Calculadora Lógica

Demonstração do funcionamento da calculadora lógica.

Introdução

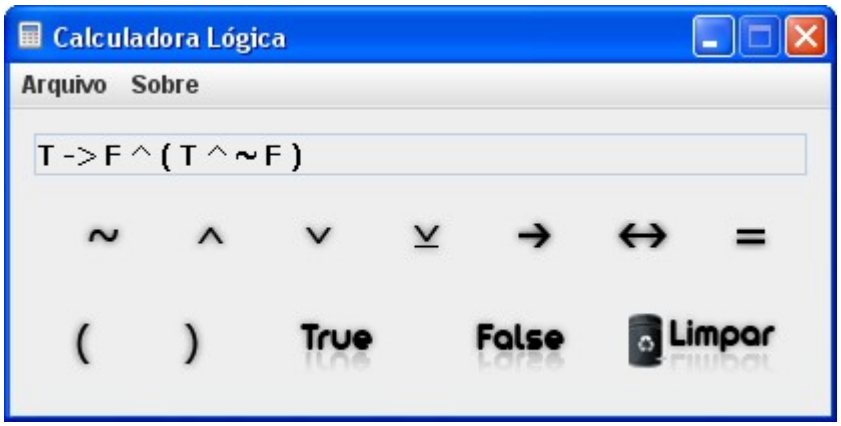
O que a calculadora deve realizar?
A calculadora deve realizar a resolução de uma expressão lógica, para tal, deve obedecer as ordens de precedência de cada operador, e os parentes.

Que linguagem de programação ela é feita?
A calculadora é feita em Java.

Como utilizar a calculadora?
A calculadora possui um formulário para a realização das expressões e exibição da resolução. Para utilizar basta clicar nos botões, ou ainda pode digitar utilizando o teclado, sendo as teclas de atalho:

- “T”, operando verdadeiro (True);
- “F”, operando falso (False);
- “N”, operador negação (~);
- “E”, operador conjunção (^);
- “O”, operador disjunção (v);
- “X”, operador disjunção exclusiva (x);
- “C”, operador condicional (⟶);
- “B”, operador bicondicional (<->);
- “A”, abertura de parentes “(“;
- “S”, fechamento de parentes “)”;
- “=”, resolve a expressão;
- “Backspace”, remove o último elemento da expressão;
- “Delete”, limpa a expressão

Formulário da calculadora



Entendendo o algorítimo

A calculadora trabalha com:

- duas **pilhas**, sendo elas a pilha de operando (apenas T ou F), e a pilha de operadores (objeto da inner class Operador);
- uma **Inner Class** Operador, o qual armazena o operador e a sua precedência;
- um **HashMap** dos operadores e suas precedências;
- uma **variável inteira** contadora de parentes.

A calculadora possui quatro métodos que necessitam ser chamados pela aplicação, sendo eles:

- O construtor da classe, o qual apenas armazena no HashMap os operadores e suas precedências;
- o método *boolean isValid(String expressão)*, o qual válida a expressão;
- se a expressão for válida chamamos o método *void tokenize(String expressão),* o qual quebra a expressão em partes (tokens), e faz o devido tratamento;
- e por fim temos o *String start(),* o qual resolve o resto da expressão que sobra devido a ordem de precedência, e retorna T ou F, que é o resultado final.

Resolvendo expressões simples

T ^ T

Pilha Operandos:

Pilha Operadores:

Resolvendo expressões simples

T ^ T

Pilha Operandos: T

Pilha Operadores:

Quando iniciarmos o tokenize, ele vai capturar o T. T é um operando ele apenas insere na pilha de Operandos.

Resolvendo expressões simples

T ^ T

Pilha Operandos: T

Pilha Operadores: ^

O tokenize, capturou o ^.
^ é um operador ele vai verificar se resolve, como não temos dois operando, e a pilha de operadores estava vazia então não resolve

Resolvendo expressões simples

T ^ T

Pilha Operandos: T T

Pilha Operadores: ^

Por fim o tokenize capturou o T
e empilhou na pilha de operandos

Resolvendo expressões simples

T ^ T

Pilha Operandos: T T

Pilha Operadores: ^

Por fim o tokenize capturou o T
e empilhou na pilha de operandos

Agora acabou nosso tokenize, mas não temos a resposta, neste caso temos o método *String start(),* que finaliza este "resto" da expressão que sobrou nas pilhas, até a pilha de operadores for vazia.

