Linguagens de Programação - CCF 340
Trabalho Prático - Parte 1

Igor Lucas (3865)[1], Lázaro Izidoro (3861)[2], Pedro Cardoso (3877)[3]

[1]Universidade Federal de Viçosa (UFV) - Campus Florestal

1. Introdução

O objetivo deste trabalho prático é utilizar os conhecimentos adquiridos durante a disciplina para investigar decisões de projeto e características de uma linguagem de programação. Neste texto, a linguagem de programação a ser estudada é a linguagem Pony.

O trabalho foi dividido em seções. As seções de 2 a 5 são referentes ao capítulo 1 do livro texto da disciplina, as quais comentam sobre o objetivo e a história da LP escolhida, as propriedades desejáveis dentro desta LP, o método de implementação e seu paradigma. As seções de 6 a 11 são referentes ao capítulo 2 e comentam sobre identificadores, tipo de escopo, definições e declarações (constantes, variáveis, tipos, subprogramas). Por fim, as seções de 12 a 15 são referentes ao capítulo 3 do livro, comentando sobre tipos primitivos e compostos, tipos ponteiro e Strings da LP. A seção 16 contém as considerações finais e a seção 17 traz as referências utilizadas para a confecção do trabalho.

2. Apresentação da Linguagem - Objetivo e História

A linguagem Pony é baseada na filosofia do "fazer". Seu **objetivo** é baseado na resolução de problemas de modo que sejam seguidos determinados critérios, os quais são: correção, desempenho, simplicidade, consistência e completude.

- **Correção:** Este critério diz que resoluções incorretas não são permitidas. É inútil tentar resolver um problema se não for possível garantir um resultado satisfatório.
- **Desempenho:** Para a linguagem Pony, o tempo de execução é muito importante. Quanto mais rápido o programa conseguir realizar soluções, melhor. No entanto, a velocidade não tem uma importância maior do que um resultado correto.
- **Simplicidade:** A simplicidade diz respeito ao quão rápido o programador consegue realizar as tarefas de forma eficiente. Não há problema em tornar as coisas mais difíceis para o programador para melhorar o desempenho, mas é mais importante tornar as tarefas mais fáceis para o programador do que para o tempo de execução.
- **Consistência:** O objetivo aqui é não permitir que a consistência excessiva se interponha no modo de realizar as tarefas.
- **Completude:** A completude refere-se ao quando realizar as tarefas. É melhor realizar algumas tarefas antecipadamente, não permitindo que tudo seja feito "em cima do prazo".

Em relação à **história** da linguagem Pony, em 2010, Sylvan Clebsch estava trabalhando em um simulador de voo e, posteriormente, em aplicativos distribuídos. Os mesmos problemas continuaram surgindo, então ele começou a trabalhar em um tempo de execução C para corrigi-los. Ele descobriu que embora a criação da biblioteca fosse

possível, era difícil ter certeza de que os clientes dessa biblioteca estavam cumprindo todas as obrigações que tornariam o código seguro e protegido.

Depois de fazer um pouco de pesquisa, ele mais tarde abandonou tal esforço e formou uma pequena equipe para mudar para uma nova linguagem. O Pony foi criado em 2014. Nessa época, uma equipe maior estava trabalhando com ele. As principais inovações foram em torno do modelo de coleta de lixo e alguns recursos de linguagem chamados recursos de referência e recursos de objeto. Esses recursos permitiram que Sylvan e a equipe Pony fornecessem cada vez mais garantias por parte do compilador.

3. Análise da LP em Relação às Propriedades Desejáveis

Nesta seção do trabalho serão discutidas e exemplificadas as propriedades desejáveis na linguagem de programação escolhida.

- **Legibilidade** - Indica a facilidade de leitura e entendimento do programa. Exemplo:

```
actor Main
  new create(env: Env) =>
    var x: String = "A maior que B"
    var y: String = "B maior que A"
    var a: U32 = 3
    var b: U32 = 22
    if a > b then
      env.out.print(x)
    else
      env.out.print(y)
    end

```

Figura 1. Exemplo para a propriedade de Legibilidade.

