1. Podstawowe Zasady

1. Przegląd funkcjonalności. Mathematica jako kalkulator.

Praktycznie wszyskie potrzebne informacje o *Mathematice* można uzyskać z Centrum Dokumentacji (aby do niego dotrzeć należy nacisnąć F1 lub użyć meni Help).

Można oczywiście używać *Mathematiki* jako zaawansowanego kalkulatora: wpisujemy formułę, naciskamy na SHIFT + ENTER (RETURN) i *Mathematica* daje nam odpowiedź.

Przykład.

```
2 + 2
```

(naciśnij SHIFT + ENTER po ustawieniu kursora w dowolnym miejscu w komórce zawierającej formułę)

```
2 + 2
```

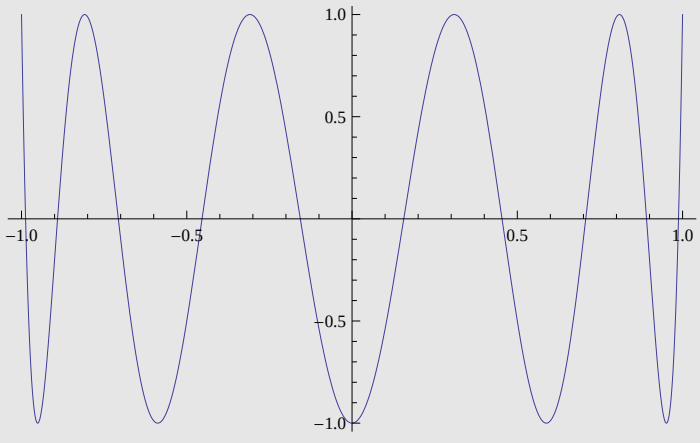
```
4
```

Widzimy tak zwaną komórkę z danymi wejściowymi (input cell) oraz komórkę z danymi wyjściowymi (output cell). W tej ostatniej jest wynik obliczenia, w tym przypadku 4. W najnowszej wersji Mathematiki (wersji 9) poniżej komórki wyjściowej pojawi się także lista sugestii następnego kroku w obliczeniach. Jest to całkiem pomocne dla początkujących ale dla bardziej zaawansowanych użytkowników może być irytujące. Naszczęście sugestie można łatwo wyłączyć używając meni “Preferences”.

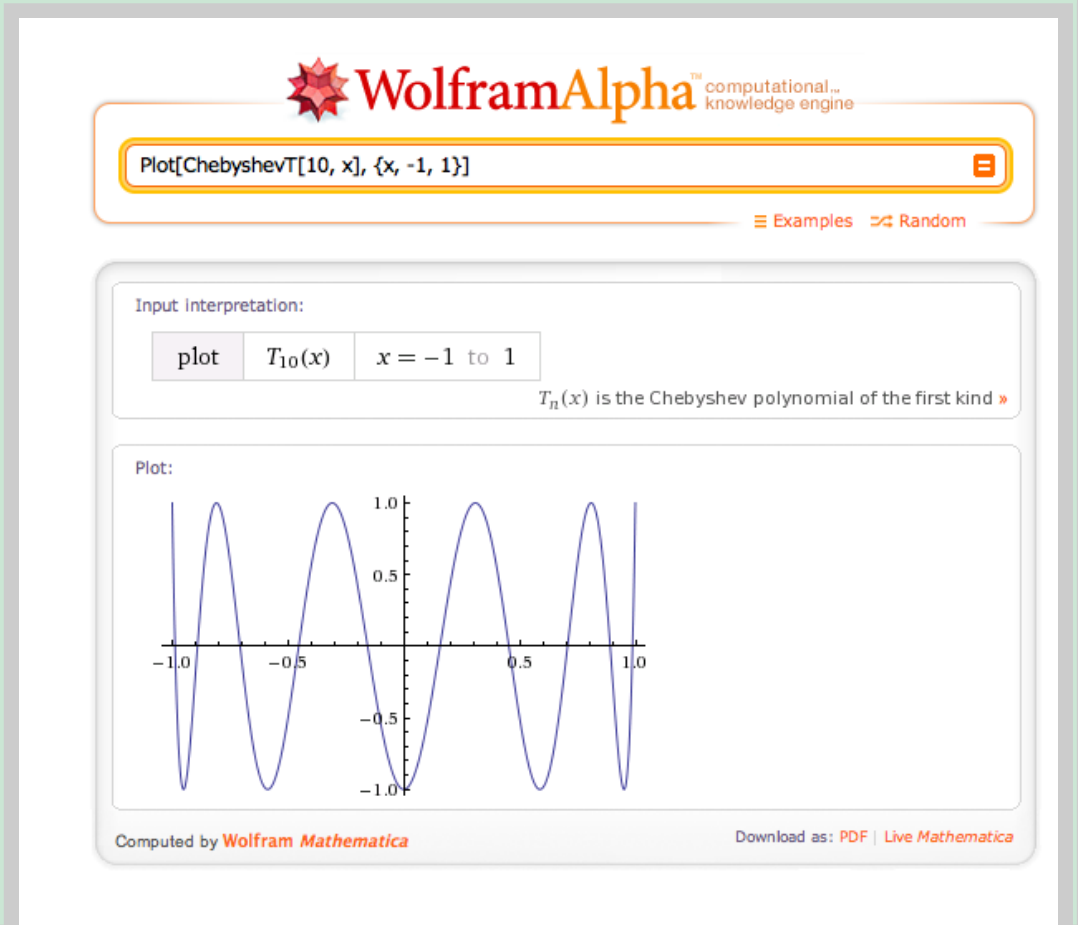
Ważną cechą *Mathematiki* jest możliwość posługiwania się nie tylko liczbami ale także formułami matematycznymi. Użytkownik *Mathematiki* dysponuję do największą na świecie kolekcją algorytmów obliczeniowych. Na przykład, *Mathematica* “zna” ogromną ilość tak zwanych funkcji specjalnych, które odgrywają ogromną rolę w matematyce i jej zastosowaniach.

Przykład. Następująca funkcja oblicza wielomian Czebyszewa dziesiątego stopnia, a następnie tworzy jego wykres nad odcinkiem [-1, 1].

```
Plot[ChebyshevT[10, x], {x, -1, 1}]
```



Ponieważ Wolfram|Alpha używa *Mathematiki,* możemy oczekiwać że coś podobnego można uzyskać używając tego programu. Zobaczmy co się stanie jeśli wpiszemy do okienka Wolfram|Alpha dokładnie tą samą instrukcję której użyliśmy powyżej:



Zwróćmy uwagę na dwie hiperlinki w prawym dolnym rogu: pobierz jako PDF i Live *Mathematica*. Pierwsza jest oczywista, druga wymaga instalacji darmowego programu CDF Player.

Pliki *Mathematiki*, zwane “notownikami” (notebooks) a także grafikę itp. sworzoną w *Mathematice* można eksporotwać do wielu formatów. *Mathematica* ma także bardzo ekstensywne możliwości importu.

Kernel i FrontEnd

Mathematica składa się z dwóch niezależnych środowisk obliczeniowych zwanych “Przyód” (FrontEnd) i “Jądro” (Kernel), które porozumiwają się z sobą za pomocą protokołu o nazwie *MathLink*. W najbardziej typowej sytuacji, użytkownik wpisuje wyrażenie do obliczenia za pomocą FrontEndu, następnie FrontEnd przekazuje je do Jądra, które wykonuje obliczenie po czym zwraca je do FrontEndu. Teoretycznie może istnieć wiele różnych FrontEndów, ale dziś w praktyce używany jest prawie wyłacznie “notatnikowy” FrontEnd (Notebook Front End), będący integralną częścią *Mathematiki* (dostępny jest także tekstowy FrontEnd, szczególnie na komputerach działających w sytemie UNIX.

Interesującym faktem jest że FrontEnd może być całkowicie kontrolowany programistycznie przez Kernel. Nie będziemy tu więcej zajmowali się tym tematem. Jest także inne możliwe podejście do programowania FrontEndu, oparte na funkcji Dynamic. Poświęcimy temu tematowi cały jeden rozdział (Rozdział IV).

Notowniki

Podstawowym pojęciem na którym opare jest funkcjonowanie FrontEndu jest tak zwany “notownik”

(notebook). “Notownik” jest interaktywnym dokumentem który może zawierać tekst, grafikę, tabele, “żywe” obliczenia i animacje i wiele innych elementów.

Notownik składa się z “komórek”, które można grupować, każda z którch może być w innym stylu itd.

Ćwiczenie. Kliknij na różne nawiasy komórek po prawej stronie tego notownika i sprawdź w jakim są one stylu.

Obok tradycyjnych notowników Wolfram Research wprowadził nowy format dokumentu zwany CDF (Computable Document Format). Dokumenty zapisane w tym formacie wyglądają identycznie jak notowniki ale mogą być otwarte przez darmowy CDF Player, ze strony Wolfram Research. W przeciwieństwie do statycznych formatów takich jak PDF, CDF może może być używany interaktywnie ponieważ CDF Player zawiera większość algorytmów używanych przez *Mathematikę*. Użytkownik CDF Playera może wykonywać obliczenia i tworzyć wizualizacje za pomocą dynamicznego interfejsu (który będzie tematem rozdziału 4). Choć obecnie pliki CDF mogą być tworzone wyłącznie przy użyciu *Mathematiki* istnieje plan stworzenia narzędzi sieciowych to tego celu.

■ Główne cechy języka *Mathematiki*

Zanim zaczniemy eksperymentownie z *Mathematicą* musimy poznać podstawowe zasady składni jej języka. Jeśli naszym celem jest używanie Mathematici wyłącznie do robienia obliczeń, grafiki itp., to w najnowszych wersjach *Mathematici* znajomość składni nie jest już aż tak konieczna jak było dawniej (można używać tzw. “Tradycyjnej Formy” (TraditionalForm), palet, a nawet “swobodnego formatu” w stylu Wolfram|Alpha) ale dla wszyskich bardziej ambitnych celów nadal jest to tak potrzebna jak znajomość gramatyki języka francuskiego do pisania poważnej książki w tym języku.

1. Jedną cechą języka Mathematiki odróżniającą go od standartowej notacji matematycznej a także większości języków programistycznych jest to że argumenty funkcji zawarte są w nawiasach kwadratowych a nie w zwykłych. Zwykłe (okrągłe) nawiasy są w *Mathematice* zarezerwowane do precyzowania kolejności operacji. Ponieważ precyzja języka programistycznego musi być większa niż języka którym posługują się na codzień matematycy, więc konieczność odróżniania tych pojęć
wydaje się być oczywista.

