Przewodnik po Scheme

Zbigniew Jurkiewicz, IIUW

December 16, 2003

Przedmowa

Przewodnik oprowadza szybko „zwiedzającego” po składni i znaczeniu najbardziej użytecznych konstrukcji języka Scheme. Nie jest oficjalną dokumentacją języka ani żadnej jego implementacji. Formalną definicję języka Scheme podaje *Revised*^{5} *Report on the Algorithmic Language Scheme*.

Notacja

W dalszej części używać będziemy następujących konwencji notacyjnych:

- Tekst zapisany pismem `maszynowym` odpowiada fragmentom programów.
- Fragmenty zapisane *italikami* są parametrami, w ich miejsce można podstawić dowolne odpowiednie wyrażenia.
- Zapis "*elem* ..." oznacza zero lub więcej elementów.
- Zapis "$elem_1$ $elem_2$ ..." oznacza jeden lub więcej elementów.
- Zapis "*[ ]*" oznacza elementy opcjonalne.

1 Szybkie wprowadzenie

Scheme jest językiem programowania opartym na wyrażeniach. Wyrażenia nazywają wartości, np. `3.14` jest nazwą przybliżenia powszechnie znanej liczby.

1.1 Wartości

Część wyrażeń to wyrażenia elementarne, nazywające *wartości*. Należą do nich

- **Napisy** ujmowane w cudzysłowy (np. `"Napis"`).
- **Znaki** reprezentowane jako `#\`*nazwa-znaku*. Dla znaków drukowalnych nazwą jest po prostu pojedynczy znak (np. `#\a`, `#\A`), inne znaki mają umowne nazwy (np. `#\space`).

- **Liczby** — standard Scheme określa różne kategorie: liczby całkowite, wymierne, rzeczywiste, zespolone (np. `3+4i`) oraz specjalne (np. `+inf.0` lub `-inf.0`). W Scheme zakres liczb całkowitych nie jest ograniczony (chyba, że przez konkretną implementację).
- **Symbole**, będące *nieobliczanymi* identyfikatorami (uzyskuje się je konstrukcją `quote` lub poprzedzając identyfikator apostrofem, zob. dalej).

1.2 Identyfikatory

Wyrażeniami prostymi są również *identyfikatory*. W nazwach identyfikatorów mozna używać liter, cyfr i znaków

```
= - > < / * ?  !  +
```

Znak `?` stawia się zwykle na końcu nazwy predykatu, zaś znakiem `!` kończy się nazwy funkcji powodujących efekty uboczne — modyfikacje stanu obiektów. Znaków `->` używa się wewnątrz nazw funkcji konwersji typów (np. `number->string`), zaś znaku `:` dla symboli z pakietów/modułów do oddzielenia nazwy pakietu od właściwej nazwy (np. `primes:fast-prime?`).

1.3 Wywołania procedur

Wywołanie procedury jest wyrażeniem złożonym — ciągiem wyrażeń ujętym w nawiasy. Pierwsze wyrażenie powinno nazywać procedurę, zaś pozostałe nazywają argumenty dla tej procedury. Wartością wywołania jest wynik zastosowania procedury do argumentów, np.

```
> (+ 1 2.14)
3.14
> (+ 1 (* 2 1.07))
3.14
```

Jak widać, oba wyrażenia nazywaja tę samą wartość. W powyższym przykładzie symbole `+` i `*` oznaczały procedury dodawania i mnożenia. Formalnie wywołanie procedury ma postać

(*operator operand-1 ... operand-n*)

1.4 Definicje

Konstrukcji `define` używa się do nadania związania nazwy z pewnym obiektem Scheme, na przykład

```
(define pi 3.14159265)
```

Zdefiniowanej nazwy można używać jako identyfikatora w wyrażeniach

```
> pi
3.14159265
> (* 4 pi)
12.5663706
```

Ogólnie konstrukcja `define` ma następującą postać

```
(define nazwa wyrażenie)
```

Nazwa nie jest obliczana i powinna być identyfikatorem. Konstrukcji `define` można również używać w definicji funkcji, treści wyrażenia `let` i podobnych kontekstach, ale tylko na początku bloku. Odpowiada ona wtedy wyrażeniu `letrec`.

