versão online e atualizada:
https://docs.google.com/document/d/1lMq0m6PCE2fFnbUdGSWxaPlaQ4AffFuG3RwTDfLdWbM/edit?usp=sharing

/** *

- MAC0216 Técnicas de Programação I
- Alunos:
    - Eduardo Freire de Carvalho Lima 10262627
    - Eduardo Rocha Laurentino 8988212
    - Lais Baum 6837731
- Tarefa: Quarta fase
- Data: 13/12/2017

**/

QUARTA FASE:

Nessa fase, corrigimos alguns erros e adaptações das fases anteriores (linhas com mais de 80 colunas por exemplo) e fizemos a implementação de funções e da sintaxe requerida no Bison e no projeto para nossa versão final, além de termos realizado o tratamento de erro, identificando onde ele ocorre e mostrando-o ao usuário. Também fizemos alguns testes, em que estes foram anexados e submetidos com todos os nossos arquivos.

No Bison e Flex:

Implementamos a função ELSE, além de termos adaptado o IF para que a função AddInstr() esteja no formato do nosso projeto - isto é, recebendo um OPERANDO como argumento ao invés de um inteiro op. Adicionamos as funções especiais (MOV, ATK, etc) também conforme requerido pelo professor.

Com isso, os arquivos compila.l (flex) e compila.y (bison) estão integrados ao nosso Sistema de modo a permitir a programação dos robôs em alto nível. A sintaxe que definimos segue, basicamente, a ideia de ação.valor (exemplo: move.leste).

Para o **tratamento de erro,** como a especificação fornecida no enunciado é vaga, decidimos tão somente implementar na integração flex+bison um contador (yylineno) que indicará para o jogador não só que há um erro, mas em que linha do código isto ocorre. Fizemos isso adicionando a impressão deste contador na função de impressão de erro já dada.

No Projeto:

Adaptamos nossas funções e alguns métodos para ficarem de acordo com o desejado pelo professor: a função ATR está funcionando corretamente, desempilhando uma célula e acessa o atributo desejado, empilhando esse atributo. Além disso, padronizamos o uso do OPERANDO (antes, utilizávamos tanto op.val.n como op.valor, agora está tudo op.val.n) a fim de evitar conflitos e melhorar o entendimento do projeto.

Nos Testes:

Escrevemos alguns exemplos de códigos, em que testamos a movimentação de um robô com while, if etc. Nossa sintaxe é: ação.direção ou celula.direção.atributo. Por exemplo: move.leste irá movimentar o robô para a direção leste. Extrai.noroeste: tenta extrair um cristal da posição noroeste do robô. Célula.nordeste.base e celula.sudoeste.cristais: checa a célula do nordeste e do sudeste e verifica se é uma base e a quantidade de cristais presentes nela respectivamente.

CONCLUSÃO:

Com isso, tentamos integrar o projeto às mudanças de última hora feitas pelo professor (com a introdução da lógica de frames, por exemplo) e conectando-o com os analisadores léxico e semântico de alguma forma. Mas o grande foco desta fase foi a parte bison e flex em si, dado que entregamos até a terceira fase um projeto completo e funcional.

TERCEIRA FASE:

Nessa fase, corrigimos alguns pequenos erros da fase anterior, alteramos características da arena para facilitar a visualização, implementamos novos comandos nas chamadas de sistemas para criar o desenho e criamos uma função *main* na própria arena para que ela controle todo o jogo, além de completar o código em python para que mais características da arena pudessem ser adicionadas no desenho.
Todas as funções do nosso programa estão executando corretamente, mas para demonstrar o ataque, ou a extração e depósito de cristais no estágio atual precisaríamos de um programa com uma estratégia para os robôs, já que os cristais são criados aleatoriamente pela arena. Como não é esse o foco dessa fase, deixamos possível apenas ver toda a criação da arena com a inserção e movimentação dos robôs com velocidades diferentes dependendo do terreno onde ele está.

