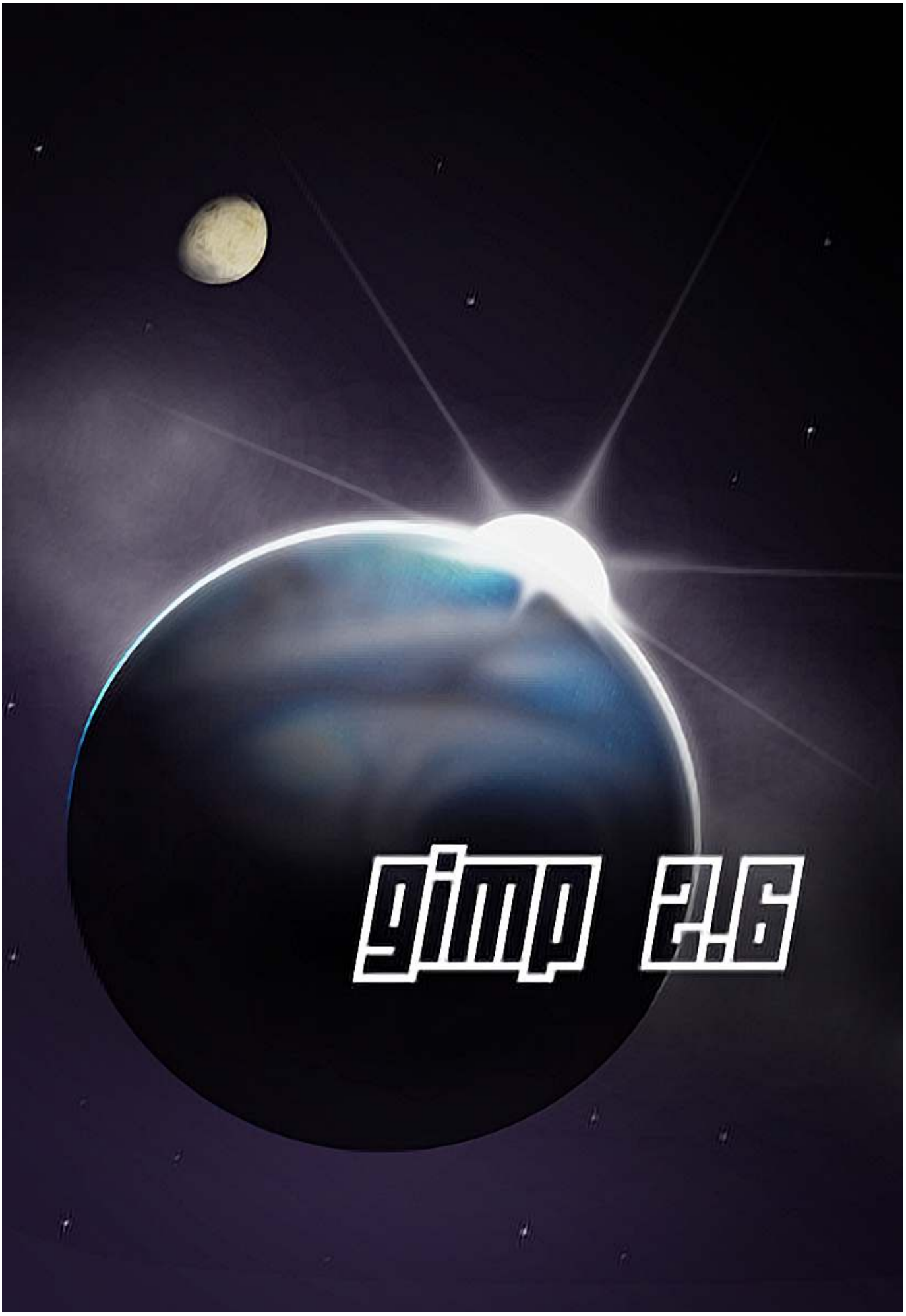

gimp 2.6

Poradnik

Prezentowany poniżej poradnik, to wynik analizy dostępnych informacji oraz własnych doświadczeń i przemyśleń. Wiedza dotycząca tego tematu jest rozproszona, a jednocześnie stale się rozwija.

Wprowadzenie do pisania Script-Fu w GIMP.

2010 - 2011r

Jak kogoś to nie interesuje, czytać nie musi !.

Spotkałem się z określeniem **Script kiddie** - na niedoświadczonych, którzy używają programów i skryptów napisanych przez innych bez znajomości zasad ich działania.
Skrypty są to – programy napisane w językach skryptowych – wykonywane są wewnątrz określonej aplikacji, w odróżnieniu od programów (nie skryptowych), które wykonują się niezależnie od innych aplikacji.

Języki skryptowe osadzane bywają w różnych programach, dzięki czemu zapewniają możliwość zautomatyzowania powtarzających się czynności.
W procesie tworzenia / przetwarzania obrazu wykonujemy sekwencje elementarnych czynności, takich jak tworzenie nowego obrazu, dodaj / usuń warstwę, wydzielenie obszaru, tworzenie tekstu, stosowanie różnorodnych filtrów, itp. przy czym ta sekwencja może być dość długa, a jeśli jeszcze wymaga się również, utworzenia dużej ilości obrazów wg. identycznego algorytmu lub wykonania identycznej obróbki wielu zdjęć, to oczywiście, pojawia się potrzeba przeniesienia przynajmniej części powtarzalnych prac na barki „maszyny". Skrypty to "makro-instrukcje", ale Script-Fu jest bardziej wydajny. Script-Fu jest oparty na języku interpretacji nazw programu i działa przy użyciu funkcji zapytań do bazy danych GIMP.
Skrypt (z łac. *scriptum* – notowanie szczegółów, scenariusz; lub *script* pismo; tekst;) jest ciągiem instrukcji dla GIMP-a.
Zastanawia skąd nazwa skryptów jako **script-fu**. Według legendy hakerów, poprzez naśladowanie chińskiego terminu *kung-fu* odnoszącego się do *osiągnięcie czegoś przez ciężką i wytrwałą pracę,* zdolności osoby do tworzenia w danej dziedzinie, do której się dochodzi przez wytężoną pracę.
W każdym razie, script-fu to zwykły plik tekstowy zawierający serię instrukcji do wykonania, otoczony przez niektóre informacje niezbędne do ich wykonania.
Głównym językiem używanym do pisania skryptów w GIMP-ie jest pochodna LISP-a - Scheme (obecnie zastąpiony przez bardziej okrojony - Tiny-Fu).
Script-Fu to nie tylko jeden język skryptowy dostępny GIMP . Ale Script-Fu jest jedynym językiem skryptowym, który jest instalowany domyślnie.
Aby więc rozpocząć pisanie skryptów system nie wymaga żadnego dodatkowego oprogramowania, innego niż tylko GIMP; wszystko co potrzeba do programowania Script-Fu znajduje się w systemie, jest to Konsola TinyScheme (*MałyScheme*) oraz wywoływana z niej Przeglądarka procedur.
LISP - skrót ten oznacza **LIS**t **P**RZETWARZANIE tj. język do przetwarzania list, lub w innej interpretacji: **L**anguage of **I**diotic **S**illy **P**arenthesis = (Język idiotyczne głupich nawiasów), twierdzenie kontrowersyjne, ale nie bez uzasadnienia - brak właściwej równowagi nawiasów – jest jednym z głównych źródeł błędów programu napisanego w LISP i jemu podobnych.
Jedną z głównych trudności w nauce języka programowania Scheme jest brak odpowiedniej literatury w j. polskim. Sytuacja jeszcze trochę się komplikuje z uwagi na to że od GIMP ver. 2.4 jest stosowany bardziej nowoczesny interpretator Scheme nazywany **TinyScheme** (TinyScheme nie istniałby, gdyby nie interpretator **miniScheme** ver. 0.85 coded by Atsushi Moriwaki (11/5/1989)), który delikatnie mówiąc, trochę się różni od Scheme. Kevin Cozens stworzył interpretator GIMP-a o nazwie Tiny-Fu , aby zastąpić wbudowany interpreter SIOD. 15 października 2006, plugin Tiny-Fu został wdrożony w dystrybucji GIMP, w oparciu o wersję 1.38 TinyScheme, do której wprowadzono potem szereg poprawek usuwających zauważone błędy. Jedną z głównych przyczyn tego przejścia jest wspieranie międzynarodowych języków i czcionek, (kod UTF-8) czego SIOD nie oferował. Istnieją także inne korzyści, ale brak wsparcia międzynarodowego był najbardziej znaczącym.
Interpretator *TinyScheme* nie obsługuje wszystkich funkcji R5RS, implementuje tak duży podzbiór R5RS jak to było możliwe, jeśli procedura jest dostępna, to ma się zachowywać jak udokumentowana.
Istnieją pewne różnice pomiędzy interpretatorem Script-Fu, SIOD a TinyScheme, które mogą się pojawić przy próbie użycia starszych skryptów w GIMP od ver 2.4.
Jest dostępny ***Script-Fu migration guide***. W poradniku znajdują się opisy niektórych problemów, które mogą się pojawić i jakie kroki podjąć aby je zlikwidować.
Jakie są korzyści Tiny-Fu nad Script-Fu?
Po pierwsze, interpretator stosowany w systemie Tiny-Fu jest najnowszą generacją, który stara się postępować zgodnie z normami systemu tak ściśle, jak to tylko możliwe, w odróżnieniu od Script-Fu w SIOD. Ułatwia to komuś, kto zna Scheme szybsze rozpoczęcie pisania skryptów do GIMP używając Tiny-Fu.
Po drugie, Tiny-Fu używa kilku rozszerzeń do systemu tłumacza. Np. rozszerzenia dodające jakieś time / date i pliki funkcji I / O, funkcje list, które mogą być używane w skryptach. Dodatki sprawiają, że możliwe jest

tworzenie niektórych skryptów, które nie byłoby możliwe w Script-Fu. Jednym z takich skryptów (w zestawie z Tiny-Fu) jest generowanie skryptu contact sheet - formularz kontaktowy.
Po trzecie, w Tiny-Fu debugowanie skryptów jest znacznie łatwiejsze. Wykonanie śledzenia skryptu może być włączane i wyłączane w (prawie) dowolnym punkcie skryptu. Kiedy włączone jest śledzenie otrzymamy bardzo szczegółowe informacje na temat wykonywania skryptu. Należy być tylko ostrożnym, że nie wpływamy na Wartość zwracaną przez procedurę po wyłączeniu śledzenia.
Tiny-Fu wykorzystuje interpreter stworzony przez Dimitriosa Souflis znany jako TinyScheme.

Poniżej postaram się przedstawić podstawy, tego stosunkowo prostego języka programowania. Prawdopodobnie znajdą się jakieś błędy, które popełnia się, chcąc w prosty sposób opisać sedno rzeczy. Ale mam nadzieję, że zostanie to mnie wybaczone, a błędy zgłoszone. Mam również nadzieję, że moje opisy będą wystarczająco pomocne, aby rozpocząć tworzenie własnych skryptów i pozwolą poznać bardziej szczegółowo „kuchnię" GIMP-a.
Być może zbytnia szczegółowość będzie Niektórych śmieszyć, ale pamiętajmy, że nie każdy wie tyle samo, a ja chciałbym żeby ten Poradnik mógł zrozumieć i powtórzyć nawet bardzo początkujący.