1.5 Definiowanie funkcji

Do definiowania funkcji służy najczęściej specjalna postać konstrukcji `define`

```
(define (nazwa parameter ...)  wyrażenie ...)
```

(nazwane funkcje można również definiować używając ogólnej postaci `define` i funkcji anonimowych).

Zdefiniujemy funkcję podnoszenia do kwadratu

```
(define (kwadrat x) (* x x))
```

Tak zdefiniowaną funkcję możemy wywołać tak samo jak funkcje systemowe

```
> (kwadrat 4)
16
```

1.6 Funkcje anonimowe

W Scheme funkcje są normalnymi obiektami danych, dlatego można je przekazywać jako argumenty i zwracać jako wartości innych funkcji. Nie muszą być nazywane, funkcje anonimowe można tworzyć konstrukcją

```
(lambda (parametr ...)  wyrażenie ...)
```

Wartością tego wyrażenia jest procedura o podanych *parametrach*, która po wywołaniu oblicza kolejno *wyrażenia* i zwraca wartość ostatniego. Na przykład wartością

```
(lambda (x) (* x x))
```

jest jednoargumentowa procedura, zwracająca kwadrat argumentu. Aby użyć lambda wyrażenia w obliczeniu zwykle umieszcza się je w pozycji operatora kombinacji, na przykład podnoszenie liczby 4 do kwadratu zapisać można jako

```
> ((lambda (x) (* x x)) 4)
16
```

Ogólnie konstrukcji tej można używać wszędzie tam, gdzie może wystąpić nazwa funkcji.

Składanie funkcji (na razie jednoargumentowych) można zdefiniować jako funkcję[1] działającą na funkcjach („funkcjonał”)

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

używając jej potem następująco

```
> ((compose kwadrat sin) 2)
0.826821810431806
> (kwadrat (sin 2))
0.826821810431806
```

1.7 Wartości Booleowskie

Wartośc prawdy jest oznaczana jako `#t`, zaś fałsz jako `#f`. Ponadto w wielu kontekstach, np. w wyrażeniach warunkowych dowolny obiekt różny od `#f` również oznacza prawdę. Nie należy tego nadużywać bez potrzeby.

1.8 Predykaty

W Scheme są zdefiniowane predykaty dla wszystkich typów danych. Ich nazwy zakończone są znakiem `?`. Predykaty te zwracają `#t` lub `#f`.

Dla żadnego obiektu nie jest równocześnie prawdziwy więcej niż jeden spośród następujących predykatów: `boolean?`, `pair?`, `symbol?`, `number?`, `char?`, `string?`, `vector?`, `procedure?`.

1.9 Wyrażenia warunkowe

Najprostszym wyrażeniem warunkowym jest

`(if` *`warunek konsekwencja [alternatywa]`*`)`

Jego wykonanie polega na obliczeniu wartości *warunku* i zależnie od tego, czy jest on spełniony, obliczenie *konsekwencj* lub *alternatywy*. Należy pamiętać, że dowolna wartość różna od `#f` desygnuje prawdę, a więc wartością `(if 0 1 2)` jest `1`.

Jeśli *alternatywa* została pominięta, a *warunek* nie jest spełniony, to wartość wyrażenia `if` nie jest określona. Takiej postaci wyrażenia `if` używa się dla warunkowych efektów ubocznych, dla czytelności lepiej jednak skorzystać wtedy z (nieco mocniejszych) wyrażeń `when` lub `unless`.

Wyrażenie `when` ma postać

[1] W realizacji MIT Scheme wprowadzono dodatkowy skrót notacyjny, pozwalający zapisać tę definicję nieco prościej

```
(define ((compose f g) x) (f (g x)))
```

`(when` *warunek* $wyrażenie_1$ $wyrażenie_2$ `...)`

Jeśli *warunek* jest spełniony, to oblicza się kolejno *wyrażenia* i zwraca wartość ostatniego z nich, w przeciwnym razie wartość wyrażenia nie jest określona.

Wyrażenie

`(unless` *warunek* $wyrażenie_1$ $wyrażenie_2$ `...)`

jest przeciwieństwem `when`. *Wyrażenia* są obliczane tylko wtedy, gdy *warunek* nie jest spełniony.

