colorlinks, linkcolor=black, citecolor=black, urlcolor=black

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como parte dos requisitos necessários para a obtenção do Título de Engenheiro de Controle e Automação.

Universidade Federal de Santa Catarina – UFSC
Centro Tecnológico – CTC
Departamento de Informática e Estatística – INE

Disciplina INE410113 – Teoria da Computação

Trabalho 1 - Desafio HDU - 5987

Arthur Raulino Kretzer – 202106307

Florianópolis, 21 de Maio de 2023.+

Sumário

1 Introdução

O cálculo Lambda foi criado por Alonzo Church na década de 1930 em paralelo ao desenvolvimento da Máquina de Turing por Alan Turing. As duas possuem a mesma capacidade de expressão e surgiram com o objetivo de formalizar o conceito de computabilidade. O contexto do cálculo Lambda é aplicado ao desafio HDU – 5987 [1] que é discutido e solucionado neste trabalho utilizando linguagem Python e C++.

2 Cálculo Lambda

O cálculo Lambda é um sistema formal que consegue expressar computação baseado em abstrações de funções e aplicações utilizando apenas atribuição de nomes e substituições. Este cálculo foi apresentado em 1932 e foi refinado até 1940 quando Church apresentou sua versão tipada. Uma curiosidade é que Church foi orientador do doutorado de Turing, porém eles desenvolveram seus trabalhos de forma paralela. Logo após a publicação de ambos os artigos, Turing chegou a acrescentar um apêndice em seu artigo referenciando que o trabalho de Church era equivalente ao seu. [1] O cálculo Lambda ainda hoje é a base de várias linguagens de programação modernas, como a Haskell, que é basicamente uma representação do cálculo Lambda, mas também em linguagens como Python e Javascript, que possuem funções anônimas que são baseadas no cálculo Lambda. [1] Nesse sentido o cálculo Lambda é capaz de resolver diversos problemas computáveis, por exemplo, problemas de decisão, como verificar se x pertence a um conjunto S, cálculo de funções, busca, como por exemplo busca em grafos, otimizações, como a maximização do retorno de uma função. [1]

2.1 Sintaxe e Semântica

O cálculo Lambda é basicamente composto de três elementos: variáveis, definição de funções e aplicação de funções [1]. A letra grega λ é utilizada para definir as funções, por exemplo:

- Função Identidade: λx.x
- Função que devolve a função identidade: λx.(λy.y)

Nestes casos pode-se perceber que λx define a variável da função e os valores após o λx. são a definição da função. Para representar a aplicação da função em uma variável, pode-se escrever x y, onde y será aplicado a função x definida previamente. Neste caso, se x for uma função e y uma variável, podemos chamar y de uma variável independente. Já no caso de λx.x, x não é uma variável independente, pois está ligada a λx. Se tivéssemos λx.y, y seria uma variável independente.

Outro exemplo que representa o escopo das funções lambda é a expressão (λx.(λy.x)) x. Neste caso o terceiro x é uma variável independente, pois está fora do escopo da função λx.

Quando é possível aplicar uma variável a uma função, podemos dizer que ela é β-redutível e simplificar a nossa representação. Exemplo:

- (λx.x) 3 = 3

Já quando não é possível fazer nenhuma redução, é dito que o termo está em sua forma normal. Quando não temos variáveis livres dentro de uma expressão, ela é chamada de expressão fechadas, ou combinadora [1].

2.1.1 Combinador ω e Y

Existem casos especiais de expressões combinadoras que ajudam a resolver problemas, como é o caso da expressão combinadora Ω.

- ω = (λx.x x) (λx.x x)

Ao tentarmos reduzir ω aplicando (λx.x x) em (λx.x x), o resultado é exatamente igual ao início.

- ω = (λx.x x) (λx.x x)
- β-redução = (λx.x x) (λx.x x)

Isso é utilizado para trazer o conceito de recursão para o cálculo Lambda, para resolver por exemplo o cálculo fatorial. Haskell Brooks Curry então criou o combinador Y.

- Y = λf. ((λx.f (x x) (λx.f (x x))

O combinador Y permite escrever uma função λ que irá se repetir indefinidamente, neste caso a função f, que é a variável de entrada do combinador Y.

2.1.2 Lógica Booleana

Exemplos de lógica booleana utilizando cálculo Lambda:

- Verdadeiro = λx y.x
- Falso = λx y.y
- IF = λb x y. b x y

Na função Verdadeiro, dadas variáveis x e y, sempre retorna x. No caso da função Falso, dadas variáveis x e y, sempre retorna y. Isso pode ser utilizado para implementar uma função "IF": Sendo b a função Verdadeiro ou Falso. Exemplo:

- IF Verdadeiro 2 3 => (λb x y. b x y) (λx y.x) 2 3 => (λx y.x) 2 3 => 2

Neste caso testa-se se for verdadeiro devolve 2, caso contrário devolve 3. O mesmo pode ser aplicado para Falso.