Modificações da arena:
Fixamos o tamanho da arena em 15x15 células e o local da célula-base para cada exército, sendo uma na posição (2,2) e outra na posição (9,9), com os robôs sendo adicionados em

células adjacentes. Os cristais ainda são criados em locais aleatórios da arena e a quantidade criada continua dependendo do tipo de terreno da célula.
Na fase anterior, nossa arena hexagonal tinha orientação vertical (flat-topped) e agora adequamos a posição das células para que a orientação seja horizontal (pointy-topped), o que a deixa de acordo com a representação disponibilizada em *apres*.
Escolhemos as seguintes cores para representar os terrenos:
Azul escuro - Base do Exército 0
Vermelho - Base do Exército 1
Azul claro - representa terreno AGUA
Cinza escuro - representa terreno MONTANHA
Marrom - representa terreno LAMA
Laranja - representa terreno TERRA
Cinza claro - representa terreno ESTRADA

Observação:
Célula Rosa - Indica terreno com cristal, mas não informa a quantidade. Uma célula rosa pode ter somente 1 cristal mas também pode ter mais e, portanto, o jogador precisa bolar sua estratégia considerando isso. Se a pessoa desejar saber o tipo de terreno de uma célula que há cristais, ela precisa usar o comando ATR.
No caso das bases, à partir do momento que for colocado o primeiro cristal nela, ela ficará com seu contorno (e apenas o contorno) cor de rosa.

Modificações do robô:
Implementamos 3 novos variáveis a ele: 2 tipos de arma e os atributo “saúde” e “isCiclo”.
Nossa variável arma é representada por um vetor de inteiros de 4 espaços, em que:
-arma[0] = armazena energia necessária para usar a arma0;
-arma[1] = armazena o quanto de saúde será retirado do oponente com a arma0;
-arma[2] = armazena energia necessária para usar a arma1;
-arma[3] = armazena o quanto de saúde será retirado do oponente com a arma1;
Já o atributo saúde indica o quanto de vida um robô possui, sendo inicializada com 1000 pontos. Ao chegar em 0, esse robô morre.
O atributo isCiclo é representado por uma variável int, indicando se o robô está ocupado realizando alguma tarefa. Existem tarefas descritas no nosso código em que o robô consegue realizar de imediato, outras ele necessita de 1 turno para realizá-la, podendo esse número chegar a 2 turnos.

Desenho do jogo:
Existem quatro protocolos em *apres* para a criação do desenho:

- Atualização da célula:
  "cel" coordenada_x coordenada_y terreno cristais base
- Inserção do robô:
  "rob" imagem registro posicao_x posicao_y

- Movimentação do robô:
  registro x_antigo y_antigo x_novo y_novo
- Fim do jogo:
  "fim"

O protocolo de atualização da célula foi criado para receber de uma vez todas as informações relacionadas a uma célula, como sua posição na arena, seu tipo de terreno, quantos cristais existem ali e também se há uma base e de qual exército seria, atualizando as informações que existem na matriz arena. Após a atualização da matriz, a função draw da célula é chamada. Se há cristais, independentemente da quantidade, a célula hexagonal ficará com o contorno rosa. Se não há cristais, o contorno fica preto. Para cada tipo de terreno também foi determinada uma cor para a célula. Porém, se há uma base ali, a cor será relacionada a um dos exércitos e não ao terreno.

O protocolo de inserção do robô continua colocando um Robô novo e sua imagem no vetor robs, mas, além disso, agora também já desenha esse mesmo robô na arena, na posição que foi recebida.

O protocolo de fim do jogo e o de movimentação do robô não foram modificados do arquivo que foi recebido para a implementação dessa fase.

Chamadas de sistema:

Ao iniciar a arena: a informação de cada célula (posição, terreno, cristais e base) é enviada para *apres*, que atualiza o desenho da arena.