2. Nazwy wbudowanych funkcji zaczynają się z dużej litery. Celem tej konwencji jest uniknięcie konfliktu między nazwami funkcji zdefiniowanych przez użytkownika i nazwami funkcji wbudowanych. Oczywiście w tym celu użytkownik powinien przestrzegać zasady zaczynania nazw własnych funkcji z małej litery.

3. Mnożenie oznacza się symbolem * lub spacją. W sytuacjach w których nie ma możliwości zajścia nieporozumienia symbol mnożenia może być opuszczony.
4. Potęga jest oznaczana przez ^.

5. Liczby w tzw. “naukowej notacji” (“scientific notation”) są pisane w formie: 2.5*^-4 or 2.5 10^-4 itp.

6. W *Mathematice* przyjęta jest ogólna zasada że nazwy funkcji są pełnymi angielskimi słowmi (lub wyrażeniami składającym się z kilku słów), chyba że są to standardowe skróty matematyczne, jak, na przykład, Log na logarytm. W przypadku nazw składających się kilku słów, każde słowo zaczyna się z dużej litery, np. CharacteristicFunction. Jest kilka wyjątków; np. ślad macierzy nazwany jest Tr a nie Trace, ponieważ Trace było już wcześniej (to znaczy, we wcześniejszej wersji *Mathematici*) użyte w innym celu. Zaletą tej konwencji jest to że łatwo jest zapamiętać a nawet zgadnąć nazwy funkcji w *Mathematice.*

8. W *Mathematice* jest kilka rodzajów liczb. Po pierwsze, liczby mogą być dokładne (exact) i przybliżone (nie dokładne). Na przykład, liczby 3 i 2/3 są liczbami dokładnymi a liczba 0.66 jest liczbą przybliżoną. Natomiast Pi E czy Sqrt[2] nie są liczbami ale są symbolami numerycznymi, które także są dokładnie w tym sensie że mają “nieskończoną precyzję”. Liczby przybliżone dzielą się na dwa rodzaje, liczby z precyzją maszynową, i liczby o przedłużonej precyzji. W obliczeniach numerycznych jest bardzo ważne odróżnianie tych dwóch rodzajów liczb. Jeśli w obliczeniach wszystkie liczby są dokładne (czyli z nieskończoną precyzją) to wynik też będzie dokładny. Jeśli choć jedna liczba jest przybliżona to wynik

będzie także przybliżony. Obliczenia z precyzją maszynową są robione bez jakiejkolwiek kontroli precyzji; są więc one bardzo szybkie ale mogą dawać całkowicie błędne wyniki. Natomiast w obliczeniach z liczbami i przedłużonej precyzji *Mathematica* szacuje precyzję wyniku. Jeśli *Mathematica* twierdzi że wynik ma precyzję 0 to nie można na nim polegać i należy albo przenalizować obliczane wyrażenie metodami analizy numerycznej albo zwiększyć precyzję wszystkich liczb i wykonać je ponownie.
Funkcja N oblicza wartość przybliżoną dokładnej liczby lub symbolu numerycznego. Jeśli funkcja N jest wywołana tylko z jednym argumentem wynik ma precyzję maszynową.

```
N[Pi]
```

3.14159

```
Precision[%]
```

MachinePrecision

Natomiast forma dwu argumentowa daje wynik z precyzją określoną w drugim argumencie:

```
N[Sqrt[2], 30]
```

1.41421356237309504880168872421

```
Precision[%]
```

30.

Poniweż litera N jest zarezerwowana do powyższego celu, ona nie może być używana w nazwach funkcji lub zmiennych. To samu stosuje się do niektórych innych dużych liter, na przykład E lub C. Jest to jeszcze jeden powód dla którego warto trzymać się do konwencji według której funkcje użytkowników zaczynają się z małej litery.

Definicję i inne informacje o wbudowanej (lub nawet zdefiniownej przez użytkownika) funkcji można zobaczyć po wpisaniu znaku zapytania a po nim nazwy funkcji:

```
? FullForm
```

FullForm[*expr*] prints as the full form of *expr*, with no special syntax. ≫

```
? Part
```

expr[[*i*]] or Part[*expr*, *i*] gives the i^{th} part of *expr*.
expr[[−*i*]] counts from the end.
expr[[*i*, *j*, …]] or Part[*expr*, *i*, *j*, …] is equivalent to *expr*[[*i*]][[*j*]] … .
expr[[{i_1, i_2, … }]] gives a list of the parts i_1, i_2, … of *expr*.
expr[[*m* ;; *n*]] gives parts *m* through *n*.
expr[[*m* ;; *n* ;; *s*]] gives parts *m* through *n* in steps of *s*. ≫

Kliknięcie na “strzałki” po prawej stronie ostaniej linijki prowadzi nas do pełnej dokumentacji w Przeglądarce Pomocy.

Bardzo ważnymi strukturami danych w *Mathematice* są listy, które oznaczają się przez {*a*, *b*, *c*} lub (w FullForm) List[*a*, *b*, *c*]. Macierze i wyżej wymiarowe tensory są listami.

Naprzykład poniższe wyrażenie jest macierzą.

```
{{1, 2, 3}, {2, 3, 4}}
```

```
{{1, 2, 3}, {2, 3, 4}}
```

Nie wygląda one wprawdzie jak macierz w książkach matematycznych, ale można temu zaradzić, i nawet jest na to kilka sposobów. W nowoczesnych wersjach *Mathematiki* istnieje możliwość pokazania każdej formuły matematycznej w tradycyjnej formie matematycznej. Można nawet tak ustawić preferencje że każdy wynik będzie automatycznie pokazywany w ten sposób. (Można nawet używać tradycyjnej formy w formułach wejściowych, choć są w tym pewne niebezpieczeństwa więc nie będziemy tej metody rozważali.) Żeby zobaczyć macież {{*a*, *b*}, {*c*, *d*}} w tradycyjnej formie należy albo wybrać tekst i użyć meniu Cell:ConvertTo:Traditional Form albo użyć instrukcji:

```
TraditionalForm[{{a, b}, {c, d}}]
```

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Jeśli usawimy w preferencjach (*Mathematica*:Preferences:Format Type of New Output Cells) TraditionalForm jako domyślny format wyjściowy to macierze zawsze będą wyglądały jak macierze (a nie listy list), i wszystkie formuły matematyczne też będą wyglądały jak w książkach (z kilkoma, mało znaczącymi wyjątkami). Zobaczmy jak to działa na przykładzie:

Przyklad.

```
Integrate[Sin[Sqrt[x]] / x, {x, 0, Infinity}]
```

$$\int_0^\infty \frac{\sin\left(\sqrt{x}\right)}{x}\, dx$$

Tak wygląda formuła w TraditionalForm - czyli w standartowej notacji matematycznej. Formułę tą można wpisać za pomocą palet lub używając specjalnych kombinacji klawiszów, ale wygodniej jest wpisać ją w InputForm:

```
Integrate[Sin[Sqrt[x]] / x, {x, 0, Infinity}]
```

Nie wygląda to atrakcyjnie ale zawiera tylko symbole ASCII, więc
nadaje się, na przykład, do wysyłania przez e-mail. Zauważmy że do pewnych celów programistycznych InputForm nie jest wystarczająco dokładną reprezentacją wyrażenia i w nie których sytuacjach (do których jeszcze wrócimy) konieczne jest użycie FullForm:

```
FullForm[Hold[Integrate[Sin[Sqrt[x]] / x, {x, 0, Infinity}]]]
```

```
Hold[Integrate[Times[Sin[Sqrt[x]], Power[x, -1]], List[x, 0, Infinity]]]
```

(Musieliśmy użyć funkcji Hold bo inaczej *Mathematica* obliczyłaby całkę i otrzymalibyśmy odpowiedź π).

Kompromisem pomiędzy TranitionalForm i InputForm jest StandardForm, która w tym przypadku wygląda tak:

$$\int_0^{\infty} \frac{\text{Sin}\left[\sqrt{x}\right]}{x}\, dx$$

Widzimy że StandardForm jest bliskie tradycynej notacji, ale z pewnymi wyraźnymi różnicami, w tym przypadku: kwadratowe nawiasy funkcji i nazwa funkcji Sin zaczynająca się z dużej litery.

W tym miejscu warto wspomnieć o jeszcze jednym rodzaju “form” które istnieją w *Mathematice.* Celem tych form jest uproszczenie zapisu i uniknięcie konieczności pisanią wielu nawiasów. Na przykład, zamiast $f[x]$ można pisac $f@x$ (prefix form) lub $x//f$ - postfix form. Dla funckji wielu argumentów istnieje także infix form: $f[x, y]$ można zapisywać jako $x \sim f \sim y$. Więcej przyk

```
Sin[π]
```

```
0
```

```
First[{a, b, c}]
```

```
a
```

(zarówno wejście jak i wyjście) są przykładami wyrażeń. Te wyrażnia wyglądają albo jak standartowe formuły matematyczne albo jak naturalne operacje na listach itp, co bardzo ułatwie szybkie ich zrozumienie przez ludzi. Tym niemniej, format w którym są zapisane nie jest wystarczająco precyzyjny dla celów programistycznych. Z tego powodu istnieje jeszcze jeden format, zwany FullForm, który jest zbyt skomplikowany do codiennych “ludzkich” potrzeb ale jest używany “wewnętrznie” przez *Mathematikę* i ktorego znajomość jest bardzo przydatna w programowaniu.

