastante focado em linguagens funcionais;
Feeley et al. (1997)^[47]	Artigo originalmente de 1993. Apresenta duas implementações de Scheme desenvolvidas de forma	Apresenta técnicas de otimização de	Foco na linguagem Scheme;

	<p>independente, que foram estendidas para compilar em código C portátil que implementa recursão de cauda completa. Como outros compiladores para idiomas de ordem superior que implementam recursão completa da cauda, as medições desses sistemas indicam uma degradação de desempenho de um fator entre dois e três em comparação com o código nativo emitido pelos mesmos compiladores. Descrevem-se os detalhes da técnica de compilação para uma linguagem não estaticamente tipada (Scheme) e mostra-se que a dificuldade de desempenho surge em grande parte do custo das chamadas de função C.</p>	<p>funções recursivas.</p>	<p>Não apresenta interface de conversão.</p>
Clinger (1998)^[49]	<p>Segundo o autor, este artigo oferece uma definição formal e independente de implementação de recursão de cauda adequada para Scheme. Ele também mostra como uma família inteira de implementações de referência pode ser usada para caracterizar propriedades seguras para espaço relacionadas e prova as desigualdades assintóticas que existem entre elas.</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p>	<p>Foco na linguagem Scheme; Não apresenta interface de conversão.</p>
Ward et al. (1999)^[52]	<p>Neste artigo, descreve-se uma operação de remoção e introdução de recursão poderosa, que descreve sua origem e destino na forma de um sistema de ação (uma coleção de rótulos e chamadas para rótulos). Uma operação de reestruturação simples e mecânica pode ser aplicada a muitos programas iterativos que os colocarão em uma forma adequada para introdução de recursão. A transformação apresentada gera versões estritamente mais iterativas do que os métodos padrão.</p>	<p>Apresenta técnicas de otimização de funções recursivas;</p> <p>Apresenta códigos de exemplo.</p>	<p>Não apresenta interface de conversão.</p>

Liu e Stoller (1999) [57]	<p>Este artigo descreve um método sistemático, baseado na incrementalização, para transformar a recursão geral em iteração: identificar um incremento de entrada, derivar uma versão incremental sob o incremento de entrada e formar um cálculo iterativo usando a versão incremental. Explorar a incrementalização produz computação iterativa de maneira uniforme e também permite que otimizações adicionais sejam exploradas de forma limpa e aplicadas sistematicamente, na maioria dos casos produzindo programas iterativos que usam espaço adicional constante, reduzindo o uso de espaço adicional assintoticamente e rodando muito mais rápido. Resumem-se as principais otimizações, melhorias de complexidade e medidas de desempenho.</p>	<p>Apresenta técnicas de otimização de funções recursivas;</p> <p>Apresenta códigos de exemplo.</p>	Não apresenta interface de conversão.
Yi, Q., Adve, V., & Kennedy, K. (2000) [58]	<p>Apresenta uma (então) nova metodologia de compilação que pode ser utilizada para converter aninhamentos de loops automaticamente. [...] Como efeito colateral deste trabalho, foi desenvolvida uma técnica de melhoria de algoritmo para análise transitiva de dependência (técnica poderosa usada na transformação de recursão e em outras transformações de loops) que é muito mais rápida do que os melhores conhecidos algoritmos em prática.</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p>	Não apresenta interface de conversão.
Nenzén, P., & Ragard, A. (2000) [59]	<p>[...] As versões mais recentes do compilador C, GCC, não suportam o uso de gotos entre funções. A Ericsson precisa desenvolver suporte para uma técnica chamada “tail calls” para substituir o código usando gotos. Este projeto envolve a investigação dos requisitos do GCC</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p>	Não apresenta interface de conversão.

	<p>para apoiar o mecanismo de tratamento de chamadas finais. Como pano de fundo, os autores descreverão em termos gerais, a função e as fases de um compilador e, em particular, as do GCC. Encontramos alguns projetos relacionados que sofrem com a falta de eliminação de chamadas de cauda.</p>		
Schinz e Odersky (2001) [60] [58]	<p>O artigo trata da questão da eliminação de recursão de cauda em uma máquina virtual Java. Apresenta-se também a implementação de um compilador, assim como os resultados experimentais sobre o impacto da aplicação da técnica, em termos de performance e tamanho dos programas compilados.</p>	Apresenta técnicas de otimização de funções recursivas;	O método apresentado é utilizado em um compilador.
Bailey e Weston (2001) [61]	<p>Este documento discute a remoção da recursão de cauda conforme ela se aplica a linguagens procedurais e descreve uma implementação específica em um compilador C. Ele também documenta os benefícios de desempenho dessa otimização e os compara com os benefícios de remover a recursão da cauda manualmente, no nível de origem. Os resultados mostram que essa otimização pode produzir benefícios significativos e, às vezes, pode ser melhor executada pelo compilador do que manualmente. Eles sugerem que, devido ao seu baixo custo e alto benefício, a remoção da recursão da cauda é uma otimização útil em um compilador.</p>	Apresenta técnicas de otimização de funções recursivas;	Descreve a implementação da otimização em um existente compilador C.
Probst (2001) [62]	<p>A linguagem de programação C tem sido usada com sucesso como linguagem alvo de compiladores para várias linguagens de programação. No entanto, a</p>	Apresenta técnicas de otimização de funções recursivas.	Descreve a implementação da otimização em um existente compilador C.

	compilação de linguagens de programação funcionais como Scheme e linguagens lógicas como Prolog para C é complicada pela falta de suporte para chamadas adequadas cauda em C. Chamadas adequadas de cauda são chamadas de ação que reutilizam o frame de pilha do chamador para o receptor. Muitas linguagens de programação de nível superior exigem que todas as chamadas de função sejam implementadas como chamadas finais adequadas. Este trabalho explica por que a convenção de chamada C padrão não pode suportar chamadas finais adequadas no caso geral e apresenta uma convenção de chamada que suporta chamadas finais adequadas para C.	
Minamide (2003)^[67]	Trata da técnica denominada “trampolim”: “Espera-se que as chamadas finais não consumam espaço de pilha na maioria das linguagens funcionais. No entanto, não há suporte para chamadas finais em alguns ambientes. Mesmo em tais ambientes, chamadas finais adequadas podem ser implementadas com uma técnica chamada trampolim[...]”.	Apresenta técnicas de otimização de funções recursivas. Aborda a técnica de otimização denominada trampolim;
Baldwin (2003)^[68]	Descreve um pequeno compilador para fins didáticos.	Incorporou a técnica no compilador MLj.
Himpe et al. (2003)^[72]	Resumo dos autores: “Neste artigo, um método para remover a recursão de algoritmos é demonstrado. O método para remover a recursão é baseado em manipulações algébricas de um modelo matemático do fluxo de controle. O método não se destina	Apresenta técnicas de otimização de funções recursivas; Abordagem didática. Apresenta um mini-compilador denominado MinimL.

	<p>a resolver todos os problemas de remoção de recursão possíveis, mas em vez disso, pode ser visto como uma ferramenta em uma caixa de ferramentas maior de transformações de programa. O método pode manipular certos tipos de recursão que não são facilmente manipulados por métodos existentes, mas pode ser um exagero para certos tipos de recursão onde métodos existentes podem ser aplicados, como recursão de cauda”.</p>	
Cowles e Gamboa (2004)^[80]	<p>Trata de recursão caudal: “É mostrado que a existência de uma função total única satisfazendo um axioma de definição recursivo de cauda garante que a recursão sempre pare. Isso está em contraste com o caso geral, quando o adjetivo cauda não precisa se aplicar à recursão: A existência de uma função total única que satisfaça um axioma definicional recursivo (geral) não precisa forçar a recursão a sempre terminar. [...]”</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p> <p>Não apresenta interface de conversão.</p>
Tang (2006)^[83]	<p>Apresenta uma transformação de otimização de compilador chamada em linha completa (complete inlining) para em linha (inline) e eliminação de recursões de cauda. A em linha completa pode eliminar as chamadas recursivas que não podem ser eliminadas pela eliminação de recursão de cauda. Pode colocar completamente em linha as chamadas recursivas, o que as procedures em linha existentes podem fazer apenas parcialmente.</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p> <p>Aborda a técnica de otimização denominada <i>complete inline</i>;</p> <p>Não apresenta interface de conversão.</p>
Gao, Y., & Guan, F. (2008)^[89]	<p>Este artigo descreve como usar a fila para criar algoritmos de não recursão da árvore de links binários. Quanto a uma árvore binária geral, se adotarmos o armazenamento de sequência, primeiro devemos</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p> <p>Utiliza técnicas de árvore binária;</p> <p>Não apresenta interface de conversão.</p>

<p>estendê-la em uma árvore binária completa, em segundo lugar, armazenamos em uma fila temporária de acordo com a sequência de cima para baixo e esquerda-direita. Com base nas propriedades da árvore binária completa e da fila, se pudermos confirmar todos os elementos na fila, então podemos encontrar os filhos esquerdo e direito do elemento na fila. Quando recorremos a essas etapas, podemos criar uma árvore de bits de link binário. Este algoritmo enriquece o método de recursão para não recursão.</p>	Rubio-Sánchez et al. (2008)^[90]	<p>Cuida da recursão mútua em um contexto didático: “Recursão é um tópico importante em currículos de ciência da computação. Diz respeito à aquisição de competências no que diz respeito à decomposição de problemas, abstração funcional e conceito de indução. Em comparação com a recursão direta, a recursão mútua é considerada mais complexa. Conseqüentemente, geralmente é abordado superficialmente em cursos de programação CS1 / 2 e livros didáticos. Mostramos que, quando um problema é abordado de forma adequada, não só a recursão mútua pode ser uma ferramenta poderosa, mas também pode ser fácil de entender e divertida. Este artigo fornece vários algoritmos intuitivos e atraentes que dependem de recursão mútua e que foram projetados para ajudar a fortalecer a capacidade dos alunos de decompor problemas e aplicar indução”.</p>	Apresenta técnicas de otimização de funções recursivas;	Abordagem didática.	Trata de recursão mútua, não abordada neste TCC;	Não apresenta interface de conversão.
Zeugmann e Zilles (2008)^[87]	<p>Outro artigo que trata do tema recursão em um contexto didático: “O estudo da capacidade de aprendizado de classes de funções</p>	Aprendizado de funções recursivas.	Estudo eminentemente teórico, repleto de definições formais,			

	<p>recursivas atraiu considerável interesse por pelo menos quatro décadas. Iniciando com o modelo de aprendizagem de Gold (1967) no limite, muitas variações, modificações e extensões foram propostas. Esses modelos diferem em alguns dos seguintes: o modo de convergência, os requisitos que as hipóteses intermediárias devem cumprir, o conjunto de estratégias de aprendizagem permitidas, a fonte de informação disponível para o aluno durante o processo de aprendizagem, o conjunto de espaços de hipóteses admissíveis e os objetivos de aprendizagem”.</p>	<p>teoremas e provas.</p>
Schwaighofer (2009)^[94]	<p>Apresenta métodos de implementação de otimização de chamada caudal para Java HotSpot™ : “Muitas implementações de linguagem de programação compilam para bytecode Java, que é executado por uma máquina virtual (por exemplo, o Java HotSpot™ VM). Entre essas línguas estão linguagens funcionais, que requerem uma otimização que garante que certos tipos de chamadas de método não façam com que a pilha de execução cresça ilimitadamente. Esta otimização é chamada de otimização de chamada final e atualmente não é suportada pelo Java HotSpot™ VM. Implementações de linguagens funcionais têm que recorrer a técnicas alternativas para garantir que o espaço de pilha não aumente sem limites. Essas técnicas complicam a implementação e também geram uma penalidade de desempenho. Esta tese apresenta técnicas de suporte à otimização de tail call na máquina virtual Java HotSpot™. [...]”</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p> <p>As técnicas apresentadas são utilizadas no contexto da Java HotSpot™ Virtual Machine.</p>

Rubio-Sánchez (2010)^[97]	<p>Proposta didática do autor (trecho): “Este artigo propõe uma metodologia simples para projetar funções de recursão de cauda usando uma abordagem declarativa e o conceito de generalização de função. Fizemos uma avaliação da técnica com alunos do segundo e terceiro anos de ciência da computação. Os resultados sugerem que este novo ponto de vista melhora a capacidade dos alunos de projetar programas recursivos de cauda, os ajuda a compreender a distinção entre os paradigmas imperativo e declarativo e pode reforçar suas habilidades de programação em geral. [...]”</p>	<p>Apresenta técnicas de otimização de funções recursivas;</p> <p>Abordagem didática.</p>	<p>Não apresenta interface de conversão.</p>
Thivierge e Feeley (2012)^[105]	<p>Este artigo descreve uma abordagem para compilar chamadas finais (de cauda) de Scheme e continuações de primeira classe para JavaScript, uma linguagem dinâmica sem esses recursos. A abordagem é baseada no uso de uma representação intermediária de máquina virtual personalizada simples que é traduzida para JavaScript.</p>	<p>Apresenta técnicas de otimização de funções recursivas;</p>	<p>Foco em Scheme e Javascript;</p> <p>Não apresenta interface de conversão.</p>
Bjarnason (2012)^[106]	<p>Trata da eliminação da chamada final (TCE) no compilador Scala. Essa eliminação é limitada a métodos auto-recursivos, mas as chamadas finais não são eliminadas. Isso torna as funções compostas de muitas funções menores sujeitas a estouros de pilha. Segundo o autor, ter um mecanismo TCE geral seria um grande benefício no Scala, particularmente para programação funcional. Apresenta a técnica de Trampolim, que é, segundo diz, uma técnica popular, que pode ser usada para TCE em linguagens que não o suportam nativamente. Este artigo fornece uma introdução aos trampolins em Scala e expande essa solução para obter a eliminação de</p>	<p>Apresenta técnicas de otimização de funções recursivas;</p>	<p>Foco no compilador Scala;</p> <p>Não apresenta interface de conversão.</p>

	qualquer chamada de método, mesmo as chamadas que não estão na posição final. Isso elimina completamente o uso da pilha de chamadas em programas Scala.		
Sonnex, W., Drossopoulou, S., & Eisenbach, S. (2012) [107]	Zeno é uma [então] nova ferramenta para a geração automática de provas de propriedades simples de funções sobre estruturas de dados definidas recursivamente. Ele pega um programa Haskell e uma afirmação como objetivo e tenta construir uma prova para esse objetivo. Se for bem-sucedido, ele converte a prova em código Isabelle. [...] Nossa nova heurística visa promover a aplicação de definições de funções e evitar a repetição de etapas de prova semelhantes. [...]	Apresenta técnicas de otimização de funções recursivas.	Foco em Haskell e Isabelle; Não apresenta interface de conversão.
Nishida e Vidal (2014) [111]	O resumo dos autores é esclarecedor e sintético: “As funções recursivas da cauda são um tipo especial de funções recursivas em que a última ação em seu corpo é a chamada recursiva. A recursão da cauda é importante por uma série de razões (por exemplo, elas geralmente são mais eficientes). Neste artigo, apresenta-se uma transformação automática de funções de primeira ordem na forma recursiva de cauda. As funções são definidas usando um sistema de reescrita de termos (de primeira ordem). Prova-se a exatidão da transformação para redução baseada em construtor em relação a sistemas de construtor (ou seja, programas funcionais típicos de primeira ordem)”.	Apresenta técnicas de otimização de funções recursivas.	Apresenta a técnica do sistema de reescrita de termos para transformar recursões gerais em recursões de cauda.
Rinderknecht (2014) [113]	Trata da literatura sobre ensino e aprendizagem de programação recursiva. Após breve histórico do advento da recursão em linguagens de programação e sua adoção por	Apresenta técnicas de otimização de funções recursivas;	Não apresenta interface de conversão.

	<p>programadores, apresenta abordagens curriculares para recursão, incluindo revisão de livros e algumas metodologias de programação, bem como os paradigmas funcional e imperativo e a distinção entre fluxo de controle vs fluxo de dados.</p> <p>Resumo dos autores: “Em engenharia de software, fazer uma boa escolha entre recursão e iteração é essencial porque sua eficiência e manutenção são diferentes. Na verdade, os desenvolvedores geralmente precisam transformar iteração em recursão (por exemplo, na depuração, para decompor o gráfico de chamadas em iterações); portanto, é bastante surpreendente que não exista uma transformação pública de loops em recursão que trate todos os tipos de loops. Este artigo descreve uma transformação capaz de transformar loops iterativos em métodos recursivos equivalentes. A transformação é descrita para a linguagem de programação Java, mas é geral o suficiente para ser adaptada a muitas outras linguagens que permitem iteração e recursão. [...]</p>	<p>Abordagem didática.</p> <p>Apresenta técnicas de otimização de funções recursivas;</p> <p>Apresenta exemplos de algoritmos.</p> <p>Não apresenta interface de conversão.</p>
Insa e Silva [114]		

	<p>Resumo do autor: “A recursão é um tanto enigmática, e os exemplos usados para ilustrar a ideia de recursão costumam enfatizar três algoritmos: Torres de Hanói, Fatorial e Fibonacci, muitas vezes sacrificando a exploração do comportamento recursivo pela noção de que uma "função chama a si mesma". Muito pouco esforço é gasto em algoritmos recursivos mais interessantes. Este artigo analisa como três algoritmos de recursão menos conhecidos podem ser usados no ensino de aspectos comportamentais da recursão: o problema de Josephus, a sequência</p>	<p>Apresenta técnicas de otimização de funções recursivas;</p> <p>Abordagem didática.</p> <p>Não apresenta interface de conversão.</p>
Wirth [123]		

	<p>de Hailstone e a função de Ackermann”.</p>		
Tauber et al. (2015)^[130]	<p>Resumo dos autores: “Este artigo apresenta FCore: uma implementação JVM do Sistema F com suporte para eliminação completa de chamada final (TCE). Nossa técnica de compilação para FCore é inovadora em dois aspectos: ela usa uma nova representação para funções de primeira classe chamadas de objetos funcionais imperativos; e fornece uma maneira de fazer TCE na JVM usando espaço constante. Ao contrário das técnicas convencionais de TCE na JVM, os objetos de função alocados são reutilizados em cadeias de chamadas finais. Portanto, os programas escritos em FCore podem usar estilos de programação funcional idiomática, dependendo do TCE, e ter um bom desempenho sem se preocupar com as limitações da JVM.[...].”</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p>	<p>Foco em JVM e FCore;</p> <p>Não apresenta interface de conversão.</p>
Madsen et al. (2018)^[146]	<p>Trechos do resumo dos autores: “A Java Virtual Machine (JVM) oferece um ambiente de tempo de execução atraente para implementadores de linguagem de programação. [...] No entanto, a JVM foi originalmente projetada para Java, e seu representante reenvio de código e dados pode causar dificuldades para outras linguagens. Neste artigo, discutimos como implementar de forma eficiente linguagens funcionais na JVM. Nós nos concentrarmos em dois desafios gerais: (a) como representar com eficiência os tipos de dados algébricos na presença de tags e tuplas, tipos de opções, novos tipos e parâmetros paramétricos polimorfismo e (b) como oferecer suporte à eliminação de chamadas de cauda completa na JVM. Apresentamos duas contribuições</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p>	<p>Foco em linguagens funcionais para JVM;</p> <p>Não apresenta interface de conversão.</p>

	técnicas: uma representação fundida de tags e tuplas e uma estratégia de eliminação de chamadas de cauda completa que é thread-safe”.	
Ralston (2019) ^[148]	<p>Resumo do autor: “A eliminação da chamada final é usada por linguagens e compiladores para otimizar a ativação de métodos na posição final. Embora essa otimização tenha sido a fonte de muitas pesquisas, ela não foi implementada anteriormente na máquina virtual OpenSmalltalk - a máquina virtual Smalltalk de código aberto usada por ambientes Smalltalk, como Pharo e Squeak. Existem muitas abordagens descritas na literatura para implementar a eliminação de chamadas finais, como a remoção de stack frames na ativação do método em vez do retorno do método. Duas implementações de eliminação de chamada final usando uma abordagem de remoção de frame de pilha são apresentadas para o Opensmalltalk VM. Uma implementação é apresentada para o intérprete e outra para o compilador Cog JIT. Essas implementações são testadas com cenários ideais e reais e mostram melhorias no tempo de execução e no uso de memória”.</p>	<p>Apresenta técnicas de otimização de funções recursivas.</p> <p>Foco em OpenSmallTalk VM;</p> <p>Não apresenta interface de conversão.</p>

Quadro 1 - Resumo de artigos relacionados à otimização de códigos recursivos ou à conversão de funções recursivas em iterativas

CONVERSÃO AUTOMÁTICA DE FUNÇÕES RECURSIVAS: APLICATIVO RECPY

4. INTRODUÇÃO

O método recursivo é desejável em diversas aplicações, pois produz algoritmos mais compactos, concisos e frequentemente mais naturais sob o ponto de vista matemático, especialmente nos casos em que há facilidade de argumentar a correção de determinada solução na forma de indução matemática. Por isso, contribui tanto para a clareza na leitura, quanto para a facilidade e robustez da prova de correção dos resultados produzidos.

Os algoritmos recursivos são, normalmente, as melhores e mais naturais opções para o uso com listas encadeadas, árvores, e problemas específicos, como ordenação por intercalação (*mergesort*), entre muitos outros.

Mas, como ocorre em muitos campos do conhecimento, e especial em Programação, são raros ou inexistentes os casos em que uma única técnica serve de panaceia para resolver todos os problemas. Os algoritmos recursivos também podem apresentar desvantagens que tornem sua utilização não recomendada.

Os algoritmos escritos recursivamente provocam, frequentemente, o problema do estouro de pilha quando utilizados em entradas que exigem muitas chamadas recursivas. Além disso, há aplicações que podem tornar sua escrita e leitura mais complexas e difíceis do que nos modelos similares iterativos.

Assim, é importante ao desenvolvedor ou analista conhecer bem as situações ideais para sua adoção ou sua substituição por outro modelo. Muitos compiladores já estão preparados para a otimização de funções recursivas. O que se faz com frequência, de modo transparente ao usuário, é tornar as chamadas recursivas em iterativas no código final gerado.

Comumente, as linguagens não oferecem essa conversão como padrão e é preciso que sejam feitas manualmente. Há também linguagens que não estão sequer preparadas para a otimização automática de recursões caudais. Nesses casos, há necessidade de se converterem os algoritmos recursivos em iterativos, o que normalmente resolve o problema do estouro de pilha.

Python é uma dessas linguagens que, apesar de ter inúmeros bons atributos, tem, em sua concepção, a discordância em relação à eliminação das recursões caudais (da *TRE – Tail Recursion Elimination*).

*Recursion Elimination ou TCE – Tail Call Elimination)*¹. Por isso mesmo, foi a escolhida como linguagem objeto do pré-compilador *RecPy*, destinado ao estudo das conversões entre algoritmos recursivos (caudais e não caudais) e iterativos. Esse aplicativo realiza automaticamente as conversões de funções escritas nos padrões reconhecíveis.

Além de Python, há outras implementações de linguagens conhecidas que não suportam nativamente a eliminação da recursão caudal: podemos citar, por exemplo, a JVM do JAVA, Javascript, Rust, Go, Dart, Scala. Portanto, os conhecimentos adquiridos com o auxílio do *RecPy* podem também servir para uso em linguagens que, como essas, requeiram tratamento manual do código para fins de otimização.

4.1. A série de artigos de Moertel

No primeiro artigo da série que inspirou o tema deste TCC, Moertel (2013)^[L5] apresenta o que ele chama de “Os truques (ou artifícios) do negócio (*The tricks of trade*): recursão para iteração, parte 1: o método simples, recursos secretos e acumuladores.”

O “método simples” pode ser sintetizado no seguinte trecho:

Este método de conversão funciona em muitas funções recursivas simples. Quando funciona, o faz bem, e os resultados são leves e rápidos. Geralmente, tento-o primeiro e cogito de métodos mais complicados apenas quando este falha.

Em poucas palavras:

1. Estude a função.
2. Converta todas as chamadas recursivas em chamadas finais. (Se você não pode, pare. Tente outro método.)
3. Introduza um loop *one-shot* [de uma vez, sem continuação] em torno do corpo da função.
4. Converta chamadas finais em instruções contínuas.
5. Arrume [reorganize a estrutura da função].

Uma propriedade importante desse método é que ele é incrementalmente correto. Após cada etapa, você tem uma função equivalente à original. Por isso, se tiver testes de unidade, pode executá-los após cada passo para se certificar de que não cometeu um erro.

Na parte 2 do artigo, Moertel (2013) apresenta um procedimento “mecânico” (como ele mesmo chama) e repetível de transformação de uma função recursiva original em outra versão iterativa “rápida, eficiente e preservada em suas propriedades”.

Na parte 3, são apresentadas técnicas de eliminação de chamadas recursivas em algoritmos de travessia em estruturas de dados do tipo árvore (*tree traversal*).

A parte 4 apresenta o método de trampolim (*trampoline*), que consiste, resumidamente, em “manualmente remover da pilha o quadro de execução (*frame*) atual e

¹ Ver, a propósito, o artigo do criador da linguagem, Guido Van Rossum, denominado: “**Neopythonic: Tail Recursion Elimination**”. Fonte: < <http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html>> (acessado em 05/12/2020).

eliminar o crescimento da pilha antes de se realizar uma chamada de cauda”.

Este TCC não aborda todas as técnicas contidas nessa série de artigos, mas indica sua leitura.

4.2. Vantagens e desvantagens da utilização de algoritmos recursivos. Quando e como utilizá-los e quando evitá-los

4.2.1. Vantagens e importância da utilização de algoritmos recursivos

- a) Recursão é um poderoso método de resolução de problemas usado largamente em computação científica;
- b) Especificações recursivas frequentemente produzem soluções elegantes e mais fáceis de entender do que as respectivas especificações iterativas. Isso é especialmente verdade para problemas de definição naturalmente recursiva;
- c) Recursão frequentemente torna mais claros programas complexos e, em alguns casos, pode ser muito eficiente (exemplo: busca binária e *MergeSort*);
- d) Algoritmos recursivos são uma solução natural em implementações que envolvam manipulações de árvores, em razão de sua própria estrutura.
- e) A recursão oferece um exemplo excepcional de resolução de problemas através do princípio de dividir e conquistar, dividindo um problema em subproblemas menores, resolvendo-os de forma independente e combinando suas soluções para chegar a uma solução para o problema original;
- f) Algoritmos recursivos são comumente mais concisos, o que possibilita redução do tamanho do código fonte, e, assim, o torna mais legível;
- g) Possibilidade do uso de induções matemáticas para comprovação de seu funcionamento correto. Além disso, algoritmos recursivos levam a modelos matemáticos que podemos usar para entender o desempenho (SEDGEWICK e WAYNE, 2014)^[B23].

4.2.2. Desvantagens da utilização de algoritmos recursivos

- a) Funções recursivas aumentam o uso intensivo de uso da memória de pilha (Skliarova e Sklyarov, 2009) [88];
- b) Normalmente, tendem a ser mais lentos do que as equivalentes implementações iterativas, justamente por usarem intensivamente a memória de pilha;
- c) Especialmente quando há repetição de subproblemas, tornam-se ineficientes. Quando isso ocorre, é necessário o uso de técnicas de memorização (memoization) para compensar;
- d) Estão propensos ao estouro de pilha;
- e) Algoritmos recursivos são mais difíceis de serem depurados, especialmente quando for alta a profundidade de recursão.

5. PRÉ-COMPILADOR *RecPy*, PARA CONVERSÃO DE FUNÇÕES RECURSIVAS

5.1. Apresentação do *RecPy*

5.1.1. Introdução

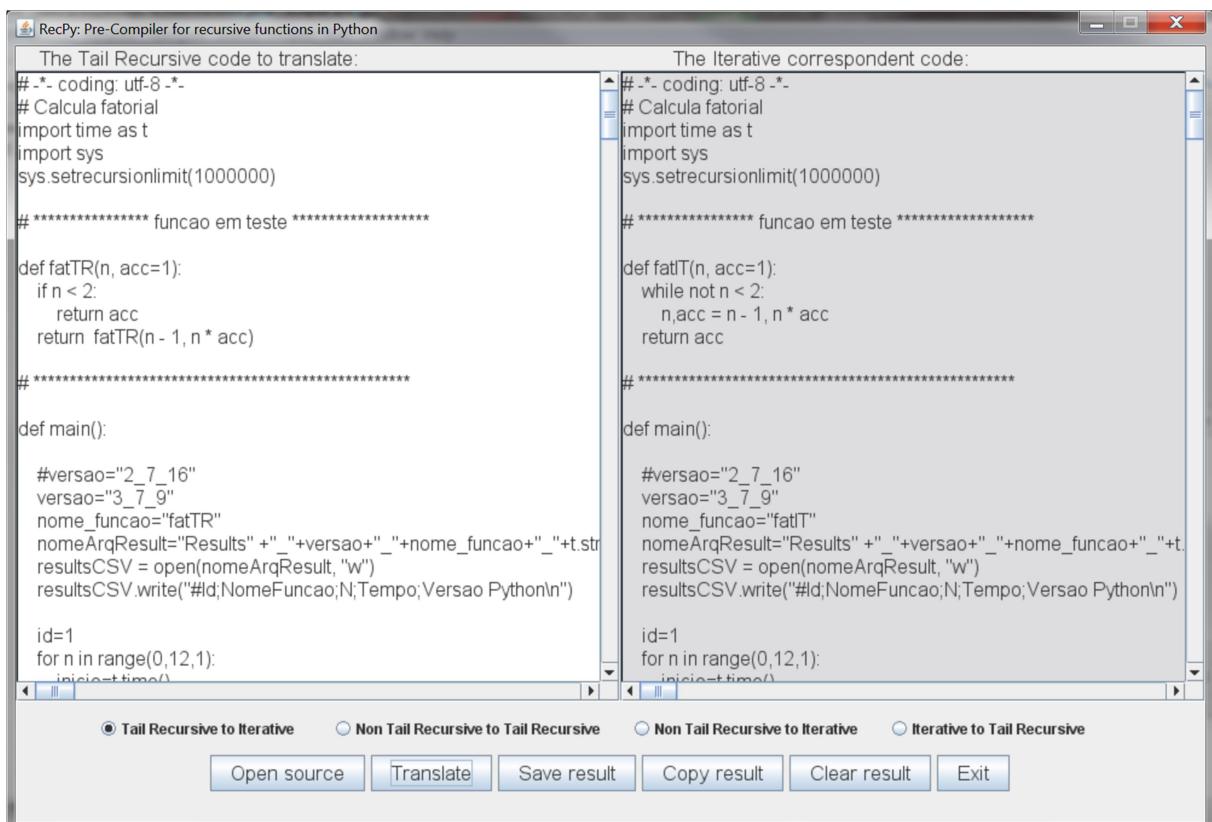


Figura 1- Tela de apresentação do *RecPy*

Durante o desenvolvimento das pesquisas necessárias à confirmação das hipóteses levantadas neste trabalho (Tópico 3.4) foi desenvolvido o pré-compilador *RecPy*. Este aplicativo destina-se não somente ao propósito de simplificar a conversão de algoritmos recursivos, mas também pode ser utilizado para fins de estudo de funções recursivas, ou até mesmo, embora esse não seja o objetivo planejado, servir potencialmente como material de estudo para a disciplina de Compiladores.

Python foi escolhida como linguagem alvo do pré-compilador *RecPy*, em razão de que, apesar de ser excelente em diversos aspectos, requer atenção no tratamento da otimização de funções recursivas.

5.1.2. Python: como trata a questão da otimização de recursões caudais

O criador da linguagem, Guido Van Rossum, no artigo Neopythonic [L3], ao tratar da eliminação da recursão caudal (*tail recursion elimination* – TRE), defendeu, em artigo de 2009, seu posicionamento contrário à incorporação do recurso de TRE pela linguagem, sob os seguintes argumentos:

Primeiro, porque TRE é incompatível com uma boa estrutura de rastreamento de pilha. Assim, torna o procedimento de depuração do código mais difícil. Entende que Python é, e deve permanecer, uma linguagem fácil de depurar.

Segundo, pois vê na ideia de a TRE ser considerada mera otimização a possibilidade de os desenvolvedores começarem a escrever código que depende disso e seus códigos não irão rodar em implementações que não provejam isso.

Terceiro, pois não acredita na recursão como a base de toda programação. Esta é, segundo ele, uma crença fundamental de certos cientistas da computação, mas recursão seria apenas uma boa abordagem teórica para a matemática fundamental, não uma ferramenta do dia-a-dia.

E conclui esse terceiro argumento dizendo que as listas no estilo *Python* e as sequências em geral são muito mais úteis do que recursões... E assevera: “usar uma lista ligada (*linked list*) para representar uma sequencia de valores é distintamente não *pythonic* e, na maioria dos casos, muito ineficiente”.

Por último, afirma que o compilador Python não pode implementar confiavelmente a TRE, pois não pode determinar se alguma chamada particular é de fato referência para a função atual apenas se parece ter o mesmo nome. E também não pode fazê-lo em tempo de compilação.

