Bartłomiej Przybylski

Lua i LaTeX

Dynamiczne tworzenie dokumentów

PWN

Lua i LaTeX

Bartłomiej Przybylski

Lua i LaTeX

Dynamiczne tworzenie dokumentów

PWN

Projekt okładki: **Hubert Zacharski**

Wydawca: **Łukasz Łopuszański**

Redaktor prowadzący: **Adam Kowalski**

Redakcja techniczna: **Maria Czekaj**

Korekta: **Agnieszka Jaworska**

Koordynator produkcji: **Anna Bączkowska**

Wydanie tej książki zostało częściowo sfinansowane ze środków Uniwersytetu im. Adama Mickiewicza w Poznaniu.

ISBN 978-83-01-19696-7

Wydanie I
Warszawa 2018

Wydawnictwo Naukowe PWN SA
02-460 Warszawa, ul. Gottlieba Daimlera 2
tel. 22 69 54 321, faks 22 69 54 288
infolinia 801 33 33 88
e-mail: pwn@pwn.com.pl; reklama@pwn.pl
www.pwn.pl

Skład i łamanie: **Bartłomiej Przybylski**

Druk i oprawa: OSDW Azymut Sp. z o.o.

Spis treści

II. LuaLaTeX

III. Praktyczne przykłady

Dodatki

Przedmowa

Kiedy mniej więcej w połowie 1977 r. Donald E. Knuth, amerykański akademik i ceniony specjalista w zakresie matematycznych podstaw informatyki, postanowił stworzyć zupełnie nowe oprogramowanie do cyfrowego składu tekstu, dawał sobie około sześciu miesięcy na ukończenie prac nad nim. I chociaż pierwsza wersja oprogramowania TEX, bo o nim mowa, ujrzała światło dzienne już w 1978 r., to jednak dopiero 11 lat później, w 1989 r., uznano je za ukończone. Od tego momentu są poprawiane w nim jedynie nieliczne napotkane błędy.

Trzydzieści minionych lat wystarczyło, aby TEX stał się nieodłącznym kompanem w codziennej pracy naukowej matematyków, informatyków, fizyków oraz inżynierów. Trudno snuć domysły, ale wiele wskazuje na to, że istotny wpływ na popularność TEXa miało opublikowanie w 1985 r. przez Lesliego Lamporta, innego amerykańskiego informatyka, pierwszego pokaźnego zbioru makr dla języka TEX. Dziś zbiór ten, nazwany LATEX, jest na stałe wkomponowany w niemal wszystkie dystrybucje systemu TEX, znacząco ułatwiając pracę z nawet najbardziej skomplikowanymi dokumentami.

W 1993 r., kilka lat po zakończeniu prac rozwojowych nad programem TEX, opublikowano pierwszą implementację nowego języka programowania o nazwie Lua. Język ten, tworzony przez brazylijski zespół specjalistów z Robertem Ierusalimschem na czele, od początku miał być szybki, przyjazny i łatwy do integracji z istniejącym oprogramowaniem. Z perspektywy czasu można śmiało stwierdzić, że cel ten udało się osiągnąć, a sam język rozwijany jest do dzisiaj. Obecnie jest wykorzystywany przez twórców programów komercyjnych (np. Adobe Photoshop Lightroom) i takich o otwartym kodzie źródłowym (np. Audacity), ale także do usprawniania procesów prototypowania, szczególnie w branży gier komputerowych. W tej ostatniej kategorii, a więc wśród języków interpretowalnych wykorzystywanych do wsparcia procesu produkcji gier wideo, Lua zajmuje czołową pozycję.

Ta książka dotyczy jednak tego, co łączy Lua i TEXa (a właściwie LATEXa). Te dwie na pozór odległe technologie – z jednej strony zaawansowany system składu tekstu, a z drugiej prosty i uniwersalny język programowania – skrzyżowały się jednak dopiero w 2007 r. To właśnie wtedy w San Diego, przepięknym mieście na zachodzie Stanów Zjednoczonych, tuż przy granicy z Meksykiem, odbywał się coroczny zjazd członków Grupy Użytkowników TEXa (*TeX Users Group*). Taco Hoekwater, z pochodzenia i zamieszkania Holender, zaprezentował tam wówczas efekty dwuletniej pracy nad programem LuaTEX. Pozwalał on kompilować dokumenty napisane w języku TEX, a przy tym osadzać w nich dodatkowy kod napisany w języku Lua, wykonywany automatycznie w czasie kompilacji dokumentu. Zastosowane w nim rozwiązania umożliwiały też dwustronną komunikację na linii TEX–Lua.

Zespół projektu LuaTEX planował wówczas, że pierwsza oficjalna wersja ich programu zostanie opublikowana pod koniec 2009 r. Tak też się stało, a już w 2010 r. została opublikowana jego pierwsza wersja stabilna. Mimo bardzo dynamicznego rozwoju, LuaTEX w wersji 1.00 (a więc w tej, która wypełniła wszystkie początkowe założenia autorów) został opublikowany dopiero we wrześniu 2016 r.

Można więc stwierdzić, że trzymasz w ręku, drogi Czytelniku, książkę, która staje na granicy dwóch na pozór odległych światów – programowania i składu tekstu. Po jednej stronie Lua – szybki, łatwy i użyteczny język programowania o licznych zastosowaniach. Po drugiej LATEX – zaawansowany system składu tekstu o rosnącej z roku na rok popularności. Ich połączenie otwiera bez wątpienia wiele nowych możliwości, dla których inspirację tu znajdziesz.

Co znajdziesz w tej książce?

Pierwsza część książki, zatytułowana *Język Lua*, stanowi łagodne wprowadzenie do tego właśnie języka. Zależało mi na tym, aby było ono dostępne nawet dla osób, których dotychczasowe doświadczenie z programowaniem jest niezbyt duże. Odpowiednie rozdziały wprowadzają więc podstawowe pojęcia związane z programowaniem, prezentują najważniejsze konstrukcje języka Lua i szczegółowo wyjaśniają ich działanie. Każdy rozdział kończy się zestawem ćwiczeń i zadań. Większość ćwiczeń nie wymaga dostępu do komputera – mają one zmotywować Cię do zastanowienia się nad zdobytą wiedzą. Zadania mają jednak charakter bardziej praktyczny – polegają na napisaniu lub zmodyfikowaniu kodu w języku Lua tak, aby spełniał on pewne wymagania. Rozwiązanie niektórych zadań może wymagać od Ciebie odrobiny kreatywności, ale wierzę, że świetnie sobie z nimi poradzisz. Gdyby tak nie było, to w dodatku C znajdziesz wiele przykładowych rozwiązań, szczególnie do zadań z początkowych rozdziałów.

