

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTEL尼亚

FRAGMENTY KSIĄŻEK ONLINE

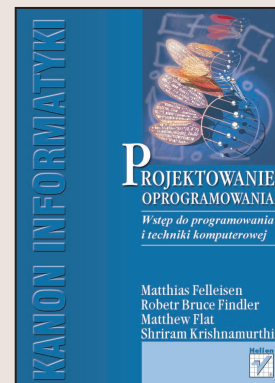
Projektowanie oprogramowania. Wstęp do programowania i techniki komputerowej

Autorzy: Matthias Felleisen, Robert Bruce Findler,
Matthew Flatt, Shriram Krishnamurthi
Tłumaczenie: Bartosz Grabski, Mikołaj Szczepaniak
ISBN: 83-7197-922-3

Tytuł oryginału: [How to Design Programs](#)

Format: B5, stron: 644

[Przykłady na ftp: 32 kB](#)



Umiejętność programowania nie ma już charakteru czysto zawodowego. Księgowi muszą się posługiwać arkuszami kalkulacyjnymi i edytorami tekstu, fotografowie korzystają z edytorów zdjęć, muzycy programują syntezatory, zaś profesjonalni programiści tworzą skomplikowane aplikacje. Programowanie jest więc bardzo pożądaną umiejętnością, potrzebną nie tylko informatykom. Projektowanie oprogramowania wymaga takich samych zdolności analitycznych, jak matematyka. Jednak, w przeciwieństwie do matematyki, praca z programami jest aktywnym sposobem zdobywania wiedzy. Obcowanie z oprogramowaniem daje możliwość stałej interakcji, co pozwala na zgłębianie wiedzy, eksperymentowanie z nią oraz na stałą samoocenę.

Autorzy tej klasycznej publikacji stawiają tezę, iż „każdy powinien nauczyć się, jak projektować oprogramowanie” i właśnie nauka podstaw projektowania jest jej tematem głównym. W książce znajdziesz wiele podstawowych algorytmów, wyjaśnienia takich pojęć, jak akumulacja wiedzy czy równość ekstensjonalna i intensjonalna, słowem wszystko to, co stanowi teoretyczną podstawę wiedzy programistycznej.

Poznasz między innymi:

- Podstawowe struktury, z których składają się programy komputerowe
- Proste i złożony typy danych
- Metody przetwarzania danych
- Programowanie z użyciem rekurencji, algorytmy z nawracaniem
- Projektowanie abstrakcyjne
- Sposoby gromadzenia wiedzy
- Wykorzystanie wektorów

Z lektury książki „Projektowanie oprogramowania. Wstęp do programowania i techniki komputerowej” skorzystają zarówno studenci informatyki, jak też i słuchacze innych kierunków oraz wszystkie osoby, które chcą podbudować swoją wiedzę praktyczną solidnymi i przydatnymi podstawami teoretycznymi.



Spis treści

Przedmowa	9
Dlaczego każdy powinien uczyć się programować?	11
Metody projektowania	12
Wybór Scheme i DrScheme	14
Podział książki	15
Podziękowania	18

Część I Przetwarzanie prostych typów danych **19**

1. Studenci, nauczyciele i komputery	21
2. Liczby, wyrażenia i proste programy	23
Liczby i arytmetyka	23
Zmienne i programy	26
Problemy ze zrozumieniem treści zadań	29
Błędy	30
Projektowanie programów	33
3. Program składa się z funkcji i definicji zmiennych	39
Składanie funkcji	40
Definicje zmiennych	43
Proste ćwiczenia w tworzeniu funkcji	44
4. Instrukcje warunkowe i funkcje	47
Wartości logiczne i relacje	47
Funkcje testujące warunki	50
Warunki i funkcje warunkowe	54
Projektowanie funkcji warunkowych	57
5. Informacje symboliczne	63
Proste ćwiczenia z symbolami	65
6. Dane złożone. Część 1.: Struktury	69
Struktury	69
Ćwiczenie rozszerzone: rysowanie prostych obrazów	72

Definicje struktur	75
Definicje danych.....	79
Projektowanie funkcji przetwarzających dane złożone	82
Rozszerzone ćwiczenie: przemieszczanie okręgów i prostokątów	87
Rozszerzone ćwiczenie: gra w szubienicę	91
7. Rodzaje danych.....	95
Mieszanie i rozróżnianie danych	95
Projektowanie funkcji przetwarzających dane mieszane	100
Składanie funkcji — powtórka	104
Rozszerzone ćwiczenie: przesuwanie figur.....	107
Błędne dane wejściowe	108
W1. Składnia i semantyka	111
Słownictwo języka Scheme	112
Gramatyka języka Scheme	112
Znaczenie w języku Scheme	114
Błędy	118
Wyrażenia logiczne	121
Definicje zmiennych	122
Definicje struktur	124
Część II Przetwarzanie danych dowolnej wielkości	127
9. Dane złożone. Część 2.: Listy	129
Listy	129
Definicje danych dla list o dowolnej długości	133
Przetwarzanie list o dowolnej długości	135
Projektowanie funkcji dla rekursywnych definicji danych.....	139
Więcej na temat przetwarzania prostych list	142
10. Więcej na temat przetwarzania list.....	147
Funkcje zwracające listy.....	147
Listy zawierające struktury	152
Rozszerzone ćwiczenie: przemieszczanie obrazów	158
11. Liczby naturalne	161
Definiowanie liczb naturalnych.....	161
Przetwarzanie liczb naturalnych dowolnej wielkości.....	163
Rozszerzone ćwiczenie: tworzenie list, testowanie funkcji.....	166
Alternatywne definicje danych dla liczb naturalnych	168
Więcej o naturze liczb naturalnych	173
12. Łączenie funkcji. Powtórka	177
Projektowanie skomplikowanych programów	177
Rekursywne funkcje zewnętrzne	178
Uogólnianie problemów i funkcji.....	183
Rozszerzone ćwiczenie: przestawianie słów	187
W2. Skracanie list	191

Część III Więcej o przetwarzaniu danych dowolnej wielkości

197

14. Więcej rekurencyjnych definicji danych	199
Struktury w strukturach	199
Rozszerzone ćwiczenie: drzewa poszukiwań binarnych	208
Listy w listach	212
Rozszerzone ćwiczenie: obliczanie wyrażeń języka Scheme	215
15. Wzajemne odwołania w definicjach danych	217
Listy struktur. Listy w strukturach	217
Projektowanie funkcji dla definicji danych zawierających wzajemne odwołania	223
Rozszerzone ćwiczenie: więcej na stronach WWW	225
16. Tworzenie programów metodą iteracyjnego ulepszania	227
Analiza danych	228
Definiowanie i ulepszanie klas danych	229
Ulepszanie funkcji i programów	232
17. Przetwarzanie dwóch skomplikowanych elementów danych	235
Jednoczesne przetwarzanie dwóch list. Przypadek 1.	235
Jednoczesne przetwarzanie dwóch list. Przypadek 2.	237
Jednoczesne przetwarzanie dwóch list. Przypadek 3.	240
Upraszczenie funkcji	245
Projektowanie funkcji pobierających dwie złożone dane wejściowe	247
Ćwiczenia z przetwarzania dwóch złożonych danych wejściowych	248
Rozszerzone ćwiczenie: obliczanie wyrażeń języka Scheme. Część 2.	251
Równość i testowanie	253
W3. Lokalne definicje i zasięg leksykalny	261
Organizowanie programów za pomocą słowa local	261
Zasięg leksykalny i struktura blokowa	276

Część IV Projektowanie abstrakcyjne

281

19. Podobieństwa w definicjach	283
Podobieństwa w funkcjach	283
Podobieństwa w definicjach danych	292
20. Funkcje są wartościami	297
Składnia i semantyka	297
Kontrakty dla abstrakcyjnych i polimorficznych funkcji	299
21. Projektowanie funkcji abstrakcyjnych na podstawie przykładów ...	303
Abstrahowanie na podstawie przykładów	303
Ćwiczenia z abstrakcyjnymi funkcjami przetwarzającymi listy	309
Abstrakcja i pojedynczy punkt kontroli	311
Rozszerzone ćwiczenie: przemieszczanie obrazów jeszcze raz	312
Uwaga: Projektowanie abstrakcji na podstawie szablonów	314

22.	Projektowanie abstrakcji	317
	Funkcje zwracające funkcje	317
	Projektowanie abstrakcji z funkcjami jako wartościami	319
	Pierwsze spojrzenie na graficzny interfejs użytkownika	322
23.	Przykłady matematyczne	331
	Ciągi i szeregi	331
	Ciągi i szeregi arytmetyczne	333
	Ciągi i szeregi geometryczne	334
	Pole powierzchni pod wykresem funkcji	338
	Nachylenie funkcji	340
W4.	Bezpośrednie definiowanie funkcji	345

Część V Rekursja generatywna 351

25.	Nowa postać rekursji	353
	Modelowanie kuli na stole	354
	Szybkie sortowanie	357
26.	Projektowanie algorytmów	363
	Zakończenie	365
	Rekursja strukturalna a generatywna	368
	Dokonywanie wyborów	369
27.	Różne algorytmy rekurencyjne	375
	Fraktale	375
	Od plików do linii, od list do list list	380
	Wyszukiwanie binarne	384
	Metoda Newtona	390
	Rozszerzone ćwiczenie: eliminacja Gaussa	392
28.	Algorytmy z nawracaniem	397
	Przechodzenie grafów	397
	Rozszerzone ćwiczenie: szachowanie hetmanów	403
W5.	Koszt obliczeniowy oraz wektory	405
	Czas konkretny, czas abstrakcyjny	405
	Definicja wyrażenia „rzędu”	410
	Pierwsze spojrzenie na wektory	412

Część VI Gromadzenie wiedzy 423

30.	Utrata wiedzy	425
	Problem przetwarzania strukturalnego	425
	Problem rekursji generatywnej	429
31.	Projektowanie funkcji z akumulatorem	433
	Czy akumulator jest potrzebny?	433
	Funkcje z akumulatorem	434
	Przekształcanie funkcji na funkcje z akumulatorem	436

32.	Dalsze użycie akumulacji	447
	Rozszerzone ćwiczenie: akumulatory i drzewa	447
	Rozszerzone ćwiczenie: misjonarze i ludożercy	452
	Rozszerzone ćwiczenie: plansza gry Solitaire	455
W6.	Natura liczb niedokładnych	457
	Arytmetyka liczb o stałym rozmiarze	457
	Przepełnienie	463
	Niedomiar	464
	Liczby w DrScheme	465