FullForm wyrażeń

Każde wyrażenie jest albo atomem albo ma formę

```
F[a1, a2, ..., an]
```

gdzie F jest tak zwaną Head (Głową) wyrażenia, a a1, a2, są wyrażeniami. Przykładami atomów są 2,a,3/4,3.2, "kot". Możemy sprawdzić czy coś jest atomem za pomocą funkcji AtomQ:

```
AtomQ[2]
```

```
True
```

```
AtomQ[{2, 3}]
```

```
False
```

Wyrażenę często bardzo rożni się od swojej FullForm, fna przykład a+b ma FullForm:

```
FullForm[a + b]
```

```
Plus[a, b]
```

```
Head[a + b]
```

```
Plus
```

```
Head[a]
```

```
Symbol
```

```
Head[2]
```

```
Integer
```

```
FullForm[{a, b}]
```

```
List[a, b]
```

```
Head[{a, b}]
```

```
List
```

Oczywiście atomy także mają Head, ale jest ona czymś innym (odpowiada “typowi” w innych językach programistycznych). Na przykład:

```
Head[2]
```

```
Integer
```

```
Head["cat"]
```

```
String
```

```
Head[cat]
```

```
Symbol
```

Zauważmy także że:

```
Head[x]
```

```
Symbol
```

Oczywiście, jeśli przypiszemy *x* jakąś wartość to sytuacja się zmieni:

```
x = 1;
```

```
Head[x]
```

```
Integer
```

Zanim funkcja Head miała szansę zadziałać, nastąpiła ewaluacja *x*, więc otrzymaliśmy Head[1] czyli Integer. Aby otrzymać prawdziwą Head musimy nie dopuścić do ewaluacji, np. używając funkcji Unevaluated.

```
Head[Unevaluated[x]]
```

```
Symbol
```

Z tego samego powodu, jeśli chcielibyśmy sprawdzić FullForm wyrażnia artymetycznego, na przykład, 2+3, musimy powstrzymać ewaluację.

```
FullForm[Hold[2 + 3]]
```

```
Hold[Plus[2, 3]]
```

```
ReleaseHold[%]
```

```
5
```

FullForm wyrażenia

```
x = 1
```

jest

```
FullForm[Hold[x = 1]]
```

```
Hold[Set[x, 1]]
```

```
Clear[x]
```

Tu należy zwrócić uwagę na różnicę między przypisaniem = (Set) i równością == (Equal)

```
FullForm[Hold[x == 1]]
```

```
Hold[Equal[x, 1]]
```

Części wyrażeń

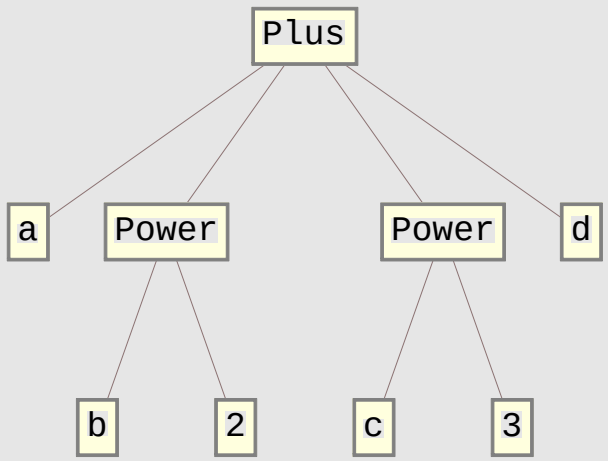
Bardzo ważną techniką programistyczną jest manipulowanie częściami wyrażeń. Każde wyrażnie może być przedstawione jako “drzewo” przy pomocy TreeForm, na przykład:

$$g = a + b^2 + c^3 + d;$$

```
FullForm[g]
```

```
Plus[a, Power[b, 2], Power[c, 3], d]
```

```
TreeForm[g]
```



Drzewo składa się z obiektów na różnych poziomach. W tym przypadku, objekty na pozionach jeden i dwa to:

```
Level[g, {1}]
```

$\{a, b^2, c^3, d\}$

```
Level[g, {2}]
```

{b, 2, c, 3}

```
Level[g, {1, 2}]
```

$\{a, b, 2, b^2, c, 3, c^3, d\}$

Każde wyrażnie ma także części. Częścią zerową jest jego Head.

```
Part[g, 0]
```

Plus

Nestępne cześci to (po kolei) wyrażenia po kwadratowym nawiasie. Tym razem użyjemy InputForm:

```
g[[1]]
```

a

```
g[[2]]
```

b^2

```
g[[3]]
```

c^3

W tym wyrażeniu trzecia część nie ma trzeciej części. *Mathematica* ostrzega nas o tym i zwraca (zgodnie z ogólna zasadą w podobnych sytuacjach) formułę wejściową (ale po dokonaniu ewaluacji, a więc zamiast *g* mamy jego wartość $a + b^2 + c^3 + d$).

```
g[[3, 3]]
```

Part::partw : Part 3 of c^3 does not exist. ≫

$(a + b^2 + c^3 + d)$ ⟦3, 3⟧

```
g[[2, 2]]
```

2

```
g[[2, 0]]
```

Power

Można oczywiście wybierać elementy “od końca”:

```
g[[-2, 1]]
```

c

a można brać także pewną rozpiętość (Span), np. od 2 pozycji do 4:

```
g[[2 ;; 4]]
```

$b^2 + c^3 + d$

A teraz bardzo przyjemny i pożyteczny fakt: można zmieniać wyrażenia przez “przypisanie” nowych wartości ich częściom. Na przykład:

```
g[[1]] = x + y;
```

```
g
```

$b^2 + c^3 + d + x + y$

```
g[[3 ;; 4]] = z; g
```

$b^2 + x + y + 2\,z$

Listy, Wektory, Macierze i Tensory

W *Mathematice* listy są, po prostu, wyrażeniami których Head jest List:

```
m = {a, b, c, d};
```

```
Length[m]
```

```
4
```

```
List[a, b, c, d]
```

```
{a, b, c, d}
```

Macierz jest, po prostu, listą list o tej samej długości

```
mat = {{a, b, d}, {c, d, e}}
```

$$\begin{pmatrix} a & b & d \\ c & d & e \end{pmatrix}$$

```
mat[[1, 1]]
```

```
a
```

```
mat[[2, 2]]
```

```
d
```

```
mat[[All, 1]]
```

```
{a, c}
```

```
mat[[All, 2]]
```

```
{b, d}
```

```
mat[[1, All]]
```

```
{a, b}
```

```
mat[[2, All]]
```

```
{c, d}
```

Później zobaczymy jak łatwo tworzy się macierze dowolnego rozmiaru za pomosą funkcji Table i Array

```
Table[i^2, {i, 1, 10}]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

```
Table[i * j, {i, 1, 5}, {j, 1, 5}]
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \\ 5 & 10 & 15 & 20 & 25 \end{pmatrix}$$

Uwaga o formach wyrażeń.

Jak już wiemy, wyrażenia *Mathematiki* wyglądają inaczej "w ludzkich oczach" od ich formy wewnętrzej (FullForm). Jednakże, sytuacja komplikuje się jeszcze bardziej z powodu faktu że tradycyjna notacja matematyczna nie jest całkowicie jednoznaczna. Z tego powodu (a także w celu utrzymania kompatybilności ze starszymi wersjami) *Mathematica* ma kilka from wejścia (input) i wyjścia (output). We wczesnych wersjach *Mathematiki* były tylko dwie formy: InputForm, oparta na sposobie pisania formuł matematycznych w popularnych językach programistycznych jak Fortran oraz OutputForm, nieco bardziej przypominająca standartową notację matematyczną. Ta forma jest dziś zupełnie przestarzała i istnieje w celu zachowania kompatybilonści. InputForm (która używa wyłącznie charakterów ASCII) jest nadal czasem używana, ale w praktyce używa się głównie StandardForm lub TradtionalForm. Domyślne ustawienie *Mathematiki* używa StandardForm zarówno dla formy wejściowej jaki wyjściowej ale to ustawinie można zmienić w Preferencjach. Używając TradtionalForm jako formy wejściowej należy jednak zachować pewną ostrożność, i z tego powodu nie jest to podejście rekomendowane przez Wolfram Research. Często, szczególnie w prezentacjach "na żywo" dla osób nie obeznanych z Mathematiką, wygodne jest ustawienie Standardowej formy wejsciowej i Tradycyjnej wyjściowej.

Podstawowe zasady składni *Mathematiki* w InputForm oraz w StandardForm.

1. Nazwy wszystkich wbudowanych funkcji zaczynają się od dużej litery.

2. Argumenty funkcji są otoczone kwadratowymi nawiasami.

3. (InputForm) Podsawowe operacje artymetyczne są oznaczane jako + (dodawanie),* or spacja (mnożenie) / (dzielenie), ^ (potęgowanie).

4. Zachodzą następujące ostre relacje włożenia InputForm ⊂ StandardForm ⊂ TraditionalForm

Links

http://reference.wolfram.com/Mathematica/guide/Expressions.html

`http://reference.wolfram.com/mathematica/tutorial/FormsOfInputAndOutput.html`

2. Operacje na Listach

Jednym z najcześciej pojawiających się wyrażen w *Mathematice* jest lista. Listy pojawiają się w bardzo

wielu kontekstach. Na przykład, rozwiązania równania lub systemu równań są zwracane przez funkcję Solve w formie listy list, których elementami są “reguły”.

```
Solve[x^3 == 1, x]
```

```
{{x → 1}, {x → -(-1)^(1/3)}, {x → (-1)^(2/3)}}
```

```
% // N
```

```
{{x → 1.}, {x → -0.5 - 0.866025 i}, {x → -0.5 + 0.866025 i}}
```

Wiele wbudowanych funkcji ma słowo lista w nazwie. Możemy nawet skonstruować listę wszystkich takich funkcji:

```
Select[Names["System`*"],
 StringMatchQ[#, StringExpression[___, "List", ___]] &]
```

```
{AdjacencyList, BinaryReadList, BinLists, ButtonStyleMenuListing,
 CoefficientList, CompletionsListPacket, ComposeList, CounterStyleMenuListing,
 DateList, DateListLogPlot, DateListPlot, DMSList, EdgeList, FactorList,
 FactorSquareFreeList, FactorTermsList, FindList, FixedPointList, FoldList,
 HistogramList, IncidenceList, List, Listable, ListAnimate, ListContourPlot,
 ListContourPlot3D, ListConvolve, ListCorrelate, ListCurvePathPlot,
 ListDeconvolve, ListDensityPlot, Listen, ListFourierSequenceTransform,
 ListInterpolation, ListLineIntegralConvolutionPlot, ListLinePlot,
 ListLogLinearPlot, ListLogLogPlot, ListLogPlot, ListPicker, ListPickerBox,
 ListPickerBoxBackground, ListPickerBoxOptions, ListPlay, ListPlot,
 ListPlot3D, ListPointPlot3D, ListPolarPlot, ListQ, ListStreamDensityPlot,
 ListStreamPlot, ListSurfacePlot3D, ListVectorDensityPlot,
 ListVectorPlot, ListVectorPlot3D, ListZTransform, MessageList,
 MonomialList, NestList, NestWhileList, ParentList, PermutationList,
 PermutationListQ, PowerModList, PropertyList, ReadList, RecordLists,
 ReplaceList, SampledSoundList, SingularValueList, StringReplaceList,
 StyleMenuListing, TrigFactorList, VertexList, WaveletListPlot, $MessageList}
```

Na przykład, funkcja CoefficientList daje listę współczynników wielomianu

```
CoefficientList[a x^2 + b x + c, x]
```

```
{c, b, a}
```

