Relatório Compiladores

Etapa 6

00274705

Andy Ruiz Garramones

Foram implementadas todas as TAC desenvolvidas na etapa 5. Sendo assim, a etapa 6 foi finalizada completamente.

Como rodar:

Para executar a etapa, siga os comandos a seguir:

```
make
make clean
./etapa6 <nome_arquivo_fonte> <nomeQualquer>
gcc
gcc my_assembly.s
./a.out
```

Ambiente de desenvolvimento:

O código assembly gerado utiliza a sintaxe AT&T de Assembly, gerando código linkável pelo gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2).

Vale ressaltar que foi-se utilizado o Windows Subsystem for Linux, o WSL2, no Windows 10:

"O Subsistema do Windows para Linux permite que os desenvolvedores executem um ambiente GNU/Linux, incluindo a maioria das ferramentas de linha de comando, utilitários e aplicativos, diretamente no Windows, sem modificações e sem a sobrecarga de uma máquina virtual tradicional ou instalação dualboot"

Meu processador é um AMD RYZEN 7 3700X 8-CORE

Implementação:

Foi utilizado a vídeo aula 1 do professor para começar a desenvolver o trabalho e desse ponto em diante o único recurso utilizado foi o gcc com a flag -S. Alguns pontos específicos precisei procurar no stackoverflow sobre como ler e manipular float em hexadecimal.

Para todas as operações, se converteram os números para hexadecimal float assim deixando tudo mais ortogonal pois há apenas 1 forma de somar, subtrair, multiplicar etc… No caso de variáveis int ou char, se faz um cast para float para operar e após as operações se fazia o cast de volta para inteiro para deixar armazenado apenas a parte inteira.

Vale mencionar que tudo é declarado em hexadecimal no arquivo contendo o programa fonte. Se for declarada uma variável com valor 10 será o valor 16 que se atribuirá, e se for declarado com valor 0A.4, receberá o valor 10.25 (ou 10 se for uma variável sem ponto flutuante pois ocorre cast automático).

A leitura e processamento de números hexadecimais em ponto flutuante é implementado se baseando no C99, o standard da linguagem C ISO/IEC 9899:1999 que implementou o ponto flutuante em hexadecimal.

Sendo assim, e tendo se baseado fielmente a como o gcc funciona, um read vai ler um valor em decimal flutuante ou hexadecimal, dependendo se tem 0X na frente para indicar. Um 10.4 será lido como o valor 10.4, porém um 0x10.4 será lido como 16.25. Da mesma forma, um inteiro pode receber 0xF.8 que será salvo 15.

A leitura de TRUE/FALSE é feita como valores numéricos igual se faz em C++ onde 0 é falso e 1 é positivo (ele aceita também outros valores não nulos como positivos na hora do comando read). No entanto, na declaração no código fonte precisa ser TRUE ou FALSE pois é como definimos a linguagem.

As variáveis tipo char imprimem seu caractere, então se for salvo 41 numa variável char, o print dela resultará em A. O mesmo não acontece para vetores char pois como se usa uma variável temp para acessar a posição do vetor, depois o print não sabe qual tipo é e assume que é float pois é como recebeu os valores no registrador.

Como já comentado com o professor, pelo fato de operar tudo em float, se eu fizer um print 5 + 5 * 4 é printado com valores decimais embora pudesse ser em inteiro, ficam os .00000 depois da parte inteira (25.00000). Isso é porque ao haver operação e não se salvar em nenhuma variável global, a variável temporária não tem conhecimento se a resposta seria inteira ou não, ela apenas recebe tudo em float após a operação.

Por último, para a passagem de parâmetros se utilizou um atributo extra que criei na Hash que era um ponteiro para uma lista. Normalmente seria null, mas em caso de identificadores de funções se preencheria com os nomes dos parâmetros do cabeçalho de

uma função. Sendo assim, a cada TAC de ARGV se deposita o valor na equivalente variável de parâmetro. Em caso que se esteja numa função não main, se salva na pilha cada variável local, que veio por parâmetro, para permitir recursão. Como o professor falou que não poderiam ter nomes de parâmetros repetidos entre si ou iguais a variáveis globais, essa implementação foi possível de ser tomada.

Execuções:

A seguir serão mostrados alguns códigos fontes e o resultado de sua execução. Tentei colocar alguns exemplos simples porém que contivessem o máximo de TACs possíveis para ver que tudo está funcionando conforme esperado e pode ser utilizado como se fosse uma linguagem de verdade completa e para tudo.

Todos os códigos rodados abaixo estão disponíveis na pasta programs do .zip submetido.

Cast entre float e int

Apesar do professor ter comentado no Teams que podia fazer tudo em inteiro e não implementar float, como mencionado acima, eu implementei float e todas as operações são em float.

No entanto,há cast automático de float para inteiro e vice-versa quando se salva o valor na variável após se operar.

```
myFloat = float: 0FA;
myInt = int: 0FA;


main () = int
{
    myInt = 5.4
    print "myInt= recebe 5.4, logo myInt eh:", myInt, ""
    myInt = 9
    print "myInt recebe 9, logo myInt eh:", myInt, ""

    myFloat = 5
    print "myFloat recebe 5, logo myFloat eh:", myFloat, ""
    myFloat = 4.4
    print "myFloat recebe 4.4, logo myFloat eh:", myFloat, ""

};
```

```
..mpiler/etapa6
→  etapa6 git:(master) ✗ ./a.out
myInt= recebe 5.4, logo myInt eh:
5

myInt recebe 9, logo myInt eh:
9

myFloat recebe 5, logo myFloat eh:
5.000000

myFloat recebe 4.4, logo myFloat eh:
4.250000
```

Note que ao receber um valor ele é convertido para o tipo da variável, adicionando ou removendo a parte decimal.