Część VII Zmiana stanu zmiennych **467**

34.	Pamięć dla funkcji	469
35.	Przypisanie do zmiennych	475
	Działanie prostych przypisań	475
	Sekwencja wyrażeń obliczeniowych	477
	Przypisania i funkcje	479
	Pierwszy użyteczny przykład	482
36.	Projektowanie funkcji z pamięcią	485
	Zapotrzebowanie na pamięć	485
	Pamięć i zmienne stanu	487
	Funkcje inicjalizujące pamięć	489
	Funkcje zmieniające pamięć	489
37.	Przykłady zastosowania pamięci	497
	Inicjalizacja stanu	497
	Zmiana stanu przez interakcję z użytkownikiem	500
	Zmiany stanu przez rekursję	508
	Ćwiczenia na zmianach stanu	514
	Rozszerzone ćwiczenie: zwiedzanie	516
W7.	Końcowa składnia i semantyka	519
	Słownik Advanced Scheme	519
	Gramatyka Advanced Scheme	519
	Znaczenie Advanced Scheme	522
	Błędy w Advanced Scheme	534

Część VIII Zmiana wartości złożonych **539**

39.	Hermetyzacja	541
	Abstrahowanie ze zmiennymi stanu	541
	Ćwiczenia z hermetyzacji	551
40.	Mutacja struktur	553
	Struktury z funkcji	553
	Mutacja struktur funkcjonalnych	556

Mutacja struktur.....	558
Mutacja wektorów	565
Zmiana zmiennych, zmiana struktur	567
41. Projektowanie funkcji zmieniających struktury	571
Po co mutować struktury	571
Zasady projektowania strukturalnego i mutacji, część 1.	572
Zasady projektowania strukturalnego i mutacji, część 2.	583
Ćwiczenie rozszerzone: ruchome obrazy po raz ostatni	594
42. Równość	595
Równość ekstensjonalna	595
Równość intensjonalna.....	596
43. Zmiana struktur, wektorów i obiektów	601
Ćwiczenia praktyczne z wektorami.....	601
Zbiory struktur z cyklami.....	616
Nawracanie ze stanem	626
Zakończenie	629
Technika obliczeniowa.....	629
Programowanie	630
Krok naprzód.....	631
Dodatki	633
Skorowidz	635

Przetwarzanie dwóch skomplikowanych elementów danych

Czasami funkcja pobiera dwa argumenty należące do klas zdefiniowanych za pomocą skomplikowanych definicji danych. W niektórych przypadkach jeden z argumentów powinien być traktowany tak, jakby był argumentem atomowym; precyzyjnie sformułowany opis celu zazwyczaj to wyjaśnia. W innych przypadkach oba argumenty muszą być przetwarzane równolegle. Czasem funkcja będzie musiała brać pod uwagę wszystkie możliwe przypadki i odpowiednio przetwarzać dane argumenty. Ten rozdział ilustruje te trzy możliwości za pomocą przykładów i wprowadza rozszerzoną metodę projektowania dla ostatniego przypadku. W ostatnim podrozdziale omawiamy równoważność danych złożonych i jej związek z procesem testowania; ma to istotne znaczenie dla automatyzacji testów funkcji.

Jednoczesne przetwarzanie dwóch list. Przypadek 1.

Przeanalizuj następujący kontrakt, opis celu i nagłówkek:

```
;; zastap-empty-lista : lista-liczb lista-liczb -> lista-liczb
;; tworzy nową listę zastępując empty w liście dana-lista-liczb1 listą dana-lista-liczb2
(define (zastap-empty-lista dana-lista-liczb1 dana-lista-liczb2) ...)
```

Kontrakt mówi, że funkcja pobiera dwie listy; z takim zdarzeniem nie spotkaliśmy się wcześniej. Zobaczmy, jak nasza metoda projektowania sprawdzi się w tym przypadku.

Po pierwsze, tworzymy przykłady. Przypuśćmy, że pierwszą daną wejściową jest empty. Funkcja *zastap-empty-lista* powinna w takim przypadku zwrócić drugi argument, niezależnie od tego, co zawiera:

```
(zastap-empty-lista empty L)
= L
```

W powyższym równaniu *L* reprezentuje dowolną listę liczb. Przypuśćmy teraz, że pierwszy argument jest różny od empty. Opis celu mówi, że powinniśmy w takim przypadku zastąpić empty na końcu listy *dana-lista-liczb1* listą *dana-lista-liczb2*:


```
(zastap-empty-lista (cons 1 empty) L)
;; oczekiwana wartość:
(cons 1 L)

(zastap-empty-lista (cons 2 (cons 1 empty)) L)
;; oczekiwana wartość:
(cons 2 (cons 1 L))

(zastap-empty-lista (cons 2 (cons 11 (cons 1 empty))) L)
;; oczekiwana wartość:
(cons 2 (cons 11 (cons 1 L)))
```

W powyższych przykładach *L* ponownie reprezentuje dowolną listę liczb.

Przykłady sugerują, że tak długo, jak drugi argument jest listą, nie ma znaczenia, co on zawiera; w przeciwnym przypadku nie miałoby sensu zastępowanie *empty* drugim argumentem. Oznacza to, że mając na uwadze pierwszy argument, powinniśmy wykończyć szablon dla funkcji przetwarzających listy:

```
(define (zastap-empty-lista dana-lista-liczb1 dana-lista-liczb2)
  (cond
    ((empty? dana-lista-liczb1) ...)
    (else ... (first dana-lista-liczb1) ... (zastap-empty-lista (rest dana-lista-liczb1)
      dana-lista-liczb2) ... )))
```

Drugi argument traktujemy na razie tak, jakby był daną atomową.

Wypełnijmy teraz puste miejsca w szablonie zgodnie z zaleceniami metody projektowania. Jeśli lista *dana-lista-liczb1* jest pusta, funkcja *zastap-empty-lista* zwraca *dana-lista-liczb2* zgodnie z naszymi przykładami. W drugiej klauzuli wyrażenia **cond**, odpowiadającej danej na wejściu niepustej liście *dana-lista-liczb1*, musimy przeanalizować dostępne wyrażenia:

- (1) *(first dana-lista-liczb1)* wyciąga pierwszy element listy,
- (2) *(zastap-empty-lista (rest dana-lista-liczb1) dana-lista-liczb2)* zamienia *empty* na liście *(rest dana-lista-liczb1)* listą *dana-lista-liczb2*.

Aby lepiej zrozumieć, co to oznacza, przeanalizuj poniższy przykład:

```
(zastap-empty-lista (cons 2 (cons 11 (cons 1 empty))) L)
;; oczekiwana wartość:
(cons 2 (cons 11 (cons 1 L)))
```

Powyżej *(first dana-lista-liczb1)* wynosi 2, *(rest dana-lista-liczb1)* ma wartość *(cons 11 (cons 1 empty))*, zaś *(zastap-empty-lista (rest dana-lista-liczb1) dana-lista-liczb2)* zwraca wartość *(cons 11 (cons 1 dana-lista-liczb2))*. Łącząc liczbę 2 z ostatnią wartością za pomocą instrukcji *cons*, możemy otrzymać pożądaną wartość. Ogólnie:

```
(cons (first dana-lista-liczb1) (zastap-empty-lista (rest dana-lista-liczb1) dana-lista-liczb2))
```

jest wynikiem drugiej klauzuli wyrażenia **cond**. Listing 17.1 zawiera kompletną definicję funkcji.

Listing 17.1. Kompletna definicja funkcji zastap-empty-lista

```
;; zastap-empty-lista : lista-liczb lista-liczb -> lista-liczb
;; tworzy nową listę zastępując empty w liście dana-lista-liczb1 listą dana-lista-liczb2
(define (zastap-empty-lista dana-lista-liczb1 dana-lista-liczb2)
  (cond
    ((empty? dana-lista-liczb1) dana-lista-liczb2)
    (else (cons (first dana-lista-liczb1) (zastap-empty-lista (rest dana-lista-liczb1)
                                                              dana-lista-liczb2))))))
```

Ćwiczenia

Ćwiczenie 17.1. W wielu ćwiczeniach używaliśmy operacji append języka Scheme, która pobiera trzy listy i zestawia ich elementy w jedną listę:

```
(append (list 'a) (list 'b 'c) (list 'd 'e 'f))
;; oczekiwana wartość:
(list 'a 'b 'c 'd 'e 'f)
```

Wykorzystaj funkcję *zastap-empty-lista* do zdefiniowania funkcji nasz-append, która powinna działać identycznie jak append udostępniany w języku Scheme.

Ćwiczenie 17.2. Opracuj funkcję *krzyzuj*, która pobierze listę symboli oraz listę liczb i zwróci wszystkie możliwe pary symboli z liczbami.

Przykład:

```
(krzyzuj '(a b c) '(1 2))
;; oczekiwana wartość:
(list (list 'a 1) (list 'a 2) (list 'b 1) (list 'b 2) (list 'c 1) (list 'c 2))
```

Jednoczesne przetwarzanie dwóch list. Przypadek 2.

W rozdziale 10. opracowaliśmy funkcję *godziny->wynagrodzenie* obliczającą wynagrodzenie pracowników na podstawie przepracowanych godzin. Funkcja pobierała listę liczb (przepracowanych przez pracowników godzin) i zwracała inną listę liczb (należnych wynagrodzeń). Dla uproszczenia założyliśmy, że wszyscy pracownicy mają taką samą stawkę godzinową. Nawet jednak małe firmy zatrudniają pracowników na zróżnicowanych warunkach. Przeważnie księgowy firmy utrzymuje dwa zbiory informacji: stałą, która między innymi zawiera stawkę godzinową danego pracownika, oraz tymczasową, w której zawarta jest informacja o liczbie przepracowanych w ostatnich miesiącach godzin.

Nowy, rozszerzony opis problemu oznacza, że funkcja powinna pobierać *dwie* listy. Aby uprościć sobie ten problem, założmy, że listy zawierają jedynie liczby: jedna — stawki godzinowe, druga — liczby przepracowanych godzin. Oto nasz kontrakt, opis celu i nagłówek:

```
;; godziny->wynagrodzenia : lista-liczb lista-liczb -> lista-liczb
;; konstruuje nową listę zawierającą iloczyny odpowiednich
;; elementów list dana-lista-liczb1 i dana-lista-liczb2
;; ZAŁOŻENIE: listy mają równą długość
(define (godziny->wynagrodzenia dana-lista-liczb1 dana-lista-liczb2) ...)
```

Możemy traktować listę *dana-lista-liczb1* jako listę stawek godzinowych, zaś *dana-lista-liczb2* jako listę przepracowanych w ostatnim miesiącu godzin. Aby otrzymać listę wynagrodzeń, musimy przemnożyć odpowiednie liczby z obu list.