Funkcja Options daje listę reguł przypisujących opcjom danej funkcji ich wartości.

```
Options[Plot]
```

$$\left\{\text{AlignmentPoint} \to \text{Center}, \text{AspectRatio} \to \frac{1}{\text{GoldenRatio}}, \text{Axes} \to \text{True},\right.$$
AxesLabel → None, AxesOrigin → Automatic, AxesStyle → {}, Background → None,
BaselinePosition → Automatic, BaseStyle → {}, ClippingStyle → None,
ColorFunction → Automatic, ColorFunctionScaling → True,
ColorOutput → Automatic, ContentSelectable → Automatic,
CoordinatesToolOptions → Automatic, DisplayFunction :→ $DisplayFunction,
Epilog → {}, Evaluated → Automatic, EvaluationMonitor → None,
Exclusions → Automatic, ExclusionsStyle → None, Filling → None,
FillingStyle → Automatic, FormatType :→ TraditionalForm, Frame → False,
FrameLabel → None, FrameStyle → {}, FrameTicks → Automatic,
FrameTicksStyle → {}, GridLines → None, GridLinesStyle → {},
ImageMargins → 0., ImagePadding → All, ImageSize → Automatic,
ImageSizeRaw → Automatic, LabelStyle → {}, MaxRecursion → Automatic,
Mesh → None, MeshFunctions → {#1 &}, MeshShading → None, MeshStyle → Automatic,
Method → Automatic, PerformanceGoal :→ $PerformanceGoal,
PlotLabel → None, PlotPoints → Automatic, PlotRange → {Full, Automatic},
PlotRangeClipping → True, PlotRangePadding → Automatic, PlotRegion → Automatic,
PlotStyle → Automatic, PreserveImageOptions → Automatic, Prolog → {},
RegionFunction → (True &), RotateLabel → True, Ticks → Automatic,
$\left.\text{TicksStyle} \to \{\}, \text{WorkingPrecision} \to \text{MachinePrecision}\right\}$

Przypomnijmy że macierze też są listami:

```
{{1, 2, 3}, {2, 3, 4}, {34, 5, 3}} // TraditionalForm
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 34 & 5 & 3 \end{pmatrix}$$

Zajmimy się teraz generowaniem list i operacjami na listach. Najbardziej pożyteczną funkcją do generowania list jest Table.

```
Table[{i, i^2 + 2}, {i, -1, 3}] // TraditionalForm
```

$$\begin{pmatrix} -1 & 3 \\ 0 & 2 \\ 1 & 3 \\ 2 & 6 \\ 3 & 11 \end{pmatrix}$$

Wynik można przedstawić w różnych formach

```
Grid[Table[{i, i^2 + 2}, {i, -1, 2}]]
```

```
-1  3
 0  2
 1  3
 2  6
```

```
Grid[Table[{i, i^2 + 2}, {i, -1, 2}], Frame → All]
```

−1	3
0	2
1	3
2	6

Można, oczywiście generować nie tylko listy liczb ale ogólnych wyrażeń:

```
Array[a, 3]
```

```
{a[1], a[2], a[3]}
```

Niektóre często używanie macieże są już zdefiniowane w *Mathematice*. Na przykład, macierz identycznościowa:

```
IdentityMatrix[3] // TraditionalForm
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Lub ogólniej:

```
DiagonalMatrix[{a, b, c, d}] // TraditionalForm
```

$$\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & d \end{pmatrix}$$

Mathematica ma wiele wbudownych operacji związanych z macierzami, na przykład:

```
Eigenvalues[{{1, 2}, {3, 2}}]
```

```
{4, -1}
```

```
Eigenvectors[{{1, 2}, {3, 2}}]
```

```
{{2, 3}, {-1, 1}}
```

Na listach można wykonywać wszystkie podstawowe operacje artytmetyczne (jest to dużo bardziej wydajne podejście od używania pętli). Jest to związane z tym że wszystkie operacje arytmetyczne mają atrybut “Listable”, co będzie wytłumaczone później. Zauważmy kilka przykładów:

```
{1, 2, 3} + {1, 2, 3}
```

```
{2, 4, 6}
```

```
{1, 2, 3} + 1
```

```
{2, 3, 4}
```

Kropką ozancza się iloczyn skalarny wektorów a także mnożenie macierzy:

```
{a, b, c}.{s, d, f}
```

```
b d + c f + a s
```

```
({{a, b}, {c, d}}).({{e, f}, {g, h}})
```

$$\begin{pmatrix} a e+b g & a f+b h \\ c e+d g & c f+d h \end{pmatrix}$$

Ta konwencja może wydawać się nie zgodna z przyjętą konwencją w matematyce że iloczyn macierzy jest oznaczany tak samo jak iloczyn liczb, ale matematycznie jest to uzasadnione tym że iloczyn macierzy jest naturalnym uogólnieniem iloczyny skalarnego wektorów. Poza tym, z powodu wyżej wspomnianego atrybutu Listable mnożenia, “zwyczajne” mnożenie macierzy daje odmienny wynik:

```
{{a, b}, {c, d}} {{e, f}, {g, h}}
```

$$\begin{pmatrix} a e & b f \\ c g & d h \end{pmatrix}$$

Jest wiele innych naturalnych operacji na listach:

```
Prepend[{a, b, c}, d]
```

```
{d, a, b, c}
```

```
Append[{a, b, c}, d]
```

```
{a, b, c, d}
```

```
Union[{a, b, c}, {a, b, d}]
```

```
{a, b, c, d}
```

```
Join[{a, b, c}, {a, b, d}]
```

```
{a, b, c, a, b, d}
```

```
Take[{a, b, c, e}, 2]
```

```
{a, b}
```

Jest wiele innych operacji, takich jak Insert, Delete, itp. których nazwy same sugerują ich funkcję. Większość z nich można stosować do ogólnych wyrażeń, nie tylko list. Na przykład, funkcja Join która łączy listy

```
Join[{a, b, c}, {d, e, f}]
```

```
{a, b, c, d, e, f}
```

działa także na ogólnych wyrażeniach, na przykład:

```
Join[a^b, c^d]
```

$$a^{b^{c^d}}$$

Ponieważ listy mogą być zagęszczane, naturalnie mogą także być “splaszczane”. Służy temu funkcja Flatten:

```
?Flatten
```

Flatten[*list*] flattens out nested lists.
Flatten[*list*, *n*] flattens to level *n*.
Flatten[*list*, *n*, *h*] flattens subexpressions with head *h*.
Flatten[*list*, {{s_{11}, s_{12}, … }, {s_{21}, s_{22}, … }, … }]
flattens *list* by combining all levels s_{ij} to make each level *i* in the result. ≫

```
ls = {{c, a, b}, {a, f, g}, {g, a}} // Flatten
```

```
{c, a, b, a, f, g, g, a}
```

Aby pozbyć się elementów powtarzających się bez sortowania listy można użyć funkcji DeleteDuplicates:

```
DeleteDuplicates[ls]
```

```
{c, a, b, f, g}
```

Podobny efekt da funkcja Union, z tym że ta funkcja sortuje (porządkuje) listę w kanonicznym porządku.

Jest wiele konstukcji które z danej listy tworzą związane z nią listy, jak na przykład listę wszystkich premutacji, krotek i podzbiorów:

```
Permutations[{a, b, c}]
```

```
{{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}}
```

```
Tuples[{0, 1}, 3]
```

```
{{0, 0, 0}, {0, 0, 1}, {0, 1, 0},
 {0, 1, 1}, {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}}
```

```
Subsets[{a, b, c, d}]
```

```
{{}, {a}, {b}, {c}, {d}, {a, b}, {a, c}, {a, d}, {b, c}, {b, d},
 {c, d}, {a, b, c}, {a, b, d}, {a, c, d}, {b, c, d}, {a, b, c, d}}
```

Apply and Map

W programowniu funkcyjnym bardzo ważną rolę odgrywają funkcje których argumentami mogą być inne funkcje. Dwa najważniejsze pszykłady to *Apply* i *Map*.

```
?Apply
```

Apply[*f*, *expr*] or *f* @@ *expr* replaces the head of *expr* by *f*.
Apply[*f*, *expr*, *levelspec*] replaces heads in parts of *expr* specified by *levelspec*. ≫

```
FullForm[{a, b, c}]
```

```
List[a, b, c]
```

Zamieniamy listę na iloczyn jej elementów. Wystarczy użyć Apply i zmienić Head wyrażenia z List na Times:

```
Times @@ {a, b, c}
```

```
a b c
```

```
FullForm[a b c]
```

```
Times[a, b, c]
```

Teraz zamienimy iloczyn wyrażen na ich sumę:

```
List @@ (a b c)
```

```
{a, b, c}
```

A tu mamy przykład użycia funkcji Subscript w celu stowrzenia listy indeksowanych wyrażeń

```
Subscript[a, #] & /@ Table[i, {i, 1, 10}]
```

$$\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}\}$$

Powyżej /@ jest formą prefiksową funkcji Map.

```
? Map
```

Map[*f*, *expr*] or *f* /@ *expr* applies *f* to each element on the first level in *expr*.
Map[*f*, *expr*, *levelspec*] applies *f* to parts of *expr* specified by *levelspec*. ≫

Najczęściej spotykane użycie Map to zastosowanie funkcji do elementów wyrażenia, które może, ale nie musi, być listą. Na przykład, użyjemy funkcji podnoszącej do kwadratu i listy liczb naturalnych od 1 do 10:

```
Function[x, x^2] /@ Range[10]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

W tym przypadku użycie Map jest nie optymalne, bo można uzyskać ten sam wynik szybciej i prościej przez:

```
Range[10]^2
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Ale map można użyć z bardziej skomplikowaną funkcją, na przykład, funkcją która podnosi do kwadratu liczby parzyste a do sześcianu nieparzyste:

```
Function[x, If[EvenQ[x], x^2, x^3]] /@ Range[10]
```

```
{1, 4, 27, 16, 125, 36, 343, 64, 729, 100}
```

Funkcja Map działa także na dowlonych wyrażeniach:

```
Map[Function[x, x^2], x + y]
```

$$x^2 + y^2$$

Można ją także stosowań na różnych poziomach wyrażenia. Domyślnie działa na poziomie 1:

```
Map[Function[x, x^2], Sin[x] + Cos[y]]
```

```
Cos[y]^2 + Sin[x]^2
```

Tu działa na poziomie 2

```
Map[Function[x, x^2], Sin[x] + Cos[y], {2}]
```

```
Cos[y^2] + Sin[x^2]
```

Ewaluacja

Bardzo ważnym pojęciem w programowaniu w Mathematice jest ewaluacja czyli obliczenie wartości wyrażenia. Oblicznie wartości wyrażenia zawartego w komórce występuje po umieszczeniu kursora w dowolnym miejscu komórki i naciśnięciu klawiszy (dwóch na raz) Shift + Enter. Oblicznie wartości wyrażenia zawartego w komórce występuje po umieszczeniu kursora w
dowolnym miejscu komórki i naciśnięciu klawiszy
Shift + Enter. Można także obliczyć wartości wyrażeń w kilku wybranych komórkach albo w całym notowniku. Proces ewaluacji wyrażenia w Mathematice jest skomplikowany i składa się z kroków wykonywanych po kolei w określonym porządku aż wyrażenie przestanie się zmieniać. Ta końcowa forma wyrażenia jest wynikiem ewaluacji. Zrozumienie procesu ewaluacji wyrażeń jest bardzo ważnym aspektem programowania w *Mathematice.* Jednym z najczęściej spotykanych zjawisk związanych z procesem ewaluacji jest otrzymanie wyrażenia wyjściowego. Zazwyczaj oznacza to że *Mathematica* nie zna żadnych reguł ewaluacji które prowadzą do zmiany wyrażenia. Tak będzie, na przykład, jeśli damy *Mathematice* do porównania dwa niezdefiniowane symbole:

```
Clear[x, y]
```

```
x + 1 == y
```

```
1 + x == y
```