Powinieneś wiedzieć, że część pierwsza ma charakter progresywny. Oznacza to, że kolejne rozdziały opierają się silnie na rozdziałach wcześniejszych. Kiedy będziesz więc ją czytać pierwszy raz, powinieneś to robić od początku do końca. W każdej chwili możesz wrócić do poznanego już materiału, jeśli będziesz akurat potrzebować informacji dotyczących konkretnego zagadnienia. Już w pierwszych rozdziałach zakładam jednak, że rozumiesz intuicyjnie pojęcia, takie jak zmienna, wartość i funkcja. Wydaje mi się, że nie jest to założenie przesadzone.

Druga część książki, zatytułowana *LuaLaTeX*, prezentuje możliwości programu o tej samej nazwie. Zależało mi, aby dopiero w rozdziale 11. pojawiły się pierwsze odwołania do TEXa i LATEXa. Dzięki temu pierwsza część książki stanowi niezależny materiał, z którego możesz skorzystać bez żadnej straty nawet wtedy, gdy nie interesuje Cię dynamiczne tworzenie dokumentów. Zależność w drugą stronę niestety nie zachodzi – przed przystąpieniem do lektury części drugiej powinieneś zapoznać się z materiałem zebranym w części pierwszej. Przyjmuję też, że Twoja znajomość LATEXa

jest wystarczająca, aby czytać i rozumieć bez problemu kod napisany w tym języku. Jeśli tak nie jest, to zrozumienie niektórych przykładów może wymagać od Ciebie sięgnięcia do innych książek (sugerowaną listę znajdziesz w dodatku A).

Podobnie jak to było w przypadku części pierwszej, tak i w części drugiej poszczególne rozdziały kończą się zestawami ćwiczeń i zadań. Ich rola nie różni się co prawda od roli ćwiczeń i zadań w części dotyczącej języka Lua, chociaż jest ich trochę mniej. Wierzę, że czytając o możliwościach LuaTEXa sam wpadniesz na wiele interesujących pomysłów, które zechcesz jak najszybciej wcielić w życie.

W trzeciej część książki znajdziesz kilka rozbudowanych przykładów wykorzystania LuaLATEXa do dynamicznego tworzenia dokumentów. Przykłady te osadzone są w kontekście fikcyjnych problemów, chociaż wykorzystywane do ich rozwiązania techniki mają wyjątkowo ogólny charakter. Jeśli więc chcesz się dowiedzieć, czy nauczysz się z tej książki czegoś przydatnego – zajrzyj właśnie tam. Praktyczne przykłady mogą Ci z jednej strony posłużyć jako motywacja na początku przygody z LuaLATEXem, a z drugiej jako inspiracja pod koniec czytania tej książki.

Podziękowania

Chciałbym podziękować osobom, których wsparcie miało nieoceniony wpływ na treść i formę tego wydania. Dziękuję Marcinowi Borkowskiemu i Pawłowi Mleczko za niezwykle inspirujące rozmowy o zawartości tej książki, jeszcze zanim na dobre zabrałem się do jej pisania. Gdyby nie ich doświadczenie, cierpliwość i otwartość, ta książka zapewne nigdy by nie powstała. Dziękuję również tym wszystkim, którzy poświęcili swój czas, żeby przeczytać fragmenty wstępnej wersji maszynopisu i podzielić się ze mną swoimi cennymi uwagami. Byli to (w kolejności alfabetycznej): Marcin Borkowski, Stanisław Gawiejnowicz, Mateusz Hromada, Marek Kaluba, Paweł Mleczko, Łukasz Pawluczuk, Piotr Rzonsowski i Paweł Urbanek. Dziękuję też dziekanowi Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu, który zgodził się dofinansować wydanie tej książki.

Bartłomiej Przybylski

Poznań, sierpień 2017

Część I
Język Lua

Rozdział 1
Zanim zaczniemy

> O tym, co postrzegamy zmysłami, możemy mieć tylko złudne pojęcie. Pewną wiedzę możemy osiągnąć o tym, co postrzegamy rozumem.
>
> Platon

Są na świecie ludzie, którzy postrzegają programowanie komputerów w kategoriach czarnej magii. Są też tacy, którym dostarcza ono na co dzień rozrywki porównywalnej z najpozytywniejszymi doświadczeniami w życiu. Niezależnie jednak od tego, do której grupy zaliczylibyśmy się sami, sięgnięcie do języka Lua zdaje się być dobrą decyzją.

Z jednej strony, Lua jest bowiem językiem wyjątkowo prostym i intuicyjnym, idealnym wręcz do nauki programowania. W swojej prostocie nie wpada on jednak w niebezpieczne objęcia abstrakcji, a tym samym chroni początkującego programistę przed zapomnieniem, że programuje rzeczywisty komputer. To ważne, gdy naszym celem jest nie tylko nauczenie się konkretnego języka, ale także zdobycie cennej umiejętności myślenia algorytmicznego. Bliski związek Lua z językiem C[1] wymusza na użytkownikach tego pierwszego zachowanie zdrowej świadomości własnych działań i decyzji, niezwykle cennej przy stawianiu pierwszych kroków w programowaniu. Mimo to wystarczy już ledwie kilkanaście minut wprowadzenia, aby móc pisać w języku Lua przydatne programy. Zachowanie przez jego autorów odpowiedniej równowagi to coś, co zasługuje na wielkie uznanie.

Z drugiej strony, Lua jest językiem o wyjątkowym potencjale do rozbudowy. Ta sama prostota, która powoduje, że jest on przyjazny dla początkujących, daje zaawansowanym użytkownikom niewyobrażalną swobodę. Uniwersalność zastosowanych w języku Lua rozwiązań stanowi solidny fundament, na którym można budować skomplikowane i różnorodne konstrukcje. Jest to zatem język dla małych i dużych – i każdy może się czuć równie dobrze, pisząc w nim programy.