Więcej szczegółów każdy znajdzie np. w:

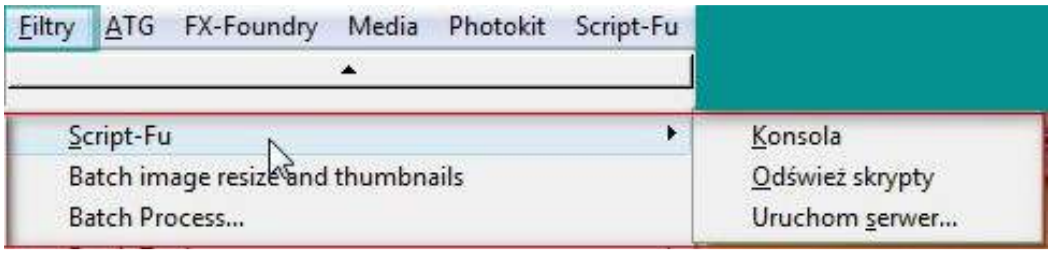
http://people.delphiforums.com/gjc/siod.html
http://www.cs.indiana.edu/scheme-repository/imp/siod.html
http://pl.wikipedia.org/wiki/Scheme
http://en.wikipedia.org/wiki/Scheme_%28programming_language%29
http://www.schemers.org/Documents/Standards/R5RS/r5rs.pdf
http://www.r6rs.org/final/r6rs.pdf
http://www.dutchgimpers.nl/index.php?n=ScriptFu.TinySchemeManual TinySCHEME Version 1.38
https://github.com/dchest/tinyscheme/blob/master/Manual.txt
http://students.mimuw.edu.pl/~ts248384/download/zjk/gimp-2.6.0/plug-ins/script-fu/tinyscheme/Manual.txt
http://www.openengine.dk/code/extensions/TinyScheme/tinyscheme/scheme.c
Download tinyscheme-1.40.tar.gz (60.2 KB) 2011-01-18
http://www.ve3syb.ca/software/gimp/tiny-fu.html
http://tinyscheme.sourceforge.net/home.html
http://www.ve3syb.ca/software/gimp/tiny-fu-faq.html
http://www.ve3syb.ca/wiki/doku.php?id=software:sf:updating-scripts Original text by Saul Goode 2007/10/04
http://www.gimp.org/tutorials/Basic_Scheme/
http://www.gimp.org/tutorials/Basic_Scheme2/
http://www.scheme.com/tspl3/
http://de.wikibooks.org/wiki/GIMP/_Band5/_Script-Fu/_ab_GIMP_2.4
http://imagic.weizmann.ac.il/~dov/gimp/scheme-tut.html
http://www.enotes.com/topic/Scheme_%28programming_language%29
http://searchcio-midmarket.techtarget.com/sDefinition/0,,sid183_gci212139,00.html

Jak wspomniano GIMP Script-fu zawiera pewne file handling and time extensions, jak również pewne funkcje w celu zapewnienia kompatybilności wstecznej z SIOD. Funkcje te zostały tylko pod warunkiem zgodności, jeżeli były one znane i były stosowane w poprzednio istniejących skryptach GIMP-a (np. funkcje "**strbreakup**" i "**unbreakupstr**" są aktualnie dostępne tylko w realizacji Script-fu GIMP-a.)
Jeśli wydaje się że coś jeszcze brakuje (np. liczby zespolone, nie są obsługiwane przez TinyScheme), możemy zapoznać się z kodem
https://github.com/dchest/tinyscheme/blob/master/Manual.txt
i "**init.scm**", który jest częścią TinyScheme. Deweloper zmienił jego nazwę na **skryptu-fu.init** plik ten znajduje się (dla Windows), w
C:/Program Files/GIMP-2.6. /GIMP-2.0/share/gimp/2.0/scripts/script-fu.init
`Initialization file for TinySCHEME 1.38`
Można również zapoznać się z:
C:/Program Files/GIMP-2.6./GIMP-2.0/share/gimp/2.0/scripts/script-fu-compat.init
lub także tutaj: http://git.gnome.org/browse/gimp/tree/plug-ins/script-fu/scripts
A w ostateczności możemy zapoznać się z **minischeme** readme.
minischeme.tar.gz

GIMP Library Reference Manual for the GIMP 2.0 API (m.in.for GIMP 2.6.0)
http://developer.gimp.org/api/2.0/libgimp/index.html

Interpreter TinyScheme w GIMP.

Do pisania skryptów będziemy potrzebować naszego ulubionego tekstowego edytora ASCI znajomości składni języka TinyScheme, oraz uruchomionego GIMP-a w którym mamy:

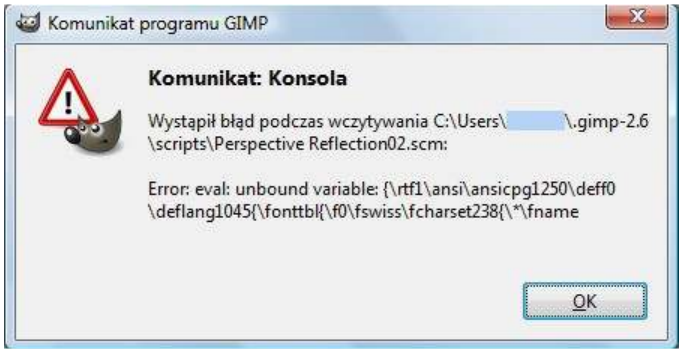


Opcje wspierające pisanie skryptów

Filtry =>Script-Fu => Konsola Script-Fu
Konsola Script-Fu pozwala nam pracować interaktywnie w TinyScheme. Pozwoli nam bezpośrednio wprowadzać polecenia (jako Procedury) i zwraca wartość jako wynik. To tutaj będziemy polecić GIMP, utworzyć nowy obraz. Tu możemy testować nowe skrypty, lub instrukcje napisane w TinyScheme.
Filtry =>Script-Fu => Przeglądarka procedur
Jak sama nazwa wskazuje, zawiera ona zbiór procedur z których można korzystać w GIMP-ie wraz z ich opisami. Dla osób chętnych pisać własne skrypty, to wręcz lektura obowiązkowa.
Filtry =>Script-Fu => Odśwież skrypty
Tej opcji należy użyć, jeśli dodaliśmy, lub zmieniliśmy jakiś skrypt w czasie trwania działania programu GIMP i chcemy testować jego działanie.

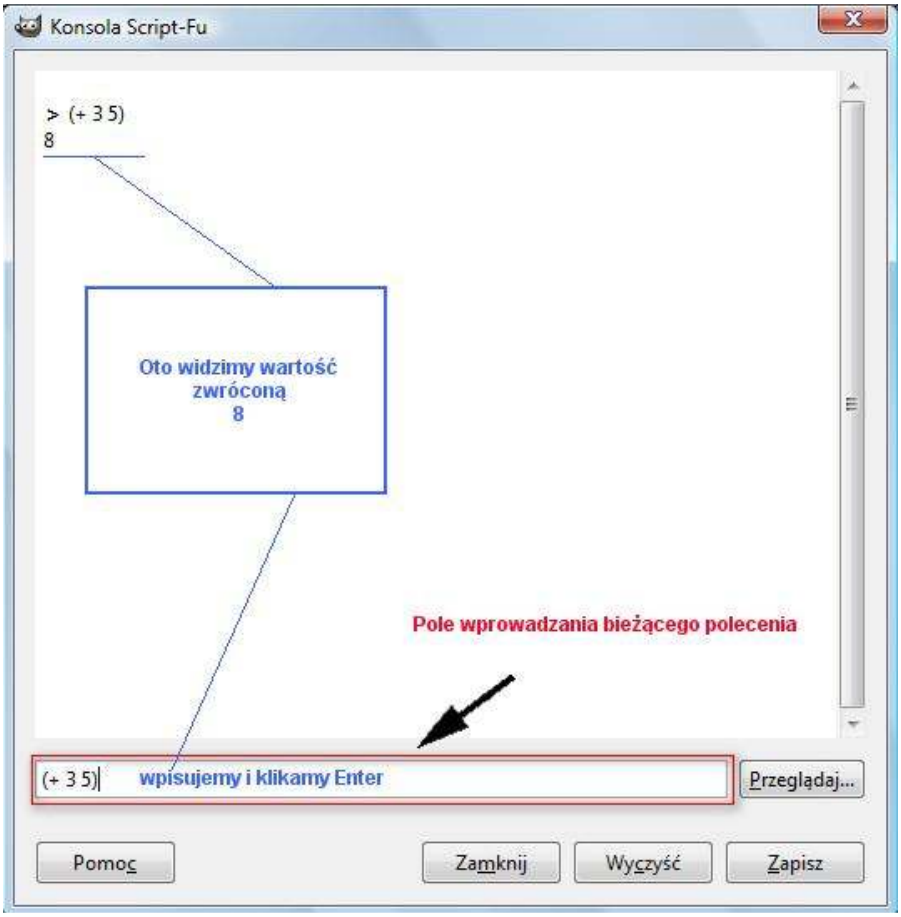
Wybieramy z menu **Filtry => Script-Fu => Konsola**

Uwaga:
W trakcie otwierania okna **Konsola Script-Fu** jest sprawdzana poprawność procedur zainstalowanych Script-Fu.
Jeśli jakiś zainstalowany przez Nas dodatkowo skrypt, ma niepoprawną składnię języka **TinyScheme** pojawi się komunikat (komunikaty) np.:



Może to być np. błąd składniowy, czyli gdzieś brakuje domknięcia nawiasu, lub tych domknięć jest za dużo – można zainstalować freeware Code edytor Notepad++(http://notepad-plus-plus.org/), PSPad (http://www.pspad.com/) lub SciTe (programy dostępne na wiele platform), które prawidłowo rozpoznają składnię języka Scheme wyszukują nawias domykający. Bardzo przydatną funkcją jest zmiana koloru tła pomiędzy odpowiadającymi sobie nawiasami w przypadku gdy kursor znajduje się koło jednego z nich.

Po chwili otworzy się okno **Konsola Script-Fu**.
Konsola Script-Fu pozwala nam pracować interaktywnie w TinyScheme. Pozwoli nam bezpośrednio wprowadzać polecenia (Procedury) i zwraca wartość jako wynik. Najpierw klikamy „Wyczyść” co usunie opis z okna.



Okno Konsola Script-Fu

Na dole tego okna po prawej znajduje się ***Pole wprowadzania bieżącego polecenia***. Tutaj możemy przetestować system interaktywnych poleceń, których przykłady zostaną podane poniżej.
Dlatego okno Konsoli pozostawiamy w GIMP cały czas otwarte.
Zacznijmy od dodawania:
Wpisujemy np. operację matematyczną

```
(+ 3 5)
```

klikamy **Enter** i pojawia się w oknie wartość zwracana **8**.
Inne przykłady znajdziemy poniżej w ramach omawianych podstaw składni.

```
Witamy w TinyScheme
Copyright (c) Dimitrios Souflis
Konsola Script-Fu - Rozwój interaktywny schematu

> (+ 2 (* 2 2))(+ 1 2 3 4)

610
> (let ((+ *)) (+ 3 8))

24
> (+ 3 5 6)
14
> (* (+ 1 2) (sqrt (- 13 4)) 10)
90
> (* (+ 3 4) (/ 5 6))
5.833333333
> (+ 3 (+ 5 6) 7)
21
```

Rzeczy po znaku zachęty **" > "** są tym, co należy wstukać do interpretera, podczas gdy inne rzeczy są odpowiedziami, zwracanymi przez TinyScheme.
Znak **;** jest znakiem komentarza: wszystko od **";"** do końca linii jest ignorowane przez TinyScheme.

WSKAZÓWKA: Gdy przy wpisaniu w ***Pole wprowadzania bieżącego polecenia*** w oknie **Konsola Script-Fu** popełnimy błąd, możemy użyć na klawiaturze strzałki w górę, aby w Polu wprowadzania ponownie pojawiło się ostatnie polecenie. Co umożliwi edytować procedurę ponownie i wprowadzić poprawki. To bardzo oszczędza czas!.

Podstawy TinyScheme

IDENTYFIKATORY

Identyfikatory stosuje się do nazwania i identyfikacji takich elementów programu jak: stałe; zmienne; typy itp. Identyfikatory można tworzyć z liter alfabetu łacińskiego (A-Z i a-z), cyfr (0-9) oraz znaków ! $ % & * + – . / : < = > ? @ ^ _ ~. Dodatkowo, aby uniemożliwić mylenie identyfikatorów ze stałymi liczbowymi, niedozwolone są identyfikatory rozpoczynające się od znaków, od których mogą zaczynać się liczby – czyli od cyfr lub jednego ze znaków: + – ..
Od tej reguły są jednak wyjątki: + – i ... są prawidłowymi identyfikatorami. Niektóre implementacje mogą dopuszczać też użycie innych identyfikatorów, które rozpoczynają się od tych znaków, ale nie są liczbami.
Dodatkowo przyjmuje się następujące konwencje tworzenia identyfikatorów:

- predykaty kończą się znakiem zapytania `?`. W szczególności zapytania o typ zmiennej tworzymy z nazwy typu i znaku zapytania (np. `vector?`).
- nazwy funkcji, które modyfikują swoje argumenty, oznaczamy wykrzyknikiem `!`, np. `set!`
- operatory konwertujące jeden typ na inny oznaczamy `typ1->typ2`
- funkcje działające na wektorach, znakach i łańcuchach oznacza się przedrostkami `vector-`, `char-` i `string-`. Czasem stosowane jest to też do operacji na listach.

