UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

AUGUSTO TIMM DO ESPIRITO SANTO
EDUARDO STEIN BRITO
LUCAS DA SILVA FLORES

Entrega Parcial – Grupo Furogga
Projeto Frogger Utilizando Julia
(Funcional)

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2018

SUMÁRIO

1 INTRODUÇÃO

O objetivo deste trabalho consiste em estudar uma linguagem de programação moderna com características híbridas permitindo o aprendizado dos princípios de programação relacionados com os diferentes paradigmas estudados ao longo do semestre. Além disso, proporcionar a oportunidade de testar a capacidade de analisar e avaliar linguagens de programação, seguindo os critérios abordados em aula.

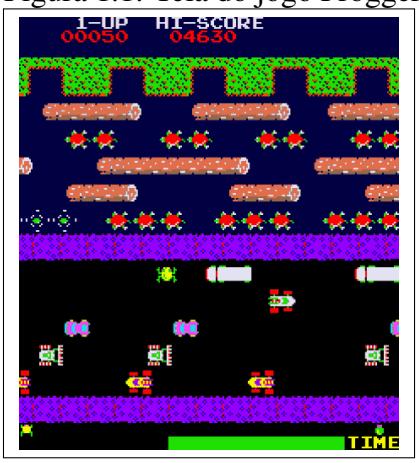
A tarefa principal do trabalho consiste em experimentar e comparar as características e funcionalidades orientadas a objeto e funcionais da linguagem de programação escolhida. De posse de uma linguagem, é necessário escolher um problema a ser solucionado com ela. O problema será, então, implementado duas vezes na mesma linguagem: uma delas usando somente Orientação a Objetos e a outra usando somente características funcionais.

A implementação funcional do trabalho e o relatório se encontram em <https://github.com/pacluke/frogger_furogga>.

1.1 Problema a Ser Resolvido

Este trabalho tem como problema a ser resolvido o desenvolvimento do jogo Frogger. Este é um jogo *arcade* que consiste em um personagem - um sapo - que deve atravessar uma estrada e um rio. O objetivo do jogo é atravessar estes lugares, desviando de obstáculos e por fim ocupar os espaços presentes no parte superior da tela, como visto na figura 1.1.

Figura 1.1: Tela do jogo Frogger



Fonte: http://www.classicgaming.cc/classics/frogger/about

2 VISÃO GERAL DA LINGUAGEM

A linguagem escolhida pelo grupo foi **Julia**. Esta é uma linguagem de programação de alto nível e alto desempenho para computação numérica (JULIA..., 2018). Ele fornece um compilador sofisticado, execução paralela distribuída, precisão numérica e uma extensa biblioteca de funções matemáticas.

2.1 Paradigma Funcional

Ao trabalhar com Julia focando no paradigma funcional, percebe-se que a linguagem possui bastante suporte para esse paradigma, mas não foi construída com o propósito de ser uma linguagem funcional. Ao mesmo tempo que a linguagem facilita para o usuário a criação de funções puras sem efeitos colaterais ela também dificulta na criação de estruturas de dados imutáveis e no suporte para cópia dessas estruturas, o que vai contra uma das principais características das linguagens funcionais.

3 RECURSOS FUNCIONAIS

Os recursos utilizados para a versão funcional do projeto da disciplina foram aplicados com base na documentação da linguagem (JULIA..., 2017) e nos estudos em (INTRODUCING..., 2017). Todas as funções referenciadas nessa seção estão disponíveis no arquivo *frogger.jl* em <https://github.com/pacluke/frogger_furogga/tree/entrega_parcial/Functional>.

3.1 Elementos Imutáveis e Funções Puras

Funções puras são funções que não causam efeitos colaterais. Em linguagens puramente funcionais, todas as funções são puras. Em Julia, funções que não são puras também são permitidas, mas quando uma função pura é criada, podemos explicitar utilizando essa macro antes da definição:

```
Base.@pure
```

Um exemplo de função pura explicitada utilizada no projeto é a função *change_tile* na linha 002:

```
Base.@pure function change_tile(y)
    if(y[1] > 2 && y[1] < 11)
            return "~"
    elseif(y[1] > 11 && y[1] < 20)
            return ":"
    else
            return "."
    end
end
```

change_tile foi criada para manter o cenário do jogo, representado por uma matriz, toda vez que o jogador se movimenta Ou seja, dependendo do "terreno"que o jogador está, a posição que ele estava é substituída por um caractere diferente.

3.2 Funções Anônimas

Funções anônimas são funções que não estão vinculadas a um identificador. Além disso, funções anônimas geralmente são Funções de Ordem Superior, como é também no caso da linguagem Julia. Seguindo esta ideia, o projeto possui o uso de funções anônimas em dois momentos diferentes.

Na linha 057, dentro da função *move_frog*:

```
position = find(x -> x == "W", m)
```

Nesse caso, uma função anonima é passada como parametro para a função *find* que devolve a posição do elemento "W"dentro da matriz *m*.