Spójrzmy na kilka przykładów:

```
(godziny->wynagrodzenia empty empty)
;; oczekiwana wartość:
empty

(godziny->wynagrodzenia (cons 5.65 empty) (cons 40 empty))
;; oczekiwana wartość:
(cons 226.0 empty)

(godziny->wynagrodzenia (cons 5.65 (cons 8.75 empty))
  (cons 40.0 (cons 30.0 empty)))
;; oczekiwana wartość:
(cons 226.0 (cons 262.5 empty))
```

We wszystkich trzech przykładach na wejściu funkcji podano pary list takich samych długości. Jak napisaliśmy w dodatku do opisu celu, funkcja zakłada, że dane spełniają ten warunek — i faktycznie, stosowanie funkcji z naruszeniem tego warunku nie ma sensu.

Warunek dotyczący danych wejściowych można wyjaśnić podczas opracowywania szablonu. Konkretnie, warunek mówi, że (`empty? dana-lista-liczb1`) ma wartość `true` wtedy i tylko wtedy, gdy (`empty? dana-lista-liczb2`) ma wartość `true`. Co więcej, (`cons? dana-lista-liczb1`) ma wartość `true` wtedy i tylko wtedy, gdy (`cons? dana-lista-liczb2`) ma wartość `true`. Innymi słowy, warunek upraszcza projekt struktury wyrażeń warunkowych **cond** w szablonie, ponieważ oznacza, że szablon jest podobny do szablonu dla funkcji przetwarzających listy:

```
(define (godziny->wynagrodzenia dana-lista-liczb1 dana-lista-liczb2)
  (cond
    ((empty? dana-lista-liczb1) ...)
    (else ... )))
```

W pierwszej klauzuli **cond** zarówno *dana-lista-liczb1* jak i *dana-lista-liczb2* są listami pustymi — `empty`. Nie potrzebujemy więc żadnego selektora. W drugiej klauzuli zarówno *dana-lista-liczb1* jak i *dana-lista-liczb2* są skonstruowanymi listami, co oznacza, że potrzebujemy czterech selektorów:

```
(define (godziny->wynagrodzenia dana-lista-liczb1 dana-lista-liczb2)
  (cond
    ((empty? dana-lista-liczb1) ...)
```

```
(else
  ... (first dana-lista-liczb1) ... (first dana-lista-liczb2) ...
  ... (rest dana-lista-liczb1) ... (rest dana-lista-liczb2) ... )))
```

Ponieważ ostatnie dwa selektory dotyczą list o identycznych długościach, w oczywisty sposób możemy je wykorzystać w naturalnej rekursji funkcji *godziny->wynagrodzenia*:

```
(define (godziny->wynagrodzenia dana-lista-liczb1 dana-lista-liczb2)
  (cond
    ((empty? dana-lista-liczb1) ...)
    (else
     ... (first dana-lista-liczb1) ... (first dana-lista-liczb2) ...
     ... (godziny->wynagrodzenia (rest dana-lista-liczb1) (rest dana-lista-liczb2)) ... )))
```

Jedynym niezwykłym elementem tego szablonu jest rekursywne wywołanie funkcji złożone z dwóch wyrażeń, z których oba są selektorami dwóch argumentów funkcji. Jak się już przekonaliśmy, idea działania funkcji jest łatwa do wytłumaczenia dzięki założeniu, że *dana-lista-liczb1* i *dana-lista-liczb2* mają równą długość.

Podczas definiowania funkcji będziemy postępować zgodnie z zaleceniami metody projektowania. Pierwszy przykład oznacza, że odpowiedzią pierwszej klauzuli wyrażenia **cond** powinna być lista pusta — *empty*. W drugiej klauzuli mamy do dyspozycji trzy wartości:

- (1) *(first dana-lista-liczb1)*, która reprezentuje pierwszy element listy stawek godzinowych;
- (2) *(first dana-lista-liczb2)*, która reprezentuje pierwszy element listy przepracowanych godzin; oraz
- (3) *(godziny->wynagrodzenia (rest dana-lista-liczb1) (rest dana-lista-liczb2))*, która jest listą wynagrodzeń dla reszt list *dana-lista-liczb1* i *dana-lista-liczb2*.

Aby otrzymać ostateczny wynik, musimy jedynie odpowiednio połączyć te wartości. A dokładniej, zgodnie z opisem celu musimy obliczyć wynagrodzenie dla pierwszego pracownika i skonstruować listę złożoną z tej wartości i wynagrodzeń pozostałych pracowników. Oznacza to, że odpowiedź dla drugiej klauzuli wyrażenia **cond** powinna wyglądać następująco:

```
(cons (wynagrodzenie (first dana-lista-liczb1) (first dana-lista-liczb2))
      (godziny->wynagrodzenia (rest dana-lista-liczb1) (rest dana-lista-liczb2))))
```

Zewnętrzna funkcja *wynagrodzenie* pobiera dwa pierwsze elementy i oblicza odpowiednie wynagrodzenie. Listing 17.2 zawiera kompletne definicje obu funkcji.

Listing 17.2. Kompletna definicja funkcji *godziny->wynagrodzenia*

```
;; godziny->wynagrodzenia : lista-liczb lista-liczb -> lista-liczb
;; konstruuje nową listę zawierającą iloczyny odpowiednich
;; elementów list dana-lista-liczb1 i dana-lista-liczb2
;; ZAŁOŻENIE: listy mają równą długość
```

```

(define (godziny->wynagrodzenia dana-lista-liczb1 dana-lista-liczb2)
  (cond
    ((empty? dana-lista-liczb1) empty)
    (else (cons (wynagrodzenie (first dana-lista-liczb1) (first dana-lista-liczb2))
      (godziny->wynagrodzenia (rest dana-lista-liczb1) (rest dana-lista-liczb2))))))

;; wynagrodzenie : liczba liczba -> liczba
;; oblicza wynagrodzenie na podstawie danych liczb: stawka-godzinowa oraz
przepracowane-godziny
(define (wynagrodzenie stawka-godzinowa przepracowane-godziny)
  (* stawka-godzinowa przepracowane-godziny))

```

Ćwiczenia

Ćwiczenie 17.3. W rzeczywistym świecie funkcja *godziny->wynagrodzenia* pobierałaby listę struktur reprezentujących pracowników i listę struktur reprezentujących przebieg prac w ostatnim miesiącu. Struktura pracownika zawiera jego nazwisko, numer PESEL oraz stawkę godzinową. Struktura opisująca przebieg pracy zawiera nazwisko pracownika i liczbę przepracowanych w danym miesiącu godzin. Wynikiem jest lista struktur zawierających nazwisko pracownika i należne mu wynagrodzenie.

Zmodyfikuj funkcję z listingu 17.2 tak, aby pracowała na powyższych klasach danych. Opracuj potrzebne definicje struktur i definicje danych. Zastosuj metodę projektowania w procesie modyfikacji funkcji.

Ćwiczenie 17.4. Opracuj funkcję *zepnij*, która pobierze listę nazwisk oraz listę numerów telefonicznych i połączy je w listę podobną do książki telefonicznej. Zakładając, że mamy następującą definicję struktury:

```
(define-struct wpis (nazwisko numer))
```

pojedynczy wpis w książce telefonicznej konstruujemy za pomocą instrukcji (*make-wpis s n*), gdzie *s* jest symbolem, a *n* jest liczbą. Załóż, że dane na wejściu listy mają identyczne długości. Uprość definicję na tyle, na ile będzie to możliwe.

Jednoczesne przetwarzanie dwóch list. Przypadek 3.

Oto trzeci opis problemu przedstawiony w formie kontraktu, opisu celu i nagłówka:

```

;; wybierz-z-listy : lista-symboli N[>= 1] -> symbol
;; określa n-ty symbol na liście dana-lista-symboli, licząc od 1;
;; sygnalizuje błąd, jeśli na danej liście nie ma n-tego symbolu
(define (wybierz-z-listy dana-lista-symboli n) ...)

```

Powyższy program wymaga opracowania funkcji, która będzie pobierać liczbę naturalną i listę symboli. Obie dane wejściowe należą do klas opisanych skomplikowanymi definicjami danych, jednak inaczej niż w dwóch poprzednich problemach, klasy te są całkowicie rozłączne. Na listingu 17.3 przypominamy obie definicje.

Listing 17.3. Definicje danych dla funkcji *wybierz-z-listy*

Definicje danych:

liczba naturalna [≥ 1] ($N[\geq 1]$) jest albo:

- (1) 1, albo
- (2) (*dodaj1* *n*), jeśli *n* należy do $N[\geq 1]$.

lista-symboli jest albo:

- (1) listą pustą, *empty*, albo
- (2) (*cons* *s* *ls*), gdzie *s* jest symbolem, a *ls* jest listą symboli.

Ponieważ problem jest nietypowy, powinniśmy upewnić się, że nasze przykłady obejmują wszystkie ważne przypadki. Ten cel osiągamy zazwyczaj wybierając po jednym przykładzie dla każdej klauzuli z definicji i wybierając losowo elementy dla pozostałych, prostych elementów danych. W tym przykładzie taka procedura prowadzi nas do wybrania co najmniej dwóch elementów dla danej *lista-symboli* i dwóch dla $N[\geq 1]$. Wybierzmy *empty* i (*cons* 'a *empty*) dla listy, oraz 1 i 3 dla liczb naturalnych. Po dwa przykłady dla obu argumentów oznaczają, że będziemy mieli ich łącznie cztery, nie mamy jednak danego wprost związku pomiędzy tymi dwoma argumentami, ani żadnych ograniczeń wspomnianych w kontrakcie:

```
(wybierz-z-listy empty 1)
;; oczekiwane zachowanie:
(error 'wybierz-z-listy "...")
```

```
(wybierz-z-listy (cons 'a empty) 1)
;; oczekiwana wartość:
'a
```

```
(wybierz-z-listy empty 3)
;; oczekiwane zachowanie:
(error 'wybierz-z-listy "...")
```

```
(wybierz-z-listy (cons 'a empty) 3)
;; oczekiwane zachowanie:
(error 'wybierz-z-listy "...")
```

Tylko jeden z czterech wyników jest symbolem; w pozostałych przypadkach otrzymaliśmy błędy związane z brakiem elementów na danych listach.

Dyskusja o przykładach wykazała, że istnieją faktycznie cztery możliwe, niezależne przypadki, które musimy brać pod uwagę podczas projektowania funkcji. Możemy analizować te przypadki za pomocą tabeli zawierającej niezbędne warunki:

	(empty? <i>dana-lista-symboli</i>)	(cons? <i>dana-lista-symboli</i>)
(= <i>n</i> 1)		
(> <i>n</i> 1)		

Wiersze tabeli opisują dane wejściowe, dla których funkcja *wybierz-z-listy* musi określić, co podano jako listę symboli; w kolumnach rozróżniamy dane liczby naturalne. Co więcej, w tabeli mamy cztery pola, z których każde reprezentuje przypadek, w którym zarówno warunek odpowiedniej kolumny jak i wiersza ma wartość true. Możemy to wyrazić za pomocą wyrażeń z operatorem **and** w odpowiednich komórkach tabeli:

	(empty? <i>dana-lista-symboli</i>)	(cons? <i>dana-lista-symboli</i>)
(= <i>n</i> 1)	(and (= <i>n</i> 1) (empty? <i>dana-lista-symboli</i>))	(and (= <i>n</i> 1) (cons? <i>dana-lista-symboli</i>))
(> <i>n</i> 1)	(and (> <i>n</i> 1) (empty? <i>dana-lista-symboli</i>))	(and (> <i>n</i> 1) (cons? <i>dana-lista-symboli</i>))

Łatwo teraz wykazać, że dla dowolnej danej pary argumentów dokładnie jeden z czterech warunków zawartych w komórkach tabeli powyżej i ma wartość true.