Celem tej części książki jest wprowadzenie Ciebie, drogi Czytelniku, w podstawy języka Lua. Jeśli programowałeś już wcześniej, na pewno znajdziesz wiele podobieństw między nim a językami, z którymi się dotąd spotkałeś. Być może nawet opuścisz niektóre fragmenty kolejnych rozdziałów, ponieważ będą poruszać bliskie

[1] Historia języka C sięga 1971 r., kiedy to Dennis Ritchie rozpoczął wraz z kilkoma kolegami prace nad ulepszeniem innego języka, nazwanego B. Więcej informacji o historii języka C można znaleźć na stronie `http://www.bell-labs.com/usr/dmr/www/chist.html` [dostęp: 11.08.2017 r.].

Ci zagadnienia. Nie daj się jednak zwieść – Lua jest językiem, który potrafi zaskoczyć zwinnością zastosowanych rozwiązań. W tej książce znajdziesz opis przynajmniej niektórych z nich. Jeśli jednak Twoje doświadczenie z programowaniem jest wyjątkowo skromne, cóż, czy jest lepszy moment niż teraz, aby to zmienić?

1.1. Środowisko pracy

Warto wiedzieć

Mózgiem każdego współczesnego komputera jest procesor. To on wykonuje operacje arytmetyczne i logiczne na danych przechowywanych w pamięci komputera. W swoim działaniu procesor nie podejmuje jednak samodzielnych decyzji – wykonuje tylko te operacje, które kazano mu wykonać.

Procesor operuje wyłącznie na ciągach bitów (zer i jedynek), które utożsamiamy z liczbami zapisanymi w systemie dwójkowym. Instrukcje przekazywane do procesora są więc po prostu liczbami. Na przykład, procesor zgodny z architekturą x86 zinterpretuje ciąg bitów `10110000 00110110` jako instrukcję zapisania (`10110`) w rejestrze AL (`000`) wartości liczbowej 54 (`00110110`). Na tym skończymy – dokładne zrozumienie działania współczesnych procesorów nie jest niezbędne do zrozumienia tej książki. Warto jednak wiedzieć, że większość programów komputerowych jest ciągiem tak zapisanych instrukcji. Mówimy o nich, że są one opisem programu w *języku maszynowym*, a więc w języku zrozumiałym bezpośrednio przez procesor.

Choć programy napisane w języku maszynowym są zwykle bardzo szybkie, to jednak ich tworzenie jest uciążliwe, trudne i niesie za sobą duże ryzyko popełnienia błędu. Nic więc dziwnego, że zdecydowana większość programistów opiera swoją pracę na bardziej intuicyjnych i łatwiejszych do zrozumienia językach, które jednak wymagają dodatkowego pośrednika, zdolnego przetłumaczyć zapisany w nich kod do postaci zrozumiałej przez procesor. Wyróżniamy dwie zasadnicze grupy języków programowania, zależne od tego, w jaki sposób odbywa się to tłumaczenie:

— **Języki kompilowane.** Aby wykonać na komputerze kod napisany w języku kompilowanym, należy go uprzednio przekształcić na ciąg instrukcji zrozumiałych dla procesora, a więc na język maszynowy. Proces ten, nazywany *kompilacją*, realizowany jest z pomocą dedykowanego programu zwanego *kompilatorem*. Uzyskany w wyniku kompilacji plik wykonywalny może być uruchamiany na różnych komputerach[a], nawet jeśli nie są one wyposażone w jakikolwiek kompilator. Do języków kompilowanych zaliczamy wspomniany już wcześniej język C, ale także języki, takie jak choćby: C++, Pascal, czy Fortran.

— **Języki interpretowane.** W ich przypadku rolę pośrednika między programistą a procesorem pełni nie kompilator, który generuje jednorazowo ciąg instrukcji w języku maszynowym, ale samodzielny *interpreter*, niezbędny do każdorazowego uruchomienia programu. Ten – gdy dostarczamy mu kod napisany w odpowiednim języku interpretowanym – odczytuje i wykonuje go w czasie rzeczywistym. Wprowadzenie stałego pośrednika między program a procesor powoduje, że aplikacje napisane w językach interpretowanych są zwykle wolniejsze niż ich odpowiedniki napisane w językach kompilowanych[b]. Jest to koszt wygody. Poza Lua, do grupy języków interpretowanych zaliczamy także języki, takie jak: Python, Ruby, PHP, czy JavaScript.

Podział na języki interpretowane i kompilowane nie jest niestety sztywny. Można dość łatwo znaleźć przykłady takich języków programowania, które są jednocześnie kompilowane i interpretowane oraz takich, które trudno byłoby zakwalifikować do którejkolwiek z tych grup. Te zagadnienia wykraczają jednak znacząco poza zakres tej książki.

[a] W dawnych czasach różne procesory mogły rozumieć zupełnie różnie te same ciągi liczb (instrukcji). Postępująca od lat standaryzacja spowodowała, że obecnie większość procesorów współdzieli między sobą uniwersalne zestawy rozkazów. Nie oznacza to jednak, że raz skompilowany program będziemy mogli z powodzeniem uruchomić na każdym komputerze. Należy wziąć tu pod uwagę fakt, że program jest tak naprawdę uruchamiany przez system operacyjny, a sam plik wykonywalny składa się zwykle nie tylko z instrukcji przeznaczonych dla procesora, ale także dla systemu operacyjnego. To między innymi z tego powodu wiele gotowych plików wykonywalnych (dostępnych np. w internecie) kategoryzowanych jest nie tylko ze względu na architekturę procesora (np. x86 lub AMD64), ale także ze względu na system operacyjny, dla którego są przeznaczone.

[b] W przypadku niektórych języków interpretowanych, w tym Lua, jest możliwe wcześniejsze przetworzenie kodu źródłowego przez program, który nazywamy *prekompilatorem*. Przekształca on oryginalny kod programu do postaci pośredniej, między wyjściowym kodem a kodem maszynowym, która to może być wykonana przez interpreter o wiele wydajniej.