Naturalmente, todos pontos levantados acima são controversos. Não será objetivo deste trabalho contestar a visão do criador da linguagem, mas apenas tratar de questões de otimizações em algoritmos recursivos que, salvo mudança radical de posição, nunca serão implementadas automaticamente pelo *Python*.

5.2. Escopo de conversões possíveis no RecPy

O *RecPy* é um pré-compilador para Python que realiza quatro tipos de conversão específicos, e com as especificações e restrições apresentadas no Item 5.5 :

- 1) a conversão de função recursiva caudal em função iterativa (TR-IT);
- 2) a conversão de função recursiva não caudal em função recursiva caudal (NTR-TR);
- 3) a conversão de função recursiva não caudal em função iterativa (NTR-IT);
- 4) a conversão de função iterativa em função recursiva caudal (IT-TR).

5.3. Como utilizar o RecPy

O arquivo Java executável do aplicativo RecPy – RecPy.jar – encontra-se na pasta \RecPyJarExecutavel do Github (<https://github.com/robertocsa/TCC_CComp_UERJ>). Para executá-lo, a biblioteca antlr-runtime-4.9.1.jar deve estar na mesma pasta.

Inicialize o aplicativo com um duplo clique. O campo com fundo branco, situado à esquerda, deve receber o código a ser traduzido. Há dois métodos de preenchimento desse campo: 1) copiar um bloco de função recursiva ou iterativa para a área de transferência (CTRL-C) e, em seguida, colá-lo, com CTRL-V, com o cursor focado no referido campo, ou: 2) carregando um arquivo de script Python, como indicado a seguir.

Caso escolha o método 2 acima, abra um arquivo de script Python (*.py ou *.pyw) que contenha a(s) função(ões) a ser(em) traduzida(s). Para isso, clique no botão “Open source”.

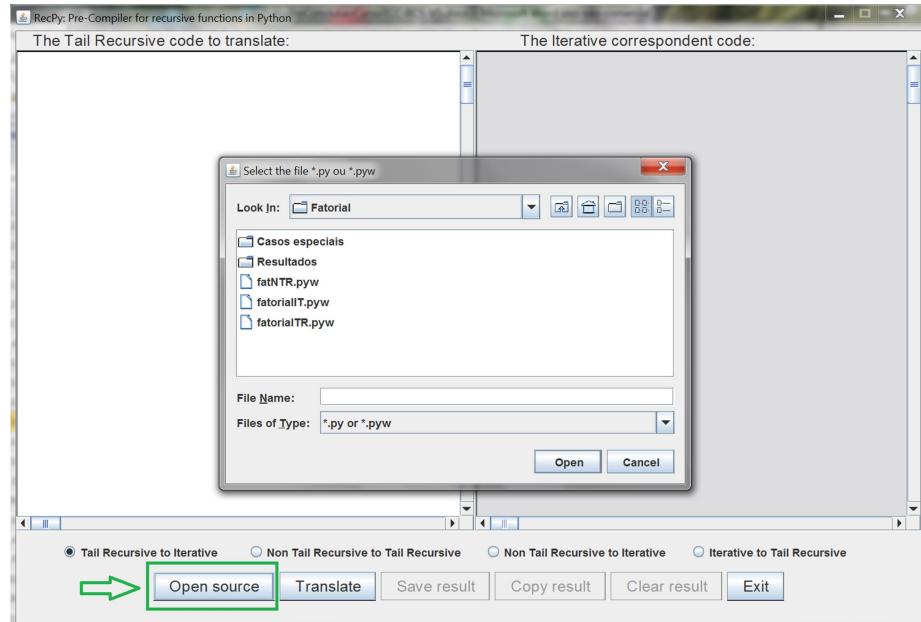


Figura 2 - Explicação sobre o carregamento do arquivo fonte (*.py ou *.pyw)

O arquivo poderá conter outros códigos além do bloco de função recursiva. O que não fizer parte desse bloco será simplesmente copiado para o campo de resultado, sem nenhuma alteração.

Carregue um arquivo compatível com o tipo de tradução a realizar. No exemplo, mostraremos a conversão TR-IT, por intermédio do carregamento do arquivo factorialTR.pyw:

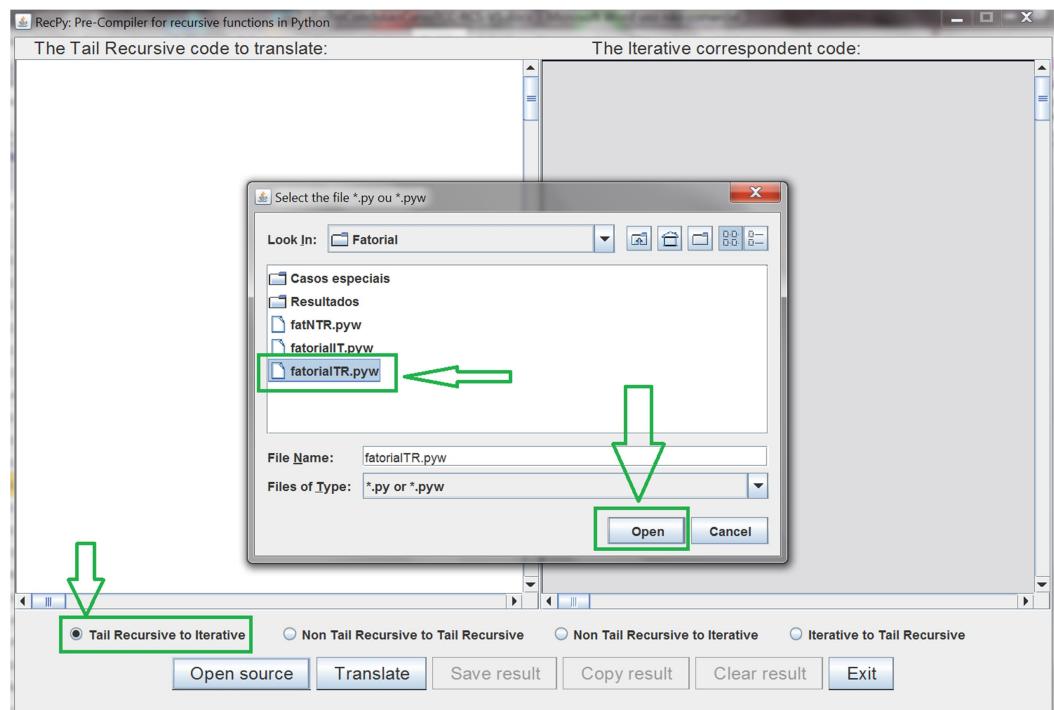


Figura 3 - Exemplo de abertura de arquivo para tradução de função recursiva caudal (TR) para iterativa (IT)

Após carregado o arquivo fonte, clique no botão *Translate*:

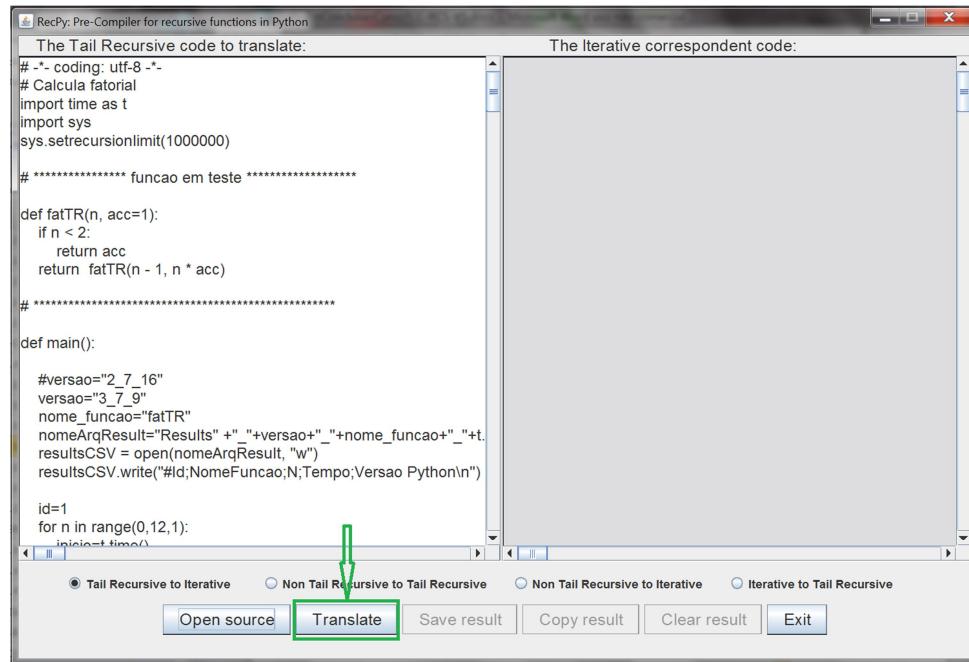


Figura 4 - Clique no botão Translate para realizar a tradução

O resultado da tradução aparecerá no campo com fundo cinza à direita. Salve o código traduzido em um arquivo de script (*.py ou *.pyw) ou copie-o para a área de transferência com CTRL-C ou com um clique no botão Copy result:

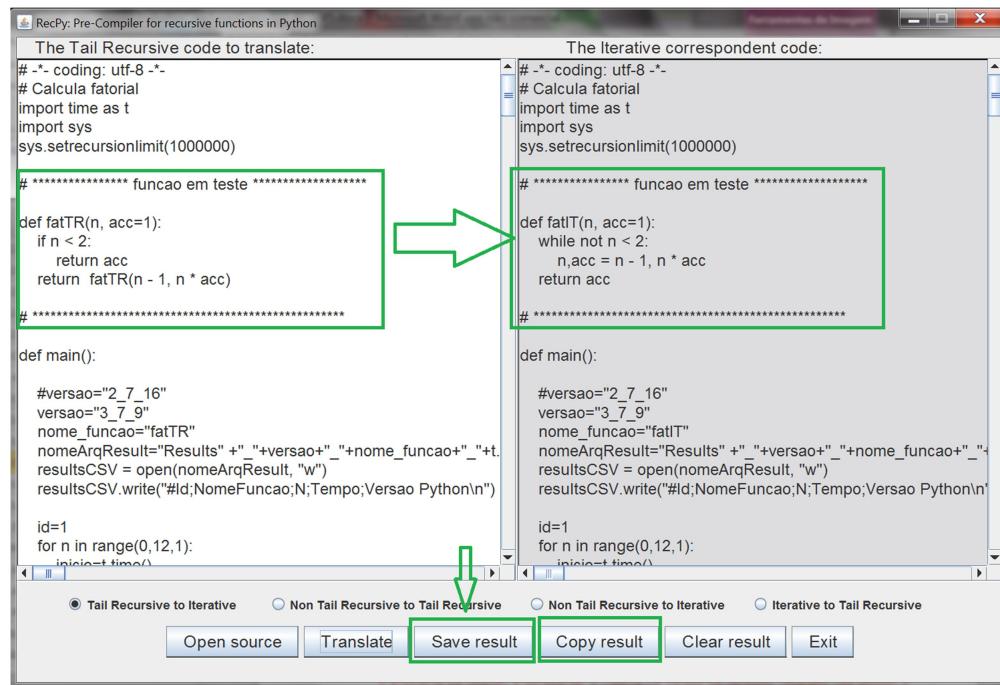


Figura 5 - Salve o resultado em um arquivo de script ou copie-o para a área de transferência

Caso o usuário deseje, poderá apagar o resultado de uma conversão anterior, antes de realizar uma nova, clicando no botão “*Clear result*”. Mesmo que não faça isso, o

aplicativo o fará automaticamente.

Para sair do aplicativo, clique no botão “*Exit*”.

5.4. Formatos de funções recursivas reconhecíveis no *RecPy*

Nesta versão inicial, o *RecPy* converte classes bem específicas de funções, em *Python*, nos seguintes formatos:

1) Funções recursivas caudais simples:

```
def <nome_da_funcao_recursiva_caudal>(<lista de parâmetros 1>):
    <bloco de códigos intermediários ou comentários>
    if <condição>:
        <bloco de códigos intermediários ou comentários>
        return <variável_de_retorno>
    else:
        <bloco de códigos intermediários ou comentários>
        return <nome_da_funcao_recursiva_caudal>(<lista de parâmetros 2>)
```

2) Funções recursivas não caudais simples:

```
def <nome_da_funcao_recursiva_nao_caudal>(<lista de parâmetros 1>):
    <bloco de códigos intermediários ou comentários>
    if <condição>:
        <bloco de códigos intermediários ou comentários>
        return <variável_de_retorno>
    else:
        <bloco de códigos intermediários ou comentários>
        return <expr> <operador> <nome_da_funcao_recursiva_nao_caudal>(<lista de
parâmetros 2>) OU return <nome_da_funcao_recursiva_nao_caudal>(<lista de parâmetros 2>)
<operador> <expr>
```

3) Funções iterativas simples:

```
def <nome_da_funcao_iterativa>(<lista de parâmetros 1>):
    <bloco de códigos intermediários ou comentários>
    while <condição>:
        <bloco de códigos intermediários ou comentários>
        <lista de parametros1> = <lista de parametros 2>
    return <expressao_de_retorno>
```

5.5. Requisitos, restrições, observações e recomendações no reconhecimento de funções

- 1) Esta versão do *RecPy* reconhece somente os padrões especificados acima, no Tópico 5.4 (Formatos de funções recursivas reconhecíveis no *RecPy*).
- 2) Nas funções recursivas traduzidas, trocam-se os nomes originais das respectivas funções. Por exemplo, na conversão de uma função iterativa (IT) para recursiva caudal (TR), se o nome da função for “fatIT”, a função recursiva caudal será renomeada para “fatTR”. Ou seja, no exemplo, todas as ocorrências de “IT” serão substituídas por “TR” no nome da função. Ao realizar essas substituições, é possível que outros nomes que contenham TR, IT ou NTR, a depender do tipo de função recursiva em tradução, venham também a ser alterados. Portanto, recomenda-se reservar essas combinações de letras exclusivamente para nomenclatura de funções.
- 3) Na tradução de função recursiva caudal para iterativa (TR-IT), as variáveis da lista de parâmetros 1 devem estar na mesma ordem das variáveis respectivas da lista 2, ou seja, em um exemplo:

Parâmetros da lista 1: (A, N, a);

Parâmetros da lista 2: (A, N-1, A[N]+a).

Assim, por exemplo, este trecho de código abaixo é reconhecido sem problemas, pois as listas de parâmetros estão seguindo a mesma ordem no posicionamento de suas variáveis de origem e de destino na penúltima linha do código traduzido (A recebendo o valor de A, N de N-1, a de A[N]+a):

```
# Recursiva caudal
def soma(A, N, a): # (primeira linha não muda nada)
    if N==0:          # (o if converte-se em while not)
        return a       # (esta linha passa para o final)
    else:             # (apaga-se esta linha)
        # ...           #(mantém-se o mesmo código, após a linha do while not)
        return soma(A, N-1, A[N]+a) # (passa para a penúltima linha, em conjunto com a
                                     # primeira lista de parâmetros, conforme se vê abaixo )
```

A função recursiva acima exemplificada converte-se na seguinte função iterativa:

```
# Função Iterativa correspondente:
def soma(A, N, a):
    while not N==0:
        # ...
        A, N, a = A, N-1, A[N]+a
    return a
```

- 4) Os IFs ou WHILEs, para serem traduzíveis pelo *RecPy*, não podem estar situados na mesma linha do comando subsequente. Exemplo:

Trecho de código reconhecível no <i>RecPy</i>	Linha de código não reconhecível pelo <i>RecPy</i>
if (L==[]): return acc	if (L==[]): return acc

Quadro 2 - Exemplo de código a ser evitado (if ou while na mesma linha do comando subsequente)

- 5) As funções recursivas, para serem traduzíveis pelo *RecPy*, não devem conter mais do que uma linha de IF ou de WHILE. Ver exemplos abaixo:

Função reconhecida e traduzível pelo <i>RecPy</i>	Função não reconhecida pelo <i>RecPy</i> , por fugir ao padrão aceito O primeiro exemplo produz como resultado a mesma função de entrada, sem tradução e o segundo exemplo retorna linhas em branco.
Exemplo 1 – tradutível def factorial(n): if n < 2: return 1 return n * factorial(n - 1)	Exemplo 1 – não reconhecível pelo <i>RecPy</i> def factorial(n): if n == 5: print(n) if n < 2: return 1 return n * factorial(n - 1)
Exemplo 2 – tradutível def factorial(n,acc=1): while not n < 2: n,acc = n - 1, n * acc return acc * 1	Exemplo 2 – não reconhecível pelo <i>RecPy</i> def factorial(n,acc=1): if n==5: print(n) while not n < 2: n,acc = n - 1, n * acc return acc * 1

Quadro 3 - Exemplo de códigos não reconhecíveis (mais de uma linha de if ou while)

- 6) As funções recursivas, para serem traduzíveis pelo *RecPy*, não devem conter mais, nem menos, linhas de RETURN do que aquelas especificados no Tópico 5.4 (Formatos de funções recursivas reconhecíveis no *RecPy*).
Ver exemplo:

Função reconhecida e traduzível pelo <i>RecPy</i>	Função hipotética não reconhecida pelo <i>RecPy</i> , por fugir ao padrão aceito. O exemplo produz como resultado a mesma função de entrada, sem tradução. Para a tradução de funções recursivas não caudais no <i>RecPy</i> , é necessário que haja duas linhas começando por <code>return</code> , como no exemplo válido da coluna ao lado.
<pre>def factorial(n): if n < 2: return 1 return n * factorial(n - 1)</pre>	<pre>def factorial(n): if n == 5: print(n) else: return n * factorial(n - 1)</pre>

Quadro 4 - Aviso quanto às linhas de `return`

- 7) Tanto quanto possível, deve-se evitar o aninhamento, ou a utilização desnecessária, de parêntesis, pois podem prejudicar o reconhecimento em alguns casos.

Ver exemplo abaixo:

Função reconhecida e traduzível pelo <i>RecPy</i>	Função não reconhecida pelo <i>RecPy</i> , por fugir ao padrão aceito (parêntesis em local que atrapalha o reconhecimento). Os parêntesis problemáticos estão marcados em vermelho, negrito e fonte maior:
<pre>def factorial(n, acc=1): if n < 2: return 1 * acc return factorial(n - 1, n * acc)</pre>	<pre>def factorial(n, acc=1): if (n < 2): return (1 * (acc)) return (factorial((n - 1), (n * acc)))</pre>

Quadro 5 - Aviso quanto à utilização de parêntesis excessivos e desnecessários

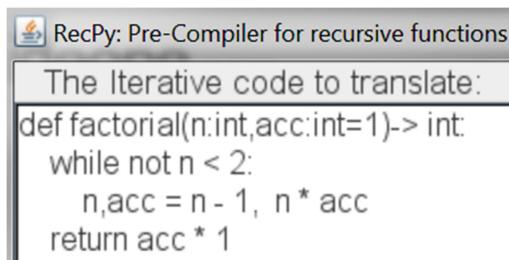
- 8) Na tradução de uma função recursiva não caudal para recursiva caudal (NTR-TR), o *RecPy* mantém a mesma posição do operador em relação à variável que é deslocada para dentro da lista de parâmetros da função. Ou seja, por exemplo, se essa variável estava à esquerda do operador na conversão,

continuará na mesma posição relativa ao operador. Assim, nos casos de manipulações de strings ou vetores em que a ordem fazia diferença, ou seja, nos casos em que a operação não atendia à propriedade comutativa, a tradução de uma função recursiva não caudal (NTR) para recursiva caudal estava invertendo a ordem de saída do resultado original. Isso ocorria nos exemplos dos algoritmos de inversão de string, de ordenação e de concatenação de texto. Para resolver esse problema, nos casos em que o operador é de concatenação (símbolo “+”), o RecPy passou a inverter a ordem final, o que anula o anterior problema da inversão nesse tipo de tradução. O operador “+” também funciona para adição. Mas, como a adição simples atende à propriedade comutativa, essa inversão não gera novos problemas.

- 9) A tradução de uma função recursiva não caudal para iterativa (NTR-IT) é realizada, internamente, por intermédio de uma tradução prévia e automática de recursiva não caudal para recursiva caudal (NTR-TR), de modo transparente ao usuário. Portanto, se algum problema ocorrer nessa primeira etapa será transmitido para a tradução final (TR-IT).

- 10) Sintaxes novas do Python, como, por exemplo, a de “*type hints*”, não foram testadas com profundidade, pois a proposta do *RecPy* é trabalhar com as versões Python até a 2.7.16. Portanto, as novidades sintáticas das novas versões Python (3.x em diante) podem eventualmente gerar problemas de reconhecimento no *RecPy*. Não obstante, no teste abaixo, único realizado nesse sentido, a função iterativa foi reconhecida e convertida para a versão recursiva caudal correspondente. Embora estes exemplos não rodem na versão 2.7.16, são executados corretamente na versão 3:

- a) Original:



```
RecPy: Pre-Compiler for recursive functions
The Iterative code to translate:
def factorial(n:int,acc:int=1)-> int:
    while not n < 2:
        n,acc = n - 1, n * acc
    return acc * 1
```

Figura 6 - Exemplo de função a ser traduzida

b) Traduzido:

```
The tail recursive correspondent code:
def factorial(n:int,acc:int=1)-> int:
    if n < 2:
        return acc * 1
    return factorial(n - 1, n * acc)
```

Figura 7 - Exemplo de função já traduzida

11) Nos experimentos realizados, a linha inicial das funções – def <nome_da_funcao> (lista de parâmetros) – situou-se normalmente no início da respectiva linha (coluna zero do arquivo de código). Além disso, não foram utilizadas dentro de classes específicas, ou seja, suas chamadas não se apresentam como: <classe>.<funcao>. A existência do caracter ponto (“.”) pode causar problemas no reconhecimento.

O *RecPy* utiliza a posição de referência da coluna para separar as funções. Ele só repassa ao lexer/parser um bloco de função por vez. Então, o aplicativo marca a posição (na linha) em que aparece a palavra reservada “def”. E vai adicionando as linhas de baixo até que encontre um comando situado na mesma posição, ou seja, no mesmo nível de identação do início do bloco da função.

12) Recomenda-se evitar comentários, ou linhas de código adicionais aos padrões aceitos, que contenham palavras reservadas da linguagem Python, em especial if, else, while, return, def, ainda que em parte. Embora essa possibilidade tenha sido tratada, em caso de erro, retire o comentário. Eventualmente, pode ser que uma palavra como “cifra”, por conter *if* em parte do nome, possa ser interpretada equivocadamente.

13) Ainda quanto aos comentários, em casos bem específicos, é possível que ocorram erros ao inserir comentários de diversas linhas.

14) Outro tipo de comentário não reconhecido é o que se situa na mesma linha do código. Exemplo:

`def potTR(n,b, acc): # não usar esta forma de comentário!!`

Se necessário, o comentário pode ser deslocado para cima da linha de código:

```
# Assim, o comentário é reconhecido...
def potTR(n,b, acc):
```

- 15) O RecPy utiliza o padrão de final de linhas do Windows (CR LF);
- 16) Quanto estiver traduzindo funções iterativas (IT-TR), evite também adicionar linhas com atribuições ou caracteres “=” que não sejam aquelas exclusivas do padrão aceito (ver Tópico 5.4 - Formatos de funções recursivas reconhecíveis no *RecPy*, Sub-tópico 3). Em palavras mais simples: apenas uma linha do bloco da função deve conter o caractere “=”.

5.5.2. Resumo dos requisitos e recomendações contidos no Tópico anterior

- 1) Utilize padrões de funções reconhecíveis pelo *RecPy*: Tópico 5.4.;
- 2) Na nomenclatura das funções a serem traduzidas, utilize o prefixo (ou sufixo) IT para a tradução de uma função originalmente iterativa, TR para recursiva caudal e NTR para recursiva não caudal;
- 3) Na tradução TR-IT, mantenha a mesma ordem das variáveis contidas em listas;
- 4) Observe os cuidados com os IFs, WHILEs e RETURNS (Sub-tópicos 4 a 6 do Tópico anterior);
- 5) Evite parêntesis desnecessários;
- 6) Utilize a sintaxe de Python 2.7.xx;
- 7) Mantenha a primeira linha (def ...) junto à margem esquerda;
- 8) Cuidado com comentários que contenham palavras reservadas do Python;
- 9) Em caso de erros, experimente retirar os comentários;
- 10) Quando experimentar uma função isoladamente, sempre insira uma linha em branco ao final;
- 11) Na tradução IT-TR, apenas uma linha do bloco da função deve conter o caractere “=”.

5.6. Desenvolvimento do aplicativo *RecPy*

Na escolha da ferramenta – ANTLR4 – utilizada para desenvolvimento do núcleo do pré-compilador *RecPy*, um fator determinante foram suas características: possibilitar desenvolvimento rápido, facilidade de aprendizado, simplicidade nos procedimentos de alterações de códigos do compilador. Mais que isso, a gramática elaborada neste protótipo pode ser ampliada para outros casos e linguagens de modo presumidamente mais rápido e simples do que ocorreria de outro modo. Na versão atual – ANTLR 4.8, lançada em janeiro de 2020 – a ferramenta pode gerar código alvo nas seguintes linguagens: Java, C#, Python 2 e 3, Javascript, Go, C++, Swift e PHP, segundo o desenvolvedor (Parr, 2013)^[B17]. O *RecPy* foi desenvolvido, em Java, para reconhecimento de funções em Python 2.7.16, mas pode ser facilmente implementado para reconhecer outras das linguagens citadas.

As alterações nos arquivos de códigos do *lexer* e do *parser* que o Antlr4 utiliza foram realizadas separadamente, na janela de comando do Windows (CMD). Os arquivos de códigos de teste contêm exclusivamente os blocos de funções recursivas que estão sendo testadas em cada novo ciclo de introdução de um novo modelo de função recursiva reconhecível.

Na pasta do Github (endereço: <https://github.com/robertocsa/TCC_CComp_UERJ> do Github), encontra-se arquivo de execução em lote (arquivo de execução em lote intitulado: *compilaLexParser.bat*, situado na pasta *RecPy/src/*, que contém os comandos necessários à compilação dos arquivos do Antlr4: *RecPyLexer.g4* e *RecPyParser.g4*). Também na pasta *RecPy/src/* encontra-se o arquivo “Leiame-procedimentos de compilacao-Lexer Parser.txt”, que contém instruções para a compilação dos arquivos do Antlr4: *RecPyLexer.g4* e *RecPyParser.g4*, na linha de comandos.

O módulo Grun, do Antlr4, permite, dentre outros, dois modos de visualização que foram utilizados na construção do *RecPy*:

- 1) Exemplo de apresentação dos tokens gerados por um determinado código de entrada:

```

C:\Windows\system32\cmd.exe
Microsoft Windows [versão 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.
O sistema não pode encontrar o caminho especificado.

E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>compilaLexParser.bat fatIT.pyw <-- Comando
"Uso: compilaLexParser.bat <nome do arquivo>
"Antes de rodar, comente a linha do Package nos arquivos *.g4"
"Exemplo: "
"compilaLexParser.bat fatmodTR.pyw"
"Coloque um segundo parametro com qualquer conteudo se quiser chamar a representacao grafica da arvore AST"
"Exemplo: "
"compilaLexParser.bat fatmodTR.pyw G"
"""

E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>call antlr RecPyLexer.g4
E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>java org.antlr.v4.Tool RecPyLexer.g4
E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>call antlr RecPyParser.g4 -visitor
E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>java org.antlr.v4.Tool RecPyParser.g4 -visitor
E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>call javac *.java
E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>if [] == [] goto tokens
E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>call grun RecPy start -tokens fatIT.pyw
E:\Estudo\UERJ\Materias\TCC\GitHb\RecPy\src\lexrParsr>java org.antlr.v4.gui.TestRig RecPy start -tokens fatIT.pyw
[@0,0:23='def factorial(n,acc=1):\n',<DEF_EXPR>,1:0] <-- Tokens gerados
[@1,24:44='    while not n < 2:\n',<WHILE_EXPR>,2:0]
[@2,45:96='        n,acc = n - 1, n * acc\n        return acc * 1\n',<END_WHILE_BLOCK>,3:0] <-- neste exemplo:
[@3,97:97='\\n',<EMPTY_LINE>,5:0]
[@4,98:98='\\n',<EMPTY_LINE>,6:0]
[@5,99:99='\\n',<EMPTY_LINE>,7:0]
[@6,100:99='<EOF>',<EOF>,8:0] <-->

```

Figura 8 - Exemplo de tokens gerados em janela de comando (CMD)

No Tópico 6.5, apresentam-se os tokens gerados para cada um dos experimentos realizados.

- 2) Exemplo de apresentação da árvore AST (árvore sintática abstrata). Para o mesmo exemplo, foi utilizada a linha de comando
- \GitHb\RecPy\src\lexrParsr>compilaLexParser.bat fatIT.pyw -g :

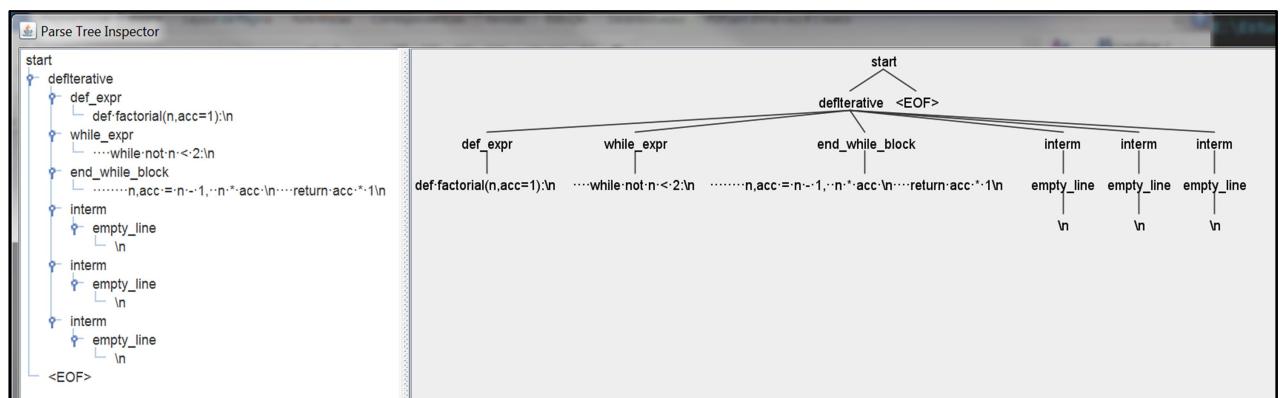


Figura 9 - Exemplo de árvore sintática abstrata (AST) gerada em linha de comando

No Tópico 6.6, apresentam-se as árvores abstratas sintáticas geradas para cada um dos experimentos realizados.

Para simplificar os procedimentos de compilação, somente são passados ao *parser* e ao *lexer* os blocos de código de funções de interesse. Portanto, para fazer os testes em linha de comando, os arquivos de exemplos devem conter apenas esses pequenos trechos de código. E, no final, devem conter uma linha em branco. Exemplo de bloco de função para teste do *parser* e do *lexer* do Antlr4:

```
def factorial(n,acc=1):
    while not n < 2:
        n,acc = n - 1,  n * acc
    return acc * 1
```

Figura 10 - Exemplo de bloco de função para teste do parser e do lexer do Antlr4

No código que roda no Eclipse (em Java), os próprios comandos de visitação cuidam de só passar ao *parser* e ao *lexer* somente esses pequenos trechos. O que não for trecho a ser traduzido é simplesmente copiado sem alterações para o campo de resultados da tradução.

Em resumo: quando se deseja fazer uma alteração do código no Eclipse e testá-la, deve utilizar o código completo do arquivo em experimento. Do contrário, para testes em janela da linha de comando (CMD do Windows), deve-se usar como código de entrada apenas o bloco da função específica em teste.

5.6.1. Descrição da construção do pré-compilador *RecPy*

O pré-compilador *RecPy* foi construído a partir dos arquivos Java criados automaticamente pelo Antlr4. O procedimento básico de construção origina-se da criação dos códigos de reconhecimento do *parser* e do *lexer* do Antlr4 (arquivos *.g4 contidos na pasta `\RecPy\src\lexrParsr`, no Github). Os códigos do pré-compilador são testados e modificados conforme vão sendo agregadas novas funções recursivas a serem reconhecidas.

Os testes de funcionamento do núcleo do pré-compilador (arquivos *.g4 do *parser* e do *lexer*) são realizados separadamente, em janela de comando CMD. Os arquivos para teste em linha de comando, contidos na pasta do Github <`\Python\TestesEmLinhaCMD`> não

contêm outros códigos senão os exclusivamente relacionados aos blocos de funções recursivas em teste.