- IF Falso 2 3 => (λb x y. b x y) (λx y.x) 2 3 => (λx y.x) 2 3 => 3

Com a função IF, pode-se implementar as funções NOT, AND e OR:

- NOT = λb.IF b Falso Verdadeiro
- AND = λb1 b2.IF b1 b2 Falso
- OR = λb1 b2.IF b1 Verdadeiro b2

Esses exemplos são mais extensos e, portanto, não serão realizados.

3 Desafio HDU – 5987

A partir da introdução feita do cálculo Lambda, é possível compreender melhor o desafio apresentado [2]. Ele propõe implementar um código que seja capaz de identificar variáveis livres em qualquer entrada de expressões do cálculo Lambda. A representação das expressões do cálculo Lambda é ligeiramente alterada para ser compatível com entradas simples de teclado. Dessa forma em vez da letra λ será utilizado o termo 'lambda' e toda expressão de aplicação de função ou declaração de função deve estar contida em parênteses. O desafio provê exemplos de entradas possíveis para o programa e a resposta que contém as variáveis independentes identificadas. Exemplos:

O programa ainda dá a dica de que uma gramática livre de contexto define as entradas. Sendo ela:

Entradas	Variáveis Independentes
x	x
y	y
(lambda (x) (x y))	y
(lambda (x) (x y))	x
((lambda (x) x) (x y))	x y
(lambda (y) (lambda (z) (x (y z))))	x

Tabela 1: Legenda da Tabela

- LcExp -> Variable
- LcExp -> (lambda (Variable) LcExp)
- LcExp -> (LcExp LcExp)

Em que Variable é qualquer string diferente de lambda.

É também importante notar que o desafio ainda informa que a entrada sempre será uma string válida da gramática e que a entrada não vai ultrapassar 10^7 valores. Além disso, toda variável terá somente letras latinas e pode conter hífens e não passará 20 caracteres.

Por fim, sempre haverá um número no início da entrada informando quantas expressões serão passadas para o programa.

3.1 Solução do Desafio

A jornada de solução deste desafio foi bastante complexa e trabalhosa, o que gerou bastante ansiedade devido à escassez de tempo, alternativas de solução e linguagens de programação requeridas. Ao mesmo tempo foi bastante satisfatório ter atingido sucesso em realizar o parsing de uma entrada por uma gramática livre de contexto, utilizar algoritmos de busca em largura, aplicar estruturas de árvore, utilizar expressões regulares e aprender a valorizar que o papel é um recurso muito útil para implementação de algoritmos e códigos.

3.1.1 Algoritmo Cocke-Younger-Kasami - CYK

O primeiro passo dado na implementação foi valorizar a dica dada pelo desafio da gramática livre de contexto. Nesse momento buscaram-se soluções para pelo menos realizar o parsing da entrada na expectativa de conseguir diferenciar as variáveis. Foi encontrado em um material de estudos da DFKI (Deutsches Forschungszentrum für Künstliche Intelligenz) [3] um algoritmo chamado de Cocke-Younger-Kasami (CYK) desenvolvido no final da década de 60 utilizando programação dinâmica. Ele serve para qualquer tipo de gramática e se propõe a armazenar uma tabela n x n, sendo n o tamanho da string, que armazena as subderivações da gramática quando analisamos a string de baixo para cima (bottom-up). Por usar a tabela, ela evita a computação duplicada de termos e pode ser resolvida em tempo polinomial. É necessário, porém, utilizar a gramática na forma normal de Chomsk.

Dessa forma o primeiro passo foi converter a gramática livre de contexto para sua forma normal. A solução encontrada foi:

- S -> AB | EF | Variable
- A -> CS
- B -> SD
- C -> (

- D ->)
- E -> CG
- F -> HB
- G -> lambda
- H -> AD

Para validar a gramática ela foi submetida ao Jflap.

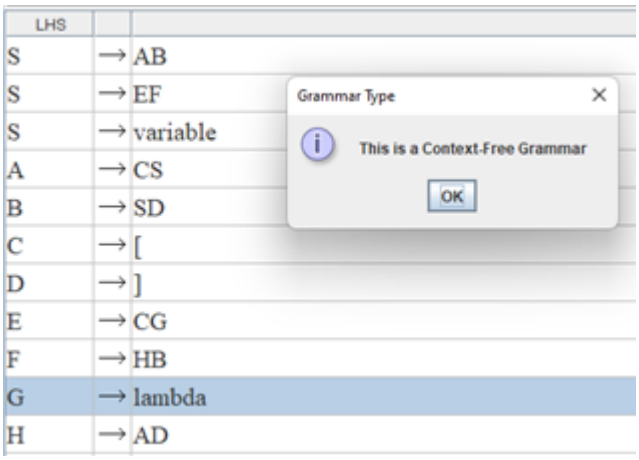

Figura 1: Verificação da gramática com Jflap.

