Coconut

Vydání v1.2.2 [Colonel]

bře 01, 2017

Obsah

KAPITOLA 1

Dokumentace

Úvod

Tato dokumentace pokrývá všechny technické detaily programovacího jazyka Coconut a je zamýšlana spíš jako referenční příručka než edukativní 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 pomocí konzoly příkazového řádku, jež rovněž obsahuje překladač pro kompilaci v reálném čase. Kromě této konzoly podporuje Coconut také notebooky IPythonu a Jupiteru.

Zatímco většina v Coconut má svoji inspiraci jednoduše ve snaze učinit programování v Pythonu funkcionální, další inspirace pochází z Haskellu, CoffeeScriptu, F# a z extenze Pythonu patterns.py.

Kompilace

Instalace

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. Coconut sám má několik závislostí (dependencies), které lze instalovat zápisem

```
pip install coconut[all]
```

což umožní používání flagů `--jobs`, `--watch` a `--jupyter`. Pro instalaci pouze vybraného flagu, napište místo `all` název příslušného flagu.

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

```
pip install coconut-develop
```

což nainstaluje nejposlednější chodící development build (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.

Použití

```
coconut [-h] [-v] [source] [dest] [-t version] [-s] [-l] [-k] [-p] [-a] [-w] [-d] [-
↪r] [-n] [-m] [-i] [-q] [-f] [-c code] [-j processes] [--jupyter ...] [--tutorial] [-
↪-documentation] [--style name] [--recursion-limit limit] [--verbose]
```

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, --target             specify target Python version (defaults to universal)
-s, --strict             enforce code cleanliness standards
-l, --line-numbers       add line number comments for ease of debugging
-k, --keep-lines         include source code in comments for ease of debugging
-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)
-w, --watch            watch a directory and recompile on changes (requires watchdog)
-d, --display            print compiled Python
-r, --run                run compiled Python (often used with --nowrite)
-n, --nowrite            disable writing compiled Python
-m, --minify             compress compiled Python
-i, --interact           force the interpreter to start (otherwise starts if no other␣
↪command is given)
-q, --quiet              suppress all informational output (combine with --display to␣
↪write runnable code to stdout)
```

```
-f, --force             force overwriting of compiled Python (otherwise only␣
↪overwrites when source code or compilation parameters change)
-c, --code code         run a line of Coconut passed in as a string (can also be␣
↪passed into stdin)
-j, --jobs processes    number of additional processes to use (defaults to 0) (pass
↪'sys' to use machine default)
--jupyter, --ipython    run Jupyter/IPython with Coconut as the kernel (remaining␣
↪args passed to Jupyter)
--tutorial              open the Coconut tutorial in the default web browser
--documentation         open the Coconut documentation in the default web browser
--style name            pygments syntax highlighting style (or 'none' to disable)
--recursion-limit       set maximum recursion depth in compiler (defaults to 2000)
--verbose               print verbose debug output
```

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`.

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`).

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` a PyPy `2.7`, `3.2`.

V rámci snahy o vzájemnou kompabilitu (cross-compatibility), přidává Coconut nové Python 3 built-ins přepisuje Python 2 built-ins na Python 3 verze tam, kde je to možné. Navíc Coconut přepisuje některé Python 3 built-ins z optimalizačních důvodů. Je-li požadován přístup k verzím Pythonu, lze staré built-ins vydolovat s použitím předložky `py_`. Dostupné Python built-ins available jsou:

- `py_chr`
- `py_filter`
- `py_hex`
- `py_input`

- `py_raw_input`
- `py_int`
- `py_oct`
- `py_open`
- `py_print`
- `py_range`
- `py_xrange`
- `py_str`
- `py_map`
- `py_zip`

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),
- function type annotation,
- 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 (requires `--target 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 níže) 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`, `26` (will work on any Python `>= 2.6` but `< 3`),
- `27` (will work on any Python `>= 2.7` but `< 3`),
- `3`, `32` (will work on any Python `>= 3.2`),
- `33`, `34` (will work on any Python `>= 3.3`),
- `35` (will work on any Python `>= 3.5`),
- `36` (will work on any Python `>= 3.6`),
- `sys` (chooses the specific target corresponding to the current version).

Poznámka: Čárky jsou ve specifikacích cíle ignorovány, takže cíl `2.7` je ekvivalentní cíli `27`.

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),
- missing new line at end of file (without `--strict` will show a Warning),
- use of `from __future__` imports (without `--strict` will show a Warning)
- trailing whitespace at end of lines,
- semicolons at end of lines,
- use of the Python-style `lambda` statement,
- use of `u` to denote Unicode strings, and
- use of backslash continuations (use *parenthetical continuation* instead).

Doporučuje se při práci na novém projektu používat flag `--strict` (nebo `-s`) protože vám bude nápomocen při psaní čistšího kódu.

Podpora pro IPython Jupyter

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

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.

Jádro

Je-li Coconut použit jako jádro (kernel), bude veškerý kód v konzoli nebo notebooku poslán k vyhodnocení do Coconut místo do Pythonu. 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.

Operátory

Lambda

Coconut poskytuje jednoduchý, čistý operátor `->` jako alternativu k příkazu `lambda` v Pythonu. Skladba s operátorem `->` je `(arguments) -> expression`. Operátor má stejné pořadí důležitosti jako starý příkaz, což znamená, že bude často nezbytné uzavřít lambdu do závorek.

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 i když je vyžadován 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))
```

Částečná aplikace

K označení částečné aplikace používá Coconut znak `$` mezi názvem funkce a závorkou před argumenty. It has the same precedence as subscription.