Inserção de exércitos: quando um novo exército entra no jogo, a informação da célula-base é enviada para *apres* e seu desenho atualizado. Depois, para cada robô, são enviadas informações como o nome do arquivo da imagem específica do robô daquele exército (já que temos duas imagens diferentes, uma para cada time) e a posição onde ele está sendo inserido e que é a célula onde deve ser desenhado.

Remoção de exércitos: quando um robô morre, a informação da célula onde ele estava é enviada para *apres*, que então é redesenhada apagando a representação desse robô. Na função de remoção do exército, além do mesmo ocorrer com cada robô do time, acontece também com a célula-base, que quando é redesenhada perde a representação da base que estava ali.

Ao destruir a arena: é enviado um comando *fim*, que fecha o desenho.

Movimentação: a cada movimento do robô é enviado um protocolo para *apres* com o seu número de registro (que serve para identificar qual das imagens que deve ter sua posição alterada no desenho da arena), suas coordenadas antigas e suas coordenadas novas.

Assim, *apres* redesenha a célula na posição antiga do robô e o desenha em sua posição nova.

Extração e depósito de cristais: a extração e depósito de cristais pode ocorrer em células adjacentes ao robô ou na própria célula onde ele se encontra. Em geral, após a execução do comando, é enviada para *apres* a informação da célula onde o número de cristais foi alterado, e sua representação é atualizada. Se a extração foi feita na célula onde o próprio robô está, ele também é redesenhado, agora em cima da célula já atualizada.

Makefile e Main:
O Makefile dessa fase não gera mais testes específicos, ele tem somente a opção de criar um executável ./*arena*, que deve rodar o sistema completo através do *main* da própria arena.c.
O main cria uma variável display que servirá para enviar comandos para a entrada padrão de *apres*. Depois chama a função de criação da arena, que está declarada como uma variável global, cria 6 máquinas, sendo 3 para cada exército, e executa um ciclo de 10 rodadas do jogo, sendo que em cada rodada cada robô executa uma única instrução por vez.
Depois de executar as 10 rodadas o sistema destrói as máquinas e a arena, desalocando a memória e fecha a visualização do desenho.

As instruções de terminal para a execução do jogo são:

> make (ou make arena)
> ./arena

SEGUNDA FASE:
Nessa fase nós corrigimos os problemas na entrega da fase anterior:

- ALC e FRE implementadas
- RBP apontando para o valor correto
- Documentação bem feita

Além disso:

- Levantamos toda a estrutura da Arena e sua criação, o que inclui toda a parte de distribuição dos terrenos, cristais, registro de máquinas, inserção e

remoção de exércitos. Tudo isso foi feito dentro de alguns parâmetros de decisões de projeto que fizemos (os quais explicamos a seguir), mas está completo, funcional e testável.

- Algumas modificações e adaptações em relação à fase anterior (por exemplo, o Operando que antes era um inteiro e agora é uma struct). Para que tudo isso funcione, fizemos todas as alterações necessárias nas estruturas das máquinas virtuais bem como no método de execução de instruções destas (exec_maquina).
- Construímos o escalonador conforme especificado e em consonância com as nossas decisões de projeto. Junto a ele levantamos alguns métodos para verificação de máquinas e exércitos ativos. O escalonador está completo e testável desde que para um número reduzido de rodadas (até 10 rodadas).
- Construímos a instrução ATR e seu respectivo case em exec_maquina, possibilitando a consulta à informações das células através do tratamento correto de empilhamento e desempilhamento, conforme especificado no pdf.
- Levantamos toda a estrutura de chamada ao sistema, com solicitações de movimentação na arena, extração de cristais, depósito de cristais e ataque de um robô a outro. Para tudo isso adicionamos o conceito de “energia” aos robôs. Os detalhes estão explicados abaixo. A chamada ao sistema está, dentro do que fizemos, completa e testável.