Interpreter Lua jest programem napisanym całkowicie w języku ANSI C[2]. Dziś trudno jest znaleźć środowisko, dla którego nie istnieje kompilator języka ANSI C, ponieważ język ten jest wyjątkowo popularny i powszechny. Co więcej, wiele systemów

[2] Błyskawiczny wzrost popularności języka C w latach 1973–1983 sprawił, że konieczne stało się jego uporządkowanie. W 1983 r. American National Standards Institute (ANSI) powołał komisję, której postawiono za cel opracowanie zestawu standardów dla języka C. Prace zakończyły się w 1989 r., a ich efektem było powstanie języka ANSI C. W kolejnym roku standard bliźniaczy do tego autorstwa ANSI został ratyfikowany przez Międzynarodową Organizację Normalizacyjną (ISO). Od tamtego czasu każda z instytucji wprowadza raz na kilka lat skromne rozszerzenia do zarządzanego przez siebie dokumentu i adaptuje te wprowadzone przez drugą z organizacji. Dziś więc określenia ANSI C i ISO C odnoszą się właściwie do tego samego, a ci, którzy nie chcą się mieszać w sprawy polityczne używają po prostu nazwy „Standard C".

operacyjnych daje swoim użytkownikom dostęp do kompilatora tego języka w standardzie. To niemalże gwarantuje, że będziesz mógł korzystać z interpretera Lua, weryfikować przykłady, eksperymentować i rozwiązywać ćwiczenia prezentowane w dalszej części tej książki na swoim komputerze, niezależnie od tego, jaki procesor nim steruje i pod kontrolą jakiego systemu operacyjnego on działa.

Zanim będzie to jednak możliwe, musisz wejść w posiadanie plików wykonywalnych interpretera. Są dwie drogi, które prowadzą do tego celu. Jeśli jesteś zaawansowanym użytkownikiem komputera lub doświadczonym programistą, możesz pobrać z internetu wszystkie niezbędne pliki źródłowe i dokonać ich kompilacji samodzielnie[3]. Jeśli nie chcesz brudzić sobie rąk, możesz zaopatrzyć się w gotowe do wykorzystania pliki wykonywalne i to z nich korzystać[4]. Jeśli nie wiesz, co robić, możesz po prostu poprosić któregoś ze swoich przyjaciół o pomoc na tym etapie.

W pierwszej części książki będziemy wykorzystywać interpreter Lua w wersji 5.3 – jest to najaktualniejsza wersja dostępna w chwili pisania tych słów. Większość prezentowanych przykładów zadziała w starszych wersjach Lua bez modyfikacji, niektóre mogą wymagać kosmetycznych zmian, ale kilka będzie wykorzystywać funkcjonalności dostępne dopiero od wersji 5.3. Wszędzie tam, gdzie będziemy mieli do czynienia z takimi przypadkami, zaznaczę to wyraźnie.

Uwaga

Książka ta dotyczy zastosowań Lua do tworzenia dokumentów w LaTeXu. Obecnie silnik LuaTeX, któremu przyjrzymy się bliżej w drugiej części książki, wykorzystuje Lua w wersji 5.2. Najbliższe rozdziały będą jednak opisywały możliwości interpretera Lua w wersji 5.3. Stoją za tym trzy zasadnicze powody. Po pierwsze, kolejne wersje Lua nie różnią się od siebie aż tak bardzo. Znając Lua w wersji 5.3, doskonale poradzisz sobie z pisaniem programów w wersji 5.2, a zakres ewentualnych zmian będzie w większości przypadków skromny. Po drugie, Lua w wersji 5.3 to rozwiązanie nowsze, stabilniejsze i bardziej dopracowane. Jego znajomość da Ci szansę zastosowania go w codziennej pracy, wykraczającej dalece poza tworzenie dokumentów w LaTeXu. Po trzecie, kwestią kilku miesięcy jest zaktualizowanie do wersji 5.3 interpretera Lua wbudowanego w silnik LuaTeX, a gdy to się stanie, będziesz w posiadaniu wiedzy o tym, jak najlepiej można wykorzystać jego możliwości.

[3] Pliki źródłowe interpretera, prekompilatora i osadzanych bibliotek dla programów pisanych w języku C można pobrać ze strony `http://www.lua.org/ftp/`. Szczegółowe instrukcje dotyczące dalszego postępowania z nimi znajdują się na stronie internetowej `http://www.lua.org/manual/5.3/readme.html`.

[4] Te dostępne są z poziomu strony internetowej `http://lua-users.org/wiki/LuaBinaries`. Zawsze korzystaj z najnowszych wersji programów, chyba że masz ważny powód, aby tego nie robić.

1.2. Programy i biblioteki

Niezależnie od tego, czy zdecydujesz się samodzielnie skompilować pliki źródłowe interpretera Lua, czy też pobierzesz je z internetu w postaci gotowych plików wykonywalnych, powinieneś znać przeznaczenie programów i bibliotek, które na końcu znajdą się na dysku Twojego komputera. Mogą one występować pod różnymi nazwami, dlatego w tabeli 1.1 znajdziesz ich skrócony opis z podziałem na poszczególne platformy systemowe[5].

Żeby zachować jednolitość prezentowanych dalej przykładów, będę zakładał, że odwołanie się do polecenia `lua` w wykorzystywanym przez Ciebie programie powłoki (wierszu poleceń) spowoduje wywołanie pliku interpretera Lua w wersji 5.3, tak jak w poniższym przykładzie.

Program powłoki

```
$ lua
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
>
```

Jeśli tak nie jest, powinieneś to albo naprawić, albo dokonać samodzielnie odpowiednich modyfikacji prezentowanych przykładów wszędzie tam, gdzie w podobny sposób będziemy się odwoływać do tego programu.

* * * *

Ćwiczenie 1-1. Znajdź przykład takiego języka programowania, który jest jednocześnie interpretowany i kompilowany. Jakie, Twoim zdaniem, mogą być zalety i wady takiego rozwiązania?