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

Python

```
import functools
expnums = map(functools.partial(pow, 2), range(5))
print(list(expnums))
```

Vedení pipeline

Coconut používá vodící (pipe) operátory 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
```

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)))
```

Skladba

Coconut používá operátor `..` pro skládání funkcí. It has a precedence in-between subscription and exponentiation. The in-place operator is `..=`.

Example

Coconut

```
fog = f..g
```

Python

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

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 vyčerpá prvky z prvního a poté z druhého iteráblu (iterovatelného objektu). 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.

Krájení 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. It has the same precedence as subscription.

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*).

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.

Alternativy Unicode

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

Full List

```
→ (\u2192)                  => "->"
 (\u21a6)                  => "|>"
* (*\u21a6)                => "|*>"
 (\u21a4)                  => "<|"
* (\u21a4*)                => "<*|"
 (\u22c5)                  => "*"
↑ (\u2191)                  => "**"
÷ (\xf7)                    => "/"
÷/ (\xf7/)                  => "//"
 (\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)                  => "..."
× (\xd7)                    => "@" (only matrix multiplication)
```

Klíčová slova

data

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.

Bloky typu `data` vytvářejí v Coconut neměnitelné třídy pomocí parametru `__slots__` a odvozené z `collections.namedtuple`. Skladba deklarace datového bloku vypadá takto:

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

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

Subtřídy datových typů lze snadno vytvořit děděním do normální `třídy` Pythonu. Aby se stala nová subtřída neměnitelná, je nutné do ní vložit řádek

```
__slots__ = ()
```

před definicemi metod nebo atributů.

Zdůvodnění

Hlavní část funkcionálního programování, které Coconut v Pythonu zlepšuje, je použití hodnot nebo neměnitelných datových typů. Neměnitelná data mohou být 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 subtřídu nové entice (tuple). Nová subtřída je použita k vytvoření entici podobným objektům, jejichž pole jsou přístupná přes vzhled (lookup) atributu, jakož i proto, že 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 vector(x, y):
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

v = vector(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
```

Demonstruje skladbu, vlastnosti a neměnitelnou povahu typů `data`.

```
data Empty(): pass
data Leaf(n): pass
data Node(l, r): pass
Tree = (Empty, Leaf, Node)

def size(Tree()) = 0

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

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

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

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

Python

```
import collections
class vector(collections.namedtuple("vector", "x, y")):
    __slots__ = ()
    def __abs__(self):
        return (self.x**2 + self.y**2)**.5

v = vector(3, 4)
```

```
print(v)
print(abs(v))
v.x = 2
```

```
import collections
class Empty(collections.namedtuple("Empty", "")):
    __slots__ = ()
class Leaf(collections.namedtuple("Leaf", "n")):
    __slots__ = ()
class Node(collections.namedtuple("Node", "l, r")):
    __slots__ = ()

def size(tree):
    if isinstance(tree, Empty):
        return 0
    elif isinstance(tree, Leaf):
        return 1
    elif isinstance(tree, Node):
        return size(tree[0]) + size(tree[1])
    else:
        raise MatchError()

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

match

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

Úvod

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
    | "(" 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
      (")" | "]")                           # must both be parens or brackets
    | (                                 # 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
    | pattern "is" exprs                # type-checking
    | pattern "and" pattern             # match all
    | pattern "or" pattern              # match any
    )
```

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>`.
- Dicts (`{<pairs>}`): spáruje pouze mapping (`collections.abc.Mapping`) stejné délky a porovná obsah vůči `<pairs>`.
- 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.

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

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(): pass
data Leaf(n): pass
data Node(l, r): pass
Tree = (Empty, Leaf, Node)

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                                    # 0
Leaf(5) |> depth |> print                                    # 1
Node(Leaf(2), Node(Empty(), Leaf(3))) |> depth |> print      # 3
```

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

```
def duplicate_first(value):
    match [x] + xs as l in value:
        return [x] + l
    else:
        raise TypeError()

[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.*

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.

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)
```

Vyhrazené proměnné

Není povoleno aby název proměnné začínal `_coconut`, protože tyto proměnné jsou vyhrazeny pro kompilátor.

Výrazy

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 -> _`, je 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)
```

Líné seznamy

Coconut podporuje vytváření líných seznamů, 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.

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
```

Example

Coconut

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

Python

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

Literály množiny

Coconut umožňuje předsadit písmeno `s` nebo `f` před deklaraci setu (množiny). 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()
```

Literály imaginárního čísla

Jako doplněk k zápisu imaginárního čísla v Pythonu pomocí literálů `<num>j` nebo `<num>J` přidává Coconut ještě literály `<num>i` nebo `<num>I` pro zlepšení čitelnosti 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))
```

Podtržítkové separátory

Pro snadnější čitelnost umožňuje Coconut použití podtržítka pro optické rozdělení velkého čísla. Kompilátor tato podtržítka ignoruje.

Příklad

Coconut