Variable teve que ser convertida para variable, pois o V maiúsculo é interpretado como um termo não terminal. E similarmente (e) tiveram que ser convertidos para [e], pois os primeiros são termos protegidos pelo Jflap ao tentar realizar um parsing com o algoritmo CYK. Note também que a representação de todas as letras latinas como valores terminais não é possível. Dessa forma foi representado somente como variable e, portanto, x e y são rejeitadas.

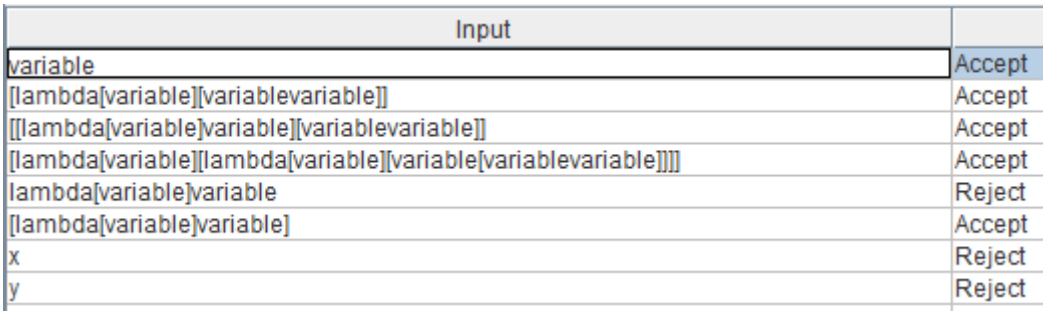

Figura 2: Verificação de entradas com CYK utilizando Jflap.

Note que o termo Variable foi convertido para um REGEX que aceita qualquer combinação de letras latinas contendo hífen ou não com exceção da string lambda. Essa foi uma forma de conseguir representar todos os valores terminais para Variable. A representação da gramática em código foi feita em python utilizando um dicionário em que a chave é o valor não terminal e o valor o valor terminal. Note que os casos que há valores não terminais, como CS, foram convertidos para duas strings em uma sublista dentro de uma lista, para facilitar a implementação do código. Já para S que possui 3 possibilidades, cada possibilidade foi representada com uma lista de termos. Segue a representação a seguir:

```python
# Define the grammar rules. Must be chomsk normalized.
grammar = {
    "S": [["A", "B"], ["E", "F"], [r"(?!lambda)[a-zA-Z]+(-[a-zA-Z]+)*"]],
     "A": [["C", "S"]],
     "B": [["S", "D"]],
     "C": [["("]],
     "D": [[")"]],
     "E": [["C", "G"]],
     "F": [["H", "B"]],
     "G": [["lambda"]],
     "H": [["A", "D"]]
}
```

Em seguida foi realizada a implementação do algoritmo CYK. O primeiro passo foi criar a tabela em que a diagonal principal é composta pelos termos que originam os valores terminais da string de entrada. Para chegar na tabela, é necessário descobrir o tamanho da string. Em python foi utilizado o recurso de quebrar a string em substrings a partir de um REGEX que identifica qualquer termo composto de letras latinas separadas ou não por hífen, já removendo espaços em branco. Isso resulta em uma lista de tamanho n que foi utilizada para gerar a tabela.

```python
def split_input_string(input_str:str ) -> list:
    return re.findall(r"[\\(\\)]|lambda|[a-zA-Z]+(?:-[a-zA-Z]+)?", input_str)
```

```
→ split_input_string(input_str)
> ['x']
→ split_input_string(input_str)
> ['y']
→ split_input_string(input_str)
> ['(', 'lambda', '(', 'x', ')', '(', 'x', 'y', ')', ')']
→ split_input_string(input_str)
> ['(', 'lambda', '(', 'y', ')', '(', 'x', 'y', ')', ')']
```

Figura 3: Exemplo de saídas da função de quebra da string de entrada.

A tabela foi então inicializada com a string '((lambda(x)x)(x y))' e para debug foi utilizada a biblioteca pandas para gerar um arquivo .csv, que foi aberto no Excel.

Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11
	0	1	2	3	4	5	6	7	8	9
0	['C']	[]	[]	[]	[]	[]	[]	[]	[]	[]
1	[]	['G']	[]	[]	[]	[]	[]	[]	[]	[]
2	[]	[]	['C']	[]	[]	[]	[]	[]	[]	[]
3	[]	[]	[]	['S']	[]	[]	[]	[]	[]	[]
4	[]	[]	[]	[]	['D']	[]	[]	[]	[]	[]
5	[]	[]	[]	[]	[]	['C']	[]	[]	[]	[]
6	[]	[]	[]	[]	[]	[]	['S']	[]	[]	[]
7	[]	[]	[]	[]	[]	[]	[]	['S']	[]	[]
8	[]	[]	[]	[]	[]	[]	[]	[]	['D']	[]
9	[]	[]	[]	[]	[]	[]	[]	[]	[]	['D']

Figura 4: Tabela em Excel mostrando o resultado da primeira interação do algoritmo CYK.

Note que por conveniência de implementação do código as strings que representam os valores terminais e não terminais foram encapsuladas em uma lista. Nesse momento o algoritmo deve identificar as interseções de duas colunas, que representam outros termos não terminais, de forma recursiva. Por exemplo, CG é gerado por E, então na célula (1,0) deve aparecer o valor E na solução final.

Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11
	0	1	2	3	4	5	6	7	8	9
0	['C']	['E']	[]	[]	[]	[]	[]	[]	[]	['S']
1	[]	['G']	[]	[]	[]	[]	[]	[]	[]	[]
2	[]	[]	['C']	['A']	['H']	[]	[]	[]	[]	['F']
3	[]	[]	[]	['S']	['B']	[]	[]	[]	[]	[]
4	[]	[]	[]	[]	['D']	[]	[]	[]	[]	[]
5	[]	[]	[]	[]	[]	['C']	['A']	[]	['S']	['B']
6	[]	[]	[]	[]	[]	[]	['S']	[]	[]	[]
7	[]	[]	[]	[]	[]	[]	[]	['S']	['B']	[]
8	[]	[]	[]	[]	[]	[]	[]	[]	['D']	[]
9	[]	[]	[]	[]	[]	[]	[]	[]	[]	['D']

Figura 5: Tabela em Excel mostrando o resultado final do algoritmo CYK.

Note que a célula (9,0) deve sempre conter S, caso contrário a string não é aceita, pois não foi encontrada uma solução. O algoritmo implementado foi chamado de $parse_cyk$. Note que a biblioteca pandas foi utilizada somente para debug e estará fora da implementação em C++ submetida ao portal.

```python
def parse_cyk(input_str):
    input_splitted = split_input_string(input_str)
    input_length = len(input_splitted)

    # Create a table for memoization
    table = [[[] for _ in range(input_length)] for _ in range(input_length)]

    pd.DataFrame(table).to_csv('table_raw.csv')

    # Initialize the table
    for i in range(input_length):
        token = input_splitted[i]
        # Iterate over the rules
        for non_terminal, rule in grammar.items():
            for rhs in rule:
                # If a terminal is found
                if is_terminal_and_equal_to_token(rhs, token):
                    table[i][i] = [non_terminal]

    pd.DataFrame(table).to_csv('table_initialized.csv')

    # Fill the table
    for length in range(2, input_length + 1):
        for i in range(input_length - length + 1):
            j = i + length - 1
            for k in range(i, j):
                # Iterate over the rules
                for non_terminal, rules in grammar.items():
                    for rule in rules:
                        if len(rule) == 2:
                            left_symbol_rule = rule[0]
                            right_symbol_rule = rule[1]
                            # Now searches for matches in table
                            for left_symbol in table[i][k]:
                                for right_symbol in table[k + 1][j]:
                                    if (left_symbol_rule == left_symbol) and (right_symbol_rule == right_symbol):
                                        table[i][j].append(non_terminal)

    pd.DataFrame(table).to_csv('table_finished.csv')

    # Check if the input string is accepted
    check = "S" in table[0][input_length - 1]
    if check:
        return check, table, input_length
    else:
        return check, [], input_length
```

A função $is_terminal_and_equal_to_token$ verifica se o símbolo encontrado é terminal e lida com o REGEX criado para identificar as variáveis. Note que símbolos terminais possuem listas de tamanho 1, enquanto símbolos não terminais só podem ter listas de tamanho 2. Como o REGEX é um caso especial, sempre se testa se é o regex com a função match. Caso contrário compara o símbolo com a regra encontrada.

```python
def is_terminal(rule):
    return len(rule) == 1

def is_terminal_and_equal_to_token(rule, token):
    try:
        match = re.match(fr"{rule[0]}", token)
    except:
        match = (rule[0] == token)
    return is_terminal(rule) and match
```