Resolvendo expressões simples

T ^ T

Pilha Operandos: T T

Pilha Operadores: ^

Por fim o tokenize capturou o T e empilhou na pilha de operandos

Agora acabou nosso tokenize, mas não temos a resposta, neste caso temos o método *String start(),* que finaliza este "resto" da expressão que sobrou nas pilhas, até a pilha de operadores for vazia.

Retiro da pilha e armazeno em uma variável (oprn1)

Resolvendo expressões simples

T ^ T

Pilha Operandos: T T

Pilha Operadores: ^

Por fim o tokenize capturou o T e empilhou na pilha de operandos

Agora acabou nosso tokenize, mas não temos a resposta, neste caso temos o método *String start(),* que finaliza este "resto" da expressão que sobrou nas pilhas, até a pilha de operadores for vazia.

Retiro da pilha e armazeno em uma variável (oprn1)

Retiro da pilha e armazeno em uma variável (oprn2)

Resolvendo expressões simples

T ^ T

Pilha Operandos: T T

Pilha Operadores: ^

Por fim o tokenize capturou o T e empilhou na pilha de operandos

Agora acabou nosso tokenize, mas não temos a resposta, neste caso temos o método *String start(),* que finaliza este "resto" da expressão que sobrou nas pilhas, até a pilha de operadores for vazia.

Retiro da pilha e armazeno em uma variável (oprn1)

Retiro da pilha e armazeno em uma variável (oprn2)

Retiro da pilha e armazeno em uma variável (opr)

Resolvendo expressões simples

T ^ T

Pilha Operandos:

Pilha Operadores:

Oprn1 = T
Oprn2 = T
Opr = ^

Faço a resolução de **Opr2 + Opr + Oprn1**

Neste caso: T ^ T

O resultado desta parte da expressão é T.

Resolvendo expressões simples

T ^ T

Pilha Operandos:

Pilha Operadores:

Capturo o resultado e adiciono a pilha de Operandos

Oprn1 = T
Oprn2 = T
Opr = ^

Faço a resolução de **Opr2 + Opr + Oprn1**

Neste caso: T ^ T

O resultado desta parte da expressão é T.

Resolvendo expressões simples

T ^ T

Pilha Operandos: T

Pilha Operadores:

Capturo o resultado e adiciono a pilha de Operandos

Oprn1 = T
Oprn2 = T
Opr = ^

Faço a resolução de **Opr2 + Opr + Oprn1**

Neste caso: T ^ T

O resultado desta parte da expressão é T.

Resolvendo expressões simples

T ^ T

Pilha Operandos:

Pilha Operadores:

E por fim exibe o resultado, para tal, o método **start**, remove o último elemento da pilha de operandos, retornando o resultado final, neste caso “T”, e encerra a calculadora.

Resolvendo expressões simples

Agora vamos resolver uma expressão um pouco mais complexa

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos:

Pilha Operadores:

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos:

Pilha Operadores: ~

O tokenize começou, e adiciona o negação na pilha de operadores.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: T

Pilha Operadores: ~

Agora como é operando, apenas adiciona na pilha de operandos

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: T

Pilha Operadores: ~

Agora temos um porém, não podemos adicionar o operador “^” na pilha de operadores pois temos um operador com precedência menor que a sua, neste caso o ~ vale 0, e o ^ vale 1.

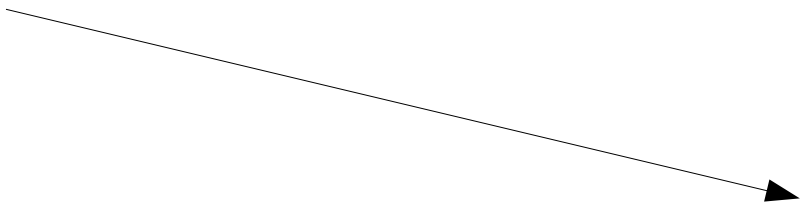
Então resolvemos o que havia na pilha antes de empilhar o “^”!

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: T

Pilha Operadores: ~



Retiro o T da pilha e armazeno na var. oprn1

Agora temos um porém, não podemos adicionar o operador “^” na pilha de operadores pois temos um operador com precedência menor que a sua, neste caso o ~ vale 0, e o ^ vale 1.

Então resolvemos o que havia na pilha antes de empilhar o “^”!