No exemplo acima, é possível verificar a facilidade do entendimento do programa. Começamos com o ator principal (actor Main), seguido pelo comando new create. Em seguida, tem-se as variáveis (var) dos tipos String e inteiras. Por fim, temos uma condicional realizada pelos comandos if e else, os quais já são bem conhecidos.

- **Redigibilidade** - Indica a quantidade de código necessário para expressar uma ideia ou lógica. Exemplo:

```
actor Main
  new create(env: Env) =>
    var a: U32 = 3
    var b: U32 = 22
    if a > b then
      env.out.print("A maior que B")
    else
      env.out.print("B maior que A")
    end
```

Figura 2. Exemplo para a propriedade de Redigibilidade.

Este é basicamente o mesmo exemplo que utilizamos para a propriedade da legibilidade. No entanto, ele foi reduzido, admitindo a propriedade da redigibilidade, ou seja, temos o mesmo programa com menos quantidade de código, que resolve o mesmo problema e que continua sendo de fácil entendimento.

- **Confiabilidade** - Expressa os mecanismos oferecidos pela LP para incentivar a construção de programas confiáveis. Exemplo:

```
actor Main
  new create(env: Env) =>
    var u: Bool = true
    var v: U32 = 0
    while (u and (v <= 9)) do
      env.out.print(v.string())
      v = v + 2
      if v == 6 then
        u = false
      end
    end
```

Figura 3. Exemplo para a propriedade de Confiabilidade.

Este trecho de código trata exceções, em que está utilizando um while que executa o laço enquanto a variável u (booleana) é verdadeira e enquanto v (inteiro) é menor ou igual que nove. A cada laço é somado + 2 na variável v e possui uma condição "if" se v é igual a 6 a variável u será considerada falsa e, consequentemente, o laço irá terminar.

- **Eficiência** - Indica como a implementação da LP possui impacto no uso de tempo de processamento, memória, dentre outros. Exemplo:

```
primitive ImprimirSona
  fun adicionar(a:U64, b:U64):U64 =>
    a + b

  actor Main
    new create(env: Env)=>
      env.out.print("2+2 = "+ImprimirSona.adicionar(2,2).string())
```

Figura 4. Exemplo para a propriedade de Eficiência.

Neste exemplo, é criada uma função do tipo "primitive" a qual possui 2 valores de 64 bits que estão nas variáveis a e b. Na função main os valores são convertidos em string para que seja possível imprimir a soma. Na figura a seguir é mostrado o resultado.

```
0.45.1-24ed8367 [release]
Compiled with: LLVM 13.0.0 -- Clang-10.0.0-x86_64
Defaults: pic=true

2+2= 4

Program ended.
```

Figura 5. Resultado da soma da figura anterior.

- **Ortogonalidade** - Indica a capacidade da LP em permitir ao programador combinar seus conceitos básicos sem produzir efeitos anômalos ou inesperados nessa combinação. Exemplo:

```
primitive Imprimir
  fun adicionar(a:U64, b:U64):U64 =>
    a + b

  fun multi(a: U64, b:U64):U64 =>
    a * b

  fun div(a:U64, b:U64):U64=>
    a / b


  actor Main
    new create(env: Env)=>
      env.out.print("2+2= "+Imprimir.adicionar(2,2).string())
      env.out.print("2*2= "+Imprimir.multi(2,2).string())
      env.out.print("2/2= "+Imprimir.div(2,2).string())

```

Figura 6. Exemplo para a propriedade de ortogonalidade.

A linguagem Pony possui uma boa capacidade de combinar conceitos básicos por meio de funções. No exemplo acima, são mostradas 3 tipos de funções em que é possível somar, multiplicar e dividir.

- **Modificabilidade** - Indica a facilidade para alterar o programa em função de novos requisitos sem a necessidade de modificações em outras partes. Exemplo:

```
primitive Imprimir
fun imprime():U32=>
var x:U32= 22
x

primitive ImprimirF
fun imprime():F32=>
var x:F32= 12.2
x

actor Main
  new create(env: Env) =>
    env.out.print(ImprimirF.imprime().string())
```

Figura 7. Exemplo para a propriedade de modificabilidade.