Para entendimento dessas etapas de preparação do *parser* e do *lexer* do Antlr4 foram necessárias leitura e revisitações ao compêndio *The Definitive ANTLR Reference*, do autor do Antlr4 [B17]

Realizada essa etapa acima descrita, passou-se à construção da interface de aplicação e do “*visitor*” da árvore de análise sintática gerada conforme modelos criados. A interface de aplicação carrega um arquivo de código Python completo e, para simplificar as etapas de reconhecimento, filtra somente as funções recursivas de interesse, ou seja, aquelas que correspondam ao tipo selecionado. Por exemplo, se o usuário selecionar traduzir uma função Recursiva Caudal (TR) para função Iterativa (IT), a interface cuidará de separar para análise somente o(s) bloco(s) de função que sejam do tipo Recursivo Caudal. Todo o resto passará direto para a janela de resultados, ou seja, será simplesmente copiado conforme o código original, sem modificações.

Daí, o *parser* e o *lexer* receberão somente o bloco de código da função recursiva em análise. Se for uma função abrangida nos padrões reconhecíveis, será gerada a árvore sintática abstrata (AST) que será “visitada” pelo *visitor* do *RecPy*. Os arquivos de código Java dos *visitors* de cada tipo estão localizados na pasta `\RecPy\src\visitor`, do Github (<https://github.com/robertocsa/TCC_CComp_UERJ>). Toda alteração nos arquivos *.g4 induz a um novo ciclo de testes, pois normalmente requer também alterações nas partes afetadas dos “*visitors*”.

Caso uma determinada função não esteja abrangida nos padrões reconhecíveis pelo *RecPy*, o esperado é que seja simplesmente copiada para o campo de resultados conforme o original, sem alterações (à exceção do nome da função). Em alguns casos de erros, é possível que apareça um espaço em branco onde deveria se situar a função traduzida. Em ambas as hipóteses, é preciso verificar o motivo que fez com que a função não fosse reconhecida. Muitas vezes, bastam pequenas alterações para fazer com que a função seja reconhecida. Ver, quanto a isso, a lista de especificações e restrições de uso do aplicativo, no Tópico 5.5 (Requisitos, restrições, observações e recomendações no reconhecimento de funções).

6. EXPERIMENTOS REALIZADOS

6.1. Introdução

Algoritmos clássicos e bem conhecidos nas comunidades acadêmicas e científicas foram utilizados nos testes. Em cada algoritmo testado, partiu-se de uma versão básica, recursiva não caudal (NTR), ou recursiva caudal (TR) quando não se conseguiu obter uma versão NTR, e, dela, foram criadas as versões correspondentes de outros tipos (Iterativa e recursiva caudal). Os experimentos foram realizados em Python, versão 2.7.16 - x64 e em Python versão 3.7.9 - x64, mas a sintaxe reconhecida segue o padrão da versão 2.7.16.

Foram testados, ao todo, 11 algoritmos distintos, em vinte e nove experimentos deles derivados. Esses resultados podem ser vistos no Tópico 6.7.

6.2. Apresentação da metodologia

Inicialmente, foram selecionados modelos de algoritmos bastante simples para a realização da conversão automatizada. Pensava-se em realizar apenas um tipo de conversão: de funções recursivas caudais para funções iterativas. Com o sucesso nessa primeira abordagem, verificou-se que seria viável também realizar os outros tipos conversão que o aplicativo proporciona: de recursiva não caudal para recursiva caudal e para iterativa; e desta para recursiva caudal.

Subsequentemente, outros algoritmos bastante conhecidos pela comunidade técnica e acadêmica foram sendo testados, dentre eles, os clássicos algoritmos de Fatorial, Fibonacci, ordenação e palíndromo.

E à medida em que o reconhecimento de alguma dessas rotinas falhava, eram feitos mais ajustes no pré-compilador para o torná-lo crescentemente genérico. Naturalmente, em cada uma dessas ocorrências, buscou-se evitar, tanto quanto possível, o aumento de dificuldade de leitura dos códigos do *parser* e do *lexer* do *RecPy*. Uma preocupação constante, e uma fonte de problemas, em cada uma dessas alterações foi a manutenção do reconhecimento dos algoritmos já testados. Ou seja: em cada novo padrão de função introduzido no rol de modelos reconhecíveis, as alterações no pré-compilador não deveriam causar, e não causaram, impactos negativos no reconhecimento de rotinas que já estavam

sendo reconhecidas antes da mudança. Na prática, a cada aumento do rol de funções reconhecíveis, uma nova rodada de testes completa necessitou ser realizada, em ciclos.

A ferramenta utilizada na confecção do *RecPy* – Antlr4 – colaborou bastante para a eficácia desse processo, em razão de seus atributos: relativa facilidade de aprendizado e de manuseio, simplicidade de conceitos. Inclusive, pode-se dizer que um “efeito colateral” positivo deste TCC é sua contribuição também no estudo da elaboração de compiladores. O leitor que tenha como foco este estudo deve encontrar subsídios neste trabalho.

No Tópico 5.6, estão descritos, de modo detalhado, os procedimentos necessários à compilação do código Java referente ao *parser* e ao *lexer* do *RecPy*.

6.3. Procedimentos e experimentos realizados

6.3.1. Configuração utilizada nos experimentos

6.3.1.1. Configuração da estação de trabalho utilizada

O computador utilizado teve a seguinte configuração:

- I. Processador Intel® Core™2 Quad CPU, Q9400, @ 2.66GHz;
- II. Memória RAM: 8,00 Gb;
- III. Sistema Operacional: Windows 7, Home Premium, Service Pack 1, 64 bits;

6.3.1.2. Python, ide Idle:

Os experimentos foram realizados em duas versões: 2.7.16 e 3.7.9, ambas x64.

A IDE Idle, na versão 3.7.9, realiza um “*auto-squeeze*” nos resultados com muitas linhas de caracteres. Essa compactação reduz o tempo de apresentação, pois não se escrevem na tela, de imediato, as linhas de resultado. Então, para que essa característica não interfira na curva de tempo dos resultados, é preciso reconfigurar o limite de “*auto-squeeze min. Lines*” (*settings>General*), que, por padrão, vem configurado para 50 linhas. Configurou-se esse limite para 10.000 linhas. Na versão 2.7.16, não existe essa configuração.

Para se ampliar o limite padrão de recursões, utilizou-se a biblioteca sys. Para isso, houve a inclusão das seguintes linhas de código, sem as quais os algoritmos recursivos não passam, em geral, de um valor aproximado de $n = 992$ (número de chamadas recursivas):

```
import sys
sys.setrecursionlimit(1000000)
```

6.4. Códigos dos exemplos de algoritmos conversíveis no pré-compilador *RecPy*

Na pasta \Python\Experimentos\ do Github (<https://github.com/robertocsa/TCC_CComp_UERJ>), encontram-se os códigos completos testados.

No quadro abaixo, constam, pela ordem, os códigos dos blocos das funções testadas:

- a) Fatorial simples com operação na última linha à esquerda;
- b) Fatorial simples com operação na última linha à direita;
- c) Fatorial modificado: retorna $n! \bmod (k+1)$;
- d) Inverte string;
- e) Ordena vetor;
- f) Potenciação;
- g) Soma de vetor;
- h) Concatenação de texto;
- i) Fibonacci;
- j) MDC;
- k) Palíndromo; e
- l) Verificação de números primos.

	Código original NTR	Código traduzido para TR	Código traduzido para IT
a)	def fatNTR(n): if n < 2: return 1 return n * fatNTR(n - 1)	def fatTR(n, acc=1): if n < 2: return acc return fatTR(n - 1, n * acc)	def fatIT(n, acc=1): while not n < 2: n,acc = n - 1, n * acc return acc
b)	def fatNTR(n): if n < 2: return 1 return fatNTR(n - 1)*n	def fatTR(n, acc=1): if n < 2: return acc return fatTR(n - 1, acc*n)	def fatIT(n, acc=1): while not n < 2: n,acc = n - 1, acc * n return acc
c)	def fatmodNTR(n, k): #retorna n! mod (k+1) if n == 0: return 1 else: return (n*fatmodNTR(n-1,k))%(k+1)	def fatmodTR(n, k, acc=1): #retorna n! mod (k+1) if n == 0: return acc return fatmodTR(n-1,k,n*acc%(k+1))	def fatmodIT(n, k, acc=1): #retorna n! mod (k+1) while not n == 0: n,k,acc = n-1,k, n * acc % (k+1) return acc
d)	def invNTR(texto): if (len(texto)<=0): return "" else: n=len(texto)-1 return texto[n]+invNTR(texto[0:n])	def invTR(texto, acc=""): if (len(texto)<=0): return acc n=len(texto)-1 return invTR(texto[0:n], acc + texto[n])	def invIT(texto, acc=""): while not (len(texto)<=0): n=len(texto)-1 texto,acc = texto[0:n], acc+texto[n] return acc
e)	def ordNTR(L): if (L==[]): return [] return menor(L) + ordNTR(subtracao(L, menor(L)))	def ordTR(L, acc=[]): if (L==[]): return acc return ordTR(subtracao(L, menor(L)), acc + menor(L))	def ordIT(L, acc=[]): while not (L==[]): L,acc = subtracao(L, menor(L)), acc + menor(L) return acc
f)	def potNTR(n,b): if (b==1): return 1 return n * potNTR(n, b-1)	def potTR(n,b, acc=1): if (b==1): return acc return potTR(n, b-1, n * acc)	def potIT(n,b, acc=1): while not (b==1): n,b,acc = n, b-1, n * acc return acc
g)	def sumVecNTR(A, n): if n==1: return 0 else: return A[n]+sumVecNTR(A, n-1)	def sumVecTR(A, n, acc=0): if n==1: return acc return sumVecTR(A, n-1, acc + A[n])	def sumVecIT(A, n, acc=0): while not n==1: A,n,acc = A, n-1, acc + A[n] return acc
h)	def textNTR(n): if (n==0): return "" return textNTR(n - 1) + str(n) + ". The quick brown fox jumps over the lazy dog.\n"	def textTR(n, acc=""): if (n==0): return acc return textTR(n - 1, str(n) + ". The quick brown fox jumps over the lazy dog.\n"+ acc)	def textIT(n, acc=""): while not (n==0): n,acc = n - 1, str(n) + ". The quick brown fox jumps over the lazy dog.\n"+ acc return acc

Quadro 6 - Códigos dos blocos de funções testadas

6.4.1.1. Algoritmos em que não houve experimentos na versão NTR:

	Código original TR	Código traduzido para IT
i)	def fibTR(a, b, c): if (a == 0): return c return fibTR(a-1, b+c, b)	def fibIT(a, b, c): while not (a == 0): a,b,c = a-1, b+c, b return c
j)	def mdcTR(a, b): if (b == 0): return a return mdcTR(b, (a % b))	def mdcIT(a, b): while not (b == 0): a,b = b, (a % b) return a
k)	def palTR(texto): if (casoBase(texto)!= None): return casoBase(texto) return palTR(texto[1:len(texto)-1])	def palIT(texto): while not (casoBase(texto)!= None): texto = texto[1:len(texto)-1] return casoBase(texto)
l)	def primTR(n,i=2): # verifica o caso base: if (casoBase(n,i)!=None): return casoBase(n,i) return primTR(n, i+1)	def primIT(n,i=2): # verifica o caso base: while not (casoBase(n,i)!=None): n,i = n, i+1 return casoBase(n,i)

Quadro 7 - Códigos de funções testadas (algoritmos em que não houve experimentos na versão NTR)

6.5. Tokens gerados para os algoritmos testados

```
[@0,0:21='def fatmodNTR(n, k):\r\n',<DEF_BLK>,1:0]
[@1,22:48='    #retorna n! mod (k+1)\r\n',<INTERM>,2:0]
[@2,49:64='        if n == 0:\r\n',<IF_EXPR>,3:0]
[@3,65:82='            return 1\r\n',<RETURN_EXPR>,4:0]
[@4,83:93='        else:\r\n',<ELSE>,5:0]
[@5,94:136='            return (n*fatmodNTR(n-1,k))%(k+1)\r\n',<RETURN_NTR>,6:0]
[@6,137:136='<EOF>',<EOF>,7:0]
```

Figura 11 - Tokens gerados para a função fatmodNTR

```
[@0,0:27='def fatmodTR(n, k, acc=1):\r\n',<DEF_BLK>,1:0]
[@1,28:53='    #retorna n! mod (k+1)\r\n',<INTERM>,2:0]
[@2,54:68='        if n == 0:\r\n',<IF_EXPR>,3:0]
[@3,69:87='            return acc\r\n',<RETURN_EXPR>,4:0]
[@4,88:127='            return fatmodTR(n-1,k,n*acc%(k+1))\r\n',<RETURN_TR>,5:0]
[@5,128:129='\r\n',<INTERM>,6:0]
[@6,130:129='<EOF>',<EOF>,7:0]
```

Figura 12 - Tokens gerados para a função fatmodTR

```
[@0,0:27='def fatmodIT(n, k, acc=1):\r\n',<DEF_BLK>,1:0]
[@1,28:54='    #retorna n! mod (k+1)\r\n',<INTERM>,2:0]
[@2,55:78='        while not n == 0:\r\n',<START WHILE_BLK>,3:0]
[@3,79:137='            n,k,acc = n-1,k, n * acc %(k+1)\r\n',<END WHILE_BLK>,4:0]
[@4,138:137='<EOF>',<EOF>,6:0]
```

Figura 13 - Tokens gerados para a função fatmodIT

```
[@0,0:15='def fatNTR(n):\r\n',<DEF_BLK>,1:0]
[@1,16:31='    if n < 2:\r\n',<IF_EXPR>,2:0]
[@2,32:50='        return 1\r\n',<RETURN_EXPR>,3:0]
[@3,51:81='        return n * fatNTR(n - 1)\r\n',<RETURN_NTR>,4:0]
[@4,82:81='<EOF> ',<EOF>,5:0]
```

Figura 14 - Tokens gerados para a função fatNTR

```
[@0,0:25='def fatTR(n, acc=1):\r\n',<DEF_BLK>,1:0]
[@1,26:49='    if n < 2:\r\n',<IF_EXPR>,2:0]
[@2,50:70='        return acc\r\n',<RETURN_EXPR>,3:0]
[@3,71:106='        return fatTR(n - 1, n * acc)\r\n',<RETURN_TR>,4:0]
[@4,107:108=''\r\n',<INTERM>,5:0]
[@5,109:108='<EOF> ',<EOF>,6:0]
```

Figura 15 - Tokens gerados para a função fatTR

```
[@0,0:21='def fatIT(n, acc=1):\r\n',<DEF_BLK>,1:0]
[@1,22:45='    while not n < 2:\r\n',<START WHILE_BLK>,2:0]
[@2,46:95='        n,acc = n - 1, n * acc\r\n',<RETURN_EXPR>,3:0]
[@3,96:95='<EOF> ',<EOF>,5:0]
```

Figura 16 - Tokens gerados para a função fatIT

```
[@0,0:23='def fibTR(a, b, c):\r\n',<DEF_BLK>,1:0]
[@1,24:41='    if (a == 0):\r\n',<IF_EXPR>,2:0]
[@2,42:59='        return c\r\n',<RETURN_EXPR>,3:0]
[@3,60:90='        return fibTR(a-1, b+c, b)\r\n',<RETURN_TR>,4:0]
[@4,91:92=''\r\n',<INTERM>,5:0]
[@5,93:92='<EOF> ',<EOF>,6:0]
```

Figura 17 - Tokens gerados para a função fibTR

```
[@0,0:20='def fibIT(a, b, c):\r\n',<DEF_BLK>,1:0]
[@1,21:46='    while not (a == 0):\r\n',<START WHILE_BLK>,2:0]
[@2,47:89='        a,b,c = a-1, b+c, b\r\n',<RETURN_EXPR>,3:0]
[@3,90:91=''\r\n',<INTERM>,5:0]
[@4,92:91='<EOF> ',<EOF>,6:0]
```

Figura 18 - Tokens gerados para a função fibIT

```
[@0,0:19='def invNTR(texto):\r\n',<DEF_BLK>,1:0]
[@1,20:44='    if (len(texto)<=0):\r\n',<IF_EXPR>,2:0]
[@2,45:63='        return ""\r\n',<RETURN_EXPR>,3:0]
[@3,64:74='    else:\r\n',<ELSE>,4:0]
[@4,75:98='        n=len(texto)-1\r\n',<INTERM>,5:0]
[@5,99:142='        return texto[n]+invNTR(texto[0:n])\r\n',<RETURN_NTR>,6:0]
[@6,143:144=''\r\n',<INTERM>,7:0]
[@7,145:144='<EOF> ',<EOF>,8:0]
```

Figura 19 - Tokens gerados para a função invNTR

```
[@0,0:26='def invTR(texto, acc=""):\r\n',<DEF_BLK>,1:0]
[@1,27:46='    n=len(texto)-1\r\n',<INTERM>,2:0]
[@2,47:71='    if (len(texto)<=0):\r\n',<IF_EXPR>,3:0]
[@3,72:95='        return acc\r\n',<RETURN_EXPR>,4:0]
[@4,96:139='        return invTR(texto[0:n], acc+texto[n])\r\n',<RETURN_TR>,5:0]
[@5,140:141=''\r\n',<INTERM>,6:0]
[@6,142:143=''\r\n',<INTERM>,7:0]
[@7,144:143='<EOF> ',<EOF>,8:0]
```

Figura 20 - Tokens gerados para a função invTR

```
[@0,0:26='def invIT(texto, acc=""):\r\n',<DEF_BLK>,1:0]
[@1,27:59='    while not (len(texto)<=0):\r\n',<START WHILE_BLK>,2:0]
[@2,60:83='        n=len(texto)-1\r\n',<INTERM>,3:0]
[@3,84:145='        texto,acc = texto[0:n], acc+texto[n]\r\n',<RETURN_EXPR>,4:0]
[@4,146:145='<EOF> ',<EOF>,6:0]
```

Figura 21 - Tokens gerados para a função invIT

```
[@0,0:17='def mdcTR(a, b):\r\n',<DEF_BLK>,1:0]
[@1,18:35='    if (b == 0):\r\n',<IF_EXPR>,2:0]
[@2,36:53='        return a\r\n',<RETURN_EXPR>,3:0]
[@3,54:83='        return mdcTR(b, (a % b))\r\n',<RETURN_TR>,4:0]
[@4,84:83='<EOF> ',<EOF>,5:0]
```

Figura 22 - Tokens gerados para a função mdcTR

```
[@0,0:17='def mdcIT(a, b):\r\n',<DEF_BLK>,1:0]
[@1,18:43='    while not (b == 0):\r\n',<START WHILE_BLK>,2:0]
[@2,44:83='        a,b = b, (a % b)\r\n        return a\r\n',<END WHILE_BLK>,3:0]
[@3,84:83='<EOF> ',<EOF>,5:0]
```

Figura 23 - Tokens gerados para a função mdcIT

```
[@0,0:15='def ordNTR(L):\r\n',<DEF_BLK>,1:0]
[@1,16:32='    if (L==[]):\r\n',<IF_EXPR>,2:0]
[@2,33:51='        return []\r\n',<RETURN_EXPR>,3:0]
[@3,52:106='        return menor(L) + ordNTR(subtracao (L, menor(L)))\r\n',<RETURN_NTR>,4:0]
[@4,107:108='\r\n',<INTERM>,5:0]
[@5,109:108='<EOF> ',<EOF>,6:0]
```

Figura 24 - Tokens gerados para a função ordNTR

```
[@0,0:22='def ordTR(L, acc=[]):\r\n',<DEF_BLK>,1:0]
[@1,23:39='    if (L==[]):\r\n',<IF_EXPR>,2:0]
[@2,40:59='        return acc\r\n',<RETURN_EXPR>,3:0]
[@3,60:119='        return ordTR(subtracao (L, menor(L)), menor(L) + acc)\r\n',<RETURN_TR>,4:0]
[@4,120:119='<EOF> ',<EOF>,5:0]
```

Figura 25 - Tokens gerados para a função ordTR

```
[@0,0:22='def ordIT(L, acc=[]):\r\n',<DEF_BLK>,1:0]
[@1,23:47='    while not (L==[]):\r\n',<START WHILE_BLK>,2:0]
[@2,48:120='        L,acc = subtracao (L, menor(L)), menor(L) + acc\r\n        return acc\r\n',<END WHILE_BLK>,3:0]
[@3,121:120='<EOF> ',<EOF>,5:0]
```

Figura 26 - Tokens gerados para a função ordIT

```
[@0,0:18='def palTR(texto):\r\n',<DEF_BLK>,1:0]
[@1,19:52='    if (casoBase(texto) != None):\r\n',<IF_EXPR>,2:0]
[@2,53:84='        return casoBase(texto)\r\n',<RETURN_EXPR>,3:0]
[@3,85:125='        return palTR(texto[1:len(texto)-1])\r\n',<RETURN_TR>,4:0]
[@4,126:125='<EOF> ',<EOF>,5:0]
```

Figura 27 - Tokens gerados para a função palTR

```
[@0,0:18='def palIT(texto):\r\n',<DEF_BLK>,1:0]
[@1,19:60='    while not (casoBase(texto) != None):\r\n',<START WHILE_BLK>,2:0]
[@2,61:127='        texto = texto[1:len(texto)-1]\r\n        return casoBase(texto)\r\n',<END WHILE_BLK>,3:0]
[@3,128:127='<EOF> ',<EOF>,5:0]
```

Figura 28 - Tokens gerados para a função palIT

```
[@0,0:21='def potNTR(n,b):\r\n',<DEF_BLK>,1:0]
[@1,22:38='    if (b==1):\r\n',<IF_EXPR>,2:0]
[@2,39:56='        return n\r\n',<RETURN_EXPR>,3:0]
[@3,57:87='        return n * potNTR(n, b-1)\r\n',<RETURN_NTR>,4:0]
[@4,88:89='\r\n',<INTERM>,5:0]
[@5,90:89='<EOF> ',<EOF>,6:0]
```

Figura 29 - Tokens gerados para a função potNTR

```
[@0,0:25='def potTR(n,b, acc):\r\n',<DEF_BLK>,1:0]
[@1,26:42='    if (b==1):\r\n',<IF_EXPR>,2:0]
[@2,43:62='        return acc\r\n',<RETURN_EXPR>,3:0]
[@3,63:98='        return potTR(n, b-1, n * acc)\r\n',<RETURN_TR>,4:0]
[@4,99:98='<EOF> ',<EOF>,5:0]
```

Figura 30 - Tokens gerados para a função potTR

```
[@0,0:24='def potIT(n,b, acc): \r\n',<DEF_BLK>,1:0]
[@1,25:49='    while not (b==1): \r\n',<START WHILE_BLK>,2:0]
[@2,50:100='        n,b,acc = n, b-1, n * acc\r\n    return acc\r\n',<END WHILE_BLK>,3:0]
[@3,101:100='<EOF>',<EOF>,5:0]
```

Figura 31 - Tokens gerados para a função potIT

```
[@0,0:23='def primTR(n, i=2): \r\n',<DEF_BLK>,1:0]
[@1,24:56='    # verifica o caso base: \r\n',<INTERM>,2:0]
[@2,57:87='        if (casoBase(n,i)!=None):\r\n',<IF_EXPR>,3:0]
[@3,88:117='            return casoBase(n,i)\r\n',<RETURN_EXPR>,4:0]
[@4,118:144='        return primTR(n, i+1)\r\n',<RETURN_TR>,5:0]
[@5,145:144='<EOF>',<EOF>,6:0]
```

Figura 32 - Tokens gerados para a função primNTR

```
[@0,0:21='def primIT(n, i=2): \r\n',<DEF_BLK>,1:0]
[@1,22:54='    # verifica o caso base: \r\n',<INTERM>,2:0]
[@2,55:93='        while not (casoBase(n,i)!=None):\r\n',<START WHILE_BLK>,3:0]
[@3,94:141='            n,i = n, i+1\r\n        return casoBase(n,i)\r\n',<END WHILE_BLK>,4:0]
[@4,142:141='<EOF>',<EOF>,6:0]
```

Figura 33 - Tokens gerados para a função primIT

```
[@0,0:22='def sumVecNTR(A, n):\r\n',<DEF_BLK>,1:0]
[@1,23:36='    if n==1:\r\n',<IF_EXPR>,2:0]
[@2,37:54='        return 0\r\n',<RETURN_EXPR>,3:0]
[@3,55:65='    else:\r\n',<ELSE>,4:0]
[@4,66:104='        return A[n]+sumVecNTR(A, n-1)\r\n',<RETURN_NTR>,5:0]
[@5,105:104='<EOF>',<EOF>,6:0]
```

Figura 34 - Tokens gerados para a função sumVecNTR

```
[@0,0:27='def sumVecTR(A, n, acc=0):\r\n',<DEF_BLK>,1:0]
[@1,28:41='    if n==1:\r\n',<IF_EXPR>,2:0]
[@2,42:61='        return acc\r\n',<RETURN_EXPR>,3:0]
[@3,62:101='        return sumVecTR(A, n-1, A[n]+ acc)\r\n',<RETURN_TR>,4:0]
[@4,102:101='<EOF>',<EOF>,5:0]
```

Figura 35 - Tokens gerados para a função sumVecTR

```
[@0,0:27='def sumVecIT(A, n, acc=0):\r\n',<DEF_BLK>,1:0]
[@1,28:49='    while not n==1:\r\n',<START WHILE_BLK>,2:0]
[@2,50:102='        A,n,acc = A, n-1, A[n]+ acc\r\n    return acc\r\n',<END WHILE_BLK>,3:0]
[@3,103:102='<EOF>',<EOF>,5:0]
```

Figura 36 - Tokens gerados para a função sumVecIT

```
[@0,0:16='def textNTR(n):\r\n',<DEF_BLK>,1:0]
[@1,17:32='    if (n==0):\r\n',<IF_EXPR>,2:0]
[@2,33:51='        return ""\r\n',<RETURN_EXPR>,3:0]
[@3,52:140='        return textNTR(n - 1) + str(n) + ". The quick brown fox jumps over the lazy dog.\r\n",<RETURN_NTR>,4:0]
[@4,141:140='<EOF>',<EOF>,5:0]
```

Figura 37 - Tokens gerados para a função textNTR

```
[@0,0:23='def textTR(n, acc=""):\r\n',<DEF_BLK>,1:0]
[@1,24:39='    if (n==0):\r\n',<IF_EXPR>,2:0]
[@2,40:59='        return acc\r\n',<RETURN_EXPR>,3:0]
[@3,60:152='        return textTR(n - 1, str(n) + ". The quick brown fox jumps over the lazy dog.\r\n" + acc)\r\n',<RETURN_TR>,4:0]
[@4,153:152='<EOF>',<EOF>,5:0]
```

Figura 38 - Tokens gerados para a função textTR

```
[@0,0:23='def textIT(n, acc=""):\r\n',<DEF_BLK>,1:0]
[@1,24:47='    while not (n==0):\r\n',<START WHILE_BLK>,2:0]
[@2,48:152='        n,acc = n - 1,str(n) + ". The quick brown fox jumps over the lazy dog.\r\n" + acc\r\n    return acc\r\n',<END WHILE_BLK>,3:0]
[@3,153:152='<EOF>',<EOF>,5:0]
```