Podobny wynik otrzymamy jeśli każemy *Mathematice* rozwiązać równanie którego *Mathematica* rozwiązać nie potrafi, np

```
Solve[x 3^x - x Sin[x] - 1 == 0, x]
```

Solve::nsmet : This system cannot be solved with the methods available to Solve. ≫

```
Solve[-1 + 3^x x - x Sin[x] == 0, x]
```

Próba ewaluacji zajęła tym razem długi czas, ale *Mathematica* nie znalazła żadnej metody rozwiązania tego równinia, o czym zostaliśmy poinformowani, po czym zwróciła nam wyrażenie wejściowe. Czasem, jednak, kiedy wydaje się że w czasie ewaluacji nic nie zaszło, prawda wygląda inaczej. Jednym takim przykładem jest:

```
2/3
```

$$\frac{2}{3}$$

Wygląda jakby nic nie zaszło, ale w tym przypadku jest to tylko iluzja, spowodowana faktem że *Mathematica* czasem reprezentuje różne wyrażenia w ten sam sposób. Przypominamy, że jeśli chcemy zobaczyć jak dane wyrażenie jest reprezentowane “wewnętrznie” musimy dotrzeć do jego FullForm. W tym przypadku, FullForm wyrażenia wejściowego i wyjściowego jest odmienna. Aby móc to stwierdzić musimy zapobiec ewaluacji zanim użyjemy FullForm:

```
FullForm[Hold[2/3]]
```

```
Hold[Times[2, Power[3, -1]]]
```

Czyli przed ewaluacją FullForm wyrażenia jest Times[2,Power[3,-1]]. Natomiast po ewaluacji:

```
FullForm[2 / 3]
```

```
Rational[2, 3]
```

Widzimy że w czasie ewaluacji coś jednak zaszło: wyrażenie Times[2,Power[3,-1]] zostało zamienione na Rational[2,3].

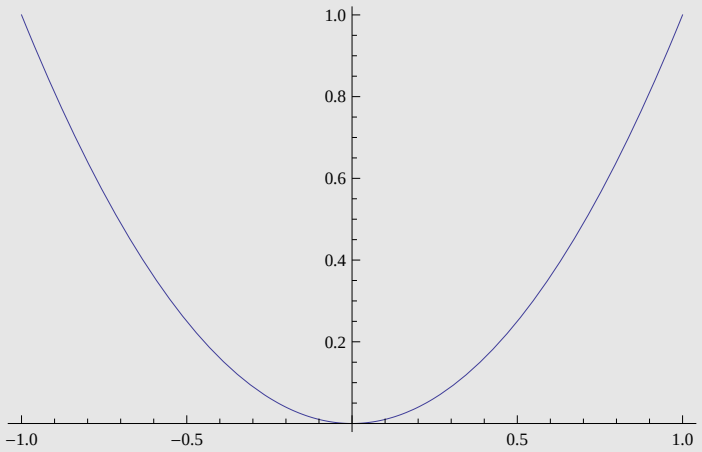
Ćwiczenie:

Opisz proces ewaluacji $1+i$ oraz $\frac{1+i}{2}$.

FullForm obiektów graficznych

Związek między FullForm, InputFrom, StandardForm i TraitionalForm jest szczególnie interesujący w przypadku obiektów graficznych.

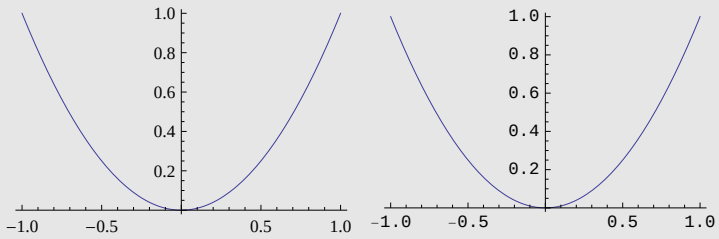
```
gr = Plot[x^2, {x, -1, 1}]
```




```
Short[InputForm[gr], 5]
```

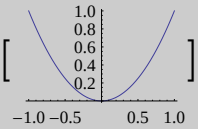
```
Graphics[{{{}, {},
   {Hue[0.67, 0.6, 0.6], Line[{{-0.9999999591836735, 0.9999999183673486},
      << 272 >>, {0.9999999591836735, << 1 >>}}]}}}, {<< 6 >>}]
```

```
Grid[{{TraditionalForm[gr], StandardForm[gr]}}]
```



Widzimy z tąd że “wykres funkcji” lub raczej, jego reprezentacja graficzna to po prostu TraditionalForm lub StandardForm kodu *Mathematica* który opisuje ten wykres. Sam kod to InputForm lub StandardForm “obrazka”.

```
InputForm[ ]
```



Przyglądając się temu kodowi, widzimy że wykres funkcji jest obiektem graficznym opisanym przez tylko jedną prymitywną funkcją graficzną Line, która tworzy łamaną łączącom punkty o danych współrzędnych.

■ Podstawowa zasada ewaluacji

Jak już wspominaliśmy, Kernel *Mathematiki* oblicza wartości wyrażeń w ściśle określonym porządku. Ten porządek opiszemy dokładniej później, po wytłumaczeniu wzorów i reguł. Teraz tylko opiszemy główną idea: *Mathematica* oblicza każdą cześć wyrażenia po kolei, zaczynając od Head, i używając reguł zdefiniowanych przez użytkownika oraz wbudowanych, aż wyrażenie przestanie się zmieniać. Wtedy ewaluacja się kończy i *Mathematica* zwraca jej wynik. (Czasem ewaluacja nigdy się nie kończy i wpadamy w nieskończoną pętlę. *Mathematica* próbuje sama wykryć takie sytuacje i zatrzymać ewaluację. Zazwyczaj, ale nie zawsze, robi to skutecznie).

Programowanie przy pomocy wzorów i reguł

W *Mathematice* można łatwo programować w różnych stylach. Jest jednak jeden styl który, choć nie całkowicie unikalny, ale najbardziej odróżnia *Mathematicę* od innych, podobnych programów. Jest to programowanie za pomocą reguł przepisywania (“re-write rules”). Podstawowymi pojęciami są tu “wzory” i “reguły”. Te ostatnie mogą być lokalne albo globalne.

■ Reguły lokalne

Reguły lokalne tworzy się przy pomocy funkcji Rule lub RuleDelayed:

```
? Rule
```

lhs –> *rhs* or *lhs* → *rhs* represents a rule that transforms *lhs* to *rhs*. ≫

```
? RuleDelayed
```

lhs :> *rhs* or *lhs* :→ rhs represents a rule that transforms *lhs* to *rhs*, evaluating *rhs* only after the rule is used. ≫

Oczywiście

```
FullForm[lhs → rhs]
```

```
Rule[lhs, rhs]
```

```
FullForm[lhs :> rhs]
```

```
RuleDelayed[lhs, rhs]
```

Różnica między Rule i RuleDelayed będzie wytłumaczona później. Najczęściej obie te funkcję są używane razem z funkcją ReplaceAll lub Repalce:

```
? ReplaceAll
```

```
expr /. rules applies a rule or list of rules in an
   attempt to transform each sub part of an expression expr. More...
```

```
? Replace
```

Replace[*expr*, *rules*] applies a rule or list of rules in an attempt to transform the entire expression *expr*.
Replace[*expr*, *rules*, *levelspec*] applies rules to parts of *expr* specified by *levelspec*. ≫

Zacznijmy od trochę sztucznego przykładu. Mamy dane wyrażenie $x^2 + \text{Sin}[x\,y] + 3$ i chcemy programistycznie zamienić x i y na π, otrzymując $x^2 + \text{Sin}[\pi^2] + 3$. W tym przypadku najłatwiej zrobić to zapomocą dwóch reguł:

```
x^2 + Sin[x y] + 3 /.{x → π, y -> π}
```

$3 + \pi^2 + \text{Sin}[\pi^2]$

(Sprawdź jaka jest FullForm wyrażenie wejściowego!)

Chcemy teraz zrobić to samo ale przy użyciu tylko jednej reguły. Do tego celu musimy użyć “wzoru” - który pozwoli na równoczesne przekształcenie x i y. Czyli potrzeba nam wyrażenie do którego będzie “pasowało” (“match”) zarówno x jak i y. Najprostrzym wzorem tego typu jest $x \mid y$ - jest to wzór do którego pasują wyłącznie symbole x i y.

```
x^2 + Sin[x y] + 3 /.x | y → π
```

$3 + \pi^2 + \text{Sin}[\pi^2]$

Oczywiście obie metody będą nie wygodne jeśli będziemy potrzebowali reguły do której będzie pasowało bardzo dużo wyrażeń. Spróbujmy użyć najbardziej ogólnego wzoru:

```
? _
```

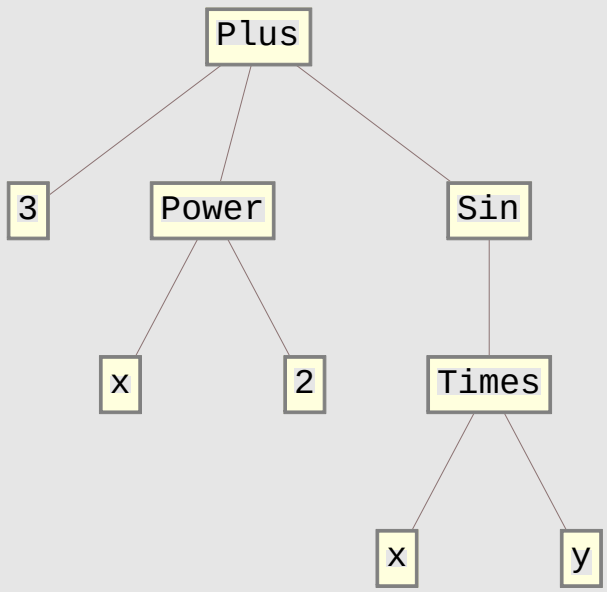
_ or Blank[] is a pattern object that can stand for any Mathematica expression.
_*h* or Blank[*h*] can stand for any expression with head *h*. ≫

```
x^2 + Sin[x y] + 3 /. _ → π
```

π

Co się stało? Otóż funkcja ReplaceAll (której formą infiksową jest /.) zaczyna poszukiwanie podwyrażenia które pasuje do wzoru od całego wyrażenia - i od razu go znajduje. Dalej już nie szuka - całe wyrażenie zostaje zastąpione przez π. Aby
ominąć ten problem spróbujmy użyć funkcji Replace, która może działać na dowlonym poziomie. Przyglądając się strukturze drzewka

```
TreeForm[x^2 + Sin[x y] + 3]
```



widzimy że wszystkie x i y znajdują się na poziomie -1 “liście” drzewa. Niestety, znajduje się tam także 2 oraz 3, więc reguła

```
Replace[x^2 + Sin[x y] + 3, _ → π, {-1}]
```

$\pi + \pi^{\pi} + \text{Sin}\left[\pi^2\right]$

nie daje tego co chcieliśmy. Musimy więc ograniczyć nasz wzór. Można to zrobić tak:

```
Replace[x^2 + Sin[x y] + 3, _Symbol → π, {-1}]
```

$3 + \pi^2 + \text{Sin}\left[\pi^2\right]$

_h jest wzorem do którego pasuje każde wyrażenie którego Head jest h. Istnieje jeszcze wiele innych sposobów tworzenia wzorów.

