

[6 - Arquivos] Compilador JIT de Brainf*ck

Disciplina: SCC0222 - Laboratório de Introdução à Ciência da Computação I
Prazo de Entrega: 30/07/2021 23:55:55 Fechado

Contexto

Compiladores Just In Time (JIT) são muito úteis e amplamente utilizados. De maneira bem simples, um compilador JIT consiste em compilar um código apenas no momento que precisamos executá-lo. Várias linguagens usam compiladores JIT, principalmente linguagens interpretadas que buscam melhoras em performance. Isso inclui Java, .NET, Julia e Pypy como alguns exemplos.

Além disso, há outro tipo de compilador chamado "transpiler", ele lê um código fonte de uma linguagem e gera código fonte em outra. Pode parecer um pouco longe da realidade, mas na verdade até mesmo o `gcc` e outros compiladores para C fazem algo similar. O `gcc` gera um código na linguagem assembly que é em si, outra linguagem de programação. Somente esse novo código será transformado em binário.

Descrição

Implemente um programa em C que interprete a linguagem de programação Brainf*ck (<https://pt.wikipedia.org/wiki/Brainfuck#:~:text=brainfuck%2C%20tamb%C3%A9m%20conhecido%20como%20brainf,%C3%A9%20%C3%BAitil%20para%20uso%20pr%C3%A1tico.>) (BF) usando as técnicas descritas a seguir.

A linguagem BF é bem simples. Nela, consideramos que há uma memória com 30000 bytes e um índice ou ponteiro que indica sobre qual byte da memória iremos operar. Esse índice se inicia na posição 0 e a memória deve ser inicializada com zeros em todas as posições. Existem 6 operações que podem ser feitas nesse contexto. Cada operação será representada por um caractere

- `+` - Incrementar o valor no índice da memória.
- `-` - Decrementar o valor no índice da memória.
- `>` - Incrementar o índice.
- `<` - Decrementar o índice.
- `.` - Imprimir o valor naquele índice da memória como um caractere ASCII.

A linguagem em si, é praticamente uma combinação de todos esses caracteres em sequência. Cada operação deve ser lida uma após a outra. Por exemplo, `+++` é um programa válido em BF que torna a posição 0 (inicial) da memória no valor 3. Ou `++.` é outro programa válido que deixa o valor na posição 0 igual a 2 e então imprime esse byte como um caractere ASCII.

Para fechar a sintaxe da linguagem, existe uma última coisa: loops. Os loops são representados entre `[` e `]`, assim como em C o código dos loops começa com `{` e termina com `}`. Quando o programa em BF encontra um `[` ele verifica o valor no índice atual, se o valor for diferente de 0, ele executa tudo até o `]` correspondente, senão, ele pula para a instrução após o `]` correspondente. Quando o programa encontra um `]` ele só pula para o `[` correspondente. Note que, assim como na linguagem C podemos ter loops dentro de loops, esse também é o caso em BF. Como um exemplo, `++[.-]` é um programa que primeiro incrementa o valor da posição 0 (inicial) para o valor 2, em seguida fica num loop que imprime o valor na posição 0 como um caractere ASCII e então decrementa o valor nessa mesma posição, na prática esse programa imprime primeiro o byte de valor 2 e depois o de valor 1, em seguida o valor na posição 0 chegará em 0 e o loop termina, encerrando também o programa.

Por fim, quaisquer outros caracteres podem estar presentes no programa, entretanto eles serão ignorados e considerados como comentários.

Para esclarecer, cada uma dessas operações possui um correspondente direto em C listados na tabela a seguir.

BF	C
<code>+</code>	<code>mem[i]++</code>
<code>-</code>	<code>mem[i]--</code>
<code>></code>	<code>i++</code>
<code><</code>	<code>i--</code>
<code>.</code>	<code>putchar(mem[i])</code>
<code>[</code>	<code>while(mem[i]) {</code>
<code>]</code>	<code>}</code>

Os outros caracteres podem ser descartados completamente e não possuem tradução para C.

Nessa tabela consideramos que o "índice atual" é uma variável `int i` e a memória é

```
char mem[30000] .
```

Interpretando BF

Para interpretar um programa nessa linguagem, iremos usar uma abordagem de tradução do código em BF (uma linguagem nova) para a linguagem C (a linguagem que conhecemos). Ou seja, podemos transformar um código de BF para C e usar um compilador C para transformar isso num binário que podemos executar. Esse processo de tradução de uma linguagem a outra é chamada transpilação (https://en.wikipedia.org/wiki/Source-to-source_compiler).

Para guardar o código C gerado, um arquivo temporário deve ser usado. Esse arquivo se tornará um arquivo fonte normal com extensão `.c`.

Por fim, esse programa que será gerado deve conter um último pedaço de código que será responsável por imprimir os valores não nulos da memória. Ou seja, ao final do código gerado, você deverá adicionar a seguinte lógica:

```
printf("\n");
for (int j = 0; j < 30000; j++) {
    if (mem[j] != 0) {
        printf("%d ", mem[j]);
    }
}
printf("\n");
```

Lembre-se que essa lógica deve **estar contida no código gerado**.

