Processamento de Linguagens (3º ano de Curso)

Trabalho Prático

Relatório de Desenvolvimento

Luís Pereira	Rui Oliveira	Tiago Pereira
(A96681)	(A95254)	(A95104)

28 de maio de 2023

Resumo

Neste projeto propõe-se Aeonius, uma extensão funcional da linguagem de programação Python. A implementação de Aeonius suporta as principais funcionalidades de uma linguagem funcional, mantendo total interoperabilidade com Python.

Conteúdo

Capítulo 1

Introdução

Supervisor: José Carlos Ramalho

1.1 Aeonius: uma extensão funcional para Python

Área: Processamento de Linguagens

A programação funcional é um paradigma baseado no cálculo *lambda*, que se caracteriza pela imutabilidade dos dados e na manipulação de funções puras. A primeira linguagem funcional foi o Lisp, desenvolvido na década de 1950 no MIT. Atualmente, este paradigma tem ganho popularidade, pela sua utilidade em problemas de computação paralela e sistemas distribuídos. Exemplos de linguagens funcionais modernas são Haskell, Elixir, e F#.

Devido a este aumento de popularidade, várias linguagens de programação - incluindo Java e Python - têm integrado alguns conceitos e ideias da programação funcional. No entanto, o modo de as utilizar é, ainda, bastante diferente de uma linguagem puramente funcional.

Deste modo, propõe-se a linguagem de programação Aeonius: uma extensão funcional para Python. Python é uma linguagem interpretada de alto nível, usada amplamente nas áreas de ciência de dados e aprendizagem automática. Aeonius permite programar com a segurança e estrutura de uma linguagem funcional, mantendo total compatibilidade com Python.

Estrutura do Relatório

O presente relatório encontra-se dividido em cinco capítulos e quatro apêndices. O capítulo 1, este mesmo, corresponde à introdução e contextualização do projeto. O capítulo 2 corresponde a uma análise detalhada do problema, e à definição dos requisitos que a extensão funcional (e o seu compilador) devem cumprir. De seguida, o capítulo 3 aborda o desenho de uma solução que cumpra com os requisitos previamente definidos, com ênfase particular na criação da linguagem em si e a sua gramática. O capítulo 4 aborda a implementação da solução, detalhando todas as decisões importantes tomadas, os seus motivos e os problemas que surgiram. Além disso, também se discute os testes que foram realizados à linguagem, quer para garantir a sua correção quer para comparar o seu desempenho com Python. O último capítulo - 5 - apresenta as conclusões do projeto, juntamente com uma análise crítica dos resultados obtidos e potencial trabalho futuro.

Relativamente a apêndices, o apêndice A corresponde à definição detalhada da linguagem Aeonius, mostrando

todas as suas estruturas e o seu comportamento esperado. De seguida, o apêndice B explica como rodar o projeto localmente. Finalmente, o apêndice C apresenta todos os símbolos de Aeonius, enquanto o D apresenta a gramática desenvolvida para este projeto.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O problema consiste em extender Python para permitir programar segundo um paradigma funcional. Com isto, pretende-se que a linguagem desenvolvida mantenha toda a interoperabilidade com Python, que é o mesmo que dizer que, se um programa for válido em Python, também o deve ser em Aeonius.
Por outro lado, o código em Aeonius deve respeitar alguns princípios inerentes à programação funcional:

1. As funções devem ser puras, isto é, não devem ter efeitos laterais (*side effects*)
2. Não deve existir estado mutável
3. Deve ser possível manipular funções como qualquer outro tipo de dados (para, por exemplo, compor duas funções, ou para criar funções de ordem superior)

Para executar um programa em Aeonius, deverá haver duas modalidades: converter para código Python *a priori*, e executar o programa resultante; ou correr o programa diretamente de forma interpretada.

2.2 Especificação do Requisitos

De seguida, apresentam-se os requisitos levantados para o projeto.

2.2.1 Controlo do Fluxo

1. Deve ser possível determinar o resultado de uma função a partir de outras funções (composição)
2. Deve ser possível determinar o resultado de uma função de forma recursiva
3. Deve ser possível determinar o resultado de uma função a partir de uma condição (*if ... else*)
4. Deve ser possível determinar o resultado de uma função a partir da estrutura dos seus argumentos (*pattern matching*)

2.2.2 Tipos e Estruturas de Dados

1. Deve ser possível operar sobre os tipos de dados básicos de Python (inteiros, número de vírgula flutuante, *strings*, booleanos, listas, tuplos e dicionários

2.2.3 Compilador / Interpretador

1. Deve ser possível converter um ficheiro de código Aeonius para um ficheiro de código Python válido
2. Deve ser possível correr um ficheiro de código Aeonius de forma idêntica a um ficheiro de Python normal
3. Deve ser possível misturar Python com Aeonius dentro de um só ficheiro

Capítulo 3

Concepção/desenho da Solução

3.1 A Linguagem

Tendo em conta a liberdade dada pelo enunciado do projeto, o primeiro passo consiste em idealizar a extensão funcional a implementar. Tendo em conta que deve ser compatível com Python, pretendeu-se que a linguagem fosse bastante parecida com esta, herdando alguns conceitos de linguagens funcionais. Neste caso, a inspiração principal foi Haskell.

Uma das principais capacidades de Haskell que se pretendeu capturar nesta extensão foi o facto de ser possível aplicar as funções parcialmente [oG20]. Esta funcionalidade dá muita flexibilidade à linguagem, por facilitar passar funções como argumentos a funções de ordem superior.

Um dos principais desvios relativamente a Haskell foi no sistema de tipos. Decidiu-se, por simplicidade e por ser o mais semelhante a Python, omitir declarações de tipo na linguagem. A sua implementação seria, ou limitadora (por exemplo, não suportar polimorfismo), ou demasiado trabalhosa tendo em conta a janela temporal do projeto.