O retorno da função $parse_cyk$ é uma variável booleana que acusa True se a string foi aceita pela gramática e False se não foi. A tabela e o tamanho da string de entrada após split são retornadas para uma etapa posterior onde a árvore de derivação é gerada. O algoritmo foi testado com as strings de exemplo do desafio e algumas adicionais.

```
Input string x is accepted
Input string y is accepted
Input string (lambda (x) (x y)) is accepted
Input string (lambda (y) (x y)) is accepted
Input string ((lambda(x)x)(x y)) is accepted
Input string (lambda (y) (lambda (z) (x (y z)))) is accepted
Input string marmota is accepted
Input string lambda(x)x is rejected
Input string (lambda(x)x) is accepted
```

Figura 6: Resultado do teste do algoritmo CYK.

3.1.2 Construção da árvore de derivação

A partir da tabela gerada, monta-se uma representação da árvore de derivação. Ela parte da busca dos nodos à direita e à esquerda na tabela similar a uma busca em largura. A procura sempre parte de um ponto inicial e procura a primeira célula com lista não vazia. Se a célula não estiver na diagonal principal, esse ponto é incluído em uma fila para a próxima busca. Ao encontrar o índice da diagonal principal, o código para. Note que no caso da palavra (lambda (x) (lambda(z) (x(y z)))) há uma ambiguidade na célula (‘)’, x) e (‘)’, z) entre o símbolo A e B, que pôde ser resolvida na próxima iteração pelo algoritmo CYK que identificou H como a ramificação adequada, ignorando B. Porém B permaneceu na tabela. Para ignorar essas ambiguidades, a cada nodo criado se inseriu a lista de possíveis valores que aquele nodo pode derivar. Por exemplo, H não pode derivar B, então ele ignora qualquer B encontrado na busca da derivação à direita.