Arena:

A arena contém o vetor de registros das máquinas que estão no jogo, que são 6 no total, sendo 3 para cada exército. Muitas das informações de cada máquina não são acessadas diretamente pelo seu endereço, mas por meio de um número de registro, que então indica a posição do seu endereço neste vetor de registros.
Como a matriz da arena é alocada dinamicamente, seria possível escolher um tamanho para ela no começo do jogo dependendo do número de exércitos que participariam. Mas com o objetivo de simplificar os testes nesta fase, decidimos inserir somente 2 exércitos e manter a arena com o tamanho e algumas outras características fixas.
Existem itens como a posição da base de cada exército e as células que contém cristais que são gerados aleatoriamente, e são as características que sempre variam na nossa arena. As máquinas são inseridas em células adjacentes a base, então também sempre iniciam o jogo em um local novo.
Cada célula contém suas próprias informações, incluindo as especificadas no projeto, como terreno, ocupação, número de cristais, e também outras que sentimos a necessidade de implementar para que o jogo pudesse funcionar corretamente, como qual a máquina que está ocupando a célula, se é uma base e de qual exército e também suas próprias coordenadas na arena.
Além das células, a arena guarda os dados de cada exército que foi inserido, com as informações dos robôs que o integram, a posição da sua base e se ainda tem algum robô ativo na arena.

Decisões de projeto:

- A arena é construída com alocação dinâmica de memória e o caráter hexagonal é obtido simplesmente através das considerações de vizinhança que são feitas em cima de uma matriz regular.
- Uma partida é composta por dois exércitos de 3 robôs cada
- Cada exército tem uma célula base
- O jogo termina ou quando o número de rodadas chega ao fim, sem nenhum vencedor, ou quando um exército é derrotado (levando o outro à vitória). A derrota de um exército pode tanto ocorrer se a base deste receber 5 cristais do exército rival quanto se a energia de todos os seus robôs chegarem a zero (robôs "morrem")
- Uma rodada é composta por um ciclo de execução de 50 instruções de cada robô, isto significa que em cada rodada um exército pode executar até 150 instruções (se tiver com todos os robôs ainda ativos)
- cabe ao jogador usar suas instruções para consultar informações sobre a Arena e suas células, por meio do case ATR, e ir descobrindo o terreno, a ocupação e onde se encontra os cristais e a base inimiga
- se um robô morrer (ficar com zero de energia), todos os cristais que estavam com ele ficam indisponíveis
- cada robô pode carregar até 3 cristais consigo, para impedir que o jogador use apenas um robô de seu exército de uma única vez para ir até a base inimiga depositar 5 cristais; a ideia é que o jogador seja forçado a de fato gerenciar o exército como um todo
- quase todo tipo de ação dos robôs leva a gastos energéticos. estes estão descritos abaixo

SISTEMA

Movimentação dos robôs (MOV):

- solicitar movimentação para o local onde já está equivale a um "descanso", aumentando a energia da máquina em questão; mas apesar de cada robô começar com 1000 de energia, não é possível voltar a ter 1000. O teto para reabastecimento energético é de 800, para aumentar a complexidade do jogo;
- só pode se movimentar para células desocupadas (e, obviamente, existentes)
- há um custo energético que depende do tipo de terreno para onde se está indo (estrada < terra < lama < água < montanha)

Extração de cristais (EXTR):

- extrações podem ser feitas tanto na célula em que se está quanto nas células vizinhas, desde que desocupadas
- extração tem um custo energético que depende do tipo de terreno (água < lama < terra < estrada < montanha)
- extração em célula vizinha tem um custo energético adicional em comparação a se fazer na própria célula em que se está. contudo este custo energético adicional é fixo e não depende do tipo de terreno. isso serve para que as estratégias levem em

consideração o uso do ATR para consulta do tipo de célula e decidam se vale mais a pena gastar energia indo até a célula extrair lá ou pagar pelo custo adicional de extrair pela vizinhança
- a extração só ocorre de fato de a célula tiver cristal disponível, mas o simples fato de tentar extrair já gasta energia (o que, mais uma vez, torna vantajoso fazer uso do ATR para consulta)
- são sempre unitárias; para extrair n cristais de um célula devem ser feitas n chamadas ao sistema;
- um robô só vai conseguir extrair se tiver menos de 3 cristais consigo, pois este é o limite que cada robô pode carregar. se tentar extrair já estando sobrecarregado não vai conseguir mas vai perder energia por tentar. isso obriga a usar o exército como um todo, não um único robô