Figura 39 - Tokens gerados para a função textIT

6.6. Árvores sintáticas abstratas (ASTs) geradas para os algoritmos testados

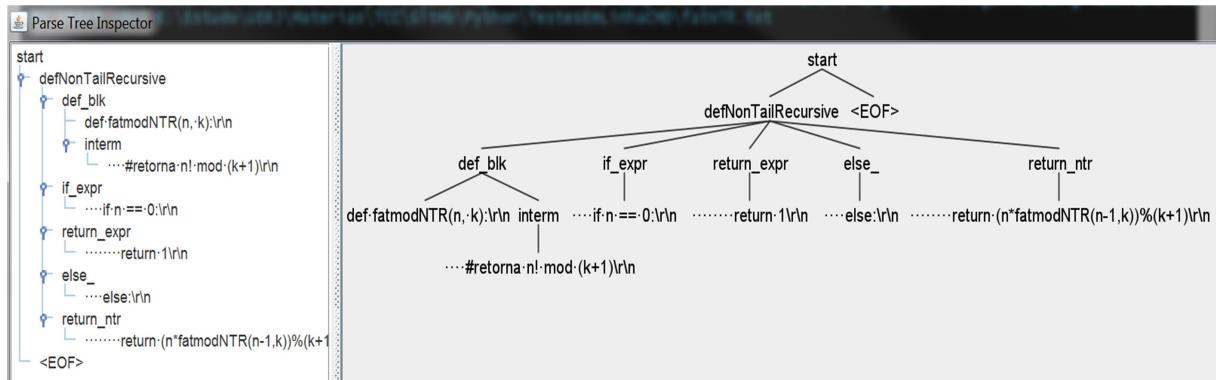


Figura 40 - Árvore sintática abstrata gerada para o algoritmo fatmodNTR

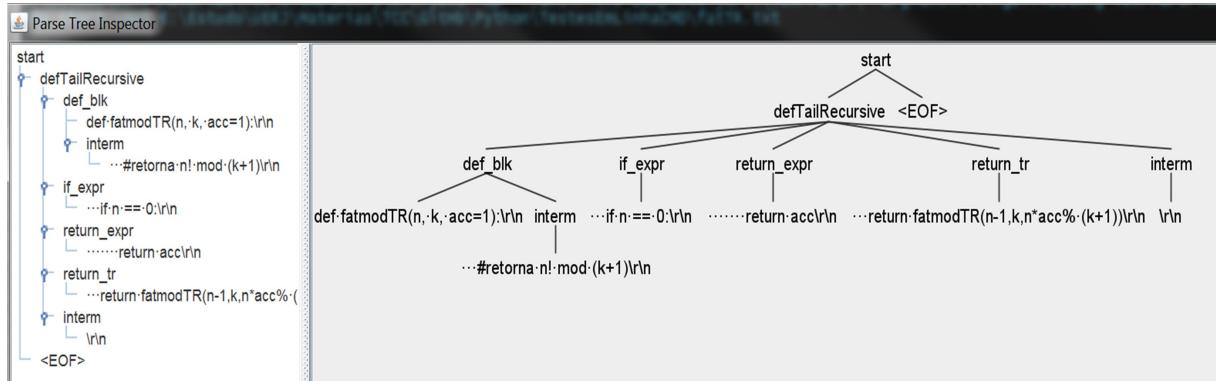


Figura 41 - Árvore sintática abstrata gerada para o algoritmo fatmodTR

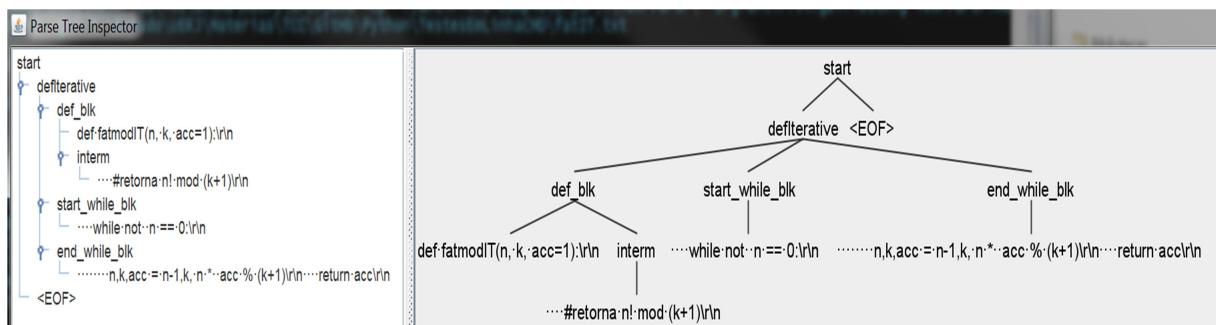


Figura 42 - Árvore sintética abstrata gerada para o algoritmo fatmodIT

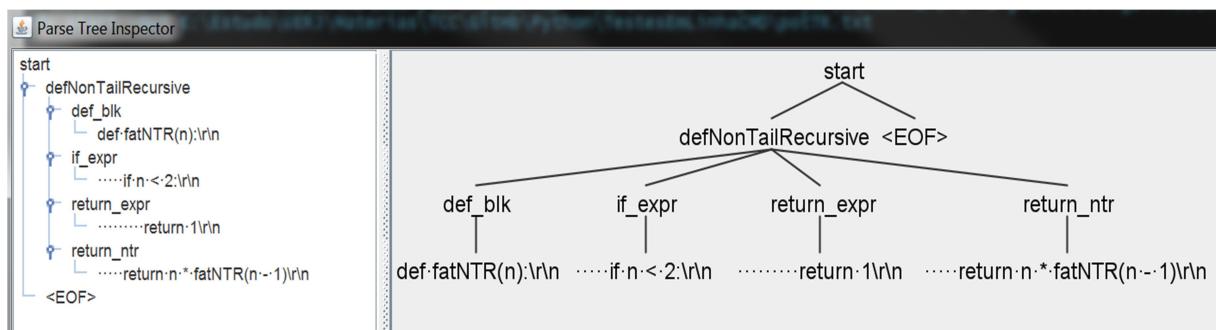


Figura 43 - Árvore sintática abstrata gerada para o algoritmo fatNTR

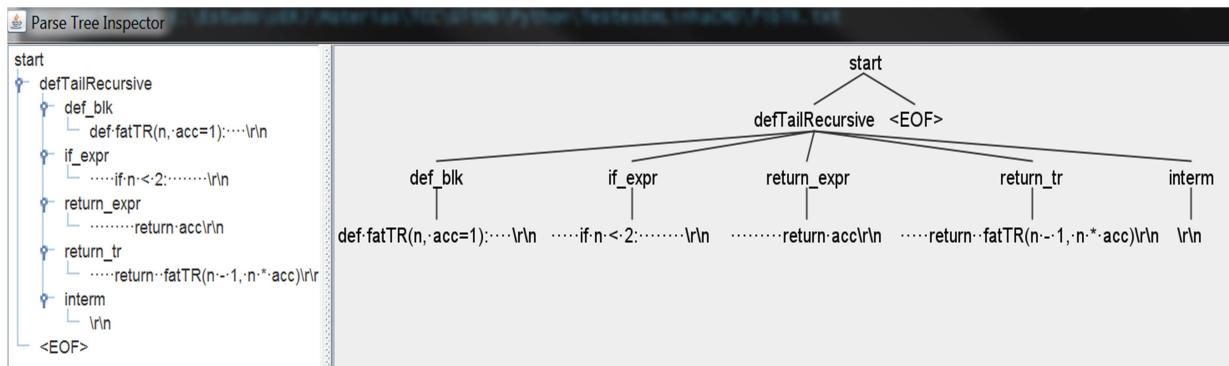


Figura 44 - Árvore sintática abstrata gerada para o algoritmo fatTR

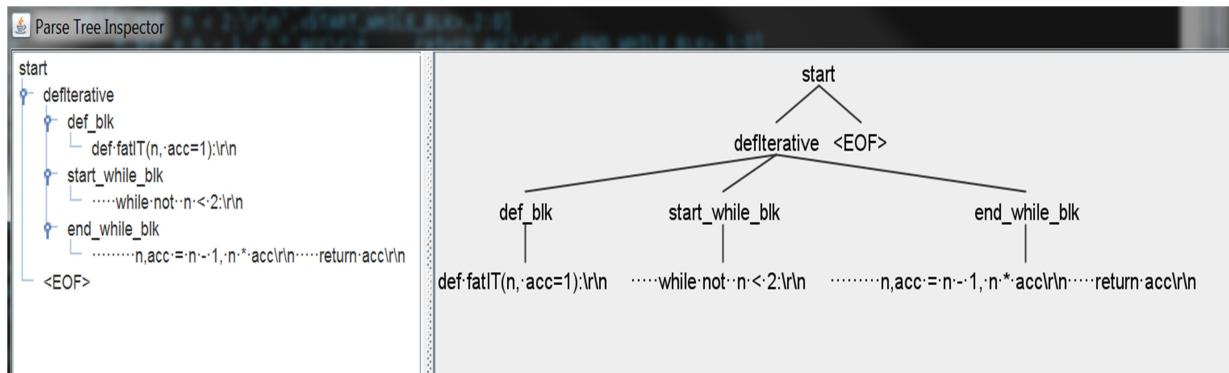


Figura 45 - Árvore sintática abstrata gerada para o algoritmo fatIT

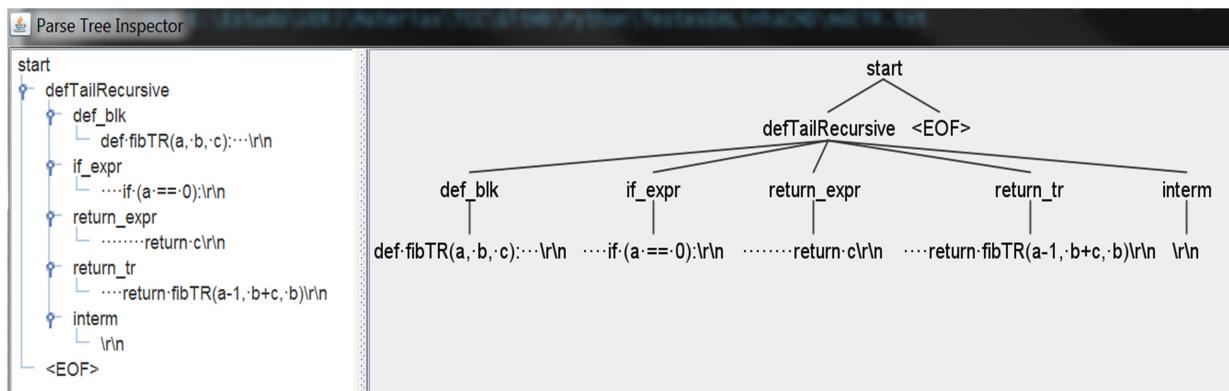


Figura 46 - Árvore sintética abstrata gerada para o algoritmo fibTR

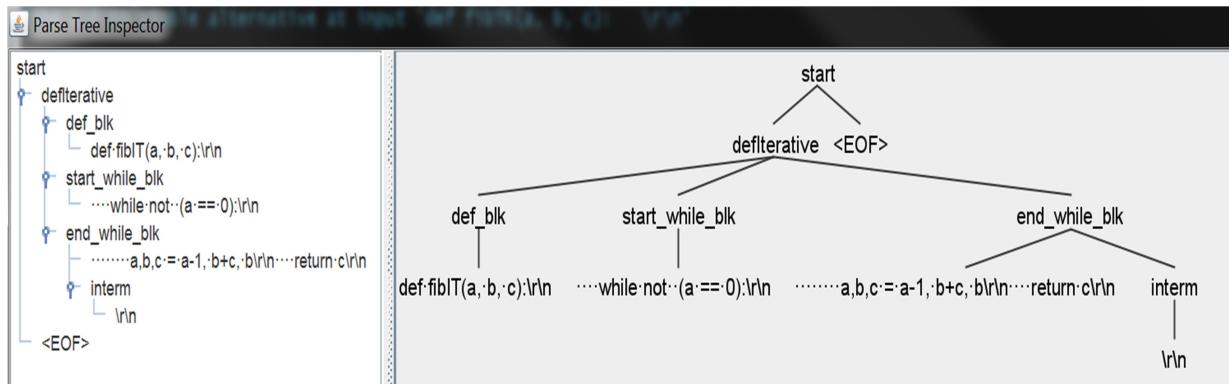


Figura 47 - Árvore sintética abstrata gerada para o algoritmo fibIT

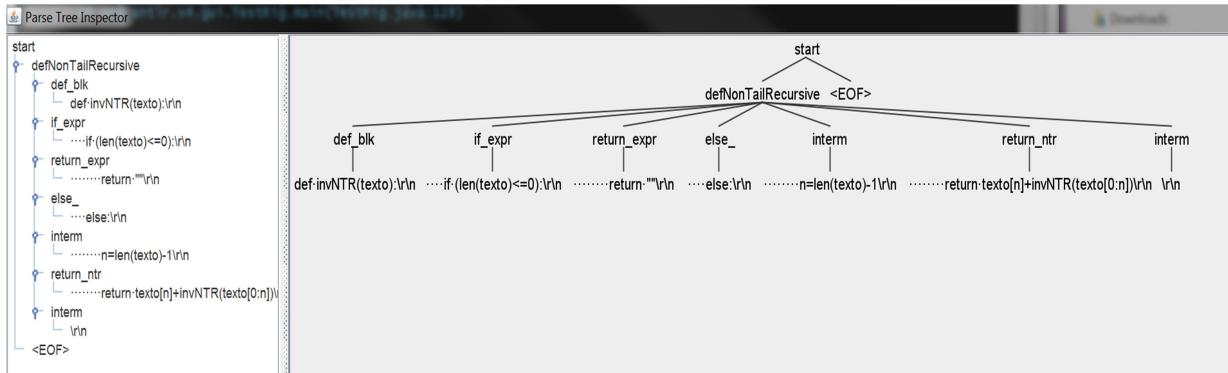


Figura 48 - Árvore sintática abstrata gerada para o algoritmo `invNTR`

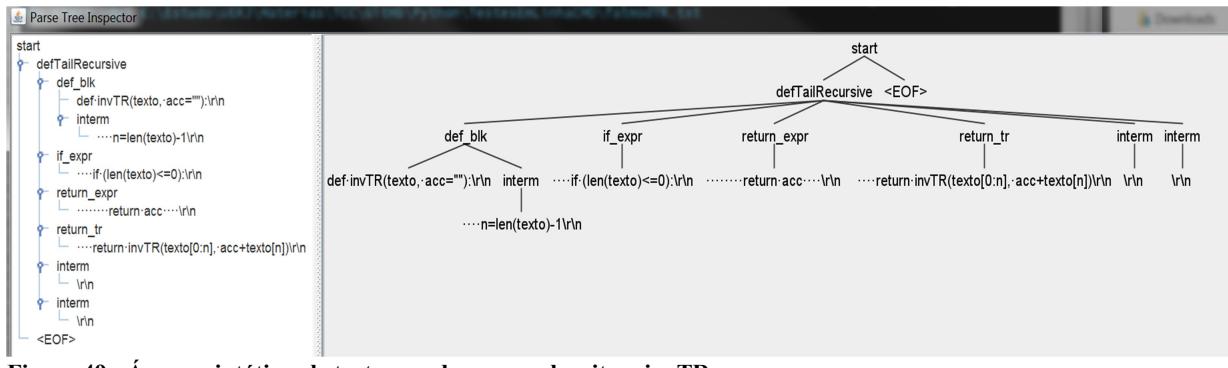


Figura 49 - Árvore sintática abstrata gerada para o algoritmo `invTR`

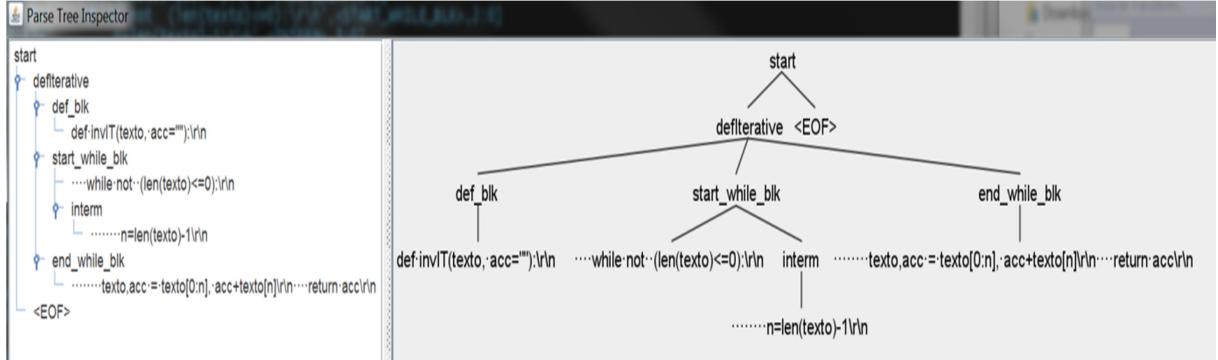


Figura 50 - Árvore sintática abstrata gerada para o algoritmo `invIT`

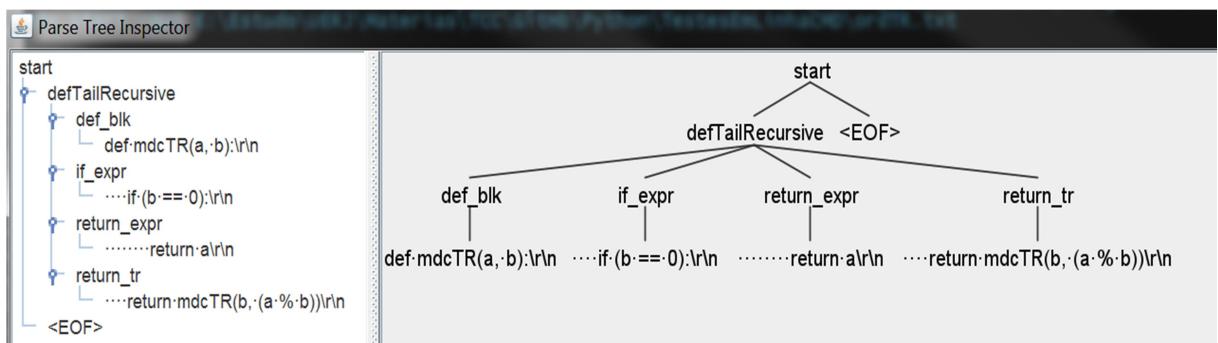


Figura 51 - Árvore sintática abstrata gerada para o algoritmo `mdcTR`

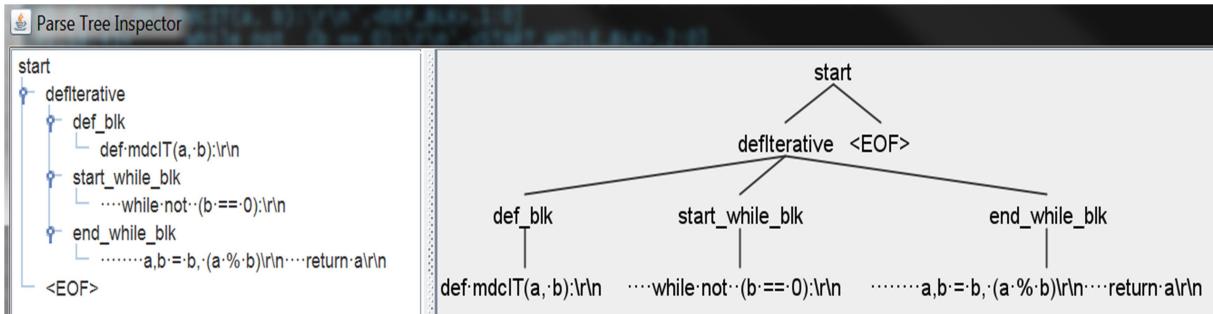


Figura 52 - Árvore sintática abstrata gerada para o algoritmo mdcIT

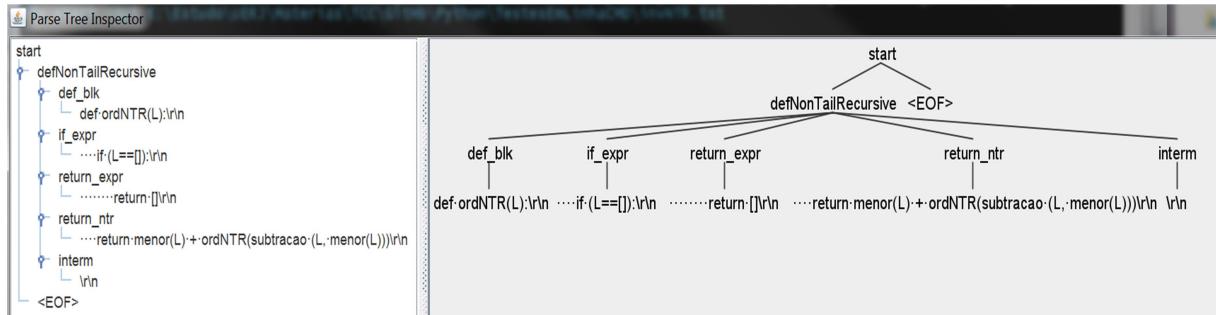


Figura 53 - Árvore sintática abstrata gerada para o algoritmo ordNTR

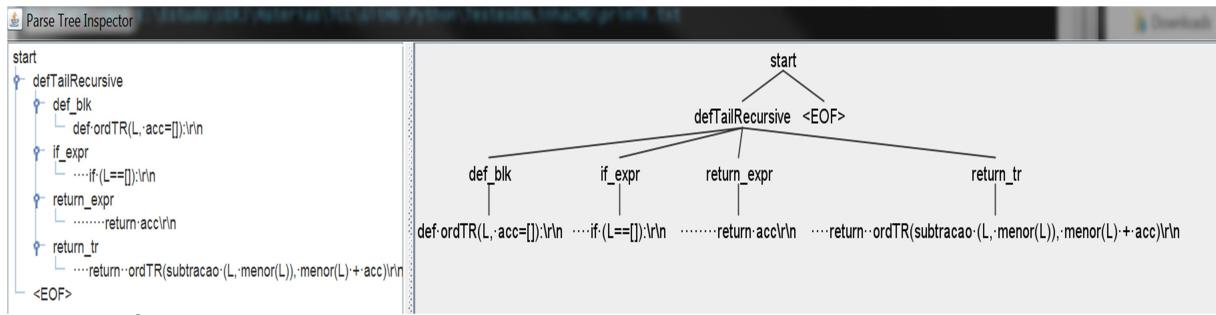


Figura 54 - Árvore sintática abstrata gerada para o algoritmo ordTR

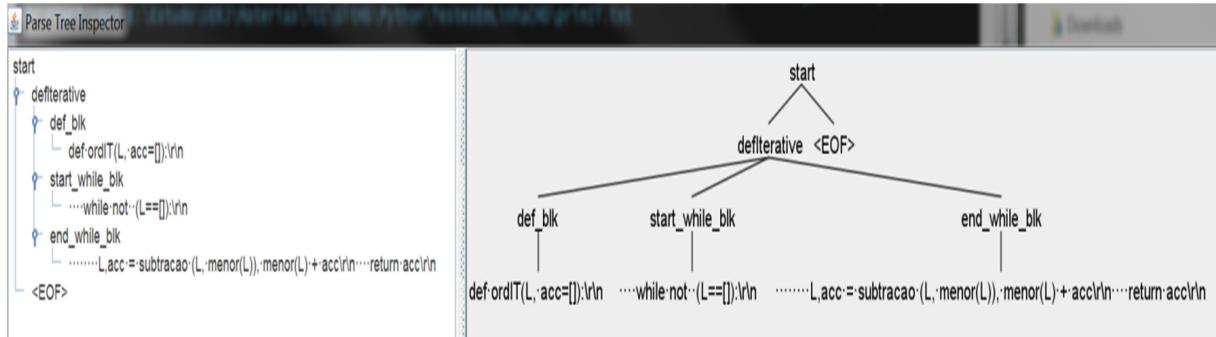


Figura 55 - Árvore sintática abstrata gerada para o algoritmo ordIT

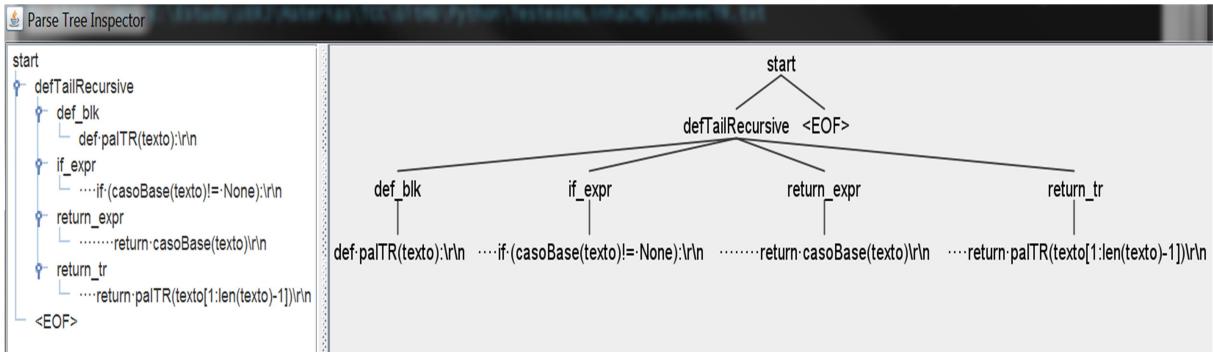


Figura 56 - Árvore sintática abstrata gerada para o algoritmo palTR

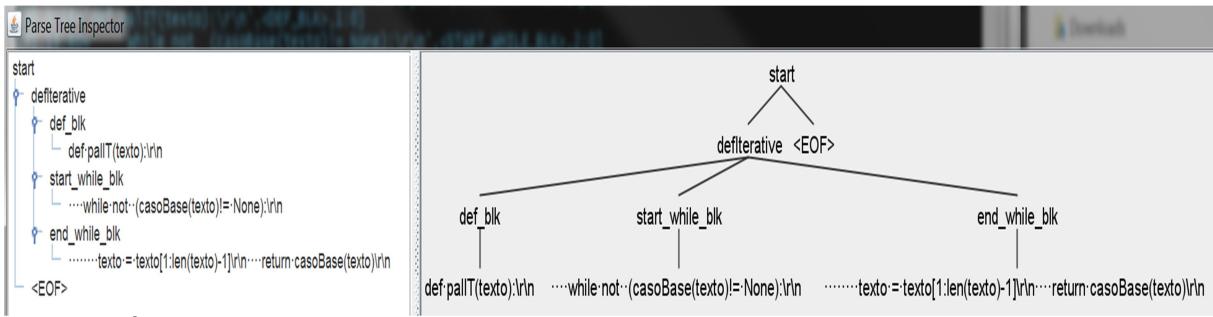


Figura 57 - Árvore sintática abstrata gerada para o algoritmo pallT

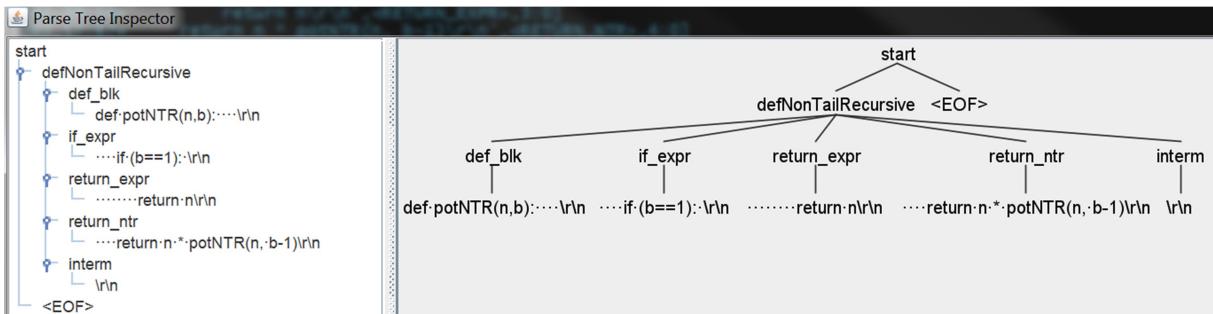


Figura 58 - Árvore sintética abstrata gerada para o algoritmo potNTR

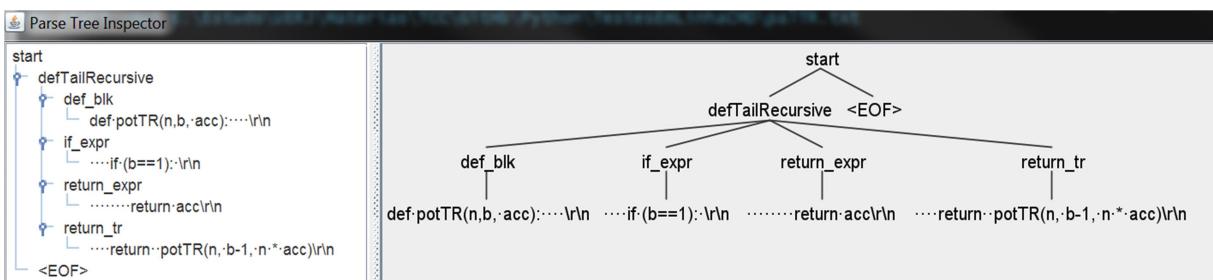


Figura 59 - Árvore sintética abstrata gerada para o algoritmo potTR

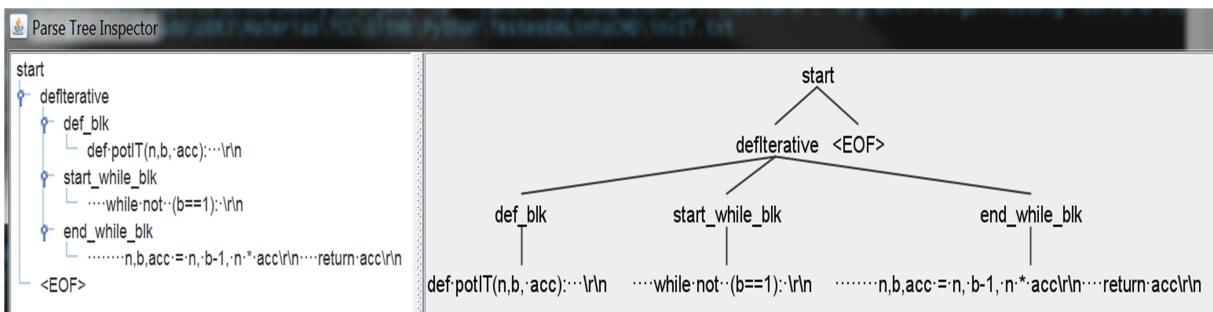


Figura 60 - Árvore sintética abstrata gerada para o algoritmo potIT

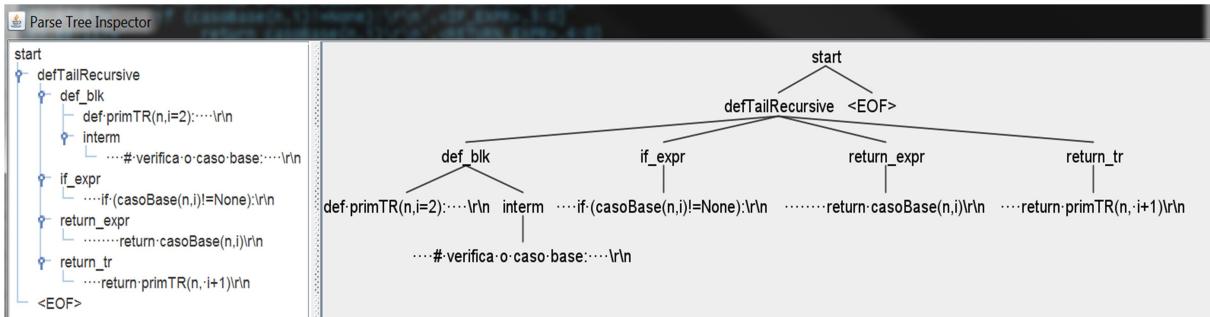


Figura 61 - Árvore sintática abstrata gerada para o algoritmo primTR

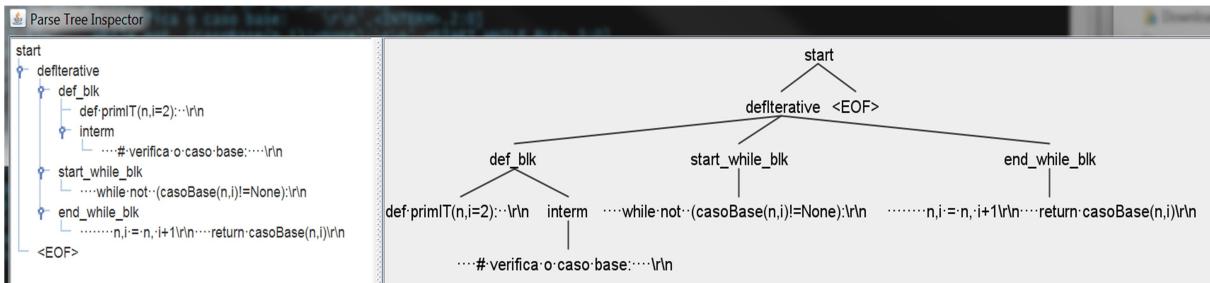


Figura 62 - Árvore sintática abstrata gerada para o algoritmo primIT

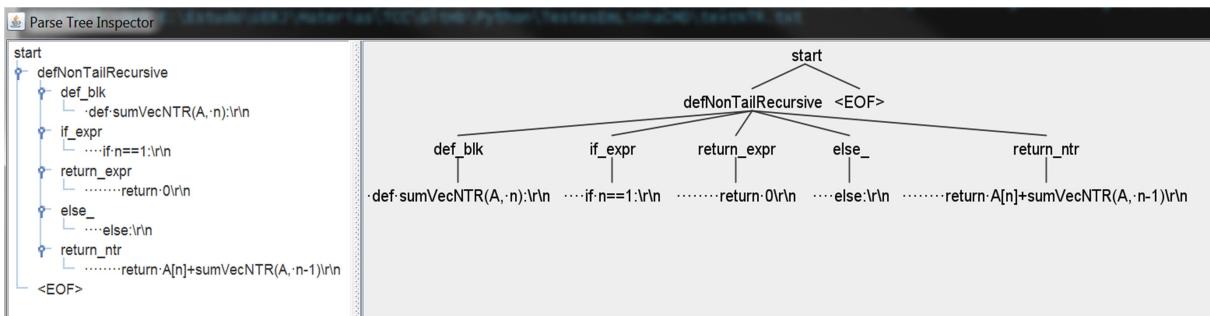


Figura 63 - Árvore sintática abstrata gerada para o algoritmo sumVecNTR

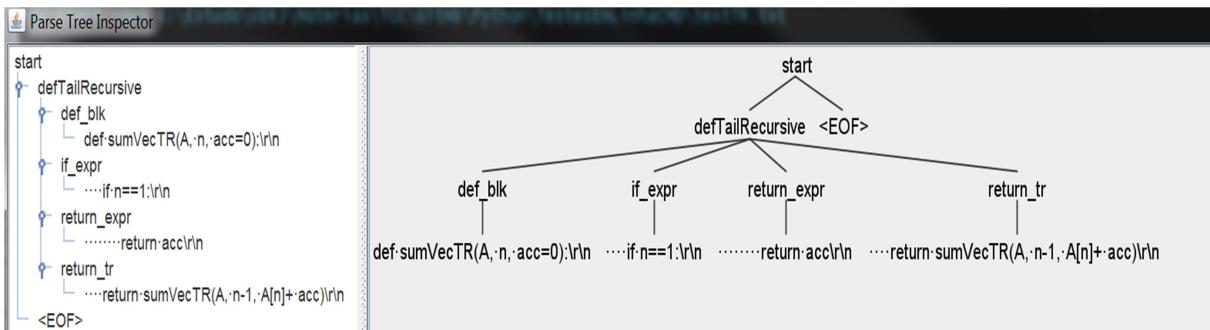


Figura 64 - Árvore sintática abstrata gerada para o algoritmo sumVecTR

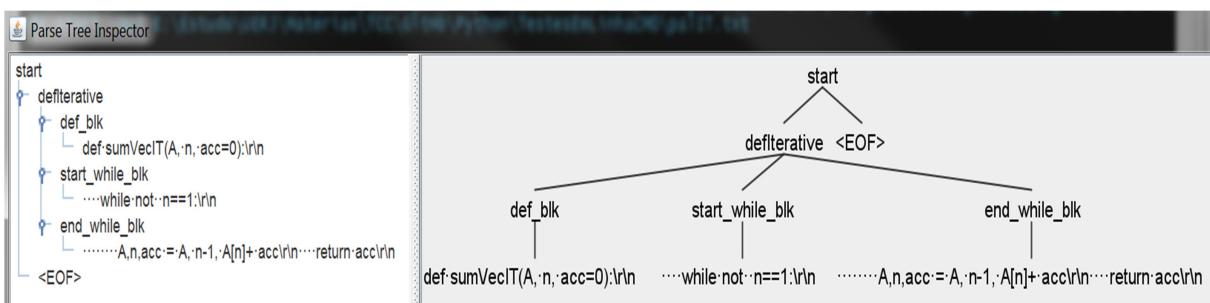


Figura 65 - Árvore sintática abstrata gerada para o algoritmo sumVecIT

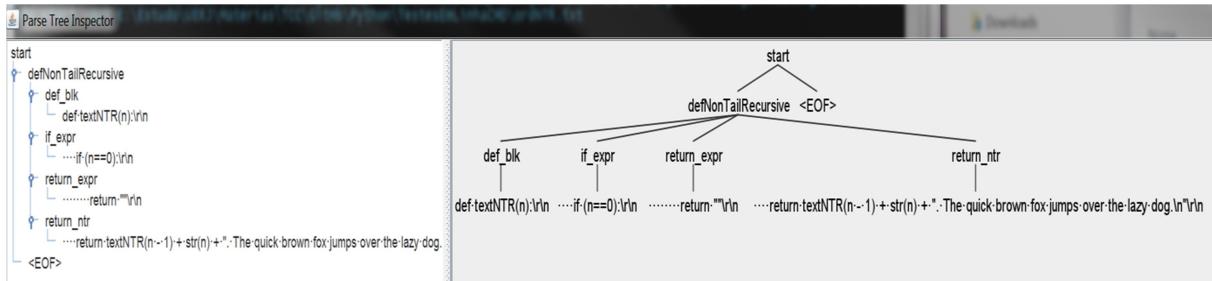


Figura 66 - Árvore sintática abstrata gerada para o algoritmo textNTR

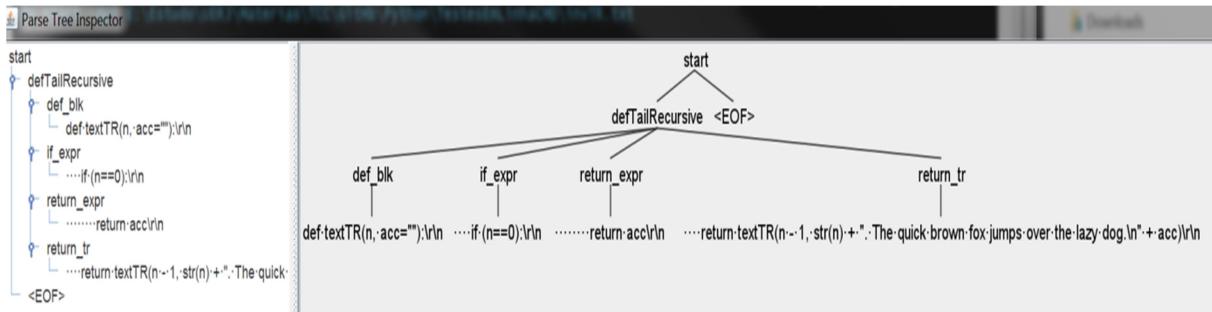


Figura 67 - Árvore sintática abstrata gerada para o algoritmo textTR

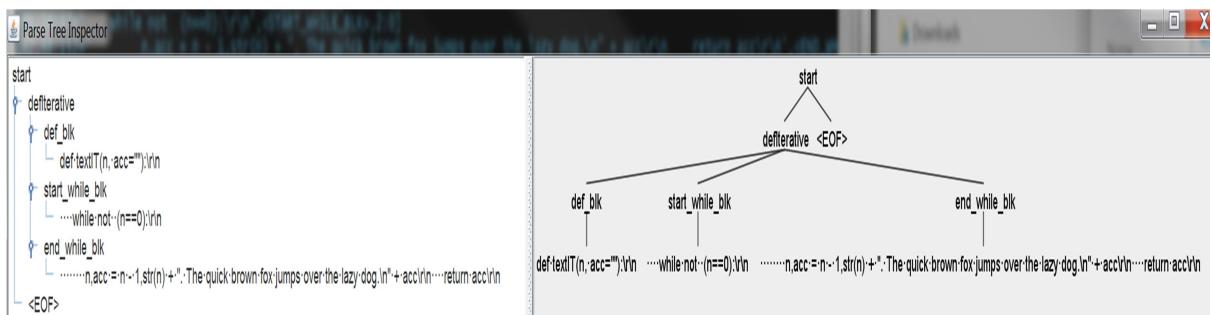


Figura 68 - Árvore sintática abstrata gerada para o algoritmo textIT

6.7. Apresentação dos gráficos dos experimentos realizados em algoritmos conversíveis no *RecPy*

Os gráficos dos experimentos foram elaborados a partir de dados coletados automaticamente, em arquivos *.CSV, com a inserção de linhas de código Python para a marcação dos tempos inicial e final de execução, em segundos, para cada valor de N experimentado, onde N = número de chamadas recursivas ou de iterações em cada ponto do gráfico. Para variar os valores de N, foram criados loops que iam incrementando N até que o script travasse (presumidamente, por estouro do limite da pilha). Nos casos em que não houve esse travamento, estipulou-se um valor alto de N para se interromper a execução do script. Cada combinação de Tempo x valor de N era gravada no arquivo *.CSV para gerar um ponto a mais na curva do respectivo gráfico.