Para manter semelhança a Python, foi necessário tornar a forma como o código está indentado relevante para o seu processamento. Isto trouxe alguma complexidade adicional ao projeto, facilmente compensada pelo valor que traz ao trabalho.

São fornecidos vários exemplos de código Aeonius na pasta examples. Uma especificação exaustiva do modo de funcionamento da linguagem pode ser consultado em A.

Nome

Sem dúvida a decisão mais difícil do projeto foi nomear a extensão funcional. Após longo debate, decidiu-se chamar Aeonius, derivada da palavra grega *aeon*, que significa *um intervalo de tempo extremamente longo.* [dic14]

3.2 *Tokenização*

A enumeração dos símbolos terminais da linguagem encontra-se em C. Duas decisões atípicas destacam-se na conceção do lexer:

- a inclusão de *tokens* correspondentes a mudanças de indentação (aumentar ou diminuir). Estes *tokens* são apelidados de INDENT e UNDENT, e foram cruciais para a definição de *scopes*. A forma como estes foram calculados será explicada em 4.1.2.

- a distinção entre operadores de diferentes preceências e associatividades através de anotações. Este passo tem de ser feito no lexer de forma a utilizar o mecanismo built-in do yacc de precedência. Este processo é explicado em detalhe em 4.1.2

Além disso, decidiu-se que código Python seria *tokenizado* inteiro, ou seja, não será alvo de qualquer tipo de processamento adicional por parte da extensão.

3.3 Gramática

Ao definir a gramática a implementar é necessário, naturalmente, ter em conta o tipo de *parser* que vai ser usado. Neste caso, será o ply.yacc, que é um parser LALR[Bea]. Isto facilita a definição da gramática, ao suportar, por exemplo, prioridade e associatividade explícita de cada produção. A gramática criada é apresentada em D.

O desenvolvimento da gramática foi um processo iterativo, e que sofreu várias revisões ao longo do tempo, de forma a resolver problemas que surgiram ou suportar novas funcionalidades.

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

4.1.1 Estrutura do Projeto

O código está dividido em dois *packages*: o principal e o *package* language. Isto permitiu melhor separar todas as estruturas auxiliares usadas para representar um programa da linguagem aeonius dos restantes ficheiros de código. O *package* principal contém a implementação do *lexer* e do *parser*, assim como o programa principal que compila / interpreta o código.

Relativamente ao programa principal, decidiu-se que a interpretação dos argumentos que este recebe - cujo conteúdo permitido pode ser consultado em B.0.3 - seria feito de raiz, sem recorrer a módulos externos, como, por exemplo, o argparse. Embora esta última opção tivesse facilitado a implementação; sendo este um projeto no âmbito de Processamento de Linguagens, e dada a relativa simplicidade relativamente ao restante projeto, o grupo decidiu não a usar.

Como referido anteriormente, a interpretação dos argumentos foi simples de implementar, não tendo havido necessidade de recorrer a ferramentas como expressões regulares. Deste modo, começou-se por processar as *flags* que não recebem nenhum argumento, removendo-as depois do *array* de argumentos. De seguida, agrupou-se os argumentos aos pares. Para cada par, o primeiro elemento é a configuração e o segundo o seu valor. Estes valores foram colocados num dicionário, para utilização futura. A função responsável por isto é a parse_args do ficheiro main.py.

4.1.2 *Tokenização*

De forma a implementar as features expostas anteriormente, é por vezes necessário adicionar ou remover *tokens* do *output* do *lexer*. Para isso, adotou-se uma estratégia de composição de filtros. Cada filtro é uma função independente que recebe a *stream* de *tokens* e realiza uma parte independente do processamento. Esta estratégia permite facilmente isolar passos independentes e estado de processamento.

No total foram usados 4 filtros, explicados em detalhe de seguida.

Indentação e *Scopes*

Para detetar indentação, é primeiro necessário calcular a posição horizontal de cada *token*, visto isto não ser feito pelo lex (a posição vertical é feita facilmente como descrito na documentação do ply). Esta é a função do primeiro filtro. O seu funcionamento é muito simples: sempre que encontra um *token*, soma o seu comprimento à posição horizontal atual e atualiza todos os *tokens* seguintes com esta posição (um problema

com esta estratégia é não suportar *tabs*, visto o comprimento de um *tab* depender do visualizador de texto). Ao receber um *token* de fim de linha, a posição atual volta a zero.

Para realizar a indentação em si usaremos outro filtro. Para esta tarefa é preciso ter em conta que um *scope* pode ser aberto simplesmente com indentação (se a posição do primeiro *token* da linha for maior que a linha anterior) ou ao encontrar certos *tokens* (RIGHTARROW, RESULTARROW e guarda (|)). Ao processar estes *tokens*, o filtro de indentação cria um novo *token* INDENT e insere posição atual na *stack* de indentação, que guarda as várias posições horizontais em que foi aberto um *scope*. Sempre que uma linha começar numa posição registada anteriormente, o topo da *stack* de indentação é removido, e são criados *tokens* UNDENT até atingir a posição desejada.

Precedência de Operadores

Como dito anteriormente, o grupo decidiu utilizar o mecanismo padrão do yacc para a precedência de operadores, que consiste em declarar uma precedência diferente para cada *token* recebido do *lexer*. Assim, o *lexer* deve gerar tokens diferentes para operadores com diferentes prioridades. No entanto, o grupo também quis permitir a um programador de Aeonius definir os seus próprios operadores, com as suas próprias precedências. Isto significa que, dependendo do input do *lexer*, um mesmo símbolo pode ser traduzido como *tokens* diferentes dependendo do seu nível de precedência.