```
10_000_000.0
```

Python

```
10000000.0
```

Zápis funkce

Optimalizace koncového volání

Coconut provede automatickou optimalizaci koncovým voláním 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`).

Poznámka: Optimalizace koncovým voláním pracuje i pro 1) vzájemnou rekurzi a 2) porovnávací (pattern-matching) funkce, rozdělené do několika definicí s pouožitím `addpattern` *nebo* `prepattern`.

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.

Příklad

Coconut

```
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")
```

Python

Nelze provést bez přepsání funkce.

Operátorové funkce

Coconut používá jednoduchou výrazovou zkratku: uzavření do závorek činí z operátoru funkci. Podobně jako u iterátorových komprehencí, je-li operátorová funkce jediný argument funkce, mohou závorky volání funkce sloužit jako závorky operátorové funkce.

Zdůvodnění

Velmi často prováděným úkonem ve funkcionálním programování je využití funkčních verzí vestavěných operátorů: 'currying them, composing them, and piping them'. Pro usnadnění těchto úkonů poskytuje Coconut zkrácenou syntaxi pro přístup k operátorovým funkcím.

Úplný seznam

```
(|>)        => # pipe forward
(|*>)       => # multi-arg pipe forward
(<|)        => # pipe backward
(<*|)       => # multi-arg pipe backward
(..)        => # function composition
(.)         => (getattr)
(::)        => (itertools.chain) # will not evaluate its arguments lazily
($)         => (functools.partial)
(+)         => (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))))
```

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))
```

Infixové funkce

Coconut umožňuje infixové použití funkce, kde je název funkce umístěn mezi operandy a je obklopen zpětnými apostrofy. Volání se zpětným apostrofem (backtick calling) má prioritu mezi 'chaining and piping'.

Skladba definice infixové funkce je

```
def <arg> `<name>` <arg>:  # asi má být `=` místo `:`
    <body>
```

kde <name> je název funkce, <arg> jsou parametry funkce a <body> je tělo funkce. Obsahuje-li <arg> ?? default ??, musí být parametry uvedeny v závorkách.

Poznámka: Definici infixové funkce lze kombinovat s definicí přířazovací a/nebo porovnávací (pattern-matching) funkce.

Zdůvodnění

Infixové funkce jsou ve funkcionálním programování obvyklé.

Příklad

Coconut

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

Python

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

Porovnávací funkce

Coconut podporuje porovnávání s předlohou (pattern-matching)/rozložené přiřazení (destructuring assignment) uvnitř definice funkce. Složení této skladby je

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

Kde `<name>` je název funkce, `<cond>` je nepovinná dodatečná kontrola, `<body>` je tělo funkce a `<pattern>` je definován *příkazem* `match`. Klíčové slovo `match` na začátku je nepovinné ale je někdy nezbytné pro odlišení definice porovnávací funkce od normální definice funkce, která má vždy přednost. Definice porovnávací (p-m) fukce je ekvivalentní *příkazu* `match`, který vypadá jako:

```
def <name>(*args):
    match (<pattern>, <pattern>, ...) in args:
        <body>
    else:
        err = MatchError(<error message>)
        err.pattern = "def <name>(<pattern>, <pattern>, ...):"
        err.value = args
        raise err
```

Selže-li průběh porovnávací funkce, vyvolá `MatchError`, stejně jako *rozložené přiřazení*.

Poznámka: 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.

Příkazy

Rozložené přiřazení

Coconut podporuje výrazně zlepšené rozložené přiřazení (destructuring assignment), podobné rozkládání entice/seznamu v Pythonu. Skladba rozloženého 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 rozloženého 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 rozloženého přiřazení. Viz kompilovaný kód pro skladbu Pythonu.

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
```

Příkazy `else`

Coconut podporuje složené příkazy `try`, `if` a `match` na konci příkazu `else` jako u každého jiného jednoduchého příkazu. To je nejvíce užitečné pro společné používání příkazů `match` a `if` a také umožňuje vytváření složených příkazů `try`.

Příklad

Coconut

```
try:
    unsafe_1()
except MyError:
    handle_1()
else: try:
    unsafe_2()
except MyError:
    handle_2()
```

Python

```
try:
    unsafe_1()
except MyError:
    handle_1()
else:
    try:
        unsafe_2()
    except MyError:
        handle_2()
```

Příkazy `except`

Má-li být v Pythonu3 podchyceno více výjimek najednou, musejí být vloženy do závorek aby se v Pythonu2 zabránilo použití čárky místo `as`. Coconut umožňuje použití čárek ve výjimkových příkazech pro odchycení vícerých výjímek bez závorek.

Příklad

Coconut

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

Python

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

Implicitní `pass`

Coconut umožňuje zjednodušený zápis `class name(base)` a `data name(args)` místo `class name(base): pass` a `data name(args): pass`.

Příklad

Coconut

```
data Empty
data Leaf(item)
data Node(left, right)
```

Python

```
import collections

class Empty(collections.namedtuple("Empty", "")):
    __slots__ = ()
class Leaf(collections.namedtuple("Leaf", "n")):
    __slots__ = ()
class Node(collections.namedtuple("Node", "l, r")):
    __slots__ = ()
```

Pokračování v závorkách

Coconut umožňuje u příkazů `del`, `global`, `nonlocal` a `with` rozložení zápisu na více řádků s použitím závorek místo zpětných lomítek \ jako u Pythonu.

Příklad

Coconut

```
global (really_long_global_variable_name_the_first_one,
        really_long_global_variable_name_the_second_one)
```

Python