Terminado o procedimento de coleta de dados, o conteúdo do arquivo *.CSV foi exportado para o aplicativo Power BI (versão 2.88.1385.0 64-bit).

6.7.1. Gráficos de resultados dos experimentos

6.7.1.1. Gráfico do algoritmo de factorial recursivo não caudal - fatNTR

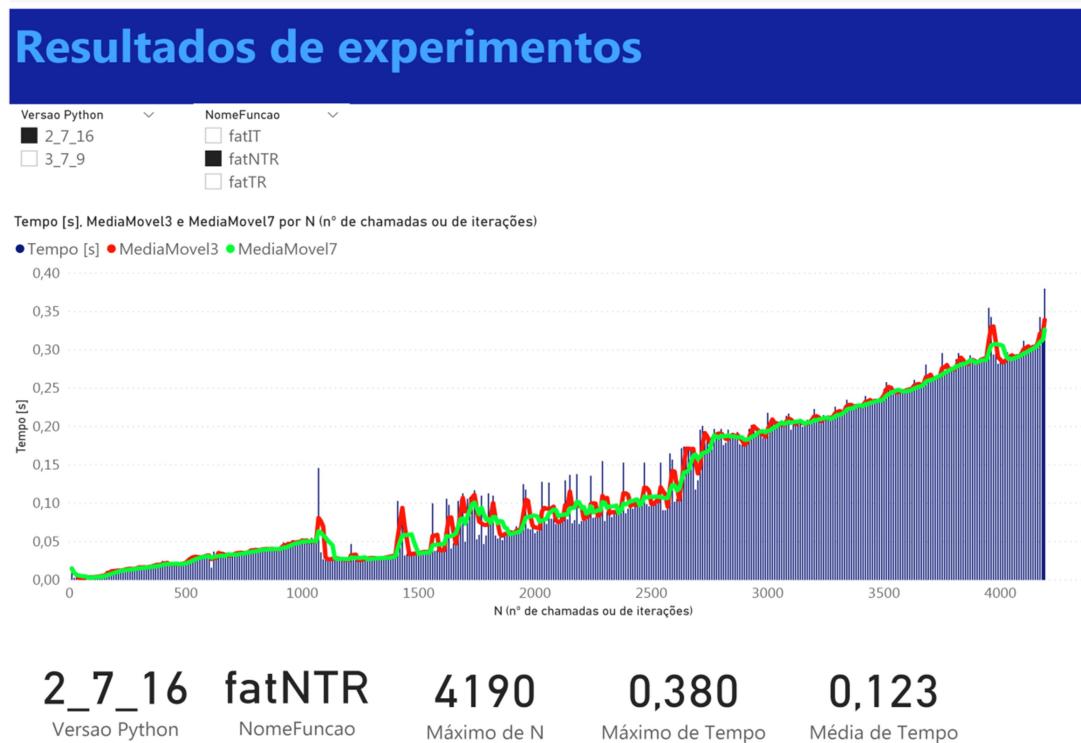


Figura 69 - Exemplo de gráfico resultante do experimento de factorial recursivo não caudal – fatNTR

6.7.1.2. Gráfico do algoritmo de factorial recursivo caudal - fatTR

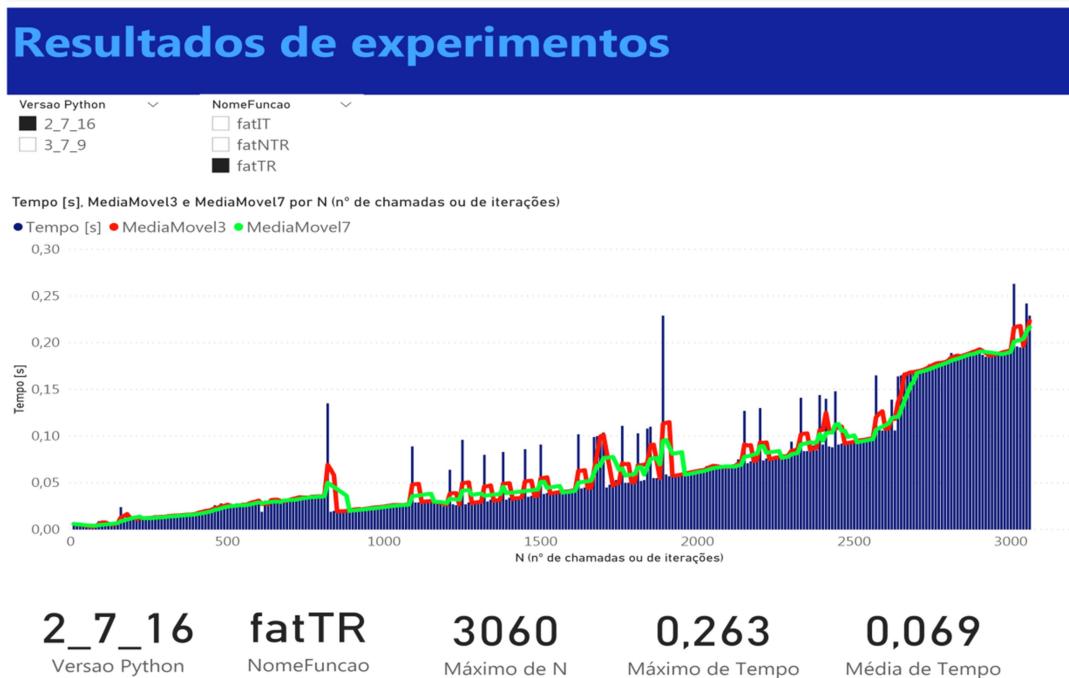


Figura 70 - Exemplo de gráfico resultante do experimento de factorial recursivo caudal – fatTR

6.7.1.3. Gráfico do algoritmo de factorial iterativo - fatIT



Figura 71 - Exemplo de gráfico resultante do experimento de factorial iterativo - fatIT

6.7.1.4. Gráfico do algoritmo factorial modificado recursivo não caudal - fatmodNTR

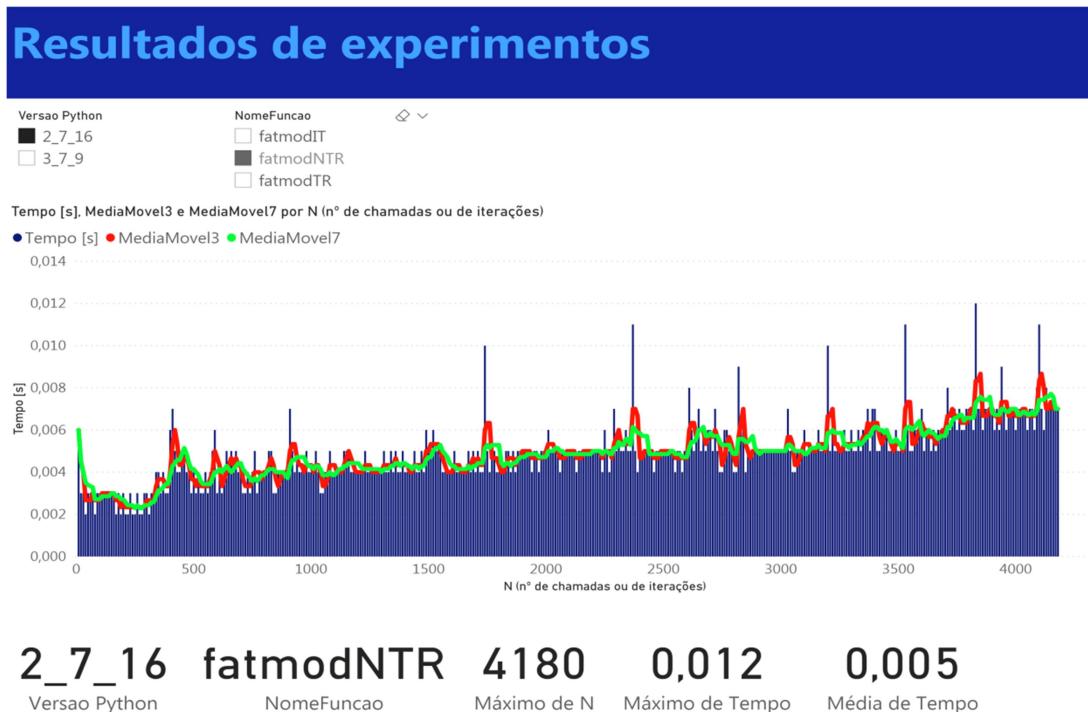


Figura 72 - Exemplo de gráfico resultante do experimento de factorial iterativo – fatmodNTR

6.7.1.5. Gráfico do algoritmo de factorial modificado recursivo caudal - fatmodTR

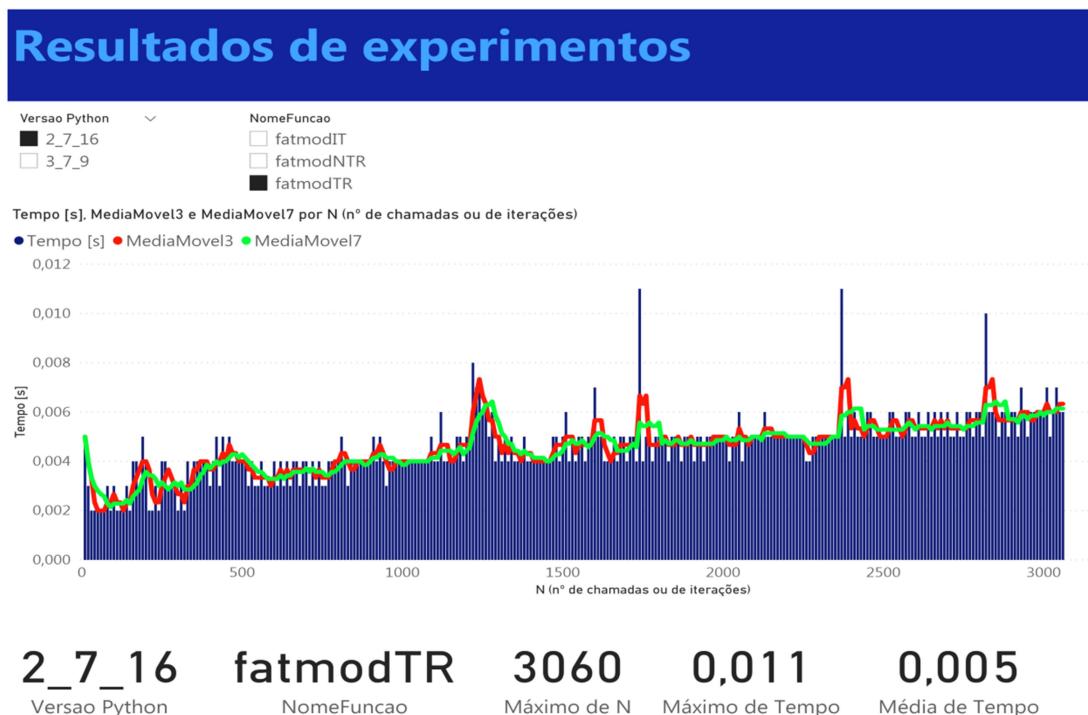


Figura 73 - Exemplo de gráfico resultante do experimento de factorial modificado recursivo caudal - fatmodTR

6.7.1.6. Gráfico do algoritmo de factorial modificado iterativo - fatmodIT

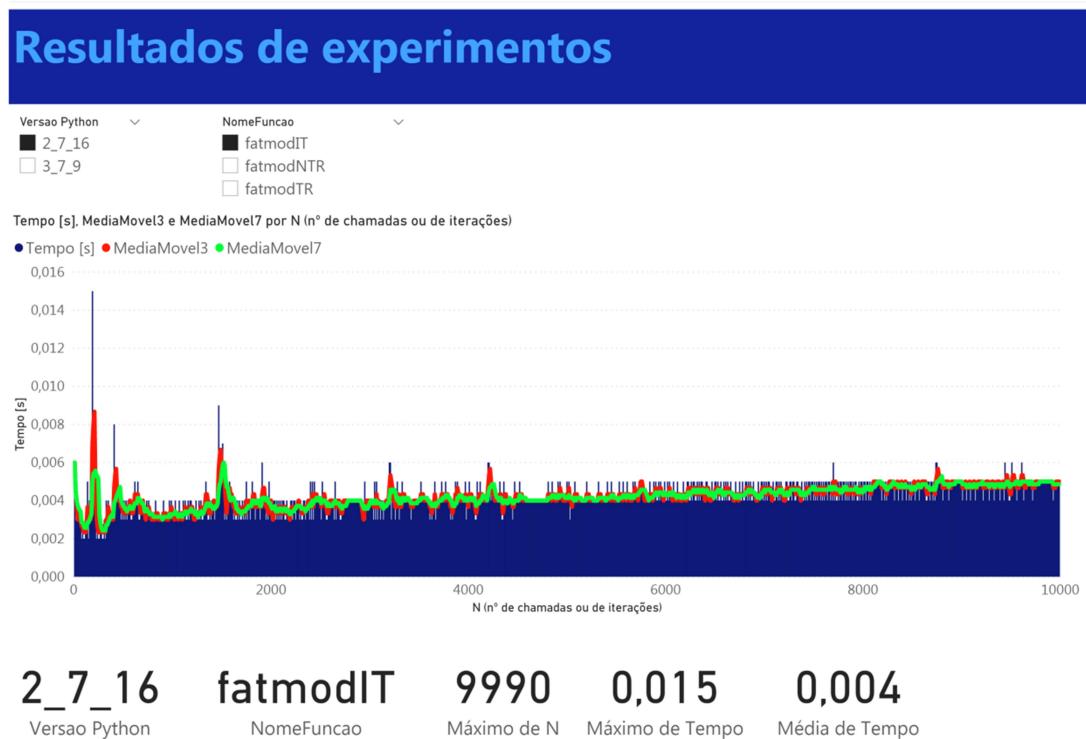


Figura 74 - Exemplo de gráfico resultante do experimento de factorial modificado iterativo - fatmodIT

6.7.1.7. Gráfico do algoritmo de Fibonacci recursivo caudal - fibTR

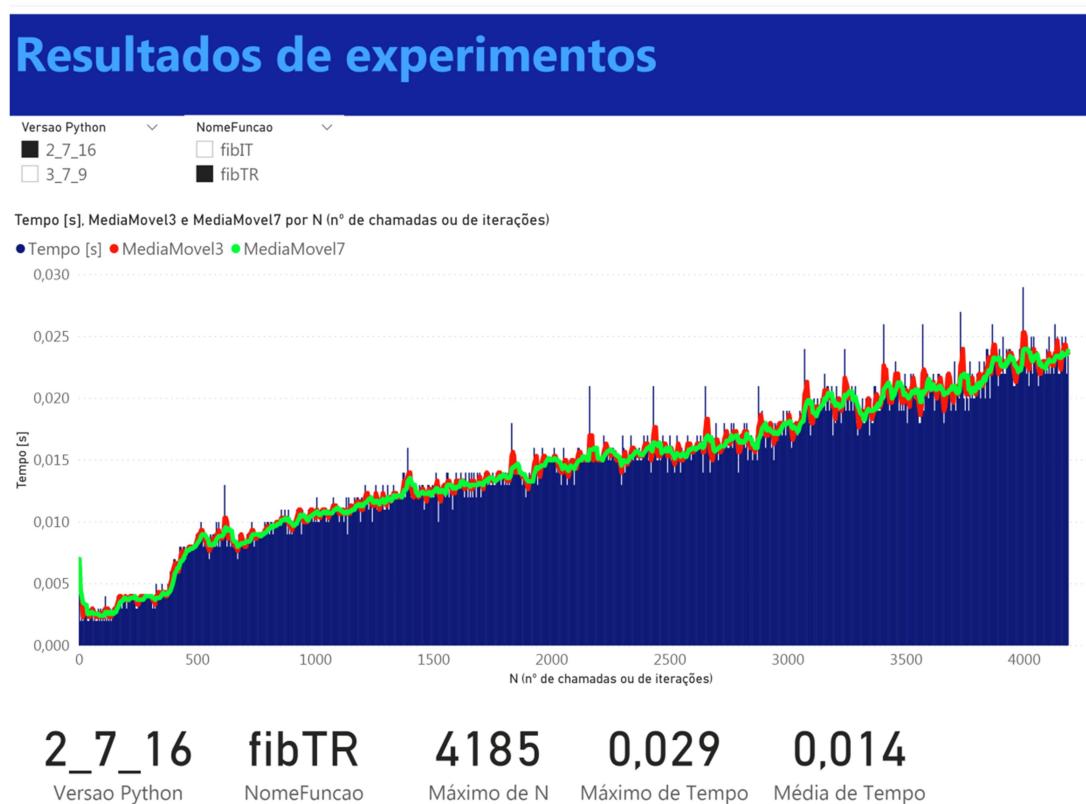


Figura 75 - Exemplo de gráfico resultante do experimento de Fibonacci recursivo caudal - fibTR

6.7.1.8. Gráfico do algoritmo de Fibonacci iterativo - fibIT



Figura 76 - Exemplo de gráfico resultante do experimento de Fibonacci iterativo - fibIT

6.7.1.9. Gráfico do algoritmo de inversão de string recursivo não caudal - invNTR

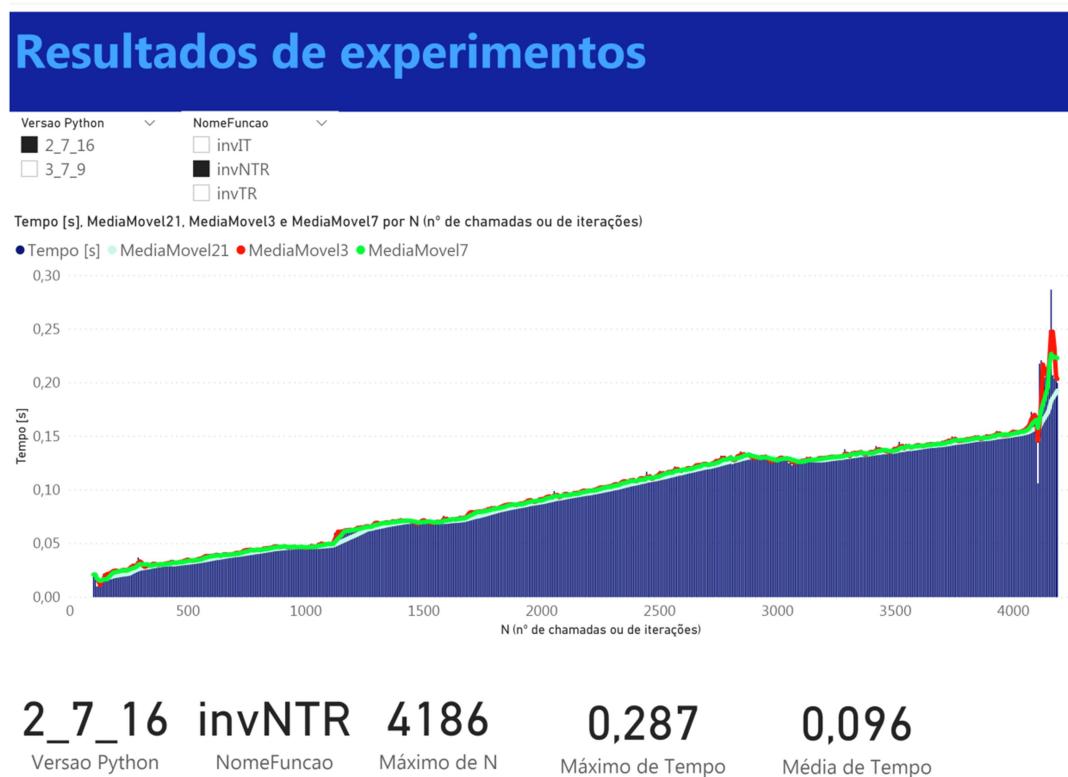


Figura 77 - Exemplo de gráfico resultante do experimento de inversão de string recursivo não caudal – invNTR

6.7.1.10. Gráfico do algoritmo de inversão de string recursivo caudal - invTR

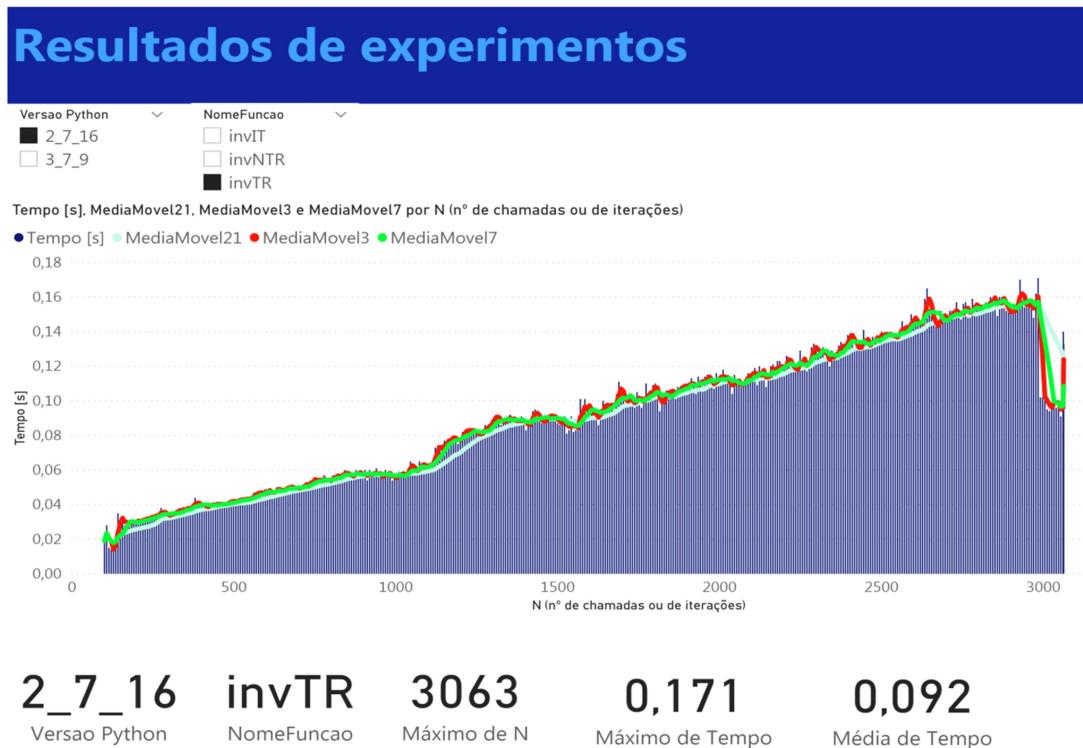


Figura 78 - Exemplo de gráfico resultante do experimento de inversão de string recursivo caudal – invTR

6.7.1.11. Gráfico do algoritmo de inversão de string iterativo - invIT

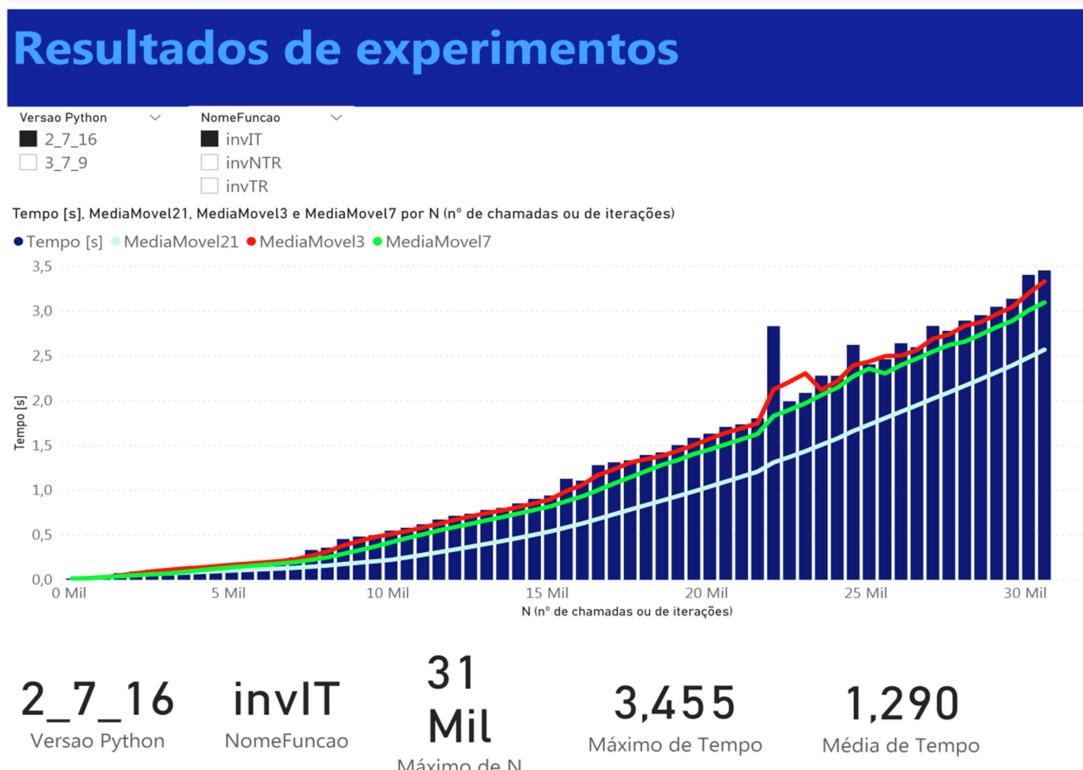


Figura 79 - Exemplo de gráfico resultante do experimento de inversão de string iterativo – invIT

6.7.1.12. Gráfico do algoritmo de MDC recursivo caudal - mdcTR

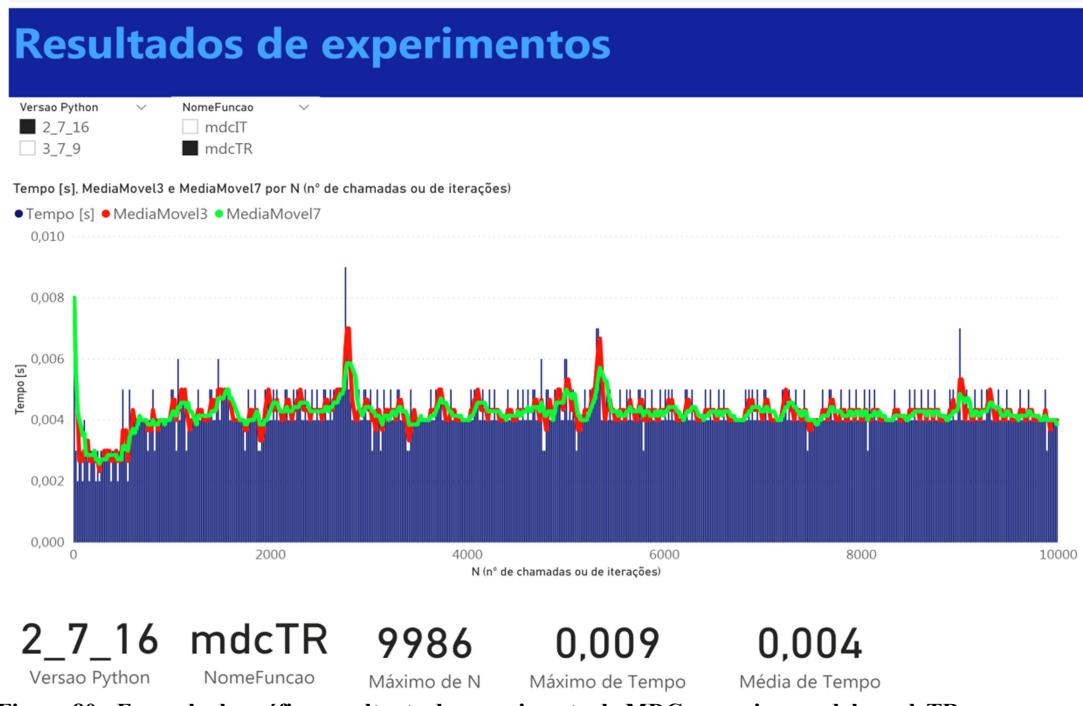


Figura 80 - Exemplo de gráfico resultante do experimento de MDC recursivo caudal – mdcTR

6.7.1.13. Gráfico do algoritmo de MDC iterativo - mdcIT

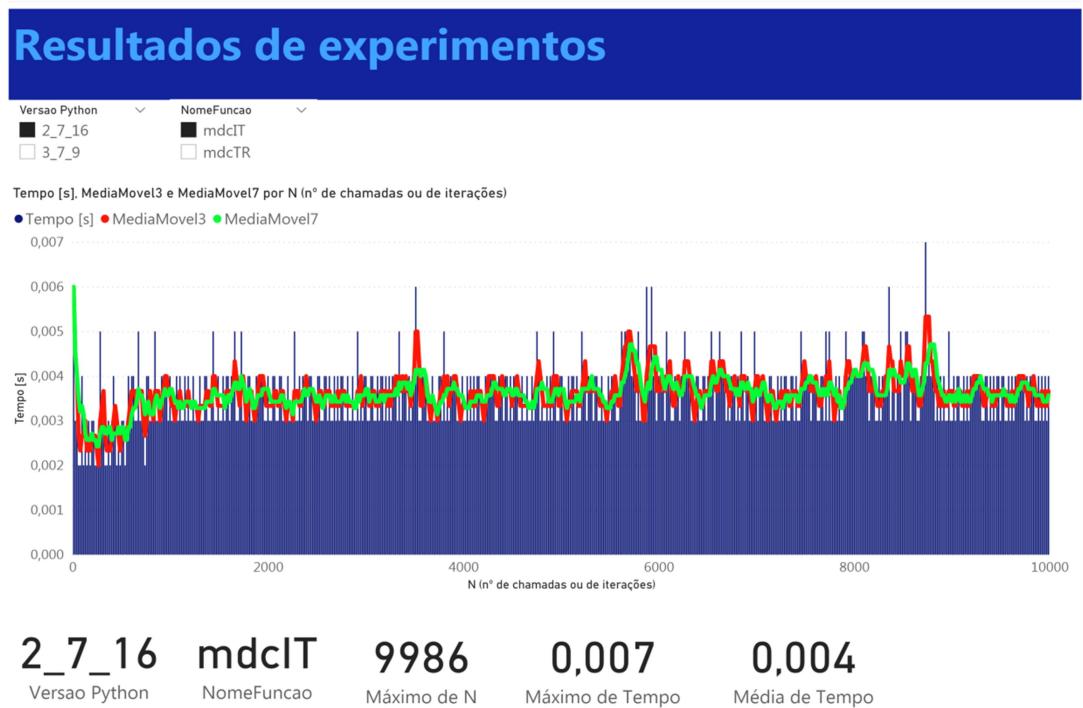


Figura 81 - Exemplo de gráfico resultante do experimento de MDC iterativo – mdcIT