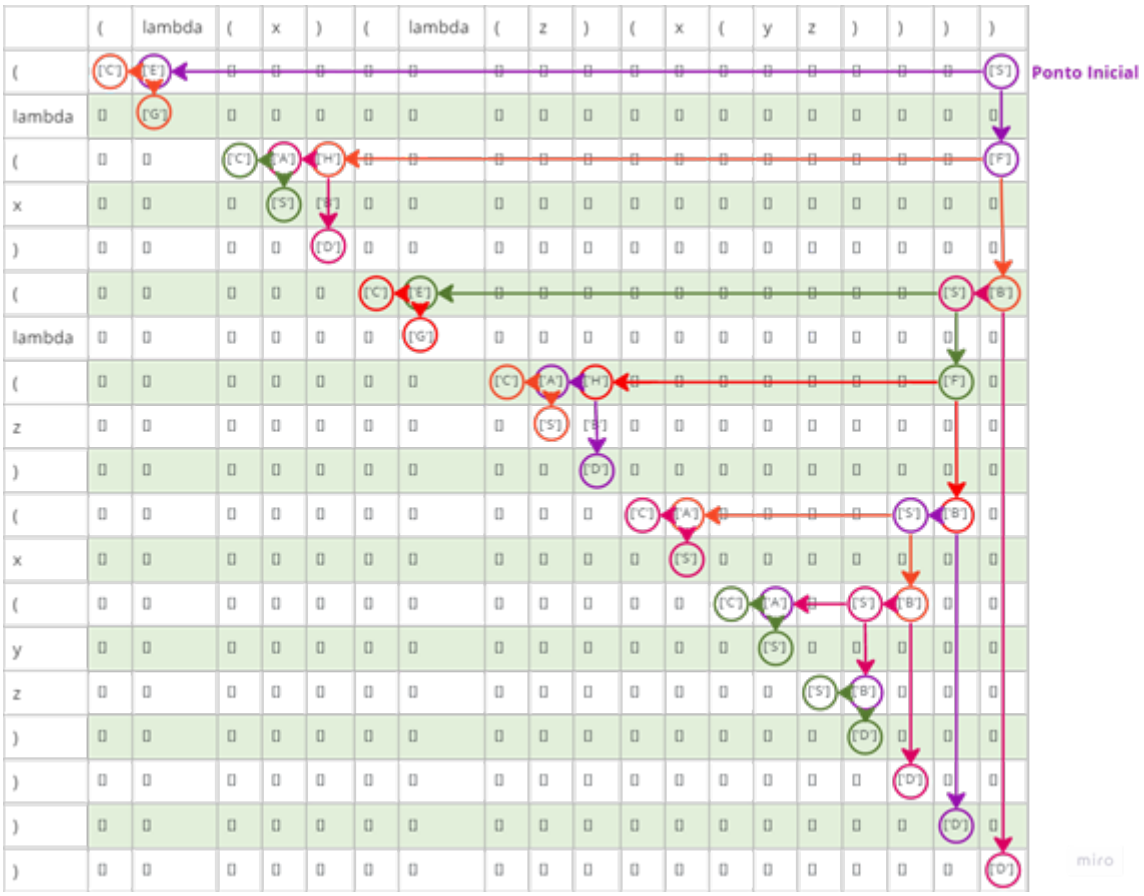

Figura 7: Construção da árvore de derivação.

```
class Node:
    def __init__(self, value, terminal_value=None):
        self.value = value
        self.terminal_value = terminal_value
        self.left = None
        self.right = None
        self.possible_symbols = [[item] for sublist in grammar[value[0]] for
    item in sublist]

    def add_right(self, node):
        self.right = node

    def add_left(self, node):
        self.left = node

def search_left(initial_x_index, line, possible_symbols):
    for x_index in range(initial_x_index-1, -1, -1):
        if line[x_index] in possible_symbols:
            return line[x_index][0], x_index

def search_right(initial_y_index, line, possible_symbols):
    for y_index in range(initial_y_index+1, len(line)):
        if line[y_index] in possible_symbols:
            return line[y_index][0], y_index

def search_nodes(table, x_index, y_index, possible_symbols):

    left_value, left_index = search_left(x_index, table[y_index],
```

```
    possible_symbols)
    left_point = [left_index, y_index]
    right_value, right_index = search_right(y_index, [line[x_index] for line in
    table], possible_symbols)
    right_point = [x_index, right_index]

    return left_value, left_point, right_value, right_point

def build_tree(table, input_length, input_str):
    input_splitted = split_input_string(input_str)
    initial_node = Node(table[0][input_length - 1])
    initial_point = [input_length - 1, 0]

    if initial_point[0] == initial_point[1]:
        initial_node.terminal_value = input_splitted[0]
        return initial_node

    queue = [(initial_node, initial_point)]

    while queue:
        node, point = queue.pop(0)
        x_index = point[0]
        y_index = point[1]
        left_symbol, left_point, right_symbol, right_point = search_nodes(table,
     x_index, y_index, node.possible_symbols)

        if (left_point[0] == left_point[1]) & (left_symbol == 'S'):
            left_node = Node(left_symbol, terminal_value=input_splitted[
    left_point[0]])
        else:
            left_node = Node(left_symbol)

        if (right_point[0] == right_point[1]) & (right_symbol == 'S'):
            right_node = Node(right_symbol, terminal_value=input_splitted[
    right_point[0]])
        else:
            right_node = Node(right_symbol)

        node.add_left(left_node)
        node.add_right(right_node)

        if left_point[0] != left_point[1]:
            queue.append((left_node, left_point))

        if right_point[0] != right_point[1]:
            queue.append((right_node, right_point))

    return initial_node
```

O retorno da função build tree é a referência para o nodo inicial, que irá conter também todos os nodos encontrados, ou seja, a árvore de derivação completa.

3.1.3 Identificação das Variáveis Livres

As variáveis Livres, chamadas aqui também de independentes, foram identificadas com um algoritmo de busca em largura e profundidade. Para chegar nesse algoritmo foi necessário identificar quando que uma ramificação gera variáveis lambda e quando pode gerar variáveis independentes. Utilizando a gramática normalizada que foi desenvolvida, foram feitos diversos desenhos manuais das árvores de derivação até perceber esse padrão.

Com isso percebeu-se que somente a ramificação F gera variáveis lambda e, portanto, variáveis ligadas ou independentes.

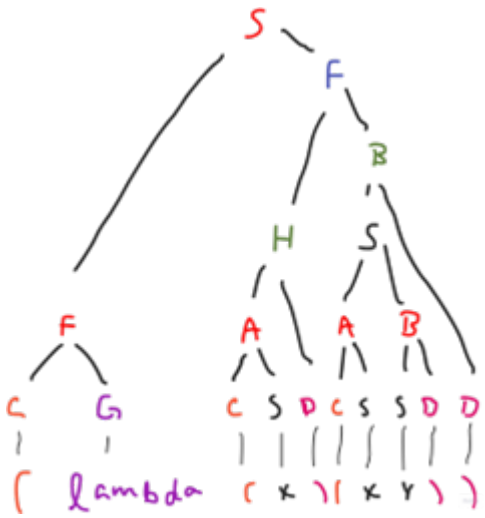

Figura 8: Árvore de derivação para (lambda(x)(x y)).

Quando uma variável está fora de qualquer ramificação F, ela é obrigatoriamente independente.