Colocar cristais (POR):
- colocação de cristais podem ser feitas tanto na célula em que se está quanto nas células vizinhas, desde que desocupadas
- por tem um custo energético depende do tipo de terreno
(água < lama < terra < estrada < montanha)
- por em célula vizinha tem um custo energético adicional em comparação a se fazer na própria célula em que se está. contudo este custo energético adicional é fixo e não depende do tipo de terreno. isso serve para que as estratégias levem em consideração o uso do ATR para consulta do tipo de célula e decidam se vale mais a pena gastar energia indo até a célula por o cristal lá diretamente ou pagar pelo custo adicional de por pela vizinhança
- obviamente só pode por um cristal numa célula de fato se o robô em questão tiver cristal disponível, mas o simples fato de tentar extrair já gasta energia (o que, mais uma vez, torna vantajoso fazer uso do ATR para consulta)
- são sempre unitárias; para por n cristais numa célula devem ser feitas n chamadas ao sistema;
-

Ataque (ATK):
- atacar um adversário tem um custo energético fixo;
- ser atacado perde bastante energia
- se atacar uma célula vazia ninguém será atingido mas o atacante perderá energia do mesmo jeito (valoriza o uso do ATR para consultar se uma célula está ocupada ou não)
- direcionar um ataque para o própria célula em que se está é o equivalente a atacar a si mesmo, ou seja, perde energia por atacar e por ter sofrido ataque (serve como penalização para comandos estúpidos)
-

Verificação de continuidade do jogo e números de robôs ativos:

A verificação da continuidade do jogo se dá na checagem de energia de todos os robôs de todos os exércitos. No nosso caso, determinamos 2 exércitos e, por isso, ambos precisam possuir pelo menos 1 robô com energia > 0 para eles serem considerados “ativos”. Logo, dentro da função “verifica_continuidade()”, enviamos cada exército para a

função “verifica_exercito_ativo(Exercito exerc)” (ambos no arena.c), que recebe como parâmetro um determinado exército e faz uma varredura em todos os robôs desse exército. Aqueles que morreram afetam o estado da célula em que se encontravam, tornando-a desocupada (atributo “ocupado” = 0). Se há 1 robô vivo, a função retorna 1, indicando que o exército continua ativo e 0, caso contrário.

Na função “verifica_continuidade()”, após verificar se há algum exército inativo, ela retorna -1, caso o jogo permaneça ativo, 0, caso o jogo tenha acabado e o exército 0 tenha vencido, ou 1, caso o jogo tenha acabado o e exército 1 tenha sido o vencedor.

ATR:

O ATR é uma operação que uma máquina pode realizar. Sua função é desempilhar uma célula que está no topo da pilha de dados e empilhar um atributo específico dessa célula na mesma pilha de dados. Qual o atributo que é desejado é indicado pelo argumento da operação ATR, por ex: ATR 3. Logo, neste exemplo, uma determinada célula, que se encontra no topo da pilha de dados, será desempilhada e armazenada num operando chamado “tmp”. Em seguida, através de um switch, verificaremos qual o valor do argumento(arg.valor) e o que ele representa (no nosso programa, decidimos que 0 = terreno, 1 = cristais, 2 = ocupado e 3 = base) e armazenamos o valor desse atributo no valor de um operando temporário (op1.valor), além de igualarmos o tipo do operando que foi desempilhado com o tipo do operando temporário “op1” (op1.t = tmp.t, ambos célula). Por fim, precisamos empilhar esse atributo na pilha de dados, que só pode receber variáveis do tipo “operando”, dai fazemos “empilha(pil, op1)”, concluindo o ATR.