Na linguagem Pony é possível imprimir valores por funções como mostrado acima. Na primeira função(Imprimir) é possível imprimir números inteiros U32 (int na linguagem C), já na segunda função é possível imprimir números reais F32 (Float na linguagem C).

- **Reusabilidade** - Indica a possibilidade de reusar o mesmo código em mais de uma aplicação. Exemplo:

```
1  primitive Imprimir
2  fun imprime(): U32=>
3  var x: U32 = 38
4  x
5
6  actor Main
7    new create(env: Env) =>
8      env.out.print(Imprimir.imprime().string())
```

Figura 8. Exemplo para a propriedade de reusabilidade.

Na Linguagem Pony para ser possível imprimir algum valor é preciso usar a função imprimir onde a mesma converte o valor para uma string como mostra na imagem acima.

- **Portabilidade** - Indica a capacidade de ser compilado ou executado em diferentes arquiteturas (seja de hardware ou de software). O termo pode ser usado também para se referir a reescrita de um código fonte para uma outra linguagem de computador.

O Pony oferece suporte à integração com outros idiomas nativos por meio da Interface de Função Estrangeira (FFI). A biblioteca FFI fornece uma API estável e portátil e uma interface de programação de alto nível, permitindo que o Pony se integre facilmente às bibliotecas nativas.

Observe que chamar C (ou outras linguagens de baixo nível) é perigoso. O código C tem fundamentalmente acesso a toda a memória do processo e pode alterar qualquer uma delas. Este é um dos recursos mais úteis, mas também perigoso, da linguagem. Chamar um código C bem escrito e livre de bugs não terá efeitos nocivos em um programa. No entanto, chamar um código C com erros ou mal-intencionado ou chamar C incorretamente pode fazer com que seu programa Pony dê errado. Consequentemente, todas as garantias do Pony em relação a não travar, segurança de memória e correção simultânea podem ser anuladas chamando as funções FFI.

4. Método de Implementação da LP

O Pony é uma linguagem compilada, e não interpretada. Na realidade, é uma linguagem compilada antecipadamente (AOT), ao invés de uma linguagem compilada just-in-time (JIT). Nesse sentido, uma vez que o programa está construído, será possível executá-lo repetidamente sem a necessidade de um compilador ou máquina virtual. No entanto, obviamente, é necessário que o programa seja construído antes de poder ser executado.

Se estiver no mesmo diretório que o seu programa, é possível simplesmente colocar: **$ ponyc**. Isso diz ao compilador Pony que o seu diretório de trabalho atual contém

o seu código fonte e que é para realizar a compilação. Se o seu código-fonte estiver em outro diretório, poderá mostrar ao ponyc onde está utilizando os caminhos de cada um: **$ ponyc path/to/my/code**

5. Paradigma da LP

Pony é uma linguagem de programação orientada a objeto, modelo de ator. É orientado a objetos pois possui classes e objetos, como Python, Java, C++, e muitas outras linguagens.

É ator-modelo porque tem atores (semelhantes a Erlang, Elixir, ou Akka). Estes comportam-se como objetos, mas também podem executar código de forma assíncrona.

Uma classe é declarada com a palavra-chave **class**, e tem que ter um nome que comece com uma letra maiúscula, como o exemplo a seguir.

```
1  class AlunoUfv
```

Figura 9. Especificando uma classe com a linguagem Pony.

Uma classe é composta de **Campos**, os quais são exatamente como os campos em estruturas na linguagem C ou campos em classes em C++, Java ou Python. Há três tipos de campos: **var, let, e embed fields**. Um campo var pode ser atribuído mais de uma vez. Em contrapartida, um campo let é atribuído apenas no construtor. Vejamos o exemplo da classe da Figura anterior, mas agora com alguns campos.

```
1  class AlunoUfv
2    let name: String
3    var matricula: U64
```

Figura 10. Especificando campos em uma classe com a linguagem Pony.

Aqui, um AlunoUfv tem um nome, que é um String, e uma matrícula, que é um U64 (um inteiro não assinado de 64 bits).

