Linguagens de Programação - CCF 340

Trabalho Prático - Parte 2

Igor Lucas (3865), Lázaro Izidoro (3861), Pedro Cardoso (3877)
Universidade Federal de Viçosa (UFV) - Campus Florestal

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 1 a 3 são referentes ao capítulo 4 do livro texto da disciplina, as quais comentam sobre gerência de memória principal e tratamento da LP no que diz respeito a variáveis e constantes, bem como se existe persistência ortogonal na linguagem estudada. As seções de 4 a 6 são referentes ao capítulo 5, as quais dizem respeito ao uso de comandos e expressões dentro da Pony. Por fim, as seções de 7 a 12 dizem respeito ao capítulo 6, comentando sobre parâmetros, TADs, dentre outros assuntos.

Desenvolvimento

1 - Armazenamento de Variáveis e Constantes em Memória Principal

A linguagem Pony não necessita de armazenar uma pilha para cada um de seus atores. Na maioria das linguagens de modelo de ator, os atores têm suas próprias pilhas. Por exemplo, em Go, os Goroutines têm um tamanho de pilha de 2 KB; Os processos Elixir/Erlang têm uma pilha e heap combinados de 1,2 KB.

Entretanto, os atores de Pony não têm suas próprias pilhas. Eles utilizam a pilha do thread do SO em que estão sendo executados. Isso traz um benefício de desempenho porque, os quadros de pilha não precisam ser preservados ao alternar entre os atores. Além disso, a memória não precisa ser desperdiçada para atores que não utilizam sua pilha.

No entanto, também existem algumas desvantagens. Ainda é necessário armazenar as variáveis em algum lugar, então agora elas vão para o heap. Isso não é tão ruim, já que em um sistema de modelo Ator sua “pilha” está realmente no heap.

Um exemplo para esse caso é mostrado na figura abaixo.

```
actor Main
  var x: I32

  new create(env: Env) =>
    x = 0
    for i in Range(1, 11) do
      x = x + i  // soma 1 a 10 em x
    end

    // gera um ator e chama seu comportamento
    Squarer.square(x, this)
```

Figura 1: Exemplo em Pony.

Observe que nada é feito após chamar a função square no ator Squarer! Isso ocorre porque não há como bloquear. Em vez disso, o ator Squarer deve executar o retorno de chamada para imprimir as mensagens, que veremos em breve. Vamos definir o Ator Squarer que realizará o comportamento:

```
actor Squarer
  be square(x: I32, main: Main) =>
    let newVal = x * x
    main.printResult(newVal)
```

Figura 2: Exemplo em Pony.

Na função square, passamos uma referência ao ator principal. Podemos pensar na referência como o PID do ator. Com o PID, podemos então enviar uma mensagem ao Ator principal, para que ele imprima o resultado. Vamos agora definir o *printResult():*

```
actor Main
  var x: I32

  be printResult(newVal: I32) =>
    _env.out.print("Original: " + x.string()
                   + ", New: " + newVal.string())
```

Figura 3: Exemplo em Pony.

É possível ver que conseguimos não ter uma pilha armazenando x em uma variável de membro Actor. Em Pony, se existe o desejo de manter a variável nas chamadas de método assíncronas, você precisa armazená-la explicitamente no Actor, o que significa que x é armazenado no heap e não na pilha.

2 - Armazenamento de Variáveis e Constantes em Memória Secundária

Variáveis e constantes necessitam ser armazenadas na memória secundária e também precisam ser recuperadas de lá. Nas linguagens de programação, também existe o desejo de utilizar mecanismos para facilitar a implementação de programas. A pretensão maior e final é oferecer LPs nas quais não exista qualquer distinção entre entidades transientes e persistentes.