Najogólniejszym (i najstarszym) wyrażeniem warunkowym jest

`(cond (`$warunek_1$ $wyrażenie_1$ `...)`
`(`$warunek_2$ $wyrażenie_2$ `...)`
`...`
`[``(else` $wyrażenie_{else}$ `...)`*`]`*`)`

Oblicza się kolejno *warunki* tak długo, aż któryś z nich będzie spełniony. Jeśli któryś warunek będzie spełniony, to oblicza się kolejno odpowiadające mu *wyrażenia* i zwraca *ostatnio* obliczoną wartość (czyli jeśli po warunku nie następują żadne wyrażenia, to zwraca się po prostu wartość warunku).

Jeśli żaden warunek nie był spełniony, to wartość wyrażenia `cond` nie jest określona. Jako ostatniego warunku można opcjonalnie użyć słowa kluczowego `else`, oznacza ono warunek zawsze spełniony (jest to typowy lukier składniowy, ponieważ tę samą rolę doskonale spełniłby warunek `#t`, a także jakakolwiek inna stała różna od `#f`).

Wyrażenie

`(case` *klucz*
`((`$wartość_{11}$ $wartość_{12}$`...)` $wyrażenie_1$ `...)`
`((`$wartość_{21}$ $wartość_{22}$`...)` $wyrażenie_2$ `...)`
`...`
`[``(else` $wyrażenie_n$`)`*`]`*`)`

stanowi uproszczoną wersję wyrażenia `cond`. Oblicza się wartość *klucza* i kolejno porównuje z każdą z *wartościami* z kolejnych klauzul. Gdy są równe, oblicza się kolejno *wyrażenia* tej klauzuli i zwraca wartość ostatniego. Jeśli nie znaleziono pasującej *wartości*, to wartość całego wyrażenia nie jest określona. Aby tego uniknąć zamiast listy *wartości* można w ostatniej klauzuli użyć stałej składniowej `else`, pasujacej do każdego *klucza*.

1.10 Inne konstrukcje syntaktyczne

Poza wyrażeniami warunkowymi inne konstrukcje syntaktyczne to:

`(quote` *obiekt*`)` lub *`'obiekt`*

> Oba powyższe wyrażenia zwracają po prostu *obiekt* (bez obliczania jego wartości). Pozwala to zapobiegać obliczeniu stałych wyglądających jak wyrażenia, np. wyrażenia `'(1 2 3)` zwraca listę `(1 2 3)`.

`(and` *`wyrażenie ...`*`)`

Oblicza kolejno *wyrażenia* do momentu, gdy wartością któregoś będzie fałsz i wtedy zwraca `#f`. W przeciwnym razie zwraca wartość ostatniego *wyrażenia*.

`(or` *`wyrażenie ...`*`)`

Oblicza kolejno *wyrażenia* do momentu, gdy wartością któregoś nie będzie fałsz i wtedy zwraca tę wartość. W przeciwnym razie (tzn. gdy wszystkie obliczenia dały fałsz) zwraca wartość `#f`.

`(not` *`wyrażenie`*`)`

Jeśli wartością *wyrażenia* `#f`, to zwraca `#t`, w przeciwnym razie zwraca `#f` (`not` jest więc zwykłą funkcją, ale dobrze pasuje w tym miejscu ;-).

```
(let ((zmienna wyrażenie-inicjujące)
      ...)
  wyrażenie ...)
```

Oblicza *wyrażenia-inicjujące* i związuje z odpowiednimi *zmiennymi*. W tak utworzonym środowisku lokalnym oblicza kolejno *wyrażenia* i zwraca wartość ostatniego z nich.

```
(let* ((zmienna_1 wyrażenie-inicjujące_1)
       (zmienna_2 wyrażenie-inicjujące_2)
       ...)
   wyrażenie ...)
```

Podobne do `let`, ale obliczanie *wyrażeń-inicjujacych* odbywa się po kolei, przy czym obliczanie *wyrażenia*$_n$ odbywa się w środowisku, w którym poprzednie zmienne są są już zainicjowane.

```
(letrec ((nazwa wyrażenie-definiujące)
         ...)
   wyrażenie ...)
```

Oblicza *wyrażenia-definiujące* w środowisku, w którym widoczne są wszystkie (niezainicjowane) *nazwy* i wiąże je z nazwami w tym środowisku. Następnie oblicza kolejno *wyrażenia* i zwraca wartość ostatniego.