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: T

Pilha Operadores: ~

Agora temos um porém, não podemos adicionar o operador "^" na pilha de operadores pois temos um operador com precedência menor que a sua, neste caso o ~ vale 0, e o ^ vale 1.

Então resolvemos o que havia na pilha antes de empilhar o "^"!

Retiro o T da pilha e armazeno na variável oprn1

Retiro o "~" e armazeno na variável opr

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos:

Pilha Operadores:

Agora temos um porém, não podemos adicionar o operador “^” na pilha de operadores pois temos um operador com precedência menor que a sua, neste caso o ~ vale 0, e o ^ vale 1.

Então resolvemos o que havia na pilha antes de empilhar o “^”!

Então resolvo, neste caso,
oprn1 + opr (~T)
que resulta **F.**

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F

Pilha Operadores:

Agora temos um porém, não podemos adicionar o operador "^" na pilha de operadores pois temos um operador com precedência menor que a sua, neste caso o ~ vale 0, e o ^ vale 1.

Então resolvemos o que havia na pilha antes de empilhar o "^"!

Então resolvo, neste caso,
oprn1 + opr (~T)
que resulta **F.**
Então adiciono na pilha
de operandos.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F

Pilha Operadores: ^

Agora sim podemos adicionar o operador “^” na pilha de operadores, pois, ela está vazia, e além disso ela não possui nenhum operador com ordem de precedência menor do que a do “^”

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: ^

Agora temos outro operando para adicionar, apenas adicionamos ele a pilha de operandos

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: ^

Agora temos o operador "→", antes de adicionarmos ele na pilha de operadores devemos verificar a ordem de precedência dele, ele é o 4, e o "^" é o 1, então devemos resolver o primeiro o que há nas pilhas para poder adicionar

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: ^

Remove da pilha e adiciona na variável oprn1

Agora temos o operador "→", antes de adicionarmos ele na pilha de operadores devemos verificar a ordem de precedência dele, ele é o 4, e o "^" é o 1, então devemos resolver o primeiro o que há nas pilhas para poder adicionar

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: ^

Remove da pilha e adiciona na variável oprn1

Remove da pilha e adiciona na variável oprn2

Agora temos o operador "→", antes de adicionarmos ele na pilha de operadores devemos verificar a ordem de precedência dele, ele é o 4, e o "^" é o 1, então devemos resolver o primeiro o que há nas pilhas para poder adicionar

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: ^

Remove da pilha e adiciona na variável oprn1

Remove da pilha e adiciona na variável oprn2

Remove da pilha e adiciona na variável opr

Agora temos o operador "→", antes de adicionarmos ele na pilha de operadores devemos verificar a ordem de precedência dele, ele é o 4, e o "^" é o 1, então devemos resolver o primeiro o que há nas pilhas para poder adicionar

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos:

Pilha Operadores:

Agora temos o operador “→”, antes de adicionarmos ele na pilha de operadores devemos verificar a ordem de precedência dele, ele é o 4, e o “^” é o 1, então devemos resolver o primeiro o que há nas pilhas para poder adicionar

Resolve o que há nas variáveis
oprn2 + opr + oprn1
F ^ F

Resulta em: F

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F

Pilha Operadores:

Agora temos o operador "→", antes de adicionarmos ele na pilha de operadores devemos verificar a ordem de precedência dele, ele é o 4, e o "^" é o 1, então devemos resolver o primeiro o que há nas pilhas para poder adicionar

Resolve o que há nas variáveis
oprn2 + opr + oprn1
F ^ F

Resulta em: F

Então adiciona na pilha de operandos

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F

Pilha Operadores: →

Agora sim podemos adicionar o operador "→" na pilha de operadores, pois ela está vazia, e estando vazia não tem ninguém com precedência menor que a sua.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: →

Agora podemos adicionar o outro operando, em sua pilha, e assim acabamos nosso tokenize

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: →

Retiro da pilha, e armazeno na variável oprn1

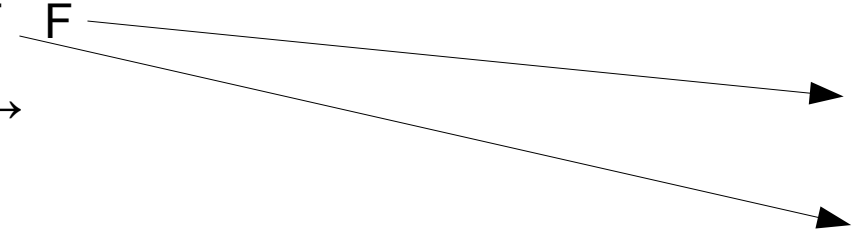
Como acabou nossa tokenize, temos ainda que exibir o resultado na tela, pra isso temos o método que já foi explicado antes, que é o *start* que resolve tudo que há nas pilhas até que a pilha de operadores fique vazia.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F F