Como visto em aula, um exemplo a ser considerado é o de leitura de arquivos. Em pony temos o seguinte código base:

```
use "files"

actor Main
  new create(env: Env) =>
    for file_name in env.args.slice(1).values() do
      let path = FilePath(env.root, file_name)
      match OpenFile(path)
      | let file: File =>
        while file.errno() is FileOK do
          env.out.write(file.read(1024))
        end
      else
        env.err.print("Error opening file '" + file_name + "'")
      end
    end
```

Figura 4: Exemplo base para leitura de arquivos em Pony.

3 - Persistência Ortogonal

A maioria das LPs fornece diferentes tipos para variáveis persistentes e transientes. Em C, por exemplo, uma variável transiente pode ser de qualquer tipo, mas uma variável persistente deve ser do tipo arquivo binário ou texto.

A persistência é “ortogonal” ou “transparente” quando é implementada como uma propriedade intrínseca do ambiente de execução de um programa. Um ambiente de

persistência ortogonal não requisita nenhuma ação específica de programas rodando nele para recuperar ou salvar seu estado.

Persistência não ortogonal requer que os dados sejam escritos e lidos do armazenamento usando instruções específicas num programa, resultando no uso de persistir como um verbo transitivo: Ao final, o programa persiste.

A vantagem do ambiente de persistência ortogonal é a simplicidade e menor propensão a erros nos programas. Na linguagem pony:

```
trait Named
  fun name(): String => "Bob"

trait Bald is Named
  fun hair(): Bool => false

 class Bob is Bald
```

Figura 5: Exemplo de operador unário em Pony.

Na figura acima, mostra um exemplo de como a mesma classe pode ser usada de diferentes formas, onde Bob é uma classe e tem acesso a *hair()* e *name()* que são suas características.

4 - Existência ou não das Expressões Apresentadas no Capítulo 5

Uma expressão é uma combinação de valores, variáveis, operadores, e chamadas de funções. Elas são avaliadas e produzem (retornam) um valor.

Os tipos de notação de expressões podem ser prefixadas (o operador está antes dos operandos), infixadas (o operador está entre os operandos) ou posfixadas (o operador está após os operandos). Vejamos um exemplo de expressão infixada na linguagem Pony.

```
1  primitive Imprimir
2    fun adicionar (a:U64, b: U64):U64 =>
3      a + b
4
5
6  actor Main
7    new create (env: Env) =>
8      env.out.print("2+2= "+Imprimir.adicionar(2,2).string())
9
```

Figura 6: Exemplo com expressão infixada (soma) em Pony.

Pelas pesquisas realizadas, não foram encontrados métodos de uso para expressões prefixadas ou posfixadas na linguagem Pony. No entanto, em Pony, existem os operadores

unários. Adiante no trabalho será discutido mais sobre esses operadores. Um exemplo é o seguinte:

```
1  -x
2  x.neg()
```

Figura 7: Exemplo de operador unário em Pony.

Vejamos agora alguns dos outros tipos de expressões existentes na Pony.

- **Expressões Literais** - São expressões matemáticas que apresentam letras e podem conter números. No Pony você pode expressar booleanos, **tipos numéricos, caracteres, strings e arrays** como literais. Vejamos alguns exemplos:

```
1  let decimal_int: I32 = 1024
2  let hexadecimal_int: I32 = 0x400
3  let binary_int: I32 = 0b10000000000
```

Figura 8: Exemplo de expressões literais em Pony.

- **Expressões de Agregação -** Uma função de agregação é utilizada para construir valores compostos a partir de seus componentes. Exemplo: um vetor ou uma struct.

```
1  struct Valores
2    var x: F32 = 5.6
3    var y: I32 = 20
4    var z: U32 = 8
```

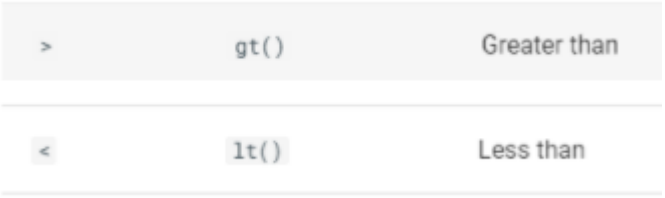
Figura 9: Exemplo de expressões de agregação em Pony.