Składnia

Składnia (syntax) - zestaw reguł określających budowę poprawnych wyrażeń w języku programowania. Różne elementy języka, jak np. deklaracje i definicje, warunki, podstawienia, selekcje itp. przedstawione są w postaci list. Lista taka składa się z elementów oddzielonych tzw. *białymi znakami* (czyli znakami odstępu, tabulacji lub nowego wiersza) i otoczona jest parą nawiasów. Pierwszym elementem listy jest identyfikator (nazwa) funkcji, kolejnymi są argumenty.

Cała składnia TinyScheme może zostać streszczona w kilku słowach.
Mianowicie, wszystko co interpreter TinyScheme ma przetworzyć jest wyrażeniem – w postaci liczby, symbolu lub listy. Wartością liczby jest ona sama, wartością symbolu jest wartość zmiennej, której nazwą jest dany symbol, a wartością listy – wartość wywołania procedury (funkcji), której nazwa jest pierwszym elementem listy, a argumenty pozostałymi jej elementami (które znowu mogą być dowolnymi wyrażeniami, a więc liczbami, symbolami lub listami).
Konsekwencją tego jest bezwyjątkowo prefiksowa notacja wyrażeń w TinyScheme

PODSTAWOWE OPERACJE MATEMATYCZNE

TinyScheme oferuje szereg stałych, oraz predefiniowanych wyrażeń do obliczeń matematycznych (jak pierwiastkowanie, potęgowanie, logarytmy tylko dziesiętne itp.)
Kilka najbardziej użytecznych zamieszczę poniżej. Możemy wypróbować ich działanie w Konsoli TinyScheme.

Obsługiwane: `exp, log, sin, cos, tan, asin, acos, atan, floor, ceiling, trunc, round and also sqrt and expt when USE_MATH=1. Number-theoretical quotient, remainder and modulo, gcd, lcm.`

Biblioteki: `exact?, inexact?, odd?, even?, zero?, positive?, negative?, exact->inexact. inexact->exact is a core function.`

Pierwszym, najprostszym do omówienia tematem w programowaniu w TinyScheme są działania i przetwarzanie prostych typów danych. Do takich należą przede wszystkim:
liczby `(całkowite i rzeczywiste)` (wyrażenia arytmetyczne), **wyrażenia logiczne** oraz **symbole** czy **ciągi** znaków (łańcuchy).

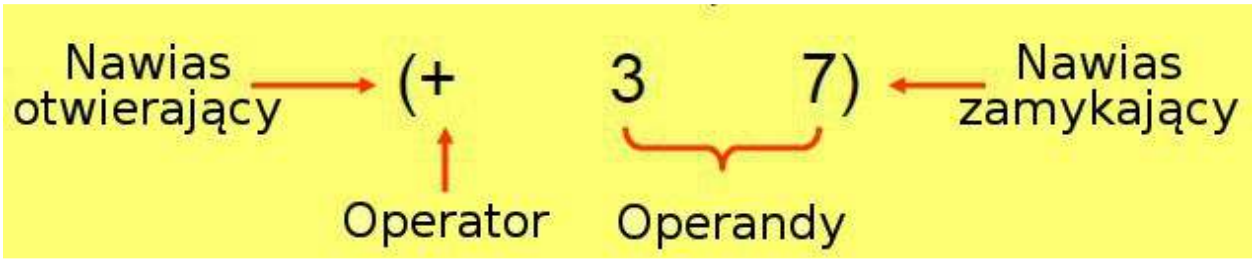
1. W TinyScheme, wszystkie funkcje wywołujemy otaczając je nawiasami **( )**.
2. Na początku wyrażenia podajemy nazwę funkcji lub operatora, a następnie jej argumenty. Działania, tak jak wszystkie inne funkcje, zapisujemy w notacji prefiksowej, to znaczy znak działania przed argumentami.

Przykłady:

1. **Suma** 3 i 7 będzie wyglądała następująco:

```
( + 3 7 )
```

gdzie '+' to nazwa operatora matematycznego, funkcji (**suma**), a 3 i 7 to **operandy** czyli **argumenty** operatora. **Operand** jest to zbiór obiektów (przedmiotów, procesów), na których wykonywane jest pewne działanie.



Uwaga:

Dodatkowe nawiasy

Wiedząc, że operator + może mieć listę liczb do dodania, możemy być skłonni do np. takiego zapisu :

```
(+ 3 (5 6) 7)
```

Jest to jednak błędne; **Pamiętamy**, że każda instrukcja w *TinyScheme* rozpoczyna się i kończy nawiasem, a więc Konsola - interpreter programu pomyśli, że próbujemy połączyć się z funkcją o nazwie " 5 " w drugiej grupie nawiasów, a nie zsumowanie tych liczb przed dodaniem ich do 3.
Poprawnie powyższe polecenie należy zapisać tak:

```
(+ 3 (+ 5 6) 7) > 21
```

Rozmieszczenie

W TinyScheme (w odróżnieniu od innych języków programowania), musimy stosować **White Sapce** "białe znaki" pomiędzy operatorem matematycznym (lub inną nazwą funkcji lub identyfikatorem), aby był on prawidłowo interpretowany przez interpreter systemu Konsola Script-Fu, podobnie pomiędzy argumentami.
"Białe znaki" (takie jak spacje czy tabulatory) np. pomiędzy argumentami nie mają znaczenia, są ignorowane przez interpreter programu *TinyScheme,* a więc mogą być stosowane obficie i przyczynić się do wyjaśnienia i organizowania kodu w skrypcie.

Jeśli pracujemy w oknie **Konsoli TinyScript-Fu**, musimy wpisać całe polecenie w jednej linii, to jest wszystkie procedury między nawiasem otwarcia i zamknięcia muszą być w jednej linii w oknie Konsoli.

2. `(* (+ 1 2) (sqrt (- 13 4)) 10) > 90`

Wartością zwróconą będzie wynik mnożenia.
Jak widać funkcja mnożenia zawiera trzy argumenty; sumę, wynik *square root* = pierwiastek kwadratowy różnicy i liczbę 10. Zwracamy uwagę na ilość niezbędnych nawiasów.

3. `(* (+ 3 4) (/ 5 6)) > 5,8333`

Wartością zwróconą jest wynik mnożenia: sumy i dzielenia.

4. `(+ 2 (* 2 2))(+ 1 2 3 4) > 610`

5. `(let ((+ *)) (+ 3 8)) > 24`

6. `(+ 3 5 6) > 14`

7. `(* 7 6) > 42`

8. `(+ (* 3 4) (/ 1 3)) > 4`

Poniższe wszystkie wyrażenia są równe 42 (lub 42.0).

```
42
(+ 36 6)
(* 3 14)
(- 100 58)
(- (* 1 2 3 4 5) (/ (* (+ 6 7) 8 9) 12))
(/ (silnia 7) (silnia 5))
(/ 596.4 14.2)
(+ 2 (* 3 (+ 4 (- 5 1)) 2 4))
```

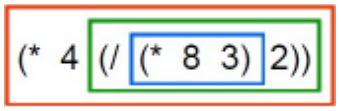
Postępujemy dokładnie tak samo jak w matematyce.

```
(- 1 (* (/ 3 3) 1))
```

Najpierw obliczamy najbardziej zagnieżdżone działania (czyli najpierw wykonywane jest dzielenie 3 przez 3, później mnożenie wyniku tego dzielenia * 1, a dopiero na końcu odejmowanie całego poprzedniego wyrażenia od jedynki na początku. Jest to dokładne wyrażenie kolejności wykonywania działań zgodnej z matematyką, regulowane przez nawiasy.

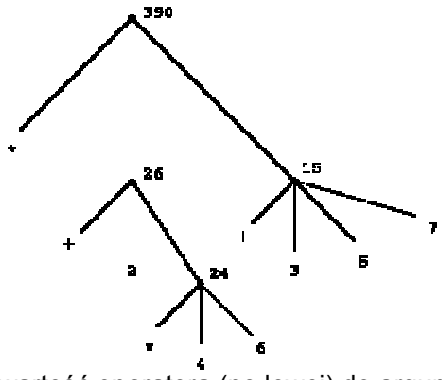
Ilustracja graficzna

```
(* 4  (/ (* 8 3) 2))
```



lub

```
(* (+ 2 (* 4 6))(+ 3 5 7))
```



Zastosowanie procedury, którą jest wartość operatora (po lewej) do argumentów, które są wartościami subargumentów. Uzyskujemy obraz tego procesu, w formie drzewa z 4 kombinacji, jak pokazano na rysunku. Każda kombinacja jest reprezentowana przez węzeł i gałęzie odpowiadające operatorowi i argumentom wynikającym z kombinacji. Węzeł końcowy (czyli węzeł bez odgałęzień) reprezentuje zarówno operator lub cyfr.

Przykłady kilkunastu dalszych procedur:

Jeśli wpiszemy w ***Pole wprowadzania bieżącego polecenia*** jakąś istniejącą w TinyScheme nazwę operatora i "Enter" pojawi się #<…PROCEDURE..nr.> np.:
> exp
#<exp PROCEDURE 52>
#<sqrt PROCEDURE 60>

procedura: `(gcd` *`int ...`*`)`
Procedury zwraca największy wspólny dzielnik lub Najmniejszą wspólną wielokrotność argumentów. Rezultat jest zawsze nieujemny, czyli czynnik -1 jest ignorowany.
Jeśli `gcd` wywołana jest bez argumentów, zwraca 0.

```
(gcd) > 0
(gcd 32 -36) > 4
(gcd 34) > 34
(gcd 33 15) > 3
(gcd 70 -42 28) > 14
```

procedura: `(lcm` *`int ...`*`)najmniejszy wspólny mnożnik`

procedura: `(expt` num_1 num_2`)`

```
(expt 2 10) > 1024
(expt 2 -10) > 0.0009765625
(expt 4 0.5) > 2
(expt 3.0 3) > 27.0
(expt 2 3) > 8
(expt 123 456) proponuję spróbować co zwróci interpreter.
```

procedura: `(sqrt` *`num`*`)`

```
(sqrt 16) > 4
(sqrt 1/4) > 1/2
(sqrt 4.84) > 2.2
(sqrt -4.84) > -1.#IND
```

procedura: `(exp` *`num`*`)`

```
(exp 0.0) ⇒ 1.0
(exp 1.0) ⇒ 2.718281828
(exp -.5) ⇒ 0.606530659
```

procedura: `(log` *`num`*`)log naturalny z num`

```
(log (exp 1)) > 1
(log 1.0) > 0
(/ (log 100) (log 10)) > 2
```

procedura: `(sin` *`num`*`)`
procedura: `(cos` *`num`*`)`
procedura: `(tan` *`num`*`)`

```
argument podajemy w radianach
(sin 0.0) > 0.0
(cos 0.0) > 1.0
(tan 0.0) > 0.0
(tan 0.7854) > 1.000003673
(asin 0.25)  > 0.2526802551
```

```
(acos 0.25)   > 1.318116072
(atan 0.5)    > 0.463647609
```
procedura:
`(remainder 5 3)` - oblicza resztę z dzielenia całkowitego 5 przez 3

`(max A B)` - Wskazuje na większą pośród podanych liczb
```
(max 1 3 4 2 3) > 4
(min 1 3 4 2 3) > 1
(abs -4) > 4
```

Oblicz pole koła o promieniu 5.
`(* (* 5 5) pi)` lub `(* pi (* 5 5)) > 78.53981634`

Wartości i operatory logiczne
Wartości logiczne ***prawda*** i ***fałsz*** reprezentowane są odpowiednio symbolami #t i #f. Możemy na nich wykonywać np. operacje **or** (alternetywa ***lub***), **and** (koniunkcja ***i***) oraz **not** (negacja ***nie***). Mają one jednak odrobinę szersze zastosowanie.
Wartości prawda i fałsz są reprezentowane przez stałe #t i #f. Do budowy wyrażeń logicznych możemy używać standardowych relacji porównywania oraz operacji logicznych:
- and (koniunkcja),
- or (alternatywa) i
- not (negacja).