Note que não é possível distinguir pelas ramificações, entretanto, quais são de fato variáveis independentes e ligadas. Porém percebe-se que a ramificação H gera sempre as variáveis lambda enquanto a ramificação B gera variáveis ligadas e independentes. Dessa forma, o algoritmo de busca utilizado procura por qualquer variável S na ponta. Durante a busca, caso alguma ramificação F seja encontrada se faz uma segunda busca em largura específica naquela ramificação pelas duas ramificações H e B por variáveis S.

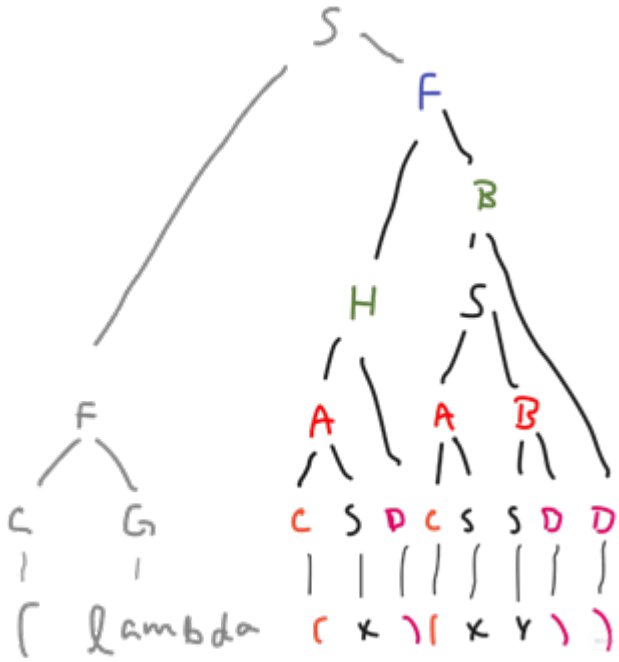

Figura 9: Ramificação F gerando variáveis.

```python
def depth_search_for_variables(root: Node) -> list:

    variables = []

    if root is None:
        return variables

    if root.value == 'F':
        independent_variables = depth_search_considering_lambda_variables(root)
        return independent_variables

    independent_variables = depth_search_for_variables(root.left)
    variables += independent_variables
    independent_variables = depth_search_for_variables(root.right)
    variables += independent_variables

    if root.terminal_value is not None:
        variables.append(root.terminal_value)

    return variables
```

Aquelas que forem encontradas em H são variáveis lambda e as variáveis encontradas em B são comparadas com as variáveis encontradas em H. Caso a variável em B esteja na lista obtida em H, ela é uma variável ligada. As que não aparecem na lista H, são variáveis independentes naquele escopo. O retorno da busca é, portanto, somente as variáveis independentes. Isso é feito de maneira recursiva de tal forma que caso alguma outra ramificação F derive da ramificação B, será feita mais uma análise de variáveis independentes naquele escopo.

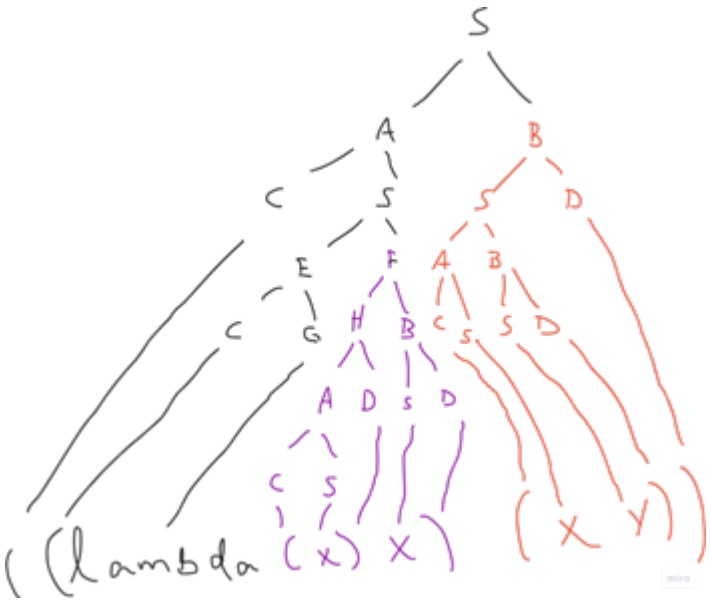

Figura 10: Ramificação F e ramificação “independente” da string ((lambda (x) x) (x y)).

```python
def depth_search_considering_lambda_variables(root: Node) -> list:

    variables = []

    if root is None:
        return variables

    lambda_variables = depth_search_for_variables(root.left)
    possible_independent_variables = depth_search_for_variables(root.right)

    variables = get_independent_variables(lambda_variables,
    possible_independent_variables)

    return variables
```