Korzystając z naszej analizy przypadków, możemy teraz zaprojektować pierwszą część szablonu — wyrażenie warunkowe:

```
(define (wybierz-z-listy dana-lista-symboli n)
  (cond
    [(and (= n 1) (empty? dana-lista-symboli)) ...]
    [(and (> n 1) (empty? dana-lista-symboli)) ...]
    [(and (= n 1) (cons? dana-lista-symboli)) ...]
    [(and (> n 1) (cons? dana-lista-symboli)) ...]))
```

Wyrażenie **cond** sprawdza wszystkie cztery warunki wyróżniając wszystkie możliwości. Następnie, jeśli to możliwe, musimy dodać selektory do każdej klauzuli tego wyrażenia:

```
(define (wybierz-z-listy dana-lista-symboli n)
  (cond
    [(and (= n 1) (empty? dana-lista-symboli))
     ...]
    [(and (> n 1) (empty? dana-lista-symboli))
     ... (odejmij1 n) ...]
    [(and (= n 1) (cons? dana-lista-symboli))
     ... (first dana-lista-symboli) ... (rest dana-lista-symboli)...]
    [(and (> n 1) (cons? dana-lista-symboli))
     ... (odejmij1 n) ... (first dana-lista-symboli) ... (rest dana-lista-symboli) ...]))
```

Dla liczby naturalnej n szablon zawiera co najwyżej jeden selektor, który określa poprzednika tej liczby w zbiorze liczb naturalnych. Dla listy *dana-lista-symboli* możliwe są maksymalnie dwa selektory. Jednak w przypadku, w którym prawdziwy jest warunek ($= n\ 1$) lub (`empty? dana-lista-symboli`), czyli jeden z dwóch argumentów jest atomowy, nie ma potrzeby stosowania odpowiadających tym danym wejściowym selektorów.

Ostatni krok w procesie konstruowania szablonu wymaga wypisania szablonu z zaznaczonymi rekursjami dla wyrażeń, w których wyrażenia selektorów zwracają dane należące do tej samej klasy, co dane wejściowe. W szablonie dla funkcji *wybierz-z-listy* takie działanie ma sens jedynie dla ostatniej klauzuli **cond**, która zawiera wyrażenia zarówno dla $N[>= 1]$, jak i dla *listy-symboli*. Wszystkie inne klauzule zawierają co najwyżej jedno odpowiednie wyrażenie. Nie jest jednak do końca jasne, jak sformułować w tym przypadku naturalną rekursję. Jeśli zbagatelizujemy cel funkcji i w kroku konstruowania szablonu wypiszemy wszystkie możliwe rekursje, otrzymamy trzy przypadki:

(*wybierz-z-listy* (rest *dana-lista-symboli*) (odejmij1 n))
 (*wybierz-z-listy* *dana-lista-symboli* (odejmij1 n))
 (*wybierz-z-listy* (rest *dana-lista-symboli*) n)

Ponieważ nie wiemy, ani która rekursja ma w tym przykładzie zastosowanie, ani czy możemy wykorzystać wszystkie trzy rekursje, przechodzimy do następnego etapu.

Zgodnie z metodą projektowania, przeanalizujemy każdą z klauzul wyrażenia **cond** naszego szablonu i znajdziemy prawidłową odpowiedź na powyższe pytanie:

- (1) Jeśli warunek (**and** ($= n\ 1$) (`empty? dana-lista-symboli`)) jest prawdziwy, funkcja *wybierz-z-listy* powinna wybrać pierwszy element z pustej listy, co jest niemożliwe. Odpowiedzią funkcji będzie więc sygnalizacja o błędzie.
- (2) Jeśli warunek (**and** ($> n\ 1$) (`empty? dana-lista-symboli`)) jest prawdziwy, funkcja *wybierz-z-listy* powinna znowu wybrać element z pustej listy. Odpowiedzią jest więc znowu błąd.
- (3) Jeśli warunek (**and** ($= n\ 1$) (`cons? dana-lista-symboli`)) jest prawdziwy, funkcja *wybierz-z-listy* powinna zwrócić pierwszy element danej listy. Selektor (`first dana-lista-symboli`) przypomina nam, jak uzyskać ten element. Właśnie on będzie odpowiedzią funkcji.
- (4) W ostatniej klauzuli, jeśli warunek (**and** ($> n\ 1$) (`cons? dana-lista-symboli`)) jest prawdziwy, musimy przeanalizować, co zwracają poszczególne selektory:
 - (a) (`first dana-lista-symboli`) wybiera pierwszy element z listy *symboli*;
 - (b) (`rest dana-lista-symboli`) reprezentuje resztę listy; oraz
 - (c) (`odejmij1 n`) zwraca liczbę mniejszą od jeden od danego indeksu listy.

Rozważmy przykład ilustrujący znaczenie powyższych wyrażeń. Przypuśćmy, że funkcję *wybierz-z-listy* zastosowano dla listy (`cons 'a (cons 'b empty)`) i liczby 2:

(*wybierz-z-listy* (`cons 'a (cons 'b empty)`) 2)

Funkcja powinna zwrócić symbol 'b, ponieważ (`first dana-lista-symboli`) zwróci 'a, natomiast (`odejmij1 n`) zwróci 1. Poniżej przedstawiamy efekty ewentualnego zastosowania trzech naturalnych rekursji dla tych wartości:

- (a) (*wybierz-z-listy* (cons 'b empty) 1) zwraca 'b, czyli pożądaną wartość;
- (b) (*wybierz-z-listy* (cons 'a (cons 'b empty)) 1) zwraca wartość 'a, która jest symbolem, ale nie jest oczekiwaną wartością dla naszego problemu;
- (c) (*wybierz-z-listy* (cons 'b empty) 2) sygnalizuje błąd, ponieważ indeks jest większy niż długość listy.

Powyższa analiza sugeruje, że powinniśmy użyć wyrażenia (*wybierz-z-listy* (cons 'b empty) 1) jako odpowiedzi ostatniej klauzuli **cond**. Oparte na przykładzie rozwiązania są jednak często zawodne, powinniśmy więc spróbować zrozumieć, dlaczego to wyrażenie jest odpowiednie dla naszej funkcji.

Przypomnij sobie, że zgodnie z opisem celu,

(*wybierz-z-listy* (rest *dana-lista-symboli*) (odejmij1 *n*))

wybiera element o indeksie (*n*-1) z listy (rest *dana-lista-liczb*). Innymi słowy, zmniejszamy indeks o 1, skracamy listę o jeden element i szukamy w nowej liście elementu o zmniejszonym indeksie. Wyrażenie zwraca wartość zgodną z oczekiwaniami, podobnie jak odpowiedź klauzuli z warunkiem ($= n - 1$), przy założeniu, że *dana-lista-symboli* i *n* są wartościami złożonymi. Nasz wybór odpowiedzi dla ostatniej klauzuli jest więc całkowicie uzasadniony. Kompletną definicję funkcji *wybierz-z-listy* przedstawia listing 17.4.

Listing 17.4. Kompletna definicja funkcji *wybierz-z-listy*

```
;; wybierz-z-listy : lista-symboli N[>= 1] -> symbol
;; określa n-ty symbol na liście dana-lista-symboli, licząc od 1;
;; sygnalizuje błąd, jeśli na danej liście nie ma n-tego symbolu
(define (wybierz-z-listy dana-lista-symboli n)
  (cond
    [(and (= n 1) (empty? dana-lista-symboli)) (error 'wybierz-z-listy "lista jest za krótka")]
    [(and (> n 1) (empty? dana-lista-symboli)) (error 'wybierz-z-listy "lista jest za krótka")]
    [(and (= n 1) (cons? dana-lista-symboli)) (first dana-lista-symboli)]
    [(and (> n 1) (cons? dana-lista-symboli)) (wybierz-z-listy (rest dana-lista-symboli) (odejmij1 n))]))
```

Ćwiczenia

Ćwiczenie 17.5. Opracuj funkcję *wybierz-z-listy0*, która wybierze elementy z listy podobnie jak *wybierz-z-listy*, ale począwszy od indeksu 0.

Przykłady:

```
(symbol=? (wybierz-z-listy0 (list 'a 'b 'c 'd) 3)
  'd)
```

```
(wybierz-z-listy0 (list 'a 'b 'c 'd) 4)
;; oczekiwane zachowanie:
(error 'wybierz-z-listy0 "lista jest za krótka")
```

Upraszczenie funkcji

Funkcja *wybierz-z-listy* zaprezentowana na listingu 17.4 jest bardziej skomplikowana niż jest to konieczne. Pierwsza i druga klauzula wyrażenia **cond** zwracają takie same odpowiedzi: błąd. Innymi słowy, jeśli wyrażenie:

```
(and (= n 1) (empty? dana-lista-symboli))
```

lub

```
(and (> n 1) (empty? dana-lista-symboli))
```

ma wartość **true**, efektem działania funkcji jest błąd. Możemy więc wykorzystać to podobieństwo i stworzyć prostsze wyrażenie **cond**:

```
(define (wybierz-z-listy dana-lista-symboli n)
  (cond
    [(or (and (= n 1) (empty? dana-lista-symboli))
         (and (> n 1) (empty? dana-lista-symboli))) (error 'wybierz-z-listy "lista jest
za krótka")]
    [(and (= n 1) (cons? dana-lista-symboli)) (first dana-lista-symboli)]
    [(and (> n 1) (cons? dana-lista-symboli)) (wybierz-z-listy (rest dana-lista-symboli)
(odejmij1 n))]))
```

Nowe wyrażenie jest prostym przełożeniem naszych obserwacji na język Scheme.