Definiu-se 5 possíveis níveis de precedência. Visto que cada nível pode ser associativo à esquerda ou à direita, isto dá no total 10 *tokens* como os quais um operador pode ser interpretado (11, se contarmos com operadores cuja precedência não é especificada pelo utilizador. Estes são interpretados como não associativos). Para definir a precedência de um operador, é usada a sintaxe exposta na secção A.0.7. Todos os operadores antes e depois desta anotação são afetados.

Para concretizar este mecanismo, o filtro de precedência percorre duas vezes a *stream* de *tokens*. Na primeira iteração, apenas procura os *tokens* correspondentes às anotações. Na segunda passagem, estes *tokens* são removidos e quaisquer operadores afetados pelas anotações são substituídos pelo *token* correspondente.

Finalmente, o último filtro é uma simples concatenação de *tokens* de fim de linha que possam ter sido separados por uma anotação e devem passar para a fase de interpretação como um único fim de linha.

4.1.3 Estruturas Intermédias

Para representar as estruturas intermédias da linguagem, definiram-se diversas classes, todas subclasses da classe abstrata Element. Isto obriga-as a implementar métodos de conversão para Python, e validação, para além do método de comparação de igualdade. Estas classes encontram-se, como mencionada anteriormente, no *package* language, e na pasta homónima.

Procurou-se abstrair ao máximo a gramática, isto é, que as classes não tivessem uma correlação direta com as regras da gramática. Apesar disso, há ainda uma proximidade forte entre ambas. Para aumentar a legibilidade do código, procurou que essas classes tivessem anotações de tipo dos seus argumentos, usando, para esse efeito, o módulo typing de Python.

4.1.4 Conversão para Python

A converter as estruturas de aeonius para Python válido, o primeiro desafio foi o facto de esta depender de fatores externos à própria estrutura: o nome atual da função, o nome do seu argumento e os símbolos existentes atualmente, mapeando o seu nome em aeonius para o seu nome no código Python gerado.

Por isso, não é possível simplesmente definir a função str para efetuar a conversão. Por isso, obrigou-se cada Element a definir um método to_python, que recebe um argumento, correspondente ao contexto atual do

compilador.

Context

Para uniformizar o tipo desse contexto entre todas as estruturas, este foi encapsulado na classe Context. Um Context contém informação sobre o *scope* atual, nomeadamente:

1. Nome da função atual
2. Nome do argumento da função atual
3. Símbolos já definidos, e o mapeamento entre o seu nome em Python e aeonius
4. Nome da próxima variável a ter de ser criada

A necessidade dos primeiros dois veio de uma convenção que foi estabelecida para facilitar a implementação de funções: definiu-se que, para qualquer função, o nome da variável que seria devolvida[1] correspondia à *string* "return_" concatenado com o nome da função. Por exemplo, a função foo devolve sempre a variável return_foo. A terceira veio do facto de ser preciso saber o mapeamento de símbolos entre Python e aeonius para, por exemplo, expressões auxiliares e variáveis definidas em *pattern matches*.

De forma análoga a *scopes*, a classe Context é, implicitamente, uma *stack*. Sempre que se entra num novo *scope*, a função to_python cria uma nova instância da classe Context, que é construída por cima da atual. Esta nova instância é enviada a todas as estruturas dentro desse *scope*, e deixa de existir no momento em que a to_python retorna. A criação de uma nova instância não afeta a instância original, mas copia todos os símbolos.

A decisão de utilizar várias instâncias de Context em vez de só uma surgiu da maior facilidade em alterar o *scope*, uma vez que esta solução não necessita de implementar nenhum mecanismo para sair do *scope* atual. Para sair, basta deixar de utilizar a instância correspondente.

lambda vs. def

A estrutura das funções de aeonius, nomeadamente o facto de só receberem um argumento de cada vez (curried), significa que estas têm de retornar novas funções caso pretendam receber vários argumentos. Em Python podem definir-se funções utilizando duas sintaxes distintas: def e lambda. O grupo teve de escolher a primeira devido ao requisito de poder definir variáveis intermédias dentro de funções; algo que expressões *lambda* não suportam.

Esta necessidade implica que o código gerado é bastante verboso e relativamente difícil de perceber, mas o benefício que esta forma de definir funções tem supera estas desvantagens.

4.1.5 Validações

Após a construção das estruturas intermédias que representam um programa aeonius, é efetuado um passo de validação dessas estruturas. O objetivo deste passo é detetar erros ao nível semântico, ou seja, programas que sejam sintaticamente corretos, mas que não façam sentido ao nível semântico (por exemplo, utilizar uma variável que não está definida).

A validação foi feita através da implementação do método validate da classe abstrata Element. Este método devolve dois valores: se a validação foi bem sucedida - ou seja, um booleano - e, caso não tenha sido, os

[1] Todas as funções terminam com uma linha return <nome variavel>

erros encontrados. Não se utilizaram exceções para permitir continuar a validação mesmo após encontrar um erro, permitindo mostrar todos os erros ao utilizador de uma só vez. Esta construção é inspirada no funcionamento da mónade Maybe de Haskell.
As validações que se realizam são as seguintes:

1. Validação da existência de todos os símbolos usados (ex: verificar que uma função está definida quando é invocada) [2]

2. Impedir que uma variável seja redefinida (ou seja, garantir a sua imutabilidade)

4.1.6 Biblioteca Padrão

Para facilitar o desenvolvimento de *software* em aeonius, criou-se uma biblioteca padrão, em Python, que é importada automaticamente; o que significa que estas funções podem ser usadas diretamente em código aeonius. Esta encontra-se no ficheiro aeonius_stdlib.py. Procurou-se desenvolver uma biblioteca que fosse semelhante ao Prelude de Haskell[oG01]. No entanto, algumas colisões de nomes com funções pré-definidas de Python (ex: map) implicaram fazer ligeiras alterações aos nomes das funções. Como é óbvio, não se desenvolveu o Prelude completo, mas apenas a parte que o grupo considerou relevante para este projeto.