Pilha Operadores: →



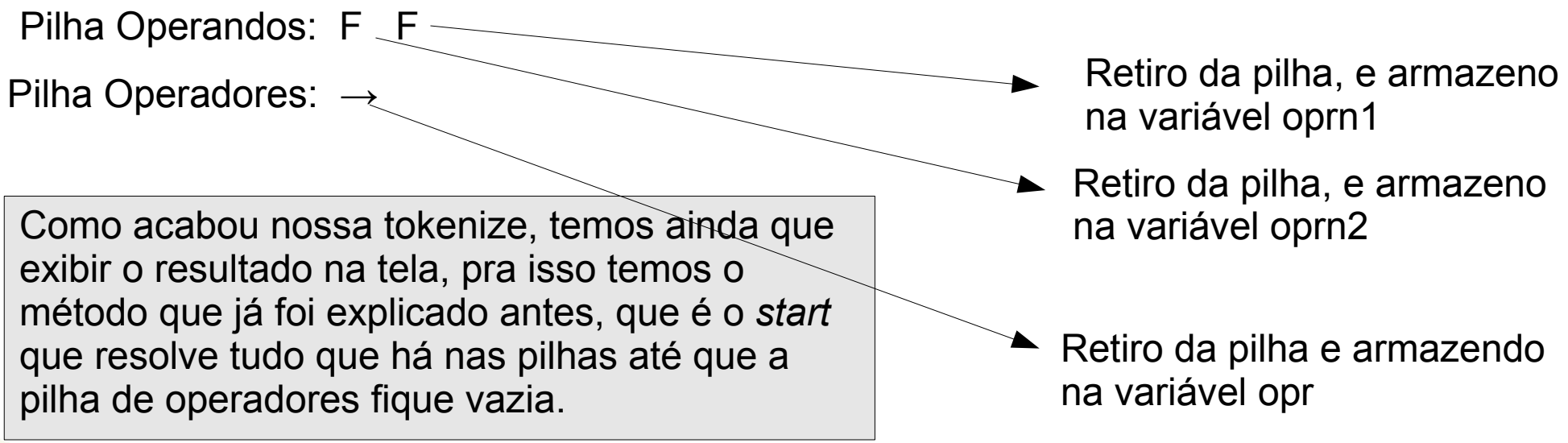
Retiro da pilha, e armazeno na variável oprn1

Retiro da pilha, e armazeno na variável oprn2

Como acabou nossa tokenize, temos ainda que exibir o resultado na tela, pra isso temos o método que já foi explicado antes, que é o *start* que resolve tudo que há nas pilhas até que a pilha de operadores fique vazia.

Resolvendo expressões simples

~T ^ F → F



Como acabou nossa tokenize, temos ainda que exibir o resultado na tela, pra isso temos o método que já foi explicado antes, que é o *start* que resolve tudo que há nas pilhas até que a pilha de operadores fique vazia.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos:

Pilha Operadores:

Agora resolve o que há nas variáveis
F → F

O que resulta em: F

Como acabou nossa tokenize, temos ainda que exibir o resultado na tela, pra isso temos o método que já foi explicado antes, que é o *start* que resolve tudo que há nas pilhas até que a pilha de operadores fique vazia.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F

Pilha Operadores:

Agora resolve o que há nas variáveis
F → F

O que resulta em: F
Então adicionamos na pilha de operandos

Como acabou nossa tokenize, temos ainda que exibir o resultado na tela, pra isso temos o método que já foi explicado antes, que é o *start* que resolve tudo que há nas pilhas até que a pilha de operadores fique vazia.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F

Pilha Operadores:

Agora a pilha de operadores está vazia, então vamos retirar o que há na pilha de operandos e pronto, finalizamos a calculadora.