6.7.1.14. Gráfico do algoritmo de ordenação recursivo não caudal - ordNTR

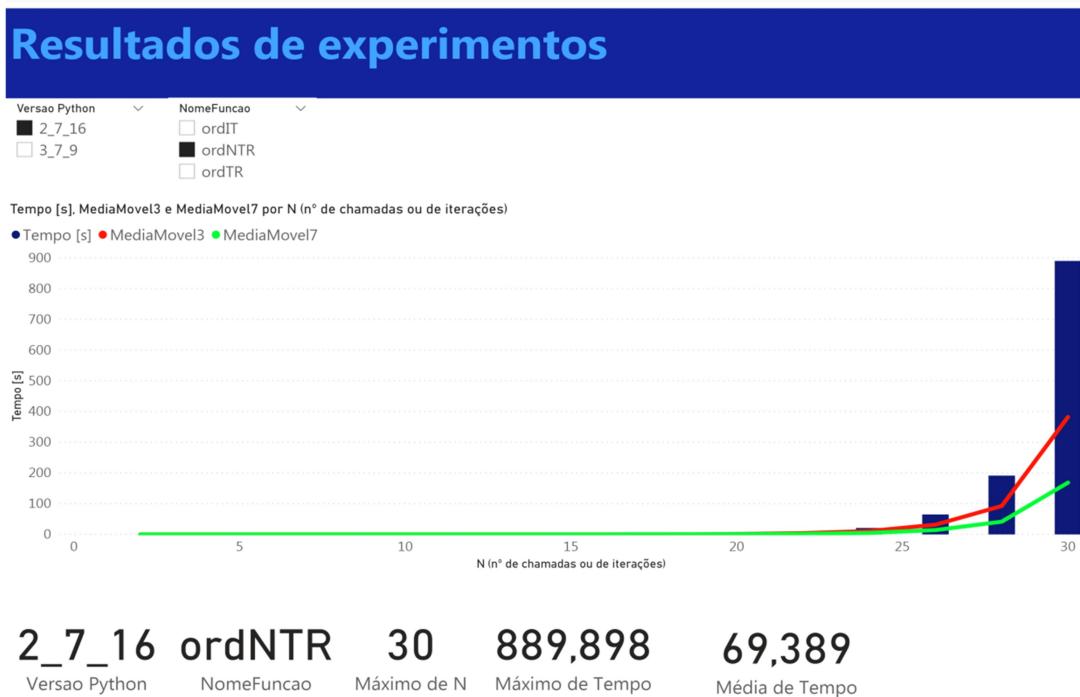


Figura 82 - Exemplo de gráfico resultante do experimento de ordenação recursivo não caudal - ordNTR

6.7.1.15. Gráfico do algoritmo de ordenação recursivo caudal - ordTR



Figura 83 - Exemplo de gráfico resultante do experimento de ordenação recursivo caudal – ordTR

6.7.1.16. Gráfico do algoritmo de ordenação iterativo - ordIT



Figura 84 - Exemplo de gráfico resultante do experimento de ordenação iterativo - ordIT

6.7.1.17. Gráfico do algoritmo de palíndromo recursivo caudal - palTR

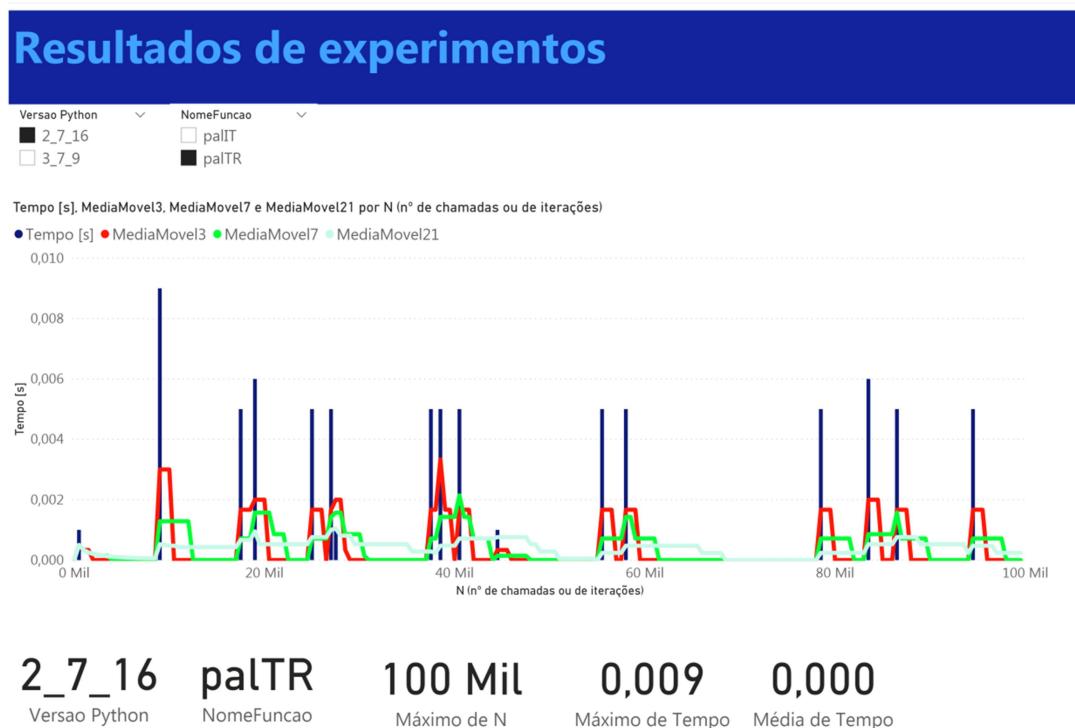


Figura 85 - Exemplo de gráfico resultante do experimento de palíndromo recursivo caudal - palTR

6.7.1.18. Gráfico do algoritmo de palíndromo iterativo - palIT

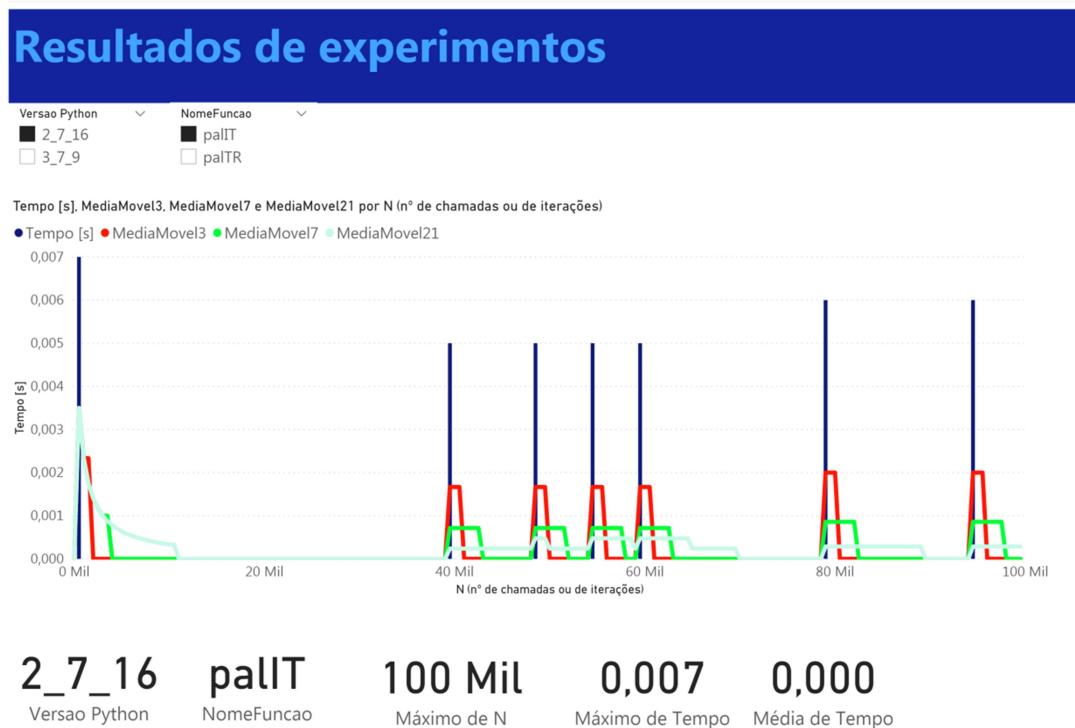


Figura 86 - Exemplo de gráfico resultante do experimento de palíndromo iterativo - palIT

6.7.1.19. Gráfico do algoritmo de potenciação recursivo não caudal - potNTR



Figura 87 - Exemplo de gráfico resultante do experimento de potenciação recursivo não caudal – potNTR

6.7.1.20. Gráfico do algoritmo de potenciação recursivo caudal - potTR

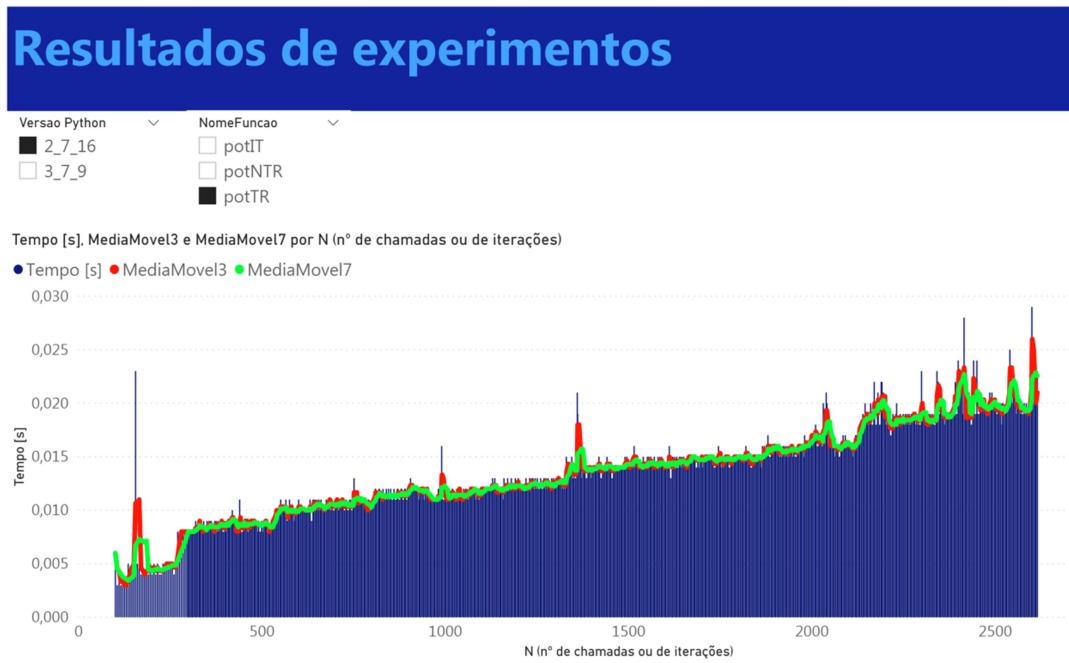


Figura 88 - Exemplo de gráfico resultante do experimento de potenciação recursivo caudal - potTR

6.7.1.21. Gráfico do algoritmo de potenciação iterativo - potIT



Figura 89 - Exemplo de gráfico resultante do experimento de potenciação iterativo - potIT

6.7.1.22. Gráfico do algoritmo de números primos recursivo caudal – primTR

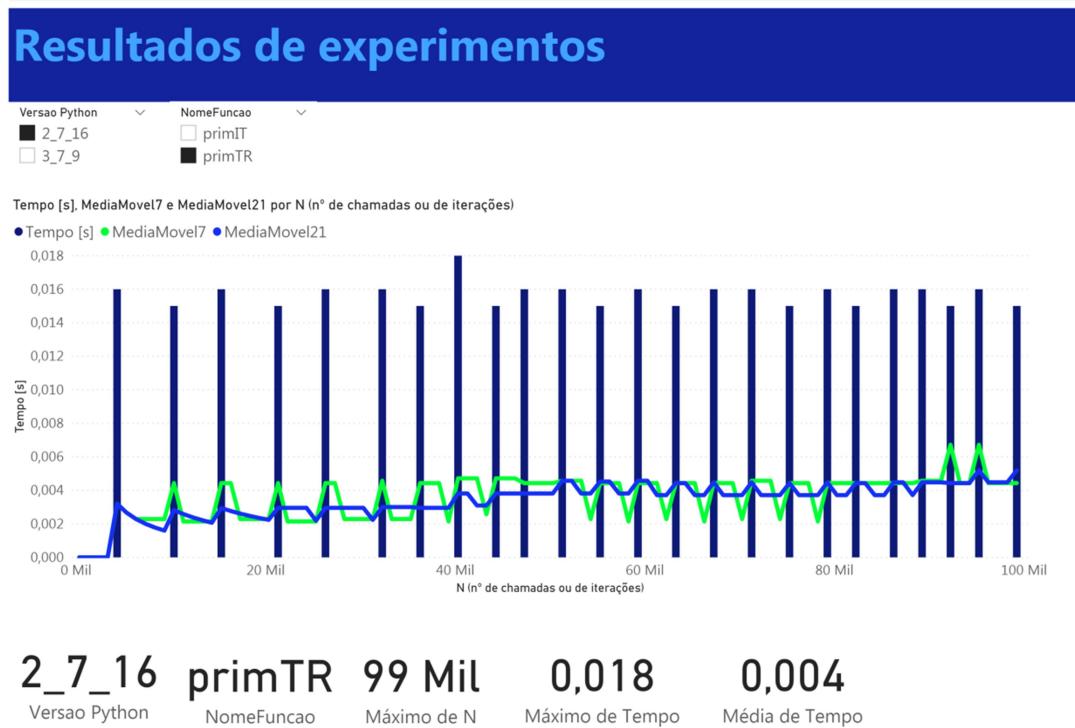


Figura 90 - Exemplo de gráfico resultante do experimento números primos recursivo caudal - primTR

6.7.1.23. Gráfico do algoritmo de números primos iterativo - primIT

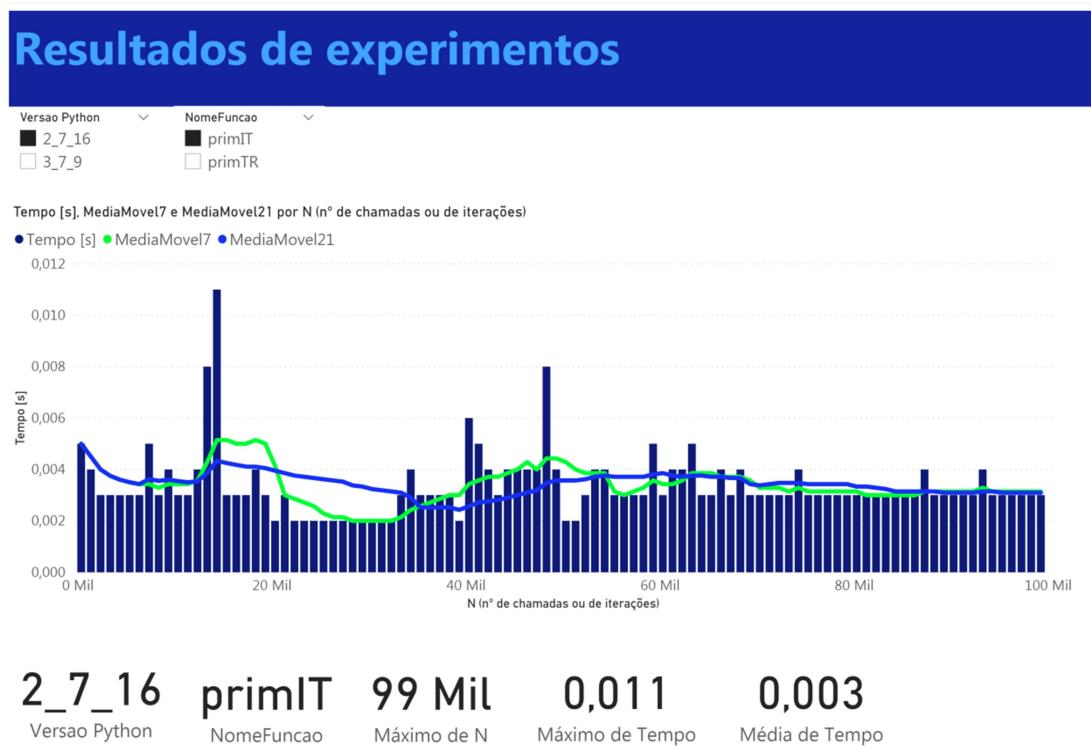


Figura 91 - Exemplo de gráfico resultante do experimento de números primos iterativo - primIT

6.7.1.24. Gráfico do algoritmo soma itens de vetor recursivo não caudal - sumVecNTR

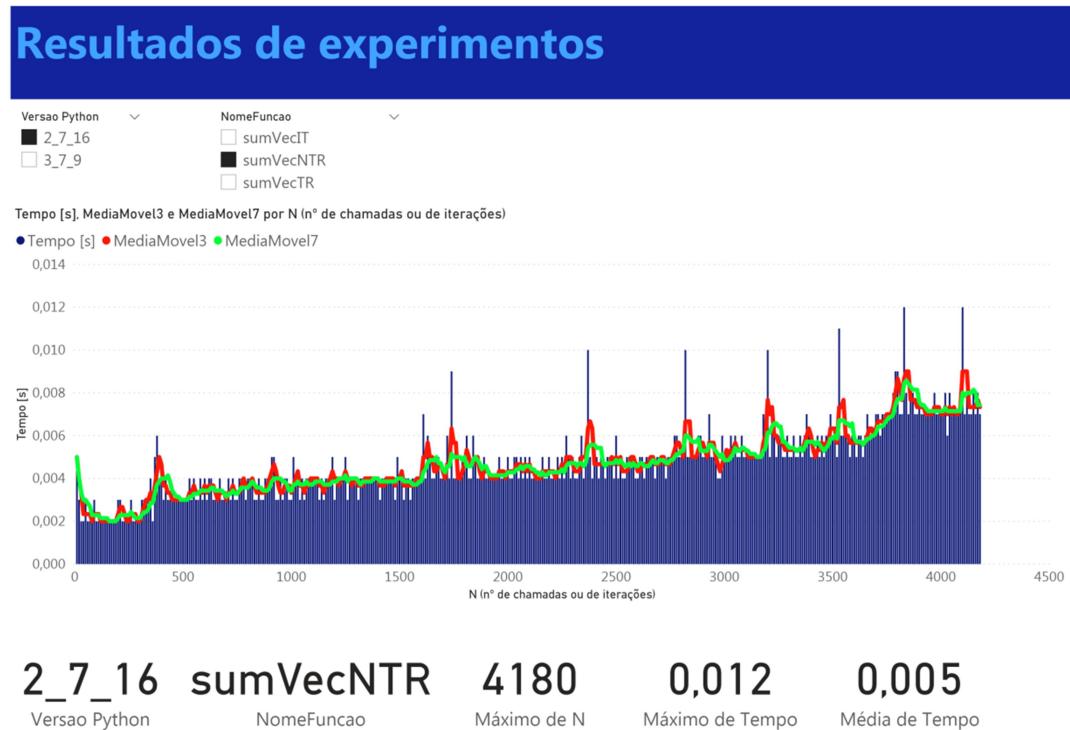


Figura 92 - Exemplo de gráfico resultante do experimento soma itens de um vetor recursivo não caudal - sumVecNTR

6.7.1.25. Gráfico do algoritmo de soma itens de um vetor recursivo caudal - sumVecTR

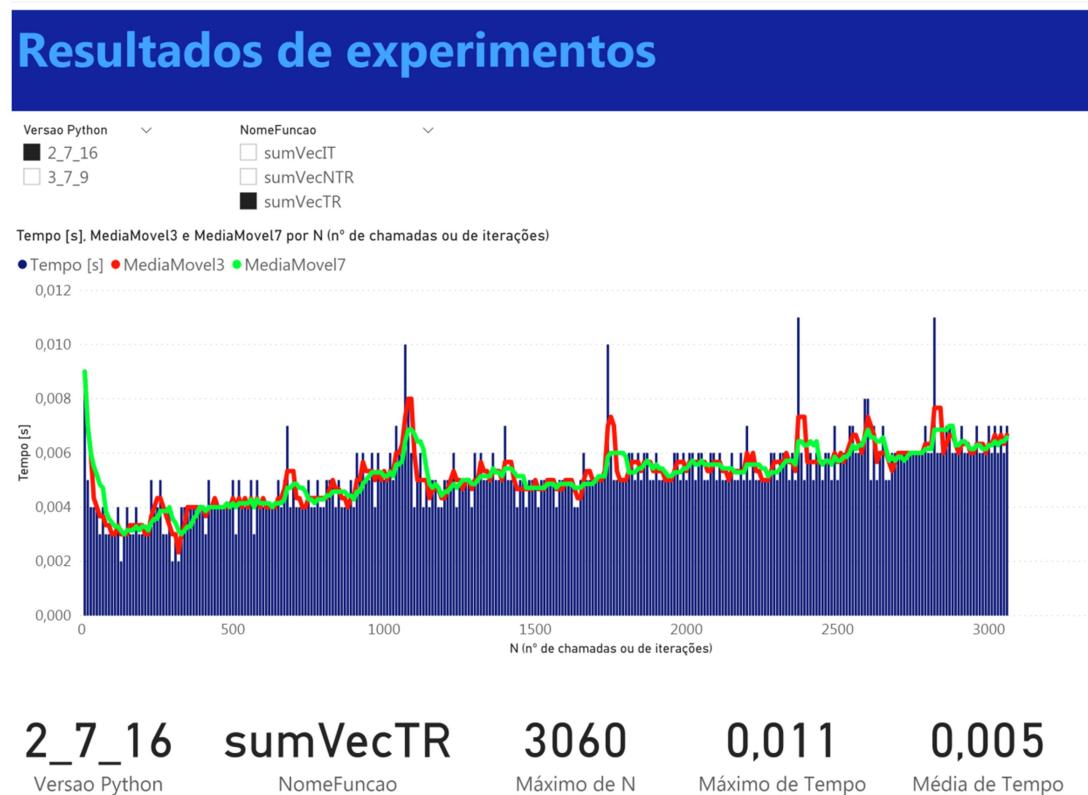


Figura 93 - Exemplo de gráfico resultante do experimento soma de itens de um vetor recursivo caudal - sumVecTR

6.7.1.26. Gráfico do algoritmo de soma de itens de um vetor iterativo - sumVecIT

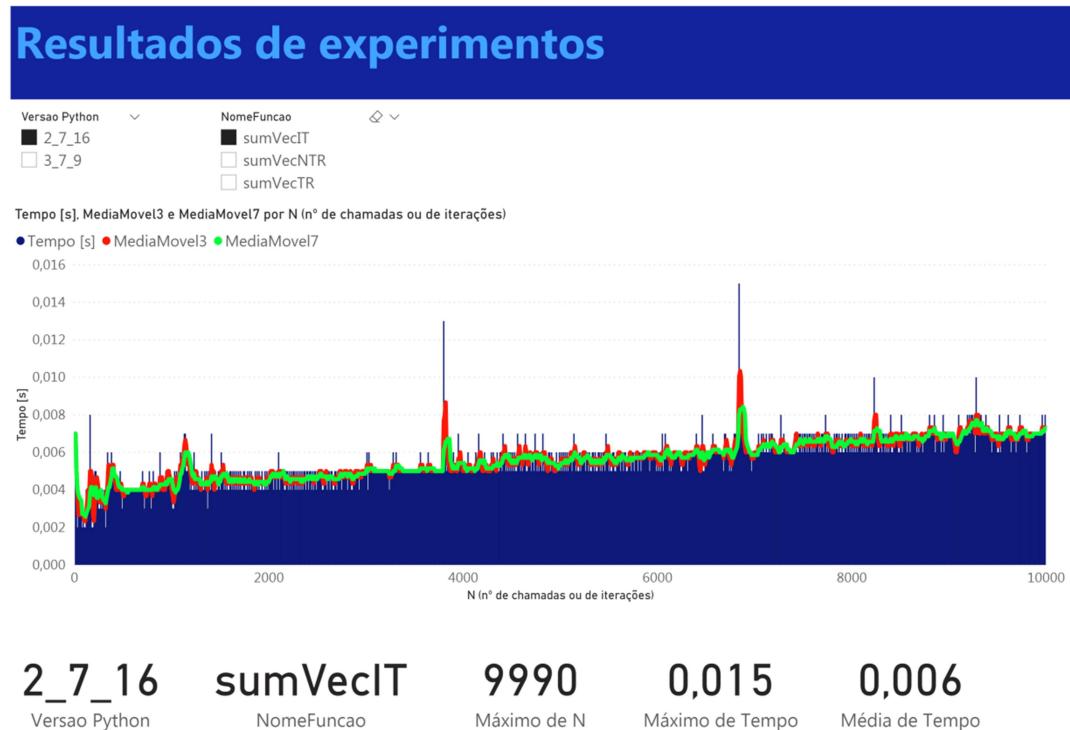


Figura 94 - Exemplo de gráfico resultante do experimento soma de itens de um vetor iterativo - sumVecIT

6.7.1.27. Gráfico do algoritmo de concatenação de texto recursivo não caudal - textNTR

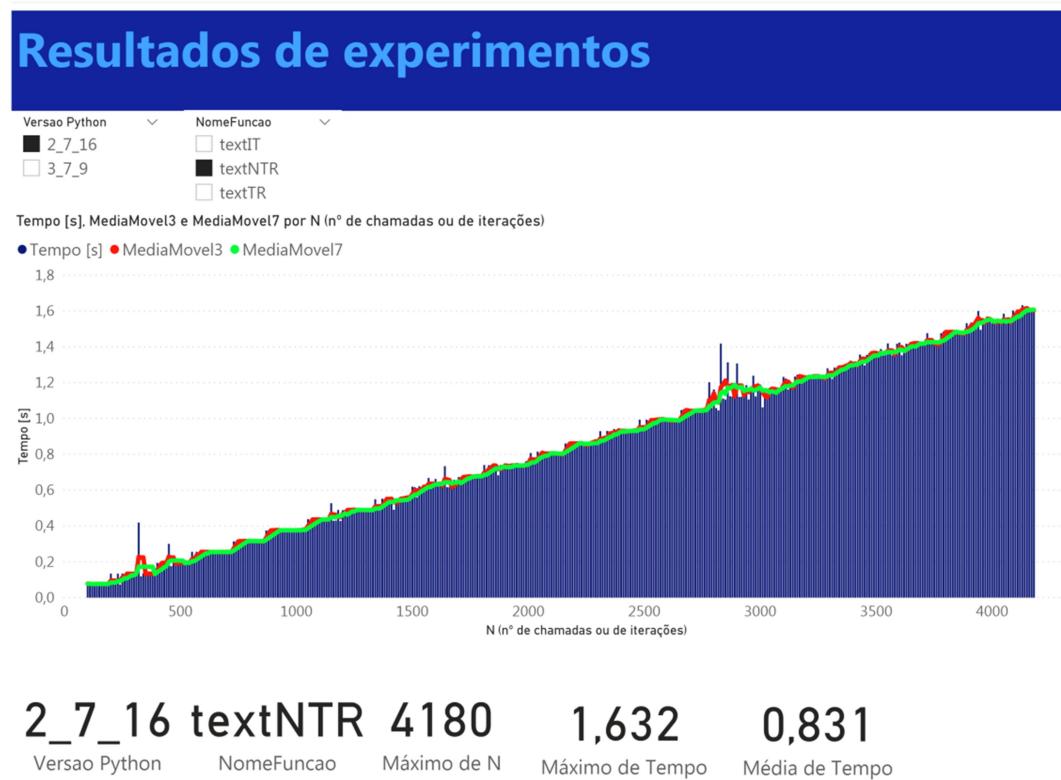


Figura 95 - Exemplo de gráfico resultante do experimento de concatenação de texto recursivo não caudal - textNTR

6.7.1.28. Gráfico do algoritmo de concatenação de texto recursivo caudal - textTR

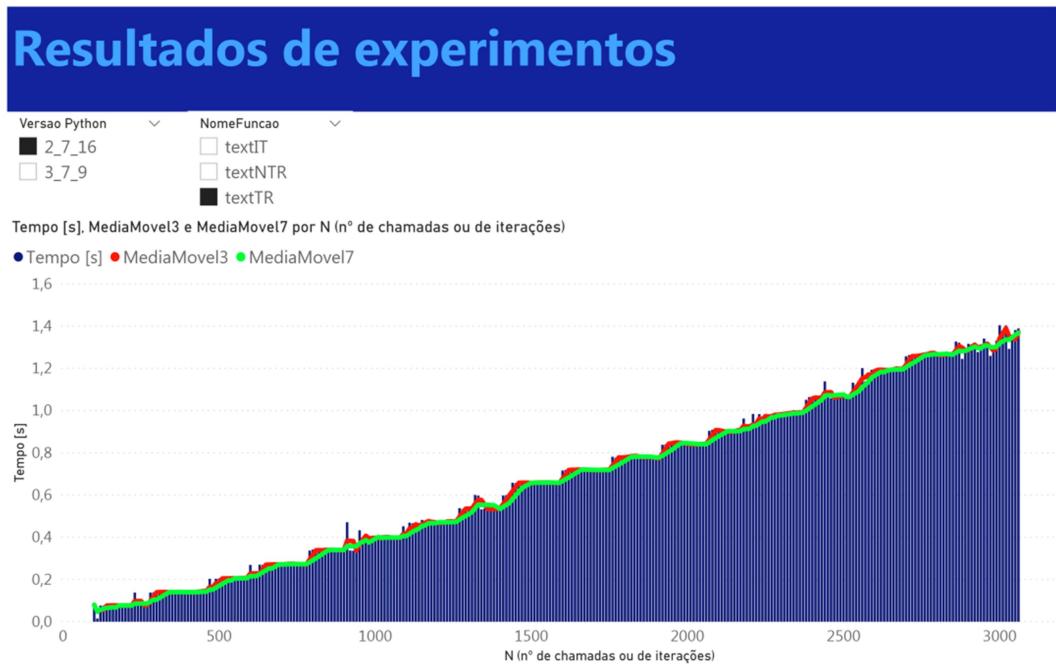


Figura 96 - Exemplo de gráfico resultante do experimento de concatenação de texto recursivo caudal – textTR

6.7.1.29. Gráfico do algoritmo de concatenação de texto iterativo - textIT



Figura 97 - Exemplo de gráfico resultante do experimento de concatenação de texto iterativo - textIT

6.8. Síntese dos Resultados

6.8.1. Quanto ao sucesso no procedimento de conversão e reconhecimento do algoritmo no RecPy

Algoritmo	Versão original	TR->IT: a conversão funcionou?	NTR->TR: a conversão funcionou?	NTR->IT: a conversão funcionou?	IT->TR: a conversão funcionou?
FatMod	fatModNTR	R	R	R	R
Fatorial	fatNTR	R	R	R	R
Fibonacci	fibTR	R	E	E	R
Inverte String	invNTR	R	R	R	R
MDC	mdcTR	R	E	E	R
Ordenação	ordNTR	R	R	R	R
Palíndromo	palTR	R	E	E	R
Potenciação	potNTR	R	R	R	R
Primo	primTR	R	E	E	R
Soma de vetor	sumVecNTR	R	R	R	R
Texto	textNTR	R	R	R	R

Quadro 8 - Síntese de resultados quanto ao sucesso no procedimento de conversão e de reconhecimento das funções testadas

Legendas:

R Sim, funcionou perfeitamente e de modo automático

E Não houve versão NTR

6.8.2. Quanto ao valor máximo da variável N (quantidade de chamadas recursivas ou de iterações)

Tabela 1 - Resultados quanto ao valor máximo da variável N (quantidade de chamadas recursivas ou de iterações)

Algoritmo	Iterativo (IT) Máx. valor N Python 2.7.16	Recursivo caudal (TR) Máx. valor N Python 2.7.16	Recursivo não caudal (NTR) Máx. valor N Python 2.7.16	Iterativo (IT) Máx. valor N Python 3.7.9	Recursão Caudal (TR) Máx. valor N Python 3.7.9	Recursão não Caudal (NTR) Máx. valor N Python 3.7.9
FatMod	> 9990	3060	4180	> 9990	2160	2610
Fatorial	> 108 mil	3064	4188	>120 mil	2163	2616
Fibonacci	> 9980	4185	£	> 9980	2615	£
Inverte String	> 31 mil	3063	4186	> 30 mil	2162	2613
MDC	> 9986	> 9986	£	> 9986	>9986	£
Ordenação	> 28	> 28	> 30	> 28	> 28	> 28
Palíndromo	> 100 mil	> 100 mil	£	> 100 mil	> 100 mil	£
Potenciação	> 310 mil	2615	4185	> 270 mil	2611	2611
Primo	> 99 mil	> 99 mil	£	> 99 mil	> 99 mil	£
Soma de vetor	> 9990	3060	4180	> 9990	2160	2610
Texto	>3710	3060	4180	>3070	2160	2610

Legendas:

> ##### Maior do que N chamadas recursivas ou iterações (onde ##### é o valor de N). Ou seja, nos casos em que há o sinal “>” na frente, o algoritmo foi experimentado até N chamadas recursivas ou iterações, mas poderia ir além.