Relacje porównywania przyjmują dwa lub więcej argumentów i są spełnione, jeśli każde dwa kolejne argumenty są w danej relacji. Operacje and i or przyjmują dowolną liczbę argumentów. Natomiast negacja przyjmuje tylko jeden argument.
Predykat procedury logicznej to: **boolean?** Który sprawdza czy jej argumenty są logiczne.
Procedura `not` neguje argument logiczny.
Przykład
```
(boolean? #t)              >  #t
(boolean? "Hello, World!") >  #f
(not #f)                   >  #t
(not #t)                   >  #f
(and #t (< 1 2 3) (not (= 1 2 1))) > #t
(= (* 2 2) (+ 2 2) 4)              > #t
```

Predykaty
Predykaty to wyrażenia, które w języku TinyScheme mają za zadanie zwracać logiczną wartość (patrz wyżej: "Wartości i oper. logiczne") zapytania które reprezentują w zależności od podanych parametrów.
Są bardzo użyteczne, często występują jako pierwsza część wyrażeń warunkowych i są łatwe w użyciu.
Zwykle kończą się znakiem zapytania (?), co wskazuje że tak jak pytanie zwracają wartość true lub false.
Typy predykatów w TinyScheme
```
boolean?,eof-object?,symbol?,number?,string?,integer?,real?,list?,null?,
char?,port?,input-port?,output-port?,procedure?,pair?,environment?',
vector?. Also closure?, macro?.
```
Przykłady niektórych zaprezentowałem poniżej:
- (number? A) - sprawdza czy A jest liczbą, jeśli tak zwraca **true**, w przeciwnym razie **false**
- (boolean? A) -sprawdza czy A jest typem logicznym (true lub false), jeśli tak zwraca true, w przeciwnym razie false
- (symbol? A) - sprawdza czy A jest symbolem, jeśli tak zwraca true, w przeciwnym razie false
- (struct? A) - sprawdza czy A jest strukturą, jeśli tak zwraca true, w przeciwnym razie false
- (even? X) - weryfikuje czy X jest liczbą parzystą, jeśli tak zwraca true, w przeciwnym razie false
- (odd? X) - weryfikuje czy X jest liczbą nieparzystą, jeśli tak zwraca true, w przeciwnym razie false
- (equal? X Y) - weryfikuje czy liczby X i Y są sobie równe, jeśli tak zwraca true, w przeciwnym razie false
- (symbol=? A B) - weryfikuje czy symbole A i B są sobie równe.
- (string=? A B) - weryfikuje czy stringi A i B są sobie równe

Dwie zmienne są równe jeśli są tą samą liczbą, tym samym symbolem lub wskazują na tę samą parę (w szczególności listę).
Pytania o typ składają się z nazwy typu i znaku zapytania (`?`).
Natomiast predykaty **=**, **<**, **>**, **<=** oraz **>=** odzwierciedlają relacje pomiędzy wszystkimi argumentami.
Przykład
```
(symbol? 'abc) > #t
(symbol? 57)   > #f
(number? 'abc) > #f
(even? 2)      >   #t
```

```
(equal? 4 4)    >    #t
(odd? 3)        >    #t
(boolean? (equal? 6 (* 2 (+ 2 1)))) > #t
```
Ostatni przykład pokazuje jak zastosować wcześniej poznane operacje matematyczne, możemy łączyć wyrażenia. Jak widać na wyjściu pojawiła się zwrócona wartość true, ponieważ nie ważne jaki rezultat przyniosło by equal? to i tak będzie on boolean (zarówno true czy false), ponieważ equal? jako że jest predykantem zwraca również boolean.

Nazwy

Funkcje, operatory, zmienne i słowa kluczowe wszystkie mają w zasadzie te same zasady pisowni. Mogą one składać się z dużych i małych liter, myślników lub znaku łącznika/minusa, podkreślenia i kilka innych znaków interpunkcyjnych.
Nie mogą one zawierać spacji. Nie mogą zawierać nawiasów klamrowych, nawiasów kwadratowych, apostrofów, przecinków, albo cudzysłowów, gdyż wszystkie mają specjalne znaczenie w TinyScheme.

Symbole i Łańcuchy (ciągi) znaków (String)

```
Symbols
     symbol->string, string->symbol
```
Symbole są dość prostym do zrozumienia typem danych. Umożliwiają wykonywanie wielu typów operacji na nich (między innymi ich porównywanie, łączenie, są też dobrym elementem wyrażeń warunkowych).
Generalnie **symbol** w języku TinyScheme traktujemy jako **ciąg** znaków lub wyrazów, *które nie zawierają spacji* (cyfr, wartości logicznych lub niektórych znaków specjalnych jak np. # czy nawias).
W TinyScheme symbole wyróżniamy znakiem cytowania, czyli apostrofu **(')**.

Cytowanie to bardzo poręczny skrót notacyjny do tworzenia struktur listowych, zwłaszcza zawierających symbole.
Umieszczając apostrof ' przed częścią składniową programu (np. fragmentem ujętym w nawiasy) powodujemy, że wszystkie identyfikatory w cytowanym fragmencie będą traktowane jak symbole, a kombinacje nie będą oznaczać wywołań procedur, tylko listy.
Przykładowe symbole to:

- `'mama`
- `'kot`
- `'symbol-nie-zawiera-spacji` (Poprawny zapis ciągu wyrazów bez spacji)
- `'@@`
- `'@`

Cytowanie:
```
'(1 2 3 ( ala ma kota) #t)  > (1 2 3 ( ala ma kota) #t)
(length '(a b c d e f))  > 6
(symbol? 'czy-to-jest-symbol)) > #t
```

String to nazwa typu danych służącego do przechowywania napisów (zmiennych łańcuchowych).
Angielski **String** znaczy dosłownie *ciąg*, *sznur*, więc jednym ze sposobów w jakim można mówić o tym typie to **ciąg znaków**. Innymi określeniami są **łańcuch znaków** lub po prostu **łańcuch**.
Strings - są ciągami znaków, zamkniętymi w doublequotes (") podwójny apostrof (**prosty** cudzysłów).
Aby utworzyć ciąg tekstowy, używamy prostego (a **nie** drukarskiego cudzysłowu ”). Wszystkie znaki pomiędzy nimi są interpretowane jako dane tekstowe, które potem można wstawić do zmiennej.
Ważne jest, aby ciąg kończył się tym samym znakiem, jakim się rozpoczął.
Długość łańcucha jest liczbą znaków, które zawiera. Pierwszy znak łańcucha ma indeks 0, drugi 1 itd..
Przykład:
" To jest skrypt ćwiczeniowy wykonany przez Zbyszka dla chętnych poznania podstaw TinyScheme i pisania Script-Fu"

Jak wpisać poprawnie prosty apostrof (') i podwójny apostrof (") do skryptu w Wordzie?
Przy rozmiarze czcionki 10 lub 12 różnica pomiędzy cudzysłowem prostym a drukarskim, jest niewidoczna, ale po zastosowaniu może okazać się nieprzyjemną pomyłką (czego sam doświadczyłem).
Jak zobaczyć różnicę? Wpisujemy np. podwójny cudzysłów. Teraz zaznaczamy ten znak i powiększamy kilkakrotnie używając skrótu (Ctrl + zamykający nawias kwadratowy). Jeśli zobaczymy ” mamy ustawiony w Word cudzysłów drukarski (a więc także taki apostrof).
Podczas wpisywania tekstu w niektórych programach pakietu Microsoft Office cudzysłowy proste (') lub podwójne (") są automatycznie zamieniane na *cudzysłowy drukarskie*. Tę funkcję można w **Word** włączyć lub wyłączyć.

1. Klikamy **przycisk Microsoft Office**  , a następnie na dole, klikamy przycisk **Opcje programu Word**.
2. Klikamy przycisk **Sprawdzanie**, a następnie klikamy polecenie **Opcje Autokorekty**.

3. W oknie dialogowym **Autokorekta** wykonamy następujące czynności:
   - Klikamy kartę **Autoformatowanie podczas pisania** i w obszarze **Zamień podczas pisania** zaznaczamy lub czyścimy pole wyboru **Cudzysłowy "proste" na „drukarskie”**.
   - Klikamy kartę **Autoformatowanie** i w obszarze **Zamień** wstawiamy zaznaczenie lub czyścimy pole wyboru **Cudzysłowy "proste" na „drukarskie”**.
     Musi być "**proste**", inaczej Konsola wyświetli "**Error:…**".

Uwaga: średnik wewnątrz string nie powoduje komentarza.
"Halo; Wszyscy!" > "Halo; Wszyscy!"

Może się jednak zdarzyć tak, że rozpoczęliśmy ciąg podwójnym apostrofem, ale chcemy umieścić w nim również apostrof (lub podwójny apostrof). Co wtedy?
Nie zadziała i skończy się wygenerowaniem komunikatu błędu.
W takim przypadku konieczne jest znalezienie sposobu na zaznaczenie tych znaków, czyli na *escape* **ucieczkę** od nich. Do ucieczki od tych znaków służy znak lewego ukośnika \ . Termin *sekwencja zmodyfikowana* dotyczy łącznie znaku modyfikacji wraz ze znakiem lub sekwencją po nim następującymi.
Przy użyciu tego znaku, np. apostrof zostaje zapisany jako **\'**, a **\(** będzie oznaczać nawias otwierający dalej **"\$"** oznacza **"$"**; **"*"** oznacza **"*"** itd. Poprzedzenie **znaków specjalnych** odwrotnym ukośnikiem powoduje, że poprzedzanym znakom nie są nadawane żadne dodatkowe znaczenia i oznaczają same siebie.
A więc - kiedy poprzedzimy apostrof znakiem backslash (\ ), **parser** uzna go za część tworzonego ciągu tekstowego.
Przykład:
`"Słowo \'rekurencja\' ma wiele znaczeń." będzie "Słowo 'rekurencja' ma wiele znaczeń."`
Niestety dla podwójnego apostrofu otrzymamy:
`"Słowo \"rekurencja\" ma wiele znaczeń." >`
`"Słowo \"rekurencja\" ma wiele znaczeń."` Ale bez sygnalizacji błędu.
Analiza składniowa (parsowanie, ang. *parsing*) – w informatyce i lingwistyce proces analizy tekstu, w celu ustalenia jego struktury gramatycznej i zgodności z gramatyką języka.
Parser jest najczęściej używany jako część interpretera. Parser sprawdza czy symbole tworzą dopuszczalne wyrażenie. Należy pamiętać, iż znak backslash jest pomijany podczas parsowania.

Strings w TinyScheme

Obsługiwane: `string, make-string, list->string, string-length, string-ref, string-set!, substring, string->list, string-fill!, string-append, string-copy.`

Biblioteki: `string=?, string<?, string>?, string>?, string<=?, string>=?.`
`(No string-ci*? yet). string->number, number->string.`
`Also atom->string,`
`string->atom (not R5RS).`
(atom - jest najprostszym rodzajem wartości - reprezentuje jedno słowo lub nazwę symbolu)

```
;;;; atom?
(define (atom? x)
  (not (pair? x))) ; wg script-fu.init Initialization file for TinyScheme 1.38
```

("-ci") => ("case insensitive") => wielkość liter
Pytania o typ składają się z nazwy typu i znaku zapytania (`?`).
=, <, >, <= oraz >= te predykaty odzwierciedlają relacje pomiędzy wszystkimi argumentami.
Przykłady:
Zamiana pomiędzy typami danych, czyli Konwersja
; zamiana symbolu na łańcuch
procedura: `(symbol->string symbol)`

```
(symbol->string 'aaa)  > "aaa"
(symbol->string 'symbole)  > "symbole"
```

; zamiana łańcucha na symbol (w tym przypadku powstanie symbol)
procedura: `(string->symbol string)`

```
(string->symbol "halo")  > halo
(string->symbol "string")  > string
```

Dane znakowe są reprezentowane przez poprzedzenie znakiem `#\` . Tak więc `#\h` jest znak `h` .
Dane znakowe musimy zapisywać ze spacją pomiędzy nimi.
Czyli nie tak: **`#\h#\a#\l#\o (wartość zwrócona Error)`**
Tylko tak: **`#\h #\a #\l #\o`**
Inne "Białe znaki" to takie jak spacje czy tabulatory: #\space, #\tab, #\newline, #\return.
procedura: `(string char ...)`

```
(string #\h #\a #\l #\o) > "halo"
(string) > ""
(string #\a #\b #\c) > "abc"
(string #\C #\Z #\E #\S #\C) > "CZESC"
```

```
(string->list "halo") > (#\h #\a #\l #\o)
(list->string (list #\A #\p #\p #\l #\e))  > "Apple"

(number->string 16) >  "16"
(string->number "16") > 16
```

procedura: `(make-string n)lub`
procedura: `(make-string n char)`

```
(make-string 0) > ""
(make-string 0 #\x) > ""
(make-string 5 #\x) > "xxxxx"
```

Możliwe jest również, aby wyodrębnić znak z ciągu znaków za pomocą procedury `string-ref`, która przyjmuje ciąg znaków i zwraca znak o liczbie całkowitej *n*. Pierwszy znak łańcucha ma indeks 0, drugi 1 itd.