Em seguida temos os **construtores**. Os construtores em Pony possuem nomes. Fora isso, eles são como os construtores em outras linguagens. Eles podem ter parâmetros, e sempre retornam uma nova instância do tipo. Como eles têm nomes, é possível ter mais de um construtor para um tipo. Os construtores são introduzidos com a palavra-chave new. Vejamos um exemplo, com base na classe criada anteriormente.

```
1  class AlunoUfv
2    let name: String
3    var matricula: U64
4
5    new create(name2: String) =>
6      name = "Pedro"
7      matricula = 3877
```

Figura 11. Especificando um construtor com a linguagem Pony.

Aqui temos um construtor que cria um AlunoUfv com o nome Pedro e matrícula 3877.

Agora vejamos um pouco sobre **funções**. As funções no Pony são como métodos em Java, C, C++, Ruby, Python, ou praticamente qualquer outra linguagem orientada a objetos. Eles são introduzidos com a palavra-chave **fun**. Elas podem ter parâmetros como os construtores têm, e também podem ter um tipo de resultado. Vejamos um exemplo:

```
class AlunoUfv
  let name: String
  var matricula: U64

  new create(name2: String) =>
    name = "Pedro"
    matricula = x

  fun realizarMatricula(): U64 => matricula
```

Figura 12. Especificando uma função com a linguagem Pony.

A função **realizarMatricula**, é bastante direta. Ela tem um tipo de resultado U64, e retorna **matricula**, a qual é um U64. O que pode soar um pouco diferente é que nenhuma palavra-chave de retorno foi utilizada. Isto porque o resultado de uma função é o resultado da última expressão na função, neste caso, o valor da matrícula do aluno.

Por fim, veremos agora um pouco sobre os atores. Um **ator** é semelhante a uma classe, mas com uma diferença crítica: um ator pode ter comportamentos. Um **comportamento** é como uma função, exceto que as funções são síncronas e os comportamentos são assíncronos. Em outras palavras, quando se chama uma função, o corpo da função é executado imediatamente, e o resultado da chamada é o resultado do corpo da função. Isto é como uma invocação de método em qualquer outra linguagem orientada a objetos.

Mas quando você chama um comportamento, o corpo da função não é executado imediatamente. Ao invés disso, o corpo do comportamento será executado em algum momento indeterminado no futuro. Um comportamento parece uma função, mas em vez de ser introduzido com a palavra-chave fun, ele é introduzido com a palavra-chave **be**.

Como uma função, um comportamento pode ter parâmetros. Ao contrário de uma função, ela não tem capacidade de receptor e não se pode especificar um tipo de retorno. Então, o que retorna um comportamento? Comportamentos sempre retornam **None**, como uma função sem tipo de resultado explícito, porque eles não podem retornar algo que eles calculam (já que ainda não executaram). Vejamos um exemplo utilizando atores.

```
actor Programador
  let name: String
  var anxietyLevel: U64 = 0

  new create(name': String) =>
    name = name'

  be drinkCoffee(amount: U64) =>
    anxietyLevel = anxietyLevel - amount.min(anxietyLevel)
```

Figura 13. Utilizando atores com a linguagem Pony.

Aqui temos um Programador que pode beber café de forma assíncrona para dimi-

nuir seu nível de ansiedade.

6. Identificadores da LP

Os **identificadores** são os nomes que são fornecidos para variáveis, tipos, funções e rótulos em um programa. Os nomes de identificadores devem ser diferentes na ortografia e nas maiúsculas e minúsculas em todas as palavras-chave.

Alguns identificadores da linguagem Pony são: **class, let, var, new**, dentre outros. Nos trechos de código a seguir é possível observar tais identificadores.

```
1  actor Main
2    new create(env: Env) =>
3      env.out.print("Hello, world!")
```

Figura 14. Alguns identificadores (palavras-chave) da linguagem Pony.

```
19  class AlunoUfv
20    let name: String
21    var matricula: U64
```

Figura 15. Alguns identificadores (palavras-chave) da linguagem Pony.

```
19  class AlunoUfv
20    let name: String
21    var matricula: U64
```

Figura 16. Alguns identificadores criados pelo programador (nomes de variáveis) na linguagem Pony.

```
30  primitive Imprimir()
31  fun imprime(): Bool =>
32  var u: Bool = true
33  u
```

Figura 17. Alguns identificadores (palavras-chave) da linguagem Pony.

