# Coconut

*Vydání v1.4.0 [Ernest Scribbler]*

**13.09.2018**

# Obsah

KAPITOLA 1

# Frequently Asked Questions

## 1.1 Mohu použít moduly Pythonu z Coconut a moduly Coconut z Pythonu?

Yes and yes! Coconut kompiluje do Pythonu, takže moduly Coconut jsou přístupné z Pythonu a moduly Pythonu jsou přístupné z Coconut, včetně celé standardní knihovny Pythonu.

## 1.2 Které verze Pythonu Coconut podporuje?

Coconut podporuje všechny verze Pythonu `>= 2.6` ve větvi `2.x` nebo `>= 3.2` ve větvi `3.x`. Ve skutečnosti je kód Coconut kompilován tak aby běžel stejně na každé z podporovaných verzí. Viz kompatibilní verze Pythonu.

## 1.3 Může být Coconut použit ke konverzi jedné verze Pythonu na jinou?

Ano, ale jen ve zpětném směru. Coconut může konvertovat Python 3 na Python 2 ale nikoliv obráceně. Coconut vlastně může přeměnit kód Pythonu 3 na Python na verzi nezávislý. Coconut zkompiluje skladbu Python 3, vestavěné entity i dokonce importy na kód, který bude pracovat v každé podporované verzi Pythonu (`2.6`, `2.7`, `>=3.2`).

Existuje však několik výminek: některé konstrukty, jako `async`, nelze replikovat nižších verzích Pythonu a k tomu aby pracovaly, je potřebné je zavést flagem `--target`. Úplný seznam viz compatibilní verze Pythonu.

## 1.4 Jak lze publikovat paket Coconut na PyPI?

Protože Coconut pouze kompiluje do Pythonu, publikování paketu Coconut na PyPI je přesně totéž jako publikování paketu Pythonu, s určitým kompilačním krokem navíc. Napíšete svůj paket v Coconut, spustíte `coconut` pro zdrojový kód a načtete kompilovaný kód na PyPI. Můžete dokonce míchat kódy Pythonu a Coconut, protože se kompilace dotýká pouze souborů `.coco`. Chcete-li vidět příklad paketu PyPI, psaného v Coconut, včetně souboru Makefile, včetně použitých kompilačních příkazů, podívejte se na pyprover.

## 1.5 Kde najdu záznam o posledních změnách Coconut?

Informace o každém vydání Coconat jsou zaznamenávány na stránce GitHub. Zde můžete nalézt všechny nové vlastnosti a výrazné změny, uvedené v jednotlivých vydáních.

## 1.6 Podporuje Coconut kontrolu statických typů?

Ano, Coconut kompiluje nejnovější a nejlepší skladbu anotace typu na komentáře, nezávislé na verzi Pythonu, které potom mohou být kontrolovány s použitím nástroje MyPy Integration.

## 1.7 Pokoušel jsem se napsat rekurzivní iterátor a můj Python způsobil chybu segmentace!

Žádný problém - stačí použít dekorátor `recursive_iterator` z Coconut a budete v pohodě. Toto je známý problém Pythonu a `recursive_iterator` vám jej vyřeší.

## 1.8 Jak rozdělím výraz přes několik řádků v Coconut?

Protože je skladba Coconut nadřazená skladbě Python 3, podporuje Coconut stejné pokračování řádků jako Python. To znamená, že jak pokračování zpětným lomítkem, tak implikované pokračování uvnitř kulatých, hranatych a složených závorek bude chodit. Závorkové pokračování je doporučená metoda a Coconut dokonce podporuje její vylepšenou verzi.

## 1.9 Jsem-li perfektně spokojený s Pythonem, proč bych se měl učit Coconut?

Jste přesně ta osoba, pro kterou byl Coconut vytvořen! Coconut vás nechá psát Python bez starostí s kompabilitou verzí, přičemž vám umožňuje provádět věci, o nichž byste si nikdy nebyl pomyslel že jsou možné, jako je pattern-matching (porovnávání předlohy) a lazy evaluation (líný výpočet). Pokud jste někdy používal funkcionální programovací jazyk, budete vědět, že funkcionální kód je často mnohem jednodušší, čistší a čitelnější. Python je úžasný imperativní jazyk, ale když přijde na moderní funkcionální programování (pro něž nebyl vytvořen), má jisté mezery, které se Coconut snaží doplnit.

## 1.10 Přináší Coconut také nějakou pomůcku pro ladění kódu?

Snadnost ladění je dlouhodobý problém u všech kompilovaných jazyků, včetně jazyků `C` a `C++`, jež jsou v současné době považovány za low-level jazyky. Řešení tohoto problému je stále stejné: párování řádků. Pokud víte, který řádek zdrojového kódu koresponduje s určitým řádkem kompilovaného kódu, můžete snadno provádět ladění přímo ve zdrojovém kódu. V Coconut to lze snadno zařídit připojením flagu `--line-numbers` nebo `-l`, jenž zajistí připojení komentáře ke každému řádku v kompilovaném kódu s číslem odpovídajícího řádku ve zdrojovém kódu. Alternativní flag `--keep-lines` nebo `-k` zajistí vložení celého řádku ze zdrojového kódu místo nebo spolu s číslem řádku. Ohlásí-li tedy Python chybu, můžete na úryvku kompilovaného kódu číst informaci o čísle problematického řádku ve zdrojovém kódu.

## 1.11 Nemám rád funkcionální programování, měl bych se přesto učit Coconut?

Definitely! Kromě toho, že je Coconut skvělý pro funkcionální programování, obsahuje také řadu dalších úžasných vlastností, včetně schopnosti kompilovat kód Python 3 do univerzální verze, která poběží v jakékoli verzi Pythonu. I když Coconut není čistě funkcionální, je to skvělý úvod do funkcionálního stylu.

## 1.12 Neznám funkcionální programování, mám se přesto pustit do Coconut?

Yes, absolutely! Tutoriál nepředpokládá absolutně žádnou předchozí znalost funkcionálního programování, pouze Pythonu. Protože Coconut není čistě funkcionálním programovacím jazykem a veškerý platný Python je platný Coconut, je Coconut skvělým úvodem do funkcionálního programování. Osvojíte-li si Coconut, budete si moci vyzkoušet nový styl programování bez ztráty jakékoli znalosti Pythonu, který znáte a milujete.

## 1.13 Neznám Python moc dobře, měl bych se přesto učit Coconut?

Možná. Znáte-li aspoň základy Pythonu a jste dobře obeznámen s funkcionálním programováním, potom zcela určitě vám Coconut umožní pokračovat v používání všech vašich oblíbených nástrojů funkcionálního programování za současného dalšího seznamování s Pythonem. Nejste-li příliš obeznámen ani s Pythonem ani s funkcionálním programováním, potom učiníte lépe, když nejprve projdete vhodným tutoriálem Pythonu.

## 1.14 Proč není Coconut čistě funkcionální?

Stučně řečeno proto, že Coconut je nadstavba Pythonu, který má sice některé funkcionální vlastnosti ale jako celek je záměrně nefunkcionální. Coconut není čistě funkcionální ze stejných důvodů, ze kterých není Python čistě imperativní - různé problémy vyžadují různé přístupy.

Coconut je záměrně vytvořen tak aby umožnil vytváření kódu v čistě funkcionálním stylu ale lze jej použít i pro jiná paradigmata.

## 1.15 Neuškodí transpilovaný jazyk jako Coconut komunitě Pythonu?

I certainly hope not! Na rozdíl od většiny transpilovaných (transpilled) jazyků, je veškerý Python platný Coconut. Cílem Coconut není nahradit Python ale *rozšířit* jej. Coconut je dokonale interoperativní s Pythonem a používá stejné knihovny. Tudíž Coconut nemůže rozdělit komunitu Pythonu, protože komunita Coconu *je* komunitou Pythonu.

## 1.16 Chci používat Coconut v produkčním prostředí; jak dosáhnu maximálního výkonu?

Za prvé, budete potřebovat rychlý kompilátor, takže byste měl buď to instalovat Coconut s volbou `cPyparsing` nebo použít `PyPy`. Za druhé, existují dvě jednoduché věci, které můžete udělat, abyste přinutili Coconut rychleji produkovat Python: kompilovat se specifikací `--no-tco` a kompilovat se specifikací `--target` pro určitou verzi Pythonu, na níž má váš kód běžet. Zadání specifikace `--target` pomůže optimalizovat kompilovaný kód pro danou verzi Pythonu a byť je koncová optimalizace (Tail Call Optimization) užitečná, obvykle výrazně zpomalí její provedení, takže nepoužití této možnosti způsobí výrazný nárůst výkonu.

## 1.17 Chci přispívat do Coconut, jak mohu začít?

That's great! Coconut is completely open-source, and new contributors are always welcome. Contributing to Coconut is as simple as forking Coconut on GitHub, making changes to the `develop` branch, and proposing a pull request.

If you have any questions at all about contributing, including understanding the source code, figuring out how to implement a specific change, or just trying to figure out what needs to be done, try asking around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.

## 1.18 Proč název Coconut?

Pokud vám to není známo, obrázek nahoře pochází z komedie Monty Python and the Holy Grail, ve které Rytíři Kulatého stolu tlučou kokosovými ořechy o sebe aby napodobili zvuk jezdce na koni. Jméno Coconut bylo zvoleno jako odkaz na skutečnost, že Python je rovněž nazván podle Monty Python.

## 1.19 Kdo vytvořil Coconut?

Evan Hubinger is an undergraduate student studying mathematics and computer science at Harvey Mudd College. He can be reached by asking a question on Coconut's Gitter chat room, through email at evanjhub@gmail.com, or on LinkedIn.

KAPITOLA 2

# Tutoriál

## 2.1 Úvod

Vítejte v tutoriálu pro Coconut Programming Language! Coconut je varianta Pythonu vytvořená pro **jednoduché, elegantní Pythonické functionální programování**.

Proč používat Coconut? Coconut rozšiřuje repertoár programátora v Pythonu o nástroje moderního funkcionálního programování. Kód Coconut běží na obou verzích Pythonu (2/3), činíce tak toto rozdělení věcí minulosti.

Coconut přidává do Pythonu *syntaktickou podporu* pro:

- pattern-matching - vyhledávání shody s předlohou
- algebraic data types (ADT) - algebraické datové typy
- destructuring assignment - rozložené přiřazení
- partial application - částečnou aplikaci
- lazy lists - líné seznamy
- function composition - skládání funkcí
- prettier lambdas - úhlednější lambdy
- infix notation - infixovou notaci
- pipeline-style programming - směrované programování
- operator functions - operátorové funkce
- tail recursion optimization - optimalizace koncové rekurze
- parallel programming - paralelní programování

a mnoho dalšího!

### 2.1.1 Instalace

Ve své podstatě je Coconut kompilátor, který převádí kód v Coconut na kód v Pythonu. To znamená, že tam, kde lze použít skript Pythonu, lze také použít skript Coconut. Pro přístup k tomuto kompilátoru poskytuje Coconut utilitu CLI (command line interface), která dovede:

- kompilovat jednotlivé soubory nebo celé projekty,
- překládat za pochodu kód Coconut,
- včlenit se (hook into) do existujících aplikací Pythonu, jako IPython/Jupiter a MyPy.

Instalace Coconut je velmi jednoduchá:

1. instalujte Python,
2. otevřte konzolu s příkazovým řádkem
3. a zadejte:

```
pip install coconut
```

*Note: Setkáváte-li se s chybami, zkuste spustit výše uvedený příkaz s flagem* `--user`*. Ujistěte se, že umístění instalace Coconut (v Unixu* `/usr/local/bin` *pokud jste nepoužil* `--user` *nebo* `${HOME}/.local/bin/`*) pokud ano je uvedeno v proměnné prostředí* `PATH`*. Pokud se při instalaci pomocí* `pip`*stále vyskytují chyby, můžete instalovat Coconut pomocí* `conda` *podle těchto pokynů.*

Pro kontrolu, že instalace proběhla správně, zkuste na příkazový řádek zadat

```
coconut -h
```

což by mělo zobrazit nápovědu pro Coconut.

*Note: If you're having trouble installing Coconut, or if anything else mentioned in this tutorial doesn't seem to work for you, feel free to ask for help on Gitter and somebody will try to answer your question as soon as possible.*

### 2.1.2 Bez instalace

Chcete-li používat Coconut bez jeho instalování, zkuste online interpreter.

## 2.2 Začínáme

### 2.2.1 Použití překladače

Nyní, když máte Coconut nainstalovaný, zkusíme s ním něco provádět. Překladač (interpret) spustíte z příkazového řádku zápisem

```
coconut
```

načež byste měl číst něco jako

```
Coconut Interpreter:
(type 'exit()' or press Ctrl-D to end)
>>>
```

což je oznámení Coconut, že je připraven pro zadávání a vyhodnocování kódu. Tož pusťme se do toho!

Pro případ, že jste to dříve přehlédli - *veškerý platný Python 3 je platný Coconut*. To neznamená, že kompilovaný Coconut poběží pouze na Python 3, protože poběží stejně i na Python 2, ale že pouze kód Python 3 je spolehlivě kompilován do kódu Coconut.

Z toho vyplývá, že jste-li důvěrně seznámen s Pythonem, jste již z větší části seznámen se skladbou Coconut a jeho celou standardní knihovnou. Zkusme pro ukázku zadat nějaký jednoduchý kód Pythonu do překladače Coconut:

```
>>> "hello, world!"
'hello, world!'
>>> 1 + 1
2
```

### 2.2.2 Psaní zdrojových souborů

Zajisté - být schopen za pochodu interpretovat kód Coconutu, je báječné - bylo by to však málo prospěšné bez schopnosti psát a kompilovat větší programy. Sestavení jednoduchého programu "hello, world!"si nyní ukážeme.

Nejprve vytvoříme soubor, do něhož svůj kód vložíme. Přípona zdrojových souborů pro Coconut je `.coco`, takže vytvořte soubor `hello_world.coco`. Poté byste měl věnovat čas nastavení vašeho textového editoru na řádné zvýrazňování kódu Coconut. Příslušné pokyny naleznete v dokumentaci Zvýraznění skladby.

Nyní do našeho souboru `hello_world.coco` vložíme kód. Na rozdíl od Pythonu, jehož záhlaví jsou obvyklá a často nezbytná,

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
from __future__ import print_function, absolute_import, unicode_literals, division
```

provede to kompilátor Coconut za nás automaticky, takže se o formální nezbytnosti vůbec nemusíme starat a můžeme se věnovat přímo svému kódu. Vložíme tedy kód našeho programu "hello, world!".

V Python 3 je text programu tento:

```
print("hello, world!")
```

Byť tento kód chodí v Coconut také, lze jej specielně pro Coconut napsat v alternativní pojítkové formě (pipeline-style):

```
>>> "hello, world!" |> print
```

Na příkladu pěkně vidíme, jak operátor Coconutu `|>` umožňuje pojítky směrované (pipeline-style) programování: předává objekt z funkce do funkce s postupně odlišnými operacemi. V tomto případě vkládáme objekt `"hello, world!"` do operace `print`. Uložme nyní náš jednoduchý program "hello, world!"program a zkusme jej spustit.

### 2.2.3 Použití kompilátoru

Kompilování souborů a projektů utilitou CLI je vemi jednoduché. Nacédujeme (`cd`) se do adresáře se souborem `hello_world.coco` a zapíšeme

```
>>> coconut hello_world.coco
```

což vytvoří výstup

```
Coconut: Compiling         hello_world.coco ...
Coconut: Compiled to       hello_world.py .
```

a nově vytvořený soubor `hello_world.py` vloží do stejného adresáře jako `hello_world.coco`. Potom je možné tento soubor spustit příkazem

```
>>> python hello_world.py
```

což by mělo v konzole vyprodukovat výstup `hello, world!`.

_Note: Můžete provést kompilaci a spuštění v jednom kroku, použijete-li flag `--run` (zkráceně `-r`).

Kompilování jednotlivých souborů postupně není jediný způsob jejich kompilace. Můžeme také kompilovat všechny soubory v daném adresáři najednou a to pouhým uvedením názvu adresáře jako prvního argumentu. Máme-li například soubory `havel.coco` a `ravel.coco` v adresáři `F:\codetest\coconut\compil`, zařídíme jejich kompilaci příkazem

```
F:\codetest\coconut\compil> coconut ./
Compiling        havel.coco ...
Compiled to      havel.py
Compiling        ravel.coco ...
Compiled to      ravel.py
```

V adresáři `compil` se navíc vytvoří soubor `__coconut__.py`.

Kompilátor si sám vyhledá všechny kompilovatelné soubory a vytvoří pomocný soubor `__coconut__.py`, do něhož uloží potřebné informace z jednotlivých souborů.

Kompilátor Coconutu podporuje velké množství různých kompilačních možností - viz nápověda `coconut -h`. Nejužitečnější z nich je opce `--linenumbers` (nebo zkráceně `-l`), která přidává čísla řádků ze zdrojového kódu do kompilovaného kódu, umožňujíce tak při ladění vidět číslo zdrojového kódu, odpovídající chybujícímu řádku kompilovaného kódu.

_Note: Nepotřebujete-li plnou kontrolu kompilátoru, můžete použít automatickou kompilaci.

### 2.2.4 Použití IPython/Jupyter

Coconut usiluje o rozsáhlou podporu zavedených nástrojů pro vědecké výpočty v Pythonu.

Za tím účlem poskytuje Coconut podporu aplikace IPython/Jupiter. Pro spuštění notebooku Jupytera s Coconut jako jádrem, použijete příkaz

```
coconut --jupyter notebook
```

### 2.2.5 Případové studie

Protože byl Coconut vytvořen se záměrem aby byl užitečný, bude nejlépe jej předvést v akci při řešení konkrétních problémů, které jsou v tomto tutoriálu označeny jako případové studie.

Tyto případové studie ovšem nepřinášejí úplný přehled všech vlastností Coconut. Ten lze nalézt v obsáhlé dokumentaci.

## 2.3 Případová studie 1: `factorial`

V první studii budeme definovat funkci `factorial`, to jest funkci, která počítá součin `n!`, kde `n` je celé číslo `>= 0`. To je poněkud dětinský příklad, protože tuto úlohu zvládne Python snadno také ale poslouží k demonstraci některých základních vlastností Coconut a jejich výhodného použití.

Nejprve musíme rozhodnout, jaký způsob výpočtu faktoriálu budeme chtít. Možných způsobů řešení je více ale pro jednoduchost se omezíme na čtyři hlavní kategorie: imperativní, recurzivní, iterativní a s použitím `addpattern`.

### 2.3.1 Imperativní metoda

Imperativní přístup bychom při psaní `factoriálu` použili v jazyce typu C. Imperativní přístupy zahrnují mnohé změny stavu, kdy jsou pravidelně měněny proměnné při procházení smyčkou. Imperativní přístup v Coconut k problému `factorial` vypadá nějak takto:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    if n `isinstance` int and n >= 0:
        acc = 1
        for x in range(1, n+1):
            acc *= x
        return acc
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print # TypeError
0.5 |> factorial |> print # TypeError
0 |> factorial |> print # 1
3 |> factorial |> print # 6
```

Předtím, než se budeme podrobně zabývat průběhem výpočtu, prověřme si nejprve jeho testovací případy. Kdybychom psali skutečný program, uložili bychom jej do souboru, jenž bychom kompilovali ale protože si jenom zkoušíme věci, vystačíme si s překopírováním kódu do překladače. Měli bychom dostat dvakrát `TypeError`, potom `1` a `6`.

Nyní, když jsme si ověřili, že nám kód chodí správně, pohleďmě o co v něm kráčí. Protože je imperativní přístup zcela nefunkcionální, Coconut nám v tomto případě příliš nepomůže. Avšak i zde použití infixové notace (vložení funkce mezi své argumenty `n` a `int`: `n `isinstance` int` ) činí kód čistší a čitelnější.