procedura `string-ref`

```
> (string-ref "cos" 0)
#\c
> (string-ref "cos" 1)
#\o
> (string-ref "cos tam" 10)
indeks 10 poza zakresem [0, 6] dla łańcucha: "cos tam"
```

procedura `(string-set! str n ch)`
umieszcza w łańcuchu **`str`** znak **`ch`** zamiast znaku o numerze indeksu **`n`**

```
> (define str (string #\c #\o #\s #\t #\a #\m))
str
> str
"costam"
> (string-set! str 2 #\d)
"codtam"
> (string-set! str 0 #\b)
"bodtam"
```

procedura: `(string-length string)`
Długość łańcucha jest liczbą znaków które on zawiera (również spacje). Liczba ta jest liczbą całkowitą. Długością pustego łańcucha jest 0. Pierwszy znak łańcucha ma indeks 0, drugi 1 itd.

```
(string-length "123") > 3
(string-length "") > 0
(string-length "1 2 3") > 5 ;bo spacje też zliczane
(string-length "Jest to bardzo proste zdanie") > 28
(string-append "mama" "tata" "ola") > "mamatataola" łączy kilka łańcuchów
(string-length (make-string 1000000)) > 1000000
```

Dalej:
Do porównania łańcuchów
`string=? string<? string<=? string>? string>=?`
np. (string=? A B)- weryfikuje czy łańcuchy A i B są sobie równe

```
(string=? "mom" "mom") > #t
(string<? "mom" "mommy") > #t
(string>? "Dad" "Dad") > #f
(string=? "Mom and Dad" "mom and dad") > #f
(string<? "a" "b" "c") > #t
```

oraz, identycznie ale z wielkością liter

`string-ci=? string-ci<?`

```
(string-ci=? "Mama i Tata" "mama i tata") > #t
(string-ci<=? "say what" "Say What!?") > #t
(string-ci>? "N" "m" "L" "k") > #t
(string-ci=? "Stra\sse" "Strasse") > #t

(string->number "3.4") > 3.4
(number->string 3.4) > "3.4"
(number->string 100) > "100"
```

Substring - usuwa określoną ilość znaków z łańcucha znaków.
procedura `(substring string start end)`
Łańcuch musi być łańcuchem znaków, a początek i koniec muszą być dokładnie liczbami całkowitymi
`0 ≤ start ≤ end ≤ (string-length string).`

`substring` zwraca nowo wydzielony łańcuch znaków, utworzony z wyłączeniem znaku *start* a kończąc na znaku *end* włącznie.

```
(substring "Apple" 1 3) > "pp"
(substring "Apple" 1)   > "pple"
(substring "123456789" 5 7)  > "67"
(substring "hi there" 3 6)   > "the"
```

Brak procedur

```
(string-upcase "Ulica")
Error: eval: unbound variable: string-upcase")
(string-downcase "ULICA")
Error: eval: unbound variable: string-downcase
```

Znakowy typ danych

```
Characters
     integer->char, char->integer –zamiana pomiędzy typami danych konwersja.
     char=?, char<?, char>?, char<=?, char>=?.
     (No char-ci*?)
```

Pytania o typ składają się z nazwy typu, podstawowego operatora logicznego tj. **=** równy, **<** dużo mniejsze, **>** dużo większe,**<=** mniejsze równe, **>=** większe równe i znaku zapytania (**?**).
Jeden z podstawowych typów danych . Jest po prostu jednobajtową liczbą całkowitą.
Jak podano wcześniej, dane znakowe **char** są reprezentowane przez poprzedzenie znakiem `#\` .
Tak więc `#\h` to znak `h` .
Niektóre znaki niegraficzne to bardziej opisowe nazwy, np. #\space, `#\newline`, `#\tab`.
Predykat: znak => char?: `(char? #\a ) > #t   (char? 1 )   > #f`
Przykłady

```
(char-ci=? #\a #\A) > #t
(char-ci<? #\a #\B) > #t
(char=? #\a #\a)  >  #t
(char<? #\a #\b)  >  #t
(char>=? #\a #\b) >  #f
(char=? #\a #\b)  >  #f
```

TinyScheme oferuje wiele procedur konwersji pomiędzy typami danych.
Konwersję wielkości liter za pomocą `char-downcase` i `char-upcase` :

```
(char-downcase #\A)      >    #\a
(char-upcase #\a)        >    #\A
```

Znaki mogą być zamienione na liczby przy użyciu `char->integer` i liczby całkowite mogą zostać zamienione na znaki przy użyciu `integer->char` . (Liczba całkowita odpowiadająca znakowi jest zazwyczaj w kodzie ASCII).
Znaki Hex są dozwolone (np. #\x20 to #\space
Szesnastkowy system liczbowy (czasem nazywany *heksadecymalnym*, skrót **hex**) – pozycyjny system liczbowy, w którym podstawą jest liczba 16. Skrót **hex** pochodzi od angielskiej nazwy *hexadecimal*. Do zapisu liczb w tym systemie potrzebne jest szesnaście cyfr.
http://pl.wikipedia.org/wiki/Szesnastkowy_system_liczbowy

```
(char->integer #\a) >  97
(char->integer #\b) >  98
(char->integer #\c) >  99
(char->integer #\d) >  100
(char->integer #\o) >  111
(char->integer #\x) >  120 itd. patrz http://web.cs.mun.ca/~michael/c/ascii-table.html
(char->integer #\x2) >  2
(char->integer #\x5) >  5
(char->integer #\x19) > 25
(char->integer #\x1e) > 30
(integer->char 0)     > #\x0
(integer->char 1)     > #\x1
(integer->char 50)    >  #\2
```

#b (binary), #o (octal), #d (decimal), i #x (hexadecimal)

```
#b101010              > 42 (w reprezentacji systemu binarnego)
#o52                  > 42 (w reprezentacji ósemkowego systemu liczbowgo to
pozycyjny system liczbowy o podstawie 8)
#x2a                  > 42 (w reprezentacji systemu heksadecymalnego)
```

Uwaga:
Flonums są obecnie tylko do odczytu w systemie dziesiętnym. Pełne wspieranie gramatyki ma być wkrótce.
"Flonum" oznacza liczbę zmiennoprzecinkową.
http://docs.freebsd.org/info/as-all/as-all.info.Flonums.html

WEKTORY

`make-vector, vector, vector-length, vector-ref, vector-set!, list->vector, vector-fill!, vector->list, vector-equal? (funkcje pomocnicze, not R5RS)`

Uwaga: tych procedur nie znajdziemy w Przeglądarce procedur Script-Fu.

Wektory - to podobny do łańcucha ciąg elementów, ale w tym przypadku mogą one być czymkolwiek, a nie tylko typu znakowego. Oczywiście, elementy mogą być także wektorami - umożliwia to stosowanie wielowymiarowych wektorów.

Wektory są to typy danych, wiążące elementy **integer** zaczynając od zera.

Różnica między wektorem i listą jest taka, że wektory zwykle zajmują mniej miejsca niż odpowiadające im listy i czas potrzebny do uzyskania dostępu do elementu losowego jest stały.

Większość procedur operujących na listach są liniowe w czasie.

wektor- tablica jednowymiarowa, czyli taka, co ma tylko jeden wiersz, a kilka kolumn.

Wektory w TinyScheme piszemy przy użyciu notacji procedury `#(elem ...)`, specjalny znak **#** poprzedza zwracany wektor zawarty w nawiasach.

Przykłady:

Wektor o długości 3 elementów, zawierający jako element 0, cyfrę 0 oraz listę (2 3 4 5) jako element 1 i łańcuch "Zbyszek" jako element 2 zapisujemy jako:

```
#(0 (2 3 4 5) "Zbyszek")
```

Długość wektora jest liczbą elementów w nim zawartych.

Długość powyższego wektora jest `3`.

Pamiętamy, że:

W TinyScheme symbole wyróżniamy znakiem cytowania, czyli apostrofu **(')**.

oraz o zmianach:

cons-array - wymieniona przez **make-vector**

aset – wymieniona przez **vector-set!**

aref - wymieniona przez **vector-ref**

DZIAŁANIA NA WEKTORACH

TinyScheme przewiduje predykat **vector?** do określenia, czy dany obiekt jest wektorem, czy nie.

Na przykład:

```
> (vector? '(1 2 3 4)) ;lista, a nie wektor
#f
> (vector? '#(1 2 3 4))
#t
> (vector? '#(alfa beta gamma))
#t
> (vector? '(alfa beta gamma))
#f
> (vector? "alfa beta gamma")  ; łańcuch, a nie wektor
#f
> (vector? '#(#f))  ; wektor jedno elementowy (element #f)
#t
```

Istnieje wiele procedur do tworzenia i modyfikowania wektorów.

Wektory mogą być tworzone z zastosowaniem procedur **vector** i **make-vector** .

Procedura **vector**

pobiera dowolną ilość argumentów i montuje je do wektora i zwraca nowy wektor zawierający argumenty jako elementy wektora.

Przykłady:

```
>(vector 0 1 2 3 4)
zwraca #( 0 1 2 3 4 )
>(vector 'A 'B 'C)
zwraca # (A B C)
> (vector 'a 'b "cos" '(1 2 3 4) '#(inny vector))
#( a b "cos" (1 2 3 4) #( inny vector ) )
>(vector 2 3 8)
#( 2 3 8 )
> (vector 'alfa 'beta 'gamma))
#(alfa beta gamma)
> (vector)  ; pusty wektor - żadnych elementów!
#()
> (vector 'alfa "beta" '(gamma 3) '#(delta 4) (vector 'epsilon))
#(alfa "beta" (gamma 3) #(delta 4) #(epsilon))
```

Ostatni przykład pokazuje, że wektory TinyScheme mogą być *niejednorodne* , zawierające elementy różnych typów, podobnie jak listy.

Procedura **make-vector**
tworzy vector o określonej długości, może mieć następującą składnię:

```
(make-vector n)
(make-vector n obi)
```

Jeśli **obi***ekt* jest podany, to procedura zwraca nowy wektor. W przeciwnym razie zawartość zwróconego wektora będzie nieokreślona.
Jeśli procedura ma obydwa argumenty, liczbę naturalną **n** oraz wartość `obi` wtedy zwraca wektor **n** elementów, w którym każdą pozycję zajmuje **obi** .
Argument **n** to ilość elementów, a
Obi*ektem* może być – `'string (np. aby wstawić napis w tablicy), 'byte, 'double, 'points...`
Przykład:

```
> (make-vector 3)
#( () () () )
> (make-vector 3 (list 1 2 3))
#( (1 2 3) (1 2 3) (1 2 3) )
> (make-vector 3 "cos")
#( "cos" "cos" "cos" )
>(make-vector 4 (/ 1 2))
#(1/2 1/2 1/2 1/2)
>(make-vector 5 'cos)
#(cos cos cos cos cos)
>(make-vector 4 0)
#(0 0 0 0)
>(make-vector 0 4)  ; pusty wektor
#()
```

Utwórz wektor o rozmiarze n elementów wszystkie o wartości v, gdzie n = 3 ilość kanałów o wartości 255.

```
> (make-vector 3 255)
#( 255 255 255 )

> (define a (make-vector 4 'byte))
a
> a
#( byte byte byte byte )
> (vector-set! a 2 42)
#( byte byte 42 byte )
```

Procedura **vector-length**
Długość wektora to liczba zawartych w nim elementów.
Procedura **vector-length**, przyjmuje jako argument wektor i zwraca długość wektora w następujący sposób:
Przykład:

```
> (vector-length '#(a b c d))
4
> (vector-length '#(a b #(a b) '(a b)))
4
> (vector-length '#())
0
```