Ćwiczenie 1-2. Spróbuj odnaleźć na dysku swojego komputera wszystkie pliki wymienione w tabeli 1.1. Zwróć szczególną uwagę na plik interpretera. Sprawdź, gdzie się on znajduje i jak się do niego odwołać z poziomu programu powłoki.

[5] *Komentarz uzupełniający dla zaawansowanych:* Sercem języka Lua jest napisana w języku ANSI C *biblioteka Lua*, z której mogą korzystać zewnętrzne programy, zwane *aplikacjami hosta*. Te programy przekazują kod napisany w języku Lua do odpowiedniej funkcji biblioteki Lua i przechwytują rezultaty jej działania. Jedną z licznych aplikacji hosta jest sam interpreter Lua (również napisany w języku ANSI C), który pośredniczy między programistą a biblioteką Lua w typowy dla interpreterów sposób. Innym przykładem aplikacji opartej częściowo na bibliotece Lua mógłby być popularny program do obróbki dźwięku, Audacity. Program ten wykorzystuje bibliotekę Lua do interpretowania kodów dodatków (wtyczek) pisanych przez użytkowników. W tej książce będziemy przyjmować, że interpreter języka Lua jest samodzielną i niezależną aplikacją. W dalszych rozdziałach będziemy też używać pojęcia biblioteki do określania wbudowanego lub zewnętrznego zestawu funkcji ułatwiających pewne działania na poziomie języka Lua.

Tabela 1.1. Znaczenie plików związanych z językiem Lua

Nazwa pliku		**Znaczenie**
UNIX/Linux	Windows	
`lua` `lua53`	`lua.exe` `lua53.exe`	Plik wykonywalny interpretera. To ten plik będzie intensywnie eksploatowany w tej części książki.
`luac` `luac53`	`luac.exe` `luac53.exe`	Plik wykonywalny prekompilatora języka Lua. Umożliwia on wstępne przetworzenie kodu w języku Lua tak, aby interpreter mógł go wykonać szybciej.
-	`wlua53.exe`	Przygotowany specjalnie dla systemu Windows interpreter Lua, niebedący aplikacją konsolową.
-	`lua53.dll`	Plik biblioteki dynamicznej Lua wykorzystywany do uruchomienia interpretera Lua w systemach operacyjnych z rodziny Windows.
Inne pliki z rozszerzeniem `.so`, `.a`, `.dll` lub `.h`		Biblioteki i pliki nagłówkowe do tworzenia w języku ANSI C programów wykorzystujących możliwości języka Lua.

Ćwiczenie 1-3. Uruchom program interpretera i upewnij się, że posiadasz jego odpowiednią wersję. Następnie sprawdź, jak się on zachowa, jeśli wprowadzisz za pomocą klawiatury i zatwierdzisz klawiszem *Enter* następujące polecenia:

```
print(2^15)
for i = 1, 20 do print(i) end
```

Aby zakończyć działanie interpretera i powrócić do programu powłoki tekstowej, wprowadź i zatwierdź polecenie

```
os.exit()
```

Ćwiczenie 1-4. Jeśli znasz jakiś inny interpretowany język programowania, taki jak choćby Python lub Ruby, poeksperymentuj, aby sprawdzić, czy niektóre występujące w nim konstrukcje są zrozumiałe również dla interpretera Lua.

Rozdział 2
Elementarz

Jeśli chcesz poznać wierzchołek baobabu,
poznaj najpierw jego korzenie.

Przysłowie afrykańskie

Niepisana tradycja nakazuje, aby przygodę z nowym językiem programowania rozpocząć od zaimplementowania bardzo prostego programu wypisującego na wyjściu ciąg znaków „Hello, world". W języku Lua, podobnie jak w większości innych języków interpretowanych, kod realizujący taką funkcjonalność zajmuje dokładnie jeden wiersz[1].

Wyciąg 2-1. Zawartość pliku `hello.lua`

```
1 print("Hello, world")
```

Aby wykonać ten program (programy napisane w językach interpretowalnych nazywamy też *skryptami*), należy odwołać się do pliku interpretera, przekazując do niego jako parametr ścieżkę do pliku `hello.lua`. W szczególnym przypadku, jeśli plik ten znajduje się w bieżącym katalogu roboczym programu powłoki, jako parametr wystarczy podać jego nazwę.

Program powłoki

```
$ lua hello.lua
Hello, world
$
```

Po zatwierdzeniu takiego polecenia, wywołany interpreter odczyta zawartość pliku `hello.lua`, wykonując natychmiast zawarte w nim instrukcje, jedna po drugiej, a następnie zakończy swoje działanie, powodując powrót do powłoki tekstowej. Nasz pierwszy skrypt jest zbudowany z jednej tylko instrukcji – wywołania funkcji `print` z argumentem będącym ciągiem znaków „Hello, world".

[1] Jeśli prezentowanemu fragmentowi kodu towarzyszy nazwa pliku źródłowego, to oznacza to, że plik ten jest dostępny do pobrania ze strony internetowej Wydawnictwa Naukowego PWN. Pamiętaj jednak, że samodzielnie przepisany do komputera kod zostawia w pamięci o wiele trwalszy ślad. Niektóre wyciagi będą prezentować jedynie fragmenty wskazanych plików, toteż po ich lewej stronie zostały umieszczone dla wygody numery odpowiadających tym fragmentom wierszy.

Wbudowana funkcja `print` wypisuje na wyjściu wartości przekazane do niej jako argumenty. Doskonale radzi sobie nie tylko z ciągami znaków, ale także z innymi wartościami, na przykład liczbami. W rozdziale 8. poznamy bardziej zaawansowane sposoby komunikowania się ze światem zewnętrznym, do tego czasu funkcja `print` będzie dla nas jednak zupełnie wystarczająca.

Warto wiedzieć

Dobrym zwyczajem jest umieszczenie na początku pliku źródłowego wskazówki, jaki program powinien być wykorzystywany do interpretacji zawartego w nim kodu. Umieszcza się ją w pierwszym wierszu, zwanym wierszem *shebang*. Powinien on zawierać informację o położeniu interpretera, poprzedzoną charakterystyczną sekwencją znaków '`#!`', na wzór poniższego przykładu.