Resolvendo expressões simples

~T ^ F → F

Pilha Operandos: F

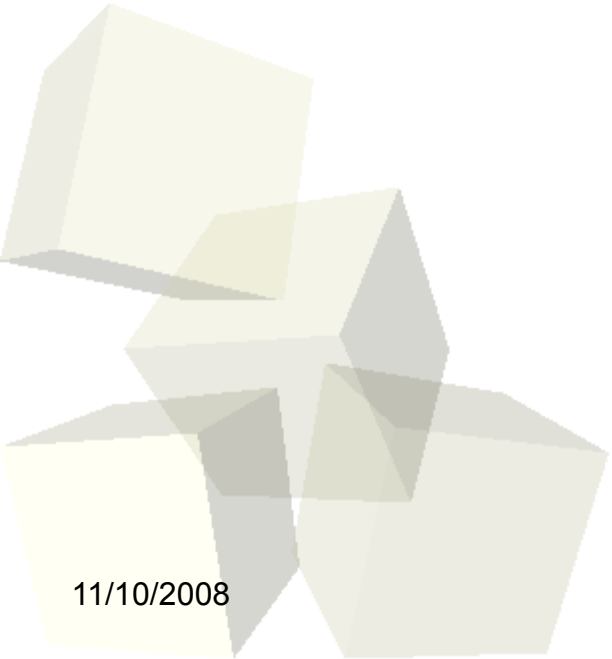
Pilha Operadores:

Retira da pilha e retornando o que removeu como resultado

Agora a pilha de operadores está vazia, então vamos retirar o que há na pilha de operandos e pronto, finalizamos a calculadora.

Resolvendo expressões complexas

Agora vamos resolver uma expressão um pouco mais complexa, contendo parentes



Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores:

Iniciando o tokenize, ele já acha um operando, então já adiciona o mesmo na pilha de operandos

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: v

O token agora é o “v”, como a pilha está vazia, e não temos nenhum operador com ordem de precedência menor do que a do “v”, então podemos adicionar direto na pilha de operadores

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: v

O token agora é o F, apenas adiciona pois é um operando

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

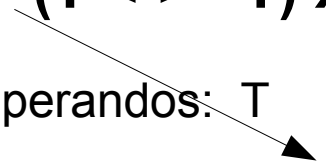
Pilha Operandos: T F

Pilha Operadores: v

O token agora é o “^”, como há operadores na pilha de operadores, temos que verificar a precedência dele, como temos o “v”, a precedência dele é 2, e a do “^” é 1, então só vamos empilhar

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T



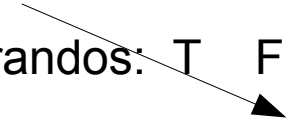
Pilha Operandos: T F

Pilha Operadores: v ^

O token agora é o “^”, como há operadores na pilha de operadores, temos que verificar a precedência dele, como temos o “v”, a precedência dele é 2, e a do “^” é 1, então só vamos empilhar

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T



Pilha Operandos: T F

Pilha Operadores: v ^ (

Contador de parentes: 6

O token agora é o “(”, agora temos um detalhe importante, como no slide 4, existe um atributo da classe que é o contador de parentes, quando acharmos um abre parentes, adicionamos contador += 6 (seis pois é a ordem de precedência maior), e então empilhamos este abre parente na pilha de operadores.

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T

Pilha Operadores: v ^ (

Contador de parentes: 6

Agora temos que empilhar mais um operando na pilha de operandos

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T

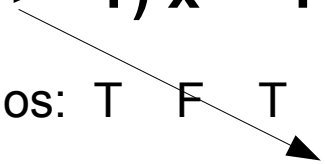
Pilha Operadores: v ^ (

Contador de parentes: 6

Agora temos o operador “<->”, mas antes de adicionar ele em sua pilha, devemos verificar se pode adicionar.
Como o último elemento da pilha de operadores é um “(”, então vamos apenas empilhar, pois não tem nada pra resolver

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T



Pilha Operandos: T F T

Pilha Operadores: v ^ (<->

Contador de parentes: 6

Agora temos o operador “<->”, mas antes de adicionar ele em sua pilha, devemos verificar se pode adicionar.
Como o último elemento da pilha de operadores é um “(”, então vamos apenas empilhar, pois não tem nada pra resolver