Foi então criado um main para execução do parsing simulando diversas entradas com teclado.

```python
def main():
    input_strs = ['x', 'y', '(lambda (x) (x y))', '(lambda (y) (x y))', "((
    lambda(x)x)(x y))", "(lambda (y) (lambda (z) (x (y z))))", 'marmota', '
    lambda(x)x', "(lambda(x)x)"]
    for input_str in input_strs:
        # Parse the input string
        is_accepted, table, input_length = parse_cyk(input_str)
        if table:
            tree = build_tree(table, input_length, input_str)
            print(f"Case #{input_strs.index(input_str)}: {' '.join(
    depth_search_for_variables(tree))}")

if __name__ == "__main__":
    main()
```

O print foi formatado para o padrão estipulado pela plataforma em que é indicado o caso em questão seguido pela variável independente identificada separada por espaço. Veja que o resultado em branco é fruto da string que não possui variáveis independentes e que a string inválida foi removida do resultado.

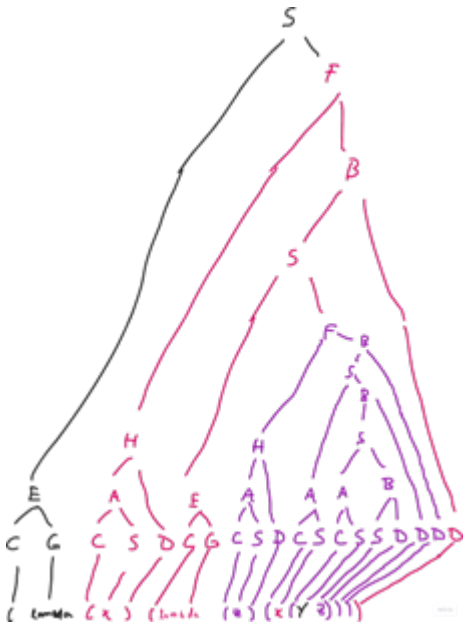

Figura 11: Árvore de derivação para string (lambda(x) (lambda (z) (x(y z)))) com duas derivações F.

```
Case #0: x
Case #1: y
Case #2: y
Case #3: x
Case #4: x y
Case #5: x
Case #6: marmota
Case #8:
```

Figura 12: Resultado da execução do código de identificação de variáveis livres.

3.2 Dificuldades encontradas

A plataforma online limita as linguagens de programação que podem ser utilizadas. Infelizmente o Python não é contemplado pela plataforma é a linguagem que o autor possui maior domínio. De qualquer forma a primeira implementação foi realizada em Python para validar os conceitos da solução sem perder tempo com as dificuldades provenientes de revisar, compilar e debugar uma linguagem já estudada, como C++. Porém, a tradução para C++ tomou muito tempo e esforço dadas as dificuldades já citadas e mudanças de tipagem e facilidades da linguagem Python não disponíveis em C++. Além disso, a falta de bagagem de conhecimento de algoritmos para solução de problemas tradicionais de programação foi também uma barreira a ser superada. Demorou-se muito tempo até encontrar algoritmos de parsing que se utilizam de linguagens livres de contexto e depois implementá-lo em código. De forma similar, o algoritmo de busca em largura era conhecido, porém nunca havia sido implementado pelo autor em código. Outra dificuldade foi lidar com as estruturas de dados, pois o algoritmo de parsing utilizado devolvia uma tabela que depois deveria ser convertida em uma estrutura de árvore correspondente a árvore de derivação da gramática. Além disso, como tudo deveria ser composto em um único arquivo para submissão na plataforma, a organização do código ficou comprometida, o que dificultou o desenvolvimento e correção de bugs.

4 Conclusão

Mesmo com as dificuldades encontradas, o algoritmo foi capaz de identificar as variáveis independentes a partir da gramática livre de contexto desta representação do cálculo Lambda. Esse exercício foi bastante proveitoso para exercitar o uso de diversos algoritmos conhecidos como o CYK e a busca em largura em árvores.

Referências

1 UFABC. *Programação Funcional com Haskell: Capítulo 2 - Funções e Lambda*. <https://haskell.pesquisa.ufabc.edu.br/haskell/02.lambda/>. Acesso em: 21 maio 2023.

2 vjudge.net. *HDU-5987 Problem*. <https://vjudge.net/problem/HDU-5987>. Acesso em: 21 maio 2023.

3 DFKI - Deutsches Forschungszentrum für Künstliche Intelligenz. *Context-Free Grammars: Slides*. <https://www.dfki.de/compling/pdfs/cfg-slides.pdf>. Acesso em: 21 maio 2023.