- **Expressões Aritméticas -** Representa as operações matemáticas em que os operadores são aritméticos e os operandos são valores do tipo numérico (inteiro ou real). Vejamos um exemplo de expressão básica de divisão em Pony.

```
1  primitive Imprimir
2    fun div(a:U64, b:U64): U64=>
3      a / b
4
5  actor Main
6  new create (env: Env) =>
7  env.out.print("2/2= "+Imprimir.div(2,2).string())
8
```

Figura 10: Exemplo com expressão aritmética (divisão) em Pony.

- **Expressões Relacionais** - Usadas para comparar valores de seus operandos. Exemplos da sua utilização podem ser visualizados utilizando-se os símbolos de maior e menor.



>	gt()	Greater than
<	lt()	Less than

Figura 11: Métodos para o uso de maior ou menor em Pony.

```
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 12: Exemplo com expressão relacional em Pony.

- **Expressões Booleanas -** Realizam as operações de negação, conjunção e disjunção da álgebra de Boole. A seguir temos um exemplo utilizando uma variável booleana. Se a condição for "academia", então teremos "treino" (true), caso contrário, false. Claro que as expressões podem ser grandes. Aqui estamos apenas retornando um valor para true e outro para false, se for o caso.

```
var x: (String | Bool) =
  if academia then
    "Treino"
  else
    false
  end
```

Figura 13: Exemplo com expressão booleana em Pony.

- **Expressões Binárias -** Em Pony, uma expressão binária (decimal, hexadecimal, binária, etc.) pode ser representada conforme o exemplo a seguir.

```
let decimal_int: I32 = 1024
let hexadecimal_int: I32 = 0x400
let binary_int: I32 = 0b10000000000
```

Figura 14: Exemplo com expressão binária em Pony.

- **Expressões Condicionais -** Uma expressão condicional retorna um valor de verdadeiro, falso ou omisso para cada caso. Para isso, são feitos testes de verificação. Um exemplo clássico é saber se um valor qualquer é maior que outro valor. Em Pony temos este exemplo conforme a próxima figura.

```
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 15: Exemplo com expressão condicional em Pony.

- **Expressões com Chamadas de Funções -** São utilizadas funções no programa. O exemplo a seguir mostra uma classe que utiliza a chamada da função “hello”. Ao final, teremos uma mensagem “Hello + nome de alguém”.

```
class Alguem
  fun hello(nome: String): String =>
    "Hello " + nome

  fun f() =>
    let a = hello("Pedro")
```

Figura 16: Exemplo com expressão com chamada de função em Pony.

- **Efeito Colateral -** Em algumas LPs, é possível avaliar uma expressão que tenha efeito colateral de atualizar variáveis. Normalmente efeitos colaterais tornam o programa mais difícil de serem lidos e entendidos. Um exemplo que, inclusive, será discutido adiante nas expressões compostas, pode causar confusão por conta do uso dos operadores, conforme a seguir:

```
1 + 2 * -3  // Compilation failed.
```

Figura 17: Exemplo de possível efeito colateral em Pony.

- **Referenciamento -** Existem seis recursos de referência no Pony e todos eles têm definições e regras estritas sobre como podem ser usados. Vejamos cada um deles:

- **Isolado**, escrito **iso**. Esse recurso serve para referências a estruturas de dados isoladas. Se houver uma variável iso, saberá que não há outras variáveis que podem acessar esses dados.
- **Valor**, escrito **val**. Esse referencia as estruturas de dados imutáveis. Se existir uma variável val, saberá que ninguém pode alterar os dados.
- **Referência**, escrito **ref**. Aqui ocorrem as referências a estruturas de dados mutáveis que não são isoladas, em outras palavras, dados "normais". Se houver uma variável ref, será possível realizar leituras ou gravar os dados da maneira que desejar.
- **Caixa**, escrito **box**. Esse serve para referências a dados que são somente de leitura. A variável box pode ser usada para ler os dados com segurança. Isso pode parecer um pouco inútil, mas permite que você escreva código que pode funcionar tanto para variáveis val, como para variáveis ref.
- **Transição**, escrito **trn**. Esse é usado para estruturas de dados nas quais se deseja gravar, enquanto também mantém variáveis box para apenas para leituras. Também é possível converter a variável trn em uma variável val, o que impede qualquer pessoa de alterar os dados e permite que eles sejam compartilhados.
- **Etiqueta**, escrito **tag**. Esse serve para referências usadas apenas para identificação. Não é possível ler ou gravar dados usando uma variável tag. Mas é possível armazenar e comparar tags para verificar a identidade do objeto e compartilhar variáveis tag.

```
String iso
String trn
String ref
String val
String box
String tag
```

Figura 18: Exemplo com referenciamento em Pony.

- **Expressões Categóricas** - Realizam operações sobre tipos de dados.

```
1 var x: (String, U64)
2 x = ("Bom Dia", 3)
3 x = ("Até mais", 7)
```

Figura 19: Exemplo com expressão categórica em Pony.

- **Expressões Compostas -** Esse tipo de expressão sempre envolve mais de uma operação em seu corpo. A ordem de avaliação das operações pode influenciar no resultado obtido, ou até mesmo gerar erros. Vejamos um pouco sobre o funcionamento de expressões compostas em Pony nos exemplos a seguir.

Ao usar operadores infixos em expressões complexas, uma questão chave é a precedência , ou seja, qual operador é avaliado primeiro. Dada esta expressão:

```
1  1 + 2 * 3  // Compilation failed.
```

Figura 20: Exemplo com expressão composta em Pony.

Obteremos o valor 9 se avaliarmos a adição primeiro e 7 se avaliarmos a multiplicação primeiro. Em matemática, existem regras sobre a ordem na qual avaliar os operadores e a maioria das linguagens de programação segue essa abordagem. O exemplo acima é ilegal no Pony e deve ser reescrito como:

```
1  1 + (2 * 3)  // 7
```

Figura 21: Exemplo com expressão composta em Pony.

O uso repetido de um único operador, no entanto, não demonstra problemas:

```
1  1 + 2 + 3  // 6
```

Figura 22: Exemplo com expressão composta em Pony.

Outro ponto importante em expressões compostas na linguagem Pony é a possibilidade de se misturar operadores unários e infixos. Vejamos o mesmo exemplo dado na Figura 17 acima, utilizando essa observação:

```
1  1 + 2 * -3  // Compilation failed.
```

Figura 23: Exemplo com expressão composta em Pony.

Ainda precisaríamos dos parênteses para remover a ambiguidade para nossos operadores infixos como fizemos acima, mas não para o operador negativo aritmético unário:

```
1  1 + (2 * -3)  // -5
```

Figura 24: Exemplo com expressão composta em Pony.

Os operadores unários também podem ser aplicados aos parênteses e atuar no resultado de todas as operações nesses parênteses antes de aplicar quaisquer operadores infixos fora dos parênteses:

```
1  1 + -(2 * -3)   // 7
```

Figura 25: Exemplo com expressão composta em Pony.

- **Curto Circuito -** A avaliação de expressões em curto circuito ocorre quando o resultado é determinado antes que todos os operadores e operandos tenham sido avaliados.

5 - Existência ou não dos Comandos Apresentadas no Capítulo 5

Nas linguagens de programação os comandos possuem o objetivo de atualizar variáveis e controlar o fluxo de execução. Vejamos alguns exemplos em Pony dos comandos existentes na linguagem.

- **Comandos de Atribuição -** Define ou redefine o valor armazenado no local indicado por um nome de variável. Em Pony, os comandos de atribuição são os dois pontos (:) e o igual (=).

```
1  var x: String = "Uma string"
2
3  var y = "Uma string"
4
5  var z: String
6  z = "Uma string"
```

Figura 26: Exemplo com comandos de atribuição em Pony.