O segundo caso, na linha 111, funciona de forma parecida, mas dessa vez ela assume uma função definida pelo usuário:

```
m = map(x -> replace_matrix(eval_things(x), x),
                                readdlm("map3.txt"))
```

3.3 Currying

A ideia de *currying* é transformar uma função vários argumentos em uma cadeia de funções onde cada uma tem apenas um argumento.

A utilização desse recurso em Julia é bastante direta, como se percebe na função *eval_things* na linha 022:

```
Base.@pure function eval_things(value01)
    equal(value02) = (value01 == value02)
    return equal
end
```

Essa função retorna uma função que compara um valor com o valor utilizado anteriormente como parâmetro de *eval_things*. Para exemplificar o uso dessa função, podemos voltar para o exemplo da linha 115:

```
global m = map(x -> replace_matrix(eval_things(x), x),
                    readdlm("map3.txt"))
```

Onde para cada elemento da matriz, é retornada uma função que possui como parâmetro para comparação o elemento atual.

3.4 Pattern Matching

Pattern matching permite que padrões simples sejam encontrados em valores e estruturas de dados como listas. Julia não possui por padrão uma implementação de *pattern matching* ou algo parecido, e o máximo encontrado pelo grupo após pesquisas foi uma biblioteca que não é oficial e que não funcionou a versão de Julia utilizada no trabalho (0.6.2). Por esse motivo o grupo concordou em não utilizar esse recurso no desenvolvimento da versão funcional do *Frogger*.

3.5 Funções de Ordem Superior

Funções de ordem superior são funções que recebem funções como parâmetro e/ou retornam funções. Um exemplo utilizado no projeto foi a função *replace_matrix*:

```
Base.@pure function replace_matrix(fun, ch)
    if(fun(1)) return ":"
    elseif(fun(2)) return "@"
    elseif(fun(3)) return "~"
    elseif(fun(4)) return "X"
    else return ch
    end
end
```

Ela recebe como parâmetro a função *fun* e um elemento *ch* e aplica a função *fun* aos números 1, 2, 3 e 4, caso contrário retorna *ch*. Isso serve para que uma matriz tenha os valores de 1 a 4 substituídos pelos seus respectivos caracteres, ajudando a formar a matriz que controla o jogo.

3.6 Funções de Ordem Maior Fornecidas Pela Linguagem (Map)

Julia fornece várias funções de ordem maior como *reduce*, *foldl*, *foldr*, entre outras. Foi escolhido pelo grupo a função *map*:

```
global m = map(x -> replace_matrix(eval_things(x), x),
                    readdlm("map3.txt"))
```

A função *map* aplica, para cada elemento da matriz **m**, uma função que substitui cada elemento da matriz (número) pelo seu caractere equivalente.

3.7 Funções como Elementos de Primeira Ordem

Funções são utilizadas como elementos de primeira ordem quando elas podem ser recebidas como parâmetro e também retornadas por outras funções, assim como já foi demonstrado na função *replace_matrix* que recebe uma função como parâmetro e na função *eval_things* que retorna uma função.

3.8 Recursão

A recursão foi utilizada em uma das funções mais importantes do projeto, que reimprime a matriz do jogo toda vez que a mesma é atualizada:

```
function print_map_rec(map, j, sizej, i, sizei)
    # print da matriz
    if(map[j, i] == "~")
        print_with_color(:cyan, map[j, i])
    elseif(map[j, i] == ":")
            print(map[j, i])
    elseif(map[j, i] == "@" || map[j, i] == "X")
            print_with_color(:red, map[j, i])
    elseif(map[j, i] == "^")
            print_with_color(:magenta, map[j, i],
            bold=true)
    elseif(map[j, i] == "W")
```

```
            print_with_color(:green, map[j, i],
            bold=true)
        else
            print_with_color(:white, map[j, i])
        end

        # condicoes da recursao
        if (i+1 > sizei && j+1 > sizej) print("\n\r")
            return 0
        elseif(i+1 > sizei)
            print("\n\r")
            print_map_rec(map, j+1, sizej, 1, sizei)
        else print_map_rec(map, j, sizej, i+1, sizei)
        end
    end
```

A função *print_map_rec* é encontrada na linha 82 do arquivo *frogger.jl*.

4 CONCLUSÃO PARCIAL

Julia não é uma linguagem de programação funcional, no entanto a mesma fornece alguns recursos de linguagens funcionais, como por exemplo funções anônimas. Mesmo não sendo uma linguagem ideal para a programação funcional, foi possível desenvolver parcialmente o jogo de forma suficientemente satisfatória.

Será feita ainda a implementação do mesmo jogo utilizando uma abordagem orientada a objetos e com isso poderemos realizar um comparativo não só entre as abordagens mas também na versatilidade desta linguagem no uso de diferentes paradigmas.

REFERÊNCIAS

INTRODUCING Julia / Functions. 2017. Available from Internet: <https://en.wikibooks.org/wiki/Introducing_Julia/Functions>.

JULIA Documentation. 2017. Available from Internet: <https://docs.julialang.org/en/stable/>.

JULIA Language. 2018. Available from Internet: <julialang.github.com/index.md>.