### 2.3.2 Rekurzivní metoda

Rekurzivní přístup je první ze zcela funkcionálních přístupů a to v tom, že nezahrnuje změnu stavu a smyčky jako u imperativního přístupu. Rekurzivní přístup se vyhýbá potřebě měnit proměnné tím, že tato změna je implicitně zahrnuta v rekurzivním volání funkce. Zde je rekurzivní přístup k problému `factorial`:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return 1
        match x is int if x > 0:
            return x * factorial(x-1)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")
```

```
# Test cases:
-1 |> factorial |> print # TypeError
0.5 |> factorial |> print # TypeError
0 |> factorial |> print # 1
3 |> factorial |> print # 6
```

Překopírujte si kód a testy do překladače. Měl byste dostat stejné vysledky jako v imperativní verzi.

Proberme si specifika syntaxe v tomto příkladu. Příkaz `case n:` spouští blok se dvěma příkazy `match`. Každý příkaz `match` se pokouší porovnat svou deklaraci (pattern) s argumentem bloku `case`. Příkaz `else` se provede jen v případě absence jakékoliv shody.

Konkretně v tomto příkladě ověřuje `match`, zda je `n` shodné s `0`. Pakliže ano, provede se `return 1`. Pokud ne, prověřuje se druhý `match`, v němž je zavedena lokální poměnná `x` s počáteční hodnotou `x = n` a v níž je opakovaně (rekurzivně) volána funkce `factorial(x)` pro snižující se hodnotu argumentu. V okamžiku, kdy `x=1`, vrátí příkaz `return` součin čísel `1` až `n-1`. Pokud se neprovede žádný z obou příkazů, příkaz `else` spustí a provede `raise TypeError("argument faktoriálu musí být celé číslo >= 0")`.

I když je tento příklad velmi prostý, je postup v něm použitý, jedním z nejmocnějších i nejsložitějších postupů v Coconut. Tento postup se nazývá **pattern-matching** neboli *porovnávání s předlohou*. Jak jsme viděli, pivotním slovem v tomto konstruktu je klíčové slovo `match`, které jsme v našem příkladě používali opakovaně pro ověření různých případů (`case`).

Jako intuitivní vodítko si lze představit *přiřazení* tam, kde vidíme klíčové slovo `match`. Případně si lze uvědomit, že všechny příkazy `match` mohou být konvertovány na ekvivalentní příkazy rozkladného (destructuring) přiřazení, které jsou rovněž platným konstruktem Coconut. V tomto případě by ekvivalentním `rozkladným přiřazením` k funkcí `factorial` nahoře bylo:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    try:
    # Jediná hodnota, kterou lze přiřadit k 0 je 0, protože 0
    # je neměnitelná konstanta; proto přiřazení selže pokud n=!0:
       0 = n
    except MatchError:
       pass
    else:
       return 1
    try
    # To se pokusí přiřadit n k x, jež bylo deklarováno jako
    # int; protože k int může být přiřazen pouze int,            # následující␣
↪podmínka selže, nebude-li n celým číslem:
       x is int = n
    except MatchError:
       pass
    else: if x > 0:   # v Coconut lze ze else použít if, match, try
       return x * factorial(x-1)
    raise TypeError("argumentem pro faktorial musí být int >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Nejprve copy and paste! I když by toto rozkladné přiřazení mělo pracovat, je mnohem nemotornější než příkaz `match` v případě, že očekáváte, že by mohlo dojít k selhání, což je důvod pro existenci příkazu `match`. Ekvivalent rozkladného (destructuring) přiřazení však objasňuje, co přesně pattern-matching dělá - ukazujíc na to, že příkazy `match`

jsou vlastně něco jako příkazy rozkladného přiřazení.

In fact, to be explicit about using destructuring assignment instead of normal assignment, the `match` keyword can be put before a destructuring assignment statement to signify it as such.

Při používání příkazů pro pattern-matching a destructuring assignment v dalších uázkách bude užitečné, když si pomyslíme *přiřazení* pokaždé, když uvidíme klíčové slovo `match`.

Dalším snadným vylepšením naší funkce `factorial` je použití žolíkového označení _. Vlastně nepotřebujeme přiřadit `x` jako novou proměnnou, protože má stejnou hodnotu jako `n`, takže když použijeme _ místo `x`, Coconut tuto proměnnou vlastně nikdy nepřiřadí. Naši funkci `factorial` můžeme tedy přepsat takto:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return 1
        match _ is int if n > 0:
            return n * factorial(n-1)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print  # TypeError
0.5 |> factorial |> print  # TypeError
0 |> factorial |> print  # 1
3 |> factorial |> print  # 6
```

Copy, paste! Tato nová funkce `factorial` by se měla chovat úplně stejně jako předtím.

Až dosud jsme se u rekurzivní metody zabývali pouze porovnáním předlohy (pattern matching) ale ve skutečnosti existuje další způsob, jímž můžeme vylepšit naši funkci `factorial`. Coconut provádí automatickou optimalizaci koncového volání, což znamená že kdykoli funkce přímo vrací volání jiné funkce, zadrží (optimalizuje) Coconut další volání. Naši funkci `factorial` tedy přepíšeme pro použití koncového volání (tail call):

```
def factorial(n, acc=1):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return acc
        match _ is int if n > 0:
            return factorial(n-1, acc*n)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print # TypeError
0.5 |> factorial |> print # TypeError
0 |> factorial |> print # 1
3 |> factorial |> print # 6
```

Copy, paste! Tato nová funkce `factorial` je ekvivalentní originální verzi s tou výjimkou, že nikdy nevyvolá `RuntimeError` v důsledku dosažení maximální hloubky rekurze v Pythonu, protože Coconut odstaví (optimalizuje) koncové rekurzivní volání.

### 2.3.3 Iterativní metoda

Tato metoda je dalším funkcionálním přístupem k řešení problému. Iterativní přístupy obcházejí potřebu změny stavu a smyček použitím funkcí vyššího řádu, které jako argumenty přijímají jiné funkce jako `map` a `reduce` k vyčlenění základních prováděných operací. Iterativní přístup k `factoriálu` v Coconut je tento:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return 1
        match _ is int if n > 0:
            return range(1, n+1) |> reduce$(*)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")

# Test cases:
-1 |> factorial |> print # TypeError
0.5 |> factorial |> print # TypeError
0 |> factorial |> print # 1
3 |> factorial |> print # 6
```

Copy, paste! Tato definice se od rekurzivní definice liší pouze v jednom řádku, což je záměrné, protože jak iterativní, tak rekurzivní přístupy jsou funkcionální. Odlišný řádek je tento:

```
return range(1, n+1) |> reduce$(*)
```

Rozložme si, co se v tomto řádku odehrává. Nejprve funkce `range` vytvoří iterátor pro všechna čísla, která mají být mezi sebou vynásobena. Ten je postoupen (piped) funkci `reduce$(*)`, která násobení provede. Ale jak? Co je to `reduce$(*)`?

Funkce `reduce` existovala jako vestavěná funkce v Python 2 a Coconut ji nyní přivádí zpět. `reduce` je funkce vyššího řádu, která přijímá jako svůj první argument funkci pro dva argumenty a iterátor jako svůj druhý argument (viz další ukázka), načež aplikuje přijmutou funkci na daný iterátor počínaje jeho prvním elementem a voláním funkce pro dosud akumulované volání a další element, dokud není iterátor vyčerpán. Zde je vizuální reprezentace:

```
reduce(f, (a, b, c, d))

acc                 iter
                    (a, b, c, d)
a                   (b, c, d)
f(a, b)             (c, d)
f(f(a, b), c)       (d)
f(f(f(a, b), c), d)

return acc
```

Nyní pohleďme, jak jsme doplnili funkci `reduce` aby pronásobila všechna čísla, která ji dodáme. Úplný výraz měl tvar `reduce$(*)`. V tomto zápise jsou použity dva konstrukty Coconut a sice operátorová funkce pro násobení ve tvaru `(*)` a příkaz k částečné aplikaci ve tvaru `$`.

Nejprve operátorová funkce. Operátorová funkce se v Coconut vytvoří uzavřením operátoru do závorek. V tomto případě je `(*)` zhruba ekvivalentní výrazu v Pythonu: `lambda x, y: x*y`. Ve skladbě lambdy v Coconut je `(*)` rovněž ekvivalentí zápisu `(x, y) -> x*y`, jenž budeme odteďka používat pro všechny lambdy, byť obě formy jsou v Coconut legální.

*Note: Kdybychom povolili režim `--strict`, jenž hlídá náš kód z hlediska úpravy textu, dostali bychom chybové hlášení, kdykoliv bychom použili příkaz `lambda` Pythonu.*

Nyní k částečné aplikaci funkce. Lze si myslet, že částečná aplikace je *volání líné funkce* s operátorem lenosti `$`, kde *lenost* znamená: "nevyhodnocuj, dokud nemusíš". Je-li v Coconut volání funkce předznamenáno znakem `$`, jako v tomto případě, je normální provedení funkce nahrazeno novou funkcí s již poskytnutými argumenty, takže je funkce volána jak pro částečně použité argumenty, tak pro nové argumenty (v tomto pořadí). V tomto případě je `reduce$(*)` ekvivalentní k `(*args, **kwargs) -> reduce((*), *args, **kwargs)`.

Spojíme-li to vše dohromady, vidíme jak jediný řádek kódu

```
range(1, n+1) |> reduce$(*)
```

je schopen spočítat celý faktoriál bez použití stavů či smyček, pouze s použitím funkcí vyššího řádu funkcionálním stylem.

S nástroji Coconut, které zde používáme, jako je částečná aplikace (`$`), usměrněné (pipeline-style) programování (`|>`), funkce vyššího řádu (`reduce`) a operátorové funkce (`(*)`) je možné sestavovat funkcionální programy snadno a úhledně.

### 2.3.4 Metoda `addpattern`

I když je iterativní přístup velmi přehledný, je stále zapotřebí tří úrovní odsazení abychom se dostali od záhlaví funkce k vlastnímu vracenému objektu:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    case n:
        match 0:
            return 1
        match _ is int if n > 0:
            return range(1, n+1) |> reduce$(*)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")
```

Použijeme-li vestavěnou funkci `addpattern`, můžeme zredukovat tři identační úrovně na jednu. Pohleďte:

```
def factorial(0):
    return 1

@addpattern(factorial)
def factorial(n is int if n > 0):
    """Compute n! where n is an integer >= 0."""
    return range(1, n+1) |> reduce$(*)

# Test cases:
-1 |> factorial |> print # MatchError
0.5 |> factorial |> print # MatchError
0 |> factorial |> print # 1
3 |> factorial |> print # 6
```

Copy, paste! Tato verze by měla pracovat stejně jako předchozí, až nato že místo `TypeError` vrací hlášení `MatchError`. Máme zde tři nové koncepty k prodiskutování: `addpattern`, zápis přiřazovací funkce a definici funkce pro porovnání předlohy (pattern-matching).

*Nejprve* zápis přiřazovací funkce. To je docela prosté. Je-li funkce definována s rovnítkem = místo dvojtečkou `:`, musí být poslední řádek výrazem, jenž je také automaticky vracen.

*Dále* definice porovnávací (pattern-matching) funkce. Tato definice zajišťuje provedení přesně toho, co je uvedeno v názvu - porovnání všech argumentů funkce se zadaným vzorem. Pokud se vzor neshoduje s žádným z argumentů

(nebo je-li zadán nesprávný počet argumentů), vyvolá funkce chybové hlášení `MatchError`. Chcete-li explicitně deklarovat definici p-m funkce, můžete přidat `match` před `def`.

*Za třetí*, `addpattern`. Dekorátor `addpattern` přijímá jako argument předtím definovanou p-m funkci a vrací dekorátor, který dekoruje novou m-p funkci přidáním nového vzoru jako další případ (case) ke starým vzorům. Dekorátor `addpattern` dělá tedy přesně to, co říká - přidává další vzor k existující p-m funkci.

Dekorátorem `addpattern` můžeme přepsat nejenom imperativní přístup, jak jsme právě provedli, ale můžeme také přepsat rekurzivní přístup, jak vidno zde:

```
def factorial(0) = 1

@addpattern(factorial)
def factorial(n is int if n > 0):
    """Compute n! where n is an integer >= 0."""
    return n * factorial(n - 1)

# Test cases:
-1 |> factorial |> print # MatchError
0.5 |> factorial |> print # MatchError
0 |> factorial |> print # 1
3 |> factorial |> print # 6
```

Copy, paste! Mělo by to chodit stejně jako předtím, kromě toho, že (stejně jako předtím) je `TypeError` nahrazen `MatchError`.

## 2.4 Případová studie 2: `quick_sort`

Ve druhé případové studii budeme používat algoritmus quick sort. Použijeme dvě verze: funkci `quick_sort`, která přijímá i vrací seznam a tutéž funkci, která přijímá i vrací iterátor.

### 2.4.1 Třídění sekvence

Nejprve `quick_sort` pro seznamy. Použijeme rekurzivní přístup založený na dekorátoru `addpattern` - podobný k přístupu, použitého u posledně psané funkce `factorial` k omezení počtu odsazení. Bez dalších okolků, zde je naše implementace `quick_sort` pro seznamy:

```
def quick_sort([]) = []


@addpattern(quick_sort)
def quick_sort([head] + tail) =
    """Sort the input sequence using the quick sort algorithm."""

    (quick_sort([x for x in tail if x < head])
        + [head]
        + quick_sort([x for x in tail if x >= head]))

# Test cases:
[] |> quick_sort |> print # []
[3] |> quick_sort |> print # [3]
[0,1,2,3,4] |> quick_sort |> print # [0,1,2,3,4]
[4,3,2,1,0] |> quick_sort |> print # [0,1,2,3,4]
[3,0,4,2,1] |> quick_sort |> print # [0,1,2,3,4]
```

Copy, paste! Zde jsou pouze dvě nové věci: head-tail pattern-matching a příkazy `where`.

Příkazy `where` velmi průzračné a jistě jste jejich působení sami objevili. Příkaz `where` je způsob výpočtu pro vložené upřesnění.

Head-tail pattern-matching ('porovnání předlohy od čela až po chvost'), zde vidíme jako `[head] + tail`, jenž má formu seznamu nebo entice přidanou k proměnné. Když se tato forma vyskytne v jakémkoli p-m kontextu, je s porovnávanou hodnotou zacházeno jako se sekvencí, seznamem nebo enticí porovnávanou s počátkem této sekvence, jejíž zbytek je vázán k proměnné. V tomto případě používáme schema head-tail, abychom odstranili čelo, jež můžeme použít jako pivot pro rozštěpení zbytku seznamu.

### 2.4.2 Třídění iterátoru

Nyní vyzkoušíme `quick_sort` pro iterátory. Náš způsob řešení problému bude kombinace rekurzivního a iterativního přístupu, jež jsme použili u `factoriálu`, a sice v tom, že budeme lenivě a rekurzivně vytvářet iterátor. Zde je kód:

```
def quick_sort(l):
    """Sort the input iterator, using the quick sort algorithm"""

    match [head] :: tail in l:
        tail = reiterable(tail)
        yield from (quick_sort((x for x in tail if x < head))
            :: (head,)
            :: quick_sort((x for x in tail_ if x >= head))
            )
    # We implicitly return an empty iterator here if the match falls through.
↪

# Test cases:
[] |> quick_sort |> list |> print # []
[3] |> quick_sort |> list |> print # [3]
[0,1,2,3,4] |> quick_sort |> list |> print # [0,1,2,3,4]
[4,3,2,1,0] |> quick_sort |> list |> print # [0,1,2,3,4]
[3,0,4,2,1] |> quick_sort |> list |> print # [0,1,2,3,4]
```

Copy, paste! Tento `quick_sort` algoritmus používá řadu nových konstruktů, takže hrr na ně.

Nejprve je to operátor `::`, který se zde objevuje jak v porovnávání shody (pattern-matching), tak samostatně. V podstatě to je líný operátor `+` pro iterátory, který spojuje nebo řetězí líně dva iterátory, nic nevyhodnocujíc, není-li žádáno; lze jej použít pro vytváření nekonečných iterátorů. V porovnání shody tuto operaci invertuje, rozkládaje (destructuring) počátek iterátoru na předlohu a zbytek, který váže k proměnné.

Což nás přivádí k další nové věci, zápisu `match ... in ...`. Zápis

```
match pattern in item:
    <body>
else:
    <else>
```

je zkratka pro

```
case item:
    match pattern:
        <body>
else:
    <else>
```

která eliminuje potřebu další úrovně identace při provedení pouze jednoho `match`.

Třetím novým konstruktem je `reiterable`. Při realizaci neměnitelného funkcionálního programování s iterátory Pythonu se vyskytuje tento problém: kdykoliv se přistoupí k elementu iterátoru, je ztracen. Procedura `reiterable` umožňuje, aby byl volaný iterábl iterován opakovaně, poskytujíc pro stejné vstupy stejné výstupy.

Konečně, byť se nejedná o nový konstrukt, protože existuje v Python 3, naše použití `yield from` si zasluhuje zmínky. V Pythonu se příkaz `yield`, který pracuje podobně jako `return`, používá k vytváření iterátorů - s tou výjimkou, že se `yield` může vyskytnout vícekrát , pokaždé vraceje jiný element. Forma `yield from` je velmi podobná, až na to, že místo přidání jediného elementu do vytvářeného iterátoru přidává jiný celý iterátor.

Spojíme-li to všechno dohromady, máme zde opět naši funkci `quick_sort`:

```
def quick_sort(l):
    """Sort the input iterator, using the quick sort algorithm."""
    match [head] :: tail in l:
        tail = reiterable(tail)
        yield from (quick_sort((x for x in tail if x < head))
            :: (head,)
            :: quick_sort((x for x in tail_ if x >= head))
            )
    # We implicitly return an empty iterator here if the match falls through.
```

Funkce se nejprve pokouší rozdělit (split) seznam `l` na počáteční element a zbývající iterátor. Je-li `l` prázdným iterátorem, porovnání selže, poskytujíce prázdný iterátor (takto funkce ošetřuje základní případ). V opačném případě vytváříme kopii zbytku iterátoru a poskytujeme (yield) spojení: (quick-sort všech zbývajících elementů menších než počáteční element) + (počáteční element) + (quick-sort všech zbyvajících elementů větších než počáteční element).

Výhody zde použitého základního přístupu s četným použitím iterátorů a rekurzí, v porovnání s klasickým imperativním přístupem, jsou mnohé. Za prvé je náš přístup čistší a čitelnější, protože popisuje co **je** `quick_sort` místo **jak** by měl být použit. Za druhé je náš přístup *líný* v tom, že náš `quick_sort` nic nevyhodnocuje bez vyžádání. A konečně, byť to není relevantní pro `quick_sort`, je to relevantní v mnoha jiných případech, jejichž příklady ještě v tomto tutoriálu uvidíme, náš přístup umožňuje pracovat s *nekonečnými* řadami jako by byly skutečně nekonečné.

Coconut činí programování s takto výhodným funkcionálním přístupem výrazně snadnější. V tomto příkladě nám `pattern-matching` Coconutu umožňuje snadné dělení daného iterátoru a jeho slučovací operátor `::` nám umožňuje jej vrátit zpět ve srovnaném pořadí.

## 2.5 Případová studie 3: `vector` - část I

V následující případové studii budeme provádět něco lehce odlišného - místo definování funkce budeme vytvářet objekt. Konkrétně se budeme pokoušet vytvořit neměnitelný n-vektor, který podporuje všechny základní vektorové operace.

Ve funkcionálním programování je často žádoucí definovat *neměnitelné* objekty, jež nelze po vytvoření měnit, jako jsou řetězce a entice Pythonu. Stejně jako řetězce a entice (tuples) jsou neměnitelné objekty užitečné z celé řady důvodů:

- lze o nich snadněji uvažovat, protože víme že se nemění,
- jsou 'hashable and pickleable', takže je lze použít jako klíče a serializovat,
- jsou výrazně efektivnější, protože vyžadují mnohem méně doprovodných aktivit,
- při kombinaci s 'pattern-matching' mohou být použity jako takzvané **algebraické datové typy** ke snadnému vytváření velkých a složitých datových struktur.

### 2.5.1 2-vector

Příkaz `data` v Coconut přivádí do Pythonu mocnou utilitu *neměnitelných algebraických datových typů*. Skladbu příkazu `data` si ukážeme na definici jednoduchého dvouprvkového vektoru. Tento vektor bude mít specielní metodu `__abs__`, která spočítá jeho délku, definovanou jako odmocninu součtu čtverců jeho prvků. Zde je:

```
data vector2(x, y):
    """Immutable 2-vector."""
    def __abs__(self):
        """Return the magnitude of the 2-vector."""
        return (self.x**2 + self.y**2)**0.5

# Test cases:
vector2(1, 2) |> print # vector2(x=1, y=2)
vector2(3, 4) |> abs |> print # 5
vector2(1, 2) |> fmap$(x -> x*2) |> print  # vector2(x=2, y=4)
v = vector2(2, 3)
v.x = 7 # AttributeError
```

Copy, paste! Tento příklad ukazuje základní skladbu příkazů `data`:

```
data <name>(<attributes>):
    <body>
```

kde `<name>` a `<body>` znamenají totéž jako v ekvivalentní definici `class`, avšak `<attributes>` jsou zde různé atributy definovaného datového typu, jež může konstruktor přijmout jako argumenty. V tomto případě je `vector2` datový typ se dvěma atributy `x` a `y`, s jednou metodou `__abs__`, která počítá jeho délku. Jak ukazují testovací případy, instance datového typu `vector2` lze vytvářet, tisknout, nikoliv však měnit.

Další věcí, na kterou je zde zapotřebí upozornit, je použití funkce `fmap`. `fmap` umožňuje `mapovat` funkce po algebraických datových typech. Datové typy Coconut podporují iteraci, takže standardní `map` s nimi může pracovat ale nevrací jiný objekt téhož datového typu. Konstrukt `fmap` je jednoduše `map` plus volání konstruktoru objektu.