W *wyrażeniach-definiujących* mogą wystąpić odwołania do *nazw*. Musi być jednak możliwe ich obliczenie, przy założeniu, że *nazwy* te nie są związane z żadnymi wartościami. W praktyce oznacza to, że wartościami *wyrażeń-definiujących* powinny być procedury lub zbudowane z nich struktury danych (najczęściej są to po prostu lambda-wyrażenia).

`(set!` *`zmienna wyrażenie`*`)`

Przypisuje *zmiennej* wartość *wyrażenia*.

`(begin` *`wyrażenie wyrażenia ...`*`)`

Oblicza kolejno *wyrażenia* i zwraca wartość ostatniego. Przydatne do umieszczenia ciągu wyrażeń wszędzie tam, gdzie dozwolone jest tylko pojedyncze wyrażenie. Oczywiście wszystkie *wyrażenia* poza ostatnim powinny wywoływać efekty uboczne, w przeciwnym razie ich użycie nie ma sensu.

`(delay` *`wyrażenie`*`)`

Służy do leniwego obliczania. Tworzy i zwraca zamrożone *wyrażenie*. Obiekt taki może być argumentem `force`, co spowoduje obliczenie *wyrażenia*.

`(force` *`zamrożone-wyrażenie`*`)`

Jeśli *zamrożone-wyrażenie* nie było jeszcze obliczane, to jest ono obliczane, a otrzymana wartość zapamiętywana w nim i zwracana. W przeciwnym razie zwracana jest po prostu zapamiętana wartość.

1.11 Porównywanie

`(eq?` *`x y`*`)`

Zwraca `#t` jeśli wartością *x* i *y* jest ten sam obiekt.

`(equal?` *`x y`*`)`

Zwraca `#t` jeśli wartości *x* i *y* są tak samo zbudowane, co w zaadzie oznacza, że są tak samo wypisywane.

`(=` *`x y`*`)`

Tylko do porównywania liczb.

1.12 Listy i pary

Parą nazywamy obiekt zawierający dwie składowe `car` i `cdr`, nazwane tak od funkcji dostępu do nich. Parę zawieerającą liczby 3 i 5 zapisuje się jako

```
(3 .  5)
```

Lista jest ciągiem elementów, zapisywanym jako

```
(1 4 9 16 25)
```

Wyróżnionym rodzajem listy jest *lista pusta*, zapisywana jako

```
()
```

Inne listy buduje się z par w taki sposób, że `car` pary zawiera element list, zaś `cdr` pary parę zawierającą kolejny element listy. Jako `cdr` pary zawierającej ostatni element listy umieszcza się listę pustą. Tak więc listę `(2 3 5 7)` mozna również zapisać jako

```
(2 . (3 . (5 . (7 . ()))))
```

Tak więc *lista* jest w rzeczywistości albo parą albo listą pustą.

1.12.1 Operacje na parach i listach

`(cons` *`x y`*`)`

> Tworzy i zwraca parę której pierwszym elementem jest wartość *x*, zaś drugim wartość *y*. Jeśli *y* jest listą, dołącza *x* na początek listy *y*.

`(car` *`l`*`)`

> Zwraca pierwszy element pary lub listy *l*.

`(cdr` *`l`*`)`

> Jeśli *l* jest parą, to zwraca jej drugi element. Dla listy (niepustej) zwraca resztę listy (tzn. listę bez pierwszego elementu).

`(null?` *`x`*`)`

> Zwraca `#t` jeśli *x* jest listą pustą, w przeciwnym razie `#f`.

`(list` *`element`* `...)`

> Tworzy listę z podanych *elementów*.

`(list-ref` *`l k`*`)`

> Zwraca *k*-ty element listy *l*.

`(length` *`l`*`)`

> Zwraca długość (liczbę elementów) listy *l*.