4.1.7 Compilação automática e integração com módulos python

Com objetivo de ter uma maior compatibilidade com Python, foi decidido que, ao importar o módulo aeonius num ficheiro que contém código aeonius, este seria reescrito em Python e executado. As funções e variáveis ficam definidas no módulo em que o código aeonius se encontra. Assim, caso se pretenda importar um ficheiro que contenha código aeonius, como por exemplo

ae.py

```
1  import aeonius
2  """aeonius
3  def enumFromTo:
4      x -> y | x > y => []
5             |       => [x] + (enumFromTo (x + 1) y)
6  """
7  print(enumFromTo(1)(10))
```

basta fazer

```
1  from ae import enumFromTo
```

Para implementar esta funcionalidade, foi preciso resolver vários problemas, que são explicados de seguida.

Descobrir que módulo importou aeonius

Para resolver este problema utilizamos a biblioteca inspect. Através desta biblioteca temos acesso à *frame stack*. Percorremo-la e, para cada elemento, obtemos qual foi o módulo em que este foi definido: caso a propriedade $\_\_name\_\_$ do módulo for igual ao $\_\_name\_\_$ no módulo aeonius, temos os dados do módulo que importou aeonius.

[2] Esta verificação é feita da mesma forma que C, ou seja, apenas são considerados os símbolos já previamente definidos. Isto significa que não é permitido invocar uma função que só esteja definida mais à frente no código

Obter o código do módulo

Tendo obtido os dados do módulo como descrito anteriormente, para obter o caminho para o ficheiro de código recorre-se à propriedade *__file__* do mesmo. Deste modo, consegue-se ler e *transpilar* o conteúdo, adicionando também a biblioteca padrão 4.1.6.

Execução múltipla do mesmo código

Visto que, ao executar o código Python do ficheiro teríamos o problema de este ser executado pelo aeonius e pelo interpretador Python, foi decidido que apenas o código aeonius seria transpilado, e que o código Python seria filtrado e apenas executado posteriormente pelo seu interpretador.

Colocar o código aeonius transpilado no módulo

Visto que o código aeonius é escrito como uma string de múltiplas linhas Python sem ser atribuída a uma variável este tem de ser *transpilado* e executado pela extensão. Como já foi obtido o módulo onde o código aeonius está definido, foi somente necessário executar o código *transpilado* passando como contexto um dicionário de variáveis do módulo que foi obtido através da propriedade *__dict__*.

4.2 Testes realizados e Resultados

4.2.1 Casos de Teste

Para testar a linguagem, as 50 questões da Unidade Curricular de Programação Funcional foram resolvidas em aeonius. O código foi entregue na pasta test_files/aeonius. As restantes subpastas de test_files correspondem a implementações semelhantes em Python, que serão explicadas em 4.2.3.

4.2.2 Correção

Para testar a implementação da linguagem, utilizou-se as 50 questões de Programação Funcional mencionadas anteriormente. Para testar a correção, compilou-se o programa para Python, e rodaram-se, manualmente, diversos casos de testes (incluindo *edge cases*), para assegurar que o *output* é o esperado.

Não foi implementada qualquer *pipeline* de testes de correção automática, por restrições temporais.

4.2.3 Desempenho

Para testar o desempenho de aeonius comparativamente a Python, as 50 questões foram resolvidas em Python, de duas formas distintas: a primeira recursiva, mais próxima do código aeonius gerado, e a segunda de forma mais *a la* Python.

De seguida, mediram-se os tempos de execução das três implementações fazendo a diferença entre os *timestamps* antes e depois de executar cada implementação. Para cada questão, executou-se cada solução dez vezes e guardou-se os dados em formato *JSON* para ser posteriormente analisados.

Posteriormente criaram-se ferramentas para analisar a média e variância dos tempos para cada solução. Os resultados da média são apresentados na Figura 4.1.

O gráfico demonstra a média de tempo de execução de cada estilo de programação.

- IT corresponde a Python iterativo[3]

[3]Há perguntas onde a definição iterativa não é a mais natural, e, nesses casos, foi utilizado um algoritmo recursivo