7. Tipo de Escopo e Estrutura de Blocos

A linguagem de programação Pony possui um **Escopo Estático**, ou seja, existem blocos que são utilizados para delimitar partes do programa (ambiente de amarração), a fim de uma melhor organização. Nos exemplos dados, podemos observar que a palavra-chave end delimita o final dos blocos utilizados.

```
if x == y then
  env.out.print("são iguais")
else
  if x > y then
    env.out.print("x é maior")
  else
    env.out.print("y é maior")
  end
end
```

Figura 18. Delimitadores de bloco na linguagem Pony ao utilizar condicionais.

```
while count <= 10 do
  env.out.print(count.string())
  count = count + 2
end
```

Figura 19. Delimitadores de bloco na linguagem Pony ao utilizar repetições.

```
for weapon in ["Espada"; "Escudo"; "Arco"].values() do
  env.out.print(weapon)
end
```

Figura 20. Delimitadores de bloco na linguagem Pony ao utilizar repetições.

8. Definições e Declarações de Constantes

Constantes são valores que não se alteram ao longo do programa. Elas podem aparecer sob forma de um valor explícito ou um nome simbólico que representa um determinado valor ao longo do programa. Exemplos de definições e declarações de constantes na linguagem Pony:

```
primitive pi    fun apply(): F32 => 3.14
primitive const_aleatoria  fun apply(): U32 => 25
primitive euler   fun apply(): F32 => 2.71

type Constantes is (pi | const_aleatoria | euler)
```

Figura 21. Constantes na linguagem Pony.

9. Definições e Declarações de Tipos

Tipos de dados são combinações de valores e de operações que uma variável pode executar.

- **Definições de Tipos** - amarra identificadores à tipos sendo criados.

```
struct Interna
  var x: I32 = 0

struct Externa
  embed interna_embed: Interna = Interna
  var interna_var: Interna = Interna
```

Figura 22. Definição de Tipos na linguagem Pony.

- **Declarações de Tipos** - amarra identificadores à tipos previamente criados.

```
type Map[K: (Hashable box & Comparable[K] box), V]
  is HashMap[K, V, HashEq[K]]
```

Figura 23. Declaração de Tipos na linguagem Pony.

10. Definições e Declarações de Variáveis

Uma variável identifica um espaço na memória do computador, reservado para armazenar valores de um determinado tipo. Exemplos de definições e declarações na linguagem Pony:

- **Definições de Variáveis** - Uma definição instancia/implementa um identificador.

```
var x: String
var y: U32
var z: F32
var w: Bool
```

Figura 24. Definições de variáveis na linguagem Pony.

- **Declarações de Variáveis** - Uma declaração introduz um identificador e descreve seu tipo. Uma declaração é o que o compilador precisa para aceitar referências a um identificador em questão.

```
var x: String = "Hello"
var y: U32 = 1000
var z: F32 = 876.34
var w: Bool = true
```

Figura 25. Declarações de variáveis na linguagem Pony.

11. Definições e Declarações de Subprogramas

Subprogramas são trechos de programa que realizam uma tarefa específica. Exemplos: funções e funções recursivas. Vejamos alguns exemplos na linguagem de programação Pony:

- **Definições de Subprogramas** - Uma definição de subprograma consiste na descrição de seu cabeçalho e de seu corpo.

```
primitive Imprimir
  fun adicionar(a:U64, b:U64):U64 =>
    a + b


  actor Main
    new create(env: Env)=>
      env.out.print("20 + 30= "+Imprimir.adicionar(20,30).string())
```

Figura 26. Definições de subprogramas na linguagem Pony.

- **Declarações de Subprogramas**

```
primitive Imprimir
  fun adicionar(a:U64, b:U64):U64 =>
    a + b

  fun multi(a: U64, b:U64):U64 =>
    a * b

  fun div(a:U64, b:U64):U64=>
    a / b


  actor Main
    new create(env: Env)=>
      env.out.print("2+2= "+Imprimir.adicionar(2,2).string())
      env.out.print("2*2= "+Imprimir.multi(2,2).string())
      env.out.print("2/2= "+Imprimir.div(2,2).string())

```

Figura 27. Declaração de subprogramas na linguagem Pony.