### 2.5.2 Konstruktor pro n-Vector

Nyní, když jsme vyřídili `2-vector`, vraťme se zpět k našemu původnímu, více komplikovanému problému s n-vektory, to jest s vektory libovolné délky. Pokusíme se, aby náš n-vektor podporoval všechny základní vektorové operace ale začneme pouze s definicí `data` a konstruktorem:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor

# Test cases:
vector(1, 2, 3) |> print # vector(*pts=(1, 2, 3))
vector(4, 5) |> vector |> print # vector(*pts=(4, 5))
```

Copy, paste! Nově se zde dozvídáme, jak psát konstruktory `data`. Protože jsou typy `data` neměnitelné, nebude zde chodit konstrukce `__init__`. Místo toho je použita jiná specielní metoda `__new__`, která musí vrátit nově vytvořenou instanci a na rozdíl od většiny metod přijímá jako první argument class, nikoliv objekt. Protože `__new__` potřebuje vrátit úplnou instanci, bude ve většině případů nezbytný přístup k výchozímu konstruktoru `data`. Pro tento

účel poskytuje Coconut vestavěnou funkci `makedata`, která přijímá datový typ a volá jeho výchozí konstruktor pro zbytek argumentů.

V tomto případě konstruktor kontroluje, zda nebylo zadáno nic jiného než další `vector`, v kterémžto případě jej vrací. Jinak vrací výsledek volání výchozího konstruktoru, jehož formou je `vector(pts)`, neboť takto jsme deklarovali datový typ. Používáme sekvenční p-m ke zjištění, zda jsme zadali jediný vektor, což je pouze seznam nebo entice vzorů, s nimiž je porovnáván obsah sekvence.

Dalším novým konstruktem zde použitým je operátor `|*>` neboli star-pipe, který funguje úplně stejně jako normální pojítko, kromě toho, že místo volání funkce s jedním argumentem, volá ji tolika argumenty, kolik je elementů v zadané sekvenci. Rozdíl mezi `|*>` a `|>` je analogický rozdílu mezi `f(args)` a `f(*args)`.

### 2.5.3 Metody pro n-vector

Nyní, když máme konstruktor pro náš n-vektor, je čas napsat jeho metody. První je metoda `__abs__`, která má počítat velikost vektoru. To bude mírně složitější než u 2-vektoru, protože musí chodit pro libovolný počet `pts`. Naštěstí můžeme použít pojítkový (pipeline) styl Coconutu a jeho částečnou aplikaci funkce:

```
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
```

Základním algoritmem zde je 'mapování' mocniny pro každý prvek, jejich celkový součet a druhá odmocnina výsledku. Novým konstruktem zde je znak `?` v částečné aplikaci, který jednoduše umožňuje přeskočení jednoho argumentu a odložení jeho použití na volání funkce. V tomto případě nám `?` umožňuje částečně aplikovat exponent místo základu ve funkci `pow` - mohli jsme rovněž ekvivalentně použít `(**)`.

Dále je sčítání vektorů. Cílem je sečíst dva vektory stejné délky sečtením jejich komponent. Za tím účelem využijeme schopnost Coconutu provést pattern-matching (nebo v tomto případě rozkladné přiřazení) pro datové typy a to takto:

```
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
```

Máme zde několik nových konstruktů ale nejvýznamnějším je pattern-matching pro `vector(*other_pts)`, na němž vidíme skladbu pro porovnávání shody s datovými typy: přesně napodobuje originální deklaraci `data` pro daný datový typ. V tomto případě se `vector(*other_pts)` bude shodovat pouze s vektorem, přičemž přiřadí atribut `pts` vektoru proměnné `other_pts`. Pokud se nebude shodovat, vyvolá chybu `MatchError`.

Další metodou je podíl vektorů, což je vlastně součet vektorů se znaménkem `(-)` místo `(+)`:

```
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
```

Za povšimnutí zde stojí to, že na rozdíl od jiných operátorových funkcí, může `(-)` znamenat buď odečtení nebo negaci. Konkretní význam závisí na počtu poskytnutých argumentů - jeden pro negaci, dva pro odečtení. Abychom si to demonstrovali, použijeme stejnou funkci `(-)` k zavedení negace vektoru, což by mělo negovat každý jeho prvek:

```
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
```

Poslední metodou, kterou zavedeme, je násobení vektorů. To je poněkud komplikované, neboť matematicky existuje více způsobů. Pro naše účely se soustředíme na dva: na (skalární) součin dvou vektorů stejné délky, což se součet

součinů příslušných elementů a na násobení vektoru skalárem, což je násobení všech elementů stejným skalárem. Zde je naše implementace:

```
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector # scalar multiple
    def __rmul__(self, other):
        """Necessary to make scalar multiplication commutative."""
        return self * other
```

Za pozornost zde stojí za prvé, že na rozdíl od součtu a podílu, kde jsme chtěli hlásit chybu při selhání shody vektoru, zde chceme při selhání shody provést násobení skalárem - takže místo použití rozloženého přiřazení použijeme příkaz `match`.

Za druhé si povšimneme použití kombinace pojítkového (pipeline) stylu programování, částečné aplikace, operátorových funkcí a funkcí vyššího řádu pro výpočet skalárního součinu a pro násobení skalárem. U skalárového součinu mapujeme násobení na dva vektory a sečteme výsledky. U násobení skalárem vytváříme nový vektor násobením všech prvků původního vektoru stejným číslem.

Nakonec to vše dáme dohromady:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other

# Test cases:
```

```
vector(1, 2, 3) |> print  # vector(*pts=(1, 2, 3))
vector(4, 5) |> vector |> print  # vector(*pts=(4, 5))
vector(3, 4) |> abs |> print  # 5
vector(1, 2) + vector(2, 3) |> print  # vector(*pts=(3, 5))
vector(2, 2) - vector(0, 1) |> print  # vector(*pts=(2, 1))
-vector(1, 3) |> print  # vector(*pts=(-1, -3))
(vector(1, 2) == "string") |> print  # False
(vector(1, 2) == vector(3, 4)) |> print  # False
(vector(2, 4) == vector(2, 4)) |> print  # True
2*vector(1, 2) |> print  # vector(*pts=(2, 4))
vector(1, 2) * vector(1, 3) |> print  # 7
```

Copy, paste! Je to pěkná řádka řádků. Když si to však poučeně procházíme, je to čisté, čitelné a stručné a dělá to přesně to, co jsme chtěli aby to dělalo: vytvořit algebraický datový typ pro neměnitelný n-vektor, který podporuje základní vektorové operace. Celou záležitost jsme přitom provedli čistě funkcionálně bez potřeby imperativních konstruktů, jako jsou stavy nebo smyčky.

## 2.6 Případová studie 4: `vector_field`

V poslední případové studii nebudu kód psát já a vy přihlížet, ale budete jej psát vy a já vám posléze ukážu, jak bych to napsal sám.

Bonusovou výzvou u tohoto odstavce bude napsat každou definovanou funkci do jednoho řádku. Zkuste přitom použít přiřazovací funkce:

Nejprve si uveďme obecný cíl této případové studie. Chceme napsat program, který nám umožní vytvářet nekonečná pole vektorů, přes něž můžeme iterovat a s nimiž můžeme provádět různé operace. V tomto případě se budeme zajímat jenom o vektory s kladnými komponenty.

Naším prvním krokem tedy bude vytvoření pole všech bodů s pozitivními hodnotami `x` a `y`, to jest, nalézajících se v prvním kvadrantu roviny `x-y`, které vypadá nějak takto:

```
...

(0,2)   ...

(0,1)   (1,1)   ...

(0,0)   (1,0)   (2,0)   ...
```

Protože chceme být schopni přes tuto rovinu iterovat, potřebujeme ji nějakým způsobem linearizovat a nejjednodušším způsobem to učiníme tak, že ji rozdělíme do diagonál, načež můžeme traverzovat po první diagonále, potom po druhé a tak dále, nějak takto:

```
(0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0, 2), ...
```

### 2.6.1 `diagonal_line`

Naše první funkce `diagonal_line(n)` by tedy měla vytvořit iterátor všech bodů, reprezentovaných jako souřadnicové entice v `n-té` diagonále, počínaje v bodě `(0, 0)` jako `0th` diagonála. Jak jsme si řekli na počátku případové studie, o řešení se pokusíte nejdřív sami s použitím všech nástrojů funkcionálního programování, které Coconut poskytuje. Až budete hotovi, posuňte se dále.

Zde je několik testů, které můžete použít:

```
diagonal_line(0) `isinstance` (list, tuple) |> print # False (should be an iterator)
diagonal_line(0) |> list |> print # [(0, 0)]
diagonal_line(1) |> list |> print # [(0, 1), (1, 0)]
```

*Nápověda: `n-tá diagonála by měla obsahovat n+1 prvků, zkuste tedy začít s funkcí range(n+1) a posléze ji nějak přetvořit.*

Nebylo to tak hrozné, že ne? Nyní se podívejme na mé řešení:

```
def diagonal_line(n) = range(n+1) |> map$((i) -> (i, n-i))
```

Prostinké, což? Vezmeme `range(n+1)` a použijeme `map` k její transformaci na potřebnou sekvenci entic.

### 2.6.2 linearized_plane

Nyní, když jsme vytvořili naše diagonální přímky, potřebujeme je spojit dohromady abychom sestavili plně linearizovanou rovinu a za tím účelem napíšeme funkci `linearized_plane()`. Funkce `linearized_plane` by měla vytvořit iterátor, který prochází všemi body roviny po diagonálách, počínaje nultou, prvou, atd. Tento iterátor musí být nezbytně nekonečný, protože musí procházet všemi body dané roviny.

Nápovědou pro sestavování funkce budiž připomínka, že operátor `::` je líný a nevyhodnotí své operandy bez požádání, což znamená, že může být použit k vytvoření nekonečných iterátorů. Až budete hotovi, posuňte se v textu dále.

Testy:

```
# Note: tyto testy používají označení $[], které jsme dosud neuvedli
#  ale bude uvedeno později ještě v této případové studii; prozatím proved'te
#  testy a ujistěte se, že dostáváte stejný výsledek jako je v komentáři:
linearized_plane()$[0] |> print # (0, 0)
linearized_plane()$[:3] |> list |> print # [(0, 0), (0, 1), (1, 0)]
```

*Nápověda: místo definování funkce `linearized_plane()`, zkuste ji definovat jako `linearized_plane(n=0)`, kde `n` je označení počáteční diagonály a pro rozvinutí funkce použijte rekurzi.*

To bylo poněkud náročnější než předtím ale doufejme, že ne příliš. Nyní se podívejme na mé řešení:

```
def linearized_plane(n=0) = diagonal_line(n) :: linearized_plane(n+1)
```

Jak vidíte, je to v základě jednoduché řešení: prostě ke spojení diagonál za sebou použijete `::` a rekurzi.

### 2.6.3 vector_field

Nyní, když máme funkci, která vytvoří všechny potřebné body, je čas přeměnit je na vektory a za tím účelem si definujeme novou funkci `vector_field()`, která přemění všechny entice v `linearized_plane` na vektory s použitím třídy `n-vector`, kterou jsme definovali dříve.

Testy:

```
# Budete potřebovat sem přenést třádu vektoru z předchozích textů aby vám chodilo␣
↪následující:
vector_field()$[0] |> print # vector(*pts=(0, 0))
vector_field()$[2:3] |> list |> print # [vector(*pts=(1, 0))]
```

*Nápověda: Vzpomeňte si, že vektor, který jsme definovali, přijímá komponenty jako separátní argumenty, nikoliv jako jedinou entici.* Při řešení vám může být nápomocný text Coconut built-in `starmap`.

Děláme velký pokrok! Než pokročíme dál, srovnejte si řešení se mnou:

```
def vector_field() = linearized_plane() |> starmap$(vector)
```

Vše, co jsme učinili, bylo to, že jsme mapovali `vector` přes funkci `linearized_plane`, avšak s použitím `starmap` místo `map`, takže je `vector` volán s každým elementem entice jako separátním argumentem.

### 2.6.4 Applikace

Nyní, když máme všechny funkce, potřebné pro naše vektorové pole, dáme je všechny dohromady a otestujeme je. Nezdráhejte se dosadit vlastní verze funkcí:

```
data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
        else:
            return self.pts |> map$((*)$(other)) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other

def diagonal_line(n) = range(n+1) |> map$(i -> (i, n-i))
def linearized_plane(n=0) = diagonal_line(n) :: linearized_plane(n+1)
def vector_field() = linearized_plane() |> starmap$(vector)

# Test cases:
diagonal_line(0) `isinstance` (list, tuple) |> print  # False (should be an iterator)
diagonal_line(0) |> list |> print  # [(0, 0)]
diagonal_line(1) |> list |> print  # [(0, 1), (1, 0)]
linearized_plane()$[0] |> print  # (0, 0)
linearized_plane()$[:3] |> list |> print  # [(0, 0), (0, 1), (1, 0)]
vector_field()$[0] |> print  # vector(*pts=(0, 0))
vector_field()$[2:3] |> list |> print  # [vector(*pts=(1, 0))]
```

Copy, paste! Poté, co jste se ujistili, že po dosazení svých funkcí chodí vše jak má, zaměřte se na poslední čtyři testy. Zjistíte, že používají novou notaci, podobnou notaci pro částečnou aplikaci, již jsme viděli dříve - ale s hranatými závorkami místo kulatých. To je notace pro krájení (slicing) iterátoru. Podobně jako byla částečná aplikace líným voláním funkce, je dělení iterátoru *líným dělením sekvence*. Podobně jako u částečné aplikace, je užitečné považovat znak `$` za *zlenivějící* (lazy-ify) operátor, v tomto případě přetvářející normální (ihned prováděné) krájení (slicing) Pythonu na líné krájení iterátoru, které se provádí jen tehdy, jsou-li prvky v řízcích (slice) potřebné.

Maje toto na mysli, nyní když jsme sestavili naše vektorové pole, je čas si s krájením iterátoru trochu pohrát. Zkuste něco smělého, jako například

- vytvořit `magnitude-field`, kde každý bod reprezentuje délku příslušného vektoru
- zkombinovat celá vektorová pole aplikací funkce `match` na dříve vytvořené metody dělení a násobení

potom použít krájení iterátoru pro vynětí a přezkoušení úseků.

## 2.7 Případová studie 5: `vector` - část II

U některých aplikací, používajících naše `vector_fields`, může být žádoucí přidat k našemu `vektoru` nějaké užitečné metody. V této případové studii se zaměříme na metodu, zvanou `.angle`.

Metoda `.angle` přijme dva vektory a spočítá úhel mezi nimi. Matematicky je úhel dvou vektorů skalárním součinem jejich příslušných jednotkových vektorů. Takže před tím, než budeme moci použít metodu `.angle`, budeme potřebovat metodu `.unit`. Matematicky je výraz pro jednotkový vektor daného vektoru dán jako podíl tohoto vektoru a jeho velikosti. Tudíž, před použitím `.unit` a potažmo `.angle`, musíme začít zavedením dělení.

### 2.7.1 `__truediv__`

Dělení vektorů je pouhé skalární dělení, pročež napíšeme metodu `__truediv__`, která přijímá `self` jako první argument a `other` jako druhý argument, vracejíc nový vektor téže velikosti jako `self`, s prvky dělenými vektorem `other`. Jako specielní výzvu, zkuste to zapsat v jediném řádku s použitím notace přiřazovací funkce.

Testy:

```
vector(3, 4) / 1 |> print # vector(*pts=(3.0, 4.0))
vector(2, 4) / 2 |> print # vector(*pts=(1.0, 2.0))
```

*Nápověda: Podívejte se zpět, jak jsme zaváděli násobení skalárem.*

Zde je mé řešení pro vaši kontrolu:

```
    def __truediv__(self, other) = self.pts |> map$(x -> x/other) |*> vector
```

### 2.7.2 `.unit`

Další je `.unit`. Napíšeme metodu `unit`, která přijímá jako argument pouze `self` a vrací nový vektor téže velikosti jako `self`, s každým prvkem děleným velikostí `self`, jež můžeme získat pomocí funkce `abs`. To by měl být velmi jednoduchý jedořádkový zápis.

Testy:

```
vector(0, 1).unit() |> print # vector(*pts=(0.0, 1.0))
vector(5, 0).unit() |> print # vector(*pts=(1.0, 0.0))
```

Zde je mé řešení:

```
    def unit(self) = self / abs(self)
```

### 2.7.3 `.angle`

Tato metoda bude poněkud složitější. Připomeňme, že matematicky se úhel mezi dvěma vektory vyjádří jako `math.acos` skalárního součinu obou vektorů, připadně jejich jednotkových vektorů a připomeňme si, že jsme již zavedli skalární součin dvou vektorů, když jsme napsali metodu `__mul__`. Takže, metoda `.angle` má přijmout `self` jako první argument a `other` jako druhý - a je-li `other` vektorem, použít tuto formuli k výpočtu úhlu mezi `self` `other`, nebo není-li `other` vektorem, má metoda `.angle` ohlásit `MatchError`. Abychom to zajistili, budeme potřebovat rozložené přiřazení k ověření, že `other` je skutečně vektor.

Testy:

```
import math
vector(2, 0).angle(vector(3, 0)) |> print # 0.0
print(vector(1, 0).angle(vector(0, 2)), math.pi/2) # should be the same
vector(1, 2).angle(5) # MatchError
```

*Nápověda: Podívejte se zpět, jak jsme s použitím rozloženého přiřazení kontrolovali, zda argument pro `factorial` bylo celé číslo.*

Pohleďte na mé řešení:

```
    def angle(self, other is vector) = math.acos(self.unit() * other.unit())
```

A nyní je čas to dát všechno dohromady. Nezdráhejte se dosadit své vlastní verze posledně definovaných metod.

```
import math # necessary for math.acos in .angle

data vector(*pts):
    """Immutable n-vector."""
    def __new__(cls, *pts):
        """Create a new vector from the given pts."""
        match [v is vector] in pts:
            return v  # vector(v) where v is a vector should return v
        else:
            return pts |*> makedata$(cls)  # accesses base constructor
    def __abs__(self) =
        """Return the magnitude of the vector."""
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)
    def __add__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Add two vectors together."""
        map((+), self.pts, other_pts) |*> vector
    def __sub__(self, vector(*other_pts)
                if len(other_pts) == len(self.pts)) =
        """Subtract one vector from another."""
        map((-), self.pts, other_pts) |*> vector
    def __neg__(self) =
        """Retrieve the negative of the vector."""
        self.pts |> map$(-) |*> vector
    def __mul__(self, other):
        """Scalar multiplication and dot product."""
        match vector(*other_pts) in other:
            assert len(other_pts) == len(self.pts)
            return map((*), self.pts, other_pts) |> sum  # dot product
```

```
        else:
            return self.pts |> map$((*)$(other)) |*> vector  # scalar multiplication
    def __rmul__(self, other) =
        """Necessary to make scalar multiplication commutative."""
        self * other
     # New one-line functions necessary for finding the angle between vectors:
    def __truediv__(self, other) = self.pts |> map$(x -> x/other) |*> vector
    def unit(self) = self / abs(self)
    def angle(self, other is vector) = math.acos(self.unit() * other.unit())

# Test cases:
vector(3, 4) / 1 |> print  # vector(*pts=(3.0, 4.0))
vector(2, 4) / 2 |> print  # vector(*pts=(1.0, 2.0))
vector(0, 1).unit() |> print  # vector(*pts=(0.0, 1.0))
vector(5, 0).unit() |> print  # vector(*pts=(1.0, 0.0))
vector(2, 0).angle(vector(3, 0)) |> print  # 0.0
print(vector(1, 0).angle(vector(0, 2)), math.pi/2)  # should be the same
vector(1, 2).angle(5)  # MatchError
```

*Jedna důležitá poznámka: dejte si pozor abyste nenechali prázdný řádek při dosazování vlastních metod, neboť v tom případě by interpret roztrhl kód. V normálním zápisu Coconut to není žádný problém, pouze zde, protože provádíme kopírování-vkládání do příkazového řádku*

Copy, paste! Jestliže všechno chodí jak má, doporučuji se vrátit ke hrátkám s *aplikacemi* `vector_field` s použitím našich nových metod.

## 2.8 Vyplnění mezer

Tímto vyčerpal tento tutoriál své případové studie, avšak to neznamená, že Coconut předvedl všechny své možnosti! V tomto posledním odstavci se dotkneme tří nejdůležitějších struktur, jež se nám podařilo opominout v případových studiích: líné seznamy, skladba funkcí a implicitní parciály (partials).