Procedura (**vector-ref** *vec k*)
Aby odwołać się do elementu wektora, można zastosować procedurę **vector-ref.**
Procedura **`vector-ref`** zawiera dwa argumenty - wektor **`vec`** i liczbą naturalną **`k`**. Zwracany element **`vec`** jest dokładnie elementem o indeksie **`k`** . (Innymi słowy, jeśli **`k`** jest zerem **0**, pojawi się element, którym rozpoczyna się wektor). Indeksy wektora zaczynają się od zera (**0!**).

```
>(vector-ref '#(1 1 2 3 5 8 13 21) 5)
8
> (vector-ref (vector 3 1 4 1 5 9) 4)
5
> (vector-ref '#(1 2 3 4) 2)
3
> (vector-ref '#(1 2 3 4) 0)
1
> (vector-ref (vector 'alfa 'beta 'gamma) 0)
alfa
```

```
> (vector-ref (vector 'alfa 'beta 'gamma) 3)
Error: vector-ref: out of bounds: 3
indeks 3 poza zakresem [0, 2] dla wektora: #(alfa beta gamma)

>(vector-ref '#(1 1 2 3 5 8 13 21)
            (let ((i (round (* 2 (acos -1)))))
              (if (inexact? i)
                  (inexact->exact i)
                  i)))
13
> (define przyklad (vector 2 3 8))
przyklad
> przyklad
#( 2 3 8 )
> (vector-ref przyklad 0)
2
> (vector-ref przyklad 2)
8
> (vector-ref przyklad 3)
Error: vector-ref: out of bounds: 3
```

Procedura **vector-set!**
TinyScheme może także edytować pewien element wektora za pomocą procedury **vector-set!**, ma ona następującą składnię:
`(vector-set! vec n k)`
vector-set! zawiera trzy argumenty - wektor `vec` , indeks wektora **n** oraz liczbę naturalną `k` (która może być też listą). Procedura zastępuje element o indeksie **n** w `vec` przez **k** (lub inaczej zapisuje obiekt *k* w miejsce o indeksie *n* wektora *vec*).
Procedura zmienia stan wektora nieodwracalnie; nie ma sposobu, aby dowiedzieć się co było w tej pozycji, po tym jak została zastąpiona.
To jest konwencja w której wykrzyknik na końcu nazwy procedury jest w rozumieniu "Należy uważać!", bo powoduje nieodwracalne zmiany w stanie obiektu.
Przypominam:
Jeśli **k** jest zerem **0**, pojawi się element, którym rozpoczyna się wektor. Indeksy wektora zaczynają się od zera (**0!**).
Przykład:

```
> (define v (vector 'a 'b 'c 'd))
v
> v
#( a b c d )
> (vector-set! v 0 1);w wektorze v indeks 0 zamienić na 1
#( 1 b c d )
> (vector-set! v 3 '(lista elementow)); indeks 3 zamienic na (lista elementów)
#( 1 b c (lista elementow) )

> (define v #(1 2 3 4))
v
> v
#( 1 2 3 4 )
> (vector-set! v 3 2)
#( 1 2 3 2 )

> (define v #(1 2 3 4 5))
v
> v
#( 1 2 3 4 5 )
>(vector-set! v 2 0)
v
#( 1 2 0 4 5 )

> (define sample-vector  '#('vector 'alfa 'beta 'gamma 'delta 'epsilon))
sample-vector
> sample-vector
#( 'vector 'alfa 'beta 'gamma 'delta 'epsilon )
> (vector-set! sample-vector 2 'zeta)
#( 'vector 'alfa zeta 'gamma 'delta 'epsilon )
```

```
> (vector-set! sample-vector 0 "cos")
#( "cos" 'alpha zeta 'gamma 'delta 'epsilon )

> (vector-set! sample-vector 2 -20)
#( "cos" 'alpha -20 'gamma 'delta 'epsilon )
```

UWAGA, procedury **vector-set!** nie może być stosowany na stałe wektorowych.

Procedury **vector->list**, **list->vector**
Procedura **`vector->list`** przyjmuje wektor jako argument i zwraca listę zawierającą te same elementy w tej samej kolejności, procedura **`list->vector`** wykonuje działanie odwrotne.
Przykłady:

```
> (vector->list '#(1 2 3 4))
(1 2 3 4)
> (list->vector '(1 2 3 4))
#(1 2 3 4)
(vector->list #(1 2 3))
(1 2 3)
> (vector->list '#(pac pac nic))
(pac pac nic)
> (list->vector '(cos cos))
#( cos cos )
> (vector->list (vector))
()
> (list->vector '(#\a #\b #\c))
#(#\a #\b #\c)
> (vector->list #((1 2) (3 4)))
((1 2) (3 4))
> (vector->list #(#(1 2) #(3 4))  )
(#( 1 2 ) #( 3 4 ))
```

Procedura **vector-fill!**
procedura z efektem ubocznym, ma następującą składnię:

```
(vector-fill! vec k)
```

ma dwa argumenty, z których pierwszy musi być wektorem. Procedura zastępuje zawartość każdego elementu wektora, zawartością drugiego argumentu.
Przechowuje *k* w każdym elemencie *vec* .
Wartość zwracana jest nieokreślona.
Przykład:

```
> (define v (vector 1 2 3 4))
 > v
 #(1 2 3 4)
 > (vector-fill! v '(1 2))
 #((1 2) (1 2) (1 2) (1 2))
```

Poniżej wymieniono najczęściej spotykane typy danych, które można spotkać w Script-Fu.

- **INT32** oznacza liczbę całkowitą ***integer*** ze znakiem (bez miejsc po przecinku) o długości 32 bitów np. dla RGBA. Zmienne typu integer nie mogą pamiętać dowolnie dużych liczb całkowitych.
- **FLOAT** jest liczbą z miejscem po przecinku (float.pl = liczba zmiennoprzecinkowa). Dla liczb zmiennoprzecinkowych należy korzystać z oddzielania części dziesiętnych kropką (**.**), a nie przecinkiem (,) np. wartości dla FLOAT OPACITY.
- **STRING** to (łańcuch) i jest używany do tekstów. Wartości, które mają tego typu dane są zawsze pisane w "cudzysłowie".
- **LIST** może zawierać wiele wartości różnych typów danych.
- **ARRAY** może zawierać wiele wartości tego samego typu. Zawsze przychodzi w połączeniu z innym typem danych **INT8ARRAY**.
- **IMAGE** zawiera niepowtarzalny numer identyfikacyjny ID obrazu w GIMP-ie.
- **LAYER** zawiera niepowtarzalny numer identyfikacyjny ID warstw w GIMP-ie.
- **DRAWABLE** zawiera niepowtarzalny numer identyfikacyjny ID wybranego (obrazu, kanału lub warstwy) w programie GIMP. Odnosi się on do warstwy aktywnej.

Teraz musimy poznać, jak tworzyć i wykorzystywać listy oraz jak tworzyć i wykorzystywać zmienne i funkcje.

Pary i Listy w TinyScheme

Budowa pary

Funkcja cons TinyScheme oczekuje dwóch argumentów, które są łączone w pary. W GIMP od 2.4, jeśli drugi argument nie jest obecny wystąpi błąd ("Error: cons: needs 2 argument(s)").
Rozwiązaniem tego problemu jeśli napotkamy, jest jawne włączenie pustej listy jako drugiego argumentu.

Podstawową konstrukcją do budowy złożonych danych jest **para**. W TinyScheme mamy dynamiczną kontrolę typów. Oznacza to, że dopiero w momencie wykonywania jakiejś operacji na danych, sprawdzane jest, czy dane te są odpowiednich typów. Dzięki temu, elementy par mogą być czymkolwiek, w tym również parami i nie jest to narzucone przez żaden system typów.
Tak więc za pomocą par możemy budować listy oraz drzewa (zwane strukturami listowymi).
Podstawowe operacje na parach to:

- cons – *(skrót od - constructs)* procedura tworząca dwuargumentowe pary,
- car -- („*c*ontent of *a*ddress *r*egister“) wskazuje na pierwszy element pary
- cdr -- („*c*ontent of *d*ecrement *r*egister“) wskazuje na drugi element pary

W TinyScheme użycie "car" z pustą listą jest niedopuszczalne i wygeneruje komunikat o błędzie ("Error: car: argument 1 must be: pair").
Użycie w TinyScheme "cdr" z pustą listą lub "cddr" z listą jednoelementową jest niedopuszczalne i wygeneruje komunikat o błędzie ("Error: cdr: argument 1 must be: pair").

Listy

Kluczową rolę dla języka TinyScheme mają listy.
Na ogół w funkcjach potrzebujemy więcej niż dwóch argumentów, np. kolory potrzebują trzech wartości.
Wtedy wszystkie wartości wpisujemy w listę, np. przy zapisie liczb w systemie RGB (red green blue).

```
'(255 127 0)
```

to przykład zapisu koloru pomarańczowego.
Jak już podano znak **'** (pojedynczy znak apostrof) lub quote zawarty przed listą, oznacza - cytowanie - czyli mówi on interpreterowi by nie traktował listy jako wywołanie funkcji 255, lecz jako lista będąca stałą.

```
'(+ 2 3) > (+ 2 3)
```

Wpisany dwa razy pojedynczy apostrof: **''a** zwraca **> 'a =** `(quote a)`

Funkcje przetwarzające listy

Do podstawowych operacji na listach należy tworzenie listy za pomocą dwóch funkcji: `cons` oraz `list` a także:

```
car, cdr, length, map, for-each, foldr, list-tail,
     list-ref, last-pair, reverse, append.
     Also member, memq, memv, based on generic-member, assoc, assq, assv
     based on generic-assoc.
```

procedura: `(cons` obi_1 obi_2`)`
zwraca: nową parę, w której car i cdr są obi_1 i obi_2
Jednym z najważniejszych rodzajów obiektów w TinyScheme jest para, którą można tworzyć za pomocą procedury ***cons***. Procedura konkatenacji *cons* przejmuje dwa argumenty i zwraca listę skonstruowaną z tych argumentów.
Są one dwuelementowymi listami z określonym porządkiem, tj. każda uporządkowana para ma głowę *car*, zwaną też first, oraz ogon *cdr* lub rest. Operacje wyodrębniania głowy lub ogona listy są realizowane przez funkcje systemowe o nazwie odpowiednio *car* oraz *cdr*.
W przypadku listy więcej niż dwu-elementowej głową będzie wiec pierwszy element tej listy, ogonem zaś pozostałe elementy listy.

Procedura **cons** może być użyta do dodania elementu na początek listy.

```
(cons 1 '(2 3))          >       zwraca (1 2 3)
(cons '(1 2) '(3 4))     >       zwraca ((1 2) 3 4)
(cons 0 '(1 2 3))        >       zwraca (0 1 2 3)
(cons 1 2)          >      (1 . 2)  para kropkowa z dwóch atomów cyfrowych
(cons a b)          >      (a . b)  para kropkowa z dwóch atomów literowych
(cons (cons 2 8) 5)      >        ((2 . 8) . 5)
```

Ponieważ w TinyScheme, procedura `cons` wymaga, minimum dwóch elementów, to jeśli lista zawiera jeden element drugi musi zostać utworzony.
Pusta lista może być zdefiniowane jako `'()` lub `()`. Listy mogą zawierać wartości atomowe i inne listy.
Listy są wypisywane jako ciąg wartości elementów, ujęty w nawiasy.
Przykład:
Gdy mamy np. `(cons 3)` to w celu uniknięcia komunikatu Error musimy zastąpić przez

```
 (cons 3 '())      > która zwróci (3)
```

```
(cons 1 nil)      >      (1)  [nil obecnie nie zalecana wymieniona przez '()]
```
lub
```
zastosować procedurę (list 3)      > która zwróci (3)
(cons 1 (cons 3 (cons 5 '())))     >      (1 3 5)
```

Jak podano, do budowy list, oprócz `cons`, dostępna jest procedura `list`, która łączy dowolną liczbę argumentów w jeden łańcuch wyrażeń, a jej wynikiem jest lista zbudowana z tych argumentów.
```
(list 1 2 3)      >      (1 2 3)
(list 1 2 'a 3) > zwraca (1 2 a 3)
(list 1 '(2 3) 4) > zwraca (1 (2 3) 4)
(list 'a 'b 'c)          >       (a b c)
(list? 'x)               >       #f
(list? '(1 2))           >       #t
(list? '(x y))           >       #t
(list? (cons 1 (cons 2 '())))  > #t
```

Skutek działania funkcji `list` ma zawsze o jeden poziom więcej nawiasów niż miało je wyrażenie na wejście.

Należy być ostrożnym i nie mylić *wyrażenia* `(list 1 2 3 4)` z listą `'(1 2 3 4)`.
Aby zrozumieć różnicę z określeniem listy przez apostrof, należy zamienić `(list a b c 4 5)` na `'(a b c 4 5)` i porównać wyniki.
Uwaga:
Jeśli jeden apostrof, już wpisaliśmy wcześniej, wewnątrz listy poprzedzać można opcjonalnie.