```
global really_long_global_variable_name_the_first_one, \
       really_long_global_variable_name_the_second_one
```

Zjednodušené určení `global` a `nonlocal`

Coconut umožňuje deklaraci `global` či `nonlocal` v jednom řádku bez opakování názvu proměnné.

Příklad

Coconut

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

Python

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

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 `\\`.

Příklad

Coconut

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

Python

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

Vestavěné funkce

`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 ekviva-

lentní 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
```

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

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. Je zhruba ekvivalentní k:

```
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
```

Příklad

Coconut

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

@prepattern(factorial)
def factorial(0) = 1
```

Python

Nelze provést bez komplikované definice dekorátoru a dlouhé řady kontrol pro každé porovnávání.

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

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

Python

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

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)
```

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)
```

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 používán; otherwise, the *iterable* could get advanced without the tee objects being informed.

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)
```

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)
```

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 vráti 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.

`map` a `zip`

Objekty `map` a `zip` v Coconut jsou vylepšené ekvivalenty Pythonu, které podporují optimalizované normální (a iterátorové) členění, postupy `reversed`, `len`, `repr` a mají přidané atributy, jež mohou použít subtřídy k přístupu k původním argumentům objektu (`map` podporuje `_func` a atributy `_iters` a `zip` podporuje atributy `_iters`).

Příklad

Coconut

```
map((+), range(5), range(6)) |> len |> print
```

Python

Nelze provést bez definování uživatelského typu `map`. Úplnou definici `map` lze nalézt v záhlaví Coconut.

`datamaker`

Coconut poskytuje funkci `datamaker` pro přímý přístup k bázovému konstruktoru datových typů, vytvořenému příkazem `data`. Toto je zejména užitečné při psaní alternativních konstruktorů pro datové typy přepsáním metody `__new__`. Ekvivalentní k:

```
def datamaker(data_type):
    """Returns base data constructor of data_type."""
    return super(data_type, data_type).__new__$(data_type)
```

Příklad

Coconut

```
data trilen(h):
    def __new__(cls, a, b):
        return (a**2 + b**2)**0.5 |> datamaker(cls)
```

Python

```
import collections
class trilen(collections.namedtuple("trilen", "h")):
    __slots__ = ()
    def __new__(cls, a, b):
        return super(cls, cls).__new__(cls, (a**2 + b**2)**0.5)
```

`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 volá samu sebe pro tytéž argumenty,
4. všechny argumenty, zadávané funkci jsou serializovatelné (pickleable).

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

Nelze provést bez dlouhé definice dekorátoru.

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))))
```

concurrent map

Coconut poskytuje concurrentní 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())))
```

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.

Utilita Coconut

Zvýraznění skladby

Současné možnosti pro zvýraznění skladby v Coconut jsou tyto:

1. use **SublimeText** (instructions below),
2. use an editor that supports **Pygments** (instructions below),
3. use `coconut.vim`, a third-party **Vim** highlighter,
4. use `coconut-mode`, a third-party **Emacs** highlighter, or
5. just treat Coconut as Python.

Pokyny pro nastavení skladby zvýrazňování pro SublimeText a Pygments jsou uvedeny níže. Pokud některý z výše uvedených zvýrazňovačů nechodí, potom by mělo stačit nastavit editor tak, aby interpretoval všechny soubory `.coco` (také `.coc` a `.coconut`, byť `.coco` je preferovaná přípona) jako kód Pythonu, neboť se tak dostatečně zvýrazní většina vašeho kódu.

SublimeText

Zvýrazňování skladby Coconut v editoru SublimeText vyžaduje aby byl instalován standardní správce paketů Package Control. Pokud tomu tak je, potom:

1. otevřte příkazovou paletu SublimeTextu stisknutím `Ctrl+Shift+P`,
2. potvrďte a zvolte `Package Control: Install Package`,
3. potvrďte a zvolte `Coconut`.

Abyste se přesvědčili, že všechno chodí jak má, otevřte soubor `.coco` file a ujistěte se, že se `Coconut` objeví v pravém dolním rohu. Objeví-li se něco jiného, jako třeba `Plain Text`, klikněte na to, zvolte `Open all with current extension as...` a potom vyberte `Coconut`.

Pygments

Tentýž příkaz `pip install coconut`, který instaluje interaktivní utilitu Coconut, instaluje také `coconut` lexer aplikace Pygments. Jak tento lexer použít záleží na použité 'Pygments-enabled' aplikaci, ale obecně se zvolí `coconut` jako zvýrazňovaný jazyk a/nebo použije se platná extenze souboru Coconut (`.coco`, `.coc` nebo `.coconut`) a Pygments by se v tom měl vyznat. Tato dokumentace je například generována pomocí Sphinx. Zvýraznění, které vidíme, bylo vytvořeno přidáním řádku