Aby jeszcze bardziej uprościć naszą funkcję, musimy zapoznać się z regułą algebryczną dotyczącą wartości logicznych:

```
(or (and warunek1 dany-warunek)
    (and warunek2 dany-warunek))
= (and (or warunek1 warunek2)
    dany-warunek)
```

Powyższe równanie nazywa się *prawem de Morgana*. Zastosowanie go w naszej funkcji spowoduje następujące uproszczenie:

```
(define (wybierz-z-listy n dana-lista-symboli)
  (cond
    [(and (or (= n 1) (> n 1))
         (empty? dana-lista-symboli)) (error 'wybierz-z-listy "lista jest za krótka")]
    [(and (= n 1) (cons? dana-lista-symboli)) (first dana-lista-symboli)]
    [(and (> n 1) (cons? dana-lista-symboli)) (wybierz-z-listy (rest dana-lista-symboli)
(odejmij1 n))]))
```

Rozważ teraz pierwszą część warunku **(or (= n 1) (> n 1))**. Ponieważ n należy do zbioru $\mathbb{N}[\geq 1]$, warunek jest zawsze prawdziwy. Gdybyśmy jednak zastąpili go słowem **true**, otrzymalibyśmy warunek:

```
(and true
  (empty? dana-lista-symboli))
```

który jest równoważny z warunkiem `(empty? dana-lista-symboli)`. Innymi słowy, funkcję możemy zapisać w następujący sposób:

```
(define (wybierz-z-listy dana-lista-symboli n)
  (cond
    [(empty? dana-lista-symboli) (error 'wybierz-z-listy "lista jest za krótka")]
    [(and (= n 1) (cons? dana-lista-symboli)) (first dana-lista-symboli)]
    [(and (> n 1) (cons? dana-lista-symboli)) (wybierz-z-listy (rest dana-lista-symboli)
      (odejmij1 n))]))
```

Ta definicja jest znacznie prostsza niż ta, którą zademonstrowaliśmy na listingu 17.4.

Możemy uprościć naszą definicję jeszcze bardziej. Pierwszy warunek w ostatniej wersji funkcji *wybierz-z-listy* odrzuca wszystkie przypadki, w których *dana-lista-symboli* jest pusta. Wyrażenie `(cons? dana-lista-symboli)` w następnych dwóch klauzulach będzie więc miało zawsze wartość `true`. Jeśli zastąpimy warunek tą wartością i uprościmy wyrażenie **and**, otrzymamy najprostszą z możliwych wersję funkcji *wybierz-z-listy*, którą przedstawiliśmy na listingu 17.5. Mimo że ostatnia funkcja jest dużo prostsza od oryginalnej, ważne jest, byśmy rozumieli, że opracowaliśmy obie wersje w sposób systematyczny i tylko dzięki temu możemy być pewni, że działają one poprawnie. Gdybyśmy próbowali od początku tworzyć wersję uproszczoną, prędzej czy później popełnilibyśmy błąd.

Listing 17.5. Uproszczona definicja funkcji *wybierz-z-listy*

```
;; wybierz-z-listy : lista-symboli N[>= 1] -> symbol
;; określa n-ty symbol na liście dana-lista-symboli, licząc od 1;
;; sygnalizuje błąd, jeśli na danej liście nie ma n-tego symbolu
(define (wybierz-z-listy dana-lista-symboli n)
  (cond
    [(empty? dana-lista-symboli) (error 'wybierz-z-listy "lista jest za krótka")]
    [(= n 1) (first dana-lista-symboli)]
    [(> n 1) (wybierz-z-listy (rest dana-lista-symboli) (odejmij1 n))]))
```

Ćwiczenia

Ćwiczenie 17.6. Opracuj funkcję *zastap-empty-lista* zgodnie ze strategią z podrozdziału „Jednoczesne przetwarzanie dwóch list. Przypadek 2.”. Następnie systematycznie upraszczaj definicję funkcji.

Ćwiczenie 17.7. Uprość definicję funkcji *wybierz-z-listy0* z ćwiczenia 17.5 lub wyjaśnij, dlaczego nie można jej uprościć.

Projektowanie funkcji pobierających dwie złożone dane wejściowe

Napotkamy czasem problemy, które wymagają funkcji pobierających dwie złożone klasy danych wejściowych. Najbardziej interesujące będą przypadki, w których obie dane będą miały nieznane rozmiary. Jak się przekonaliśmy w pierwszych trzech podrozdziałach, możemy postępować z takimi danymi wejściowymi na trzy różne sposoby.

Odpowiednim podejściem do tego typu problemów jest postępowanie zgodne z zaleceniami metody projektowania. Przede wszystkim musimy przeprowadzić analizę danych i zdefiniować odpowiednie klasy danych. Następnie możemy stworzyć kontrakt, opis celu funkcji, które kolejno doprowadzają nas do momentu, w którym możemy przejść do następnego kroku. Zanim to jednak zrobimy, powinniśmy się zastanowić, z którą z następujących sytuacji mamy do czynienia:

- (1) W niektórych przypadkach jeden z parametrów ma rolę dominującą. I odwrotnie, możemy traktować jeden z parametrów jako atomowy fragment danych z punktu widzenia opracowywanej funkcji.
- (2) W innych przypadkach oba parametry są zsynchronizowane. Muszą obejmować tę samą klasę wartości o takiej samej strukturze. Np. jeśli mamy dwie listy, to obie muszą mieć taką samą długość. Jeśli mamy dwie strony WWW, muszą one mieć taką samą długość i jeśli jedna z nich zawiera osadzoną stronę, druga również musi zawierać taką stronę. Jeśli decydujemy, że dwa parametry mają taki sam status i muszą być przetwarzane w sposób zsynchronizowany, możemy wybrać jeden z nich i zorganizować funkcję wokół niego.
- (3) Wreszcie, w rzadkich przypadkach, może nie być oczywistego związku pomiędzy dwoma parametrami. Dla takich danych wejściowych musimy analizować wszystkie możliwe przypadki, zanim opracujemy przykłady i zaprojektujemy szablon.

Dla pierwszych dwóch przypadków stosujemy istniejącą metodę projektowania. Ostatni przypadek wymaga dodatkowych uwag.

Po tym, jak zdecydujemy, że funkcja należy do trzeciej kategorii, ale zanim opracujemy przykłady i szablon funkcji, musimy opracować dwuwymiarową tabelę. Oto ponownie tabela dla funkcji *wybierz-z-listy*:

		<i>dana-lista-symboli</i>	
		(empty? <i>dana-lista-symboli</i>)	(cons? <i>dana-lista-symboli</i>)
<i>n</i>	(= <i>n</i> 1)		
	(> <i>n</i> 1)		

W kolumnach wyliczamy warunki rozróżniające podklasy pierwszego parametru, w wierszach wyliczamy warunki dla drugiego parametru.

Tabela pomaga w opracowaniu zarówno zbioru przykładów dla naszej funkcji, jak i jej szablonu. Jeśli chodzi o przykłady, muszą one pokrywać wszystkie możliwe przypadki. Oznacza to, że musimy opracować przynajmniej po jednym przykładzie dla każdej komórki tabeli.

Jeśli chodzi o szablon, musimy w nim zawrzeć po jednej klauzuli wyrażenia **cond** dla każdej komórki. Każda z tych klauzul musi zawierać wszystkie możliwe selektory dla obu parametrów. Jeśli jeden z nich jest atomowy, nie ma oczywiście potrzeby stosowania selektorów. Wreszcie, zamiast pojedynczej naturalnej rekursji może zaistnieć konieczność wprowadzenia wielu rekursji. Dla funkcji *wybierz-z-listy* znaleźliśmy trzy przypadki. Generalnie wszystkie możliwe kombinacje selektorów są potencjalnymi kandydatami do wykorzystania w naturalnej rekursji. Ponieważ nie możemy wiedzieć, które z nich są konieczne, a które nie są, wypisujemy je wszystkie i wybieramy te, które będą odpowiednie dla naszej definicji funkcji.

Podsumowując, projektowanie funkcji dla wielu parametrów opiera się na pewnej odmianie starej metody projektowania. Kluczowe znaczenie ma przełożenie definicji danych na tabelę demonstrującą wszystkie możliwe i interesujące nas kombinacje. Tworzenie przykładów dla funkcji i jej szablonu musi opierać się w jak największym stopniu właśnie na tej tabeli. Wypełnienie pustych przestrzeni w szablonie wymaga, jak zresztą wszystkie pozostałe czynności, praktyki.

Ćwiczenia z przetwarzania dwóch złożonych danych wejściowych

Ćwiczenia

Ćwiczenie 17.8. Opracuj funkcję *scalaj*, która pobierze dwie listy liczb posortowane rosnąco. Wynikiem funkcji będzie pojedyncza, posortowana lista liczb zawierająca wszystkie liczby z obu list (i żadnej innej). Poszczególne liczby powinny występować w liście wyjściowej tyle razy, ile razy pojawiły się w dwóch listach wejściowych.

Przykłady:

```
(scalaj (list 1 3 5 7 9) (list 0 2 4 6 8))  
;; oczekiwana wartość:  
(list 0 1 2 3 4 5 6 7 8 9)
```

```
(scalaj (list 1 8 8 11 12) (list 2 3 4 8 13 14))  
;; oczekiwana wartość:  
(list 1 2 3 4 8 8 8 11 12 13 14)
```

Ćwiczenie 17.9. Celem tego ćwiczenia jest opracowanie nowej wersji gry w szubienicę z rozdziału 6. dla słów o dowolnej długości.

Stwórz definicję danych reprezentujących słowa dowolnej długości za pomocą list. Litera powinna być reprezentowana przez symbole od 'a do 'z oraz symbol '_.

Opracuj funkcję *odslon-liste*, która pobierze trzy argumenty:

- (1) *Wybrane* słowo, które należy odgadnąć.
- (2) Słowo *statusu*, które określa, jak dużą część słowa odgadliśmy do tej pory.
- (3) Literę, która reprezentuje naszą aktualną próbę.

Funkcja zwróci nowe słowo statusu, które składa się z dowolnych liter i znaków '_'. Pola w nowym słowie statusu określamy porównując próbę z każdą parą liter ze słowa statusu i wybranego słowa:

- (1) Jeśli próba jest równa danej literze w wybranym słowie, wstawiamy tę literę w odpowiednie miejsce w słowie statusu.
- (2) W przeciwnym przypadku nowa litera odpowiada literze ze słowa statusu.

Przetestuj funkcję *odslon-liste* dla następujących przykładów:

```
(odslon-liste (list 't 'e 'a) (list ' _ 'e ' _ ) 'u)
(odslon-liste (list 'a 'l 'e) (list 'a ' _ ' _ ) 'e)
(odslon-liste (list 'a 'l 'l) (list ' _ ' _ ' _ ) 'l)
```

Określ — najpierw ręcznie — jakie powinny być rezultaty powyższych wywołań.

Zastosuj pakiet szkoleniowy *hangman.ss* i funkcje *dorysuj-nastepna-czesc* (patrz ćwiczenie 6.27) oraz *odslon-liste*, aby rozegrać grę w szubienicę. Oblicz także wartość następującego wyrażenia:

```
(hangman-list odslon-liste dorysuj-nastepna-czesc)
```

Funkcja *hangman-list* (udostępniona we wspomnianym pakiecie nauczania) wybiera losowe słowo i wyświetla okno z menu wyboru liter. Wybierz litery i gdy będziesz gotowy, kliknij przycisk *Check*, aby sprawdzić, czy Twój strzał był trafny. Powodzenia!