### 2.8.1 Líné seznamy

Líné seznamy jsou líně vyhodnocované iterátorové literály, podobné ve své lenosti operátoru `::` - a to v tom, že jakýkoli výraz uvnitř líného seznamu není vyhodnocen, dokud jej není zapotřebí. Syntaxe pro líné seznamy je přesně táž jako syntaxe pro normální seznamy, až na "banánové závorky"(`(|` and `|)`) místo normálních závorek, takto:

```
abc = (| a, b, c |)
```

Jako u všech iterátorů Pythonu, lze volat `next` k získání následného objektu v iterátoru. S použitím líného seznamu je možné definovat hodnoty, použité ve výrazech dle potřeby bez vyvolání hlášení `NameError`:

```
abcd = (| d(a), d(b), d(c) |)  # a, b, c, and d are not defined yet
def d(n) = n + 1

a = 1
next(abcd)  # 2
b = 2
next(abcd)  # 3
c = 3
next(abcd)  # 4
```

### 2.8.2 Skladba funkcí

Skladba funkcí v Coconut se zajišťuje operátorem `..`, který přijímá dvě funkce a spojí je do nové funkce, ekvivalentní zápisu `(*args, **kwargs) -> f1(f2(*args, **kwargs))`. To může být užitečné u částečné aplikace při spojování několika funkcí vyššího řádu, jako zde:

```
zipsum = map$(sum)..zip
```

Jsou-li skládané funkce uzavřeny v závorkách, lze do nich vložit argumenty:

```
def plus1(x) = x + 1
def square(x) = x * x

(plus1..square)(3) == 10  # True
```

Funkce s různými aritami lze skládat dohromady, pokud jsou uvedeny ve správném pořadí. Při nesprávném pořadí je vyvoláno hlášení `TypeError`. V tomto příkladu spojíme unární funkci s binární:

```
def add(n, m) = n + m  # binary function
def square(n) = n * n  # unary function

(add..square)(3, 1)    # Raises TypeError: square() takes exactly 1 argument (2 given)
(square..add)(3, 1)    # 16
```

Jiný důležitý trik zahrnuje skládání funkce s funkcí vyššího řádu:

```
def inc_or_dec(t):
    # Our higher-order function, which returns another function
    if t:
        return x -> x+1
    else:
        return x -> x-1

def square(n) = n * n

square_inc = square..inc_or_dec(True)
square_dec = square..inc_or_dec(False)
square_inc(4)  # 25
square_dec(4)  # 9
```

*Note: Coconut také podporuje skladebné pojítkové (pipe) operátory `..>`, `<..`, `..*>` a `<*..`.*

### 2.8.3 Implicitní parciály

Coconut podporuje řadu různých "neúplných"výrazů, jež se rozvinou do funkce, která přijme jen část argumentů, nezbytných pro dokončení, to jest do funkce s implicitně částečnou aplikací. Různé přípustné výrazy jsou:

```
.attr
.method(args)
obj.
func$
seq[]
iter$[]
.[slice]
.$[slice]
```

Úplné vysvětlení co každý implicitní parciál dělá lze nalézt v části implicit partials.

### 2.8.4 Anotace typů

Pro mnoho lidí je velkou nevýhodou Pythonu skutečnost, že je to dynamicky typovaný jazyk. V Pythonu je tento problém osloven v MyPy, což je analyzátor statických typů v Pythonu, který umí kontrolovat anotace typu, zavedené v Python 3, například:

```
def plus1(x: int) -> int:
    return x + 1
a: int = plus1(10)
```

Bohužel, tyto anotace typu existují pouze v Python 3. Coconut ovšem kompiluje tyto anotace na univerzálně kompatibilní komentáře typu. Kromě toho má Coconut vestavěnou MyPy integraci pro automatické ověřování typu v kódu a vlastní vylepšenou skladbu pro anotaci typu pro snadnější vyjádření složitých typů.

### 2.8.5 Další čtení

Všechny vlastnosti popsané v tomto tutoriálu, stejně jako řada dalších, jsou podrobně dokumentovány v podrobné dokumentaci.

Also, if you have any other questions not covered in this tutorial, feel free to ask around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.

Finally, Coconut is a new, growing language, and if you'd like to get involved in the development of Coconut, all the code is available completely open-source on Coconut's GitHub. Contributing is a simple as forking the code, making your changes, and proposing a pull request. See Coconuts contributing guidelines for more information.

KAPITOLA 3

# Dokumentace

## 3.1 Úvod

Tato dokumentace pokrývá všechny technické detaily programovacího jazyka Coconut a je zamýšlena spíš jako referenční příručka než podrobný úvod. Úplný úvod a tutoriál pro Coconut - viz Tutoriál.

Coconut je varianta jazyka Python, vytvořená pro **jednoduché, elegantní a funkcionální programování v Pythonu**. Skladba Coconut je podmnožna skladby Pythonu 3. To znamená, že uživatel, obeznámený s Pythonem, bude již obeznámený s většinou obsahu Coconut.

Kompilátor jazyka Coconut převádí kód Coconut na kód Pythonu. Primární způsob přístupu ke kompilátoru Coconut je z příkazového řádku utility REPL, jež rovněž obsahuje překladač pro kompilaci v reálném čase. Kromě této konzoly podporuje Coconut také notebooky IPythonu a Jupiteru.

Zatímco většina kódu v Coconut vychází ze snahy umožnit a zjednodušit funkcionální programování v Pythonu, další inspirace pochází z Haskellu, CoffeeScriptu, F# a z extenze Pythonu patterns.py.

## 3.2 Kompilace bez instalace

Chcete-li vyzkoušet Coconut ve svém webovém prohlížeči, můžete použít online interpreter.

### 3.2.1 Instalace

#### Použití Pip

Protože je Coconut hostován v Python Package Index, lze jej snadno instalovat s použitím `pip`. Jednoduše nainstalujte Python, otevřte příkazový řádek (cmd) a zadejte

```
pip install coconut
```

což nainstaluje Coconut a jeho požadované závislosti.

*Note: Máte-li nainstalovanou starou verzi Coconut a chcete ji aktualizovat, zadejte `pip install --upgrade coconut`.*

Když při spuštění `pip install coconut` narazíte na chybu, spusťte příkaz znovu s volbou `--user`. Když `pip install coconut` chodí ale nemáte přístup k příkazu `coconut`, ujistěte se, že umístění vaší instalace Coconut je uvedeno v proměnné prostředí `PATH`. V UNIXu to je `/usr/local/bin` (bez `--user`) nebo `${HOME}/.local/bin/` (s `--user`).

#### Použití Conda

Preferujete-li pro správu vašich paketů pro Python použití `conda` místo systému `pip` , můžete instalovat Coconut s použitím nástroje `conda`. Pouze install `conda`, otevřte terminál a zadejte

```
conda config --add channels conda-forge
conda install coconut
```

což řádně vytvoří a sestaví `conda recipe` z [`conda-forge` feedstock] (https://github.com/conda-forge/coconut-feedstock).

*Note: Pro použití `conda` k instalaci alternativního `coconut-develop`,nahraďte slovo `coconut` souslovím `coconut-develop` v posledních třech příkazech nahoře.*

### 3.2.2 Volitelné závislosti

Coconut má také volitelné dependence, instalovatelné zadáním

```
pip install coconut [název volitelné závislosti]
```

nebo pro instalaci více dependencí,

```
pip install coconut [opt_dep_1, opt_dep_2]
```

Úplný seznam volitelných depencencí:

- `all`: alias pro `jupyter, watch, jobs, mypy` (doporučný způsobinstalace úplné verze Coconut)
- `jupyter/ipython`: umožňuje použití flagu `--jupyter` / `--ipython`
- `watch`: umožňuje flag `--watch`

- `jobs`: umožňuje flag `--jobs`
- `mypy`: umožňuje flag `mypy`
- `cPyparsing`: výrazně urychluje kompilaci (pokud to vaše platformapodporuje) použitím [`cPyparsing`](https://github.com/evhub/ cpyparsing)
- `tests`: všechno nezbytné pro používání sestovací soupravy Coconutu
- `docs`: všechno nezbytné pro vytváření dokumentace Coconutu
- `dev`: všechno nezbytné pro vyvíjení Coconutu, včetně všech výše uvedených dependencí

### 3.2.3 Vývojářská verze

Případně, chcete-li si vyzkoušet poslední a nejlepší Coconut, zapište

```
pip install coconut-develop
```

což nainstaluje nejposlednější chodící verzi Coconutu z `develop` branch. Volitelná instalace závislostí je podporována stejným způsobem, jak popsáno výše. Více informací o aktuální vývojové sestavě najdete na development version of this documentation. Buďte varováni: `coconut-develop` může být nestabilní — narazíte-li na chybu, prosím ohlašte ji vytvořením nového issue.

## 3.3 Kompilace

### 3.3.1 Použití

```
coconut [-h] [-v] [-t version] [-i] [-l] [-k] [-p] [-a] [-w] [-r] [-n]
        [-d] [-q] [-s] [-f] [-c code] [-j processes] [--no-tco]
        [--minify] [--jupyter ...] [--mypy ...] [--argv ...]
        [--tutorial] [--documentation] [--style name]
        [--recursion-limit limit] [--verbose] [--trace]
        [source] [dest]
```

#### Poziční argumenty

```
source          cesta k souboru coconut, který má být kompilován
dest            cílová složka pro compilované soubory (implicitně jí je zdrojový␣
↪adresář)
```

#### Volitelné argumenty

```
-h, --help              show this help message and exit
-v, --version           print Coconut and Python version information
-t version, --target version
                        specify target Python version (defaults to
                        universal)
-i, --interact          force the interpreter to start (otherwise starts if no
                        other command is given) (implies --run)
-p, --package           compile source as part of a package (defaults to only
                        if source is a directory)
```

```
-a, --standalone        compile source as standalone files (defaults to only
                        if source is a single file)
-l, --line-numbers      add line number comments for ease of debugging
-k, --keep-lines        include source code in comments for ease of debugging
-w, --watch             watch a directory and recompile on changes
-r, --run               run compiled Python (often used with --nowrite)
-n, --no-write          disable writing compiled Python
-d, --display           print compiled Python
-q, --quiet             suppress all informational output (combine with
                        --display to write runnable code to stdout)
-s, --strict            enforce code cleanliness standards
--no-tco, --notco       disable tail call optimization
-c code, --code code    run Coconut passed in as a string (can also be piped
                        into stdin)
-j, --jobs processes    number of additional processes to use (defaults to
                        0) (pass 'sys' to use machine default)
-f, --force             force overwriting of compiled Python (otherwise only
                        overwrites when source code or compilation
                        parameters change)
--minify                compress compiled Python
--jupyter, --ipython    run Jupyter/IPython with Coconut as the kernel (
                        remaining args passed to Jupyter)
--mypy ...              run MyPy on compiled Python (remaining args passe to
                        MyPy) (implies --package--no-tco)
--argv ...              set sys.argv to source plus remaining args for use
                        in Coconut script being run
--tutorial              open the Coconut tutorial in the default web browser

--style name            Pygments syntax highlighting style (or 'none' to
                        disable) (defaults to COCONUT_STYLE environment
                        variable, if it exists, otherwise 'default')
--recursion-limit limit
                        set maximum recursion depth in compiler (defaults to
                        2000)
--verbose               print verbose debug output
--trace                 print verbose parsing data (only available in
                        coconut-develop)
```

### 3.3.2 Skripty Coconutu

```
Ke spuštění souboru Coconut jako skriptu poskytuje Coconut příkaz
```

coconut-run

```
jako alias pro
```

coconut –run –quiet –target sys –argv

```
který se potichu zkompiluje a spustí `<source>`, předávajíc skriptu jakýkoliv␣
↪dodatečný argument, napodobujíc tak práci příkazu pythonu.

`coconut-run` může být použit v řádku s shebangem Unixu pro vytvoření skriptu Coconut␣
↪přidáním následujícího řádku na začátek skriptu:
```bash
#!/usr/bin/env coconut-run
```
```

### 3.3.3 Názvy zdrojových souborů

Zdrojové soubory používají extenze `.coco` (upřednostněno), `.coc` nebo `.coconut`. Soubor `.coco` (či `.coc`/`.coconut`) je kompilován do souboru s příponou `.py`. Je-li požadována jiná extenze než `.py`, například `.pyde` pro Python Processing, může být vložena před `.coco` a tato složená extenze bude použita místo `.py`. Například, `name.coco` bude kompilovat na `name.py`, zatímco `name.pyde.coco` bude kompilovat na `name.pyde`.

### 3.3.4 Kompilační režimy

Soubory, kompilované konzolou `coconut` se mohou lišit v závislosti na kompilačních parametrech. Je-li kompilována celá složka souborů (ve které bude kompilátor rekurzivně vyhledávat soubory s extenzí `.coco`, `.coc` nebo `.coconut`), vytvoří se soubor `__coconut__.py`, pro ukládání nezbytných funkcí (package mode), zatímco při kompilaci jediného souboru se nezbytné informace ukládají v záhlaví uvnitř souboru (standalone mode). Standalone mode je lepší pro jednotlivé soubory, protože se obejde bez nadbytečného importování `__coconut__.py`, avšak package mode je lepší pro velké pakety, protože se nemusí v každém souboru spouštět kód v záhlaví, jelikož může být jednoduše importován z `__coconut__.py`.

Je-li `zdrojovým` argumentem pro CLI konzolu soubor, provede se implicitně samostatná kompilace, zatímco je-li jím složka, provede se rekurzivní vyhledávání všech souborů `.coco` (nebo `.coc` či `.coconut`), pro něž se provede paketová kompilace. Coconut takto ve většině provede správnou volbu režimů. Je-li však důležité aby se nevytvářely žádné dodatečné soubory jako např. `__coconut__.py`, potom lze přinutit CLI konzolu aby použila určený režim použitím flagů `--package` (`-p`) a `--standalone` (`-a`).

### 3.3.5 Kompatibilní verze Pythonu

Protože je skladba Coconut založena na Python3, měl by kód Coconut, kompilovaný kompilátorem Coconut v univerzálním režimu (implicitní `--target`) běžet v libovolné verzi Pythonu `>= 2.6` nebo `>= 3.2`.

*Poznámka: Vyzkoušené implementace jsou CPython `2.6`, `2.7`, `3.2`, `3.3`, `3.4`, `3.5`, `3.6` a PyPy `2.7`, `3.2`.*

Aby byly nativní objekty (built-ins) Coconut univerzálně přístupné pro různé verze Pythonu, přepisuje **Coconut automaticky built-iny Pythonu 2 na příslušné protějšky Pythonu3**. Navíc, Coconut také přepisuje některé built-iny Pythonu3 z optimalizačních důvodů. Je-li žádán přístup k původním verzím přepsaných built-inů, lze je získat s použitím prefixu `py_`.

Pro kompatibilitu se standardní knihovnou **mapuje Coconut automaticky importy pod názvy Python3 s importy pod názvy Python2**. Takto se Coconut automaticky postará o všechny moduly standardní knihovny, které byly přejmenovány z Python2 na Python3, pokud je použit pouze Python3. Ovšem, pro moduly nebo objekty, které existují pouze v Python3, neumí Coconut kompatibilitu zajistit.

Konečně, zatímco se Coconut pokusí kompilovat skladbu Python3 na jeho univerzální ekvivalent, následující konstrukty nemají žádný ekvivalent v Python2 a vyžadují specifikaci alespoň `3` před svým použitím:

- destructuring assignment with `*`s (use Coconut pattern-matching instead),
- the `nonlocal` keyword,
- `exec` used in a context where it must be a function,
- keyword class definition,
- tuples and lists with `*` unpacking or dicts with `**` unpacking (requires `--target 3.5`),
- `@` as matrix multiplication (requires `--target 3.5`),
- `async` and `await` statements (requires `--target 3.5`), and
- formatting `f` strings by prefixing them with `f`(requires `--target 3.6`).

### 3.3.6 Přípustné cíle

Je-li verze Pythonu, v níž bude kompilovaný kód běžet, známa předem, měl by být cíl určen flagem `--target`. Daný cíl (target) ovlivní pouze kompilovaný kód a zda je určitá syntaxe Pythonu3 (viz výše) povolena. Tam, kde se standardy skladeb pro Python3 a Python2 liší, bude skladba Coconut vždy používat skladbu Python3 pro všechny cíle. Podporované cíle jsou:

- universal (default) (will work on *any* of the below),
- `2`, `2.6` (will work on any Python `>= 2.6` but `< 3`),
- `2.7` (will work on any Python `>= 2.7` but `< 3`),
- `3`, `32` (will work on any Python `>= 3.2`),
- `3.3`, `34` (will work on any Python `>= 3.3`),
- `3.5` (will work on any Python `>= 3.5`),
- `3.6` (will work on any Python `>= 3.6`),
- `sys` (chooses the specific target corresponding to the current version).

*Note: Tečky jsou ve specifikacích cíle ignorovány, takže cíl `2.7` je ekvivalentní cíli `27`.*

### 3.3.7 Režim `strict`

Je-li povolen flag `--strict` (or `-s`), ohlásí Coconut chyby pro různé problémy stylu. Jsou jimi

- mixing of tabs and spaces (without `--strict` will show a Warning),
- use of `from __future__` imports (without `--strict` will show a Warning)
- missing new line at end of file ),
- trailing whitespace at end of lines,
- semicolons at end of lines,
- use of the Python-style `lambda` statement,
- inheriting from `object` in classes (Coconut does this automatically)
- use of `u` to denote Unicode strings (all Coconut strings aru Unicode strings)
- use of backslash continuations (use [parenthetical continuation](# pokracovani-v-zavorkach) instead).

Navíc, `--strict` zneplatňuje (disables) zavrhnuté vlastnosti, činíce je zcela nedostupnými při kompilaci s flagem `--strict`. Doporučuje se při práci na novém projektu používat flag `--strict` (nebo `-s`) protože vám bude nápomocen při psaní čistšího kódu.

## 3.4 Integrace

### 3.4.1 Zvýraznění syntaxe

Textové editory, které podporují zvýraznění syntaxe Coconut, jsou tyto:

- **SublimeText**: Viz sekci SublimeText níže.
- **Vim**: Viz `coconut.vim`.
- **Emacs**: Viz `coconut-mode`.

- **Atom**: Viz `language-coconut`.
- Každý editor, který podporuje Pygments (např. **Spyder**): Viz sekci Pygments níže.

Případně, pokud žádný z výše uvedený editorů vám nevyhovuje, můžete v Coconut pracovat jako v Pythonu. Jednoduše nastavte svůj editor tak, aby interpretoval všechny souboury `.coco` jako soubory Pythonu, čímž by mělo být zvýraznění vašeho kódu vyhovující.

#### SublimeText

Zvýraznění syntaxe Coconutu v editoru SublimeText vyžaduje, aby byl instalován standardní manažer Package Control. Je-li tak učiněno, potom:

1. otevřte příkazovou paletu SublimeTextu stiskem `Ctrl+Shift+P` (nebo `Cmd+Shift+P` v Mac)
2. zadejte `Package Controll: Install Package`
3. zadejte `Coconut`.

Abyste se ujistili, že je všechno OK, otevřte soubor `.coco` a ujistěte se, že se v pravém spodním rohu objeví `Coconut`. Objeví-li se něco jiného, třeba `Plain Text`, klikněte na to, vyberte `Open all with current extension as...` na vrchu výsledného menu a vyberte `Coconut`.

*Note: Zvýraznění syntaxe Coconutu je poskytnuto paketem sublime-coconut.*

#### Pygments

Tentýž příkaz `pip install coconut`, který instaluje utilitu příkazového řádku Coconutu, instaluje také lexer `coconut` Pygments. Způsob použití závisí na použité `Pygments-enabled` aplikaci ale normálně zadejte `coconut` jako zvýrazňovaný jazyk a/nebo použijte platnou extenzi souboru Coconut (`.coco`, `.coc` nebo `.coconut`) a Pygment by se měl umět zorientovat.

Na příklad, tato dokumentace je generována Sphinx se zvýrazněním syntaxe vytvořené přidáním řádku

```
highlight_language = "coconut"
```

do souboru `conf.py` v distribuci Coconut.

### 3.4.2 Podpora pro IPython Jupyter

Dáváte-li přednost prostředí IPython (jádro Pythonu pro framework Jupyter ) před normální konzolou Pythonu, lze použít Coconut jako extenzi IPythonu nebo jádro Jupyteru.

#### Jádro

Je-li Coconut použit jako jádro (kernel), bude veškerý kód v konzoli nebo notebooku poslán k vyhodnocení do Coconut místo do Pythonu. Jinak se jádro Coconut chová stejně jako jádro iPythonu, včetně podpory pro příkazy `%magic`.

Příkaz `coconut --jupyter notebook` (nebo `coconut --ipython notebook`) spustí notebook IPython/ Jupyter s použitím Coconut jako jádra a příkaz `coconut --jupyter console` (nebo `coconut --ipython console`) spustí konzoli IPython/ Jupyter s použitím Coconut jako jádra. Navíc, příkaz `coconut --jupyter` (nebo `coconut --ipython`) přidá Coconut jako jazykovou volbu uvnitř všech notebooků IPython/ Jupyter - i těch, které nejsou spouštěny aplikací Coconut. Tento příkaz musí být opakovaně proveden při instalaci nové verze Coconut.