Wyciąg 2-2. Zawartość pliku `hello-shebang.lua`

```
1 #!/usr/bin/env lua
2 print("Hello, world")
```

Jeśli używasz systemu operacyjnego zgodnego ze standardem POSIX (np. systemu z rodziny Linux), to wiersz *shebang* będzie z pewnością brany pod uwagę przez środowisko uruchomieniowe zawsze wtedy, gdy odwołasz się do pliku zawierającego kod tak, jak do zwykłego programu. Jego nieobecność może natomiast sprawić, że w dobrej wierze zostanie wywołany nie ten interpreter, który powinien. W szczególności, jeśli plik `hello-shebang.lua` znajduje się w bieżącym katalogu (oznaczanym symbolem '`.`') i masz na poziomie systemu operacyjnego prawo do jego wykonania, to możesz przekazać odpowiedni kod do interpretera Lua, pisząc

Program powłoki

```
$ ./hello-shebang.lua
Hello, world
$
```

Lua ignoruje pierwszy wiersz interpretowanego pliku, jeśli rozpoczyna się on od sekwencji znaków '`#!`'[a]. Z tego powodu umieszczenie w pliku źródłowym wiersza *shebang* nigdy nie zaszkodzi, a nada Twoim plikom uniwersalnego charakteru. To ważne, jeśli planujesz dzielić się efektami swojej pracy z innymi.

[a] W wielu językach programowania symbol '`#`' oznacza początek komentarza. W nich wiersz *shebang* jest ignorowany z zasady. W języku Lua symbol '`#`' nie rozpoczyna komentarza, a zatem jego autorzy musieli wprowadzić dla wiersza *shebang* specjalny wyjątek.

2.1. Tryb interaktywny

Wywołanie interpretera języka Lua bez dodatkowych parametrów powoduje uruchomienie go w *trybie interaktywnym*. W tym trybie interpreter oczekuje od użytkownika wprowadzenia instrukcji, a każdą z nich natychmiast wykonuje.

Program powłoki

```
$ lua
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
> print("Hello, world")
Hello, world
> print(30)
30
> print("Hello, world", 30)
Hello, world  30
>
```

Zwróć uwagę, że w ostatnim przypadku wywołaliśmy funkcję `print` z dwoma argumentami oddzielonymi przecinkiem – ciągiem znaków „Hello, world” oraz liczbą całkowitą o wartości 30. Gdy funkcja `print` wypisuje na wyjściu wiele wartości, oddziela je symbolem tabulatora. Obie wartości zostały więc wypisane w jednym wierszu z zachowaniem eleganckiego odstępu.

Jak już się przekonałeś, jeśli do interpretera Lua przekażemy jako parametr ścieżkę do pliku, wykona on zawarte w nim instrukcje, a następnie zakończy swoje działanie. Nie zawsze jest to jednak zachowanie pożądane. Rozważmy na przykład plik `sum.lua`, który zawiera definicję rekurencyjnej funkcji `sum`, zwracającej sumę liczb naturalnych od 1 do *n*.

Wyciąg 2-3. Zawartość pliku `sum.lua`

```
1 function sum(n)
2     if n <= 0 then
3         return 0
4     else
5         return n + sum(n-1)
6     end
7 end
```

Wywołanie interpretera z pojedynczym parametrem `sum.lua` nie byłoby zbyt praktyczne. Interpreter co prawda odczytałby zawartość wskazanego pliku i wykonał zawarte w nim instrukcje, tworząc funkcję `sum`, ale zaraz potem zakończyłby swoje działanie, usuwając przy okazji funkcję `sum` z pamięci tymczasowej i uniemożliwiając nam jej późniejsze wykorzystanie.

Program powłoki

```
$ lua sum.lua
$
```

Aby rozwiązać ten problem, możemy uruchomić interpreter z dodatkowym parametrem `-i`, który spowoduje, że po zapisaniu funkcji `sum` w pamięci (a w ogólności po wykonaniu instrukcji zawartych we wskazanym pliku), pozostanie on w trybie interaktywnym, umożliwiając nam jej wywołanie.

Program powłoki

```
$ lua -i sum.lua
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
> print(sum(10))
55
> print(sum(15))
120
>
```

Innym sposobem podejścia do tego wyzwania jest uruchomienie interpretera Lua w trybie interaktywnym, a następnie wywołanie wbudowanej funkcji `dofile` w celu wykonania kodu zawartego w zewnętrznym pliku. Funkcja `dofile` przyjmuje jako argument ścieżkę do pliku, który ma zostać przetworzony.

Program powłoki

```
$ lua
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
> dofile("sum.lua")
> print(sum(16))
136
>
```

Aby zakończyć działanie interpretera `lua`, a w konsekwencji powrócić do programu powłoki tekstowej, możesz wywołać wbudowaną funkcję `os.exit` bez żadnych argumentów (por. ćw. 1-3.).

Program powłoki

```
$ lua
Lua 5.3.4  Copyright (C) 1994-2017 Lua.org, PUC-Rio
> print("Hello, world")
Hello, world
> os.exit()
$
```

Uwaga

Od tego momentu będę zakładać, że potrafisz już uruchomić interpreter Lua w trybie interaktywnym, poruszać się w tym trybie i opuścić go. Prezentowane dalej przykłady będą więc osadzone w kontekście trybu interaktywnego, a nie programu powłoki.

2.2. Anatomia programu

Przydatne programy rzadko kiedy składają się z pojedynczych instrukcji. Zwykle są zbudowane z wielu powiązanych ze sobą fragmentów kodu, często znajdujących się w kilku plikach źródłowych. Niezależnie jednak od tego, z jak rozbudowanym programem mamy do czynienia, da się go rozłożyć na uniwersalne części – małe i uniwersalne klocki, z których go zbudowano.

2.2.1. Wartości i typy danych

Najmniejszymi elementami składowymi programu napisanego w języku Lua – niejako jego molekułami – są proste wartości związane z jednym z czterech podstawowych typów danych: `nil`, `boolean`, `number` oraz `string`.

Istnieje tylko jedna wartość typu `nil` – jest to `nil`. Jej podstawowym zadaniem jest reprezentowanie *braku* wartości. Choć może to brzmieć zaskakująco, w języku Lua wartość nieokreślona jest reprezentowana przez konkretną, odróżniającą się od każdej innej wartość. Jak się przekonasz, wartość `nil` będziemy spotykać na swojej drodze bardzo często.

