MC202 - Estruturas de Dados

Lab 08 - Árvores Binárias e Percurso de Árvores

Data da Primeira Chance: 09 de outubro de 2023

Link da atividade: https://classroom.github.com/a/J6VjMpgH

Peso: 4

Uma companhia está desenvolvendo uma nova funcionalidade para ser adotada em seus sistemas de análise. Essa nova funcionalidade consiste em otimizar formulações compostas de expressões relacionais informadas pelo usuário a partir de valores que podem ser inteiros ou de variáveis, além dos operadores que conectam os valores.

Inteiros nesse contexto são apenas os números inteiros positivos de zero (`0`) a nove (`9`) e as variáveis são letras minúsculas do alfabeto latino (`a`, …, `z`). Dependendo das operações entre esses valores, pode-se resultar valores lógicos que assumem dois valores, verdadeiro e falso, T e F,[1] respectivamente.

Uma expressão é composta de dois operandos, que podem ser um valor variável ou constante (inteiro ou valor lógico), e um operador binário. Além disso, operandos também podem ser outra expressão. Os operadores das expressões consideradas podem ser relacionais, lógicos e de comparação.

Na expressão relacional comparam-se inteiros ou variáveis. Seus operadores, conhecidos como operadores relacionais, são os seguintes: maior que (>), menor que (<), menor igual ({), maior igual (}){2}. Por exemplo, a expressão `(x > 5)` é uma expressão relacional.

Os operadores lógicos são e (&) e ou (|)[3]. Por exemplo, a expressão `(2 > 1) & (y } 9)`, é uma expressão lógica que, após as reduções que realizaremos, concluiremos ser equivalente a `(T & T)`, que por fim resultará em `T` (verdadeiro).

---

[1] Em maiúsculo, para diferenciar de variáveis.
[2] Usualmente na programação esses símbolos são representados da seguinte forma: maior que (>), menor que (<), maior igual (>=), menor igual (<=). Aqui será usada a representação com um caractere a fim de simplificar a leitura dos símbolos.
[3] Em linguagens de programação como *C* e *Java* os operadores “e” e “ou” são representados da seguinte forma respectivamente: “&&” e “||”.

Temos também os operadores igualdade (=), diferente(!)[4] sendo que os seus operandos são do mesmo tipo se ambos forem constantes. Diferentemente dos operadores relacionais e lógicos, exclusivos para suas respectivas expressões, os operadores de igualdade podem ser aplicados a ambas as expressões.

Note que variáveis podem assumir qualquer valor, mas consideramos que assumem valores lógicos se o operador é lógico e assumem valores inteiros se o operador for relacional. Se o operador for de comparação, então consideramos que ela assume o mesmo tipo de valor que o outro operando (se os dois operandos forem variáveis, então não importa o tipo). Note que o valor de uma expressão (dado valores para as suas variáveis) é sempre um valor lógico.

Você deverá considerar algumas reduções nas suas expressões para otimizá-las.

- `T | (qualquer expressão)` deve resultar em *T*
- `F & (qualquer expressão)` deve resultar em *F*
- `(qualquer variável) { 9` deve resultar em `T`
- `(qualquer variável) } 0` deve resultar em `T`
- `(qualquer variável) > 9` deve resultar em `F`
- `(qualquer variável) < 0` deve resultar em `F`
- Se `x` é uma variável ou uma constante (lógica ou inteira), `x = x` deve resultar em `T`
- Se `x` é uma variável ou uma constante (lógica ou inteira), `x ! x` deve resultar em `F`
- Se `x` é uma variável ou uma constante inteira `x < x` deve resultar em `F`
- Se `x` é uma variável ou uma constante inteira `x > x` deve resultar em `F`
- Se `x` é uma variável ou uma constante inteira `x { x` deve resultar em `T`
- Se `x` é uma variável ou uma constante inteira `x } x` deve resultar em `T`

Note que você também precisa considerar expressões simétricas às da lista acima, como, por exemplo, `(qualquer expressão)| T`.

Para implementar essas funcionalidades, é necessário conhecimento em estruturas de dados. Entretanto, os programadores com tal conhecimento estão alocados a outros projetos. Como você está fazendo a disciplina de MC202, será que você consegue implementar essa funcionalidade?