```
highlight_language = "coconut"
```

v souboru `conf.py` Coconutu.

`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`.

Example

Python

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

coconut.convenience

Někdy je užitečné mít možnost použít kompilátor Coconutu z kódu místo z příkazového řádku. Doporučuje se použít `from coconut.convenience import` a importovat potřebnou užitečnou (convenience) funkci. Specifikace různých 'convenience' funkcí jsou uvedeny dále.

parse

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

Patrně nejužitečnější z 'výhodných' funkcí je `parse`, která přijme kód Coconut a vrací ekvivalentní kompilovaný kód Pythonu. Druhý argument, *mode*, se použije k indikaci kontextu pro parsing.

Každý *mode* má dvě komponenty: jaký parser používá a jaké záhlaví předesílá (prepends). Parser určuje, jaký kód Coconutu je přípustný jako vstup a záhlaví určuje, jak může být kompilovaný Python použit. Možné hodnoty *mode* jsou:

- `"exec"`: (the default)
  - parser: file The file parser can parse any Coconut code.
  - header: exec When passed to `exec` at the global level, this header will create all the necessary Coconut objects.
- `"file"`:
  - parser: file
  - header: file This header is meant to be written to a `--standalone` file and should not be passed to `exec`.
- `"module"`:
  - parser: file
  - header: module 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 the exec 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 and allows leading whitespace.
  - header: none

`setup`

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

`setup` lze použít k zadání flagů příkazového řádku, použitých v akci `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`

`cmd`

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

Zpracuje dané *args*, jakoby byly zadány z příkazového řádku, s tou výjimkou, že pokud *interact* není `true` nebo nebylo-li zadáno `-i`, interpret se nespustí. Navíc, protože `parse` a `cmd` sdílejí tentýž 'convenience parsing' objekt, jakékoli změny pro parsing zadané přes `cmd`, budou pracovat stejně, jakoby byly zavedeny přes `setup`.

`version`

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

Vyhledá řetězec obsahující informaci o verzi Coconut. Nepovinný argument *which* upřesňuje požadovanou verzi informace. Možné hodnoty *which* jsou:

- `"num"`: číselná verze (implicitní)
- `"name"`: kódové označení verze
- `"spec"`: číselná verze s připojeným kódovým označením
- `"tag"`: tag verze, použitý v GitHub a v URL dokumentace
- `"-v"`: výstup příkazu `coconut -v` (úplný řetězec)

`CoconutException`

Je-li v 'convenience' funkci detekována chyba, je aktivováno hlášení `CoconutException`. `coconut.convenience.CoconutException` umožňuje odchycení takových chyb.

KAPITOLA 2

Tutoriál

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 - porovnávání předlohy
- algebraic data types - ADT
- destructuring assignment - rozložené přiřazení
- partial application - částečnou aplikaci
- lazy lists - líné seznamy
- function composition - skládání funkcí
- prettier lambdas - úhlednější lambdy
- infix notation
- pipeline-style programming - směrované programování
- operator functions - operátorové funkce
- tail recursion optimization - optimalizace koncové rekurze
- parallel programming - paralelní programování

a mnoho dalšího!

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 Coconute 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 či Jupiter.

Instalace Coconut je velmi jednoduchá:

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

```
pip install coconut
```

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

```
coconut -h
```

což by mělo zobrazit nápovědu pro Coconut.

Začínáme

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
```

Použití kompilátoru

Ovšemže, být schopen za běhu interpretovat kód Coconut je velká věc ale bez schopnosti psát a kompilovat programy by naše programování nebylo příliš užitečné. Pojďme si proto napsat první program v Coconut: “Hello, world!”.

Nejprve vytvoříte soubor, do něhož náš kód vložíte. Doporučená extenze pro zdrojové soubory Coconut je `.coco`, vytvořte tedy soubor s názvem `hello_world.coco`. Poté, co to uděláte, měli byste nastavit svůj textový editor na správné zvýrazňování zdrojového kódu. Příslušné instrukce naleznete v odstavci Zvýraznění skladby. Dokumentace

Nyní vložíme kód do souboru `hello_world.coco`. Na rozdíl od Pythonu, kde záhlaví a různé importy jsou obvyklé a velmi často velmi nezbytné,

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

provede kompilátor Coconut potřebné importy automaticky, takže jediné o co se musíte starat, je vlastní kód.

V čistém Pythonu 3 má příkaz k tisku formát:

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

a stejně tak i v Coconut, kde navíc je možné použít potrubní (pipeline) usměrnění v zápisu:

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

z něhož je zřetelně vidět, jak operátor `|>` způsobí předání řetězce coby argument následné funkci, jíž je v tomto případě příkaz k tisku. Nyní náž jednoduchý program "hello_world" uložíme a zkusíme jej spustit.

Kompilování souborů a projektů utilitou Coconut je velmi jednoduché. Zapíšeme pouze

```
coconut hello_world.coco
```

což vytvoří výstup

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

Soubor `hello_world.py` uložte do stejného adresáře jako `hello_world.coco` a měl byste být schopen spustit soubor příkazem

```
python hello_world.py
```

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

Kompilování jednotlivých souborů ovšem není jediný způsob použití kompilátoru Coconut. Můžeme také kompilovat všechny soubory v daném adresáři najednou a to pouhým uvedením názvu adresáře.

```
coconut `název_adresáře`
```

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

Kompilátor Coconut 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.

Použití IPython / Jupyter

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

To zahrnuje podporu aplikace IPython (jádro Pythonu pro framework Jupyter) místo klasické konzoly Pythonu. Coconut je použit jak jako jádro pro notebooky a konzoly Jupytera, tak jako rozšíření uvnitř jádra IPythonu.

Pro spuštění notebooku Jupytera s Coconut jako jádrem použijete příkaz

```
coconut --jupyter notebook
```

a pro spuštění konzoly Jupytera použijete příkaz

```
coconut --jupyter console
```

nebo lze ekvivalentně v obou příkazech zaměnit `--ipython` za `--jupyter`.