`(append` *`list`* `...)`

Tworzy nową listę powstałą ze złączenia podanych *list*.

`(reverse l)`

Zwraca nową listę zawierającą elementy listy *l* w odwrotnej kolejności.

`(member obiekt l)`

Przegląda listę *l* w poszukiwaniu elementu równego (`equal?`) *obiektowi*. Zwraca podlistę („ogon *l*"), rozpoczynającą się od znalezionego elementu. Jeśli takiego elementu nie ma zwraca `#f`.

`(memq obiekt l)`

Podobna do `member`, ale używa funkcji `eq?` do porównywania.

`(set-car! para obiekt)`

Przypisuje (destrukcyjnie) pierwszemu elementowi *pary obiekt*. W przypadku listy zmienia pierwszy element listy.

`(set-cdr! para obiekt)`

Przypisuje (destrukcyjnie) drugiemu elementowi *pary obiekt*. W przypadku listy zmienia resztę listy. Jeśli *obiekt* nie jest listą, to *para* przestanie być listą!

`(append! lista ...)`

Destrukcyjnie łączy ze sobą listy, tzn. zamienia `cdr` ostatniej pary tworzącej *listę* (gdzie poprzednio było `()`) na dowiązanie do pierwszej pary następnej `listy`. Zwraca pierwszą listę (oczywiście trwale zmodyfikowaną).

1.12.2 Listy asocjacji

Listą asocjacji nazywamy listę składającą się z pozycji, będących parami postaci

(*klucz* . *wartość*)

`(assoc klucz a-lista)`

Znajduje pierwszą pozycję o podanym *kluczu* (używając `equal?` do porównywania) i zwraca ją. Jeśli takiej pozycji nie było, to zwraca `#f`.

`(assq klucz a-lista)`

Podobna do `assoc`, ale używa funkcji `eq?` do porównywania.

1.13 Symbole

Z każdym symbolem związane jest jego *lista własności*. Dla list własności określone są funkcje

`(remprop` *`symbol własność`*`)` *UW*

Usuwa podaną *własność* z listy własności podanego *symbolu*. *Własność* musi być symbolem.

```
> (putprop 'kawa 'kofeina? 'tak)
> (getprop 'kawa 'kofeina?)
TAK
> (remprop 'kawa 'kofeina?)
> (getprop 'kawa 'kofeina?)
#f
```

1.14 Liczby i operacje arytmetyczne

`(=` *`x y`*`)`
`(<` *`x y`*`)`
`(>` *`x y`*`)`
`(<=` *`x y`*`)`
`(>=` *`x y`*`)`

Klasyczne predykaty arytmetyczne.

`(max` *`x ...`*`)`
`(min` *`x ...`*`)`

Zwracają maksimum i minimum z podanych liczb.

`(gcd` *`x ...`*`)`
`(lcm` *`x ...`*`)`

Return the greatest common divisor, least common multiple of all *x*'s. The result is always non-negative.

`(+` *`x ...`*`)`
`(*` *`x ...`*`)`
`(-` *`x ...`*`)`

Klasyczne operacje arytmetyczne

`(/` *`x ...`*`)`

Dzielenie, wyniki zależy od typów argumentów, np. dla liczb całkowitych będzie to liczba wymierna lub całkowita.

```
(- x)
(/ x)
```

Szczególne przypadki poprzednich: negacja i odwrotność.

```
(expt x y)
```

Potęgowanie, podnosi Raises *x* do potęgi *y*.

```
(quotient n m)
```

Dzielenie całkowite.

```
(remainder dzielna dzielnik)
```

Reszta z dzielenia całkowitego, ma znak *dzielnej*.

```
(modulo dzielna dzielnik)
```

Reszta z dzielenia całkowitego, ma znak *dzielnika*.

```
(zero? x)
(positive? x)
(negative? x)
```

Sprawdza czy $x = 0, x > 0$ lub $x < 0$.

```
(even? x)
(odd? x)
```

Sprawdzają czy *x* jest liczbą parzystą lub nieparzystą.

```
(1+ x)
(1- x)
```

Skrótowy zapis dla `(+ x 1)` oraz `(- x 1)`

```
(abs x)
```

Wartośc bezwzględna liczby *x*.

```
(floor x)
```

Najbliższa liczbie *x* liczba całkowita i taka, że $|i| \leq |x|$.