_Note: Coconut také podporuje příkaz `coconut --jupyter lab` pro použití s JupyterLab místo standardního notebooku Jupyter.

#### Extenze

Je-li Coconut použit jako extenze, bude speciální "magic command"posílat útržky kódu k vyhodnocení s použitím Coconut místo IPythonu ale IPython bude stále použit jako implicitní aplikace.

Řádkový magic `%load_ext coconut` načte Coconut jako extenzi, připojujíc magics `%coconut` a `%%coconut`. Řádkový magic `%coconut` spustí řádek Coconut s implicitními parametry a blokový magic `%%coconut` přijme CL (command line) argumenty z prvního řádku a vyhodnotí kód Coconut pro dané parametry ve zbytku buňky.

### 3.4.3 Integrace s MyPy

Coconut se umí integrovat s MyPy za účelem optimální statické kontroly typů, včetně všech vestavěných nástrojů Coconut. Jednoduše zadejte `--mypy` abyste umožnili integraci s MyPy, ale dejte si pozor abyste to zadali jako poslední argument, protože všechny argumenty po `--mypy` jsou poslány do `mypy`, nikoliv do Coconut.

_Note: Protože *optimalizace koncového volání* vylučuje řádnou typovou kontrolu, `--mypy` ji implicitně vypíná.

Pro explicitní typovou kontrolu kódu v MyPy podporuje Coconut anotace typu funkcí v Python 3, anotace typu proměnných v Python 3.6 a dokonce vlastní *vylepšenou skladbu* anotace typů. Implicintě jsou všechny anotace typu kompilovány na signaturu typu, kompatibilní s Python 2, což znamená že všechny anotace chodí ve všech verzích Pythonu.

Coconut dokonce podporuje `--mypy` v interpretu, který inteligentně skenuje každý nový řádek kódu, číhaje na nově zavedené chyby MyPy. Na příklad:

Coconut dokonce podporuje `--mypy` v překladači, jenž skenuje inteligentně každý nový řádek kódu v kontextu s předchozím řádkem zda neobjeví nově zavedené chyby MyPy. Na příklad:

```
>>> a: str = count()[0]
<string>:14: error: Incompatible types in assignment (expression has type "int",␣
↪variable has type "str")
```

:Note: Někdy si MyPy nebude vědět rady s jistými konstrukty Coconut, např. s `adaptern`. V tom případě jednoduše zadejte `# type: ignore` na řádek Coconut, na jehož kompilaci si MyPy stěžuje (o který řádek se jedná, zjistíte použitím flagu `--line-numbers`).

## 3.5 Operátory

Toto jsou operátory Coconut, uvedené v pořadí podle precedencí (nejvyšší nahoře):

```
==================== ============================
Symbol                Asociativita
==================== ============================
..                    n/a wont't capture call
**                    right
+, -, ~               unary
*, /, //, %, @        left
+, -                  left
<<, >>                left
&                     left
^                     left
|                     left
::                    n/a lazy
a `b` c               left captures lambda
??                    left short-circuit
..>, <.., ..*>, <*..  n/a captures lambda
```

```
|>, <|, |*>, <*|       left captures lambda
==, !=, <, >,
    <=, >=,
    in, not in,
    is, is not        n/a
not                   unary
and                   left short-circuit
or                    left short-circuit
a if b else c         ternary left short-circuit
->                    right
===================== ========================
```

### 3.5.1 Lambdy

Coconut poskytuje jednoduchý, čistý operátor `->` jako alternativu k příkazu `lambda` v Pythonu. Skladba s operátorem `->` je `(parameters) -> expression` (nebo `parameter -> expression` pro lambdy s jedním argumentem). Operátor má stejné pořadí důležitosti jako starý příkaz, což znamená, že bude často nezbytné uzavřít lambdu do závorek a je asociativní vpravo.

Navíc, Coconut také podporuje implicitní použití operátoru `->` ve formě `(-> expression)`, jež je ekvivalentní k `((_=None) -> expression)`, což umožňuje použití implicitní lambdy když nejsou vyžadovány žádné argumenty nebo když je vyžadován jen jeden argument (vyjádřený znakem _).

*Note: Je-li normální skladba lambdy nedostatečná, Coconut také podporuje rozšířenou skladbu lambdy ve formě* příkazu lambda.

#### Zdůvodnění

Použití funkce lambda je v Pythonu neúhledné a neohrabané, vyžadující vypsání celého slova `lambda` pokaždé, když je vytvářena. To je dobré tehdy, jsou-li in-line funkce používány zřídka ale ve funkcionálním programování jsou in-line funkce základním nástrojem.

#### Python Docs

Formy lambda mají totéž skladebné postavení jako obecné výrazy. Jsou zkratkou při vytváření anonymních funkcí; výraz `(arguments) -> expression` vytváří objekt funkce. Nepojmenovaný objekt se chová jako objekt funkce, definovaný:

```
def <lambda>(arguments):
    return expression
```

Všimněte si, že funkce vytvořené formou lambda nemohou obsahovat příkazy nebo anotace.

#### Příklad

**Coconut**

```
dubsums = map((x, y) -> 2*(x+y), range(0, 10), range(10, 20))
dubsums |> list |> print
```

**Python**

```
dubsums = map(lambda x, y: 2*(x+y), range(0, 10), range(10, 20))
print(list(dubsums))
```

### 3.5.2 Částečná aplikace

Coconut používá znak `$` hned za názvem funkce, však před závorkou, použitou k volání funkce.

Částečná aplikace Coconutu také podporuje použití `?` a by se přeskočilo částečné použití argumentu, odkládajíc použití tohoto argumentu až na volání částečtě aplikované funkce. To je důležité, chcete-li částečně aplikovat argumenty, které nejsou první v pořadí argumentů.

#### Zdůvodnění

Částečná aplikace neboli currying je ústřední pilíř funkcionálního programování a to z dobrého důvodu: umožňuje dynamickou úpravu funkce pro potřebu v místě použití. Částečná aplikace umožňuje vytvoření nové funkce ze staré pro specifikované některé argumenty.

#### Python Docs

Má se vrátit nový objekt `partial`, který se při volání bude chovat jako *func* volaná s pozičními argumenty *args* a keyword-argumenty *keywords*. Jsou-li další argumenty zadány při volání, jsou připojeny k *args*. Jsou-li další keyword-argumenty zadány, rozšiřují a přepisují *keywords*. Zhruba ekvivalentní k:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*(args + fargs), **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

Objekt `partial` je použit pro částečnou (partial) aplikaci funkce, která "zmrazí" (freezes) některé argumenty a/nebo keywords funkce, vytvářejíc tak nový objket se zjednodušenou signaturou.

#### Příklad

**Coconut**

```
expnums = range(5) |> map$(pow$(?, 2))
expnums |> list |> print
```

**Python**

```
# unlike this simple lambda, $ produces a pickleable object
expnums = map(lambda x: pow(x, 2), range(5))
print(list(expnums))
```

### 3.5.3 Spojovník (pipeline)

Coconut používá spojovníky (|) pro usměrnění průběhu aplikace funkcí. Všechny operátory mají precedenci infixových evokací a jsou levostranně asociativní. Všechny operátory také podporují 'in-place versions'. Těmito operátory jsou:

```
(|>)     => pipe forward
(|*>)    => multiple-argument pipe forward
(<|)     => pipe backward
(<*|)    => multiple-argument pipe backward
```

Navíc, všechny spojovníkové operátory podporují lambdu jako poslední argument, přesto že má lambda nižší precedenci. Takže, `a |> x -> b |> c` je ekvivalentní s `a |> (x -> b |> c)`, nikoliv s `a |> (x -> b) |> c`.

*Note: Pro vizuální rozložení operací přes několik řádek použijte* parenthetical continuation.

#### Optimalizace

V Coconut je obvyklé psát kód, který používá spojovníky pro zadávaní objektu řadou *partials* a/nebo *implicit partials*, jako v

```
obj |> .attribute |> .method(args) |> func$(args) |> .[index]
```

což je často mnohem čitelnější, protože to umožňuje aby byly operace psány v pořadí, v němž jsou vykonávány, místo jako v

```
func(args, obj.attribute.method(args))[index]
```

kde musí `func` přijít jako první.

Kdyby Coconut kompiloval každou část ve spojovníkové syntaxi jako skutečný objekt částečné aplikace, stala by se skladba ve stylu Coconut do té míry pomalejší než skladba ve stylu Python, že by byla téměř nepoužitelná. Coconut tento problém obchází tím, že `partials` i `implicit partials` jsou kompilovány na skladbu ve stylu Python, nevytvářejíc tak žádné mezilehlé objekty.

#### Příklad

**Coconut**

```
def sq(x) = x**2
(1, 2) |*> (+) |> sq |> print
```

**Python**

```
import operator
def sq(x): return x**2
print(sq(operator.add(1, 2)))
```

### 3.5.4 Skladba funkcí

Coconut má tři základní operátory pro skladbu funkcí: `..`, `..>` a `<...` Jak `..`, tak `<..` používají "zpětnou"skladbu funkcí, kdy je první funkce volána jako poslední, zatímco `..>` používá "dopřednou"skladbu funkcí, kde je první funkce volaná jako první.

Spojovníkové operátory `..>` a `<..` mají také formu `..*>` a `<*..`, která je ekvivalentní k operátorům `|*>` a `<*|`. Zpětné a dopředné spojovníkové operátory nemohou být použitý společně v jednom výrazu (na rozdíl od normálních spojovníků) a jeich precedence je mezi spojovníkem `None` a normálním spojovníkem.

Operátor `..` má nižší precedenci než přístup k atributu (`.`), slicing (`[]`), atd, kromě volání funkce, vůči níž má precedenci vyšší. Takže je `a.b..c.d` ekvivalentní k `(a.b)..(c.d)`, zatímco `f..g(x)` je ekvivalentní k `(f.g)(x)`.

'In-place' operátory pro skladbu funkcí jsou `..=`, `..>=`, `<..=`, `..*>=` a `<*..=`.

#### Příklad

**Coconut**

```
fog = f..g
f_into_g = f ..> g
```

**Python**

```
# unlike this simple lambda, Coconut produces a pickleable object
fog = lambda *args, **kwargs: f(g(*args, **kwargs))
f_into_g = lambda *args, **kwargs: g(f(*args, **kwargs))
```

### 3.5.5 Řetězení

Coconut používá operátor `::` pro řetězení iterátoru. Toto řetězení je prováděno líně - to jest tak, že argumenty se nevyhodnocují, pokud jich není zapotřebí. Tato forma má precedenci 'in-between bitwise or and infix calls'. 'In-place' operátorem je `::=`.

#### Zdůvodnění

Důležitým nástrojem pro práci s iterátory stejně snadno jako při práci se sekvencemi je schopnost líně kombinovat více iterátorů dohromady. Tato operace se nazývá řetěz (chain) a je ekvivalentní přidávání u sekvencí s tím rozdílem, že se nic nevyhodnocuje, pokud to není zapotřebí.

#### Python Docs

Vytvořte iterátor, který vrací prvky z prvního iteráblu (iterovatelného objektu) dokud je nevyčerpá, potom přejde do dalšího iteráblu až projde všemi iterábly. Používá se pro ošetření následných sekvencí jako jediné sekvence. Zřetězené vstupy jsou vyhodnocovány líně. Zhruba ekvivalentní k:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

#### Příklad

**Coconut**

```
def N(n=0) = (n,) :: N(n+1) # no infinite loop because :: is lazy

(range(-10, 0) :: N())$[5:15] |> list |> print
```

**Python:**

*Nelze provést bez komplikované komprehence iterátoru namísto líného řetězení. Viz kompilovaný kód pro skladbu Pythonu.*

### 3.5.6 Krájení (slicing) iterátoru

K provedení iterátorového členění používá Coconut znak `$` mezi iterátorem a označením jeho úseku. Iterátorové členění pracuje stejně jako sekvenční členění v Pythonu a vypadá stejně jako částečná aplikace, avšak s hranatymi místo kulatých závorek.

Iterátorové členění pracuje stejně jako sekvenční členění, včetně podpory negativních indexů a úseků (slices) a podpory pro objekty `úseků` stejně jako u normálního členění. Iterátorové členění však nezaručuje, že bude zachován původní iterátor (pro jeho zachování použijte *funkcitee* nebo *reiterable*).

Iterátorové členění v Coconut je velmi podobné `itertools.islice` v Pythonu, avšak na rozdíl od `itertools.islice`, podporuje iterátorové členění negativní index a přednostně použije `__getitem__` objektu, pokud existuje. Iterátorové členění je také optimalizované pro práci s objekty `map`, `zip`, `range` a `count`, počítaje pouze ty prvky, které jsou nezbytné pro vynětí žádaného úseku.

#### Příklad

**Coconut**

```
map((x)->x*2, range(10**100))$[-1] |> print
```

**Python** *Nelze provést bez komplikované funkce pro iterátorové členění a inspekce uživatelských objektů. Nezbytné definice v Pythonu lze nalézt v záhlaví Coconut.*

### 3.5.7 None Coalescing

Coconut poskytuje označení `??` pro operátor `none-coalescing` (none-sloučení), podobný operátoru null-coalescing `??` v C#. Operátor `none-coalescing` vyhodnocuje svůj levý operand, nemá-li hodnotu `None`, v opačném případě svůj pravý operand. Tento operátor také vyjadřuje zkratku, spočívající v tom, že když jeho levý operand není `None`, nevyhodnocuje pravý operand. `None-coalescing` má precedenci mezi voláním infixové funkce a spojovníkem a je asociativní vlevo. Operátor `in-place` má označení `??=`.

Coconut také umožňuje použití jediného znaku `?` před přístupem k atributu, při volání funkce, částečné aplikaci a při (iterátorovém) indexování pro (zkratkovité) upuštění od vyhodnocení zbytku, pokud dosavadní vyhodnocení má hodnotu `None`. Tudíž, `a?.b` je ekvivalentní k `a.b if a is not None else a`.

#### Příklad

**Coconut:**

```
could_be_none() ?? calculate_default_value()
could_be_none()?.attr[index].method()
```

**Python**

```
(lambda result: result if result is not None else
 calculate_default_value())(could_be_none())
(lambda result: None if result is None else
 result.attr[index].method())(could_be_none())
```

### 3.5.8 Alternativy Unicode

Coconut podporuje alternativy Unicodu pro různé operátové symboly. Alternativy jsou poměrně nápovědné, se záměrem reflektovat vzhled nebo účel originálního symbolu.

#### Úplný seznam

```
→ (\u2192)                    => "->"
 (\u21a6)                    => "|>"
⋆ (⋆\u21a6)                  => "|⋆>"
 (\u21a4)                    => "<|"
⋆ (\u21a4⋆)                  => "<⋆|"
× (\xd7)                     => "*"
↑ (\u2191)                   => "**"
÷ (\xf7)                     => "/"
÷/ (\xf7/)                   => "//"
 (\u2218)                    => ".."
> (\u2218>)                  => "..>"
< (<\u2218)                  => "<.."
⋆> (\u2218⋆>)                => "..⋆>"
<⋆ (<⋆\u2218)                => "<⋆.."
 (\u2212)                    => "-" (only subtraction)
 (\u207b)                    => "-" (only negation)
¬ (\xac)                     => "~"
 (\u2260) or ¬= (\xac=)      => "!="
 (\u2264)                    => "<="
 (\u2265)                    => ">="
 (\u2227) or  (\u2229)       => "&"
 (\u2228) or  (\u222a)       => "|"
 (\u22bb) or  (\u2295)       => "^"
« (\xab)                     => "<<"
» (\xbb)                     => ">>"
... (\u2026)                 => "..."
 (\u22c5)                    => "@" (only matrix multiplication)
```

## 3.6 Klíčová slova

### 3.6.1 `data`

Klíčové slovo `data` se používá k vytvoření neměnitelných algebraických datových typů s nativní podporou pro rozkladný (destructuring) *pattern-matching* a *`fmap`*.

Syntaxe datového bloku `data` je něco mezi syntaxí pro funkce a syntaxí pro třídy. První řádek vypadá jako definice funkce, zatímco zbytek těla připomíná třídu, obvykle obsahující definice metod. Je to tak proto, že zatímco blok `data` vlastně v Pythonu končí jako třída, Coconut automatický vytváří specielní, neměnitelný konstruktor, založený na daných argumentech.

Deklarace datového typu vypadá takto:

```
data <name>(<args>) [from <inherits>]:
    <body>
```

`<name>` je název nového datového typu, `<args>` jsou argumenty jeho konstruktoru stejně jako názvy jeho atributů, `<body>` obsahuje metody datového typu a nepovinně obsahuje libovolnou bázovou třídu.

Coconut připouští aby datová pole v `<args>` měla přiřazené implicitní hodnoty a *anotace typu* a podporuje hvězdičkové parametry na konci, pro posbírání extra argumentů.

Konstruktory pro `datové` typy musí být vytvářeny s použitím metody `__new__` místo `__init__`. Pro snadnější psaní metod `__new__` poskytuje Coconut vestavěnou funkci *makedata*.

Subtřídy datových typů lze snadno vytvořit jejich děděním v jiné deklaraci `datového` typu nebo v normální `třídě` Pythonu. Použije-li se normální příkaz `class`, vytvoření nové neměnitelné subtřídy vyžaduje přidání řádku

```
__slots__ = ()
```

do těla subtřídy před definicemi metod nebo atributů.

### Zdůvodnění

Hlavní část funkcionálního programování, které Coconut v Pythonu zlepšuje, je použití hodnot nebo neměnitelných datových typů. Neměnitelná data jsou velmi užitečná ale vytvoření takových typů v Pythonu je velice obtížné. Coconut vytváří neměnitelné datové typy velice snadno použitím bloků typu `data`.

### Python Docs

Vrací novou subtřídu entice (tuple). Nová subtřída je použita k vytvoření entici podobných objektů, jejichž pole jsou přístupná přes vzhled (lookup) atributu a jsou indexovatelná a iterovatelná. Instance subtřídy mají také nápomocný docstring (se jménem typu a pole) a metodu `__repr__()`, která uvádí obsah entice ve formátu `name=value`.

Pro název pole lze použít libovolný platný identifikátor Pythonu. Platné identifikátorý se skládají z písmen, číslic a podtržítek ale nezačínají číslicí nebo podtržítkem a nejsou klíčovým slovem jako *class, for, return, global, pass nebo raise*.

Pojmenované instance entic nemají individuální slovníky (dictionaries), takže jsou úsporné a nevyžadujíc více paměti než normální entice.

### Příklady

**Coconut**

```
data vector2(x:int=0, y:int=0):
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

v = vector2(3, 4)
v |> print  # all data types come with a built-in __repr__
v |> abs |> print
v.x = 2  # this will fail because data objects are immutable
vector2() |> print
```

*Demonstruje skladbu, vlastnosti a neměnitelnou povahu typů `data`, stejně jako použití implicitních argumentů a anotací typů.*

```
data Empty()
data Leaf(n)
data Node(l, r)

def size(Empty()) = 0

@addpattern(size)
def size(Leaf(n)) = 1

@addpattern(size)
def size(Node(l, r)) = size(l) + size(r)

size(Node(Empty(), Leaf(10))) == 1
```

*Demonstruje algebraickou povahu typů* `data` *při kombinaci s pattern-matching.*

```
data vector(*pts):
    """Immutable arbitrary-length vector."""

    def __abs__(self) =
        self.pts |> map$(pow$(?, 2)) |> sum |> pow$(?, 0.5)

    def __add__(self, other) =
        vector(*other_pts) = other
        assert len(other_pts) == len(self.pts)
        map((+), self.pts, other_pts) |*> vector

    def __neg__(self) =
        self.pts |> map$((-)) |*> vector

    def __sub__(self, other) =
        self + -other
```

*Demonstruje* `hvězdičkovou` *deklaraci typu* `data`.

**Python** *Nelze provést bez definování řady metod pro každý datový typ. Viz kompilovaný kód pro syntaxi Pythonu.*

### 3.6.2 `match`

Coconut poskytuje plnohodotné, funkcionální `pattern-matching` prostřednictvím svých příkazů `match`.

#### Přehled

Příkazy `match` konvenují se základní skladbou `match <pattern> in <value>`. Příkaz match se pokusí porovnat hodnotu se vzorkem a v případě shody sváže proměnnou ve vzorku s odpovídající pozicí v hodnotě a provede následný kód za příkazem match. Příkazy match také ve své základní skladbě podporují podmínku `if <cond>`, která se vyhodnotí po nalezení shody před provedením následného kódu a příkaz `else`, který se provede, pokud ke shodě nedojde. Všechny možnosti příkazu match nemají ekvivalent v Pythonu a proto následuje vysvětlení jednotlivých specifikací.