---

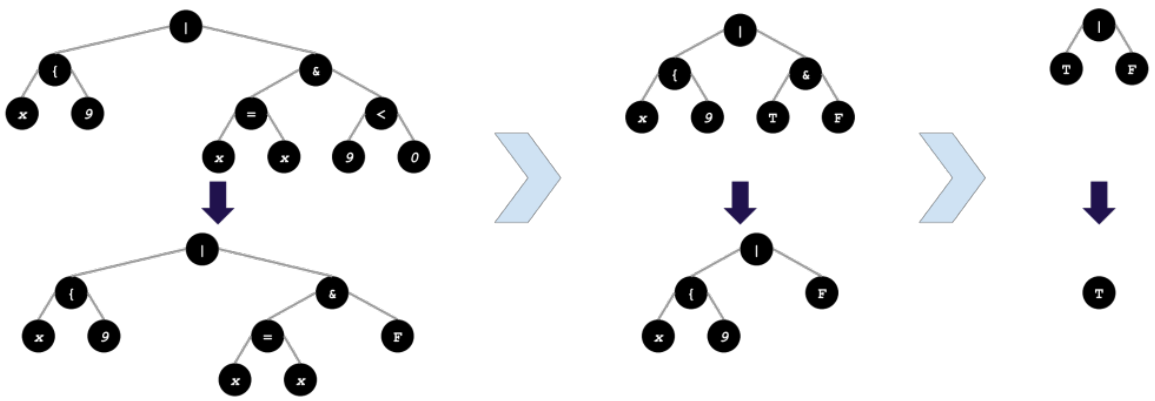
[4] Em C representados por igualdade (==), diferente de (!=).

Tarefa

Escreva um programa em *C* que recebe uma expressão como entrada e gera uma **árvore binária** que representa a expressão dada. Após a construção da árvore, o programa deverá otimizar a árvore enquanto houver uma subárvore para a qual possamos aplicar uma das reduções mencionadas. Note, porém, que há uma forma eficiente de fazer isso, sem ter que analisar a árvore várias e várias vezes…

A saída do seu programa deverá ser a expressão otimizada, conforme as regras acima. Para ilustrar, veja o exemplo a seguir da figura 1 para a expressão `((x { 9) | ((x = x) & (9 < 0)))`.

Figura 1: Árvore para a expressão `((x { 9) | ((x = x) & (9 < 0)))`



A primeira árvore binária da esquerda corresponde à árvore construída a partir da expressão dada como entrada; enquanto a última árvore binária da direita corresponde à árvore obtida após a otimização da expressão.

Entrada

A entrada consiste em um inteiro `n` com o número de expressões a serem otimizadas, seguido de `n` expressões em notação pós-fixa, cada uma em uma única linha.

Saída

A saída para cada expressão lida é a expressão lida em notação infixa em uma linha, seguida da expressão otimizada em notação infixa na próxima linha.

Para a notação infixa, sempre coloque todos os parênteses na expressão, exceto quando a expressão é uma variável ou uma constante. Exemplos:

- Escreva `x` ao invés de `(x)`
- Escreva `T` ao invés de `(T)`
- Escreva `5` ao invés de `(5)`
- Escreva `((x > 5) | (y < 3))` ao invés de `x > 5 | y < 3` ou `(x > 5) | (y < 3)`

Exemplos

Exemplo 1:

Entrada

```
2
90<xx=&x9{|
x5>y3<|
```

Saída

```
(((9 < 0) & (x = x)) | (x { 9))
T

((x > 5) | (y < 3))
((x > 5) | (y < 3))
```

Exemplo 2:

Entrada

```
3
ab=13>|d0{11!&&
x3>mn=|
y7<x9{|
```

Saída

```
(((a = b) | (1 > 3)) & ((d { 0) & (1 ! 1)))
```

```
F

((x > 3) | (m = n))
((x > 3) | (m = n))

((y < 7) | (x { 9))
T
```

Regras e Avaliação

Neste laboratório, é obrigado a usar **os conceitos aprendidos de árvores binárias e percurso de árvores.** Seu código será avaliado não apenas pelos testes do *GitHub*, mas também pela qualidade. Dentre os critérios subjetivos de qualidade de código analisaremos:

- O uso apropriado de funções, de comentários;
- A escolha de bons nomes de funções e variáveis;
- A ausência de trechos de código repetidos desnecessariamente;
- O uso apropriado de funções;
- A eficiência dos algoritmos propostos;
- A correta utilização das Estruturas de Dados;
- O tempo de execução e uso de memória dos algoritmos projetados;
- Vazamento e outros erros de memória (valgrind);
- **O uso de árvores binárias para representar as expressões.**

Note, porém, que essa não é uma lista exaustiva, pois outros critérios podem ser analisados dependendo do código apresentado visando mostrar ao aluno como o código poderia ser melhor.

Submissão

Você deverá criar o arquivo `expressoes.c` e submeter no repositório criado no aceite da tarefa. Você pode enviar arquivos adicionais caso deseje para serem incluídos por `expressoes.c`. Não se esqueça de dar `git push`!

Lembre-se que sua atividade será corrigida automaticamente na aba "*Actions*" do repositório. Confirme a correção e o resultado, já que o que vale é o que está lá e não na sua máquina.

Após a correção da primeira entrega, será aberta uma segunda chance, com prazo de entrega apropriado.

Atenção: O repositório da sua atividade conterá alguns arquivos iniciais. Fica **estritamente proibido** ao aluno alterar os arquivos já existentes, tais como o testador existente ou demais arquivos de configuração do laboratório.

Lembre-se que sua atividade será corrigida automaticamente na aba “Actions” do repositório. Confirme a correção e o resultado, já que o que vale é o que está lá e não na sua máquina.

Após a correção da primeira entrega, será aberta uma segunda chance, com prazo de entrega apropriado.