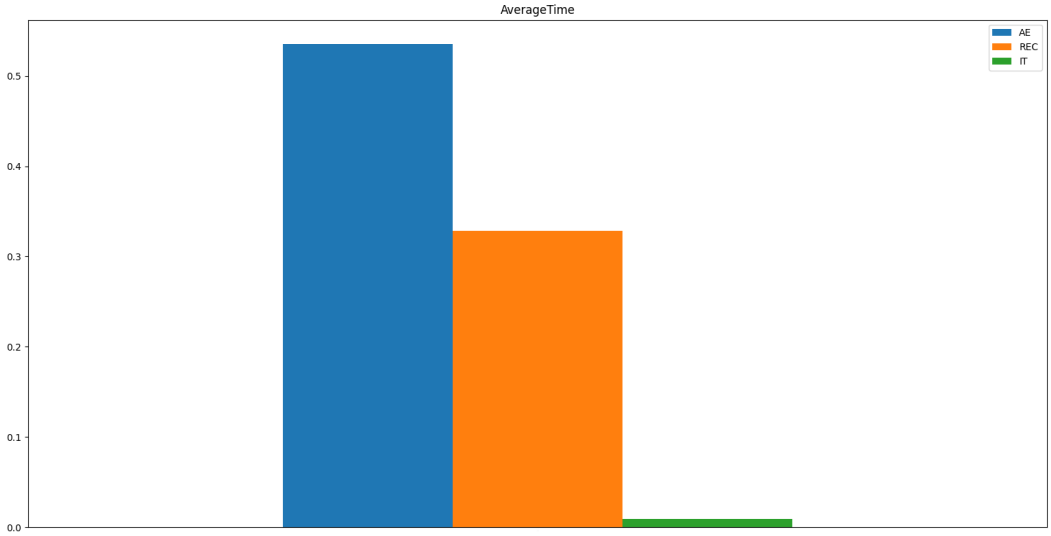

Figura 4.1: Desempenho das três soluções

- REC corresponde a Python recursivo
- AE corresponde a Aeonius

Os valores são os esperados. Aeonius tem um desempenho inferior a Python. Isto deve-se, em parte, à utilização de recursividade, que tem grande impacto no desempenho, visível pela diferença entre as versões iterativas e recursivas de Python.

A diferença entre Python recursivo e Aeonius pode ser explicada pelo maior *overhead* de chamadas recursivas de funções com mais do que 1 argumento, graças à forma como Aeoonius lida com funções *curried*.

Capítulo 5

Conclusão

Com isto, especificou-se uma extensão funcional para Python baseada em Haskell, definiu-se a sua gramática e símbolos terminais, abordou-se as ações semânticas responsáveis por converter Aeonius em Python, e realizaram-se testes de desempenho, que revelam uma diminuição da *performance* relativamente a Python.

Deste modo, o grupo considera que o projeto, tendo em conta os objetivos inicialmente propostos, está num estado bastante bom. Todas as funcionalidades básicas previstas no enunciado estão implementadas, juntamente com outras funcionalidades mais avançadas, que, no entender do grupo, valorizam bastante o projeto.

No futuro, seria possível expandir significativamente o trabalho desenvolvido, nomeadamente através da melhoria e adição das validações que são feitas ao código, e da correção das limitações existentes referidas em A.0.8.

Além disso, aumentar a dimensão da biblioteca padrão da linguagem, suportar as classes de Python, assim como iteradores adicionaria bastante valor à linguagem. Finalmente, a implementação de mónades *a la* Haskell, em particular o mónade IO [Tho00] permitiria desenvolver aplicações completamente em Aeonius, sem necessitar de escrever código Python, impuro.

Apêndice A

Especificação da Linguagem Aeonius

A.0.1 Interoperabilidade com Python

Um ficheiro de código em Aeonius - que deve ter a extensão `.py` como qualquer outro ficheiro Python - é um ficheiro de texto, que está dividido em blocos de 2 tipos:

1. Código Python
2. Código Aeonius

Para indicar que um segmento de código corresponde a Aeonius, este deve ser começar com uma linha `"""aeonius` e terminar com `"""`. Caso contrário, o bloco é tido como sendo de Python. O código Python é transpilado verbatim, enquanto que o código Aeonius é processado. Um exemplo de ficheiro é o seguinte:

```
1  """aeonius
2      def ola_mundo:
3          x -> "Hello world"
4  """
5
6  print(ola_mundo(5))
```

O *output* deste programa é, naturalmente, a *string Hello World*.

A.0.2 Funções

Fundamentos

1. Funções são o bloco fundamental da linguagem
2. Funções recebem exatamente um argumento e devolvem um único valor
3. O resultado de uma função deve depender exclusivamente do seu argumento (deterministicamente)
4. A chamada de uma função não deve alterar o seu argumento nem nenhum tipo de estado global

Declaração

Existem 2 formas equivalentes de declarar funções:

1.
```
        def x:
            y -> 5 * y
```

declara uma função `x` que recebe um argumento `y` e devolve-o multiplicado por 5

2.
```
        x = lambda y: 5 * y
```

declara uma função anónima como um *lambda*, à semelhança de Python. Depois atribui essa função à variável x.

Os nomes de funções devem ter no mínimo 1 caracter, começar por uma letra ou um *underscore*. A partir do segundo caracter, são permitidos caracters alfanuméricos e o *underscore* (no entanto, um único *underscore* não é um nome válido e deve ser seguido de pelo menos um caracter). Os nomes não podem coincidir com nomes de funções já existentes em Python (ex: print). As mesmas regras aplicam-se para nomes de variáveis.

Pattern Match

Para declarar funções utilizando *pattern matching*, é necessário utilizar a notação com `def`:

```
def f:
    0 -> None
    n -> 1 / n
```

A função `f` retorna `None` se receber `0` e retorna o resultado da divisão `1 / n` em qualquer outro caso.

É possível definir patterns usando primitivas, tuplos, listas, dicionários e variáveis (não deve estar atribuída no *scope* atual). É também possível usar um *underscore* (_) para permitir um *match* sem indicar uma variável. Neste caso, qualquer valor entra nessa cláusula, apesar de ser ignorado para a computação do resultado.

```
def g:
    (0, x, *_) -> x
    [0, x, *_] -> x
    _ -> None
```

A função `g` retorna o segundo elemento de tuplos e listas que comecem por `0` e None em qualquer outro caso. Todas as formas de fazer *pattern matching* sobre os diferentes tipos de dados de Aeonius é explicado em detalhe em A.0.5.

Curry

Em Aeonius é possível criar funções que recebem mais do que um parâmetro. Para o fazer, os restantes argumentos devem ser passados *curried*. Eis dois exemplos, usando as duas formas de definição de funções:

```
def h:
    x -> y -> x * y
```

```
h = lambda x: lambda y: x * y
```

A função `h` recebe um argumento `x` e devolve uma outra função que recebe um argumento e calcula o produto desse pelo valor `x`. Na prática, pode ser interpretada como uma função que recebe 2 argumentos e calcula o seu produto.