Ćwiczenie 17.10. Robotnicy w fabryce podbijają swoje karty czasowe rano, gdy przychodzą do pracy, i po południu — gdy wychodzą. Obecnie stosuje się elektroniczne karty zawierające numer pracownika i liczbę przepracowanych godzin. Ponadto akta pracownika zawierają zawsze jego nazwisko, numer i stawkę godzinową.

Opracuj funkcję *godziny->wynagrodzenia2*, która pobierze listę akt pracowniczych oraz listę (elektronicznych) kart. Funkcja będzie obliczać miesięczne wynagrodzenie każdego pracownika odpowiednio dopasowując, na podstawie numeru pracownika, dane z jego akt z danymi zapisanymi na karcie. Jeśli zabraknie danej pary lub numery pracowników nie będą się zgadzały, funkcja zatrzyma się z odpowiednim komunikatem o błędzie. Załóż, że istnieje co najwyżej jedna karta dla jednego pracownika i dla odpowiadającemu mu numeru.

Wskazówka: Księgowy posortowałby najpierw obie listy według numerów pracowników.

Ćwiczenie 17.11. *Kombinacja liniowa* jest sumą kilku składników liniowych, co oznacza, że zwraca zmienne i liczby. Te drugie nazywamy w tym kontekście współczynnikami. Oto kilka przykładów:

$$\begin{aligned} &5 * x \\ &5 * x + 17 * y \\ &5 * x + 17 * y + 3 * z \end{aligned}$$

We wszystkich trzech przykładach współczynnikiem przy x jest 5, przy y jest liczba 17, a jedynym przy z jest 3.

Jeśli mamy dane wartości zmiennych, możemy określić wartość wielomianu. Np. jeśli $x = 10$, wartością iloczynu $5 * x$ będzie 50; jeśli $x = 10$ i $y = 1$, wielomian $5 * x + 17 * y$ przyjmie wartość 67; jeśli zaś $x = 10$, $y = 1$ i $z = 2$, wyrażenie $5 * x + 17 * y + 3 * z$ będzie miało wartość 73.

W przeszłości opracowalibyśmy funkcje obliczające wartości liniowych kombinacji dla poszczególnych wartości. Alternatywną reprezentacją takich wielomianów jest lista współczynników. Powyższe kombinacje byłyby reprezentowane przez takie listy:

```
(list 5)
(list 5 17)
(list 5 17 3)
```

Powyższa reprezentacja zakłada, że zgadzamy się zawsze używać zmiennych w określonej kolejności.

Opracuj funkcję *wartosc*, która pobierze reprezentację wielomianu (lista współczynników) oraz listę liczb (wartości zmiennych). Obie listy mają tę samą długość. Funkcja powinna zwrócić wartość tego wielomianu dla danych wartości zmiennych.

Ćwiczenie 17.12. Ewa, Joanna, Dorota i Maria są siostrami, które chciałyby zaoszczędzić pieniądze i ograniczyć wysiłek związany z kupowaniem prezentów gwiazdkowych. Postanowiły więc przeprowadzić losowanie, które przypisze każdej z nich jedną osobę, która następnie zostanie obdarowana prezentem. Ponieważ Joanna jest programistą komputerowym, siostry poprosiły ją o napisanie programu, który przeprowadzi losowanie w sposób bezstronny. Program nie może oczywiście przypisać żadnej siostrze jej samej jako odbiorcy prezentu.

Oto definicja funkcji *wyberz-prezent*, która pobiera listę różnych imion (symboli) i wybiera losowo jeden z układów listy, w którym żaden z elementów nie pozostał na swoim miejscu:

```
;; wybierz-prezent: lista-imion -> lista-imion
;; wybiera "losowy", inny niż oryginalny układ imion
(define (wyberz-prezent imiona)
  (wyberz-losowo
   (nie-takie-same imiona (układy imiona))))
```

Przypomnij sobie podobną funkcję z ćwiczenia 12.6, która pobierała listę symboli i zwracała listę wszystkich możliwych układów złożonych z elementów tej listy.

Opracuj zewnętrzne funkcje:

- (1) *wyberz-losowo* : *lista-list-imion* -> *lista-imion*, która pobierze listę elementów i losowo wybierze jeden z nich;
- (2) *nie-takie-same* : *lista-imion lista-list-imion* -> *lista-list-imion*, która pobierze listę imion *L* oraz listę układów i zwróci listę takich układów, w których żadne imię nie występuje na takiej samej pozycji co w danej liście imion.

Dwie permutacje są ze sobą zgodne na pewnej pozycji, jeśli możemy pobrać to samo imię z obu list za pomocą słowa *first* lub takiej samej liczby słów *rest*. Np. listy (*list 'a 'b 'c*) oraz (*list 'c 'a 'b*) nie są ze sobą zgodne; natomiast listy (*list 'a 'b 'c*) i (*list 'c 'b 'a*) zgadzają się ze sobą na drugiej pozycji. Możemy to udowodnić stosując dla obu list operację *rest* poprzedzającą *first*.

Postępuj ostrożnie i zgodnie z odpowiednią metodą projektowania dla każdej klauzuli.

Wskazówka: Przypomnij sobie, że funkcja (`random n`) wybiera losową liczbę od 0 do n (patrz także ćwiczenie 11.5).

Ćwiczenie 17.13. Opracuj funkcję *prefiksDNA*. Funkcja pobierze dwa argumenty będące listami symboli (w DNA występują jedynie 'a', 'c', 'g' oraz 't', ale możemy to ograniczenie na razie pominąć). Pierwsza lista nazywana jest *wzorcem*, druga *szukanym łańcuchem*. Funkcja zwraca wartość `true`, jeśli wzorzec jest prefiksem szukanego łańcucha. We wszystkich innych przypadkach funkcja powinna zwrócić wartość `false`.

Przykłady:

```
(PrefiksDNA (list 'a 't) (list 'a 't 'c))
(not (PrefiksDNA (list 'a 't) (list 'a)))
(PrefiksDNA (list 'a 't) (list 'a 't))
(not (PrefiksDNA (list 'a 'c 'g 't) (list 'a 'g)))
(not (PrefiksDNA (list 'a 'a 'c 'c) (list 'a 'c)))
```

Jeśli to możliwe, uprość funkcję *PrefiksDNA*.

Zmodyfikuj funkcję tak, aby zwracała pierwszy element szukanego łańcucha, który nie znalazł się we wzorcu, jeśli oczywiście wzorzec jest prawidłowym prefiksem szukanego łańcucha. Jeśli listy do siebie nie pasują lub wzorzec jest dłuższy od szukanego łańcucha, zmodyfikowana funkcja powinna nadal zwracać `false`. Jeśli listy są tej samej długości i pasują do siebie, wynikiem powinno być nadal `true`.

Przykłady:

```
(symbol=? (PrefiksDNA (list 'a 't) (list 'a 't 'c))
           'c)
(not (PrefiksDNA (list 'a 't) (list 'a)))
(PrefiksDNA (list 'a 't) (list 'a 't))
```

Czy ten wariant funkcji *PrefiksDNA* może być uproszczony? Jeśli tak, zrób to. Jeśli nie, wyjaśnij dlaczego.

Rozszerzone ćwiczenie: obliczanie wyrażeń języka Scheme. Część 2.

Celem tego podrozdziału jest rozszerzenie możliwości programu stworzonego w rozdziale 14. (podrozdział „Rozszerzone ćwiczenie: obliczanie wyrażeń języka Scheme”) tak, by mógł radzić sobie z wywołaniami i definicjami funkcji. Innymi słowy, nowy program powinien symulować, co stałoby się w środowisku DrScheme, gdybyśmy wpisali dane wyrażenie w oknie *Interactions* i kliknęli przycisk *Execute*. Aby uprościć sobie zadanie, zakładamy, że wszystkie funkcje zdefiniowane w oknie *Definitions* pobierają po jednym argumentem.

Ćwiczenia

Ćwiczenie 17.14. Rozszerz definicję danych z ćwiczenia 14.17 tak, aby było możliwe reprezentowanie wywołań funkcji w wyrażeniach. Wywołanie powinno być reprezentowane jako struktura z dwoma polami. Pierwsze pole zawiera nazwę funkcji, drugie — jedną reprezentację wyrażenia będącego argumentem funkcji.

Kompletny program obliczający wartości wyrażeń powinien obsługiwać także definicje funkcji.

Ćwiczenie 17.15. Opracuj definicję struktury i definicję danych dla definicji funkcji. Przypomnij sobie, że definicja funkcji zawiera trzy istotne atrybuty:

- (1) Nazwę funkcji.
- (2) Nazwę parametru.
- (3) Ciało funkcji.

Taka charakterystyka funkcji sugeruje wprowadzenie struktury z trzema polami. Pierwsze dwa zawierają symbole, ostatni reprezentuje ciało funkcji, które jest wyrażeniem.

Przełóż następujące definicje na wartości w języku Scheme:

```
(define (f x) (+ 3 x))
(define (g x) (* 3 x))
(define (h u) (f (* 2 u)))
(define (i v) (+ (* v v) (* v v)))
(define (k w) (* (h w) (i w)))
```

Opracuj więcej przykładów i podobnie przełóż je na naszą reprezentację.

Ćwiczenie 17.16. Opracuj funkcję *interpretuj-z-jedna-definicja*. Funkcja pobierze reprezentację wyrażenia Scheme i reprezentację definicji funkcji — P .

Pozostałe wyrażenia z ćwiczenia 14.17 powinny być interpretowane jak wcześniej. W przypadku pojawienia się reprezentacji zmiennej w wyrażeniu funkcja *interpretuj-z-jedna-definicja* powinna zasygnalizować błąd. Dla wywołania funkcji P w wyrażeniu funkcja *interpretuj-z-jedna-definicja* powinna:

- (1) obliczyć wartość argumentu;
- (2) zastąpić wszystkie wystąpienia parametru funkcji P w jej ciele wartością obliczonego w poprzednim kroku argumentu; oraz
- (3) obliczyć nowe wyrażenie za pomocą rekursji. Oto szkic takiego działania:¹

```
(oblicz-z-jedna-definicja (zastąp ... ..)
  dana-definicja-funkcji)
```

Dla wszystkich innych wywołań funkcja *interpretuj-z-jedna-definicja* powinna sygnalizować błąd.

¹ Omówimy szczegółowo tę formę rekursji w części V.

Ćwiczenie 17.17. Opracuj funkcję *interpretuj-z-definicjami*. Funkcja pobierze reprezentację wyrażenia Scheme i listę reprezentacji definicji funkcji — *definicje*. Funkcja powinna zwrócić liczbę, jaką środowisko DrScheme wyświetliłoby w oknie *Interactions*, gdybyśmy wpisali wyrażenie reprezentowane przez pierwszy parametr w tym oknie, zaś okno *Definitions* zawierałoby, reprezentowane przez drugi parametr, definicje funkcji.