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T

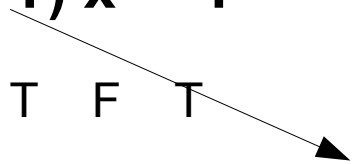
Pilha Operadores: v ^ (<->

Contador de parentes: 6

Agora temos o operador “~”, mas antes de adicionar ele em sua pilha, devemos verificar se pode adicionar.
Como o último elemento da pilha é o “<->” e está valendo -1 (obs: lembre-se do cotnador de parentes) e o “~” vale -6 (0-6 = -6) então só adiciona, pois deve ser maior para poder resolver

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T



Pilha Operandos: T F T

Pilha Operadores: v ^ (<-> ~

Contador de parentes: 6

Agora temos o operador “~”, mas antes de adicionar ele em sua pilha, devemos verificar se pode adicionar.
Como o último elemento da pilha é o “<->” e está valendo -1 (obs: lembre-se do cotnador de parentes) e o “~” vale -6 (0-6 = -6) então só adiciona, pois deve ser maior para poder resolver

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T T

Pilha Operadores: v ^ (<-> ~

Contador de parentes: 6

Agora temos o operando T, apenas adicionamos ele na pilha de operandos

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T T

Pilha Operadores: v ^ (<-> ~

Contador de parentes: 6

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T T

Pilha Operadores: v ^ (<-> ~

Contador de parentes: 6

Retiro e armazeno na variável oprn1

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T T

Pilha Operadores: v ^ (<-> ~

Contador de parentes: 6

Retiro e armazeno na variável oprn1

Como temos uma “~”, retiro e armazeno na variável opr

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T

Pilha Operadores: v ^ (<->

Contador de parentes: 6

Resolve a operação
~T = F

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T F

Pilha Operadores: v ^ (<->

Contador de parentes: 6

Resolve a operação e armazena na pilha de operandos
~T = F

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T F

Pilha Operadores: v ^ (<->

Retira e armazena na variável oprn1

Contador de parentes: 6

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T F

Pilha Operadores: v ^ (<->

Contador de parentes: 6

Retira e armazena na variável oprn1

Retira e armazena na variável oprn2

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F T F

Pilha Operadores: v ^ (<->

Contador de parentes: 6

Retira e armazena na variável oprn1

Retira e armazena na variável oprn2

Retira e armazena na variável opr

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: v ^ (

Contador de parentes: 6

Resolve a expressão
T <-> F = F

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F F

Pilha Operadores: v ^ (

Contador de parentes: 6

Resolve a expressão
T <-> F = F

Então armazena o resultado na pilha de operandos

Muita atenção agora, pois temos um “)”, quando temos um “)”, devemos resolver tudo o que tem na pilha de operadores até que sobre o “(”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F F

Pilha Operadores: v ^ (

Contador de parentes: 6 - 6

Agora como nos restou apenas o “(” na pilha de operadores, então apenas vamos remover ele e o contador de parentes decrementa 6

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F F

Pilha Operadores: v ^

Contador de parentes: 0

Agora como nos restou apenas o “(” na pilha de operadores, então apenas vamos remover ele e o contador de parentes decrementa 6

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F F

Pilha Operadores: v ^

Contador de parentes: 0

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “^” vale 1, então temos que resolver primeiro o “^”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F F

Pilha Operadores: v ^

Contador de parentes: 0

Retira e armazena na variável oprn1

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “^” vale 1, então temos que resolver primeiro o “^”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F F

Pilha Operadores: v ^

Contador de parentes: 0

Retira e armazena na variável oprn1

Retira e armazena na variável oprn2

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “^” vale 1, então temos que resolver primeiro o “^”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F F

Pilha Operadores: v ^

Contador de parentes: 0

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “^” vale 1, então temos que resolver primeiro o “^”

Retira e armazena na variável oprn1

Retira e armazena na variável oprn2

Retira e armazena na variável opr

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: v

Contador de parentes: 0

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “^” vale 1, então temos que resolver primeiro o “^”

Resolvo o que tenho nas variáveis:
F ^ F = F

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: v

Contador de parentes: 0

Resolvo o que tenho nas variáveis:
F ^ F = F

Então armazendo o resultado na pilha de operandos

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “^” vale 1, então temos que resolver primeiro o “^”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: v

Contador de parentes: 0

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “v” vale 2, então temos que resolver primeiro o “v”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: v

Contador de parentes: 0

Retiro e armazeno na variável oprn1

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “v” vale 2, então temos que resolver primeiro o “v”