Invocação

Invocar uma função em Aeonius é diferente de invocar uma função em Python. Para invocar uma função faz-se o seguinte

```
1    x = f 5
```

Se a função devolver outra função (ou seja, que "receba"vários argumentos) esta é invocada da seguinte forma

```
1    x = f 5 3
```

Neste caso, e por consequência do facto de funções receberem um só argumento, pode aplicar-se a função parcialmente. Por exemplo:

```
1    def f:
2        x -> y -> x * y
3
4    g = f 5
```

A função `g` recebe um argumento e devolve-o multiplicado por 5.

Indentação e *Scopes*

Um *scope* é definido como um conjunto de código dentro do qual todas as atribuições de variáveis, funções e operadores são invisíveis para o exterior. Símbolos atribuídos num *scope* podem ser apenas utilizados dentro desse *scope* ou de *scopes* aninhados. Para representar este conceito visualmente no código é usada indentação. Código escrito com maior indentação que o código da linha anterior abre um *scope*, e código escrito com menor indentação fecha um ou mais *scopes*.

Aeonius permite abrir *scopes* apenas ao declarar funções e operadores. Para além disso, sempre que uma seta (`->` ou `=>`) é encontrada é aberto um *scope* com a indentação do próximo símbolo encontrado. As guardas (`|`) têm um efeito semelhante, porém abrem um *scope* na posição da guarda em vez do símbolo seguinte.

Nem todos os *scopes* são iguais. Dentro de *scopes* iniciados por uma seta intermédia (`->`) deve ser colocado pelo menos um *pattern match* antes de eventuais atribuições. *Scopes* iniciados por uma seta de resolução (`=>`) esperam uma expressão antes de eventuais atribuições. Por fim, *scopes* guardados (`|`) recebem uma condição (uma expressão cujo valor é avaliado em True ou False) após cada guarda.

Atribuições declaradas sem qualquer indentação encontram-se no *scope* global, e os símbolos que definem são acessíveis em todo o documento.

A.0.3 Operadores

Operadores são semelhante a funções, excetuando o facto de receberem obrigatoriamente dois argumentos, e a sua aplicação ser feita de forma *inline*. Por exemplo:

```
1    op (.):
2        f -> g -> lambda x: f (g x)
3
4    x = f . g
```

`x` corresponde à função composta $f.g$. O nome do operador deve ser declarado dentro de parênteses e pode conter apenas os caracteres `+`, `-`, `*`, `/`, `%`, `<`, `=`, `>`, `$`, `^`, `&`, `|`, `.`, `!`, `?` e `\`. Deve ter no mínimo 1 carácter.

É possível definir como este operador se deve associar com outros operadores em redor através do mecanismo de precedência discutido na secção 4.1.2.

A.0.4 Variáveis

É possível declarar variáveis (imutáveis) atribuindo-lhes um valor.

```
1    x = 2 * 5
```

declara uma variável `x` e atribui-lhe o valor do produto `2 * 5`. O produto é calculado imediatamente e o resultado é guardado em memória.

Uma vez atribuída, uma variável não pode ser alterada. Tentar atribuir novamente a mesma variável no mesmo *scope* (ou num *scope* aninhado) ou realizar *pattern match* com o mesmo identificador leva a um erro. É possível usar a variável atribuída no mesmo *scope* e em todos os *scopes* aninhados neste.

A.0.5 Tipos Primitivos

Os tipos primitivos de Aeonius correspondem aos mesmos tipos de Python. Enumerando

1. **Números inteiros e de vírgula flutuante.** Estes podem ser *matched* de duas formas diferentes: ou pelo valor exato, ou por atribuição a uma variável. Por exemplo

```
1        def fatorial:
2            0 -> 1
3            n -> n * factorial (n - 1)
```

   A comparação entre números segue as mesmas regras que Python.

2. **Booleanos**. Os booleanos seguem a mesma estrutura de Python. Correspondem a dois valores: True e False. Só é possível fazer *match* a estes valores ou diretamente ou atribuindo a uma variável.

3. **Tipo Nulo**. O tipo nulo é semelhante a Python, escrevendo-se None. Apenas pode ser *matched* diretamente ou por atribuição a uma variável.

4. **Listas**. Uma lista pode conter elementos de qualquer tipo de dados (incluindo listas). Uma lista pode ser *pattern matched* relativamente ao seu número de elementos, da seguinte forma

```
1        def length:
2            []        -> 0 #Lista vazia
3            [a]       -> 1 #Lista de 1 elemento
4            [a,b]     -> 2 #Lista de 2 elementos
5            [a,*b]    -> 1 + length b #Lista com pelo menos 1 elemento
6            [a,b,*c] -> 2 + length c #Lista com pelo menos 2 elementos
```

   No caso de listas matched por um número mínimo de elementos, a sintaxe [a,*b] divide a lista na sua cabeça (o valor `a`) e a sua cauda, que também é uma lista (o valor `b`).

   Além de fazer *pattern matching* pelo número de elementos da lista, também é possível fazê-lo pelo valor de cada elemento (não é possível fazer isso para a cauda da lista).

```
1        def g:
2            [1,2,3]    -> False
3            [2,3,*t] -> True
```

5. ***Strings***. *Strings* são casos particulares de listas. Isto significa que qualquer *match* que funcione para listas funciona para *strings*. Além disso, é possível utilizar diretamente o valor da *string*, como no exemplo seguinte

```
        def isHelloWorld:
            "hello world!" -> True
            _              -> False
```

As *strings* são internamente convertidas para arrays de caracteres (ou, em Python, *strings* de tamanho 1). Isto traz consequências ao trabalhar com *strings* vindas de Aeonius a partir de código Python, sendo necessário concatenar a lista para uma única *string* para a poder manipular normalmente.