£ Não houve versão NTR

6.8.3. Quanto à complexidade de tempo de execução dos algoritmos das funções recursivas experimentadas, na notação assintótica

Complexidade de tempo

Algoritmo	Versão original	Iterativo (IT)	Recursivo caudal (TR)	Recursivo não caudal (NTR)
FatMod	fatModNTR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Fatorial	fatNTR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Fibonacci	fibTR	$\Theta(n)$	$\Theta(n)$	\mathcal{E}
Inverte String	invNTR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
MDC	mdcTR	$O(\log n)$	$O(\log n)$	\mathcal{E}
Ordenação	ordNTR	$O(n^2)$	$O(n^2)$	$O(n^2)$
Palíndromo	palTR	$\Theta(n)$	$\Theta(n)$	\mathcal{E}
Potenciação	potNTR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Primo	primTR	$\Theta(n)$	$\Theta(n)$	\mathcal{E}
Soma de vetor	sumVecNTR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Texto	textNTR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Quadro 9 - Complexidade de tempo de execução dos algoritmos testados

CONCLUSÃO

Há uma grande variedade de problemas em que a abordagem recursiva é a mais natural para resolvê-los, como é o caso daqueles relacionados ao conceito de indução matemática.

A escrita recursiva, em geral, torna o código mais conciso e, portanto, mais legível. Porém, não raro, causa problemas para a execução de um número muito grande (N). Quando esse for o caso, é recomendável traduzir o código para suas versões iterativas.

A escrita recursiva pode ser de difícil compreensão para um programador que não domina o conceito da indução matemática. Contudo, seu estudo é necessário não somente em função da importância de seu uso, como também para que se possa compreender o código elaborado por outro programador. Para facilitar este estudo, o *RecPy* pode ser empregado.

Conforme se pode verificar no Quadro 6.8.3, na conversão entre os tipos de funções, não se verificou melhoria na complexidade de tempo de execução dos algoritmos das funções recursivas experimentadas, na notação assintótica. Não obstante, a maior vantagem na conversão de funções recursivas para o modo iterativo é a possibilidade de executar quantidades muito grandes de iterações sem travamento da aplicação por estouro da pilha. Além disso, as versões iterativas são executadas em menor tempo, mesmo que seja por uma constante (e não assintoticamente).

Os resultados experimentados em algoritmos conversíveis no pré-compilador *RecPy* são apresentados no Tópico 6.7.1. Ali se situam os gráficos de tempo x quantidade N de chamadas dos três tipos de funções (recursivas caudais, não caudais e iterativas), na versão Python 2.7.16.

A abrangência do reconhecimento de algoritmos e restrições sintáticas impostas na linguagem reconhecida pelo *RecPy* encontra-se no Tópico 5.5 (Requisitos, restrições, observações e recomendações no reconhecimento de funções). Esse tópico é indicado ao usuário do *RecPy*, como leitura prévia à sua experimentação, pois é nele que estão especificados os padrões de preparação para a tradução das funções.

O desenvolvimento do projeto do pré-compilador *RecPy* é descrito no tópico 5.6 .

Quanto às hipóteses levantadas (tópico 3.4), concluímos que:

- a) De fato, à luz dos experimentos realizados, ficou claro que, em alguns casos, os algoritmos recursivos provocam o estouro de pilha;

- b) Na conversão de funções recursivas para iterativas, ficou bem claro o aumento na escalabilidade da quantidade de iterações, nos casos em que as funções recursivas estavam apresentando limitações em suas versões recursivas. Em nenhum dos experimentos as versões iterativas apresentaram limitações;
- c) Na elaboração de experimentos, percebeu-se maior facilidade em se criar versões recursivas caudais (TR) do que recursivas não caudais (NTR), mas, em termos de otimização dos códigos, não se vislumbraram diferenças muito significativas nos algoritmos testados.
- d) O pré-compilador RecPy mostrou-se ferramenta bastante útil tanto para agilizar o procedimento de conversão de funções recursivas e iterativas quanto no estudo dessas funções. Portanto, cumpre seus objetivos de colaborar no ensino e aprendizado desse tema.

SUGESTÃO DE TRABALHOS FUTUROS

O presente trabalho apresentou técnicas para a construção do pré-compilador (*RecPy*), destinado à tradução de algoritmos recursivos caudais, recursivos não caudais e iterativos. Há oportunidade a várias extensões da pesquisa. Por exemplo, podem-se utilizar linguagens alvo diferentes de Python. Pode-se, também, ampliar o rol de funções recursivas e iterativas reconhecíveis. Algoritmos mais complexos, como os da tradicional representação da sequência de Fibonacci, ou os da busca recursiva binária, que envolvem mais de uma recursão em uma única função, requerem tratamento especial. Isso porque, quando se introduz uma recursão interna a mais, aumenta-se a complexidade dos procedimentos de elaboração do código do pré-compilador.

Quanto ao algoritmo de Fibonacci, apresentamos exemplos de funções reconhecíveis no *RecPy*: uma iterativa e outra recursiva caudal.

As traduções de funções recursivas ou iterativas que tenham ficado fora do escopo do atual podem ser abrangidas em futuros trabalhos.

Em razão da pouca utilidade prática, não se elaborou um módulo de conversão de funções iterativas para recursivas não caudais (IT-NTR) ou de recursivas caudais para recursivas não caudais (TR-NTR). Para fins exclusivamente teóricos, podem ser sugeridos esses acréscimos em futuras versões do *RecPy*.

ANEXOS

7. ANEXO I - BREVE HISTÓRICO DA RECURSÃO

7.1. Introdução

De acordo com Sobral (2015)^[B24], “o desenvolvimento de funções recursivas teve origem no século XIX, quando recursão foi primeiro usada como um método de definir funções aritméticas simples, até o meado dos anos 30 no século XX, quando a classe de funções recursivas gerais foi introduzida por Herbrand-Gödel, formalizada por Kleene e usada por Church no seu λ -Cálculo”.

Dentre os artigos colacionados, um autor destaca-se no estudo das teorias de computabilidade e de recursão: Robert I. Soare. O tema é visto em Soare (1996)^[37] [36] , Soare (1999)^[54] [53] , Soare (2007)^[87] [84] , Soare (2009)^[95] , Soare(2014)^[116] . Em “Uma breve revisão da Computabilidade”, o autor relata as contribuições de Hilbert no aprimoramento das teorias de conjunto de Georg Cantor. Hilbert foi quem formulou o clássico problema da decisão (*Entscheidungsproblem*), que consistiria em se encontrar um algoritmo para decidir se determinada declaração seria válida ou não. Em 1931, Gödel viria a refutar a consistência dessa formulação de Hilbert. Entre 1931 e 1934, Church e seu então aluno Kleene trabalharam o sistema formal das funções lâmbda (λ) definíveis. Em 1935, Church e Kleene se inclinaram ao formalismo das funções recursivas (de Gödel). Já em 1931, Gödel havia utilizado funções recursivas primitivas para computar um valor $f(n)$ utilizando-se de valores computados previamente.

7.2. λ -*Calculus* (cálculo lambda)

Steele e Sussman (1976)^[4] apresentam o lambda cálculo com o resumo de suas vantagens (semântica limpa, simples, minimalista, ideal para uso lógico) e desvantagens (desajeitada, primitiva). Dizem que foi desenhada para ser uma linguagem minimalista, sem a preocupação de ser “conveniente”.

Barendregt (1998)^[13] [12] apresenta o λ -Cálculo com uma breve introdução histórica que inclui a questão filosófico-teórica do Entscheidungsproblem (problema de decisão). O autor relata que Leibniz tinha como um ideal criar uma “linguagem universal” que encontrasse um método de solução de todos os problemas declaráveis nessa linguagem. Informa também que, em 1936, Turing provou que tanto seu modelo de máquinas de computação (atualmente chamadas de Turing-Machines) quanto o sistema formal de lambda-cálculo de Church seriam igualmente fortes no sentido de que definem a mesma classe de funções computáveis.

7.3. Tese de Church-Turing

Copeland e Shagrir (2018)^[147] [142] fornecem um dos textos mais diretos e esclarecedores sobre a tese de Church-Turing. Esse texto informa, também, que houve extensões da tese. Segundo eles, Turing utilizava a expressão “máquinas de computação lógica” para designar o que hoje conhecemos por “máquinas de Turing”. Tais máquinas poderiam fazer qualquer coisa que pudesse ser descrita como “puramente mecânica”. Church, em paralelo, concentrou seu trabalho sobre a noção de que as funções efetivamente calculáveis seriam as funções λ -definíveis, calcadas em números inteiros positivos. No que diz respeito especificamente à computação sobre números positivos inteiros, as teses de Church e de Turing seriam equivalentes.

Zach (2006)^[84] [81] é um dos autores que apresenta minuciosa revisão histórica das teorias relacionadas à recursão e à computabilidade. Segundo ele, em 1936 Gödel declarou que sua noção de computabilidade é absoluta, no sentido de que se um função é computável em um sistema de ordem-alta, é também computável em aritmética de primeira-ordem, ou seja, as funções recursivas em geral são todas computáveis em um sistema S contendo aritmética. E isso tem a ver com a noção de provas formais recursivamente enumeráveis.

Participante importante da evolução das teorias de funções recursivas, Stephen C. Kleene (1981)^[9] menciona que Turing, trabalhando independentemente, chegou a algumas das mesmas conclusões que ele e Church. A tese de Church-Kleene dizia que todas as funções para as quais houvesse algoritmos seriam λ -definíveis ou, equivalentemente à teoria de Herbrand-Gödel seriam geral-recursivas. Turing, por seu

lado, usou uma terceira noção equivalente de computação ao idealizar máquinas de computação para uma certa categoria (livre de erro e com memória ilimitada).

8. ANEXO II - Resumo da revisão bibliográfica

AHO et al. (1995)^[B1] escreveram um capítulo sobre otimização de código na construção de compiladores. Nesse capítulo, há um título que trata das principais fontes de otimização.

AHO et al. (2001)^[B2], assim como ATALLAH e BLANTON (2010)^[B3] escreveram sobre a eliminação da recursão caudal e sobre a completa eliminação da recursão caudal.

CORMEM et al. (2009)^[B5] apresentam a questão da profundidade de pilha. Ao final deste trecho, dão o exemplo de uma versão do algoritmo *quicksort* que simula recursão de cauda.

DASGUPTA et al. (2006)^[B6] dissertam sobre algoritmos de divisão e conquista, os quais são amplamente utilizados em problemas recursivos.

GRAHAM (1993)^[B10] escreveu um capítulo especialmente sobre recursão de cauda.

Na linguagem *Clojure*, as chamadas de função são consumidoras da pilha pois alocam quadros que acabam por esgotar o espaço da pilha. HALLOWAY (2009)^[B11] aborda a questão.

HOPCROFT et al. (2001)^[B13] relembram a necessidade de conjugação de várias técnicas para se alcançar eficiência e elegância. Dizem que a “recursão por si só não necessariamente conduz a algoritmos mais eficientes. Contudo, quando é combinada com outras técnicas como balanceamento, divisão e conquista e simplificação algébrica... produz ambos: eficiência e elegância.

MCMILLAN (2014)^[B15], ao tratar de programação dinâmica, menciona o quanto as soluções recursivas podem ser ineficientes, apesar de elegantes.

RABHI e LAPALME (1999)^[B18] cuidam da otimização de recursividade caudal.

SEBESTA (2012)^[B21] apresenta dois assuntos de grande interesse ao estudo: Suporte para programação funcional em linguagens prioritariamente imperativas e Comparação entre linguagens funcionais e imperativas.

SEDGEWICK e FLAJOLET (2013)^[B22] abordam a possibilidade de natural

mapeamento entre a forma recursiva ou iterativa de um programa e as respectivas relações de recorrência nas quais se possam representar.

SEGEWICK e WAYNE (2014)^[B23] apresentam o conceito e as características fundamentais da técnica recursiva, ressaltando a elegância, clareza e compactação normais desse método. Enfatizam duas qualidades importantes: 1) maior facilidade de entendimento em relação aos algoritmos não recursivos e 2) possibilidade do uso de induções matemáticas para comprovação de seu funcionamento correto.

SUMMERFIELD (2012)^[B25], menciona as vantagens da recursão de cauda na linguagem GO.

SZWARCFITER e MARKENZON (1994)^[B27] mencionam vantagens e desvantagens do procedimento recursivo. Além disso, lembram que, muitas vezes, há notória relação direta entre procedimentos recursivos e uma prova por indução matemática.

TOURETZKY (1990)^[B28] escreveu sobre as vantagens da recursão de cauda. Ele menciona que sempre que possível, é melhor escrever funções recursivas na forma de recursão caudal, pois os sistemas em Lisp podem executar funções recursivas mais eficientemente do que em recursões comuns.

VAREJÃO (2004)^[B29] apresenta o objetivo da programação funcional: definir uma função que retorne um valor como a resposta do problema. Um programa funcional é uma chamada de função que normalmente chama outras funções para gerar um valor de retorno. As principais operações nesse tipo de programação são a composição de funções e a chamada recursiva de funções. Outra característica importante é que funções são valores de primeira classe que podem ser passados para outras funções.

APÊNDICES

9. APÊNDICE I – Agradecimentos e menções (continuação)

Por não caber em uma única página, e para não deixar de mencionar, com gratidão, todos aqueles que contribuíram para o sucesso nesta caminhada, trago para este apêndice a continuação dos agradecimentos.

Gratificou-me conhecer a dedicação, o carinho e o profissionalismo com que a Professora Maria Alice Silveira de Brito trata de seus alunos e das disciplinas que ministra. Participei de sua turma de Compiladores, em 2019-1. Outro Professor que também mantém ótimo relacionamento com os alunos, em área correlata, é o Professor Luérbio Farias, com quem aprendi os primeiros fundamentos de Teoria da Computação, disciplina base para o aprendizado de compiladores.

As Professoras Rosa Maria E. M. da Costa e Vera Maria Werneck iniciaram-me no estudo de metodologias para pesquisas científicas. A Professora Rosa ministrou a matéria de Inteligência Artificial. A Professora Karla Tereza Figueiredo Leite apresentou os fundamentos de Mineração de Dados e de Redes Neuronais. Seguramente, darei continuidade ao estudo nessas promissoras e interessantes áreas de conhecimento. Às citadas Professoras, meu reconhecimento e gratidão por sua dedicação na transmissão desses conhecimentos.

Já gostava de Computação Gráfica. O Professor Gilson Alexandre Ostwald Pedro da Costa contribuiu para aumentar o meu gosto pela disciplina, pois a apresentou de um modo muito agradável e produtivo! Tive imenso prazer em realizar o trabalho de escorpiões em movimento, em 3D, com OpenGL e C++. Esse trabalho pode ser visto aqui:

<<https://www.youtube.com/watch?v=70XbQiKJZas>>.

Ao Professor Luciano Barreto agradeço pelos bons resultados no aprendizado de Redes e de Sistemas Distribuídos.

Aos Professores Eduardo Gonçalves Galúcio e Alexandre Solon Nery agradeço por aprofundarem meus conhecimentos em tema bastante complexo e estimulante para todos aqueles que, como eu, gostam de conhecer a fundo o funcionamento interno de computadores e seus recursos computacionais: Sistemas Operacionais. O mesmo posso dizer das disciplinas de Arquitetura de Computadores ministradas pelas dedicadas professoras Maria Clícia Stelling de Castro e Roseli Suzi Wedemann.

Igual reconhecimento é devido aos Professores Leandro Augusto Justen Marzullo, Alexandre Sztajnberg e Alexandre Rojas por ampliarem meus horizontes nas linguagens C/C++, Java e Python, respectivamente.

As Professoras Christina Fraga Esteves Maciel Waga, Ana Carolina Almeida e Nayat Sanchez Pi ministraram excelentes ensinamentos de Lógica, Banco de Dados e Interfaces Humano-Computador, respectivamente.

O Professor Antônio de Pádua Albuquerque Oliveira apresentou as bases necessárias à disciplina de Engenharia de Software. A importância dessa matéria é inquestionável! Ao desenvolvedor de sistemas é necessário conhecer metodologias para a confecção de softwares com maior eficiência e melhor desenho. Na sequência desse estudo, o extremamente dedicado Professor Marcelo Schots de Oliveira enfatizou a necessidade de se solidificar esse conhecimento. Acredito que os aprendizados nessa disciplina transcenderam a mera aplicação técnica e, por isso, ficarão guardados na memória com gratidão!

O Professor de Cálculo I, Laurent Marcos Prouvé, será lembrado não só por sua didática, mas também pelo carinho e dedicação. Fato marcante é sua boa memória para guardar o nome de seus alunos. Suas aulas propiciaram-me renovado gosto pela matéria e, por isso, ocorreram em momento extremamente oportuno.

Em Cálculo II, o Professor Raphael Constant da Costa foi exigente na medida certa e ofereceu boa didática.

O Professor Rubens André Sucupira, de Cálculo III, teve facilidade para explicar temas complexos e serenidade para conduzir as aulas. Aliás, inegavelmente, essa foi uma característica comum aos professores de Cálculo aqui mencionados.

André Luiz Furtado, Professor de Cálculo IV, ficou registrado em minha memória por sua excelente metodologia de ensino. Paciente, talentoso, bem humorado, muito atencioso. Conseguiu, com maestria, tornar menos complexos mesmo aqueles temas mais abstratos.

Em Otimização Combinatória, sobre a Professora Luiza Maria Oliveira da Silva, tenho a dizer que demonstrou excelente didática, atenção aos seus alunos, bom andamento da evolução da matéria. Suas aulas foram experiência bastante gratificante!

Tópicos Especiais II (Inteligência Computacional e Artificial): ao Professor Igor Machado Coelho, a gratidão por sua dedicação, bom relacionamento com alunos, conhecimento profundo da matéria, bom humor e simplicidade na exposição de aulas.

Física I: Professores Márcio André Lopes Capri e Letícia Faria Domingues Palhares: Suas didáticas fizeram com que tivesse bom aproveitamento.

Física II: Emílio Jorge Lydia e Ada Petrolina Lopez Gimenez: o mesmo que disse acima pode ser repetido aqui. Acrescento, em relação ao professor Emílio, que sua empolgante dinâmica de ensino proporcionou ainda maior satisfação no acompanhamento da matéria.

O Professor de Cálculo Numérico, Paulo Vinícius Santos, assim como outros citados, é um jovem talentoso. Conseguiu tornar agradável uma matéria que envolve cálculos bastante trabalhosos. Muito bom relacionamento com a turma.

Em Matemática Discreta, o Professor Augusto Cesar de Castro Barbosa foi exigente na medida certa. Iniciou-me em temas muito importantes para este trabalho, em especial, no aprendizado de Indução Matemática.

Ainda em Matemática Discreta, tive o prazer de conhecer o Professor Américo Barbosa da Cunha Junior. Excelente didática, profundo conhecimento da matéria, além de elevadíssimo grau de profissionalismo e de comprometimento com a turma. Fato marcante é que mesmo com turma reduzida manteve assiduidade e pontualidade exemplares.

O Professor Claudio Plinio Campana Chacca foi quem primeiro me apresentou a beleza e a utilidade da Álgebra Modular.

O Professor de Álgebra e de Álgebra Linear, Sajad Salami foi sereno e atencioso. Os conhecimentos adquiridos nessas matérias foram muito úteis em várias outras disciplinas.

Geometria Analítica: Professora Paula de Oliveira Ribeiro: no primeiro período, essa foi a disciplina na qual tive melhor aproveitamento. A didática e objetividade da Professora Paula favoreceram esse bom desempenho.

Português Instrumental: Professora Cláudia Moura da Rocha: aulas muito descontraídas e proveitosas, em matéria fundamental em qualquer ramo de conhecimento e que aprendo a gostar cada vez mais que estudo.

Cálculo das Probabilidades: Professora Maria Luiza Viana Lisboa: grato pela simpatia e boa condução da matéria.

Ética, Computadores e Sociedade: Professor Antônio Carlos de Azevedo Ritto: lembrei pelo estímulo ao debate dialético, pelos temas que trouxe à discussão e sobretudo por valorizar e respeitar a diversidade de ideias e de opiniões, atributo marcante em sociedades evoluídas.

Empreendedorismo: Professora Marinilza Bruno de Carvalho: agradecido pela contagiante energia e alegria de suas aulas, assim como pelos bons valores passados a seus alunos.

Também ficam na memória as boas conversas que tive com o culto e sempre simpático Professor Geraldo Magela da Silva, que, apesar de não ter sido meu professor, manteve, ao que percebi, presença constante e atenta na direção do Instituto de Matemática e Estatística.

Com trabalho sério e dedicado, os (as) Professores (as) acima lembradas(os), e outros(as) tantos(as) que não cheguei a conhecer bem, mas que tenham as mesmas características, valorizam o diploma e o curso de Ciências da Computação da Universidade Estadual do Rio de Janeiro. Recebam, portanto, essas singelas palavras como sinal de gratidão e homenagem!

Não podiam faltar palavras de afeto e amizade aos (às) colegas das várias turmas que integrei. Apesar de ter sido frequentemente o mais velho da turma, sentia-me em geral tão à vontade entre eles(as) que quase nunca lembraava de que a média de idade era aproximada à de minha filha mais velha, Mariana, também aluna da UERJ. Quero levar adiante a amizade com muitos desses(as) colegas.

Agradeço, também, aos chefes e colegas de trabalho pelo incentivo que me dão, mesmo que não saibam, sempre que me acionam na resolução de alguma questão vinculada à área de Tecnologia da Informação. A mesma alegria ocorre quando estou incumbido de desenvolver alguma solução computacional, por menor que seja, sobretudo quando inovadora.

Por fim, agradeço às pessoas que compartilham publicamente, e com livre acesso na Internet, seus conhecimentos.

Seguramente, não vou me lembrar de todos os indivíduos, instituições ou grupos que merecem terem seus trabalhos bastante divulgados, pela sua imensa utilidade aos estudantes. Além de muitos outros não mencionados, foram muito úteis as aulas ou conteúdos de reforço:

1) Dos vídeos do canal da Univesp no Youtube, em especial:

a) as séries de Cálculo com o Professor Cláudio Possani, a exemplo de:

- <<https://www.youtube.com/playlist?list=PLxI8Can9yAHdCutIIIKca1wrkuRLvBhHs>>
- <<https://www.youtube.com/watch?v=zZobguo-YX4>>
- <<https://www.youtube.com/playlist?list=PLxI8Can9yAHdSstaijzbnJp405wWmRLnD>>
- <<https://www.youtube.com/playlist?list=PLxI8Can9yAHeOiMYCBlkYCALLoROQ58OY>>

b) as aulas de Redes e de Sistemas Distribuídos do Professor Jó Ueyama, a exemplo de:

<https://www.youtube.com/watch?v=1csTmCZj-jo&list=PLPp3g7TxayKxEiA8v1U_CEvVAk1zlg5JA>

2) As aulas do Professor Grings. Exemplo:

- <https://www.youtube.com/playlist?list=PLhlx0uNVbDQjVBANr vuGfpPJ7Ub f_Ycz3>

3) As aulas do curso de Licenciatura em Matemática da Universidade Estadual da Bahia (UNEBA) - EaD:

a) Professora Odete Amanda, Álgebra (exemplo):

- <<https://www.youtube.com/watch?v=ut820P2Wdow>>

b) Vídeos de Álgebra Linear da Professora Rosely Ouais Pestana Bervian:

- <<https://www.youtube.com/watch?v=ufQasCq9LiM>>

4) Os vídeos do Canal Me Salva no Youtube:

- <<https://www.youtube.com/channel/UCWv7JMNjrWIVtkiBmygefHQ>>

5) As aulas do Professor Fabio Henrique Teixeira de Souza, do Programa de Iniciação Científica da OBMEP:

- <<https://www.youtube.com/playlist?list=PLrVGp617x0hAttp3LQVBBF2td1uHjf hbr>>
- <<https://www.youtube.com/watch?v=p8FhAF6Qltg&list=PLrVGp617x0hAttp3LQVBBF2td1uHjf hbr&index=11>> (aula sobre resolução de recorrência)

REFERÊNCIAS

10. Mecanismos de busca e páginas de consulta utilizados para a elaboração deste TCC

Além do próprio Google, que foi utilizado para se encontrar a maioria dos demais, estes foram os portais em que foram encontrados os artigos de referência:

- [S1] Google: <www.google.com>
- [S2] <<https://www.semanticscholar.org/>>
- [S3] <<https://pdfs.semanticscholar.org/>>
- [S4] <<https://www.sciencedirect.com/>>
- [S6] <<http://www.periodicos.capes.gov.br/>>
- [S7] <<https://www.cs.bu.edu/>>
- [S8] <<https://arxiv.org/>>
- [S9] <<https://academic.oup.com/>>
- [S10] <<http://repositorio.unicamp.br/>>
- [S11] <<https://pantheon.ufrj.br/>>
- [S12] <http://catalogo-redesirius.uerj.br/sophia_web/>
- [S13] <<http://www.rsirius.uerj.br/novo/index.php/livros-eletronicos/bases-e-periodicos/bases-de-dados/>>
- [S14] <<https://search.scielo.org/?lang=pt>>
- [S15] <<https://www.researchgate.net/>>
- [S16] <<https://books.google.com.br/>>
- [S17] <<https://github.com/emscripten-core/emscripten/>>
- [S18] <<https://www.cs.ubc.ca/>>
- [S19] <<https://www.ncbi.nlm.nih.gov/pmc/>>
- [S20] <<https://stackoverflow.com/>>
- [S21] <<https://softwareengineering.stackexchange.com/>>
- [S22] <<https://news.ycombinator.com/>>
- [S23] <<http://citeseerx.ist.psu.edu/>>
- [S24] <<https://cseweb.ucsd.edu/>>
- [S25] <<http://www.cs.uwyo.edu/>>
- [S26] <<https://www.ida.liu.se/>>
- [S27] <<http://ssw.jku.at/>>
- [S28] <<https://www.lua.org/pil/6.3.html>>
- [S29] <<https://pdf.sciencedirectassets.com/>>

- [S30] <<https://cs.wellesley.edu/>>
- [S31] <<https://developer.ibm.com/>>
- [S32] <<https://nostarch.com/>>
- [S33] <<https://www.cs.cmu.edu/>>
- [S34] <<https://eprints.kingston.ac.uk/>>
- [S35] <<https://dblp.org/>>
- [S36] <<https://marmelab.com/>>
- [S37] <<https://eli.thegreenplace.net/>>
- [S38] <<https://www.uraimo.com/>>
- [S39] <<https://jacob.jkrall.net/>>
- [S40] <<https://www.cs.indiana.edu/>>
- [S41] <<https://cseweb.ucsd.edu/>>
- [S42] <<https://medium.com/>>
- [S43] <<https://rosettacode.org/>>
- [S44] <<http://www.danzig.us/>>
- [S45] <<https://ocaml.org/>>
- [S46] <<http://functorial.com/>>
- [S47] <<http://glat.info/>>
- [S48] <<http://www.lazutkin.com/>>
- [S49] <<https://news.ycombinator.com/>>
- [S50] <<https://www.antlr.org/download.html>> <acesso em 12/09/2020>
- [S51] <<https://dspace.mit.edu/>>

De 1 a 49: acessos realizados em 07/07/2019.

11. Leituras recomendadas na Internet

[L1] <<https://win-vector.com/2019/08/22/eliminating-tail-calls-in-python-using-exceptions/>> (acesso em 15/09/2020)

[L2] <<https://www.kylem.net/programming/tailcall.html>> (acesso em 15/09/2020)

[L3] <<http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html>> (acesso em 15/09/2020)

[L4] <<https://ics.upjs.sk/~novotnryr/blog/1970/recursion-tail-call-optimizations-and-trampolines>> (acesso em 15/09/2020)

[L5] MOERTEL, Tom. Recursion to Iteration Series. 2013. Disponível em:

- [L5a]<<http://blog.moertel.com/tags/recursion-to-iteration%20series.html>>
- [L5b]Tricks of the trade: Recursion to Iteration, Part 1: The Simple Method, secret features, and accumulators.2013. Disponível em:
<<http://blog.moertel.com/posts/2013-05-11-recursive-to-iterative.html>>
- [L5c]Tricks of the trade: Recursion to Iteration, Part 2: Eliminating Recursion with the Time-Traveling Secret Feature Trick.2013. Disponível em:
<<http://blog.moertel.com/posts/2013-05-14-recursive-to-iterative-2.html>>
- [L5d]Tricks of the trade: Recursion to Iteration, Part 3: Recursive Data Structures. 2013. Disponível em: <<http://blog.moertel.com/posts/2013-06-03-recursion-to-iteration-3.html>>
- [L5e]Tricks of the trade: Recursion to Iteration, Part 4: The Trampoline. 2013. Disponível em: <<http://blog.moertel.com/posts/2013-06-12-recursion-to-iteration-4-trampolines.html>>. Acesso em 07/07/2019.

[L6] <<https://vanemden.wordpress.com/2014/06/18/how-recursion-got-into-programming-a-comedy-of-errors-3/>> . Acesso em 16/09/2020

[L7] <<https://plato.stanford.edu/entries/recursive-functions>> Acesso em 16/09/2020.

[L8] <<https://www.britannica.com/topic/history-of-logic/Theory-of-recursive-functions-and-computability>>. Acesso em 16/09/2020.

[L9] <<http://lambda-the-ultimate.org/node/1014>> Acesso em 20/09/2020

[L10] <<https://www.codeproject.com/Articles/418776/How-to-replace-recursive-functions-using-stack-and>> Acesso em 20/09/2020

[L11] <<https://fabianooliveira.ime.uerj.br/lecture-notes>> Acesso inicial em 07/07/2019.

[L12] <<https://mitpress.mit.edu/sites/default/files/sicp/full-text/sicp/book/node85.html>> Acesso em 20/09/2020

[L13] <<https://mitpress.mit.edu/sites/default/files/sicp/full-text/sicp/book/node85.html>> Acesso em 20/09/2020

[L14] <<http://matt.might.net/articles/by-example-continuation-passing-style/>> acesso em 28/09/2020.

12. Índice de artigos científicos de referência

Observação: formatação automática, no padrão ABNT, obtida no aplicativo Mendeley.

- [1] **McCarthy (1960)** John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*. 3, 4 (1960), 184–195. doi: 10.1145/367177.367199.
- [2] **Landin (1964)** P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*. 6, 4 (jan.-1964), 308–320. doi: 10.1093/comjnl/6.4.308.
- [3] **Sussman e Steele (1975)** Gerald Jay Sussman e Guy Lewis Jr. Steele. Scheme: A Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation*. doi: 10.1023/A:1010035624696.
- [4] **Steele e Sussman (1976)** Guy Lewis Jr. Steele e Gerald Jay Sussman. LAMBDA: The ultimate declarative. *Memo 379, MIT AI Lab*. 353 (1976), AIM-353. Recuperado 20-jul.-2019, de <<http://dspace.mit.edu/handle/1721.1/6091>>.
- [5] **Sussman e Steele (1978)** Gerald Jay Sussman e Guy Lewis Jr. Steele. The Art of the Interpreter of the Modularity Complex (Parts Zero, One, and Two). *Aim453*. Recuperado de <<http://dspace.mit.edu/handle/1721.1/6094>>.
- [6] **Sickel (1978)** Sharon Sickel. Removing Redundant Recursion. [*1*] *California Univ Santa Cruz Information Sciences*. (1978).
- [7] **Sussman e Steele (1980)** Guy Lewis Jr. Steele e Gerald Jay Sussman. The dream of a lifetime: A lazy variable extent mechanism. *Proceedings of the 1980 ACM Conference on LISP and Functional Programming, LFP 1980*. (1980), 163–172. doi: 10.1145/800087.802802.
- [8] **Bird, R. S. (1980)**. Tabulation Techniques for Recursive Programs. *ACM Computing Surveys (CSUR)*, 12(4), 403–417. <https://doi.org/10.1145/356827.356831>
- [9] **Kleene (1981)** Stephen C. Kleene. The theory of recursive functions, approaching its centennial. *Bulletin of the American Mathematical Society*. 5, 1 (1981), 43–62. doi: 10.1090/S0273-0979-1981-14920-X.
- [10] **Brooks et al. (1982)** Rodney A. Brooks et al. Lisp-in-Lisp : High Performance and Portability. *Symposium A Quarterly Journal In Modern Foreign Literatures*. 2, (1982), 845–849.
- [11] **Rosser (1982)** J.Barkley Rosser. Highlights of the History of the Lambda Calculus. *Foreign Affairs*. (1982).
- [12] **Kranz et al. (1984)** David Kranz et al. ORBIT: An optimizing compiler for scheme. *Orbit An International Journal On Orbital Disorders And Facial Reconstructive Surgery*. (1984), 219–233. doi: 10.1145/12276.13333.