12. Tipos Primitivos da LP

Os **tipos de dados primitivos** são os tipos básicos que devem ser implementados pelas linguagens de programação, Alguns deles são: números reais, inteiros, booleanos, caracteres e strings.

Além disso, um tipo de dado caracteriza um conjunto de valores, determinando a **natureza, o tamanho, a representação e a faixa de representação**. A natureza caracteriza o tipo representado (um caractere, um número inteiro, um número real ou uma cadeia de caracteres). O tamanho determina a quantidade de bits necessários para armazenar os valores do tipo. A representação determina a forma de como os bits armazenados devem ser interpretados. A imagem ou faixa de representação determina a faixa de valores válidos para o tipo.

Vejamos agora, um pouco sobre os tipos apresentados pela linguagem Pony.

- **Tipo Inteiro** - Corresponde a um intervalo do conjunto de números inteiros. Exemplo:

```
actor Main
  new create(env: Env) =>
    var x: U32 = 3
    var y: U32 = 9
```

Figura 28. Tipo Inteiro na linguagem Pony.

No trecho acima podemos ver que o tipo das variáveis x e y é inteiro de tamanho 32 bits, representado pelo U32.

- **Tipo Caractere** - Caracteres ocupam 1 byte (8 bits) na memória. Exemplo:

```
var x: String = "P"
var y = "X"
var z: String
z = "Y"
```

Figura 29. Tipo Caractere na linguagem Pony.

Aqui temos diferentes maneiras de declarar um caractere, podendo ou não utilizar a palavra-chave String.

- **Tipo Booleano** - Possui apenas dois valores, podendo retornar verdadeiro ou falso. Exemplo:

```
actor Main
  new create(env: Env) =>
    var u: Bool = true
    u
```

Figura 30. Tipo Booleano na linguagem Pony.

No trecho acima podemos ver que o tipo da variável u é booleana, representada pelo Bool.

- **Tipo Ponto Flutuante** - Modela números reais. Exemplo:

```
actor Main
  new create(env: Env) =>
    var x: F32 = 9.3
    x
```

Figura 31. Tipo de Ponto Flutuante na linguagem Pony.

No trecho acima podemos ver que o tipo da variável x é de ponto flutuante de tamanho 32 bits, representado pelo F32.

- **Tipo Enumerado** - Permitem que o programador defina novos tipos primitivos através da enumeração de identificadores dos valores do novo tipo. Exemplo:

```
1  primitive Peixe
2  primitive Gato
3  primitive Cachorro
4  type Pet is (Peixe | Gato | Cachorro)
```

Figura 32. Tipo de Enumeração na linguagem Pony.

No trecho acima temos uma enumeração. Por exemplo, imagine que queremos dizer que o pet de alguém possa ser um peixe, um gato ou um cachorro. Podemos então escrever da forma como mostrado na linha 4.

É notado que nas 3 primeiras linhas utilizamos a palavra chave **primitive**. Um primitivo é semelhante a uma classe, mas há duas diferenças críticas:

- Um primitivo não tem campos.
- Há apenas uma instância de um primitivo definido pelo usuário.

E a pergunta é: por que utilizar primitivos? Algumas situações são as seguintes:

- Como um **"valor de marcador"**: Por exemplo, o Pony frequentemente utiliza o primitivo **None** para indicar que algo não tem "nenhum valor".
- Como um tipo de **"enumeração"**: Ao ter uma união de tipos primitivos, você pode ter uma enumeração de tipo seguro. Mostraremos os tipos de união mais tarde.
- Como uma **"enumeração de funções"**: Como os primitivos podem ter funções, é possível agrupar funções em um tipo primitivo.

13. Tipos Compostos da LP

Um tipo composto pode ser construído em uma linguagem de programação a partir de tipos primitivos e de outros tipos compostos. Alguns exemplos são o **produto cartesiano, uniões, mapeamentos, conjuntos potência e tipos recursivos.**

Vejamos agora quais são os tipos compostos apresentados pela linguagem Pony.