- **Comandos Condicionais -** Verificam condições. Se uma condição for verdadeira, um bloco ou trecho de código é executado. Caso contrário tal bloco ou trecho não será executado. Podemos ter apenas um caminho condicional (apenas um if), caminho duplo (vários “ifs”) ou múltiplos (exemplo do switch).

```
1  if a > b then
2    env.out.print("a é maior")
3  end
```

Figura 27: Exemplo de comando condicional de caminho simples em Pony.

```
1  if a == b then
2    env.out.print("São iguais")
3  else
4    if a > b then
5      env.out.print("a é maior")
6    else
7      env.out.print("b é maior")
8    end
9  end
```

Figura 28: Exemplo de comando condicional com caminho duplo em Pony.

Observações:

- ➢ Os comandos "else" e "if" que encontram-se nas linhas 3 e 4 respectivamente, poderiam ser escritos de forma simplificada como "elseif".
- ➢ Não foi encontrado um comando em Pony equivalente ao switch. O que precisaríamos fazer é utilizar vários caminhos condicionais com if e else, conforme mostrado no exemplo anterior de caminho duplo.

- **Comandos Iterativos -** A ideia de iteração é fazer com que uma série de comandos seja repetida enquanto uma condição definida for verdadeira. Temos dois casos de comandos iterativos, sendo eles os com pré-teste ou pós-teste. Vejamos um exemplo de cada na linguagem Pony.

```
1  var contador: U32 = 1
2
3  while contador <= 10 do
4    env.out.print(contador.string())
5    contador = contador + 1
6  end
```

Figura 29: Exemplo de comando iterativo com pré-teste em Pony.

```
1  actor Main
2    new create(env: Env) =>
3      var contador = U64(1)
4      repeat
5        env.out.print("Show")
6        contador = contador + 1
7      until contador > 7 end
```

Figura 30: Exemplo de comando iterativo com pós-teste em Pony.

Podemos observar que, da mesma forma que em outras linguagens, o “while” na Pony possui o mesmo funcionamento (pré-teste), enquanto que o comando “repeat until” é equivalente ao “do while” das outras linguagens (pós-teste).

- **Comandos de Repetição -** É um recurso que permite que certo trecho do código seja repetido um certo número de vezes. No exemplo a seguir, o trecho de código será repetido até que mostre todos os locais listados.

```
1 for local in ["Mercado"; "Shopping"; "Academia"].values() do
2   env.out.print(local)
3 end
```

Figura 31: Exemplo com comando de repetição em Pony.

- **Comandos com Desvio Irrestrito -** O desvio irrestrito, também é conhecido como “Goto”. Não foi encontrado na linguagem Pony um comando equivalente a ele. E, como comentado nas aulas da disciplina, nem todas as linguagens utilizam tal comando. Algumas até o excluíram.

- **Comandos com Escapes -** São comandos que podem interromper a execução de subprogramas ou programas. Exemplos: break e continue. Existem ambos comandos na linguagem Pony. Vejamos um exemplo utilizando o break.

```
1  var personagem =
2    while morePersonagens() do
3      var personagem' = getPersonagem()
4      if personagem' == "Mago" or personagem' == "Guerreiro" then
5        break personagem'
6      end
7      personagem'
8    else
9      "Arqueiro"
10   end
```

Figura 32: Exemplo com comando de escape “break” em Pony.

Às vezes se o programador deseja “parar” no meio de uma iteração em um loop e para passar para a próxima, então utiliza-se o comando **Continue**. Se o programador, no entanto, quiser parar no meio de um loop e desistir completamente dele, utiliza-se o comando **break**.

6 - Orientação à Expressões

A linguagem Pony pode ser considerada orientada à expressões, visto que seus comandos retornam algum tipo, a fim de demonstrar que em suas instruções podem existir efeitos colaterais. Algumas vantagens de linguagens orientadas à expressões são a obtenção de simplicidade (evitando duplicação de comandos e expressões), e uniformidade (elimina a separação das operações de controle de fluxo). Uma desvantagem é encorajar o uso de efeitos colaterais, visto que, se o programador não for muito experiente, pode vir a cometer erros, como por exemplo, de precedência de operadores, tendo visão de que seu código está correto, quando na verdade está equivocado.