Wartość typu `boolean` może odwzorowywać jeden z dwóch stanów logicznych: prawda (`true`) lub fałsz (`false`). Typ `number` pozwala nam zapisać w programie wartości liczbowe, zarówno całkowite (np. `-2`), jak i zmiennoprzecinkowe (np. `2.4`)[2]. W wielu językach programowania wartościom całkowitym i zmiennoprzecinkowym odpowiadają różne typy danych. W języku Lua wszystkie liczby mieszczą się w granicach jednego typu: `number`[3]. W końcu, wartości typu `string` to stałe ciągi bajtów (najczęściej interpretowanych jako opisy znaków). Podając je wprost, otaczamy je symbolami pojedynczych lub podwójnych cudzysłowów, pisząc przykładowo `"Hello, world"` lub `'Hello, world'`.

[2] Liczba zmiennoprzecinkowa jest zdefiniowana przez trzy niezależne od siebie wartości: *znak*, *mantysę* oraz *cechę*. Jeśli operujemy w systemie dziesiętnym, to znak równy -1, mantysa równa 2, 5 i cecha równa 3 opisują liczbę $-2,5 \cdot 10^3 = -2500$.

[3] We wcześniejszych wersjach Lua liczby całkowite były zapisywane w pamięci komputera jako liczby zmiennoprzecinkowe. Dopiero od wersji 5.3 Lua traktuje liczby całkowite i zmiennoprzecinkowe jako osobne *podtypy* typu `number`. Wrócimy do tego w rozdziale 4.

Zwróć uwagę, że wartości żadnych dwóch z wymienionych powyżej typów danych nie pokrywają się ze sobą. Oznacza to, że dowolna wartość niesie za sobą jednoznaczną informację o swoim typie. Możemy się o tym przekonać, korzystając z wbudowanej w Lua funkcji `type`. Z każdym wywołaniem zwraca ona ciąg znaków reprezentujący typ danych przekazanych do niej jako argument.

Tryb interaktywny Lua 5.3

```
> print(type(nil))
nil
> print(type(12), type("12"))
number  string
> print(type(true), type("true"))
boolean string
> print(type(false), type(3.1415))
boolean number
> print(type(type(nil)))
string
```

Zauważ, że `12` i `"12"` to dwie różne wartości. Pierwsza reprezentuje liczbę równą tuzinowi, druga zaś ciąg znaków zbudowany z dwóch symboli: jedynki i dwójki. Podobnie zresztą `true` i `"true"` są wartościami różnych typów – odpowiednio wartością logiczną i ciągiem znaków. W ostatnim przykładzie wypisujemy typ wartości zwracanej przez funkcję `type`. Ta zaś jest zawsze ciągiem znaków – niezależnie od tego, z jakim argumentem ją wywołamy.

Poza czterema typami podstawowymi, Lua udostępnia cztery typy obiektowe: `function`, `table`, `thread` oraz `userdata`. Są one związane z rozbudowanymi strukturami, również tymi reprezentującymi dane składające się z instrukcji i wielu wartości różnych typów.

Wartości typu `function` to po prostu funkcje. Każda z nich, jako ciąg uniwersalnych instrukcji, do których możemy się później odwoływać, jest przechowywana w pamięci jako wartość typu `function`. Wartości typu `table` (nazywane *tabelami*) są workami, w których możemy przechowywać wiele opisanych etykietami wartości dowolnych typów[4]. Wartości typu `thread` są związane ze współprogramami wykonywanymi przez program napisany w języku Lua[5]. W końcu, wartości typu `userdata` obejmują wszystko to, co nie mieści się w żadnym z siedmiu pozostałych typów. W praktyce są to binarne dane tworzone i interpretowane przez biblioteki napisane w języku C. Chociaż typy `thread` i `userdata` są wyjątkowo interesujące,

[4] Stosując dosyć hermetyczny język techniczny, powiedzielibyśmy, że są to bliscy krewni heterogenicznych tablic asocjacyjnych znanych z innych języków programowania.

[5] Mówiąc bardzo ogólnie, współprogramy (*coroutines*) pozwalają wyodrębnić w programie niezależne od siebie, ale ściśle współpracujące ze sobą części. Tworzenie, a nawet rozumienie idei współprogramów niestety znacząco wykracza poza zakres tej książki. Wspominam o nich zresztą jeszcze na końcu tej części, w rozdziale pt. „Czego nie znajdziesz w tej książce".

to jednak ze względu na tematykę tej książki pominiemy ich szczegółowe omówienie. Warto jednak wiedzieć o ich istnieniu[6].

Tryb interaktywny Lua 5.3

```
> print(type(print))
function
> print(type(io))
table
> print(type(io.stdin))
userdata
```

W każdym z powyższych przykładów odwołujemy się do wbudowanych w Lua wartości powiązanych z określonymi nazwami. Więcej o nazwach związanych z danymi (zmiennych) powiemy w paragrafie 2.3.

W języku Lua wszystkie wartości, niezależnie od typu, są pierwszoklasowe. Oznacza to, że mogą one być przypisywane do zmiennych, przekazywane do funkcji jako argumenty oraz zwracane przez te funkcje. Jak się wkrótce okaże, niesie to za sobą szereg możliwości.

2.2.2. Wyrażenia

Z wartości możemy budować wyrażenia. Przykładem wyrażenia jest oczywiście pojedyncza wartość (np. `2`). Te wykorzystywane w praktyce są jednak często bardziej rozbudowane. Przykładowo, `2+3` jest już wyrażeniem zbudowanym z dwóch wartości typu `number` oraz symbolu operatora dodawania '+'. Nasz mózg automatycznie utożsamia to wyrażenie z wartością 5, reprezentującą sumę liczb 2 i 3. Podobnie jest w języku Lua – każde zrozumiałe dla interpretera wyrażenie jest przekształcane automatycznie do postaci pojedynczej wartości lub ciągu kilku wartości, których nie można już dalej w żaden sposób przekształcić. Z tego powodu możemy powiedzieć na przykład, że wyrażenie `2+3` *przyjmuje* lub *ma* wartość `5`. Z kolejnych rozdziałów dowiesz się, jak tworzyć wyrażenia oparte na poszczególnych typach wartości.