- **Produto Cartesiano** - São combinações de tipos diferentes em tuplas. Exemplo:

```
1  struct Interna
2    var x: I32 = 0
3
4  struct Externa
5    embed interna_embed: Interna = Interna
6    var interna_var: Interna = Interna
```

Figura 33. Exemplo de produto cartesiano na linguagem Pony.

Os campos das structs na linguagem Pony são definidos da mesma maneira como eles são para as classes, utilizando **embed, let var**. Um campo **embed** é embutido em seu objeto pai, como uma estrutura C dentro de uma estrutura C. Um campo **var / let** é um ponteiro para um objeto alocado separadamente.

- **Uniões** - Consiste na união de valores de tipos distintos para formar um novo tipo de dados. Exemplo:

```
1 var x: (String | U32)
```

Figura 34. Tipo de União na linguagem Pony.

Aqui temos um exemplo de utilização de uma união, em que x pode ser uma string, mas também pode ser um inteiro de 32 bits.

- **Mapeamentos** - Tipo de dados cujo conjunto de valores corresponde a todos os possíveis mapeamentos de um tipo de dados S em outro T. Exemplo:

```
1 type Map[K: (Hashable box & Comparable[K] box), V] is HashMap[K, V, HashEq[K]]
```

Figura 35. Exemplo de mapeamento na linguagem Pony.

- **Conjuntos Potência** - Tipos de dados cujo conjunto de valores corresponde a todos os possíveis subconjuntos que podem ser definidos a partir de um tipo base.
- **Tipos Recursivos** - As funções recursivas no Pony podem causar problemas. De acordo com a documentação da linguagem, cada chamada de função em um programa adiciona um quadro à pilha de chamadas do sistema, que é limitada. Se a pilha "estourar", normalmente isso fará com que o programa trave. Este é um tipo de erro de falta de memória e não pode ser evitado pelo que a linguagem oferece. Uma maneira de evitar alguns problemas é utilizando a recursividade de cauda, na qual a chamada recursiva é a última instrução da função. Vejamos um exemplo utilizando fatorial:

```
1  fun fatorial_recursivo(x: U32): U32 =>
2    if x == 0 then
3      1
4    else
5      x * fatorial_recursivo(x - 1)
6    end
7
8  fun cauda_fatorial_recursivo(x: U32, y: U32): U32 =>
9    if x == 0 then
10     y
11   else
12     cauda_fatorial_recursivo(x - 1, x * y)
13   end
```

Figura 36. Tipo de recursividade na linguagem Pony.

14. Ponteiros e Referências na LP

Na figura a seguir temos dois construtores. Um que cria um novo Ponteiro nulo e outro cria um Ponteiro com espaço para muitas instâncias do tipo para o qual o Ponteiro está apontando.

```
struct Pointer[A]

  /*
  Um Pointer [A] é um ponteiro de memória bruta. Não tem descritor e,
  portanto, não pode ser incluído em uma união ou interseção, ou ser
  um subtipo de qualquer interface. A maioria das funções em um
  Pointer [A] são privadas para manter a segurança da memória.
  */

  new create() =>
    // Um ponteiro nulo.
    compile_intrinsic

  new _alloc(len: USize) =>
    // Espaço para len instâncias de A.
    compile_intrinsic
```

Figura 37. Utilização de ponteiros na linguagem Pony.

15. Tipo String na LP

Strings são valores que representam uma sequência de caracteres. A seguir temos algumas formas de criação de Strings.

```
actor Main
  new create(env: Env) =>
    var x: String = "Hello"
    var y = "Hello"
    var z: String
    z = "Hello"
```

Figura 38. Diferentes formas de criação de Strings.