Funkcja *append* łączy dwie lub więcej list w jedną.
```
(append '(1 2) '(3 4))          >       zwraca (1 2 3 4)
(append '(1 2 3) '() '(a) '(5 6))   >   zwraca  (1 2 3 a 5 6)
(append '(a (b)) '((c)))            >   zwraca (a (b) (c))
Inne:
(length '(1 2 3))                  >   zwraca 3 funkcja 'length' w TinyScheme
zwraca długość listy.
(reverse '(1 2 3))                  >   zwraca (3 2 1)
(member 'b '(a b c))                >   zwraca (b c)
(assoc 'b '((a 1) (b 2) (c 3)))     >   zwraca (b 2) zarządzanie dołączonymi  listami
(null? '())                         >   zwraca #t
(null? '(A B))                      >   zwraca #f
(null? (car '(a)))                  >   zwraca #f
```

car, cdr i inne

Wszystko ładnie i pięknie, ale jak otrzymać zawartość listy?
Aby to zrobić, mamy dwie funkcje.
Po pierwsze, jak już wspomniano, lista, składa się "*głowy*" i "*ogona*". "*Głową*" nazywamy pierwszy element listy, a "*ogonem*" resztę listy.
Dla przykładu dla listy '(127 0 0) , głową jest 127, a ogonem lista (!) '(0 0).
Funkcja `car` **zwraca** nam "*głowę*", `cdr` "*ogon*". Obie funkcje zakładają, że lista nie jest pusta.
Przykłady:
```
((car (list + - * /)) 2 3) zwraca 5
(car (cons 2 8))        > zwraca 2
(car '(1 2 3 4) )       > zwraca 1
(car '(1) )             > zwraca 1
(cdr '(1) )             > zwraca ()
(cdr '( 1 . 2 ) )        > 2
(cdr '(1 2 3 4) )       > zwraca (2 3 4)
(cdr (cons 2 8))        > zwraca 8
```

Operacje car i cdr nie są zbyt poręczne, bo w jaki sposób uzyskać dostęp do drugiego, trzeciego lub innego elementu listy?
TinyScheme udostępnia kilka "*udogodnień*", czyli skróconą formę zapisu ich złożeń i tak zamiast (car (cdr a)) możemy napisać (cadr a). Dla innych złożeń dostępne są analogiczne skrótowce takie jak:
`- cadr - caddr - cadddr`
dzięki którym można w dosyć prosty sposób dostać się do odpowiednich elementów.
Podstawa konwencji nazewnictwa jest prosta: **a** i **d** reprezentują głowy i ogony listy.
Dla przykładu, aby uzyskać drugi element na liście, wpisujemy następujące:
```
(car (cdr '("Pierwszy" "Drugi") ) )
```
równoważne

```
(cadr '("Pierwszy" "Drugi") )
(car (cdr (cdr '(1 2 3 4)))) > 3
(caddr '(1 2 3 4)) > 3

(cadr '("Pierwszy" "Drugi" "Trzeci" "Czwarty"))       > zwraca "Drugi"
(caddr '("Pierwszy" "Drugi" "Trzeci" "Czwarty"))      > zwraca "Trzeci"
(cadddr '("Pierwszy" "Drugi" "Trzeci" "Czwarty"))     > zwraca "Czwarty"
(caddr (list 1 2 3 4))          > 3
```

Patrzymy jeszcze w **init.scm**

Zadanie - utworzyć listę i powiązać ją z symbolem "ls"

```
      (define ls (list 1 2 3 4 5 6 7))
Po wpisaniu ls zwraca (1 2 3 4 5 6 7)
(reverse ls) > (7 6 5 4 3 2 1)
(length ls) > 7
(car ls) > 1
(cdr ls) > (2 3 4 5 6 7)
(cadr ls) > 2
(caddr ls) > 3
(append ls (list 8 9 10)) > (1 2 3 4 5 6 7 8 9 10)
(reverse ls) > (10 9 8 7 6 5 4 3 2 1)
(symbol? (car ls)) > #f
(list? ls) > #t
(define y ls)> y
Po wpisaniu y zwraca (1 2 3 4 5 6 7)
(list? y) > #t
(eqv? ls y) > #t
(define (f) (list 'symbol-nie-zawiera-spacji))
```

procedura: **map** *proc list1 list2* ...
procedura podejmuje tyle argumentów, ile jest *list* i zwraca pojedynczą wartość. Jeśli podana jest więcej niż jedna *lista*, to wszystkie **muszą być tej samej długości**.
"map" stosuje element *proc*, do elementów *list* i zwraca uporządkowaną listę wyników.
Kolejności dynamiczna, w jakiej *proc* stosuje się do elementów *list* jest nieokreślona.
Jeśli dwie albo więcej list są dane jako argumenty do map-owania, procedura będzie odniesiona do każdego n-tego elementu list, np.:

```
(map + '(1 2 3) '(1 2 3))       > (2 4 6)
(map + '(1 2 3) '(4 5 6))       >  (5 7 9)
(map * '(1 2 3) '(1 2 3) '(1 2 3))     > (1 8 27)
(map car '((a b) (c d) (e f)))      >      (a c e)
(map cadr '((a b) (d e) (g h)))     >       (b e h)
(map (lambda (x) (* x (+ 1 x))) '(1 3 6 9)) > (2 12 42 90) ; obliczy listę
wartości wyrażenia x*(+ 1 x) kolejno dla 1, 3, 6 i 9
(map (lambda (n) (expt n n))'(1 2 3 4 5 6)) > (1 4 27 256 3125 46656)

(define map1
  (lambda (p ls)
    (if (null? ls)
        '()
        (cons (p (car ls))
              (map1 p (cdr ls))))))
(map1 abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)

> (map cons '(1 2 3) '(10 20 30))
((1 . 10) (2 . 20) (3 . 30))
> (map + '(1 2 3) '(10 20 30))
(11 22 33)
```

Funkcje Script-Fu zwracają wartości w postaci list, co czyni `car` jedną z najbardziej użytecznych funkcji. Np. dla funkcji **`gimp-new-image`** czy **`gimp-new-layer`** (które za chwilę użyjemy), wartością zwracaną jest tylko jeden element, ale ponieważ zawiera się on w liście, dostęp do niego mamy dzięki funkcji `car`

Procedury i funkcje to wyodrębnione części programu (podprogramy), stanowiące pewną całość, posiadające jednoznaczną nazwę i ustalony sposób wymiany informacji z pozostałymi częściami programu. Są stosowane do wykonania czynności wielokrotnie powtarzanych przez dany program.

Różnice między funkcją a procedurą
• sposób przekazywania wartości
• odmienne sposoby wywołania
• zadaniem procedury jest wykonanie pewnej sekwencji czynności, polegających zwykle na obliczaniu jednej lub wielu wartości
• zadaniem funkcji jest obliczenie jednej wartości

Funkcje

Pomału dobrnęliśmy do momentu, kiedy już wiemy o zmiennych, listach i innych, przejdźmy do pracy z funkcjami.
Funkcje są bardzo ważnym elementem. W funkcjach umieszcza się fragment kodu, który może być używany w różnych momentach i miejscach programu, czasem nawet w różnych kontekstach. Funkcja stanowi swego rodzaju czarną skrzynkę — nie interesuje nas najczęściej, co jest w środku, ważne jest, jakie informacje funkcji przekażemy i jaką wartość funkcja zwróci.

Poza funkcjami arytmetycznymi (procedury prymitywne), TinyScheme ma w bibliotece wbudowanych wiele innych procedur lub funkcji.

Biblioteka

- W TinyScheme powtarza się znaczna liczba różnych elementarnych zadań.
  W celu ułatwienia pracy programisty wiele z nich zostało już wcześniej zaprogramowane i zebrane w **bibliotece**, dołączonej do interpretatora.
- Biblioteka zawiera przede wszystkim gotowe **funkcje i procedury**
- W celu skorzystania z funkcji lub procedury bibliotecznej, należy wcześniej albo samodzielnie wpisać jej typ albo dokonać importu **pliku** za pomocą Przeglądarki procedur.
- Domyślnie do wyszukania funkcji lub procedury, służy przycisk "**Zastosuj**" na dole Przeglądarki, który wklei ją do interpretatora **Konsola Script-Fu.**

Każda z procedur tam zawartych ma odpowiednik w postaci funkcji TinyScheme. Mamy do dyspozycji "Wewnętrzne procedury GIMP" oraz "Procedury tymczasowe".

Funkcja to wydzielony fragment programu (podpogram), któremu nadano **nazwę**.
Może on zawierać **deklaracje lokalne** oraz **instrukcje**.
Funkcję można wywołać w dowolnym miejscu programu podając tylko jej nazwę z nawiasami ( )
Funkcji używa się w sytuacji, gdy:

- Pewien fragment programu powtarza się kilkakrotnie w jednakowej lub zbliżonej formie
- Program jest długi i skomplikowany, dobrze jest wtedy podzielić go na mniejsze zadania, czyli budować całość z mniejszych „klocków”
- W TinyScheme działanie funkcji wiąże się z wygenerowaniem (**zwracaniem**) pewnej **wartości**
- Jeżeli funkcja ma nie zwracać żadnej wartości, można ją wtedy nazwać **procedurą**.

W celu uruchomienia gotowej funkcji należy umieścić jej wywołanie wewnątrz innej funkcji
Większość funkcji potrzebuje do swego działania **danych wejściowych**. Przekazanie ich **do** funkcji może odbywać się poprzez zmienne lokalne, parametry wywołania funkcji.

- Dane wyjściowe, "wyprodukowane" przez funkcję wywoływaną, można przekazać do funkcji wywołującej poprzez:
  - Wartość zwracaną (tylko jedna wartość)
  - Symbole definiowane wewnątrz funkcji, nazywane są **lokalnymi**
  - **Parametry wywołania** funkcji pozwalają na przekazanie danych wejściowych dla funkcji w sposób jawny i czytelny. Jeżeli funkcja została zadeklarowana z parametrami, podczas wywołania **muszą** być one podane
  - Liczba wartości podanych w wywołaniu i ich typy muszą być zgodne z nagłówkiem funkcji
  - Parametry funkcji można rozumieć jako deklaracje zmiennych lokalnych

Wszystkie funkcje w TinyScheme mają formę:

```
(foo param1 param2 ...)
```

w którym *foo* "coś" jest nonsensownym określeniem rodzaju zastępczego (formalnie znane jako *zmienna metasyntactic*) dla procedury/funkcji, które zostaną przedstawione podczas korzystania z tego szablonu w celu określenia rzeczywistego polecenia.
(`"param1"` i `"param2"` są argumentami lub informacjami które można zdefiniować i które będą przekazywane wraz z poleceniem).
Przykładowo procedury, czy funkcje *wbudowane* (biblioteczne) to funkcje, które są zawsze dostępne do wywołania w TinyScheme np.:

```
(Gimp-image-new 200 300 RGB)
```

Tworzy nowy obraz o wymiarach 200 x 300 typu RGB

Całe środowisko TinyScheme opiera się na funkcjach. Najważniejsze co musimy o nich wiedzieć to to, że funkcje zwracają wartości ID!.

W TinyScheme mamy też możliwość, wiązać nazwę z wartością lub własnych funkcji z wyrażeniem za pomocą słowa kluczowego ***define.***

W pierwszym przypadku powstają nowo nazwane stałe (nie zmienne!), np.:

`(define nasze-pi 3.14)` lub `(define width 400)`

Ma ona postać kombinacji trzech elementów, z których pierwszy to słowo kluczowe `define`, drugi element to nazwa definiowanej stałej, a jej wartość określa trzeci element.