Passagem de parâmetros para função chamada

```
c = float: 0;
a = int: 8;
b = float: 9;
myChar = char: 47;

foo (a1 = int, b1 = int, c1 = char) = int
{
    print "\nvalor de a1 que recebeu a", a1
    print "valor de b1 que recebeu b", b1
    print "valor de c1 que recebeu myChar", c1

    a1 = 0
    b1 = 1
    c1 = 61

    print "\nvalor de a1 ao atribuir 0", a1
    print "valor de b1 ao atribuir 1", b1
    print "valor de c1 ao atribuir 61", c1, "\n"
};

main () = int
{
    print "valor de a antes de chamar foo", a
    print "valor de b antes de chamar foo", b
    print "valor de myChar antes de chamar foo", myChar

    c = foo(a, b, myChar)

    print "valor de a depois de chamar foo", a
    print "valor de b depois de chamar foo", b
    print "valor de myChar depois de chamar foo", myChar
};
```

```
..mpiler/etapa6
→  etapa6 git:(master) x ./a.out
valor de a antes de chamar foo
8
valor de b antes de chamar foo
9.000000
valor de myChar antes de chamar foo
G

valor de a1 que recebeu a
8
valor de b1 que recebeu b
9
valor de c1 que recebeu myChar
G

valor de a1 ao atribuir 0
0
valor de b1 ao atribuir 1
1
valor de c1 ao atribuir 61
a


valor de a depois de chamar foo
8
valor de b depois de chamar foo
9.000000
valor de myChar depois de chamar foo
G
→  etapa6 git:(master) x
```

Note que em foo as variáveis são recebidas e modificadas corretamente

Printar uma variável char e uma posição de vetor

```
i = int: 0FA;

vec = char[3]: 47 47 47;
myChar = char: 47;

main () = int
{
    print vec[0]
    print myChar

};
```

```
..mpiler/etapa6
→  etapa6 git:(master) ✗ ./a.out
71.000000
G
```

Note que variável char é printado seu caractere

Loop percorrendo vetor float e somando inteiro a cada elemento

```
→  etapa6 git:(master) ✗ ./a.out
vetor original inteiro iniciado com valores int (03 04 05)
3.000000
4.000000
5.000000
vetor após iterar sobre cada elemento e somar 0A
13.000000
14.000000
15.000000
```

```
i = int: 0FA;

vec = float[3]: 03.A 04.4 05.2;

main () = int
{
    print "vetor original inteiro iniciado com valores int (03 04 05)"
    loop(i:1, 4, 1)
    {
        print vec[i - 1]
    }

    loop(i:1, 4, 1)
    {
        vec[i-1] = vec[i-1] + 0A
    }

    print "vetor após iterar sobre cada elemento e somar 0A"
    loop(i:1, 4, 1)
    {
        print vec[i - 1]
    }

};
```

Note que se acessa, modifica e printa cada posição do vetor indexando o i do loop que apesar de adicionar inteiro o valor é salvo como float

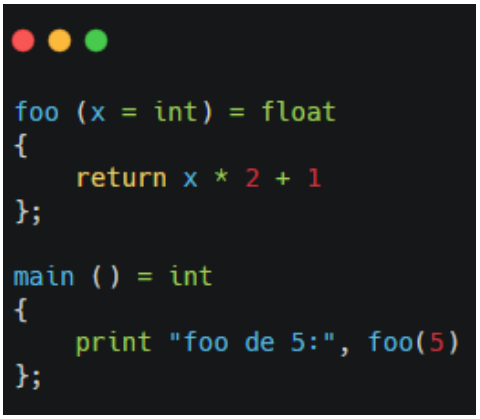
Dizer se os valores são pares ou ímpares

```
i = int: 4;
resto = int: 4;

main () = int
{


    i = 0
    while ( i < 0A)
    {
        resto = i / 2
        if (resto * 2 == i) then
            print i, "eh par\n"
        else
            print i, "eh impar\n"

        i = i+1
    }
};
```

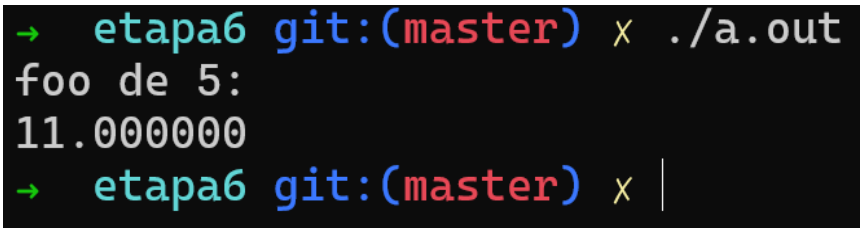
```
→  etapa6 git:(master) ✗ ./a.out
0
eh par

1
eh impar

2
eh par

3
eh impar

4
eh par

5
eh impar

6
eh par

7
eh impar

8
eh par

9
eh impar

→  etapa6 git:(master) ✗
```

Note que se faz um while i é menor que 10 e se aumenta manualmente o i e se usa if then else

Função com retorno de valor



```
foo (x = int) = float
{
    return x * 2 + 1
};

main () = int
{
    print "foo de 5:", foo(5)
};
```



```
→  etapa6 git:(master) ✗ ./a.out
foo de 5:
11.000000
→  etapa6 git:(master) ✗
```

Note que se printa direto o retorno da função, pois na chamada é colocado o retorno em uma variável temp