`(ceiling x)`

Najbliższa liczbie *x* liczba całkowita i taka, że $|i| \geq |x|$.

`(round x)`

Najbliższa liczbie *x* liczba całkowita, w przypadku „remisu” wybiera liczbę parzystą.

`(sqrt x)`

Pierwiastek kwadratowy z *x*.

```
(sin x)
(cos x)
(tan x)
```

Funkcje trygonometryczne.

```
(asin x)
(acos x)
(atan x)
```

Odwrotne funkcji trygonometryczne.

`(exp x)`

Liczba niewymierna `e` podniesiona do potęgi *x*.

`(log x)`

Logarytm naturalny z *x*. `(log 0)` zwraca `-inf.0`.

1.15 Wektory

Wektory są to tablice jednowymiarowe, indeksowane zawsze od zera.

`(make-vector n [wartość-początkowa])`

Tworzy nowy wektor składający się z *n* elementów. Jeśli podano *wartość-początkową*, to wszystkie elementy są nią zainicjowane, w przeciwym razie wartości początkowe elementów nie są określone.

`(vector element ...)`

Tworzy i zwraca wektor składający się z podanych elementów.

`(vector-length` *`wektor`*`)`

Zwraca długość *vektora* (liczbę jego elementów, czyli jego rozmiar).

`(vector-ref` *`wektor i`*`)`

Zwraca *i*-ty element *vektora*.

`(vector-set!` *`wektor i w`*`)`

Przypisuje *i*-temu elementowi *wektora* wartość *w*.

1.16 Funkcjonały (funkcje wyższych rzędów

`(apply` *`procedura lista-argumentów`*`)`

Wywołuje *procedurę* od podanej *listy-argumentów*.

`(apply-if` *`warunek procedura wyrażenie`*`)` *`makro/UW`*

Najpierw oblicza się *warunek*. Jeśli wynikiem obliczenia warunku nie jest fałsz, to wywołuje się *procedurę* od tego wyniku i zwraca otrzymaną wartość. W przeciwnym razie oblicza się *wyrażenie* i zwraca jego wartość.

```
> (apply-if (assq 'c '((a b) (c d) (e f))) cadr 'niemożliwe)
d
```

`(map` *`procedura lista lista ...`*`)`

Wywołuje się *procedurę* kolejno od pierwszych elementów *list*, drugich elementów *list* itd. Otrzymane wynki zwraca się w postaci listy. *Listy* muszą być tej samej długości, ma ich być tyle, ilu argumentów oczekuje *procedura*. Uwaga: kolejność obliczeń *nie* jest określona.

`(for-each` *`procedura lista lista ...`*`)`

Podobna do `map`, ale nie zwraca żadnego wyniku. Ponieważ jest używana dla efektów ubocznych, więc wykonuje obliczenia w ustalonej kolejności.

`(filter` *`predykat l`*`)`

Zwraca listę zawierającą tylko te elementy listy *l*, dla których *predykat* był spełniony.

`(find-if` *`pred list`*`)`

Zwraca pierwszy element *listy* spełniający *predykat*.

1.17 Wejście-wyjście

`(write` *`wyrażenie`*`)` lub `(display` *`wyrażenie`*`)`

Oblicza *wyrażenie* i wypisuje wynik.

`(close-input-port` *`port`*`)`

Zamyka port wejściowy *port*. Od tego momentu nie jest możliwe pobieranie zeń dalszych znaków.

`(close-output-port` *`port`*`)`

Zamyka port wyjściowy *port*. Od tego momentu nie jest możliwe wypisywanie nań żadnych znaków.

`(fresh-line)` *`???`*

Powoduje przejście do nowej linii na bieżącym porcie wejściowym o ile nie jest on aktualnie na początku nowej linii.

`(flush-input` *`[port]`*`)` *`UW`*

Pomija wszystkie znaki czekające na wskazanym porcie wejściowym (domyślnie bieżącym). Zwykle nie ma sensu dla plików, lecz jedynie dla portów „interakcyjnych”.