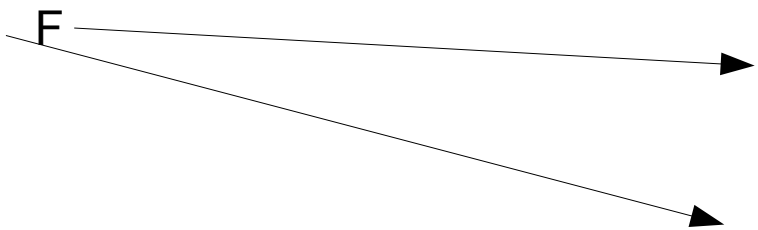
Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: v

Contador de parentes: 0



Retiro e armazeno na variável oprn1

Retiro e armazeno na variável oprn2

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “v” vale 2, então temos que resolver primeiro o “v”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: v

Contador de parentes: 0

Retiro e armazeno na variável oprn1

Retiro e armazeno na variável oprn2

Retiro e armazeno na variável opr

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “v” vale 2, então temos que resolver primeiro o “v”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos:

Pilha Operadores:

Contador de parentes: 0

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “v” vale 2, então temos que resolver primeiro o “v”

Resolvo o que tenho nas variáveis:
T v F = T

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores:

Contador de parentes: 0

Resolvo o que tenho nas variáveis:
T v F = T

Então armazeno na pilha de operandos

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Como o “x” vale 3 e o “v” vale 2, então temos que resolver primeiro o “v”

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: x

Contador de parentes: 0

O tokenize agora encontrou o “x”, mas antes de adicionar ele em sua pilha, devemos verificar se pode.
Agora sim posso adicionar o “x”, pois a pilha de operadores está vazia

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: x

Contador de parentes: 0

O tokenize agora encontrou o “~”, como o “~” vale 0 e o “x” vale 3, então só vamos empilhar

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: x ~

Contador de parentes: 0

O tokenize agora encontrou o “~”, como o “~” vale 0 e o “x” vale 3, então só vamos empilhar

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: x ~

Contador de parentes: 0

O tokenize agora encontrou outro “~”, como o “~” vale 0 e o último elemento da pilha de operadores é outro “~” que também vale 0, então só vamos empilhar

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: x ~ ~

Contador de parentes: 0

O tokenize agora encontrou outro “~”, como o “~” vale 0 e o último elemento da pilha de operadores é outro “~” que também vale 0, então só vamos empilhar

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x ~ ~

Contador de parentes: 0

Agora restou apenas o operando T, vamos apenas empilhar ele

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x ~ ~

Contador de parentes: 0

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x ~ ~

Retira e armazena na variável oprn1

Contador de parentes: 0

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x ~ ~

Contador de parentes: 0

Retira e armazena na variável oprn1

Retira e armazena na variável opr

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: x ~

Contador de parentes: 0

Resolve o que tem nas variáveis
~T = F

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: x ~

Contador de parentes: 0

Resolve o que tem nas variáveis
~T = F

Então adiciona na pilha de operandos

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: x ~

Contador de parentes: 0

Retira e armazena na variável oprn1

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T F

Pilha Operadores: x ~

Contador de parentes: 0

Retira e armazena na variável oprn1

Retira e armazena na variável opr

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T

Pilha Operadores: x

Contador de parentes: 0

Resolve o que tem nas variáveis
~F = T

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x

Contador de parentes: 0

Resolve o que tem nas variáveis
~F = T

Então adiciona na pilha de operandos

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x

Contador de parentes: 0

Retiro e armazeno na variável oprn1

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x

Contador de parentes: 0

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: T T

Pilha Operadores: x

Contador de parentes: 0

Retiro e armazeno na variável oprn1

Retiro e armazeno na variável oprn2

Retiro e armazeno na variável opr

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos:

Pilha Operadores:

Contador de parentes: 0

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolve o que há nas variáveis
T x T = F

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: F

Pilha Operadores:

Contador de parentes: 0

Resolve o que há nas variáveis
T x T = F

Então adiciona na pilha de operandos

Agora acabou o tokenize, só precisamos resolver o que resta nas pilhas, pelo método *start(), ele vai resolver até que a pilha de operadores fique vazia*

Resolvendo expressões complexas

T v F ^ (T <-> ~T) x ~~T

Pilha Operandos: F

Pilha Operadores:

Contador de parentes: 0

E por fim, o start() acabou, e retorna o que tem na pilha de operandos retirando o que há dentro dela, neste caso, o F