- [13] **Barendregt (1984)** H. Barendregt. Introduction to lambda calculus. 372. December 1998 (1984).
- [14] **Cardelli (1984)** Luca Cardelli. Compiling a Functional Language. *Proceedings of the 1984 ACM Symposium on LISP and functional programming - LFP '84*. (1984), 208–217. doi: 10.1145/800055.802037.
- [15] **Holloway (1986)** C.Michael Holloway. A Survey of Functional Programming Language Principles. September 1986 (1986). Recuperado de <https://ntrs.nasa.gov/search.jsp?R=19870002073>.
- [16] **Felleisen (1987)** MATTHIAS Mattias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*.
- [17] **Feeley e Lapalme (1987)** Marc Feeley e Guy Lapalme. *Using closures for code generation*. doi: 10.1016/0096-0551(87)90012-9.
- [18] **Dybvig (1987)** R.Kent Dybvig. Three Implementation Models for Scheme. (1987).
- [19] **Appel e Jim (1989)** Andrew W. Appel e Trevor Jim. Continuation-passing, closure-passing style. *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89* (New York, New York, USA, 1989), 293–302. doi: 10.1145/75277.75303.
- [20] **Burke (1990)** Michael Burke. An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 12, 3 (1990), 341–395. doi: 10.1145/78969.78963.
- [21] **Boiten (1991)** Eerke Boiten. The many disguises of accumulation. 518, December (1991), 1–17.
- [22] **Baker (1992)** Henry G. Baker. CONS Should not CONS its Arguments, or, a Lazy Alloc is a Smart Alloc. *ACM SIGPLAN Notices*. 27, 3 (mar.-1992), 24–34. doi: 10.1145/130854.130858.
- [23] **Jones (1992)** Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*. 2, 1 (1992), 73–79. doi: 10.1017/S0956796800000277.
- [24] **Geuvers (1992)** J. H. Geuvers. Inductive and co-inductive types with iteration and recursion in a polymorphic framework. *Informal Proceedings Workshop on Types for Proofs and Programs, Båstad, Sweden, 8–12 June 1992*. May 1991 (1992).
- [25] **Tarditi et al. (1992)** David Tarditi et al. No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)*. 1, 2 (1992), 161–177. doi: 10.1145/151333.151343.
- [26] **Peyton & Wadler (1992)** Simon L. Peyton Jones; Philip Wadler. *Imperative functional programming*.

- [27] **Barry Jay (1993)** C. Barry Jay. Tail recursion through universal invariants. *Theoretical Computer Science*. 115, 1 (1993), 151–189. doi: 10.1016/0304-3975(93)90059-3.
- [28] **Kelsey (1993)** Richard A. Kelsey. Tail-Recursive Stack Disciplines for an Interpreter. March (1993), 1–13.
- [29] **Riecke (1993)** Jon G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*. 3, 4 (1993), 387–415. doi: 10.1017/S0960129500000293.
- [30] **Bacon et al. (1994)** David F. Bacon et al. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys (CSUR)*. 26, 4 (1994), 345–420. doi: 10.1145/197405.197406.
- [31] **Clinger e Hansen (1994)** William D. Clinger e Lars Thomas Hansen. *Lambda, the ultimate label or a simple optimizing compiler for scheme*. doi: 10.1145/182590.156786.
- [32] **Danvy (1994)** Olivier Danvy. Back to direct style. *Science of Computer Programming*. 22, 3 (1994), 183–195. doi: 10.1016/0167-6423(94)00003-4.
- [33] **Alexander e Emberson (1995)** Mark Alexander e Hugh Emberson. a Dynamic Compiler for Scheme. (1995).
- [34] **Ramakrishnan e Ullman (1995)** Raghu Ramakrishnan e Jeffrey D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*. 23, 2 (1995), 125–149. doi: 10.1016/0743-1066(94)00039-9.
- [35] **Friedman e Sheard (1995)** Harvey Friedman e Michael Sheard. Elementary descent recursion and proof theory. *Annals of Pure and Applied Logic*. 71, 1 (1995), 1–45. doi: 10.1016/0168-0072(94)00003-L.
- [36] **Ichikawa (1995)** Yoshihiko Ichikawa. Database States in Lazy Functional Programming Languages: Imperative Update and Lazy Retrieval. (1995), 1–17. doi: 10.14236/ewic/dbpl1995.14.
- [37] **Soare (1996)** Robert I. Soare. Computability and Recursion. *Bulletin of Symbolic Logic*. 2, 3 (1996), 284–321. doi: 10.2307/420992.
- [38] **Howard (1996)** Brian T. Howard. Inductive, coinductive, and pointed types. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*. 31, 6 (1996), 102–109. doi: 10.1145/232629.232640.
- [39] **William D. Clinger et al. (1996)** William D. Clinger et al. Implementation Strategies for First-Class Continuations. *International Journal of Operations & Production Management*. 45, (1996), 7–45.
- [40] **Okasaki (1996)** Chris Okasaki. Functional data structures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 1129, September (1996), 131–158. doi: 10.1007/3-540-61628-4_5.

- [41] **Ross (1996)** Kenneth A. Ross. Tail Recursion Elimination in Deductive Databases. *ACM Transactions on Database Systems*. 21, 2 (1996), 208–237. doi: 10.1145/232616.232628.
- [42] **Appel e Shao (1996)** Andrew W. Appel e Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*. 6, 1 (1996), 47–74. doi: 10.1017/S095679680000157X.
- [43] **Fitzpatrick et al. (1996)** Stephen Fitzpatrick et al. Unfolding Recursive Function Definitions Using the Paradoxical Combinator. April (1996), 1–2.
- [44] **Steele e Gabriel (1996)** Guy Lewis Jr. Steele e Richard P. Gabriel. The evolution of Lisp. *History of programming languages--II*. (1996), 233–330. doi: 10.1145/234286.1057818.
- [45] **Reddy (1996)** U. S. Reddy. Imperative functional programming. *ACM Computing Surveys*. 28, 2 (1996), 312–314. doi: 10.1145/234528.234736.
- [46] **Gupta e Pontelli (1996)** Gopal Gupta e Enrico Pontelli. Last Alternative Optimization. *IEEE Symposium on Parallel and Distributed Processing - Proceedings*. (1996), 538–541. doi: 10.1109/spdp.1996.570380.
- [47] **Feeley et al. (1997)** Marc Feeley et al. Compiling Higher-Order Languages into Fully Tail-Recursive Portable C. *Work*. (1997), 1–12. Recuperado de <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.8788>.
- [48] **Danvy e Schultz (1997)** Olivier Danvy e Ulrik P. Schultz. Lambda-Dropping: Transforming Recursive Equations into Programs with Block Structure. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*. 32, 12 (1997), 90–106. doi: 10.1145/258994.259007.
- [49] **Clinger (1998)** William D. Clinger. Proper Tail Recursion and Space Efficiency. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*. 33, 5 (1998), 174–185. doi: 10.1145/277652.277719.
- [50] **Cooper et al. (1998)** Keith D. Cooper et al. Compiler-Based Code-Improvement Techniques. (1998), 1–65.
- [51] **Peyton Jones e Santos (1998)** Simon L. Peyton Jones e André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*. 32, 1–3 (1998), 3–47. doi: 10.1016/s0167-6423(97)00029-4.
- [52] **Ward et al. (1999)** Martin P. Ward et al. Recursion removal/introduction by formal transformation: an aid to program development and program comprehension. *Computer Journal*. 42, 8 (1999), 650–673. doi: 10.1093/comjnl/42.8.650.
- [53] **Muth e Debray (1999)** Robert Muth e Saumya Debray. Partial Inlining. 1974, May 1974 (1999), 4–18.

- [54] **Soare (1999)** Robert I. Soare. The history and concept of computability. *Studies in Logic and the Foundations of Mathematics*. 140, C (1999), 3–36. doi: 10.1016/S0049-237X(99)80017-2.
- [55] **Ramsdell (1999)** John D. Ramsdell. Tail-recursive SECD machine. *Journal of Automated Reasoning*. 23, 1 (1999), 43–62. doi: 10.1023/A:1006151910336.
- [56] **Johansson e Nystrom (1999)** Erik Johansson e Sven-olof Nystrom. *HiPE : High Performance Erlang HiPE : High Performance Erlang*.
- [57] **Liu e Stoller (1999)** Yanhong A. Liu e Scott D. Stoller. From recursion to iteration. *ACM SIGPLAN Notices*. 34, 11 (1999), 73–82. doi: 10.1145/328691.328700.
- [58] **Yi, Q., Adve, V., & Kennedy, K.** (2000). Transforming loops to recursion for multi-level memory hierarchies. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 35(5), 169–181. <https://doi.org/10.1145/358438.349323>
- [59] **Nenzén, P., & Ragard, A.** (2000). *Tail Call Elimination in GCC*.
- [60] **Schinz e Odersky (2001)** Michel Schinz e Martin Odersky. Tail call elimination on the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*. 59, 1 (2001), 158–171. doi: 10.1016/S1571-0661(05)80459-1.
- [61] **Bailey e Weston (2001)** Mark W. Bailey e Nathan C. Weston. Performance Benefits of Tail Recursion Removal in Procedural Languages. June (2001). Recuperado 6-jul.-2019, de <https://www.semanticscholar.org/paper/Performance-Benefits-of-Tail-Recursion-Removal-in-Bailey-Weston/bd68df8bcae9f1d8cb185b6be0fbcc6fd58c941f>.
- [62] **Probst (2001)** Mark Probst. Proper Tail Recursion in C. (2001), 60.
- [63] **Walton (2001)** Christopher David Walton. Abstract Machines for Dynamic. *Communication*. (2001).
- [64] **Rhiger (2001)** Morten Rhiger. Higher-Order Program Generation. *Science*. August (2001).
- [65] **Biraben (2001)** Rodolfo Biraben. Questões conceituais de computabilidade.
- [66] **Jeannin et al. (2001)** Jean-Baptiste Jeannin et al. CoCaml: Functional Programming with Regular Coinductive Types. *Fundamenta Informaticae*. 2001 (2001), 1001–1028. doi: 10.3233/FI-2012-0000.
- [67] **Stenman (2002)** Erik Stenman. *Efficient implementation of concurrent programming languages*. Recuperado de <http://uu.diva-portal.org/smash/get/diva2:162047/FULLTEXT01>.
- [68] **Kapur e Sakhanenko (2003)** Deepak Kapur e Nikita A. Sakhanenko. Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial*

Intelligence and Lecture Notes in Bioinformatics). 2758, August (2003), 136–154. doi: 10.1007/10930755_9.

- [69] **Clements e Felleisen (2003)** John Clements e Matthias Felleisen. A Tail-Recursive Semantics for Stack Inspections. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2618, (2003), 22–37. doi: 10.1007/3-540-36575-3_3.
- [70] **Minamide (2003)** Yasuhiko Minamide. Selective Tail Call Elimination. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2694, (2003), 153–170. doi: 10.1007/3-540-44898-5_9.
- [71] **Baldwin (2003)** Doug Baldwin. *A compiler for teaching about compilers*. doi: 10.1145/792548.611974.
- [72] **Himpe et al. (2003)** Stefaan Himpe et al. Control flow analysis for recursion removal. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 101–116. doi: 10.1007/978-3-540-39920-9_8.
- [73] **Cunha (2003)** Alcino Cunha. Automatic visualization of recursion trees: A case study on generic programming. *Electronic Notes in Theoretical Computer Science* (2003), 70–84. doi: 10.1016/S1571-0661(04)80694-7.
- [74] **Wagle e Cowan (2003)** Perry Wagle e Crispin Cowan. StackGuard: Simple Stack Smash Protection for GCC. *GCC Developers Summit*. (2003), 243–256.
- [75] **Fournet e Gordon (2003)** Cédric Fournet e Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. 25, 3 (2003), 360–399. doi: 10.1145/641909.641912.
- [76] **Ager et al. (2003)** Mads Sig Ager et al. A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects. *BRICS Report Series*. 11, 28 (dez.-2004), 149–172. doi: 10.7146/brics.v11i28.21853.
- [77] **Clements e Felleisen (2004)** John Clements e Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*. 26, 6 (2004), 1029–1052. doi: 10.1145/1034774.1034778.
- [78] **Mycka e Costa (2004)** Jerzy Mycka e José Félix Costa. Real recursive functions and their heirarchy. *Journal of Complexity*. 20, 6 (2004), 835–857. doi: 10.1016/j.jco.2004.06.001.
- [79] **Flanagan et al. (2004)** Cormac Flanagan et al. *The essence of compiling with continuations*. doi: 10.1145/989393.989443.
- [80] **Cowles e Gamboa (2004)** John Cowles e Ruben Gamboa. Contributions to the Theory of Tail Recursive Functions. (2004), 1–17.

- [81] **Cardone e Hindley (2006)** Felice Cardone e J.Roger Hindley. History of Lambda-calculus and Combinatory Logic. 123, 17 (2006), 2315–2319.
- [82] **Sieg (2006)** Wilfried Sieg. Gödel on computability. *Philosophia Mathematica*. 14, 2 (2006), 189–207. doi: 10.1093/philmat/nkj005.
- [83] **Tang (2006)** Peiyi Tang. Complete inlining of recursive calls: Beyond tail-recursion elimination. *Proceedings of the Annual Southeast Conference*. 2006, (2006), 579–584. doi: 10.1145/1185448.1185574.
- [84] **Zach (2006)** Richard Zach. Kurt Gödel and Computability Theory. (2006), 575–583.
- [85] **Shagrir (2006)** Oron Shagrir. Gödel on Turing on Computability. (2006). Ontos Verlag.
- [86] **Gasarch (2007)** William Gasarch. A Survey of Recursive Combinatorics. (2007), 1–131.
- [87] **Soare (2007)** Robert I. Soare. Computability and incomputability. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 4497 LNCS, (2007), 705–715. doi: 10.1007/978-3-540-73001-9_75.
- [88] **Rescorla (2007)** Michael Rescorla. Church’s thesis and the conceptual analysis of computability. *Notre Dame Journal of Formal Logic*. 48, 2 (2007), 253–280. doi: 10.1305/ndjfl/1179323267.
- [89] **Gao, Y., & Guan, F. (2008)**. Explore a new way to convert a recursion algorithm into a non-recursion algorithm. *IFIP International Federation for Information Processing*, 258, 187–193. https://doi.org/10.1007/978-0-387-77251-6_21
- [90] **Rubio-Sánchez et al. (2008)** Manuel Rubio-Sánchez et al. *A gentle introduction to mutual recursion*. doi: 10.1145/1597849.1384334.
- [91] **Zeugmann e Zilles (2008)** Thomas Zeugmann e Sandra Zilles. Learning recursive functions: A survey. *Theoretical Computer Science*. 397, 1–3 (2008), 4–56. doi: 10.1016/j.tcs.2008.02.021.
- [92] **Skliarova e Sklyarov (2009)** Ioulia Skliarova e Valery Sklyarov. Recursion in reconfigurable computing: A survey of implementation approaches. *FPL 09: 19th International Conference on Field Programmable Logic and Applications*. August 2009 (2009), 224–229. doi: 10.1109/FPL.2009.5272304.
- [93] **Reichenbach et al. (2009)** Christoph Reichenbach et al. Program metamorphosis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 5653 LNCS, (2009), 394–418. doi: 10.1007/978-3-642-03013-0_18.
- [94] **Schwaighofer (2009)** Arnold Schwaighofer. Tail Call Optimization in the Java HotSpot VM. (2009), 122. Recuperado de <http://ssw.jku.at/Research/Papers/Schwaighofer09Master/>.

- [95] **Soare (2009)** Robert I. Soare. Turing oracle machines, online computing, and three displacements in computability theory. *Annals of Pure and Applied Logic*. 160, 3 (2009), 368–399. doi: 10.1016/j.apal.2009.01.008.
- [96] **Rubio-Sánchez e Velázquez-Iturbide (2009)** Manuel Rubio-Sánchez e J.Ángel Velázquez-Iturbide. *Tail recursion by using function generalization*. ACM Press. doi: 10.1145/1595496.1563036.
- [97] **Rubio-Sánchez (2010)** Manuel Rubio-Sánchez et al. Tail recursive programming by applying generalization. *ITiCSE'10 - Proceedings of the 2010 ACM SIGCSE Annual Conference on Innovation and Technology in Computer Science Education* (2010), 98–102. doi: 10.1145/1822090.1822119.
- [98] **Daylight (2010)** Edgar G. Daylight. The Advent of Recursion in Programming, 1950s-1960s. *Computability in Europe (Programs, Proofs, Processes)*. JANUARY 2007 (2010).
- [99] **Bakoev (2010)** Valentin P. Bakoev. *The Recurrence relations in teaching students of Informatics*. doi: 10.15388/infedu.2010.10.
- [100] **Barendregt et al. (2011)** Henk Barendregt et al. The Imperative and Functional Programming Paradigm. (2011), 1–8.
- [101] **Myreen (2011)** M. Myreen. Formal verification of machine-code programs. *PhD Thesis*. December (2011). Recuperado de papers://cff96cb1-96b7-4b11-a3e3-f4947c1d45b9/Paper/p6901.
- [102] **Meister (2011)** Jeff Meister. OCaml : Tail Recursion. (2011), 1–3.
- [103] **Tang (2012)** Xiaolong Tang. *Lifting the Abstraction Level of Compiler Transformations*.
- [104] **Martins (2012)** Maurício Dias Martins. Distinctive signatures of recursion. *Philosophical Transactions of the Royal Society B: Biological Sciences*. 367, 1598 (2012), 2055–2064. doi: 10.1098/rstb.2012.0097.
- [105] **Thivierge e Feeley (2012)** Eric Thivierge e Marc Feeley. Efficient compilation of tail calls and continuations to JavaScript. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*. (2012), 47–57. doi: 10.1145/2661103.2661108.
- [106] **Bjarnason (2012)** Rúnar Oli Bjarnason. Stackless Scala With Free Monads. *The Third Scala Workshop*. (2012). Recuperado de <http://blog.higher-order.com/assets/trampolines.pdf>.
- [107] **Sonnex, W., Drossopoulou, S., & Eisenbach, S.** (2012). Zeno: An automated prover for properties of recursive data structures. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7214 LNCS(March), 407–421. https://doi.org/10.1007/978-3-642-28756-5_28

- [108] **Kneuss et al. (2013)** Etienne Kneuss et al. Synthesis modulo recursive functions. *ACM SIGPLAN Notices*. 48, 10 (2013), 407–426. doi: 10.1145/2544173.2509555.
- [109] **Bergstrom et al. (2013)** Lars Bergstrom et al. *Practical Inlining of Functions with Free Variables*. Recuperado de <http://arxiv.org/abs/1306.1919>.
- [110] **Lima (2013)** Ewerton Daniel de Lima. Soluções para o Problema da Seleção de Otimizações. (2013), 2615–2622.
- [111] **Nishida e Vidal (2014)** Naoki Nishida e Germán Vidal. Conversion to tail recursion in term rewriting. *Journal of Logic and Algebraic Programming*. 83, 1 (2014), 53–63. doi: 10.1016/j.jlap.2013.07.001.
- [112] **Watumull et al. (2014)** Jeffrey Watumull et al. On recursion. *Frontiers in Psychology*. 4, JAN (2014), 1–7. doi: 10.3389/fpsyg.2013.01017.
- [113] **Rinderknecht (2014)** Christian Rinderknecht. A Survey on teaching and learning recursive programming. *Informatics in Education*. 13, 1 (2014), 87–119.
- [114] **Insa e Silva (2014)** David Insa e Josep Silva. Transforming while/do/for/foreach-Loops into Recursive Methods. (out.-2014). Recuperado de <http://arxiv.org/abs/1410.4956>.
- [115] **Frame e Coffey (2014)** Scott Frame e John W. Coffey. *A Comparison of Functional and Imperative Programming Techniques for Mathematical Software Development*.
- [116] **Soare (2014)** Robert Irving Soare. Turing and the discovery of computability. *Turing's Legacy*. (2014), 467–492. doi: 10.1017/cbo9781107338579.014.
- [117] **William (2015)** John William. Função Geradora: Uma ferramenta de contagem. (2015).
- [118] **Lei  a et al. (2015)** Roland Lei  a et al. A graph-based higher-order intermediate representation. *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015* (2015), 202–212. doi: 10.1109/CGO.2015.7054200.
- [119] **Cao e Nguyen (2015)** Son Thanh Cao e Linh Anh Nguyen. https://link.springer.com/chapter/10.1007%2F978-3-319-10518-5_9. Accessed: 06-jul.-2019. doi: 10.1007/978-3-319-10518-5_9.
- [120] **Pinto (2015)** F  bio Lima Pinto. Uso Recorr  ncias e do Racioc  nio Recursivo no Ensino M  dico. *Sociedade Brasileira de Matem  tica - SBM*. (2015).
- [121] **Van Werven (2015)** J. F. Van Werven. Reconsidering Recursion. (2015).
- [122] **Ekblad (2015)** Anton Ekblad. A Distributed Haskell for the Modern Web. (2015).
- [123] **Wirth (2015)** Michael A. Wirth. The far side of recursion. *Teaching Mathematics and Computer Science*. 13, 1 (2015), 57–71. doi: 10.5485/tmcs.2015.0381.

- [124] **Hu et al. (2015)** Zhenjiang Hu et al. How functional programming mattered. *National Science Review*. 2, 3 (2015), 349–370. doi: 10.1093/nsr/nwv042.
- [125] **Sekar (2015)** R. Sekar. Functional Programming for Imperative Programmers Bottom-up vs Top-down Recursive Computation. (2015), 1–12.
- [126] **Silva (2015)** Israel Carley da Silva. Recorrências: uma abordagem sobre sequências recursivas para aplicações no Ensino Médio. (2015).
- [127] **Bassiliades et al. (2015)** Nick Bassiliades et al. An Empirical Approach to Query-Subquery Nets with Tail-Recursion Elimination. *Advances in Intelligent Systems and Computing*. 312, September (2015), 109–110. doi: 10.1007/978-3-319-10518-5.
- [128] **Brázdil et al. (2015)** Tomáš Brázdil et al. Runtime analysis of probabilistic programs with unbounded recursion. *Journal of Computer and System Sciences* (2015), 288–310. doi: 10.1016/j.jcss.2014.06.005.
- [129] **Avanzini et al. (2015)** Martin Avanzini et al. A new order-theoretic characterisation of the polytime computable functions. *Theoretical Computer Science*. 585, 25 (2015), 3–24. doi: 10.1016/j.tcs.2015.03.003.
- [130] **Tauber et al. (2015)** Tomáš Tauber, Xuan Bi, Zhiyuan Shi, Weixin Zhang, Huang Li, Zhenrui Zhang, Bruno C. D. S. Oliveira. https://link.springer.com/chapter/10.1007%2F978-3-319-26529-2_2. Accessed: 06-jul.-2019. doi: 10.1007/978-3-319-26529-2_2.
- [131] **Downen et al. (2016)** Paul Downen et al. Sequent calculus as a compiler intermediate language. *ACM SIGPLAN Notices*. 51, 9 (2016), 74–88. doi: 10.1145/3022670.2951931.
- [132] **Schneider et al. (2016)** Sigurd Schneider et al. An Inductive Proof Method for Simulation-based Compiler Correctness. (2016), 1–29. Recuperado de <http://arxiv.org/abs/1611.09606>.
- [133] **Widemann (2016)** Baltasar Trancón y Widemann. Higher-Order Recursion Abstraction: How to Make Ackermann, Knuth and Conway Look Like a Bunch of Primitives, Figuratively Speaking. (2016), 1–14. Recuperado de <http://arxiv.org/abs/1602.05010>.
- [134] **Mazzanti (2016)** S. Mazzanti. Unbounded Recursion and Non-size-increasing Functions. *Electronic Notes in Theoretical Computer Science*. 322, (2016), 197–210. doi: 10.1016/j.entcs.2016.03.014.
- [135] **Bove et al. (2016)** Ana Bove et al. Partiality and recursion in interactive theorem provers—an overview. *Mathematical Structures in Computer Science*. 26, 1 (2016), 38–88. doi: 10.1017/S0960129514000115.
- [136] **Maurer et al. (2017)** Luke Maurer et al. *Compiling without continuations*. doi: 10.1145/3140587.3062380.

- [137] **Cai et al. (2017)** Jonathon Cai et al. Making Neural Programming Architectures Generalize via Recursion. 2016 (2017), 1–20.
- [138] **Diehl (2017)** Larry Diehl. Fully Generic Programming Over Closed Universes of Inductive- Recursive Types. (2017). doi: 10.15760/etd.3426.
- [139] **William (2017)** Régis William. Verification by Reduction to Functional Programs. 7636, (2017).
- [140] **Alnaeli (2017)** Saleh Alnaeli. on the Usage of Recursive Function Calls in C / C ++. General Purpose on the Usage of Recursive Function Calls in C / C ++. October (2017).
- [141] **Stump (2017)** Aaron Stump. The calculus of dependent lambda eliminations. *Journal of Functional Programming*. 27, November (2017). doi: 10.1017/S0956796817000053.
- [142] **Manuel e Assunção (2017)** Paulo Manuel e Fernandes De Assunção. Verificação Formal de Programas com SPARK2014. (2017).
- [143] **Farvardin (2017)** Kavon Farvardin. Weighing Continuations for Concurrency. (2017).
- [144] **Fakult (2018)** Von Der Fakult. Automated Complexity Analysis of Rewrite Systems. (2018).
- [145] **Bołotina e Pelenitsyn (2018)** Anna Bołotina e Artem Pelenitsyn. Handling Recursion in Generic Programming Using Closed Type Families. (2018), 1–22.
- [146] **Madsen et al. (2018)** Magnus Madsen et al. Tail call elimination and data representation for functional languages on the Java virtual machine. *CC 2018 - Proceedings of the 27th International Conference on Compiler Construction, Co-located with CGO 2018*. 2018–Febru, (2018), 139–150. doi: 10.1145/3178372.3179499.
- [147] **Copeland e Shagrir (2018)** B. Jack Copeland and Oron Shagrir. 2018. The Church-Turing thesis: logical limit or breachable barrier? *Commun. ACM* 62, 1 (January 2019), 66–74. DOI:<https://doi.org/10.1145/3198448>
- [148] **Ralston (2019)** Matthew Ralston. Tail Call Elimination In The Opensmalltalk Virtual Machine. (2019).
- [149] **Saleil (2019)** Baptiste Saleil. Simple Optimizing JIT Compilation of Higher-Order Dynamic Programming Languages. (2019).
- [150] **Gidney (2019)** Craig Gidney. Asymptotically Efficient Quantum Karatsuba Multiplication. (2019), 1–11. Recuperado de <http://arxiv.org/abs/1904.07356>.
- [151] **Cong et al. (2019)** Youyou Cong et al. Compiling with continuations, or without? whatever. *Proceedings of the ACM on Programming Languages*. 3, ICFP (2019), 1–28. doi: 10.1145/3341643.
- [152] **Hickey (2020)** Rich Hickey. A history of Clojure. *Proceedings of the ACM on Programming Languages*. 4, HOPL (2020), 1–46. doi: 10.1145/3386321.

BIBLIOGRAFIA

13. Índice de obras referenciadas

- [B1] AHO, Alfred V. et al. **Compiladores, Princípios, Técnicas e Ferramentas.** Tradução de Daniel de Ariosto Pinto. Editora LTC, Livros Técnicos e Científicos Editora S.A. 1995.
- [B2] AHO, Alfred V; HILL, Murray; HOPCROFT, John E; ULLMAN, Jeffrey D. **Data Structures and Algorithms.** 2001.
- [B3] ATALLAH, Mikhail J.; BLANTON. Marina. **Algorithms and theory of computation handbook.** General concepts and techniques. 2nd ed. Taylor and Francis Group, LLC. ISBN 978-1-58488-822-2 (alk. paper). 2010.
- [B4] COPELAND, B. Jack, et.al. **Computability: Turing, Gödel, Church, and Beyond,** MIT Press, 2013, pp.77-104. <<https://mitpress.mit.edu/books/computability>> acesso em 26/09/2020.
- [B5] CORMEM, Thomas H. et al. Thomas H. Cormem, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. **Introduction to Algorithms.** 3rd ed. Massachussets Institute of Technology. 2009.
- [B6] DASGUPTA, Sanjoy; PAPADIMITRIOU, Christos; VAZIRANI, Umesh. **Algorithms.** 2006.
- [B7] DRESCH, Aline et al. Design Science Research: **Método de Pesquisa para Avanço da Ciência e Tecnologia.** Porto Alegre/RS: Bookman Editora,. 204 p. 2015.
- [B8] ERICKSON, Jeff. **Algorithms.** This work is available under a Creative Commons Attribution 4.0 International License. Download this book at <<http://jeffe.cs.illinois.edu/teaching/algorithms/>> acesso 27/09/2020. Copyright 2019 Jeff Erickson.
- [B9] GABBRIELLI, Maurizio e MARTINI, Simone. **Programming Languages: Principles and Paradigms.** Undergraduate Topics in Computer Science. Springer. ISBN 978-1-84882-913-8 e-ISBN 978-1-84882-914-5. DOI 10.1007/978-1-84882-914-5. 2010. Tradução do original italiano editado pela ed. McGraw-Hill de 2006.
- [B10] GRAHAM, Paul. **On Lisp: Advanced Techniques for Common.** 1993. Disponível na página do autor: <<http://www.paulgraham.com/onlisp.html>> acesso em 08/07/2019.
- [B11] HALLOWAY, Stuart. **Programming Clojure. The Pragmatic Programmers.** 2009.
- [B12] HINDLEY, J. Roger e SELDIN, Jonathan P. **Lambda-Calculus and Combinators – an Introduction.** Cambridge University Press. ISBN 9780521898850. ISBN-13 978-0-11-41423-7. 2008 <www.cambridge.org/9780521898850>.
- [B13] HOPCROFT. John e ULLMAN, Jeffrey D. et al. **Formal Languages and their relation to automata.** 1969.

- [B14] HOPCROFT, John; AHO, Alfred V; ULLMAN, Jeffrey D. **The Design and Analysis of Computer Algorithms.** 2001.
- [B15] MCMILLAN, Michael. **Data Structures & Algorithms With Javascript.** Bringing Classic Computing Approaches To The Web. 2014.
- [B16] KNUTH, Donald Ervin. **The art of computer programming: fundamental algorithms.** Original: 1938. ISBN 0-201-89683-4. 3^a Edição. Addison Wesley. 1997.
- [B17] PARR, Terence. **The Definitive ANTLR Reference, Building Domain Specific Languages.** 2013.
- [B18] RABHI, Fethi; LAPALME, Guy. **Algorithms. A Functional Programming Approach.** 2nd ed. Pearson Educational. 1999.
- [B19] ROBERTS, Eric S. **Thinking Recursively.** ISBN 0-471-81652-3. Ed. Wiley; 1^a ed. 1986.
- [B20] SANTOS, José Plínio O. ; MELLO, Margarida P.; e MURARI, Idani T. C. **Introdução à Análise Combinatória.** Editora Ciência Moderna, 4^a edição, 2007.
- [B21] SEBESTA, R.. **Concepts of Programming Languages.** 10th Edition.: Addison-Wesley, 816 pp. ISBN 978-0-13-139531-2. 2012.
- [B22] SEDGEWICK, Robert e FLAJOLET, Philippe. **An Introduction To The Analysis Of Algorithms,** Second Edition. 2013.
- [B23] SEDGEWICK, Robert e WAYNE, Kevin. **Algorithms, part 1.** 4th ed. Princeton University. 2014.
- [B24] SOBRAL, João Bosco M. **Da Computabilidade Formal às Máquinas Programáveis.** 1^a edição. Volume II. Edição do Autor. Florianópolis. 2015.
- [B25] SUMMERFIELD, Mark. **Programming in Go.** 2012.
- [B26] SUSSMAN, Gerald Jay; SUSSMAN, Julie; ABELSON, Harold. **Structure and Interpretation of Computer Programs.** Second Edition. 1993. Disponível em <<https://mitpress.mit.edu/sites/default/files/sicp/index.html>> acesso em 20/09/2020
- [B27] SZWARCFITER, Jayme; MARKENZON, Lilian. **Estrutura de Dados e Seus Algoritmos.** 2^a ed. 1994.
- [B28] TOURETZKY, David S. **Common Lisp: a Gentle Introduction to Symbolic Computation,** ISBN 0-8053-0492-4. 1990.
- [B29] VAREJÃO, Flávio. **Linguagens de Programação: Conceitos e Técnicas.** Rio de Janeiro: Campus, v. 4. 2004.