7 - Correspondência entre Parâmetros Formais e Reais

Primeiramente, os **parâmetros reais** são os valores que são passados para a função quando ela é chamada. Já os **parâmetros formais** são as variáveis definidas pela função que recebe valores quando a função é chamada.

Vejamos nos exemplos a seguir códigos que utilizam esses tipos de parâmetros. A começar pelos parâmetros reais:

```
1  primitive Imprimir
2    fun adicionar(a:U64, b:U64):U64 =>
3      a + b
4
5
6    actor Main
7      new create(env: Env)=>
8        env.out.print("20 + 30= "+Imprimir.adicionar(20,30).string())
```

Figura 33: Exemplo com parâmetros reais em Pony.

Aqui podemos observar o uso dos parâmetros reais, pois seus valores (20 e 30) são passados para a função quando ela é chamada.

Agora vejamos para os parâmetros formais:

```
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 34: Exemplo com parâmetros formais em Pony.

Neste exemplo temos o uso dos parâmetros formais, pois o valor da variável (38) está definido pela função que irá recebê-lo quando a função for chamada.

8 - Direções de Passagem de Parâmetros Suportados

Em Pony, as direções de passagem de parâmetros suportadas são unidirecional e bidirecional pois as funções na linguagem são baseadas com métodos em Java, C#, C++, Ruby, Python ou praticamente qualquer outra linguagem orientada a objetos.

Funções em Pony são introduzidos com a palavra-chave **fun**. Elas podem ter parâmetros como os construtores, e também podem ter um tipo de resultado (se nenhum tipo de resultado for fornecido, o padrão será None).

```
class Wombat
  let name: String
  var _hunger_level: U64
  var _thirst_level: U64 = 1

  new create(name': String) =>
    name = name'
    _hunger_level = 0

  new hungry(name': String, hunger': U64) =>
    name = name'
    _hunger_level = hunger'

  fun hunger(): U64 => _hunger_level

  fun ref set_hunger(to: U64 = 0): U64 => _hunger_level = to
```

Figura 35: Exemplo de direções de passagem de parâmetros em Pony.

A primeira função, *hunger*, é bastante direta. Ele tem um tipo de resultado inteiro U64, e retorna *_hunger_level*, que é um inteiro U64. A única coisa um pouco diferente aqui é que nenhuma palavra-chave **return** é usada. Isso ocorre porque o resultado de uma função é dado na última expressão na função, neste caso, o valor de *_hunger_level*.

A segunda função, *set_hunger* possui uma *ref*, a qual é uma palavra-chave utilizada logo após a *fun*, ou seja, significa que o receptor no qual a função *set_hunger* está sendo chamada, deve ser um tipo ref. Um tipo ref é um tipo de referência, o que significa que o

objeto é mutável . Isso é necessário porque estamos escrevendo um novo valor para o campo do *_hunger_level*.

9 - Mecanismos e Momentos de Passagem de Parâmetros

Em Pony os mecanismos de passagem de parâmetros podem ser utilizados com ponteiros. Um Pointer[A] é um ponteiro de memória bruta. Ele não possui 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. Como no exemplo abaixo.

```
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 36: Exemplo com uso de ponteiros em Pony.

Aqui 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.

10 - Implementação de TADs

Assim como outras linguagens orientadas a objetos, Pony possui classes. A **classe** é composta por:

- ➢ Campos
- ➢ Construtores
- ➢ Funções

Um campo *var* pode ser atribuído repetidamente, mas um *campo let* é atribuído apenas uma vez no construtor. Exemplo:

```
class Wombat
  let name: String
  var _hunger_level: U64
```

Figura 37: Exemplo de classe em Pony.