Pozostałe wyrażenia z ćwiczenia 14.17 powinny być interpretowane jak wcześniej. Dla wywołania funkcji *P* w wyrażeniu funkcja *interpretuj-z-definicjami* powinna:

- (1) obliczyć wartość argumentu;
- (2) znaleźć definicję odpowiedniej funkcji na liście *definicje*;
- (3) zastąpić wszystkie wystąpienia parametru funkcji *P* w jej ciele wartością obliczonego w pierwszym kroku argumentu; oraz
- (4) obliczyć nowe wyrażenie za pomocą rekursji.

Podobnie jak DrScheme, funkcja *interpretuj-z-definicjami* sygnalizuje błąd w przypadku wywołania funkcji, której nazwy nie ma na liście, oraz w przypadku odwołania się do reprezentacji zmiennej w wyrażeniu.

Równość i testowanie

Wiele opracowanych przez nas funkcji zwraca listy. Kiedy je testujemy, musimy porównywać dwie listy: wyniki zwracane przez funkcje z przewidywanymi wartościami. Porównywanie list ręcznie jest nudne i nie daje pewności, że nie popełniliśmy błędu.

Opracujmy funkcję, która pobiera dwie listy liczb i określa, czy są sobie równe:

```
;; lista=? : lista-liczb lista-liczb -> boolean
;; określa, czy dana-lista i inna-lista
;; zawierają te same liczby w tym samym porządku
(define (lista=? dana-lista inna-lista) ...)
```

Opis celu ulepsza ogólne przeznaczenie funkcji i przypomina nam, że o ile klienci mogą uważać dwie listy za równe, jeśli zawierają te same elementy, niezależnie od kolejności, to programiści są bardziej dokładni i włączają kolejność elementów jako element porównania. Kontrakt i opis celu pokazują także, że *lista=?* jest funkcją przetwarzającą dwie skomplikowane wartości — i w rzeczywistości to nas najbardziej interesuje.

Porównywanie dwóch list oznacza, że musimy przejrzeć wszystkie elementy obu list. Eliminuje to możliwość projektowania funkcji *lista=?* linia po linii, jak w przypadku funkcji *zastap-empty-lista* z podrozdziału „Jednoczesne przetwarzanie dwóch list. Przypadek 1.”. Na pierwszy rzut oka nie istnieje żaden związek pomiędzy dwiema listami wejściowymi, co sugeruje, że powinniśmy wykorzystać zmodyfikowaną metodę projektowania.

Zacznijmy więc od tabeli:

	(empty? dana-lista)	(cons? dana-lista)
(empty? inna-lista)		
(cons? inna-lista)		

Tabela ma cztery komórki do wypełnienia, co oznacza, że potrzebujemy (przynajmniej) czterech testów i czterech klauzul wyrażenia **cond** w szablonie.

Oto pięć testów:

```
(lista=? empty empty)

(not
 (lista=? empty (cons 1 empty)))

(not
 (lista=? (cons 1 empty) empty))

(lista=? (cons 1 (cons 2 (cons 3 empty)))
 (cons 1 (cons 2 (cons 3 empty))))

(not
 (lista=? (cons 1 (cons 2 (cons 3 empty)))
 (cons 1 (cons 3 empty))))
```

Drugi i trzeci pokazują, że *lista=?* musi obsługiwać swoje argumenty symetrycznie. Dwa ostatnie testy pokazują natomiast, dla jakich danych wejściowych funkcja *lista=?* powinna zwracać true lub false.

Trzy z czterech klauzul **cond** zawierają selektory, natomiast jedna z nich zawiera naturalne rekursje:

```
(define (lista=? dana-lista inna-lista)
  (cond
    [(and (empty? dana-lista) (empty? inna-lista)) ...]
    [(and (cons? dana-lista) (empty? inna-lista))
     ... (first dana-lista) ... (rest dana-lista) ...]
    [(and (empty? dana-lista) (cons? inna-lista))
     ... (first inna-lista) ... (rest inna-lista) ...]
    [(and (cons? dana-lista) (cons? inna-lista))
     ... (first dana-lista) ... (first inna-lista) ...
     ... (lista=? (rest dana-lista) (rest inna-lista)) ...
     ... (lista=? dana-lista (rest inna-lista)) ...
     ... (lista=? (rest dana-lista) inna-lista) ...]))
```

W czwartej klauzuli mamy trzy naturalne rekursje, ponieważ możemy połączyć parami dwa selektory oraz możemy połączyć w pary każdy parametr z jednym selektorem.

Od powyższego szablonu do kompletnej definicji dzieli nas jedynie niewielki krok. Dwie listy mogą zawierać te same elementy tylko wtedy, gdy obie są puste lub skonstruowane za pomocą instrukcji cons. Ten warunek natychmiast sugeruje wartość true jako odpowiedź dla pierwszej klauzuli oraz false dla dwóch następnych. W ostatniej klauzuli mamy dwie liczby będące pierwszymi elementami obu list oraz trzy naturalne rekursje. Musimy porównać te dwie liczby. Co więcej, wyrażenie *(lista=? (rest dana-lista) (rest inna-lista))* oblicza, czy reszty obu list są identyczne. Dwie listy są sobie równe wtedy i tylko wtedy, gdy spełnione są oba warunki, co oznacza, że musimy je połączyć za pomocą operatora logicznego **and**:

```
(define (lista=? dana-lista inna-lista)
  (cond
    [(and (empty? dana-lista) (empty? inna-lista)) true]
    [(and (cons? dana-lista) (empty? inna-lista)) false]
    [(and (empty? dana-lista) (cons? inna-lista)) false]
    [(and (cons? dana-lista) (cons? inna-lista))
     (and (= (first dana-lista) (first inna-lista))
           (lista=? (rest dana-lista) (rest inna-lista))))]))
```

Dwie pozostałe naturalne rekursje nie odgrywają dla naszego rozwiązania żadnej roli.

Spójrzmy po raz drugi na związki pomiędzy dwoma parametrami. Pierwszy sposób, w jaki rozwiązaliśmy problem porównywania list, sugeruje, że jeśli dwie listy mają być sobie równe, to drugi parametr musi mieć identyczny kształt jak pierwszy. Inaczej mówiąc, moglibyśmy opracować funkcję opartą na strukturze pierwszego parametru i sprawdzającą w razie potrzeby strukturę innego parametru.

Pierwszy parametr jest listą liczb, możemy więc ponownie wykorzystać szablon funkcji przetwarzających listy:

```
(define (lista=? dana-lista inna-lista)
  (cond
    [(empty? dana-lista) ...]
    [(cons? dana-lista)
     ... (first dana-lista) ... (first inna-lista) ...
     ... (lista=? (rest dana-lista) (rest inna-lista)) ...]))
```

Jedyną różnicą jest to, że druga klauzula przetwarza drugi parametr w taki sam sposób jak pierwszy. Takie rozwiązanie bardzo przypomina funkcję *godziny->wynagrodzenia* z podrödziału „Jednoczesne przetwarzanie dwóch list. Przypadek 2.”.

Wypełnienie wolnych przestrzeni w szablonie jest trudniejsze niż w pierwszej próbie rozwiązania tego problemu. Jeśli *dana-lista* jest pusta, odpowiedź zależy od drugiej danej wejściowej: *inna-lista*. Jak pokazują przykłady, odpowiedzią będzie w tym przypadku *true* wtedy i tylko wtedy, gdy *inna-lista* również będzie listą pustą. Przekładając to na język Scheme, otrzymamy następującą odpowiedź dla pierwszej klauzuli: *(empty? inna-lista)*.

Jeśli *dana-lista* nie jest pusta, szablon sugeruje, że powinniśmy obliczyć odpowiedź funkcji na podstawie:

- (1) *(first dana-lista)*, pierwsza liczba na liście *dana-lista*;
- (2) *(first inna-lista)*, pierwsza liczba na liście *inna-lista*;
- (3) *(lista=? (rest dana-lista) (rest inna-lista))*, wyrażenie, które określa, czy reszty obu list są sobie równe.

Mając dany opis celu funkcji i przykłady jej działania, możemy po prostu porównać elementy *(first dana-lista)* oraz *(first inna-lista)* i połączyć wynik z naturalną rekursją za pomocą operacji **and**:

```
(and (= (first dana-lista) (first inna-lista))
     (lista=? (rest dana-lista) (rest inna-lista)))
```

Mimo że wykonany przez nas krok wygląda na prosty, jego efektem jest niepoprawna definicja. Celem wypisania warunków w wyrażeniu **cond** jest upewnienie się, że wszystkie selektory są poprawne.

Żaden element w specyfikacji funkcji *lista=?* nie sugeruje jednak, że *inna-lista* jest skonstruowana za pomocą instrukcji *cons*, jeśli *dana-lista* jest skonstruowana za pomocą tej samej instrukcji.

Możemy poradzić sobie z tym problemem za pomocą dodatkowego warunku:

```
(define (lista=? dana-lista inna-lista)
  (cond
    [(empty? dana-lista) (empty? inna-lista)]
    [(cons? dana-lista)
     (and (cons? inna-lista)
          (and (= (first dana-lista) (first inna-lista))
               (lista=? (rest dana-lista) (rest inna-lista))))]))
```

Dodatkowym warunkiem jest *(cons? inna-lista)*, co oznacza, że *lista=?* zwróci *false*, jeśli warunek *(cons? dana-lista)* będzie prawdziwy oraz *(cons? inna-lista)* będzie fałszywy. Jak pokazują przykłady, taki właśnie powinien być pożądaný efekt.

Podsumowując, funkcja *lista=?* pokazuje, że czasami możemy zastosować więcej niż jedną metodę projektowania w celu opracowania danej funkcji. Efekty prac są różne, mimo że są ze sobą blisko powiązane; faktycznie, moglibyśmy udowodnić, że obie funkcje zwracają zawsze takie same wyniki dla tych samych danych wejściowych. Podczas drugiego procesu tworzenia funkcji wykorzystaliśmy także spostrzeżenia z pierwszej próby.

Ćwiczenia

Ćwiczenia 17.18. Przetestuj obie wersje funkcji *lista=?*.

Ćwiczenie 17.19. Uprość pierwszą wersję funkcji *lista=?*. Oznacza to, że powinieneś połączyć sąsiadujące klauzule wyrażenia **cond**, które zwracają takie same wyniki, łącząc ich warunki za pomocą operatora **or**. Jeśli to konieczne, poprzestawiaj klauzule i użyj słowa **else** w ostatniej klauzuli w ostatecznej wersji funkcji.

Ćwiczenie 17.20. Opracuj funkcję *sym-lista=?*. Funkcja określi, czy dwie listy symboli są sobie równe.

