INF1018 - Software Básico (2023.1)
Segundo Trabalho

Um gerador de código para linguagem básica

O objetivo deste trabalho é desenvolver em C uma função chamada **gera**, que implementa um pequeno gerador de código (um "micro-compilador") para uma linguagem de programação bem básica, chamada *Simples*.

A função `gera` deverá ler um arquivo texto contendo o código fonte de uma função escrita em *Simples*, gerar o código de máquina que corresponde à tradução da função contida no arquivo na área passada no segundo parâmetro e retornar um ponteiro para a função gerada. A indicação é que a função main que chamar `gera` declare como variável local um vetor de unsigned char de tamanho apropriado e o passe como segundo parâmetro.

Instruções Gerais

Leia com atenção o enunciado do trabalho e as instruções para a entrega. Em caso de dúvidas, não invente. Pergunte!

- O trabalho deve ser entregue até a data agendada no EaD.
- Trabalhos entregues com atraso perderão **um ponto por dia de atraso**.
- Trabalhos que não compilem **não serão considerados**, ou seja, receberão grau zero.
- Os trabalhos podem ser feitos em grupos de dois alunos.
- Alguns grupos poderão ser chamados para apresentações orais / demonstrações dos trabalhos entregues.

A Linguagem *Simples*

O único tipo de dado de *Simples* é inteiro, com sinal, de 32 bits.

Variáveis locais são da forma **v***i*, sendo o índice *i* utilizado para identificar a variável (ex. `v1`, `v2`, etc...). A linguagem permite o uso de no máximo 5 variáveis locais.

Parâmetros de funções *Simples* são denotados por **p***i*, e podem ser usados no máximo 3 parâmetros (`p1`, `p2`, `p3`).

Constantes são escritas na forma **$***i*, onde *i* é um valor inteiro, com um sinal opcional. Por exemplo, **$10** representa o valor **10** e **$-10** representa o valor **-10**.

Funções *Simples* contém atribuições, operações aritméticas e instruções de desvio e de retorno.

- Uma atribuição tem a forma

```
var '<' varpc
```

  onde **var** é uma variável local e **varpc** é uma variável local, um parâmetro ou uma constante inteira.

Como exemplo, se temos

```
v1 < p1
v2 < $1
v3 < v2
```

o valor do parâmetro `p1` será armazenado na variável local `v1` e o valor inteiro `1` será armazenado nas variáveis locais `v2` e, consequentemente, em `v3`.

- Uma operação aritmética tem a forma

```
var = varc op varc
```

onde: **var** é uma variável local, **varc** é uma variável local ou uma constante inteira, e **op** é um dos operadores '`+`' , '`-`' ou '`*`'.

Como exemplo, se temos

```
v1 = v2 + $1
v4 = v2 * v3
```

o resultado da operação `v2 + 1` será armazenado na variável local `v1` e o resultado da operação `v2 * v3` será armazenado na variável local `v4`.

- A instrução de desvio tem a forma

```
'iflez' var n
```

  onde **var** é uma variável local e **n** é o número de uma linha no código fonte. A semântica dessa instrução é a seguinte:
  - se o valor da variável local é **menor ou igual a 0**, será executado um desvio para a instrução que está na linha **n**;
  - se o valor da variável local é **maior que 0**, não há desvio e a execução segue normalmente para a instrução da linha seguinte.

- Finalmente, a instrução de retorno tem a forma

```
'ret' varc
```

  Neste caso, a função deverá retornar, e seu valor de retorno é o valor da variável local ou constante inteira indicada.

A sintaxe da linguagem *Simples* pode ser definida formalmente como abaixo. Note que as cadeias entre ' ' são símbolos terminais da linguagem: os caracteres ' não aparecem nos comandos!

```
func   ::= cmd '\n' | cmd '\n' func
cmd    ::= att | expr | dif | ret
att    ::= var '<' varc
expr   ::= var '=' varc op varc
var    ::= 'v' num
varc   ::= var | '$' snum
varpc  ::= varc | 'p' num
op     ::= '+' | '-' | '*'
ret    ::= 'ret' varc
dif    ::= 'iflez' var num
num    ::= digito | digito num
snum   ::= [-] num
digito ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Alguns Exemplos

Veja a seguir alguns exemplos de funções *Simples*.

Note que os comentários não fazem parte da linguagem! Eles estão incluídos nos exemplos abaixo apenas para facilitar seu entendimento.

- O primeiro exemplo implementa uma função *f(x) = x + 1*

```
v1 < p1      // 1: obtem argumento
v1 = v1 + $1 // 2: soma 1
ret v1       // 3: retorna o resultado
```

- O exemplo a seguir implementa uma função que avalia se seu argumento é negativo