Makefile e Testes:

- O makefile do projeto foi modificado para gerar um executável ./arena, o qual é feito a partir do arquivo testearena.c, e um executável ./movimento, o qual é feito a partir do arquivo teste_sistema_e_escalonador.c.
- Como a saída dos testes é extensa, pois printamos muitas coisas, recomendamos que a saída seja enviada para um arquivo de texto, para que todo o resultado possa ser visualizado sem as limitações da janela do terminal (por exemplo fazendo ./arena > testearena.txt) depois de chamar o comando make que compila os testes.

Teste de Arena:

É feito pelo testearena.c. Nele é testado todo o processo envolvido na criação da arena, bem como dos robôs, exércitos, bases e cristais distribuídos nela. Printamos a arena fase a fase, indicando sua criação, depois a distribuição dos tipos de terrenos, depois as regiões em que há cristais (e a quantidade), bem como a base de cada exército e seus robôs, mostrando também que a arena realmente fica vazia após invocarmos o método de remoção dos exércitos. Reparem como os robôs de cada exército sempre nascem próximos às suas respectivas bases.

Durante esse processo é testado, portanto, toda a parte relacionada à construção da matriz que representa a arena bem como também é testada toda a parte relacionadas ao registro de máquinas virtuais na arena.

Obs.: Na saída do teste, printamos a arena indicando por . as células vazias (em relação aquilo que estamos mostrando, seja base, seja cristais, sejam robôs, etc). Apesar

de a impressão da arena se dar desta maneira, a dinâmica do jogo se dá com uma arena de fato hexagonal, já que apesar de usarmos a noção de matriz regular, nós fazemos todos os tratamentos de vizinhança diferenciado para que o aspecto hexagonal seja obtido (basta ver o código do nosso método de chamada ao sistema).

Instrução de terminal:

> make (ou make arena)

> ./arena > textearena.txt

e, então, ver o resultado no .txt gerado.

Teste de chamada ao Sistema e escalonador:

É feito pelo teste_sistema_e_escalonador.c. Nele usamos a requisição de movimentação como exemplo de chamada ao sistema (e, também, de uso do escalonador). Todos as demais chamadas ao sistema (para ataque, extração ou recolocação de cristais ou ataque) são feitos de maneira análoga. Nesse teste são criados dois exércitos (cada um com três robôs). Num primeiro momento é mostrado a posição destes robôs imediatamente após serem inseridos na arena (nascer) e logo depois as novas posições depois que cada um deles executou o seu programa de movimentação (após a chamada de escalonador). Há três programas de movimento, que foram distribuídos entre as máquinas. É possível verificar o sucesso das movimentações tanto pelo mapa impresso quanto pela sessão com informações dos robôs, onde se nota que as informações sobre as coordenadas de posição de cada robô se alteram coerentemente com o seu programa de movimentação. É possível ver também o consumo energético da movimentação ao notar que este valor é reduzido de cada robôs de acordo com as movimentações feitas (uns perderam mais e outros menos, a depender do tipo de terreno em que se movimentaram). Por padrão deixamos escalonador(a, 8), o que significa que serão executadas oito rodadas de movimentação para cada robô, sendo que em cada rodada há 10 instruções executadas. Para conferir movimentações menores, basta reduzir o número de rodadas (fazendo, por exemplo, escalonador(a, 1), onde cada máquina executará apenas um ciclo de movimentação).

Instrução de terminal:

> make movimento

> ./movimento > testesistema.txt

e, então, ver o resultado no .txt gerado.

Guia da modularização:

arena.h e arena.c são responsáveis pela implementação de tudo que envolve a Arena e chamadas ao sistema, sendo que a chamada ao sistema é uma possível instrução tratada no exec_maquina de maq.c. No maq.c também está o comando ATR para consulta às células. Em instr.h estão o novo OPERANDO (agora struct e não mais int) e a Célula (também struct). Nas demais partes, fizemos todas as adaptações necessárias para o bom funcionamento depois que mudamos os tipos de algumas variáveis.