Ćwiczenie 17.21. Opracuj funkcję *zawiera-te-same-liczby*, która określi, czy dwie listy liczb zawierają te same liczby, niezależnie od ich kolejności. Np. wyrażenie:

```
(zawiera-te-same-liczby (list 1 2 3) (list 3 2 1))
```

zwróci wartość *true*.

Ćwiczenie 17.22. Klasy liczb, symboli i wartości logicznych nazywamy czasami atomami:²

² Niektórzy włączają do tej klasy także wartość *empty* i pojedyncze znaki.

atom jest albo:

- (1) liczbą;
- (2) wartością logiczną (*boolean*);
- (3) symbolem.

Opracuj funkcję *rowne-listy?*, która pobierze dwie listy atomów i określi, czy są sobie równe.

Porównanie obu wersji funkcji *lista=?* sugeruje, że druga wersja jest łatwiejsza do zrozumienia od pierwszej. Stwierdzamy w niej, że dwie złożone wartości są sobie równe, jeśli druga jest stworzona za pomocą tego samego konstruktora co pierwsza oraz jeśli elementy wykorzystane w tym konstruktorze są takie same. Ten pomysł można wykorzystać podczas opracowywania innych funkcji porównujących ze sobą dane wejściowe.

Aby potwierdzić nasze przypuszczenie, spójrzmy na funkcję porównującą strony WWW:

```
;; www=? : strona-www strona-www -> boolean
;; określa, czy dana-strona i inna-strona mają taki sam kształt drzewa
;; i zawierają te same symbole ułożone w tym samym porządku
(define (www=? dana-strona inna-strona) ...)
```

Przypomnij sobie definicję dla prostych stron WWW:

strona-WWW (w skrócie SW) jest albo:

- (1) *empty*;
- (2) (*cons s sw*), gdzie *s* jest symbolem, zaś *sw* jest stroną WWW;
- (3) (*cons esw sw*), gdzie *esw* i *sw* są stronami WWW.

Definicja danych zawiera trzy klauzule, co oznacza, że jeśli chcielibyśmy opracować funkcję *www=?* zgodnie ze zmodyfikowaną metodą projektowania, musielibyśmy przestudiować dziewięć przypadków. Wykorzystując zamiast tego doświadczenie zdobyte podczas tworzenia funkcji *lista=?*, możemy rozpocząć pracę od prostego szablonu dla stron WWW:

```
(define (www=? dana-strona inna-strona)
  (cond
    [(empty? dana-strona) ...]
    [(symbol? (first dana-strona))
     ... (first dana-strona) ... (first inna-strona) ...
     ... (www=? (rest dana-strona) (rest inna-strona)) ...]
    [else
     ... (www=? (first dana-strona) (first inna-strona)) ...
     ... (www=? (rest dana-strona) (rest inna-strona)) ...]))
```

W drugiej klauzuli wyrażenia **cond** ponownie musimy postępować podobnie jak w przypadku funkcji *godziny->wynagrodzenia* i *lista=?*. Oznacza to, że mówimy, iż *inna-strona* musi mieć taki sam kształt co *dana-strona*, jeśli ma być identyczna i mamy przetwarzać obie strony w analogiczny sposób. Rozumowanie dla drugiej klauzuli jest podobne.

W miarę ulepszania powyższego szablonu musimy znowu dodać warunki związane z parametrem *inna-strona*, aby upewnić się, że odpowiednie selektory będą działać poprawnie:

```
(define (www=? dana-strona inna-strona)
  (cond
    [(empty? dana-strona) (empty? inna-strona)]
    [(symbol? (first dana-strona))
     (and (and (cons? inna-strona) (symbol? (first inna-strona)))
           (and (symbol=? (first dana-strona) (first inna-strona))
                 (www=? (rest dana-strona) (rest inna-strona))))])
    [else
     (and (and (cons? inna-strona) (list? (first inna-strona)))
           (and (www=? (first dana-strona) (first inna-strona))
                 (www=? (rest dana-strona) (rest inna-strona))))))])
```

Musimy zwłaszcza upewnić się w drugiej i trzeciej klauzuli, że *inna-strona* jest skonstruowaną listą, oraz że jej pierwszy element jest symbolem lub listą. W przeciwnym przypadku funkcja byłaby analogiczna z funkcją *lista=?* i działałaby w identyczny sposób.

Ćwiczenia

Ćwiczenie 17.23. Narysuj tabelę opartą na definicji danych dla prostej strony WWW. Opracuj przynajmniej po jednym przykładzie dla każdego z dziewięciu przypadków. Przetestuj funkcję *www=?* dla tych przykładów.

Ćwiczenie 17.24. Opracuj funkcję *posn=?*, która pobiera dwie struktury *posn* i określa, czy są identyczne.

Ćwiczenie 17.25. Opracuj funkcję *drzewo=?*, która pobierze dwa drzewa binarne i określi, czy są identyczne.

Ćwiczenie 17.26. Przeanalizuj poniższe dwie wzajemnie rekursywne definicje danych:

s-lista jest albo:

- (1) listą pustą, *empty*, albo
- (2) (*cons s sl*), gdzie *s* jest *s-wyr*, zaś *sl* jest listą *s-lista*.

s-wyr jest albo:

- (1) liczbą,
- (2) wartością logiczną,
- (3) symbolem,
- (4) listą *s-lista*.

Opracuj funkcję *s-lista=?*, która pobiera dwie listy *s-lista* i określa, czy są identyczne. Podobnie jak w przypadku list liczb, dwie listy zgodne z definicją klasy *s-lista* są sobie równe, jeśli zawierają te same elementy na analogicznych pozycjach.

Skoro przeanalizowaliśmy już problem równości pewnych wartości, możemy wrócić do oryginalnego źródła rozważań w tym rozdziale: funkcji testujących. Przypuśćmy, że chcemy przetestować funkcję *godziny->wynagrodzenia* z podrozdziału „Jednoczesne przetwarzanie dwóch list. Przypadek 2.”:

```
(godziny->wynagrodzenia (cons 5.65 (cons 8.75 empty))
  (cons 40 (cons 30 empty)))
= (cons 226.0 (cons 262.5 empty))
```

Jeśli po prostu wpisujemy wywołanie tej funkcji w oknie *Interactions* lub dodamy je na końcu okna *Definitions*, musimy ręcznie porównać otrzymany wynik z przewidywaną wartością. Dla krótkich list, podobnych do powyższej, jest to możliwe; dla długich list, głębokich stron WWW lub innych dużych danych złożonych ręczne porównywanie może być źródłem błędów.

Korzystając z funkcji porównujących podobnych do *lista=?*, możemy znacznie zredukować konieczność ręcznego porównywania wyników testów. W naszym obecnym przypadku możemy dodać następujące wyrażenie:

```
(lista=?
  (godziny->wynagrodzenia (cons 5.65 (cons 8.75 empty))
    (cons 40 (cons 30 empty)))
  (cons 226.0 (cons 262.5 empty)))
```

na końcu okna *Definitions*. Jeśli klikniemy teraz przycisk *Execute*, musimy tylko odczytać w oknie *Interactions*, czy wszystkie podobne testy zwróciły wartość *true*.

W rzeczywistości możemy iść jeszcze dalej. Możemy napisać funkcję testującą podobną do tej z listingu 17.6. Klasa *wynik-testow* składa się z wartości i listy czterech elementów: łańcucha "złe wyniki testów:" i trzech list. Korzystając z naszej nowej zewnętrznej funkcji, możemy przetestować *godziny->wynagrodzenia* w sposób następujący:

```
(testuj-godziny->wynagrodzenia
  (cons 5.65 (cons 8.75 empty))
  (cons 40 (cons 30 empty))
  (cons 226.0 (cons 262.5 empty)))
```

Listing 17.6. Funkcja testująca

```
;; testuj-godziny->wynagrodzenia : lista-liczb lista-liczb lista-liczb -> wynik-testow
;; testuje funkcję godziny->wynagrodzenia
(define (testuj-godziny->wynagrodzenia dana-lista inna-lista oczekiwany-wynik)
  (cond
    [(lista=? (godziny->wynagrodzenia dana-lista inna-lista) oczekiwany-wynik)
     true]
    [else
     (list "złe wyniki testów:" dana-lista inna-lista oczekiwany-wynik)]))
```


Jeśli coś nie zadziała poprawnie w naszych testach, wspomniana czteroelementowa lista określi dokładnie, w którym przypadku wynik różni się od oczekiwanego.

Testowanie za pomocą równości? Twórcy języka Scheme przewidzieli konieczność stosowania ogólnej funkcji porównującej dane i udostępnili ją:

```
;; equal? : dowolna-wartosc dowolna-wartosc -> boolean  
;; określa, czy dwie wartości są strukturalnie identyczne  
;; i zawierają te same wartości atomowe na analogicznych pozycjach
```

Jeśli stosujemy `equal?` dla dwóch list, funkcja porównuje je w taki sam sposób, jak robiła to funkcja `lista=?`; kiedy dajemy na wejściu funkcji `equal?` parę struktur, to funkcja porównuje kolejno odpowiednie pola, jeśli obie dane należą do tego samego typu struktur; jeśli natomiast uruchomimy funkcję `equal?` dla dwóch danych atomowych, dane te zostaną porównane za pomocą `=`, `symbol=?` lub `boolean=?`, w zależności od ich typu.

Wskazówka dotycząca testowania

Używaj funkcji `equal?` w procesie testowania (jeśli konieczne jest porównywanie wartości).

Nieposortowane listy: W niektórych przypadkach stosujemy listy, mimo że porządek elementów nie gra roli. Ważne jest wówczas posiadanie takich funkcji, jak *zawiera-te-same-liczby* (patrz ćwiczenie 17.21), jeśli chcemy określić, czy wyniki wywołania jakiejś funkcji zawierają odpowiednie elementy.

Ćwiczenia

Ćwiczenie 17.27. Zdefiniuj, za pomocą funkcji `equal?`, funkcję testującą funkcję *zastap-empty-lista* z podrozdziału „Jednoczesne przetwarzanie dwóch list. Przypadek 1.”. Sformułuj także przykłady będące przypadkami testowymi w tej funkcji.

Ćwiczenie 17.28. Zdefiniuj funkcję *testuj-wybierz-z-listy*, która będzie zarządzać przypadkami testowymi dla funkcji *wybierz-z-listy* z podrozdziału „Jednoczesne przetwarzanie dwóch list. Przypadek 1.”. Sformułuj przykłady z rozdziału jako przypadki testowe dla funkcji *testuj-wybierz-z-listy*.

Ćwiczenie 17.29. Zdefiniuj funkcję *testuj-interpretuj*, która będzie przeprowadzać testy funkcji *interpretuj-z-definicjami* za pomocą funkcji `equal?`. Sformułuj ponownie przypadki testowe za pomocą nowej funkcji.