Agora vejamos alguns métodos possíveis de se utilizar com Strings na linguagem Pony, tais como deixar uma string por completa em maiúscula ou minúscula; adicionar alguma string na string atual; contar a ocorrência de algum caractere na string; encontrar um determinado caractere na string; clonar a string; dentre outros.

```
actor Main
  new create(env: Env) =>
    try
      // construindo a string
      let string = "Hello"

      // Deixar a string em maiúsculo
      let str_upper = string.upper()
      // Fazer o reverso da string
      let str_reversed = string.reverse()

      // Adicionar " world" ao final da string original
      let str_new = string.add(" world")

      // Contar a ocorrência de L na String
      let count = str_new.count("l")

      // Encontrar o primeiro w
      let first_w = str_new.find("w") ?
      // Encontrar o primeiro d
      let first_d = str_new.find("d") ?

      // Realizar uma substring
      let substr = str_new.substring(first_w, first_d+1)
      // Clonar
      let substr_clone = substr.clone()

      // print a substring
      env.out.print(consume substr)
  end
```

Figura 39. Métodos utilizados com strings.

É importante salientar que há muitas outras aplicações de strings na linguagem Pony e as mesmas podem ser encontradas nas nossas referências.

16. Considerações Finais

Com este trabalho pudemos estudar uma linguagem nova, desde seus conceitos básicos até os conceitos que estão sendo dados na disciplina. Com isso, foi possível observar que o mundo das LPs é bem maior do que apenas aquelas linguagens que utilizamos normalmente.

17. Referências

1. Github, Ponylang. Disponível em: https://github.com/ponylang/ponyc Acesso em Dezembro de 2021.

2. Github, Pony_Documentation. Disponível em: https://github.com/aksh98/Pony_Documentation Acesso em Dezembro de 2021.

3. Deinfo, Tutorial do Pony. Disponível em: https://deinfo.uepg.br/~alunoso/2020/SO/PONY/index.html Acesso em Dezembro de 2021.

4. Ponylang.io, An Early History of Pony. Disponível em: https://www.ponylang.io/blog/2017/05/an-early-history-of-pony/ Acesso em Dezembro de 2021.

5. Docs.Microsoft, Identificadores C. Disponível em: https://docs.microsoft.com/pt-br/cpp/c-language/c-identifiers?view=msvc-170 Acesso em Dezembro de 2021.

6. Embarcados.com, Tipos de dados para uso em algoritmos. Disponível em: https://www.embarcados.com.br/tipos-de-dados/ Acesso em Dezembro de 2021.

7. inf.pucrs.br, Identificadores, Variáveis e Expressões. Disponível em: https://www.inf.pucrs.br/~gustavo/disciplinas/ppeia/material/progEngVB_conceitosBasicos.pdf Acesso em Dezembro de 2021.

8. ic.uff.br, Subprogramas. Disponível em: http://www2.ic.uff.br/ hcgl/subprograma.htm https://stdlib.ponylang.io/builtin-Pointer/ Acesso em Dezembro de 2021.

9. infoq.com, Pony, Actors, Causality, Types, and Garbage Collection. Disponível em: https://www.infoq.com/presentations/pony-types-garbage-collection/ Acesso em Dezembro de 2021.

10. tutorial.ponylang, Calling C from Pony. Disponível em: https://tutorial.ponylang.io/c-ffi/calling-c.html#get-and-pass-pointers-to-ffi Acesso em Dezembro de 2021.

11. ti-enxame.com, Qual é a diferença entre uma definição e uma declaração? Disponível em: https://www.ti-enxame.com/pt/c/qual-e-diferenca-entre-uma-definicao-e-uma-declaracao/967110315/ Acesso em Dezembro de 2021.

12. tutorial.ponylang, Recursion. Disponível em: https://tutorial.ponylang.io/gotchas/recursion.html Acesso em Dezembro de 2021.

13. tutorial.ponylang, Structs. Disponível em: https://tutorial.ponylang.io/types/structs.html#structs-are-classes-for-ffi Acesso em Dezembro de 2021.

14. grox.io, Pony Course. Disponível em: https://grox.io/language/pony/course#:~:text=Em\%202010\%2C\%20Sylvan,parte\%20do\%20compilador Acesso em Dezembro de 2021.

15. F. Varejão, Linguagens de Programação: Conceitos e Técnicas, Elsevier, 2004. Disponível em: http://www.inf.ufes.br/~fvarejao/livroLP.html Acesso em Dezembro de 2021.