```
v1 < p1      // 1: obtem argumento
v1 = v1 + $1 // 2: soma 1 para poder testar <= 0
iflez v1 5   // 3: se menor ou igual a 0, desvia para linha 5
ret $0       // 4: retorna 0 (não é negativo)
ret $1       // 5: retorna 1 (é negativo)
```

- O próximo exemplo implementa uma função *f(x,y) = (x+y) * (x-y)*

```
v1 < p1      // 1: obtem primeiro argumento
v2 < p2      // 2: obtem segundo argumento
v3 = v1 + v2 // 3: calcula a soma
v4 = v1 - v2 // 4: calcula a diferença
v1 = v3 * v4 // 5: multiplica
ret v1       // 6: retorna o resultado
```

- A seguir, a função fatorial!

```
v1 < p1      // 1: obtem argumento
v2 < $1      // 2: inicializa valor do fatorial
v3 < $0      // 3: para poder fazer um desvio incondicional
iflez v1 0   // 4: termina o loop quando n == 0
v2 = v2 * v1 // 5: multiplica fatorial
v1 = v1 - $1 // 6: decrementa n
iflez v3 4   // 7: volta ao teste do loop
ret v2       // 8: retorna o valor do fatorial
```

Implementação e Execução

O que fazer

Você deve desenvolver em C uma função chamada **gera** que leia um arquivo de entrada contendo o código fonte de uma função na linguagem *Simples*, gere o código de máquina correspondente no vetor que é passado como segundo parâmetro, e retorne um ponteiro para a função gerada.

O arquivo de entrada terá no máximo 30 linhas, com um comando *Simples* por linha.

O protótipo de `gera` é o seguinte:

```
typedef int (*funcp) ();
funcp gera (FILE *f, unsigned char codigo[]);
```

O parâmetro **f** é o descritor de um arquivo texto, **já aberto para leitura**, de onde deve ser lido o código fonte da função escrita em *Simples*. Note que a função **não deve fechar o arquivo**! Esses protótipos estão definidos no arquivo `gera.h`, disponível AQUI.

Implementação

A função geraCodigo

A função **gera** armazenará o código gerado na região de memória passada como segundo parâmetro. O endereço retornado por **gera** será o endereço do início desta memória.

Para cada instrução *Simples*, imagine qual seria uma tradução possível para *assembly*. Além disso, lembre-se que a tradução de uma função *Simples* deve começar com o prólogo usual (preparação do registro de ativação, incluindo o espaço para variáveis locais) e terminar com a finalização padrão (liberação do registro de ativação antes do retorno da função).

O código gerado deverá seguir as convenções de C/Linux quanto à passagem de parâmetros e valor de retorno. As variáveis locais deverão ser alocadas na pilha de execução.

Para ler e interpretar cada linha da linguagem *Simples*, teste se a linha contém cada um dos formatos possíveis. Veja um esboço de código C para fazer essa interpretação AQUI. Lembre-se que você terá que fazer adaptações pois, dentre outros detalhes, essa interpretação **não será feita na função *main***!

Não é necessário fazer tratamento de erros no arquivo de entrada, você pode supor que o código fonte *Simples* desse arquivo **sempre estará correto**.

O código gerado por `gera` deverá ser um ***código de máquina x86-64***, e não um código fonte assembly. Ou seja, você deverá descobrir o código de máquina que corresponde às instruções de assembly que implementam a tradução das instruções da linguagem *Simples*. Para isso, você pode usar o programa `objdump` e, se necessário, uma documentação das instruções da Intel.

Por exemplo, para descobrir o código gerado por `movl %eax, %ecx`, você pode criar um arquivo meuteste.s contendo apenas essa instrução, traduzi-lo com o gcc (usando a opção -c) para gerar um arquivo objeto `meuteste.o`, e usar o comando