Większość wyrażeń nie jest instrukcjami języka, ponieważ nie ma samodzielnego wpływu na stan programu. Obliczenie przez Lua wartości wyrażenia `2+3` to bowiem za mało, aby można było powiedzieć, że program coś zrobił. Konkretne wartości nabierają sensu dopiero wtedy, gdy są umieszczone w szerszym kontekście (np. są przekazywane do funkcji jako argumenty). Niemniej jednak, wprowadzanie

[6] W rozdziale 8. przyjrzymy się bliżej mechanizmom obsługi wejścia i wyjścia. W języku Lua opierają się one na standardowych rozwiązaniach dostarczanych przez język ANSI C, przez co wymuszają wykorzystanie wartości typu `userdata`. Będziemy go jednak traktować jak *czarną skrzynkę*, nie analizując szczegółowo cech tego typu.

jednej lub kilku oddzielonych symbolem przecinka wartości w trybie interaktywnym (i tylko w nim!) powoduje ich wypisanie na wyjściu[7].

Tryb interaktywny Lua 5.3

```
> 10
10
> 2+3
5
> 10, 2+3
10      5
> "Hello, world"
Hello, world
> type(12)
number
> type(12), "Hello, world", 10-5
number  Hello, world    5
```

Zauważ, że w ostatnich dwóch przypadkach wyrażenie `type(12)` przyjęło wartość `"number"`. Jest tak dlatego, że funkcja `type` wywołana z argumentem liczbowym `12` zwraca ciąg znaków `"number"`. Przypomnij sobie ponadto, że wykorzystywaliśmy już wcześniej symbol przecinka do oddzielania wartości przekazywanych jako argumenty do funkcji `print`. Nie jest to przypadek – wywołanie funkcji z kilkoma argumentami to nic innego, jak przekazanie do niej ciągu kilku wartości. Do tego tematu wrócimy jeszcze w rozdziale 7.

2.2.3. Instrukcje

Instrukcje to najmniejsze *funkcjonalne* części, z których jest złożony program. Wiążą się one z wykonaniem operacji wpływających na jego stan. Instrukcjami są zatem, między innymi, operacje przypisania wartości do zmiennych, konstrukcje warunkowe, pętle czy wywołania funkcji.

Być może zauważyłeś już podświadomie, że wywołania funkcji mają charakter dwojaki. Z jednej strony są one instrukcjami, bo przecież inicjują wykonanie instrukcji, z których te funkcje się składają (np. wywołanie funkcji `print` powoduje wypisanie pewnych danych na wyjściu), a z drugiej strony *mogą* być częścią wyrażenia, jeśli zwracają wartość (tak jak robi to choćby funkcja `type`). Wywołania funkcji to jedyny przypadek takiego dualizmu.

Interpreter Lua doskonale radzi sobie z rozpoznawaniem instrukcji. Jeśli jeden wiersz kodu zawiera kilka instrukcji, zostaną one wykonane jedna po drugiej tak,

[7] Z tej wygodnej własności trybu interaktywnego będziemy korzystać dość obficie. W trybie interaktywnym Lua w wersji 5.2 i wcześniejszych należy poprzedzić wyrażenie symbolem '=', aby jego wartość została wypisana na wyjściu, pisząc np. '`=2+3`'. Zawsze pamiętaj jednak o tym, że dowolny plik źródłowy języka Lua musi się składać z instrukcji. Jeśli będziesz chciał wypisać na wyjściu wartość konketnego wyrażenia poza trybem interaktywnym, możesz użyć np. funkcji `print`.

jak byśmy się mogli tego spodziewać. Podobnie, jeśli rozpoczęta instrukcja zostanie przerwana znakiem nowej linii w rozsądnym miejscu, to Lua będzie oczekiwać jej kontynuacji na początku kolejnego wiersza.

Możemy wyraźnie zaznaczyć, gdzie kończy się wprowadzona instrukcja, stosując znany z innych języków programowania symbol średnika. Choć nie jest to obowiązkowe, dobrym zwyczajem jest oddzielanie nim kilku instrukcji umieszczonych w jednym wierszu.

Tryb interaktywny Lua 5.3

```
> print(5) print(5+5); print(5+10)
5
10
15
> print(
>> "Instrukcja dwuwierszowa")
Instrukcja dwuwierszowa
```

Wyświetlana w trybie interaktywnym sekwencja '>>' oznacza, że Lua oczekuje na zakończenie lub wprowadzanie dalszej części rozpoczętej wcześniej instrukcji.

2.2.4. Porcje kodu i programy

Najwyższe miejsce w anatomicznej strukturze programu napisanego w języku Lua zajmują porcje kodu (*chunk*). Są to zestawy instrukcji wykonywanych przez interpreter za jednym zamachem.

Kiedy wykonywaliśmy w trybie interaktywnym pojedyncze instrukcje lub zestawy instrukcji umieszczone w pojedynczym wierszu, to one stanowiły odpowiednie porcje kodu. Podobnie, gdy odwoływaliśmy się do zawartości zewnętrznych plików, czy to wywołując program interpretera, czy odwołując się w trybie interaktywnym do funkcji `dofile`, instrukcje zawarte w tych plikach były wykonywane razem. Stanowiły więc porcje kodu. Każdy program napisany w języku Lua i wykonany przez interpreter jest zestawem jednej lub większej liczby (często zależnych od siebie) porcji kodu – pojedynczych instrukcji, wierszy instrukcji lub plików zawierających wiele wierszy z instrukcjami.

2.3. Zmienne

Lua, podobnie jak niemalże wszystkie inne języki programowania, udostępnia mechanizmy obsługi zmiennych. Z perspektywy programisty sprowadzają się one do możliwości wykonania dwóch czynności: przypisania wartości do zmiennej o określonej nazwie (identyfikatorze) i późniejszego odwołania się do tej wartości z wykorzystaniem tej nazwy. W poniższym przykładzie przypisujemy do zmiennej x wartość liczbową 5, a następnie kilkukrotnie się do niej odwołujemy.