A teraz przykład gdzie Rule i RuleDelayed dają inną odpowiedź:

```
a = 0;
```

```
Sin[2] /. a_Integer → a^2
```

```
0
```

```
Sin[2] /. a_Integer :> a^2
```

```
sin(4)
```

W pierszym przykładzie prawa strona reguły została obliczona zanim sama reguła została wykonana a poniważ *a* miało wartoś 0 więc Sin[2] zamieniło się w Sin[0] czyli w 0. W drugim przypadku reguła została wykonana bez oblicznia prawej strony więc 2 zostało zamienione przez 2^2 czyli 4.

Zanim zastosujemy regułę do wyrażenia konieczna jest znajomość FullForm tego wyrażenia. Podajemu przykład jednej z wielu pułapek w jakie można wpaść jeśli się nie będzie przestrzegać tej zasady.

```
Clear[a]
```

$$\sqrt{2} + \frac{1}{\sqrt{2}} /. \sqrt{2} \to a$$

$$a + \frac{1}{\sqrt{2}}$$

Patrząc się na FullForm widzimy dlaczego nasza reguła nie dała oczekiwanego rezultatu:

$$\text{FullForm}\left[\sqrt{2} + \frac{1}{\sqrt{2}}\right]$$

```
Plus[Power[2, Rational[−1, 2]], Power[2, Rational[1, 2]]]
```

Są dwa sposoby na obejście tego problemu. Możemy użyć reguły na nie-ewaluawanym wyrażeniu (do tego służy Unevaluated).

$$\text{Unevaluated}\left[\sqrt{2} + \frac{1}{\sqrt{2}}\right] /. \sqrt{2} \to a$$

$$\frac{1}{a} + a$$

Drugą metodą jest użycie poprawnego wzoru dla FullForm wyrażenia po ewaluacji. Na przykład

```
√2 + 1/√2 /. 2^Rational[x_, y_] → a^x
```

$$a + \frac{1}{a}$$

Możemy, oczywiście, także użyć dwóch reguł równocześnie:

```
√2 + 1/√2 /. {√2 → a, 1/√2 → 1/a}
```

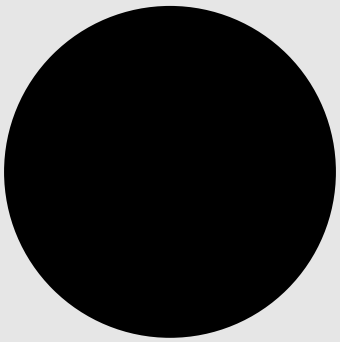
$$a + \frac{1}{a}$$

Reguły i Grafika

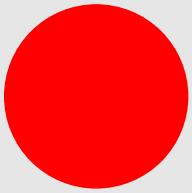
Programowanie za pomocą reguł jest szczególnie wygodne w przypadku grafiki.

```
gr = Disk[{0, 0}, 1];
```

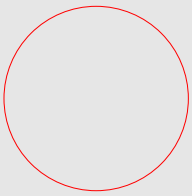
```
Graphics[gr, ImageSize → Small]
```



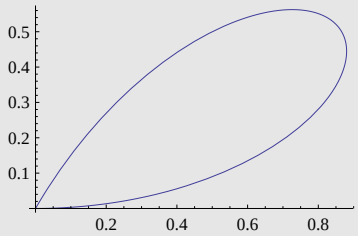
```
Graphics[{Red, gr}, ImageSize → Tiny]
```



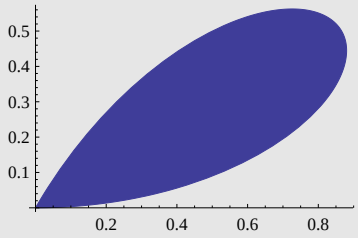
```
Graphics[{Red, gr}, ImageSize → {100, 100}] /. Disk → Circle
```



```
gr2 = PolarPlot[Sin[3 θ], {θ, 0, π/3}, ImageSize → Small]
```




```
gr2 /. Line → Polygon
```



Jak już wspominaliśmy, wiele funkcji *Mathematici* zwraca listę reguł jako wyrażenie wyjściowe. Na przykład:

```
rules = Solve[x^3 + 3 x + 4 == 0, x]
```

$$\left\{\{x \to -1\}, \left\{x \to \frac{1}{2}\left(1 - i\sqrt{15}\right)\right\}, \left\{x \to \frac{1}{2}\left(1 + i\sqrt{15}\right)\right\}\right\}$$

Zauważmy że otrzymaliśmy listę list, każda z których zwiera regułę (kilka reguł jeśli zamiast rownania użyjemy listy kilku równań).

Wygoda takij formy wyrażenia wyjściowego bierze się stąd że możemy teraz użyć ReplaceAll i zastosować otrzymane reguły do innego wyrażenia. Wyrażeniem tym może być nawet samo *x* :

```
x /. rules
```

$$\left\{-1, \frac{1}{2}\left(1 - i\sqrt{15}\right), \frac{1}{2}\left(1 + i\sqrt{15}\right)\right\}$$

```
x^3 + 3 x + 4 /. rules // Simplify
```

```
{0, 0, 0}
```

```
Simplify[%]
```

```
{0, 0, 0}
```

Oto podobny przykład z FindRoot zamiast Solve

```
FindRoot[x e^-x == 0.2, {x, 0.1}]
```

```
{x → 0.259171}
```

Sprawdzimy czy otrzymana odpowiedź spełna równanie wejściowe:

```
x Exp[-x] == 0.2 /. %
```

```
True
```

Links

http : // reference.wolfram.com/mathematica/tutorial/PatternsAndTransformationRules.html

Reguły Globalne ("Funkcje")

Przedstawimy teraz jeden z dwóch głównych sposobów zdefinowania "funkcji" w *Mathematice*:

```
Clear[f]
```

```
f[x_] := x^2
```

```
f[I]
```

```
-1
```

```
Clear[a]
```

```
f[a]
```

```
a^2
```

```
f[3]
```

```
9
```

```
?f
```

```
Global`f
```

```
f[x_] := x^2
```

```
DownValues[f]
```

$$\{\text{HoldPattern}[f(\text{x\_})] :\to x^2\}$$

Chociaż czesto takie *f* nazywa się “funkcją”, w rzeczywistości jest to “reguła globalna”, w sensie że jest stosowana nie do wybranego wyrażnia ale do każdego wyrażenia które jest ewaluawane przez Kernel. Dokładniej, kiedy definicja tego typu jest ewaluowana, *Mathematica* tworzy regułę dla symbolu *f*. Teraz przy każdej ewaluacji *f* reguła będzie zastosowana. Reguła jest przechowywana jako tak zwana DownValue *f*:

```
DownValues[f]
```

$$\{\text{HoldPattern}[f(\text{x\_})] :\to x^2\}$$

$$\{\text{HoldPattern}[f(\text{x\_})] :\to x^2\}$$

Tutaj x_ to wzór, do którego pasuje wszytko, z tym że po dopasowaniu wyrażenie tymczasowo zostaje nazwane *x*. Reguła teraz brzmi: "zamień *f*(cokolowiek) na cokolwiek^2". HoldPattern jest to konieczne aby zapobiec ewaluacji f(x_) (które inaczej byłoby zastąpione przez by x_^{2}) ale HoldPattern[f[x_]] jest traktowane tak jak *f*[*x*] podczas dopasowywania.

Czyli, po ewaluacji tego rodzaju definicji *Mathematica* tworzy pewne reguły, zachowuje je jako DownValues, poczym stosuje w pewnej kolejności. Oto przykłady:

```
Clear["Global`*"]
```

```
f[x_Real] := 3
```

```
f[1] := 2
```

```
f[x_Symbol] := x^2
```

```
f[x_Integer] := 5
```

```
DownValues[f]
```

$$\{\text{HoldPattern}[f(1)] :\to 2,\ \text{HoldPattern}[f(\text{x\_Real})] :\to 3,\ \text{HoldPattern}[f(\text{x\_Symbol})] :\to x^2,\ \text{HoldPattern}[f(\text{x\_Integer})] :\to 5\}$$

```
DownValues[f] = Rest[DownValues[f]]
```

$$\{\text{HoldPattern}[f(\text{x\_Real})] :\to 3,\ \text{HoldPattern}[f(\text{x\_Symbol})] :\to x^2,\ \text{HoldPattern}[f(\text{x\_Integer})] :\to 5\}$$

```
DownValues[f]
```

$\{\text{HoldPattern}[f(\text{x\_Real})] :\to 3,\ \text{HoldPattern}[f(\text{x\_Symbol})] :\to x^2,$
$\text{HoldPattern}[f(\text{x\_Integer})] :\to 5,\ \text{HoldPattern}[f(\text{x\_})] :\to 0\}$

```
f[2 / 3]
```

0

```
f[x_] := 0
```

```
f["cat"]
```

$f(\text{cat})$

```
f[1]
```

5

```
f[x_] := 0
```

```
f["dog"]
```

0

```
Clear[f]
```

```
DownValues[f]
```

{}

```
f[2]
```

5

```
f[7]
```

5

```
f[1.1]
```

3

```
f[a]
```

a^2

```
f["cat"]
```

f(cat)

```
Map[f, {.5, 1, a}]
```

$\{3, 2, a^2\}$

Kolejność stosowania zapisanych reguł jest (w przybliżeniu) określony przez dwa czynniki: bardziej specyficzne reguły są stosowane przed bardziej ogólnymi - w przypadku reguł które *Mathematica* traktuje jako “równo ogólne”, decyduje kolejność w której powstały.