#### Specifikace skladby

Skladba příkazu match v Coconut je

```
match <pattern> in <value> [if <cond>]:
    <body>
[else:
    <body>]
```

kde <value> je položka, v níž se hledá shoda, <cond> je volitelná dodatečná podmínka a <body> je kód, který se provede při splnění výše uvedeného záhlaví. Vstup <pattern> má svoji vlastní specifickou skladbu, definovanou zhruba takto:

```
pattern ::= (
    "(" pattern ")"                 # závorky
    | "None" | "True" | "False"     # konstanty
    | "=" NAME                      # ověření (checks)
    | NUMBER                        # čísla
    | STRING                        # řetězce
    | [pattern "as"] NAME           # capture
    | NAME "(" patterns ")"         # datové typy
    | pattern "is" exprs            # type-checking
    | pattern "and" pattern         # match all
    | pattern "or" pattern          # match any
    | "{" pattern_pairs             # dictionaries
      ["," "**" NAME] "}"
    | ["s"] "{" pattern_consts "}"  # sets
    | "(" patterns ")"              # sekvence mohou mít formu entice
    | "[" patterns "]"              #  nebo formu seznamu
    | "(|" patterns "|)"            # líné seznamy
    | "{" pattern_pairs "}"         # slovníky
    | ["s"] "{" pattern_consts "}"  # sety
    | ("(" | "[")                   # star splits
      patterns,
      "*" middle,
      patterns
    (")" | "]")
    | (                             # head-tail splits
      "(" patterns ")"
      | "[" patterns "]"
    ) "+" pattern
    | pattern "+" (                 # init-last splits
      "(" patterns ")"
      | "[" patterns "]"
    )
    | (                             # head-last splits
      "(" patterns ")"
      | "[" patterns "]"
    ) "+" pattern "+" (
      "(" patterns ")"                  # this match must be the same
      | "[" patterns "]"                #  construct as the first match
    )
    | (                             # iterator splits
      "(" patterns ")"
      | "[" patterns "]"
      | "(|" patterns "|)"
    ) "::" pattern
    | ([STRING "+"] NAME            # complex string matching
      ["+" STRING])
    )
```

### Specifikace významů

Příkaz `match` přijme vzorek a pokusí se k němu nalézt shodu v zadaných argumentech. Vzorek může obsahovat různé struktury:

- Konstanty, čísla a řetězce: se budou shodovat se stejnou konstantou, číslem či řetězcem na stejných pozicích argumentů.
- Proměnné: se budou shodovat a budou propojené s čímkoli - s několika výjimkami:
  - Je-li táž proměnná použita vícekrát, provede se kontrola, zda se každý výskyt shoduje se stejnou hodnotou.
  - Je-li názvem proměnné _, všechno se s ní bude shodovat ale nic nebude připojeno.
- Explicitní vazby (`<pattern> as <var>`): připojí `<var>` k `<pattern>`.
- Ověření (`=<var>`): ověří (checks), zda je kontrolovaná pozice rovna dříve definované proměnné `<var>`.
- Ověření typu (`<var> is <types>`): před připojením k proměnné `<var>` ověří, zda kontrolovaná pozice je typu `<types>`.
- Datové typy (`<name>(<args>)`): ověří, zda kontrolovaná pozice je typu `<name>` a spáruje atributy s `<args>`.
- Seznamy (`[<patterns>]`), entice (`(<patterns>)`) nebo líné seznamy (`(|<patterns>|)`): spáruje pouze sekvence (`collections.abc.Sequence`) stejné délky a porovná obsah vůči `<patterns>`.
- Líné seznamy (`(|<patterns>|)`): totéž jako při hledání shody (matching) u seznamů nebo entic, ale místo sekvencí kontroluje iterovatelné objekty - `iterably` (`collections.abc.Iterable`).
- Fixed-Length Dicts (`{<pairs>}`): porovná pouze `mapování` (`collections.abc.Mapping`) stejné délky a obsahy porovná s `<pairs>`.
- Dicts With Rest (`{<pairs>, **<rest>}`): porovná `mapování` (`collections.abc.Mapping`) obsahující všechny `<pairs>` a vloží `dict` všeho ostatního do `<rest>`.
- Sety (`{<constants>}`): spáruje pouze set (`collections.abc.Set`) se stejnou délkou a obsahem.
- Head-Tail Splits (`<list/tuple> + <var>`): porovná počátek sekvence vůči `<list/tuple>`, zbytek připojí k `<var>` a učiní jej typem použitého konstruktu.
- Init-Last Splits (`<var> + <list/tuple>`): přesně totéž jako head-tail splits ale vzhledem ke konci, nikoliv k počátku sekvence.
- Head-Last Splits (`<list/tuple> + <var> + <list/tuple>`): kombinace předchozích dvou operací.
- Iterator Splits (`<list/tuple/lazy list> :: <var>` nebo `<lazy list>`): porovná počátek iteráblu (`collections.abc.Iterable`) s `<list/tuple/lazy list>`, potom připojí zbytek k `<var>` nebo ověří, že je iteráble proveden.
- Complex String Matching (`<string> + <var> + <string>`): porovná stringy, které začínají a končí danými substringy, přiřazujíce prostředek k `<var>`.

*Poznámka: Podobně jako u* krájení iterátoru*, porovnávání iterátoru a líného seznamu nezaručují, že původní porovnávaný iterátor zůstane zachovaný (pro zachování iterátoru použijte funkci* `tee` *nebo* `reitarable`*).*

Při ověřování zda může být objekt porovnáván určitým způsobem používá Coconut abstraktní bázové třídy Pythonu. Je tedy nutné registrovat uživatelský objekt jako příslušnou bázovou třídu.

### Příklady

**Coconut**

```
def factorial(value):
    match 0 in value:
        return 1
    else: match n is int in value if n > 0: # possible because of Coconut's
        return n * factorial(n-1)           #   enhanced else statements
    else:
        raise TypeError("invalid argument to factorial of: "+repr(value))

3 |> factorial |> print
```

*Demonstrace příkazů `else`, které pracují skoro stejně jako v Pythonu: kód pod příkazem `else` je proveden pouze tehdy, když selže příslušející srovnávání.*

```
data point(x, y):
    def transform(self, other):         # konstruktor
        match point(x, y) in other:
            return point(self.x + x, self.y + y)
        else:
            raise TypeError("arg to transform must be a point")
    def __eq__(self, other):            # konstruktor
        match point(=self.x, =self.y) in other:
            return True
        else:
            return False

point(1,2) |> point(3,4).transform |> print
point(1,2) |> point(1,2).__eq__ |> print
```

*Demonstrace porovnávání datových typů. Hodnoty, definované příkazem `data` mohou být konfrontovány a jejich obsahy zpřístupněny s použitím konstruktorů datového typu `point`.*

```
data Empty()
data Leaf(n)
data Node(l, r)
Tree = (Empty, Leaf, Node)  # type union

def depth(Tree()) = 0

@addpattern(depth)
def depth(Tree(n)) = 1

@addpattern(depth)
def depth(Tree(l, r)) = 1 + max([depth(l), depth(r)])

Empty() |> depth |> print
Leaf(5) |> depth |> print
Node(Leaf(2), Node(Empty(), Leaf(3))) |> depth |> print
```

*Ukázka kombinace datových typů a porovnávacích (match) příkazů při opakovaném použití algebraických datových typů v jiných funkcionálních programovacích jazycích.*

```
def duplicate_first([x] + xs as l) =
    [x] + l

[1,2,3] |> duplicate_first |> print
```

*Ukázka head-tail krájení (splitting), jednoho z nejvíce používaného způsobu užití pattern-matching, kde `+ <var>` (nebo `:: <var>` pro jakýkoli iterábl) na konci seznamu nebo enticového literálu může být použit k porovnání se*

*zbytkem sekvence.*

```
def sieve([head] :: tail) =
    [head] :: sieve(n for n in tail if n % head)

@addpattern(sieve)
def sieve((||)) = []
```

*Ukazuje, jak porovnávat vůči iterátorům, totiž že případ prázdného iterátoru ( (\|\|) ) musí přijít jako poslední, jinak tento případ vyčerpá celý iterátor před tím, než přijde ke slovu porovnání s jakoukoli jinou předlohou.*

**Python**

*Nelze provést bez dlouhé řady kontrol pro každý příkaz* `match`*. Viz kompilovaný kód pro skladbu Pythonu.*

### 3.6.3 `case`

Příkaz `case` je rozšíření příkazu `match` pro potřebu opakovaného použití příkazů `match` vůči stejné hodnotě. Na rozdíl od osamělých příkazů `match` může uvnitř bloku `case` být úspěšný pouze jeden příkaz match. Obecnější shody (matches) mají být uvedeny pod konkretnějšími shodami.

Každý vzorek v bloku case je porovnáván, dokud není nalezena shoda. Poté se provede příslušné tělo a blok je ukončen. Skladba pro bloky case je

```
case <value>:
    match <pattern> [if <cond>]:
        <body>
    match <pattern> [if <cond>]:
        <body>
    ...
[else:
    <body>]
```

kde `<pattern>` je jakýkoli vzorek pro hledání shody, `<value>` je porovnávaná položka, `<cond>` je volitelná kontrola a `<body>` je kód, který se provede při úspěchu záhlaví. Všimněte si nepřítomnosti `in` v příkazech `match`.

#### Příklad

**Coconut**

```
def classify_sequence(value):
    out = ""        # unlike with normal matches, only one of the patterns
    case value:     #  will match, and out will only get appended to once
        match ():
            out += "empty"
        match (_,):
            out += "singleton"
        match (x,x):
            out += "duplicate pair of "+str(x)
        match (_,_):
            out += "pair"
        match _ is (tuple, list):
            out += "sequence"
    else:
        raise TypeError()
    return out
```

```
[] |> classify_sequence |> print
() |> classify_sequence |> print
[1] |> classify_sequence |> print
(1,1) |> classify_sequence |> print
(1,2) |> classify_sequence |> print
(1,1,1) |> classify_sequence |> print
```

**Python**

*Nelze provést bez dlouhé řady kontrol pro každý příkaz* `match`*. Viz kompilovaný kód pro skladbu Pythonu.*

### 3.6.4 `where`

Příkaz `where` je velmi přímočarý s touto syntaxí:

```
<stmt> where:
    <body>
```

kde `<body>` se skládá pouze z příkazů přiřazení. Příkaz `where` pouze provede zadaná přiřazení v `<body>` a potom vyhodnotí výchozí `<stmt>`.

#### Example

**Coconut:**

```
c = a + b where:
    a = 1
    b = 2
```

**Python:**

```
a = 1
b = 2
c = a + b
```

### 3.6.5 Backslash-Escaping

Klíčová slova `data`, `match`, `case`, `async` (keyword in Python 3.5) a `await` (keyword in Python 3.5) jsou v Coconut rovněž platná jména proměnných. I když Coconut umí tyto dva případy použití rozlišit, je možné pro zvýraznění použít před takovýmto názvem proměnné zpětné lomítko.

#### Příklad

**Coconut:**

```
\data = 5
print(\data)
```

**Python:**

```
data = 5
print(data)
```

## 3.7 Výrazy

### 3.7.1 Příkaz lambda

Skladba příkazu lambda je rozšířením normální skladby *lambda* pro podporu příkazů, nikoliv pouze výrazů.

Skadba pro příkaz lambda je:

```
def (arguments) -> statement; statement; ...
```

kde `statement` může být příkaz přiřazení nebo keyword statement. Je-li poslední `statement` (nenásledovaný středníkem) `výrazem`, je automaticky vrácen.

Příkazy lambda rovněž podporují implicitní skladbu lambda, u níž je při vypuštění argumentů, jako v `def -> _`, předpokládán výraz `def (_=None) -> _`.

#### Příklad

**Coconut:**

```
L |> map$(def (x) -> y = 1 / x; y*(1 - y))
```

**Python:**

```
def _lambda(x):
    y = 1 / x
    return y*(1 - y)
map(_lambda, L)
```

### 3.7.2 Líné seznamy

Coconut podporuje vytváření líných seznamů (lazy lists), jejichž obsah je považován za iterátor a není vyhodnocen, dokud není zapotřebí. Líné seznamy (lazy lists) se v Coconut vytvářejí jednoduše uzavřením čárkami odděleného výčtu do specielních závorek `(|` a `|)` (takzvaných "banana brackets") místo do `[` a `]` u seznamů nebo do `(` a `)` u entic.

Líné seznamy používají ke zlenivění stejný mechanizmus jako u iterátorového řetězení a tudíž je líný seznam `(| x, y |)` ekvivalentní výrazu iterátorového řetězení `(x,) :: (y,)`, byť líný seznam nevytváří mezilehlé entice.

#### Zdůvodnění

Líné seznamy, jejichž sekvence jsou vyhodnocovány jen v případě potřeby, jsou stěžejním útvarem funkcionálního programování, umožňujícím dynamické vyhodnocování jejich obsahu.

#### Příklad

**Coconut:**

```
(| print("hello,"), print("world!") |) |> consume
```

**Python:** *Nelze provést bez složité komprehence iterátoru. Viz kompilovaný kód pro skladbu Pythonu.*

### 3.7.3 Implicitní částečná aplikace

Coconut podporuje řadu různých skladebných aliasů pro obecné případy částečné aplikace. Jsou to:

```
.attr           =>      operator.attrgetter("attr")
.method(args)   =>      operator.methodcaller("method", args)
obj.            =>      getattr$(obj)
func$           =>      ($)$(func)
seq[]           =>      operator.getitem$(seq)
iter$[]         =>      # the equivalent of seq[] for iterators
.[a:b:c]        =>      operator.itemgetter(slice(a, b, c))
.$[a:b:c]       =>      # the equivalent of .[a:b:c] for iterators
```

#### Příklad

**Coconut:**

```
1 |> "123"[]
mod$ <| 5 <| 3
```

**Python:**

```
"123"[1]
mod(5, 3)
```

### 3.7.4 Operátorové funkce

Coconut používá jednoduchou zkratku pro vyjádření operátorové funkce: obklopení operátoru kulatými závorkami. Podobně jako u komprehencí iterátoru, je-li operátorová funkce jediným argumentem funkce, mohou závorky pro volání funkce sloužit také jako závorky operátorové funkce.

#### Zdůvodnění

Obvyklou věcí při funkcionálním programování je využití funkčních verzí vestavěných operátorů: jejich 'currying', `composing` a `piping`. Coconut nabízí zjednodušenou syntaxi pro přístup k operátorovým funkcím.

#### Úplný seznam

```
(|>)         => # pipe forward
(|*>)        => # multi-arg pipe forward
(<|)         => # pipe backward
(<*|)        => # multi-arg pipe backward
(..), (<..) => # backward function composition
(..>)        => # forward function composition
(<*..)       => # multi-arg backward function composition
(..*>)       => # multi-arg forward function composition
(.)          => (getattr)
(::)         => (itertools.chain)  # will not evaluate its arguments lazily
($)          => (functools.partial)
($[])        => # iterator slicing operator
(+)          => (operator.add)
(-)          => # 1 arg: operator.neg, 2 args: operator.sub
```

```
(*)          => (operator.mul)
(**)         => (operator.pow)
(/)          => (operator.truediv)
(//)         => (operator.floordiv)
(%)          => (operator.mod)
(&)          => (operator.and_)
(^)          => (operator.xor)
(|)          => (operator.or_)
(<<)         => (operator.lshift)
(>>)         => (operator.rshift)
(<)          => (operator.lt)
(>)          => (operator.gt)
(==)         => (operator.eq)
(<=)         => (operator.le)
(>=)         => (operator.ge)
(!=)         => (operator.ne)
(~)          => (operator.inv)
(@)          => (operator.matmul)
(not)        => (operator.not_)
(and)        => # boolean and
(or)         => # boolean or
(is)         => (operator.is_)
(in)         => (operator.contains)
```

#### Příklad

**Coconut:**

```
(range(0, 5), range(5, 10)) |*> map$(+) |> list |> print
```

**Python:**

```
import operator
print(list(map(operator.add, range(0, 5), range(5, 10))))
```

### 3.7.5 Vylepšené anotace typu

Protože je syntaxe Coconutu nadmnožinou syntaxe Python3, podporuje sladbu anotace typu Pythonu 3 a skladbu anotace proměnné typu Pythonu 3.6. Implicitně kompiluje Coconut všechny anotace typu na typové komentáře, kompatibilní s Python 2. Chcete-li zachovat anotace typu, zadejte flag –target, který je podporuje.

Protože ne všechny podporované verze Pyhonu podporují modul `typing`, poskytuje Coconut vestavěnou proceduru *`TYPE_CHECKING`* pro zakrytí importů `typing` a definicí `TypeVar` před vyhodnocením při runtime. Kromě toho při kompilaci anotace typů do syntaxe Python3, zabalí Coconut tyto anotace do řetězců aby je uchránil před vyhodnocením při runtime.

Navíc, Coconut přidává speciální syntaxi pro zjednodušení zápisu anotací. Uvnitř anotace typu zachází Coconut s některými konstrukty odlišně, kompilujíc je na anotaci typu místo na to, co by normálně představovaly. Konkrétně, Coconat používá následující transformace:

```
<type>?
    => typing.Optional[<type>]
<type>[]
    => typing.Sequence[<type>]
<type>$[]
```

```
    => typing.Iterable[<type>]
() -> <ret>
    => typing.Callable[[], <ret>]
<arg> -> <ret>
    => typing.Callable[[<arg>], <ret>]
(<args>) -> <ret>
    => typing.Callable[[<args>], <ret>]
-> <ret>
    => typing.Callable[..., <ret>]
```

kde `typing` je standardní modul Pythonu 3.5.

#### Příklad

**Coconut:**

```
def int_map(
    f: int -> int,
    xs: int[],
) -> int[] =
    xs |> map$(f) |> list
```

**Python:**

```
import typing  # unlike this typing import, Coconut produces universal code
def int_map(
    f,  # type: typing.Callable[[int], int]
    xs,  # type: typing.Sequence[int]
):
    # type: (...) -> typing.Sequence[int]
    return list(map(f, xs))
```

### 3.7.6 Literály setu

Coconut umožňuje předsadit písmeno `s` před literály setu. Byť to ve většině případů nedělá nic, v případě prázdného setu to indikuje, že se jedná o `set` a nikoliv o `dictionary`. Spojení `s{}` informuje Coconut, že jde o prázdný set a nikoli o prázdný slovník. Spojení `f{}` generuje `frozenset`.

#### Příklad

**Coconut**

```
empty_frozen_set = f{}
```

**Python**

```
empty_frozen_set = frozenset()
```

### 3.7.7 Imaginární literály

Jako doplněk k notaci imaginárních literálů`<num>j` nebo `<num>J` v Pythonu, podporuje Coconut také notace `<num>i` nebo `<num>I` pro zlepšení čitelnosti imaginárních literálů při použití v matematickém kontextu.

#### Python Docs

Literály imaginárního čísla (imaginární literály) jsou popsány následujícími lexikálními definicemi:

```
imagnumber ::= (floatnumber | intpart) ("j" | "J" | "i" | "I")
```

Imaginární literál generuje komplexní číslo s hodnotou reálné části o velikosti 0.0. Komplexní čísla jsou prezentována jako dvojice desetinných čísel se stejným omezením jejich rozsahu. Komplexní číslo s nenulovou reálnou částí vytvoříte přidáním desetinného čísla, např. (3+4i). Několik příkladů imaginárních literálů (neboli imaginárních částí):

```
3.14i   10.i    10i     .001i   1e100i  3.14e-10i
```

#### Příklad

**Coconut**

```
3 + 4i |> abs |> print
```

**Python**

```
print(abs(3 + 4j))
```

## 3.8 Definice funkce

### 3.8.1 Optimalizace koncového volání

Coconut provede automatickou optimalizaci a eliminaci koncové rekurze u každé funkce, která vyhoví následujícím kriteriím:

1. musí přímo vrátit (s použitím buď `return` nebo *přiřazovací funkce*) volání sama sebe (eliminace koncového volání - nejúčinnější optimalizace) nebo jiné funkce (optimalizace koncového volání).
2. nesmí to být generátor (používající `yield`) nebo asynchronní funkce (používající`async`).

*Note: Optimalizace koncového volánií (byť ne eliminace koncové rekurze) pracuje i pro 1) vzájemnou rekurzi a 2) porovnávací (pattern-matching) funkce, rozdělené do několika definicí s pouožitím* `addpattern`.

Setkáte-li se s `RuntimeError` v souvislosti s maximální hloubkou rekurze, je velmi vhodné přepsat svou funkci aby vyhověla výše uvedenému kriteriu pro optimalizaci koncovým voláním nebo odpovídajícímu kriteriu pro *`recursive_iterator`*, obojí by mělo takové chybě zabránit.