Na classe *Wombat* temos um nome, que é uma *String* e a *_hunger_level*, que é um inteiro de U64(64 bits).

O pony também utiliza structs. Assim como as classes, as estruturas Pony podem conter campos e métodos. Ao contrário das classes, as estruturas Pony têm o mesmo layout binário que as estruturas C e podem ser usadas de forma transparente em funções C. Structs não possuem um descritor de tipo, o que significa que eles não podem ser usados em tipos algébricos ou implementar traits/interfaces.

Os campos de estrutura Pony são definidos da mesma forma que são para classes Pony, usando embed, lete var. Um campo de incorporação é incorporado 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. Um exemplo é mostrado na figura abaixo.

```
struct Inner
  var x: I32 = 0

struct Outer
  embed inner_embed: Inner = Inner
  var inner_var: Inner = Inner
```

Figura 38: Exemplo de struct em Pony.

11 - Uso de Pacotes e Compilação Separada de Arquivos

Para usar um pacote em um código na linguagem Pony, é necessário utilizar o comando ***use***. Isso dirá ao compilador para encontrar o pacote em questão e tornar os tipos definidos nele disponíveis. Todo arquivo Pony que precisa saber sobre um tipo de um pacote deve ter um comando ***use*** para ele.

Os comandos de uso são um conceito semelhante aos comandos “importar” de Python e Java, “#include” de C/C++ e “using” de C#, mas não exatamente iguais. Eles vêm no início dos arquivos Pony e se parecem com isso:

```
1 use "collections"
```

Figura 39: Comando **use** em Pony.

Esse comando encontrará todos os tipos visíveis publicamente definidos no pacote de coleções e os adicionará ao namespace de tipo do arquivo que contém o usecomando. Esses tipos ficam disponíveis para uso neste arquivo, como se fossem definidos localmente.

Por exemplo, a biblioteca padrão contém o pacote time. Este contém a seguinte definição (entre outras):

```
1  primitive Time
2    fun now(): (I64, I64)
```

Figura 40: Pacote Time da linguagem Pony.

Para acessar a função, basta adicionar um comando use:

```
1  use "time"
2
3  class Foo
4    fun f() =>
5      (var secs, var nsecs) = Time.now()
```

Figura 41: Uso de Pacotes em Pony.

A biblioteca padrão Pony é uma coleção de pacotes que podem ser usados conforme necessário para fornecer uma variedade de funcionalidades. Por exemplo, o pacote de **arquivos** fornece acesso a arquivos e o pacote de **coleções** fornece listas genéricas, mapas, conjuntos e assim por diante.

Há também um pacote especial na biblioteca padrão chamado **builtin**. Este contém vários tipos que o compilador tem que tratar especialmente e são muito comuns. Todos os arquivos de origem Pony têm um comando "builtin" implícito. Isso significa que todos os tipos definidos no pacote integrado estão automaticamente disponíveis no namespace. A documentação para a biblioteca padrão está disponível online em um dos links das referências.

Já em relação ao compilador da linguagem, vejamos como ele funciona. O **ponyc**, o compilador, geralmente é chamado no diretório do projeto, onde encontram-se os *.ponyarquivos* e suas dependências automaticamente. Lá, ele criará o binário com base no nome do diretório. É possível substituir isso e ajustar a compilação com várias opções conforme descrito em **ponyc --help** e é possível passar um diretório de origem separado como um argumento.

```
ponyc [OPTIONS] <package directory>
```

Figura 42: O compilador da linguagem Pony.

As opções mais úteis são **--debug, --path** ou apenas **-p**, **--output** apenas **-o, --docs** ou**, -g**.