Poza DownValues, są także OwnValues, UpValues, SubValues i NValues:

```
Clear[a]
```

```
a = 1; OwnValues[a]
```

{HoldPattern[a] :↦ 1}

```
ClearAll[b];
```

```
b /: Sin[b] = 2;
```

```
b /: Cos[b] = 2;
```

```
cos^2(b) + sin^2(b)
```

8

```
UpValues[b]
```

{HoldPattern[cos(b)] :↦ 2, HoldPattern[sin(b)] :↦ 2}

W procesie ewaluacji wyrażenia *Mathematica* stosuje reguły zachowane w UpValues, DownValues itd, po czym stosuje reguły wbudowane. Ewaluacja to proces rekursywny: zaczyna się od głowy całego wyrażenia, potem ewaluawane są argumenty. Kolejność ewaluacji może być zmienione przez Atrybuty - o których będzie mowa dalej, i jest kontynuowana aż wyrażenie przestanie się zmieniać.

Zobaczmy teraz przykład sytuacji kiedy reguły zapisane w UpValues i DownValues są z sobą sprzeczne. Przykład ten ilustruje znaczenie kolejności stosowania reguł.

Najpierw definiujemy DownValue dla symbolu *f*:

```
f[x_] := x^2
```

Następnie definiujemy UpValue dla symbolu *y*:

```
g_[y] ^:= y^3
```

Sprawdzamy że wszytko działa tak jak powinno:

```
{f[p], q[y]}
```

$$\{p^2, y^3\}$$

Następny wynik pokazuje że UpValues mają pierwszeństwo przed DownValues:

```
f[y]
```

$$y^3$$

Podamy jeszcze przykład wbudowanych reguł które *Mathematica* stosuje automatycznie

```
Clear[a]
```

```
a^n a^m
```

$$a^{m+n}$$

Wykonywanie tego rodzaju “uproszczeń” automatycznie w czasie ewaluacji wyrażenia jest uzasadnione potrzebami szybkośći. Z drugiej strony, oznacza to że jedyny sposób otrzymania w formule wyjściowej wyrażenia z rozdzielonymi potęgami wymaga użycia funkcji Hold lub HoldForm:

```
HoldForm[a^2 a^3] / a^2
```

$$\frac{a^2\, a^3}{a}$$

Z drugiej strony, wiele podstaowych uproszczeń nie zostanie wykonane automatycznie:

```
cos(α)^2 + sin(α)^2
```

$$\sin^2(\alpha) + \cos^2(\alpha)$$

mimo że to wyrażenie jest zawsze równe 1, uproszczenie wymaga zastosowania jednej z funkcji “upraszczających”:

```
Simplify[cos^2(a) + sin^2(a)]
```

```
1
```

Niektóre uproszczenia wymagają dodatkowych założeń. Domyślnie *Mathematica* zakłada że każda zmienna reprezentuję dowolną liczbę zespoloną, a więc

```
Simplify[Sqrt[x^2]]
```

$$\sqrt{x^2}$$

```
Assuming[x >= 0, Simplify[Sqrt[x^2]]]
```

```
x
```

```
Assuming[x <= 0, Simplify[Sqrt[x^2]]]
```

```
-x
```

Jak już wspominaliśmy, ogólnie *Mathematica* stosuje do wyrażenia wszytkie transformacje aż wyrażenie przestanie się zmieniać. W niektórych sytuacjach jednak tak nie jest; na przykład funkcja ReplaceAll przeszukuje wszystkie części wyrażenia aż znajdzie coś co pasuje do danego wzoru, ale w każdej części szuka tylko jednego pasującego podwyrażenia.

```
rules = {Log[x_ y_] :> Log[x] + Log[y], Log[x_^k_] :> k Log[x]};
```

```
Log[Sqrt[a (b c^d)^e]] /.rules
```

$$\frac{1}{2}\log\left(a\left(b\,c^{d}\right)^{e}\right)$$

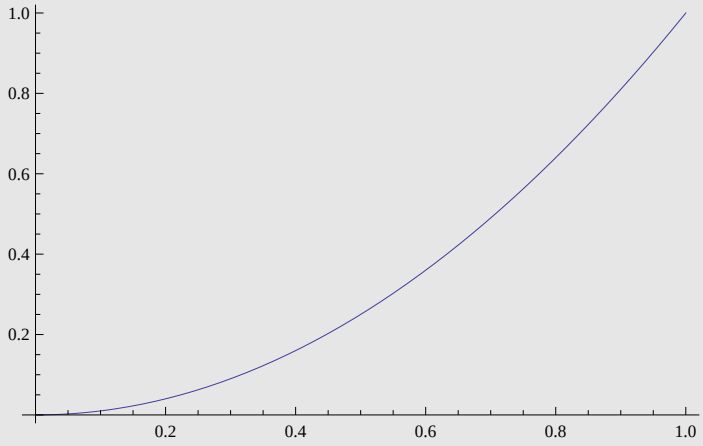
Tym razem ReplaceAll nie znalazło nic co pasuje do pierszej reguły i tylko jedno podwyrażenie pasujące do drugiej.
Oczywiście po przepisaniu wyrażenia pojawia się nowy przypadek pasujący do wzorów ale ReplaceAll już go nie szuka. Aby wykonać wszystkie możliwe transformacje musimy użyć ReplaceRepeated (//.), które kontynuje poszukiwanie rekursywnie aż wyrażenie przestanie się zmieniać:

```
Log[Sqrt[a (b c^d)^e]] //.rules
```

$$\frac{1}{2}\left(\text{Log}[a] + e\left(\text{Log}[b] + d\,\text{Log}[c]\right)\right)$$

Inny ważny fakt dotyczący reguł (lokalnych) : Opcje funkcji *Mathematiki* są także regułami.

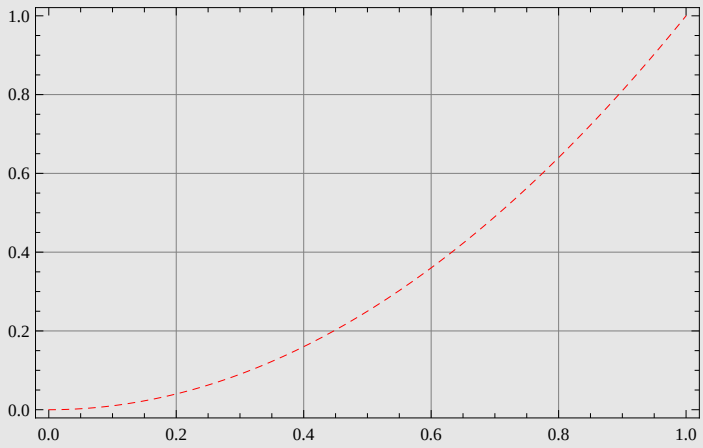
```
Plot[x^2, {x, 0, 1}]
```




```
Options[Plot]
```

$$\left\{\text{AlignmentPoint} \to \text{Center},\ \text{AspectRatio} \to \frac{1}{\phi},\ \text{Axes} \to \text{True},\ \text{AxesLabel} \to \text{None},\ \text{AxesOrigin} \to \text{Automatic},\right.$$
$$\text{AxesStyle} \to \{\},\ \text{Background} \to \text{None},\ \text{BaselinePosition} \to \text{Automatic},\ \text{BaseStyle} \to \{\},\ \text{ClippingStyle} \to \text{None},$$
$$\text{ColorFunction} \to \text{Automatic},\ \text{ColorFunctionScaling} \to \text{True},\ \text{ColorOutput} \to \text{Automatic},$$
$$\text{ContentSelectable} \to \text{Automatic},\ \text{DisplayFunction} :\to \$\text{DisplayFunction},\ \text{Epilog} \to \{\},\ \text{Evaluated} \to \text{Automatic},$$
$$\text{EvaluationMonitor} \to \text{None},\ \text{Exclusions} \to \text{Automatic},\ \text{ExclusionsStyle} \to \text{None},\ \text{Filling} \to \text{None},$$
$$\text{FillingStyle} \to \text{Automatic},\ \text{FormatType} :\to \text{TraditionalForm},\ \text{Frame} \to \text{False},\ \text{FrameLabel} \to \text{None},$$
$$\text{FrameStyle} \to \{\},\ \text{FrameTicks} \to \text{Automatic},\ \text{FrameTicksStyle} \to \{\},\ \text{GridLines} \to \text{None},$$
$$\text{GridLinesStyle} \to \{\},\ \text{ImageMargins} \to 0.,\ \text{ImagePadding} \to \text{All},\ \text{ImageSize} \to \text{Automatic},\ \text{LabelStyle} \to \{\},$$
$$\text{MaxRecursion} \to \text{Automatic},\ \text{Mesh} \to \text{None},\ \text{MeshFunctions} \to \{\#1\ \&\},\ \text{MeshShading} \to \text{None},$$
$$\text{MeshStyle} \to \text{Automatic},\ \text{Method} \to \text{Automatic},\ \text{PerformanceGoal} :\to \$\text{PerformanceGoal},$$
$$\text{PlotLabel} \to \text{None},\ \text{PlotPoints} \to \text{Automatic},\ \text{PlotRange} \to \{\text{Full},\ \text{Automatic}\},$$
$$\text{PlotRangeClipping} \to \text{True},\ \text{PlotRangePadding} \to \text{Automatic},\ \text{PlotRegion} \to \text{Automatic},$$
$$\text{PlotStyle} \to \text{Automatic},\ \text{PreserveImageOptions} \to \text{Automatic},\ \text{Prolog} \to \{\},\ \text{RegionFunction} \to (\text{True}\ \&),$$
$$\left.\text{RotateLabel} \to \text{True},\ \text{Ticks} \to \text{Automatic},\ \text{TicksStyle} \to \{\},\ \text{WorkingPrecision} \to \text{MachinePrecision}\right\}$$

```
Plot[x^2, {x, 0, 1}, Axes → False, Frame → True, GridLines → Automatic, PlotStyle → {Red, Dashing[0.01]}]
```



Różnica pomiędzy SetDelayed i Set

Różnica pomiędzy SetDelayed (:=) i Set (=) jest dokładnie taka sama jak ta pomiędzy RuleDelayed (:→) i Rule → .

Dla przykładu, rozważmy dwie definicje:

```
f[p_] := Expand[p]
```

```
g[p_] = Expand[p];
```

Stosując f do wyrażnia $(a+b)^3$ otrzymamy inny wynik:

```
f[(a + b)^3]
```

$$a^3 + 3\,a^2\,b + 3\,a\,b^2 + b^3$$

```
g[(a + b)^3]
```

$$(a + b)^3$$

Proste wytłumaczenie tego zachowania jest takie. Set oblicza prawą stronę wyrażenia zanim przypisze wynik lewej stronie, natomiast SetDelayed przypisuje lewej stronie nie obliczoną (nie-ewaluowaną) wartość prawej strony.