*Note: Optimalizace koncového volání (byť ne eliminace koncové rekurze) se vypne, zadáte-li flag `--no-tco`, což je užitečné, máte-li potíže se čtením svých `tracebacks` a potřebujete maximální výkon.*

#### Příklad

**Coconut**

```
# na rozdíl od Pythonu, tato funkce nikdy nedospěje k chybě maximální hloubky rekurze
def factorial(n, acc=1):
    case n:
        match 0:
            return acc
```

```
        match _ is int if n > 0:
            return factorial(n-1, acc*n)
```

*Demonstruje eliminaci koncové rekurze.*

```
# unlike in Python, neither of these functions will ever hit a maximum recursion␣
↪depth error
def is_even(0) = True
@addpattern(is_even)
def is_even(n is int if n > 0) = is_odd(n-1)

def is_odd(0) = False
@addpattern(is_odd)
def is_odd(n is int if n > 0) = is_even(n-1)
```

*Demonstruje optimalizaci koncové rekurze.*

**Python**

*Nelze provést bez přepsání funkce.*

### 3.8.2 Přiřazovací funkce

Coconut umožňuje definování přiřazovací funkce tak, aby automaticky vrátila poslední řádek těla funkce. Přiřazovací funkce je vyjádřena náhradou = za :, takže složení přiřazovací funkce je buď

```
def <name>(<args>) = <expr>
```

pro jednořádkovou funkci nebo

```
def <name>(<args>) =
    <stmts>
    <expr>
```

pro víceřádkovou funkci, kde <name> je název funkce, <args> jsou argumenty funkce, <stmts> jsou přípustné příkazy a <expr> je hodnota, kterou má funkce vrátit.

*Note: Definice přiřazovací funkce může být kombinována s definicí infixové a/nebo porovnávací (pattern-matching) funkce.*

#### Zdůvodnění

Zápis definice přiřazovací funkce je stejně snadný jako přiřazení k funkci lambda a objeví se ve zpětných výpisech (tracebacks), protože kompiluje na normální definici funkce Pythonu.

#### Příklad

**Coconut**

```
def binexp(x) = 2**x
5 |> binexp |> print
```

**Python**

```
def binexp(x): return 2**x
print(binexp(5))
```

### 3.8.3 Funkce pro pattern matching

Tyto funkce Coconutu jsou normální funkce, kde argumenty jsou vzory k porovnávání, místo proměnných pro přiřazení hodnot. Skladba definice porovnávací (pattern-matching) funkce je

```
[match] def <name>(<arg>, <arg>, ... [if <cond>]):
    <body>
```

where <arg> je definován jako

```
[*|**] <pattern> [= <default>]
```

kde <name> je název funkce, <cond> je nepovinná dodatečná kontrola, <body> je tělo funkce, <pattern> je definován *příkazem* `match` a <default> je volitelná implicitní hodnota, není-li žádný argument zadán. Klíčové slovo `match` na začátku je nepovinné ale je někdy nezbytné pro odlišení definice porovnávací funkce od normální definice funkce, která má vždy přednost.

Je-li <pattern> jméno proměnné (přímo nebo s <as>), podporuje výsledná porovnávací funkce klíčové argumenty stejného jména. Jestliže provedení porovnávací funkce selže, vyvolá objekt `MatchError`, stejně jako *rozložené přiřazení*.

*Note: Definice porovnávací funkce může být kombinována s definicí přiřazovací a/nebo infixové funkce.*

#### Příklad

**Coconut**

```
def last_two(_ + [a, b]):
    return a, b
def xydict_to_xytuple({"x":x is int, "y":y is int}):
    return x, y

range(5) |> last_two |> print
{"x":1, "y":2} |> xydict_to_xytuple |> print
```

**Python**

*Nelze provést bez dlouhé řady kontrol na počátku funkce. Viz kompilovaný kód pro skladbu Pythonu.*

### 3.8.4 Infixové funkce

Coconut umožňuje infixové volání funkce, kde je výraz, vyhodnocovaný na funkci, obklopen zpětnými apostrofy; argumenty mohou být uvedeny před nebo za funkcí. Infixové volání má prioritu mezi 'chaining and None-coalescing' (řetězením a sloučením) a je asociativní vlevo.

Coconut také podporuje definování jednodušší infixové funkce:

```
def <arg> `<name>` <arg>:
    <body>
```

kde <name> je název funkce, <arg> jsou parametry funkce a <body> je tělo funkce. Obsahuje-li <arg> implicitní hodnotu, musí být uvedena v závorkách.

*Note: Definici infixové funkce lze kombinovat s definicí přiřazovací a/nebo porovnávací (pattern-matching) funkce.*

#### Zdůvodnění

Obvyklým idiomem ve funkcionálním programování je psaní funkcí, zamýšlených jako operátory a volat je i definovat vložením mezi své argumenty. Infixová syntaxe Coconutu to umožňuje.

#### Příklad

**Coconut**

```
def a `mod` b = a % b
(x `mod` 2) `print`
```

**Python**

```
def mod(a, b): return a % b
print(mod(x, 2))
```

### 3.8.5 Definice funkce s tečkami

Coconut umožňuje definovat funkci s použitím vytečkovaného jména pro přiřazení funkce jako methody objektu, jak je specifikováno v PEP 542.

#### Příklad

**Coconut:**

```
def MyClass.my_method(self):
    ...
```

**Python:**

```
def my_method(self):
    ...
MyClass.my_method = my_method
```

## 3.9 Příkazy

### 3.9.1 Rozkladné přiřazení

Coconut podporuje výrazně zlepšené rozkladné přiřazení (destructuring assignment), podobné rozkládání entice/seznamu v Pythonu. Skladba rozkladného přiřazení je

```
[match] <pattern> = <value>
```

kde <value> je libovolný výraz a <pattern> je definován *příkazem match*. Klíčové slovo match na začátku je nepovinné ale je někdy nezbytné pro odlišení rozloženého přiřazení od normálního přiřazení, které má vždy přednost. Rozložené přiřazení v Coconut je ekvivalentní příkazu match, jehož skladba je:

```
match <pattern> in <value>:
    pass
else:
    err = MatchError(<error message>)
    err.pattern = "<pattern>"
    err.value = <value>
    raise err
```

Selže-li provádění rozkladného přiřazení, potom místo pokračování jako při selhání u bloku match, je evokován objekt *MatchError*, popisující selhání.

#### Příklad

**Coconut**

```
def last_two(l):
    _ + [a, b] = l
    return a, b

[0,1,2,3] |> last_two |> print
```

**Python**

*Nelze provést bez dlouhé řady kontrol místo příkazu rozkladného přiřazení. Viz kompilovaný kód pro skladbu Pythonu.*

### 3.9.2 Dekorátory

Narozdíl od Pythonu, který v dekorátoru podporuje pouze jedinou proměnnou nebo volání funkce, podporuje Coconut libovolný výraz.

#### Příklad

**Coconut**

```
@ wrapper1 .. wrapper2 $(arg)
def func(x) = x**2
```

**Python**

```
def wrapper(func):
    return wrapper1(wrapper2(arg, func))
@wrapper
def func(x):
    return x**2
```

### 3.9.3 Zanořování příkazů

Coconut podporuje vnořování složených příkazů na témže řádku. To umožňuje spojování příkazů match a if, stejně jako složené příkazy try.

#### Příklad

**Coconut:**

```
if invalid(input_list):
    raise Exception()
else: match [head] + tail in input_list:
    print(head, tail)
else:
    print(input_list)
```

**Python:**

```
from collections.abc import Sequence
if invalid(input_list):
    raise Exception()
elif isinstance(input_list, Sequence):
    head, tail = inputlist[0], inputlist[1:]
    print(head, tail)
else:
    print(input_list)
```

### 3.9.4 Příkazy `except`

Python 3 vyžaduje, že mají-li být odchyceny víceré výjimky, musí být umístěny v uvozovkách aby se znemožnilo použití čárky místo `as` v Python 2. Coconut umožňuje čárky ve výjimkových příkazech za účelem odchycení vícerých výjimek bez použití uvozovek, protože - stejně jako v Python3 - od `as` se vždy požaduje připojit výjimku ke jménu.

#### Example

**Coconut:**

```
try:
    unsafe_func(arg)
except SyntaxError, ValueError as err:
    handle(err)
```

**Python:**

```
try:
    unsafe_func(arg)
except (SyntaxError, ValueError) as err:
    handle(err)
```

### 3.9.5 Implicitní `pass`

Coconut umožňuje jednoduchý zápis `class name(base)` a `data name(args)` jako aliasy pro `class name(base): pass` a `data name(args): pass`.

#### Příklad

**Coconut**

```
class Tree
data Empty from Tree
data Leaf(item) from Tree
data Node(left, right) from Tree
```

**Python:** *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

```
### In-line `global` a `nonlocal` přiřazení

Coconut umožňuje použití slov `global` nebo `nonlocal` před přiřazením k proměnné␣
↪nebo k seznamu proměnných, činíce tak přiřazení `globální` případně `nelokální`.

##### Příklad

**Coconut:**
```coconut
global state_a, state_b = 10, 100
```

**Python:**

```
global state_a, state_b; state_a, state_b = 10, 100
```

### 3.9.6 Průchod kódu

Kvůli kompatibilitě s jinými variantami Pythonu, jako je Cython nebo Mython, podporuje Coconut schopnost protáhnout inertním způsobem libovolný kód kompilátorem. Cokoli umístěného mezi `\(` a `\)` projde netečně kompilátorem, stejně jako řádek, začínající `\\`, umožňující navíc následnou indentaci.

#### Příklad

**Coconut**

```
\\cdef f(x):
    return x |> g
```

**Python**

```
cdef f(x):
    return g(x)
```

### 3.9.7 Vylepšené závorkové pokračování

Protože je syntaxe Coconut nadmnožinou syntaxe Python 3, podporuje Cooconut stejnou formu pokračování řádku jako Python. To znamená, že lze použít jak pokračování se zpětným lomítkem nebo implikované pokračování uvnitř kulatých, hranatých či složených závorek.

V Pythonu je ovšem několik případů (např. víceré příkazy `with`), kde lze použít pouze pokračování se zpětným lomítkem. Ve všech těchto případech podporuje Coconut i závorkové pokračování.

Podporu univerzálního použití závorkového pokračování povoluje konvence PEP 8 :
```

Upřednostňovaný způsob ukončování dlouhých řádků je implikované pokračování uvnitř kulatých, hranatých či složených závorek. Dlouhé řádky mohou být uvnitř závorek rozděleny do více kratších řádků. Tento způsob má přednost před používáním zpětných lomítek pro pokračování řádků.

*Note: Použití flagu `--strict` vyloučí použití zpětných lomítek.*

#### Příklad

**Coconut:**

```
with (open('/path/to/some/file/you/want/to/read') as file_1,
      open('/path/to/some/file/being/written', 'w') as file_2):
    file_2.write(file_1.read())
```

**Python:**

```
# split into two with statements for Python 2.6 compatibility
with open('/path/to/some/file/you/want/to/read') as file_1:
    with open('/path/to/some/file/being/written', 'w') as file_2:
        file_2.write(file_1.read())
```

## 3.10 Vestavěné funkce

### 3.10.1 Vylepšené vestavěné funkce

Objekty `map`, `zip`, `filter`, `reversed` a `enumerate` jsou vylepšené verze svých ekvivalentů v Pythonu, které podporují procedury `reversed`, `repr`, optimalizované (a iterátorové) `krájení` (slicing), `len` (vše až na `filter`) a mají přidané atributy, jež mohou subtříty použít pro přístup původním argumentům objektu:

- `map`: `_func`, `_iters`
- `zip`: `_iters`
- `filter`: `_func`, `_iter`
- `reversed`: `_iter`
- `enumerate`: `_iter`, `_start`

#### Příklad

**Coconut:**

```
map((+), range(5), range(6)) |> len |> print
range(10) |> filter$((x) -> x < 5) |> reversed |> tuple |> print
```

**Python:** *Nelze provést bez definování uživatelskéko typu `map`. Úplnou definici `map` lze nalézt v záhlaví Coconut.*

### 3.10.2 `addpattern`

Tato funkce přijímá argument, jenž je *pattern-matching funkcí* a vrací dekorátor, který přidává předlohy z existující funkce do nové dekorované funkce, v níž je existující předloha ověřována jako první. Její skladba je zhruba ekvivalentní k:

```
def addpattern(base_func):
    """Decorator to add a new case to a pattern-matching function, where the new case␣
↪is checked last."""
    def pattern_adder(func):
        def add_pattern_func(*args, **kwargs):
            try:
                return base_func(*args, **kwargs)
            except MatchError:
                return func(*args, **kwargs)
        return add_pattern_func
    return pattern_adder
```

Mějte na paměti, že funkce, převzatá dekorátorem `addpattern`, musí být 'pattern-matchingová' funkce. Obdrží-li `addpattern` 'non-pattern-matchingovou' funkci, nevyvolá původní funkce hlášení `MatchError`. Tudíž, při volání této funkce nebude `addpattern` vědět, že první shoda selhala a správné cesty nebude nikdy dosaženo.

Například, následující kód vyvolá `TypeError`:

```
def print_type():
    print("Received no arguments.")

@addpattern(print_type)
def print_type(x is int):
    print("Received an int.")

print_type()  # appears to work
print_type(1) # TypeError: print_type() takes 0 positional arguments but 1 was given
```

Toto může být napraveno použitím klíčového slova `match`, činíce tak z funkce `print_type()` funkci pattern-matchingovou. V důsledku toho se všechna hlášení `TypeErrors` přemění na `MatchErrors`. Tato hlášení mohou být potom ošetřena dle potřeby novými `addpattern` dekorovanými funkcemi.

```
match def print_type():
    print("Received no arguments.")

@addpattern(print_type)
def print_type(x is int):
    print("Received an int.")

print_type(1)  # Works as expected
print_type("This is a string.") # Raises MatchError
```

`addpattern` může být použit bez klíčového slova `match`, pokud je přijímaná funkce *přiřazovací funkcí*, jak ukázáno v následujícím příkladu:

### Příklad

**Coconut**

```
def factorial(0) = 1

@addpattern(factorial)
def factorial(n) = n * factorial(n - 1)
```

**Python** *Nelze provést bez komplikované definice dekorátoru a dlouhé řady kontrol pro každé porovnávání. Viz kompilovaný kód pro skladbu Pythonu.*

#### prepattern

**DEPRECATED:** Coconut also has a `prepattern` built-in, which adds patterns in the opposite order of `addpattern`; `prepattern` is defined as:

```
def prepattern(base_func):
    """Decorator to add a new case to a pattern-matching function,
    where the new case is checked first."""
    def pattern_prepender(func):
        return addpattern(func)(base_func)
    return pattern_prepender
```

*Note: Passing `--strict` disables deprecated features.*

### 3.10.3 reduce

Coconut znovu uvádí funkci `reduce` z Python2, používaje verze `functools.reduce`.

#### Python Docs

**reduce**(*function, iterable*[, *initializer*])

Funkce `reduce` použije opakovaně funkci se dvěmi proměnnými pro iterovatelný objekt, kumulujíc mezivýsledky do výsledné výstupní hodnoty. Například, `reduce((x, y) -> x+y, [1, 2, 3, 4, 5])` počítá `((((1+2)+3)+4)+5)`. Levý argument *x* je akumulovaná hodnota a pravý argument *y* je aktuální hodnota ze sekvence. Je-li přítomen nepovinný *iniciátor*, je umístěn před položky sekvence a slouží jako implicitní hodnota, je-li sekvence prázdná.

#### Příklad

**Coconut**

```
product = reduce$(*)
range(1, 10) |> product |> print
```

**Python**

```
import operator
import functools
product = functools.partial(functools.reduce, operator.mul)
print(product(range(1, 10)))
```

### 3.10.4 takewhile

Coconut poskytuje `itertools.takewhile` jako vestavěnou funkci pod názvem `takewhile`.

#### Python Docs

**takewhile**(*predicate, iterable*)

Vytvoří iterátor, který vrací prvky *iteráblu*, pokud je *predicate* pravdivý. Ekvivalentní k:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

#### Příklad

**Coconut**

```
negatives = takewhile(numiter, (x) -> x<0)
```

**Python**

```
import itertools
negatives = itertools.takewhile(numiter, lambda x: x<0)
```

### 3.10.5 `dropwhile`

Coconut poskytuje `itertools.dropwhile` jako vestavěnou funkci pod názvem `dropwhile`.

#### Python Docs

**dropwhile**(*predicate, iterable*)

Vytvoří iterátor, který vypouští prvky z *iteráblu* pokud je *predicate* pravdivý; poté vrací každý prvek. Poznámka: iterátor neprodukuje žádný výstup, dokud se predikát poprvé nestane nepravdivý. Ekvivalentní k:

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

#### Příklad

**Coconut**

```
positives = dropwhile(numiter, (x) -> x<0)
```

**Python**

```
import itertools
positives = itertools.dropwhile(numiter, lambda x: x<0)
```

### 3.10.6 memoize

Coconut poskytuje `functools.lru_cache` jako 'built-in' pod jménem `memoize`, s tou úpravou, že parametr *maxsize* je nastaven implicitně na `None`. Procedura `memoize` usnadňuje optimalizaci rekurzivních funkcí, neboť *maxsize* argumentu `None` je obvykle to, co se žádá.

Použití `memoize` vyžaduje `functools.lru_cache`, jež existuje ve standardní knihovně v Python 3 ale v Python 2 bude požadovat `pip install backports.functools_lru_cache`. Navíc, při přítomnosti `backports.functools_lru_cache` v Python2, bude Coconut 'patch' `functools` tak, že `functools.lru_cache = backports.functools_lru_cache.lru_cache`.

#### Python Docs

**memoize**(*maxsize=None, typed=False*)

Dekorátor pro spojení funkce s memoizačním volatelným objektem (memoizing callable), který ukládá poslední volání v počtu až *maxsize*. Může šetřit čas, je-li opakovaně volána náročná I/O funkce pro stejné argumenty.

Protože se pro ukládání výsledků používá slovník, musejí být poziční a klíčově slovní (KW) argumenty hešovatelné.

Je-li parametr *maxsize* nastaven na `None`, je LRU vlastnost vypnuta (disabled) a cache může růst bez omezení. LRU feature pracuje nejlépe, je-li *maxsize* rovno druhé mocnině (power-of-two).

Je-li parametr *typed* nastaven na `true`, jsou argumenty funkce různých typů ukládány odděleně. Například, `f(3)` a `f(3.0)` jsou šetřeny jako odlišná volání s odlišnými výsledky.

Za účelem měření účinnosti cache a pro vyladění parametru *maxsize* je obalová funkce vybavena funkcí `cache_info()`, která vrací pojmenovanou entici, ukazující *hits*, *misses*, *maxsize* a *currsize*. Ve vícevláknovém prostředí jsou hodnoty 'hits' a 'mises' přibližné.

Dekorátor také poskytuje funkci `cache_clear()` pro mazání nebo zneplatnění cache.

Základní funkce je přístupná přes atribut `__wrapped__`. Ten je užitečný pro introspekci, pro obejití cache nebo pro překrytí funkce (rewrapping) jinou cache.

Cache LRU (least recently used) pracuje nejlépe, když poslední volání jsou prediktory nadcházejících volání (například, nejpopulárnější články na serveru se mění každý den). Limit pro velikost této paměti zajišťuje, že cache neroste bez omezení u dlouho běžících procesů, jako jsou webové servery.

Příklad obsahu cache LRU cache pro statický webový obsah:

```
@memoize(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

#### Example

**Coconut:**

```
def fib(n if n < 2) = n

@memoize()
@addpattern(fib)
def fib(n) = fib(n-1) + fib(n-2)
```

**Python:**

```
try:
    from functools import lru_cache
except ImportError:
    from backports.functools_lru_cache import lru_cache
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

### 3.10.7 `groupsof`

Coconut poskytuje funkci `groupsof` pro rozdělení (splitting) iterovatelného objektu do skupin určiné délky. Konkretně, `groupsof(n, iterable)` rozdělí `iterábl` to entic délky `n`, případně u poslední entice délky `< n`, není-li délka `iteráblu` dělitelná `n`.

#### Příklad

**Coconut:**

```
pairs = range(1, 11) |> groupsof$(2)
```

**Python:**

```
pairs = []
group = []
for item in range(1, 11):
    group.append(item)
    if len(group) == 2:
        pairs.append(tuple(group))
        group = []
if group:
    pairs.append(tuple(group))
```

### 3.10.8 `tee`

Coconut poskytuje optimalizovanou verzi `itertools.tee` jako vestavěnou funkci pod názvem `tee`.

#### Python Docs

**tee**(*iterable, n=2*)

Vrací *n* nezávislých iterátorů z jediného iteráblu. Ekvivalentní k:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                newval = next(it)       # fetch a new value and
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

Jakmile `tee()` provede rozdělení, neměl by být původní *iterábl* jinde používáný, neboť by se mohl *iterable* přesunout bez uvědomění objektu tee.

Tento itertool může vyžadovat významný pomocný úložný prostor (v závisloti na tom, jak mnoho dočasných dat má být uloženo). Obecně lze říci, že když jeden iterátor použije většinu ze všech dat před tím, než spustí další iterátor, je rychlejší použít `list()` místo `tee()`.