Pro použití Coconut jako extenzi uvnitř jádra IPythonu zapište

```
%load_ext coconut
```

do svého notebooku či konzoly IPythonu a poté spusťte kód Coconut zápisem

```
%coconut <code>
```

nebo

```
%%coconut <command-line-args>
<code>
```

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.

Případová studie 1: `factorial`

V první ukázce 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 vlastnoctí 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 kategorie: imperativní, recurzivní, iterativní a s použitím `addpattern`.

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 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 zabývat průběhem výpočtu, ověřme si nejprve 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 (byť úprava skriptu má také své výhody). Měli bychom dostat `1`, `6` a dvakrát `TypeError`.

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 činí použití infixové notace (vložení funkce `isinstance` mezi argumenty: `n` a `int`) kód čistší a čitelnější.

Rekurzivní metoda

Rekurzivní metoda je první ze zcela funkcionálních přístupů a to v tom, že nezahrnuje změnu stavu ve smyčce jako u imperativního přístupu. Rekurzivní přístup nahrazuje potřebu explicitně měněné proměnné její implicitní změnou v rekurzivním volání funkce:

```
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
```

Překopírujte si kód a testy do překladače.

Proberme si specifika syntaxe v tomto příkladu. První věcí je `case n`. Tento příkaz spouští blok `case`, v němž se mohou vyskytnout pouze příkazy `match`. Každý příkaz `match` se pokouší porovnat svou předlohu s hodnotou bloku `case`. U první úspěšné shody jsou realizována všechna připojení proměnných. Navíc, jak je tomu v tomto případě, mohou mít příkazy `match` také kontrolky (guards) `if`, které deklarují podmínku pro další provedení kódu. Posléze, za blokem `case` je příkaz `else` který se provede jen v případě absence jakékoliv shody.

Konkretně v tomto příkladě ověřuje první příkaz `match`, zda je `n` shodné s `0`. Pakliže ano, provede se `return 1`. Následně druhý příkaz `match` ověřuje, zda se `n` shoduje s `_ is int`, což je adekvátní idiomu `n je instancí int` a zda je `n > 0`. Jsou-li všechny kontroly pozitivní, provede se příkaz `return n * factorial(n-1)`. Nedojde-li k provedení žádného příkazu, přichází ke slovu příkaz `else`, který spustí a provede `raise TypeError("argument faktoriálu musí být celé číslo >= 0")`.

I když je tento příklad velmi prostý, je postup v něm použitý, zvaný **pattern-matching** (porovnání předlohy), jedním z nejmocnějších i nejsložitějších postupů v Coconut. Jako obecné vodítko poslouží asociativní spojení pojmu *přiřazení* s klíčovým slovem `match`.

Svým způsobem ještě složitější je inverzní postup k `pattern matching`, jímž je **destructuring assignment** (rozložené přiřazení), jež v našem případě pro funkci `factorial` má skladbu:

```
def factorial(n):
    """Compute n! where n is an integer >= 0."""
    try:
        0 = n # destructuring assignment
    except MatchError:
        try:
            _ is int = n # also destructuring assignment
        except MatchError:
            pass
        else: if n > 0: # in Coconut, if, match, and try are allowed after else
            return n * factorial(n-1)
    else:
        return 1
    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
```

Ukázku si nejprve překopírujte do překladače! I když toto rozložené přiřazení bude také chodit, je mnohem více neohrabané než příkazy `match`. Tato alternativa nám ale ozřejmí, že příkazy `match` jsou ve skutečnosti nóbl *rozložená přiřazení*, jež jsou ve skutečnosti nóbl *normální přiřazení*. Při použití *rozloženého* místo *normálního* přiřazení, lze před rozložené přiřazení vložit pro zdůraznění klíčové slovo `match`.

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

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 vrací přímo volání jiné funkce, zadrží (optimize away) Coconut další volání. Naši funkci `factorial` tedy přepíšeme pro použití koncového volání:

```
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 (optimize away) koncové rekurzivní volání.

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 a sice:

```
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$(*)`. V tomto zápise jsou použity dva konstrukty Coconut a sice operátorová funkce pro násobení ve tvaru `(*)` a příkaz k částečné aplikaci ve tvaru `$`.

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, byť i Pythonní forma je v Coconut legální. Pokud bychom si však zadali režim `--strict`, vyvolalo by použití lambda z Pythonu chybové hlášení.

Nyní k částečné aplikaci. 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 funkce 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.

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 Coconut 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 dva nové koncepty k prodiskutování: `addpattern` a definici funkce pro porovnání předlohy (pattern-matching).

Definice funkce pro pattern-matching dělá přesně to co říká její označení - porovnává předlohu se všemi zadanými argumenty. Je zde několik věcí, které je nutné si pohlídat. Předně, aby funkce vyvolala `MatchError`, nenalezne-li se žádná shoda. Dále že nejsou přípustné "keyword" argumenty a konečně stejně jako u rozloženého (destructuring) přiřazení, chcete-li být více explicitní u použití definice pro pattern-matching, můžete přidat `match` před `def`.

Dekorátor `addpattern` přijímá jako argument předtím definovanou p-m funkci a umožňuje v následné funkci přidat novou předlohu.

V našem případě je první předlohou hodnota argumentu `n=0` a druhou podmínka, že `n` je celé číslo `>=0`.