```
objdump -d meuteste.o
```

para ver o código de máquina gerado.

Estratégia de Implementação

Para desenvolver o seu programa, use a técnica de **TDD (Test Driven Design)**, na qual testes são escritos antes do código. O propósito é garantir ao desenvolvedor (você) ter um bom entendimento dos requisitos do trabalho antes de implementar o programa. Com isto a automação de testes é praticada desde o início do desenvolvimento, permitindo a elaboração e execução contínua de testes de regressão. Desta forma fortalecemos a criação de um código que nasce simples, testável e próximo aos requisitos do trabalho. Os passos gerais para seguir tal técnica:

- Escrever/codificar um novo teste
- Executar todos os testes criados até o momento para ver se algum falha
- Escrever o código responsável por passar no novo teste inserido
- Executar todos os testes criados até o momento e atestar execução com sucesso
- Refatorar código testado

Por exemplo:

1. Implemente a tradução de uma função *Simples* trivial, como o retorno de uma constante:

   ```
   ret $100
   ```

   Note que, neste momento, apenas a tradução da instrução `ret` é necessária!

   Crie uma função `main` e teste esse primeiro passo, chamando a sua função `gera`, chamando a função criada por ela, e imprimindo o valor de retorno da função gerada.

2. Continue implementando **e testando** uma instrução por vez. Experimente usar constantes, parâmetros, variáveis locais, e combinações desses tipos como operandos.

   Pense em que informações você precisa extrair para poder traduzir as instruções (quais são os operandos, qual é a operação, onde armazenar o resultado da operação).

   Lembre-se que é necessário alocar espaço (na pilha) para as variáveis locais!

3. Deixe para implementar a instrução de desvio apenas quando **todo o resto** estiver funcionando!

   Pense em que informações você precisa guardar para traduzir essas instruções (note que você precisa saber qual o endereço da instrução correspondente à linha para onde o controle deve ser desviado...)

Testando o gerador de código

Você deve criar um arquivo contendo apenas as funções `gera` e auxiliares (se for o caso) e **outro arquivo** com uma função `main` para testá-las.

Sua função *main* deverá abrir um ou mais arquivo texto que contém um "programa fonte" na linguagem *Simples* (i.e, uma função *Simples*) e chamar *gera*, passando o arquivo aberto como argumento juntamente com um ponteiro para a area onde deverá ser gerado o código de máquina. Em seguida, sua *main* deverá chamar a função retornada por *gera*, passando os argumentos apropriados.

Por exemplo:

```
#include "gera.h"

int main(int argc, char *argv[]) {
  FILE *myfp;
  unsigned char codigo[];
  funcp funcaoSimples;
  int res;

  /* Abre o arquivo fonte */
  if ((myfp = fopen("programa", "r")) == NULL) {
    perror("Falha na abertura do arquivo fonte");
    exit(1);
  }
  /* compila a função Simples */
  funcaoSimples = gera(myfp, codigo);
  fclose(myfp);

  /* chama a função */
  res = (*funcaoSimples) (...);  /* passando argumentos apropriados */
  ...
}
```

Não esqueça de compilar seu programa com

`gcc -Wall -Wa,--execstack -o testagera testagera.c gera.c`

para permitir a execução do código de máquina criado por `gera`!

Uma sugestão (não obrigatória!) para testar a chamada de uma função *Simples* com diferentes argumentos é sua função *main* receber argumentos passados na linha de comando. Para ter acesso a esses argumentos (representados por *strings*), a sua função `main` deve ser declarada como

```
int main(int argc, char *argv[])
```

sendo **argc** o número de argumentos fornecidos na linha de comando e **argv** um array de ponteiros para *strings* (os argumentos).

Note que o primeiro argumento para `main` (argv[0]) é sempre o nome do seu executável. Os parâmetros que deverão ser passados para a função criada por `gera` serão os argumentos de 1 em diante, convertidos para um valor inteiro (você pode usar a função **atoi** para fazer essa conversão).

Entrega

Deverão ser entregues **via Moodle** quatro arquivos:

1. Um arquivo fonte chamado **`gera.c`**, contendo as funções `gera` (e funções auxiliares, se for o caso).
   - Esse arquivo **não** deve conter a função `main`.
   - Coloque no início do arquivo, como comentário, os nomes dos integrantes do grupo da seguinte forma:

     ```
     /* Nome_do_Aluno1 Matricula Turma */
     /* Nome_do_Aluno2 Matricula Turma */
     ```

2. Um arquivo fonte chamado **`testagera.c`** contendo a função main e os testes das suas funções. Coloque o nome dos integrantes, como comentário, no início d arquivo.
3. Um arquivo ZIP com os códigos em Simples que você usou para testar o seu trabalho
4. Um arquivo texto, chamado **`relatorio.txt`**, contendo um pequeno relatório.
   - O relatório deverá explicar o que está funcionando e o que não está funcionando. Não é necessário documentar suas funções no relatório. Seu código deverá ser claro o suficiente para que isso não seja necessário.
   - O relatório deverá conter **alguns** exemplos de funções da linguagem *Simples* que você usou para testar o seu trabalho. Mostre tanto as funções *Simples* traduzidas e executadas com sucesso como as que não tiveram sucesso (se for o caso).
   - Coloque também no relatório o nome dos integrantes do grupo

Indique na área de texto da tarefa do Moodle o nome dos integrantes do grupo. Apenas uma entrega é necessária (usando o *login* de um dos integrantes do grupo) se os dois integrantes pertencerem à mesma turma.