#### Příklad

**Coconut**

```
original, temp = tee(original)
sliced = temp$[5:]
```

**Python**

```
import itertools
original, temp = itertools.tee(original)
sliced = itertools.islice(temp, 5, None)
```

### 3.10.9 `reiterable`

Někdy, kdy je zapotřebí aby byl iterátor opakovaně iterován, může být použití `tee` nešikovné. Pro takový případ poskytuje Coconut proceduru `reiterable`, která zabalí daný iterátor tak, že iterace se provádí po objektu `tee` místo po původním iterátoru.

#### Příklad

**Coconut:**

```
def list_type(xs):
    case reiterable(xs):
        match [fst, snd] :: tail:
            return "at least 2"
        match [fst] :: tail:
            return "at least 1"
        match (| |):
            return "empty"
```

**Python:** *Nelze provést bez dlouhé řady kontrol pro každý příkaz `match`. Viz kompilovaný kód pro skladbu Pythonu.*

### 3.10.10 consume

Coconut poskytuje funkci `consume` pro účinné vyčerpání iterátoru a pro provedení líného výpočtu. Funkce `consume` přijímá volitelný argument, `keep_last`, jehož implicitní hodnota je 0 a určuje kolik položek od konce vrátit jako iterábl (`None` zachová všechny prvky).

Ekvivalentní k:

```
def consume(iterable, keep_last=0):
    """Fully exhaust iterable and return the last keep_last elements."""
    return collections.deque(iterable, maxlen=keep_last) # fastest way to exhaust an␣
↪iterator
```

#### Zdůvodnění

V procesu líného provádění operací na iterátorech je posléze dosaženo místa, kde je vyhodnocení iterátoru nezbytné. Aby to mohlo být provedeno efektivně, poskytuje Coconut funkci `consume`, která zcela vyčerpá poskytnutý iterátor.

#### Příklad

**Coconut**

```
range(10) |> map$((x) -> x**2) |> map$(print) |> consume
```

**Python**

```
collections.deque(map(print, map(lambda x: x**2, range(10))), maxlen=0)
```

### 3.10.11 count

Coconut poskytuje modifikovanou verzi `itertools.count`, která podporuje `in`, normální členění (slicing), optimalizoané členění iterátoru, sekvenční metody `count` a `index`, atributy `repr`, `_start` a `_step` jako vestavěnou funkci jménem `count`.

#### Python Docs

**count**(*start=0, step=1*)

Vytvoří iterátor, který vrátí rovnoměrně rozmístěné hodnoty, počínajíc číslem *start*. Používá se často jako argument funkci `map()` ke generování postupných datových bodů. Také se používá s funkcí `zip()` pro připojení pořadových čísel. Zhruba ekvivalentní k:

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

#### Příklad

**Coconut**

```
count()$[10**100] |> print
```

**Python** *Nelze provést rychle bez iterátorového členění Coconutu, jež vyžaduje mnoho složitých částí. Nezbytné definice v Pythonu lze nalézt v záhlaví Coconut.*

### 3.10.12 makedata

Funkce `makedata` poskytuje přímý přístup k bázovému konstruktoru datových typů, vytvořenému příkazem `data`. To je zejména užitečné při psaní alternativních konstruktorů pro datové typy přepisem `__new__`.

Funkce `makedata` přijímá datový typ jako první argument, následovaný potřebnými argumenty pro vytvoření datového typu. Pro objekty `data` se funkce `makedata` chová jako konstruktor výchozího datového typu, přesně jak byl datový typ deklarován. Pro `nedatové` objekty je `makedata` ekvivalentní k:

```
def makedata(data_type, *args, **kwargs):
    """Returns base data constructor of data_type."""
    return super(data_type, data_type).__new__(data_type, *args, **kwargs)
```

**DEPRECATED:** Coconut má také vestavěný `datamaker`, který částečně aplikuje `makedata`; `datamaker` je definován jako:

```
def datamaker(data_type):
    """Get the original constructor of the given data type or class."""
    return makedata$(data_type)
```

*Note: Passing `--strict` disables deprecated features.*

#### Příklad

**Coconut:**

```
data Tuple(elems):
    def __new__(cls, *elems):
        return elems |> makedata$(cls)
```

**Python:** *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

### 3.10.13 fmap

Ve funkcionálním programování přijímá funkce `fmap(func, obj)` datový typ `obj` a vrací nový datový typ s mapovanou `func` pro celý obsah. Funkce `fmap` v Coconut provádí totéž.

Funkce `fmap` může být rovněž použita pro objekty `str`, `list`, `set` a `dict` jako varianta `map`, vracejíc objekt téhož typu. Chování `fmap` může být pro daný objekt změněno definováním metody `__fmap__(self, func)`, jež bude volána při každé invokaci funkce `fmap`.

Pro `dict` nebo každé `collections.abc.Mapping` je `fmap` voláno pro `.items()` mappingu namísto implicitní iterace po jeho klíčích (`.keys()`).

#### Příklad

**Coconut:**

```
[1, 2, 3] |> fmap$(x -> x+1) == [2, 3, 4]

data Nothing()
data Just(n)

Just(3) |> fmap$(x -> x*2) == Just(6)
Nothing() |> fmap$(x -> x*2) == Nothing()
```

**Python:** *Can't be done without a series of method definitions for each data type. See the compiled code for the Python syntax.*

### 3.10.14 `starmap`

Coconut poskytuje modifikovanou verzi `itertools.starmap` která podporuje procedury `reversed`, `repr`, optimized normal (and iterator) slicing, `len` a `_func` a atributy `_iter`.

#### Python Docs

**starmap**(*function, iterable*)

Vytvoří iterátor, který počítá funkci s použitím argumentů, získaných z iteráblu. Používá se místo `map()`, jsou-li argumenty parametrů již seskupeny do entic z jednoho iteráblu (the data has been "pre-zipped"). Rozdíl mezi `map()` a `starmap()` je obdobný rozdílu mezi `function(a,b)` a `function(*c)`. Je zhruba ekvivalentní k:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

#### Příklad

**Coconut:**

```
range(1, 5) |> map$(range) |> starmap$(print) |> consume
```

**Python:**

```
import itertools, collections
collections.deque(itertools.starmap(print, map(range, range(1, 5))), maxlen=0)
```

### 3.10.15 `scan`

Coconut poskytuje modifikovanou verzi `itertools.accumulate` s opačným pořadím argumentů než má `scan`, který rovněž podporuje `repr`, `len` a `func` a atributy `iter`. `scan` pracuje stejně jako *reduce*, kromě toho, že místo vracení poslední akumulované hodnoty, vrací iterátor se všemi mezilehlými hodnotami.

### Python Docs

**scan**(*func, iterable*)

Vytvoří iterátor, který vrací akumulované výsledky některých funkcí pro dva argumenty. Typy elementů vstupního iteráblu musí být akceptovatelné u argumentů funkce. Například pro sčítání mohou být elementy jakéhokoli sčítatelného typu včetně Decimal nebo Fraction. Je-li vstupní iterábl prázdný, je výstupní iterábl rovněž prázdný.

Jest to zhruba ekvivalentní k:

```
def scan(func, iterable):
    'Return running totals'
    # scan(operator.add, [1,2,3,4,5]) --> 1 3 6 10 15
    # scan(operator.mul, [1,2,3,4,5]) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

### Příklad

**Coconut:**

```
input_data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
running_max = input_data |> scan$(max) |> list
```

**Python:**

```
input_data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
running_max = []
max_so_far = input_data[0]
for x in input_data:
    if x > max_so_far:
        max_so_far = x
    running_max.append(x)
```

## 3.10.16 TYPE_CHECKING

Proměnná `TYPE_CHECKING` je nastavena na `False` při runtime a na `True` v průběhu ověřování typu (type-checking), umožňujíc zabránit při runtmie vyhodnocení importů `typing` a definicí `TypeVar`. Při zahrnování vašich importů `typing` do bloku `if TYPE_CHECKING:`, můžete dokonce použít modul `typing` i v těch verzích Pythonu, které jej nativně nepodporují.

### Python Docs

Speciální konstanta, u níž aplikace třetích stran pro ověřování typu předpokládají hodnotu `True`. Při runtime má hodotu `False`. Použití:

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: expensive_mod.SomeType) -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

#### Příklad

**Coconut:**

```
if TYPE_CHECKING:
    from typing import List
x: List[str] = ["a", "b"]
```

**Python:**

```
try:
    from typing import TYPE_CHECKING
except ImportError:
    TYPE_CHECKING = False

if TYPE_CHECKING:
    from typing import List
x: List[str] = ["a", "b"]
```

### 3.10.17 `recursive_iterator`

Coconut poskytuje dekorátor `recursive_iterator`, který poskytuje výraznou optimalizaci pro každou bezstavovou (stateless) rekurzivní funkci, která vrací iterátor. Pro použití `recursive_iterator` u funkce musí být splněna následující kritéria:

1. vaše funkce buď vždy `vrací` iterátor nebo generuje iterátor pomocí `yield`,
2. při opakovaném volání pro tytéž argumenty produkuje vaše funkce tentýž iterátor (vaše funkce je bezestavová),
3. vaše funkce je volána (obvykle volá samu sebe) několikrát pro tytéž argumenty.

Setkáte-li se s `RuntimeError` následkem maximální hloubky rekurze, je vhodné přepsat funkci tak, aby vyhověla buď výše uvedenému požadavku na `recursive_iterator`nebo odpovídajícím kritériím pro *optimalizaci koncového volání*, jež obojí by mělo takovým chybám zabránit.

Nadto, `recursive_iterator` také umožňuje řešení of nasty segmentation fault in Python's iterator logic that has never been fixed. Konkrétně, místo zápisu

```
seq = get_elem() :: seq
```

které havaruje v důsledku výše uvedeného problému, pište

```
@recursive_iterator
def seq() = get_elem() :: seq()
```

které poběží uspokojivě.

#### Příklad

**Coconut**

```
@recursive_iterator
def fib() = (1, 2) :: map((+), fib(), fib()$[1:])
```

**Python** *Can't be done without a long decorator definition. The full definition of the decorator in Python can be found in the Coconut header.*

### 3.10.18 `parallel map`

Coconut poskytuje paralelní verzi `map` pod názvem `parallel_map`. `parallel_map` využívá více procesů a je proto mnohem rychlejší než `map` pro úlohy, svázané s CPU.

Použití `parallel_map` vyžaduje `concurrent.futures`, jež existují ve standardní knihovně Python 3, avšak v Python 2 bude zapotřebí provést `pip install futures`.

Protože `parallel_map` používá ke svému provedení více procesů, je nezbytné aby všechny její argumenty byly serializovatelné. Serializovatelné (pickleable) jsou pouze objekty definované na úrovni modulu, uvnitř funkce nebo uvnitř interpreta. Navíc, ve Windows je nezbytné aby se všechna volání `parallel_map` vyskytla uvnitř dozoru `if __name__ == "__main__"`.

#### Python Docs

**parallel_map**(*func, *iterables*)

Ekvivalentní k `map(func, *iterables)` až nato, že *func* je provedena asynchronně a několik volání *func* může být provedeno současně. Vyvolá-li volání výjimku, je tato výjimka zvednuta při vyzvedávání jeho hodnoty z iterátoru.

#### Příklad

**Coconut**

```
parallel_map(pow$(2), range(100)) |> list |> print
```

**Python**

```
import functools
import concurrent.futures
with concurrent.futures.ProcessPoolExecutor() as executor:
    print(list(executor.map(functools.partial(pow, 2), range(100))))
```

### 3.10.19 `concurrent_map`

Coconut poskytuje concurentní verzi `map` pod názvem `concurrent_map`. `concurrent_map` využívá více vláken a je proto mnohem rychlejší než `map` u úloh související s IO. Použití `concurrent_map` vyžaduje `concurrent.futures`, jež existuje ve standardní knihovně Python 3, avšak v Python 2 bude zapotřebí provést `pip install futures`.

#### Python Docs

**concurrent_map**(*func, *iterables*)

Ekvivalentní k `map(func, *iterables)` až nato, že *func* je provedena asynchronně a několik volání *func* může být provedeno současně. Vyvolá-li volání výjimku, je tato výjimka zvednuta při vyzvedávání jeho hodnoty z iterátoru.

#### Příklad

**Coconut**

```
concurrent_map(get_data_for_user, get_all_users()) |> list |> print
```

**Python**

```
import functools
import concurrent.futures
with concurrent.futures.ThreadPoolExecutor() as executor:
    print(list(executor.map(get_data_for_user, get_all_users())))
```

### 3.10.20 `MatchError`

Objekt `MatchError` je vyvolán, když selže *destructuring assignment*, načež je `MatchError` poskytnut jako vestavěná procedura pro odchycení takovýchto chyb. Objekty `MatchError` podporují dva atributy, `pattern`, což je řetězec, popisující selhávající předlohu a `value`, což je objekt, který selhal při porovnávání s předlohou.

## 3.11 Moduly Coconut

### 3.11.1 Automatická kompilace

Pokud nemáte speciení požadavky ohledně parametrů kompilace, můžete použít automatickou kompilaci, která se vám postará o všechno potřebné. Pokud před čímkoli ostatním importujete *`coconut.convenience`*, Coconut prověří každý váš import a zjistí-li, že importujete soubor `~.coco`, automaticky jej kompiluje. Pamatujte, že aby Coconut věděl, jaký soubor importujete, musí být dostupný via `sys.path`, stejně jako normální import.

Automatická kompilace vždycky kompiluje moduly a pakety v místě a vždy používá `--target sys`. Automatická kompilace je vždy dostupná v interpretu Coconut, stejně jako vestavěná funkce `reload` pro snadné znovunačtení importovaných modulů.

### 3.11.2 `coconut.convenience`

Kromě automatické kompilace lze také použít volání kompilátoru z kódu místo z příkazového řádku. Specifikace různých 'convenience functions' je uvedena níže.

#### `parse`

**coconut.convenience.parse**([*code,* [*mode*]])

Patrně nejužitečnější z 'convenience functions' je funkce `parse`, která jako vstup přijímá kód Coconut a vrací ekvivalentní kompilovaný kód v Pythonu. Druhý argument *mode* se používá k označení kontextu pro parsování.

Pokud není *code* zadán, vrací `parse` pouze záhlaví argumentu *mode*, jež může být vyhodnoceno pro nastavení prostředí, v němž může být budoucí kód parsován a vyhodnocen bez záhlaví.

Každý argument *mode* má dvě komponenty: jaký používá parser a jaké jaké záhlaví připojuje. Parser určuje přípustný kód a záhlaví (header) určuje jak má být kompilovaný kód použit. Možné hodnoty *mode* jsou:

- `"sys"`: (the default)
  - parser: file
    - The file parser can parse any Coconut code.
  - header: sys
    - This header imports *`coconut.__coconut__`* to access the necessary Coconut objects.
- `"exec"`:
  - parser: file
  - header: exec
    - When passed to `exec` at the global level, this header will create all the necessary Coconut objects itself instead of importing them.
- `"file"`:
  - parser: file
  - header: file
    - This header is meant to be written to a `--standalone` file and should not be passed to `exec`.
- `"package"`:
  - parser: file
  - header: package
    - This header is meant to be written to a `--package` file and should not be passed to `exec`.
- `"block"`:
  - parser: file
  - header: none
    - No header is included, thus this can only be passed to `exec` if code with a header has already been executed at the global level.
- `"single"`:
  - parser: single
    - Can only parse one line of Coconut code.
  - header: none
- `"eval"`:
  - parser: eval
    - Can only parse a Coconut expression, not a statement.
  - header: none
- `"debug"`:
  - parser: debug

        * Can parse any Coconut code, allows leading whitespace, and has no trailing newline.
    - header: none

### Příklad

```
from coconut.convenience import parse
exec(parse())
while True:
    exec(parse(input(), mode="block"))
```

#### setup

**coconut.convenience.setup**(*target, strict, minify, line_numbers, keep_lines, no_tco*)

`setup` lze použít pro zadání flagů příkazového řádku, použitých funkcí `parse`. Možné hodnoty flagů jsou:

- *target*: `None` (default), or any *allowable target*
- *strict*: `False` (default) or `True`
- *minify*: `False` (default) or `True`
- *line_numbers*: `False` (default) or `True`
- *keep_lines*: `False` (default) or `True`
- *no_tco*: `False` (default) or `True`

#### cmd

**coconut.convenience.cmd**(*args*, [*interact*])

Vyhodnotí dané *args* jakoby byly zadané interpretu `coconut` z příkazového řádku. Pokud však *interact* má hodnotu True nebo je zadáno `-i`, interpret se nespustí. Navíc, protože `parse` a `cmd` sdílejí tentýž 'convenience parsing' objekt, všechny změny provedené při parsování přes `cmd` budou působit jako by byly zavedené přes `setup`.

#### version

**coconut.convenience.version**([*which*])

Přináší řetězec, obsahující informaci o verzi Coconutu. Volitelný argument *which* je typ verze požadované informace. Možné hodnoty argumentu *which* jsou:

- `"num"`: the numerical version (the default)
- `"name"`: the version codename
- `"spec"`: the numerical version with the codename attached
- `"tag"`: the version tag used in GitHub and documentation URLs
- `"-v"`: the full string printed by `coconut -v`

#### auto_compilation

**coconut.convenience.auto_compilation**([*on*])

Zapíná či vypíná *automatickou compilaci* (implicitně je zapnuta). Tato funkce je volána automaticky při importu `coconut.convenience`.

#### CoconutException

Vyskytne-li se při vyhodnocení 'convenience function' chyba, vyvolá se instance `CoconutException`. Pomocná funkce `coconut.convenience.CoconutException` slouží k odchycení těchto chyb.

### 3.11.3 coconut.__coconut__

Toto je občas užitečné pro přístup k vestavěným objektům Coconutu z čistého Pythonu. Za tím účelem Coconut poskytuje `coconut.__coconut__`, jenž se chová přesně jako hlavičkový soubor `__coconut__.py`, připojený když je Coconut kompilován v režimu 'package'.

Všechny nativní objekty Coconutu jsou přístupné z `coconut.__coconut__`. Doporučený způsob jejich importu je použití `from coconut.__coconut__ import`.

#### Příklad

**Python**

```
from coconut.__coconut__ import parallel_map
```

Coconut (coconut-lang.org) je varianta jazyka Python, která k syntaxi Pythonu **přidává** nové vlastnosti pro jednoduché a elegantní Pythonické **funkcionální programování**.

Coconut je vyvíjen na GitHub a hostován na PyPI.

KAPITOLA 4

# Instalace

Instalace Coconut je stejně snadná jako otevření konzoly s promptem a zadání:

```
pip install coconut
```

načež máte celý svět Coconut u svých nohou.

KAPITOLA 5

# Ukázky kódu

**Usměrnění programu** (pipeline-style programming):

```
"hello, world!" |> print
```

**Pohlednější lambda**:

```
(x) -> x ** 2
```

**Částečná aplikace** (partial application):

```
range(10) |> map$((x) -> x ** 2) |> list
```

**Porovnání předlohy** (pattern-matching):

```
match [head] + tail in [0, 1, 2, 3]:
    print(head, tail)
```

**Rozložené přiřazení** (destructuring assignment):

```
{"list": [0] + rest} = {"list": [0, 1, 2, 3]}
```

**Infixová notace**:

```
5 `mod` 3 == 2
```

**Operátorové funkce**:

```
range(15) |> map$((*)$(2)) |> list
```

**Kompozice funkcí**:

```
(f .. g .. h)(x, y, z)
```

**Líné seznamy**:

```
(| first_elem() |) :: rest_elems()
```

**Paralelní programování**:

```
range(100) |> parallel_map$((**)$(2)) |> list
```

**Optimalizace koncové rekurze**:

```
def factorial(n, acc=1):
    case n:
        match 0:
            return acc
        match _ is int if n > 0:
            return factorial(n-1, acc*n)
    else:
        raise TypeError("the argument to factorial must be an integer >= 0")
```

**Algebraické datové typy**:

```
data Empty()
data Leaf(n)
data Node(l, r)

def size(Empty()) = 0

@addpattern(size)
def size(Leaf(n)) = 1

@addpattern(size)
def size(Node(l, r)) = size(l) + size(r)

size(Node(Empty(), Leaf(10))) == 1
```

KAPITOLA 6

# Užitečné odkazy

Podporu pro své první kroky v prostředí Coconut naleznete na těchto odkazech:

- Coconut
- Tutoriál: Dobrým výchozím bodem pro začátečníka v Coconut je jeho tutorial s případovými studiemi.
- Dokumentace: Hledáte-li informaci o konkretní entitě, zkuste referenční dokumentaci jazyka Coconut.
- FAQ: Chcete-li se zeptat, pro koho je Coconut určen a zda byste jej měl používat, navštivte Frequently Asked Questions .
- Create a New Issue: If you're having a problem with Coconut, creating a new issue detailing the problem will allow it to be addressed as soon as possible.
- Gitter: For any questions, concerns, or comments about anything Coconut-related, ask around at Coconut's Gitter, a GitHub-integrated chat room for Coconut developers.

Poznámka: Pokud výše uvedené linky nechodí, zkuste mirror.