6. **Tuplos**. Tuplos funcionam de forma igual a listas, excetuando o facto de serem envolvidos por `()` em vez de `[]`.

7. **Dicionários**. Os dicionários, à semelhança de Python e outras linguagens de programação, são estruturas de dados associativas, que permitem associar valores a chaves, as quais permitem aceder aos elementos de forma rápida. Em Aeonius, os dicionários funcionam de forma idêntica a Python. Neste momento, o seu *pattern match* está reduzido a constantes. Por exemplo:

```
        def dict_soma:
           {} -> 0 #Dicionario vazio
           {1:v} -> v #Dicionario de 1 elemento com a chave 1
           {2:x, 3:y} -> x + y #Dicionario de 2 elementos com aa chaves 2 e 3
```

Não é permitido fazer *match* da chave e do valor simultaneamente utilizando `_`.

A.0.6 Expressões

Condicionais

Aeonius permite alterar o resultado de uma expressão dependendo de uma dada condição. Isto é feito através da construção a `if condition else b`, à semelhança do que é possível fazer em Python.

```
    def paridade_string:
        n -> "par" if n % 2 == 0 else "impar"
```

Conjuntos Por Compreensão

Quer listas quer dicionários podem ser construídos por compreensão, da seguinte forma:

```
    x = [2 * y for y in [1,2,3]] # Devolve a lista [2,4,6]
    z = {y : 2 * y for y in [1,2,3]}  # Devolve o dicionario {1:2,2:4,3:6}
```

Opcionalmente, é possível utilizar uma condição para apenas considerar alguns valores do conjunto. Por exemplo:

```
    x = [2 * y for y in [1,2,3] if y % 2 == 1] # Devolve a lista [2,6]
    z = {y : 2 * y for y in [1,2,3] if y % 2 == 1} # Devolve o dicionario {1:2,3:6}
```

Como em *pattern matches*, é possível decompor os elementos num *loop for*.

```
    x = [k + v for (k,v) in {1:2, 2:3}] # Devolve a lista [3,5]
```

Precedência	Associatividade	Operação
1	esquerda	==
1	esquerda	!=
1	esquerda	<
1	esquerda	>
1	esquerda	<=
1	esquerda	>=
2	esquerda	||
2	esquerda	&&
2	esquerda	|
2	esquerda	&
3	esquerda	+
3	esquerda	-
4	esquerda	*
4	esquerda	/
4	esquerda	%
5	esquerda	**

Tabela A.1: Prioridade e associatividade das operações predefinidas em Aeonius

A.0.7 Precedências

As operações em Aeonius têm a seguinte precedência (1 significa menor prioridade):
É possível definir a precedência de um ou mais operadores usando a seguinte notação:

```
infixl 3 + -
```

Nesta notação, o infixl indica associatividade à esquerda (associatividade à direita seria infixr), o 3 indica a precedência (deve ser um número inteiro de 1 a 5) e os operadores afetados são listados em seguida, separados por um espaço.

A.0.8 Limitações

Neste momento, existem algumas limitações com esta especificação.

1. *Strings* não são chaves de um dicionário válidas, uma vez que são internamente convertidas para listas
2. Não é possível fazer atribuições com *pattern matches* de dicionários no corpo da expressão. Por exemplo:

```
    def foo:
        x => y
            {"y": y} = bar x
```

   não é válido em Aeonius

3. Não é possível usar variáveis como constantes em pattern matches. Por exemplo:

```
    def getValue:
        i -> {i: x, **_} => x
```

   dará o erro "Redefinition of symbol i"

4. Ao fazer *pattern match* de dicionários, o dicionário vazio deve ser sempre a última cláusula

Apêndice B

Como rodar o Projeto

B.0.1 Pré-requisitos

Rodar o projeto implica ter uma versão de Python 3.10 ou superior instalada [1]. É necessário utilizar o `pip` para gestão das dependências do projeto.

B.0.2 Configuração

1. Opcionalmente, criar um ambiente virtual para instalar as dependências

```
pip install virtualenv
python -m venv aeonius
source aeonius/bin/activate # Univ
aeonius/Scripts/activate.bat # Windows, cmd
```

2. Instalar as dependências

```
pip install -r requirements.txt
```

B.0.3 Execução

Executar um ficheiro de código Aeonius pode ser feito simplesmente rodando-o como um *script* normal de Python:

```
python code.py
```

Alternativamente, pode rodar-se o compilador de Aeonius e gerar um novo ficheiro rodando:

```
python main.py --input aeonius.py --output python.py
```

Para enviar o output do compilador para o stdout em vez de um ficheiro a flag *d* pode ser usada.

```
python main.py -d --input aeonius.py
```

Para obter ajuda de como rodar o programa usa-se a flag *-h*.

```
python main.py -h
```

[1] Esta exigência advém do facto de o Aeonius utilizar a estrutura match, que apenas foi introduzida nessa versão

Para gerar o grafo com a linguagem a flag *-g* pode ser usada.

```
python main.py -g --input aeonius.py --output graph
```

B.0.4 Testes

Para rodar os testes, correr o *script* `tester.py`.
Para executar os testes e guardar os resultados usa-se a flag *r*.

```
python tester.py -r
```

Para mostrar os gráficos com os resultados guardados a flag *s* pode ser usada.

```
python tester.py -s
```

Ambas as flags podem ser usadas em simultâneo.