De BF a C

Por exemplo, um programa em BF como `+++. [-]` (completo com a lógica adicional ao fim) seria traduzido para:

```
#include <stdio.h>

int main() {
    char mem[30000];
    int i = 0;
    // Seta todos os lugares da memória para 0.
    for (int j = 0; j < 30000; j++) {
        mem[j] = 0;
    }

    /* Começo do código traduzido de BF */
    mem[i]++;
    mem[i]++;
    i++;
    mem[i]++;
    mem[i]++;
    putchar(mem[i]);
    while (mem[i]) {
        mem[i]--;
    }
    /* Fim do código traduzido de BF */

    printf("\n");
    for (int j = 0; j < 30000; j++) {
        if (mem[j] != 0) {
            printf("%d ", mem[j]);
        }
    }
    printf("\n");

    return 0;
}
```

Note que esses comentários, espaços e indentação **não precisam estar presentes no seu código gerado**. Aqui eles estão presentes apenas por propósitos didáticos.

Digamos que esse código esteja armazenado em um arquivo `jit-gerado.c`. Então, podemos compilar ele `gcc jit-gerado.c -o jit-exe`. E então rodar o executável `./jit-exe`.

Resumo

As seguintes etapas devem ser feitas pela implementação:

1. Seu programa deverá ler da entrada padrão um código em BF e abrir um arquivo temporário com extensão `.c` com um nome qualquer.
2. Para cada operação de BF lida (até o final da entrada, EOF), escrever o código em C correspondente nesse arquivo aberto.
3. Adicionar a lógica para imprimir valores não nulos da memória ao final do programa.
4. Fechar o arquivo temporário.
5. Rodar o comando de compilação do `gcc` nesse arquivo temporário para gerar um executável.
6. Rodar o executável gerado pelo comando de compilação.

Exemplo 1

Explicação:
Incrementa o valor na posição 0, 65 vezes o que deixa o valor 65. Depois, imprime o
nessa posição como um caractere, nesse caso 65 -> 'A'.

Entrada: >+++++++[-]
Saída:
Explicação:
Incrementa o valor na posição 1, 10 vezes. Em seguida roda um loop que decrementa o valor naquela posição. Quando o número chega em 0, ele termina o loop. Não há saída.

```
Entrada: ++++++++[->+++++++<]>++++++.  
Saída:  
a  
97
```

```
Entrada:  
+++++++[>++++++>+++++++>+++<<-]>  
>+>+.+++++. .++>+.<<+++++++>+>.  
>.+++----->+>.  
  
Saída:  
Hello World!  
  
87 100 33 10
```

Atenção: É possível que em sistemas operacionais não baseados em Linux esses comandos precisem ser diferentes. Entretanto como seu código será rodado no run.codes (<http://run.codes>) que funciona com Linux, esses comandos ou versões equivalentes devem estar presentes na sua submissão final. Para testar o código considere utilizar sites como Replit (<https://replit.com/>) caso você só tenha acesso a um sistema Windows. Alternativamente pode-se utilizar do Windows Subsystem for Linux (WSL) que é melhor, mas requer maior configuração.

Algumas instruções poderiam ser facilmente otimizadas. Por exemplo, 10 instruções seguidas de `+` em BF poderiam ser traduzidas para uma única instrução `mem[i] += 10`. O mesmo vale para quase qualquer outra instrução que pode ser repetida. Inclua esses tipos de otimização e veja se consegue tornar seu código mais eficiente.

Envie uma mensagem para o monitor Gabriel Dertoni via telegram @GabrielDertoni (<https://t.me/GabrielDertoni>) ou no Discord da disciplina (<https://discord.gg/9gQ5YQffsA>). Se preferir, também pode enviar um email para o professor Leonardo leonardop@usp.br (<mailto:leonardop@usp.br>) ou para o Gabriel gab.dertoni@usp.br (<mailto:gab.dertoni@usp.br>)

📄 Baixar Casos de Teste (/Exercises/downloadCases/20831)

 (/Exercises/exportExerciseToGoogleCalendar/20831)

<https://run.codes/exercises/view/20831>

 Fechado

Meu Último Envio

 Download (/Commits/download/1452050)

status

Finalizado

compilado

Sim

casos corretos

6/6

pontuação

10.00


Caso	Status	Tempo de CPU	Tam. de Memória Utilizado	Mensagem
Caso 1	Correto	0.0758 s	-1 Kb	Resposta Correta
Caso 2	Correto	0.0525 s	-1 Kb	Resposta Correta
Caso 3	Correto	0.0590 s	-1 Kb	Resposta Correta
Caso 4	Correto	0.3180 s	-1 Kb	Resposta Correta
Caso 5	Correto	0.0652 s	-1 Kb	Resposta Correta
Caso 6	Correto	0.0905 s	-1 Kb	Resposta Correta

Detalhes dos Casos de Teste

Selecione um caso de teste...



Histórico de Entregas

Data	Status	Corretos	Notas	Ações	
22/07/2021 22:53:27	Finalizado	6/6	10.00	 Download (/Commits/download/1452050)	Detalhes (/commits/details/1452050)