Dekorátorem `addpattern` můžeme upravit nejenom imperativní přístup, jak jsme právě provedli, ale i 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! Nevyhovující seance jsou zde označeny opět jako `MatchError`.

Případová studie 2: `quick_sort`

Ve druhé případové studii budeme používat quick sort algorithm. Použijeme dvě verze funkce `quick_sort` - funkci, která přijímá i vrací seznam a funkci, která přijímá i vrací iterátor.

Třídění sekvence

Nejprve `quick_sort` pro seznamy. Použijeme rekurzivní přístup založený na `addpattern`, podobný posledně psané funkci `factorial`. A to proto, že jelikož nebudeme psát `quick_sort` koncově rekurzivním stylem, nemůžeme použít `tail_recursive`, tudíž není důvod psát celou věc jako jednu funkci a mohli bychom stejně dobře použít `addpattern` k redukci identací. Bez dalších okolků, zde je naše implementace `quick_sort` pro seznamy:

```
def quick_sort([]):
    return []

@addpattern(quick_sort)
def quick_sort([head] + tail):
    """Sort the input sequence using the quick sort algorithm."""
    return (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 je pouze jedna nová věc: head-tail pattern-matching. Máme zde předlohu čelo-chvost (`[head] + tail`), která má obecně 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 s jejímž počátkem je porovnáván seznam nebo entice jehož zbytek je vázán k proměnné. V tomto případě používáme head-tail předlohu abychom odstranili čelo, jež můžeme použít jako pivot pro rozštěpení zbytku seznamu.

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 rekurzivně vytvářet lenivý iterátor. Zde je kód:

```
def quick_sort(l):
    """Sort the input iterator, using the quick sort algorithm, and without using any␣
↪data until necessary."""
    match [head] :: tail in l:
        tail, tail_ = tee(tail)
        yield from (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 |> 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, 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 porovnávání pouze jedné předlohy.

Třetím novým konstruktem je vestavěná funkce `tee`. Funkce `tee` řeší problém funkcionálního programování vytvořený použitím Pythonních iterátorů: kdykoliv je prvek iterátoru evokován, je také zároveň ztracen. Funce `tee` rozdělí iterátor na dva (nebo více, je-li zadán volitelný argument) nezávislé iterátory, které oba pro přístup k datům používají týž skrytý iterátor, takže je-li evokován prvek jednoho iterátoru, zůstává zachován ve druhém.

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, and without using any␣
↪data until necessary."""
    match [head] :: tail in l:
        tail, tail_ = tee(tail)
        yield from (quick_sort((x for x in tail if x < head))
            :: (head,)
            :: quick_sort((x for x in tail_ if x >= head))
            )
```

Funkce se nejprve pokouší rozštěpit seznam `l` na počáteční element a zbývající iterátor. Je-li `l` prázdným iterátorem, porovnání selže, poskytujíce prázdný iterátor. 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.

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-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
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.

Konstruktor pro n-Vector

Nyní, když jsme dostali za opasek `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-vector 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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(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! Velkou novou věcí zde je, 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 `datamaker`, která přijímá datový typ, často jako první argument funkce `__new__` a vrací výchozí konstruktor `data`.

V tomto případě konstruktor kontroluje, zda nebylo zadáno nic jiného než další `vector`, v kterémžto případě jej vrací. Jinak vrací výsledek vytvoření entice argumentů a její předání výchozímu konstruktoru, jehož forma je `vector(pts)`; takto přiřazujíc entici k atributu `pts`.

Metody pro n-vector

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

```
    def __abs__(self):
        """Return the magnitude of the vector."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
```

Základním algoritmem zde je 'mapování' (map) mocniny pro každý prvek, jejich celkový součet a druhá odmocnina výsledku. Zápis celého postupu je přehledně zapsán v jednom řádku.

Další metodou je součet vektorů stejné délky, realizovaný součtem jejich komponent. Využijeme k tomu schopnost Coconut provádět porovnávání shody (pattern-matching) či v tomto případě rozložené přiřazení (destructuring assignment) a to takto:

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

Máme zde několik nových konstruktů ale nejvýznamnějším je příkaz k rozloženému přiřazení `vector(other_pts) = other`, 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) = other` bude shodovat pouze s vektorem, přičemž přiřadí atribut `pts` vektoru k proměnné `other_pts`. Nenajde-li se vhodný vektor pro shodu, je evokována výjimka `MatchError`.

Dalším novým konstruktem je zde znak `|*>`, což je korýtkový operátor (zde 'star-pipe') pro více argumentů. Rozdíl mezi `|*>` a `|>` je analogický rozdílu `f(args)` a `f(*args)`.

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

```
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return 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 funkci `(-)` k zavedení negace vektoru, což by mělo negovat každý jeho prvek:

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

Další metodou je rovnost. Zde opět použijeme pattern-matching pro `data` ale tentokrát uvnitř příkazu `match` místo uvnitř rozloženého přiřazení, neboť při selhání shody chceme odezvu `False`, nikoliv chybové hlášení. Zde je kód:

```
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
```

Jediným novým kostruktem zde je použití `=self.pts` v příkazu `match`. Tento konstrukt provádí kontrolu uvnitř pattern-matching, zajišťujíce, že ke shodě dojde pouze tehdy, když `other.pts == self.pts`.