Dokładniejsze wytłumacznie wymaga pojęcia atrybutu funkcji, rozważanego poniżej.

Links

http : // reference.wolfram.com/mathematica/tutorial/ManipulatingValueLists.html

http : // reference.wolfram.com/mathematica/tutorial/ManipulatingOptions.html

Funkcje i Programowanie Funkcyjne

Czyste Funkcje (Pure Functions)

Poza “funkcjami” opisanymi powyżej, które, w rzeczywistości są gloablnymi regułami, w *Mathematice* można także tworzyć “prawdziwe funkcje”, za pomocą funkcji Function. Na przykład, Function$[x, x^3]$ jest funkcją podnoszenia do sześcianu:

```
Function[x, x^3][c]
```

$$c^3$$

Zauważmy że taka funkcja (w przeciwienstwie to funkcji zdefiowanej przez globalną regułę) nie potrzebuje nazwy, stąd takie funkcje zwane są także “funkcjami anonimowymi” (“anonymous functions”). Oczywiście jeśli chcemy, możemy nadać jej imie za pomocą reguły globalnej,

```
cube = Function[x, x^3];
```

```
cube[3]
```

```
27
```

```
OwnValues[cube]
```

```
{HoldPattern[cube] :> Function[x, x^3]}
```

W taki sam sposób możemy tworzyć funkcje wielu zmiennych:

```
Function[{x, y}, x^3 + y^2][4, 2]
```

```
68
```

Te podejście ma dwa oczywiste problemy. Jeden to niewygoda konieczności używania liter do oznaczania argumentów funkci. Ten problem jest rozwiązany przez użycie notacji #1, #2 ,... na pierwszy, drugi, trzeci itd argument funckcji. Na przykład:

```
Function[#1^2 + #2^2][1, 5]
```

```
26
```

Można także zdefiniować funkcję z nieograniczoną ilością argumentów:

```
Function[{##}^2][1, 1, 2]
```

```
{1, 1, 4}
```

Drugim problemem jest długość słowa Function. Zazwyczaj zastępuję się je przez skrót: po prostu omijamy słowo Function i oznaczamy koniec definicji funkcji symbolem &, tak jak tu

```
#1^2 + #2^2 &[2, 3]
```

```
13
```

Predykaty (Funkcje Boolowskie)

Ważną klasą funkcji są funkcje o wartościach True i False. W Mathematice takie funkcje zwane są predykatami. Wiele z predykatów w *Mathematica* kończy się literą Q (zapewne od “question”):

```
PrimeQ[25]
```

```
False
```

```
PrimeQ[18]
```

```
False
```

```
EvenQ[7]
```

```
False
```

Według konwencji *Mathematiki* tylko te funkcje boolowskie kończą się na Q które zawsze dają odpowiedź True lub False. Na przykład, dla nie symbolu *x*:

```
EvenQ[x]
```

```
False
```

Z drugiej strony, funkcja (dwu argumentów) Element także jest boolowska, ale nie kończy się na Q. Widzimy że

```
Element[π, Reals]
```

```
True
```

```
Element[x, Reals]
```

```
x ∈ Reals
```

Pokażemy teraz dwa sposoby zdefiniowania funkcji boolowskiej która sprawdza czy dana liczba jest większa od 5. Pierwsza metoda to zdefiniowanie globalnej reguły:

```
LargerThanFive[n_] := n > 5
```

```
LargerThanFive[1]
```

```
False
```

```
LargerThanFive[p]
```

$p > 5$

Teraz to samo za pomocą czystej funkcji:

```
# > 5 &[7]
```

```
True
```

Czyste funkcje mogą być używane do tworzenia nowych wzorów:

```
Clear[f]
```

```
f[x_?(# > 5 &)] := x^2
```

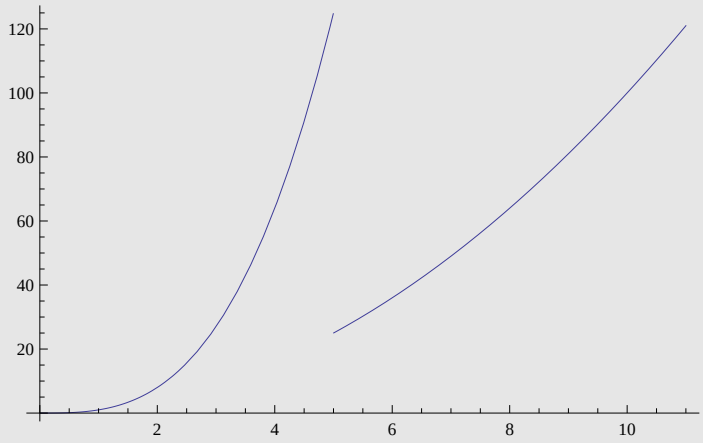
```
f[x_?(# ≤ 5 &)] := x^3
```

Jest to typowy przykład definicji wieloczęściowej. Funkcje f jest kwadratem dla $x > 5$ i sześcianem dla $x \leq 5$. Funkcja oznaczona przez ? to PatternTest:

```
?PatternTest
```

p?*test* is a pattern object that stands for any
expression which matches *p*, and on which the application of *test* gives True. ≫

```
Plot[f[x], {x, 0, 11}, Exclusions → {5}]
```



Atrybuty

Zachowanie się funkcji i globalnych reguł *Mathematiki* jest kontrolowane przez ich Atrybuty (Attributs). Każda wbudowana funkcja ma pewne atrybuty:

```
Attributes[Sin]
```

```
{Listable, NumericFunction, Protected}
```

Prawie każda funkcje ma atrybut “Protected”. Dzięki niemu nie można niechcący zmienić definicji funkcji. Na przykład

```
Sin[Pi] = 0
```

Set::write : Tag Sin in Sin[π] is Protected. ≫

```
0
```

Jeśli naprawdę chcielibyśmy zmienić wartość funkcji sinus w π, to możemy to zrobić, ale najpierw musimy użyć funkcji Unprotect

```
Unprotect[Sin]; Sin[Pi] = 2; Protect[Sin];
```

Teraz

```
Sin[Pi]
```

```
2
```

Oczywiście taka zmiana zazwyczaj nie jest dobrym pomysłem. Aby przywrócić oryginalną definicję sinusa musimy znowu wykonać ciąg operacji:

```
Unprotect[Sin]; Clear[Sin]; Protect[Sin];
```

Oczywiście Clear usuwa tylko definicje użytkownika a więc funkcja wraca do oryginalnego zachowania:

```
Sin[Pi]
```

```
0
```

Jednym z najbardziej pożytecznych atrybutów jest Listable. Wytłumaczymy jak on działa. Zacznijmy od funkcji *f* bez tego atrybutu.

```
Clear[f]
```

```
ls = Range[10]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
Map[f, ls]
```

$\{f(1), f(2), f(3), f(4), f(5), f(6), f(7), f(8), f(9), f(10)\}$

Jeśli nadamy *f* atrybut Listable uzyskamy powyższy wynik bez używania Map:

```
SetAttributes[f, Listable]
```

```
f[ls]
```

$\{f(1), f(2), f(3), f(4), f(5), f(6), f(7), f(8), f(9), f(10)\}$

Atrybut wpływa także na zachoanie się funkcji z więcej niż jednym argumentem:

```
f[{a, b}, {c, d}]
```

$\{f(a, c), f(b, d)\}$

```
f[a, {b, c}]
```

$\{f(a, b), f(a, c)\}$

To właśnie fakt że funkcja Plus ma atrybut Listable powoduje znane nam zachowanie:

```
{1, 2, 3} + {4, 5, 6}
```

{5, 7, 9}

```
1 + {2, 3, 4, 5}
```

{3, 4, 5, 6}

Atrybuty Orderless, Flat i OneIdentity są interesujące ale bardziej skomplikowane. Zobaczmy to na ilustracji:

```
ClearAll[f]
```

```
f[a, b] /. f[b, x_] → g
```

$f(a, b)$

Powyżej wyrażenie $f[a, b]$ nie pasuje do wzoru $f[b,$ x_] bo, oczywiście, kolejność argumentów się nie zgadza. Po dodaniu atrubutu Orderless funkcja f staje się "komutatywna":

```
SetAttributes[f, Orderless]
```

```
f[a, b] /. f[b, x_] → g
```

g

Poniżej $f[a, b, c]$ nie pasuje do wzoru $f[a, f[b, c]]$

```
f[a, b, c] /. f[a, f[b, c]] → g
```

$f(a, b, c)$

Po dodaniu atrybutu Flat funkcja f staje się spójna:

```
SetAttributes[f, Flat]
```

```
f[a, b, c] /. f[a, f[b, c]] → g
```

g

```
ClearAll[f]
```

Inną bardzo ważną grupą atrybutów stanowią atrybuty ze słowem Hold, HoldFirst, HoldAll, HoldRest i kilka innych (HoldComplete, HoldAllComplete, SequenceHold,NHoldFirst,NHoldRest,NHoldAll). Normalnie funkcja w Mathematice oblicza wszystkie swoje argumenty zanim zacznie stosować definicję. Jednakże, jeśli funkcja ma jeden z powyższych atrybutów, wszystkie albo część argumentów nie jest obliczona przez ewaluacją. Na przykład, funkcje Set nie oblicza swojego pierszego atrybutu:

```
Attributes[Set]
```

```
{HoldFirst, Protected, SequenceHold}
```

Natomiast funkcja SetDelayed nie oblicza żadnego ze swoich atrybutów:

```
Attributes[SetDelayed]
```

```
{HoldAll, Protected, SequenceHold}
```

Na koniec, zobaczmy przykład użycia atrybutu HoldFirst w definiowaniu własnej funkcji. Poniżej zdefiniowana funkcja *f* przyjmuje dwa argumenty. Pierwszy z nich jest zmienną a drugi liczbą. Funkcja przypisuję danej zmiennej kwadrat danej liczby. W sytuacji kiedy zmienna ma już przypisaną wartość nastąpiłby błąd polegający na przypisaniu liczbie innej liczby. Nadając funkcji *f* atrybut HoldFirst, unikamy tego problemu.

```
ClearAll[f]
```

```
SetAttributes[f, HoldFirst]
```

```
f[x_, y_] := (x = y^2)
```

```
x = 3;
```

```
f[x, 2];
```

```
x
```

```
4
```

Links

http : // reference.wolfram.com/mathematica/tutorial/PureFunctions.html

http : // reference.wolfram.com/mathematica/tutorial/ApplyingFunctionsToListsAndOtherExpressions.html

http : // reference.wolfram.com/mathematica/tutorial/Attributes.html

http : // reference.wolfram.com/mathematica/tutorial/SelectingPartsOfExpressionsWithFunctions.html