Apêndice C

Símbolos da Linguagem Aeonius

Literal	Nome
def	DEF
if	IF
else	ELSE
for	FOR
in	IN
op	OP
lambda	LAMBDA
True	TRUE
False	FALSE
None	NONE
->	RIGHTARROW
=>	RESULTARROW

Tabela C.1: Palavras Reservadas em Aeonius

Símbolo
IDENTIFIER
OPIDENT
INTEGER
FLOAT
STRING
PYTHONCODE
BEGIN
END
WS
EOL
INFIX
INDENT
UNDENT

Tabela C.2: Símbolos em Aeonius

Apêndice D

Gramática da Linguagem Aeonius

```
# aeonius grammar specification

grammar: &
       | aeonius grammar
       | PYTHONCODE grammar

aeonius: BEGIN EOL code END

code: &
    | assignment code

assignment: pattern '=' exp EOL
   | DEF IDENTIFIER ':' EOL INDENT defbody UNDENT
          | DEF IDENTIFIER ':' pattern match
          | OP '(' operator ')' ':' EOL INDENT defbody UNDENT
          | OP '(' operator ')' ':' pattern match

defbody: multipatternmatch code

multipatternmatch: pattern match
                 | multipatternmatch pattern match

multicondmatch: exp match
              | multicondmatch '|' exp match
              | multicondmatch '|' match

match: RESULTARROW INDENT exp EOL code UNDENT
     | RESULTARROW EOL INDENT exp EOL code UNDENT
     | RIGHTARROW INDENT defbody UNDENT
     | RIGHTARROW EOL INDENT defbody UNDENT
     | '|' INDENT multicondmatch code UNDENT
```

```
primitive: INTEGER
         | FLOAT
         | STRING
         | FALSE
         | TRUE
         | NONE
         | '[' ']'
         | '{' '}'


exp: IDENTIFIER
   | primitive %prec PRIMITIVE
   | '(' tupleexp ')'
   | '[' iterexp ']'
   | '{' dictexp '}'
   | exp exp %prec FUNC                 #function call
   | '(' exp ')'                      #Wrapped in brackets
   | LAMBDA pattern ':' exp
   | exp IF exp ELSE exp
   | exp FOR pattern IN exp
   | exp FOR pattern IN exp IF exp
   | '{' IDENTIFIER ':' exp FOR pattern IN exp '}'
   | '{' IDENTIFIER ':' exp FOR pattern IN exp IF exp '}'
   | '(' operator ')'                     #operator function
   | exp OPIDENT exp
   | exp OPIDENT_L1 exp
   | exp OPIDENT_L2 exp
   | exp OPIDENT_L3 exp
   | exp OPIDENT_L4 exp
   | exp OPIDENT_L5 exp
   | exp OPIDENT_R1 exp
   | exp OPIDENT_R2 exp
   | exp OPIDENT_R3 exp
   | exp OPIDENT_R4 exp
   | exp OPIDENT_R5 exp
   | exp UNPACKITER exp
   | exp UNPACKDICT exp
   | exp '|' exp

operator: OPIDENT
        | OPIDENT_L1
        | OPIDENT_L2
        | OPIDENT_L3
        | OPIDENT_L4
        | OPIDENT_L5
        | OPIDENT_R1
        | OPIDENT_R2
        | OPIDENT_R3
```

```
        | OPIDENT_R4
        | OPIDENT_R5
        | UNPACKITER
        | UNPACKDICT
        | '|'


elemexp: exp
       | UNPACKITER exp

tupleexp: elemexp ','
        | nonsingletupleexp
        | nonsingletupleexp ','

nonsingletupleexp: elemexp ',' elemexp
                 | nonsingletupleexp ',' elemexp

iterexp: nonemptyiterexp
       | nonemptyiterexp ','

nonemptyiterexp: elemexp
        | nonemptyiterexp ',' elemexp

dictexp: nonemptydictexp
       | nonemptydictexp ','

nonemptydictexp: exp ':' exp
        | nonemptydictexp ',' exp ':' exp
            | nonemptydictexp ',' UNPACKDICT exp



pattern: primitive %prec PRIMITIVE
       | '_'
       | IDENTIFIER
       | '(' pattern ')'
       | '(' tuplepattern ')'
       | '[' iterpattern ']'
       | '{' dictpattern '}'

tuplepattern: pattern ','
            | pattern ',' UNPACKITER pattern
            | nonsingletuplepattern
            | nonsingletuplepattern ','
            | nonsingletuplepattern ',' UNPACKITER pattern

nonsingletuplepattern: pattern ',' pattern
                     | nonsingletuplepattern ',' pattern
```

```
iterpattern: nonemptyiterpattern
    | nonemptyiterpattern ','
          | nonemptyiterpattern ',' UNPACKITER pattern

nonemptyiterpattern: pattern
     | nonemptyiterpattern ',' pattern

dictpattern: nonemptydictpattern
    | nonemptydictpattern ','

nonemptydictpattern: pattern ':' pattern
     | nonemptydictpattern ',' pattern ':' pattern
                  | nonemptydictpattern ',' UNPACKDICT pattern
```

Bibliografia

[Bea] David Beazley. PLY (Python Lex-Yacc. https://www.dabeaz.com/ply/ply.html. Accessed: 2023-05-25.

[dic14] *Longman Dictionary of Contemporary English 6th Edition*. Pearson, 2014.

[oG01] The University of Glasgow. Prelude. https://hackage.haskell.org/package/base-4.18.0.0/docs/Prelude.html, 2001. Accessed: 2023-05-06.

[oG20] The University of Glasgow. Partial application - Haskell. https://wiki.haskell.org/Partial_application, 2020. Accessed: 2023-05-20.

[Tho00] Reuben Thomas. A Gentle Introduction to Haskell, Version 98. https://www.haskell.org/tutorial/io.html, 2000. Accessed: 2023-05-23.