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, definovaný jako součet součinů jednotli-

vých elementů a na násobení vektoru číslem, definované jako násobení všech elementů stejným číslem. 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 korýtkového (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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(cls) # accesses base constructor
    def __abs__(self):
        """Return the magnitude of the vector."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
    def __add__(self, other):
        """Add two vectors together."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((+), self.pts, other_pts) |*> vector
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((-), self.pts, other_pts) |*> vector
    def __neg__(self):
        """Retrieve the negative of the vector."""
        return self.pts |> map$((-)) |*> vector
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
    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."""
        return 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.

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.

Premiovou výzvou u tohoto odstavce bude napsat každou definovanou funkci do jednoho řádku. Nápomocna k tomu bude tak zvaná přiřazovací funkce:

```
def <name>(<args>) = <return value>
```

která je zjednodušením klasického zápisu v Pythonu:

```
def <name>(<args>): return <return value>
```

Maje toto vyjasněno, je čas uvést obecný cíl naší případové studie. Chceme napsat program, který nám umožní vytvářet nekonečná vektorová pole, přes něž můžeme iterovat a s nimiž můžeme operovat. Úlohu si zúžíme na vektory s pozitivními 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 toto pole procházet (iterovat), potřebujeme jej nějakým způsobem linearizovat a nejjednoduším způsobem to učiníme tak, že jej rozdělíme do diagonál, načež můžeme traverzovat po první diagonále,

potom po druhé a tak dále, nějak takto:

```
...

(0,2)<  ...
      \_
(0,1)<  (1,1)<  ...
      \_      \_
(0,0) > (1,0) > (2,0) > ...
```

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)` `nulté` diagonály. Jak jsme si řekli na počátku případové studie, o řešení se pokusíte nejdřív sami s použitím všech nástrojů funkcionálního programování, které Coconut poskytuje. 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.

linearized_plane

Nyní, když jsme vytvořili naše diagonální čáry, 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 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: these tests use $[] notation, which we haven't introduced yet
#  but will introduce later in this case study; for now, just run the
#  tests, and make sure you get the same result as is in the comment
linearized_plane()$[0] |> print # (0, 0)
linearized_plane()$[:3] |> list |> print # [(0, 0), (0, 1), (1, 0)]
```

Nápověda: místo definování funkce jako `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.

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:

```
# You'll need to bring in the vector class from earlier to make these work
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.

Děláte velký pokrok! Než pokročíte dál, srovnejte si řešení se mnou:

```
def vector_field() = linearized_plane() |> map$((xy) -> vector(*xy))
```

Vše, co jsme učinili, bylo to, že jsme mapovali funkci `linearized_plane` přes `vector` s tím, že jsme volali každý element entice jako separátní argument.

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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(cls) # accesses base constructor
    def __abs__(self):
        """Return the magnitude of the vector."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
    def __add__(self, other):
        """Add two vectors together."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((+), self.pts, other_pts) |*> vector
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((-), self.pts, other_pts) |*> vector
    def __neg__(self):
        """Retrieve the negative of the vector."""
        return self.pts |> map$((-)) |*> vector
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
```

```
    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."""
        return self * other

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

# 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.

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.

`__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
```

.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)
```

.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."""
        if len(pts) == 1 and pts[0] `isinstance` vector:
            return pts[0] # vector(v) where v is a vector should return v
        else:
            return pts |> tuple |> datamaker(cls) # accesses base constructor
    def __abs__(self):
        """Return the magnitude of the vector."""
        return self.pts |> map$((x) -> x**2) |> sum |> ((s) -> s**0.5)
    def __add__(self, other):
        """Add two vectors together."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((+), self.pts, other_pts) |*> vector
    def __sub__(self, other):
        """Subtract one vector from another."""
        vector(other_pts) = other
        assert len(other_pts) == len(self.pts)
        return map((-), self.pts, other_pts) |*> vector
    def __neg__(self):
        """Retrieve the negative of the vector."""
        return self.pts |> map$((-)) |*> vector
    def __eq__(self, other):
        """Compare whether two vectors are equal."""
        match vector(=self.pts) in other:
            return True
        else:
            return False
    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."""
        return 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.

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).

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 |)
```

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
```

Skladba funkcí se také zbavuje potřeby mnoha závorek při zřetězeném volání funkcí, jako zde:

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

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]
```

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.

KAPITOLA 3

Frequently Asked Questions

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.

Které verze Pythonu Coconut podporuje?

Coconut podporuje všechny verze Pythonu `>= 2.6` ve větvi `2.x` nebo `>= 3.2` ve větvi `3.x`. Viz kompatibilní verze Pythonu.

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.

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.

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.

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.

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.

Neznám funkcionální programování, mám se přesto pustit do Coconut?

Yes, absolutely! Tutoriál pro Coconut 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.

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

Maybe. 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.

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.

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.

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.

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.

Kdo vytvořil Coconut?

Evan Hubinger is an undergraduate student studying mathematics and computer science at Harvey Mudd College. You can find his resume online at http://evhub.github.io/resume.pdf.

Coconut je varianta jazyka Python vytvořená 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
- Dokumentace: Hledáte-li informaci o konkretní entitě, zkuste referenční dokumentaci jazyka Coconut.
- Tutoriál: Dobrým výchozím bodem pro začátečníka v Coconut je jeho tutorial s případovými studiemi.
- 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.