- **--debug** irá ignorar os passos de otimizações do LLVM. Isso não deve ser misturado com make config=debug, o destino de configuração padrão do make.
- **--path** ou **-p** pega uma lista de caminhos separada como argumento e os adiciona aos caminhos da biblioteca em tempo de compilação, para que o vinculador encontre os pacotes de origem e as bibliotecas nativas, estáticas ou dinâmicas, sendo vinculadas em tempo de compilação ou via FFI em tempo de execução.
- **--output** ou **-o** recebe um nome de diretório onde o binário final é criado.
- **--docs** ou **-g** cria um diretório do pacote com documentação no formato Read the Docs, ou seja, markdown com boa navegação.

12 - Fortemente Tipada

Em linguagens fortemente tipadas, as operações entre valores de tipos diferentes, resultará em erros ou exceções, portanto essas operações serão dadas apenas por valores do mesmo tipo ou através da conversão (casting) de valores.

```
1  var x: String = "Olá"
2
3  var y = "Olá"
4
5  var z: String
6  z = "Olá"
```

Figura 43: Exemplo em Pony.

Na imagem acima mostra que a linguagem pony é fortemente tipada, pois é necessário informar o tipo de variável.

Considerações Finais

Com este trabalho pudemos continuar o estudo de uma linguagem nova, destacando seus conceitos intermediários e avançados conforme estão sendo dados na disciplina. Com isso, foi

possível observar que o mundo das LPs é bem maior do que apenas aquelas linguagens mais comuns que utilizamos.

Referências

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

[2] Wikipedia, **Expressão (Computação).** Disponível em: https://pt.wikipedia.org/wiki/Express%C3%A3o_(computa%C3%A7%C3%A3o)#:~:text=Uma%20express%C3%A3o%20em%20linguagens%20de,%2C%20produz%20(retorna)%20um%20valor Acesso em Dezembro de 2021.

[3] Pony Tutorial, **Operators**. Disponível em: https://tutorial.ponylang.io/expressions/ops.html Acesso em Fevereiro de 2022.

[4] Pony Tutorial, **Literals**. Disponível em: https://tutorial.ponylang.io/expressions/literals.html Acesso em Fevereiro de 2022.

[5] Pony Tutorial, **Variables**. Disponível em: https://tutorial.ponylang.io/expressions/variables.html#local-variables Acesso em Fevereiro de 2022.

[6] Pony Tutorial, **Control-structures**. Disponível em: https://tutorial.ponylang.io/expressions/control-structures.html Acesso em Fevereiro de 2022.

[7] Pony Tutorial, **Methods**. Disponível em: https://tutorial.ponylang.io/expressions/methods.html#functions Acesso em Fevereiro de 2022.

[8] Pony Tutorial, **Packages**. Disponível em: https://tutorial.ponylang.io/packages/use-statement.html Acesso em Fevereiro de 2022.

[9] Pony Tutorial, **Standard Library**. Disponível em: https://tutorial.ponylang.io/packages/standard-library.html Acesso em Fevereiro de 2022.

[10] Pony Tutorial, **Compiler Arguments**. Disponível em: https://tutorial.ponylang.io/appendices/compiler-args.html#runtime-options-for-pony-programs Acesso em Fevereiro de 2022.

[11] Pony Tutorial, **Classes**. Disponível em: https://tutorial.ponylang.io/types/classes.html Acesso em Fevereiro de 2022.

[12] Pony Tutorial, **Structs**. Disponível em: https://tutorial.ponylang.io/types/structs.html Acesso em Fevereiro de 2022.

[13] Pony Tutorial, **Files**. Disponível em: https://stdlib.ponylang.io/files--index/ Acesso em Fevereiro de 2022.

[14] Wikipedia, **Persistência**. Disponível em: https://pt.wikipedia.org/wiki/Persist%C3%AAncia_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o) Acesso em Fevereiro de 2022.

[15] Thomas' Weblog, **Pony actors don't have (their own) stacks** Disponível em**:** https://ttay.me/blog/pony_actors_no_stacks/ Acesso em Fevereiro de 2022.

[16] Inf.pucrs, **Passagem de Parâmetros.** Disponível em: https://www.inf.pucrs.br/~manssour/LinguagemC++/PassagemParam.pdf Acesso em Fevereiro de 2022.