W drugim przypadku, definiujemy procedury, jako parametry potrzebne są dwie listy:
prototyp funkcji i wyrażenie (lub wyrażenia), z którymi nazwa jest wiązana, np.:

```
(define (pole-kola r) (* nasze-pi (* r r)))
```

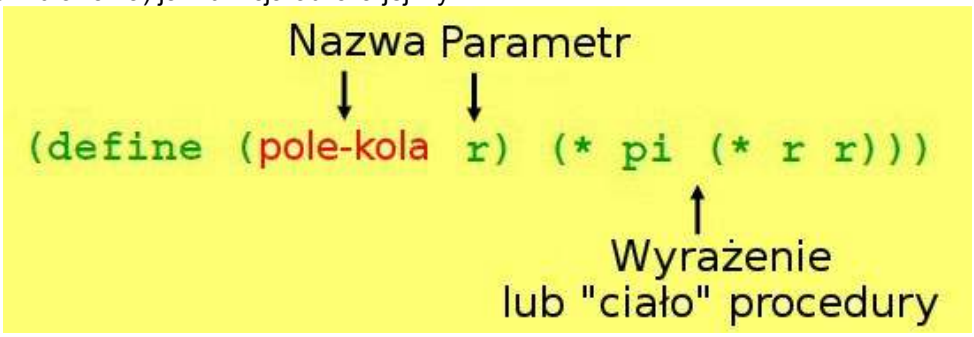
czyli

```
(define ([nazwa funkcji] [parametry formalne par1...par n]) (ciało funkcji))
```

lub

```
(define(name param-list)wyrażenia)
```

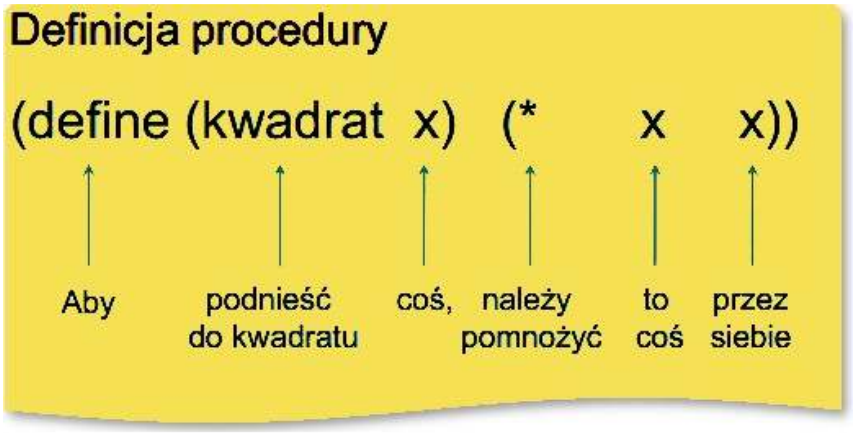
gdzie *name* jest nazwą funkcji przypisaną do tej procedury, *param-list* jest rozdzielana spacjami lista parametrów funkcji (*minimum jeden*), i *ciało funkcji* – *wyrażenia*.
Zauważymy, że przed NazwaFunkcji występuje nawias, tworzy on nagłówek funkcji (który kończy się po ostatnim parametrze zamknięciem nawiasu), następny nawias tworzy właściwą część funkcji, czyli ciało funkcji (lub wyrażenia) miejsce na procedury. Ciało funkcji (body) jest wywołaniem funkcji, czyli wszystkie instrukcje (sformułowania) jak funkcja oblicza jej wyniki.



`area-of-disk => pole koła`

Ciało funkcji jest najważniejszą częścią definicji, gdyż stwierdza, co faktycznie powinna ***robić*** funkcja. Można wywoływać tę funkcję, albo wprost albo za pośrednictwem innej funkcji. Gdy się tak dzieje, mówimy, że funkcja jest ***rekurencyjna***. Do użycia potrzeba jest tylko wiedza o jej nazwie oraz parametrach wywołania. Ciało funkcji niezbędne jest dopiero na etapie tzw. linkowania, a więc może znajdować się np. w bibliotece.
Zwrócę uwagę na jeszcze jedno:
W drugim przypadku, kiedy pierwszy argument "`define`" jest zamknięty w nawiasach (Nazwa i Parametr/y), nie jest oceniany i jest traktowany jako symbol, który zostanie określony (symbol w tym przypadku jest nazwą funkcji).
W tej sytuacji, kolejne parametry, traktowane są jak gdyby "cytowane" i **nie** ocenie (są one przechowywane w celu późniejszego wykonania, gdy wywołamy nazwę "**funkcji**").
Gdy w Konsoli wywołamy nazwę samej funkcji, **czasami** zostanie wyświetlone "# <CLOSURE>" czyli "***Domknięcie***" co znaczy, że funkcja jest przechowywana.
Jeśli pojawiają się nazwy zmiennych po nazwie funkcji, są one traktowane jako zdefiniowane parametry funkcji (i zostaną zastąpione, gdy funkcja zostanie ostatecznie oszacowana).
Domknięcie - jest to obiekt wiążący funkcję oraz środowisko, w jakim ta funkcja ma działać.
Realizacja domknięcia jest zdeterminowana przez język, jak również przez kompilator.
Domknięcia występują głównie w językach funkcyjnych, w których funkcje mogą zwracać inne funkcje, wykorzystujące zmienne utworzone lokalnie.

```
> (define (f x y) (+ x y))
f
> f
#<CLOSURE>
> (f 5 10)
15
```




```
(define (kwadrat x) (* x x))
kwadrat
> (kwadrat 5)
25
> (kwadrat (- 5 2))
9
> (kwadrat (+ 5 2))
49
> (kwadrat (* 2 5))
100
```

Przykłady zastosowania:
Procedura utworzenia funkcji użytkownika z jednym parametrem i jej późniejsze wykorzystanie:
wartość wyrażenia stałej **pi** zdefiniowanej w TinyScheme jest dość dokładna, daleko poza setną część liczby (3.141592654). W przykładach nie potrzebujemy tak dokładnej wartości, wystarczy wynik do setnej części np. 21.98 zamiast 21.99114858.
Nazwy stałej lub funkcji muszą być jednoznaczne - czytelne.

Przypomnienie:
Nazwy (np. funkcje) to **symbole,** w języku TinyScheme traktujemy jako **ciąg** znaków lub wyrazów, *które nie zawierają spacji* – odstępu, pomiędzy dwoma wyrazami. Mogą one składać się z dużych i małych liter, myślników lub podkreślenia. Nazwy funkcji zdefiniowane przez użytkownika mogą zawierać prawie każdy nie alfanumeryczny znak np. typu operatora logicznego (jest tylko kilka wyjątków, wymienionych w R^5RS).

Zapis **nasze-pi** jest jednoznaczny i czytelny.
Zdefiniujmy **nasze-pi**, o wartości 3.14.

```
(define nasze-pi 3.14)
```

Gdy potem w ***Pole wprowadzania bieżącego polecenia*** wpiszemy zdefiniowaną nazwę funkcji "nasze-pi" wartością zwróconą będzie 3.14, czyli wynikiem jest liczba z drugim miejscem po przecinku.
Od tej chwili Interpreter TinyScheme potraktuje naszą "stałą" jak pi.

Uwaga:
Ponieważ nie zapisujemy definiowanych w ramach przykładu stałych lub funkcji będą one w pamięci, tylko podczas danej sesji otwarcia Konsoli Script-Fu.

Teraz spróbujmy stworzyć działanie obliczające pole koła o określonym promieniu, używając definicji **nasze-pi**.
Zapiszemy to wpisując kolejno do KSF:

```
; definiujemy nasze-pi (komentarz)
> (define nasze-pi 3.14)
nasze-pi ;wpisujemy
> nasze-pi
3.14
; definiujemy pole-kola (komentarz)
> (define (pole-kola r) (* nasze-pi (* r r)))
pole-kola
> (pole-kola 10) ;wpisujemy
314 - wartość zwrócona
> (pole-kola 30)
2826
```

Zdefiniowane powyżej procedury mogą być połączone w nową, mocniejszą procedurę.
Czyli możemy utworzyć funkcję wielo parametryczną:

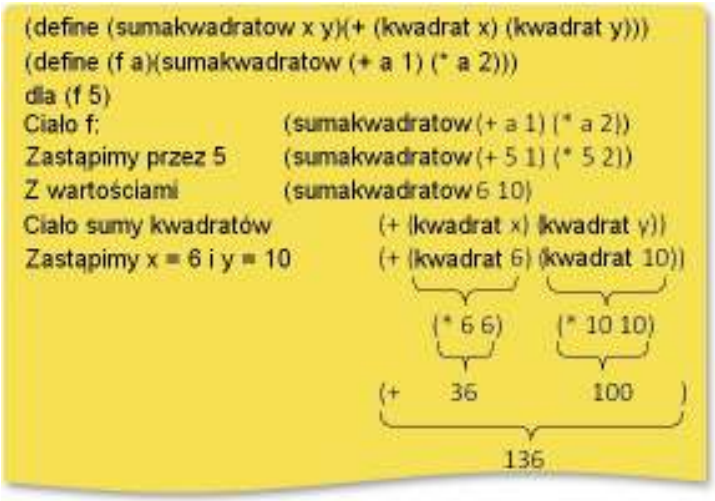
Obliczenie powierzchni pierścienia

```
(define (pole-piersc zew wew)
(- (* nasze-pi (* zew zew))
(* nasze-pi (* wew wew))))
pole-piersc
> pole-piersc
#<CLOSURE>
> (pole-piersc 30 10)
2512
```

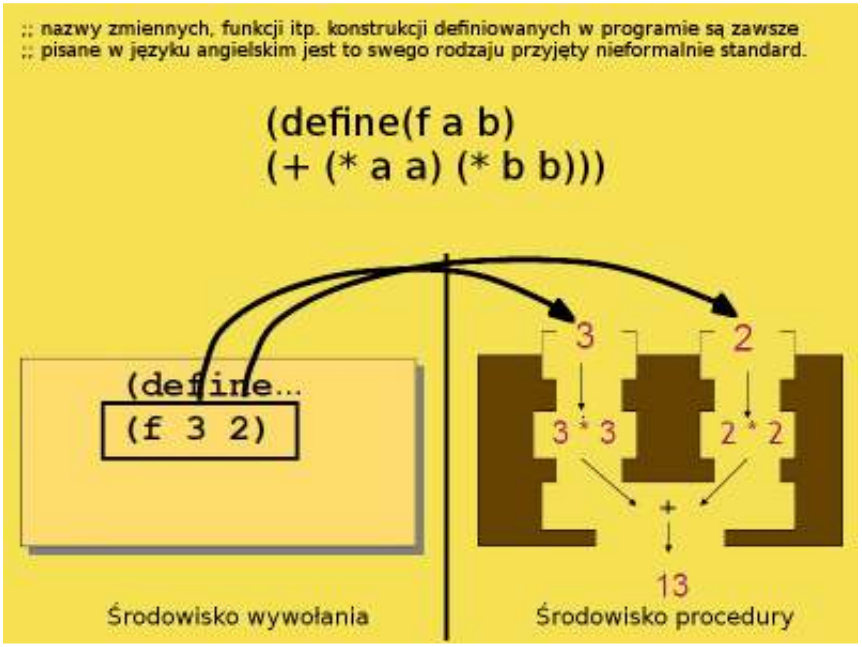
Inne zadanie

Obliczyć pole prostokąta o bokach a i umownym b, który jest dwa razy większy od a.
Ciało funkcji, zakłada, że podajemy tylko parametr a, z którego jest obliczamy b.

```
(define (PoleProstokata a) (* a (* a 2)))
PoleProstokata
> (PoleProstokata 5)
50
(define dlugosc-boku 20)
> dlugosc-boku
(define pole-pow (* dlugosc-boku dlugosc-boku))
pole-pow
> pole-pow
400

(define a 6)
> a
(define b (+ a 1))
> b
(* a b) > 42

>(define (dodaj x y) (+ x y))
dodaj
> dodaj
#<CLOSURE>
>(dodaj 2 3)
5
(define (razy2 x) (* 2 x))
razy2
>(razy2 4)
8
(define (f x) (+ 3 x))
f
> (f 3)
6
(define (g x) (* 3 x))
g
> (g 3)
9
(define (i v) (+ (* v v) (* v v)))
i
> (i 5)
50
(define (k w) (* (h w) (i w)))
k
> (k 5)
650
(define (sumakwadratow x y)(+ (kwadrat x) (kwadrat y)))
sumakwadratow
> (sumakwadratow 5 4)
41
(define (f a)(sumakwadratow (+ a 1) (* a 2)))
f
> (f 5)
136
```




```
(define(f a b)(+ (* a a) (* b b)))
zwraca f
Wprowadzam:
(f 3 2) zwraca 13
```




```
(define (srednia a b) (/ (+ a b) 2))
> srednia
(srednia 5 10) > 7.5

(define (f x y z . 1) (...))
```

oznacza, że funkcja `f` ma co najmniej trzy argumenty (x, y i z). Jeżeli `f` dostaje więcej niż trzy argumenty, są one pisane do listy `1`.

Pamiętajmy że, gdy symbole są "*define"*, są one przypisane, do całego Script-fu. Jeśli Nasz program używa zmiennej o nazwie "b" i wywołamy inny podprogram, który również "define" zmienną "b", zmienna zostanie zmienione przez ten inny program.
Aby tego uniknąć, w Script-fu stosujemy zmienne lokalne let*. **Zmienne lokalne** - to zmienne widoczne tylko w obrębie danej procedury, funkcji lub bloku, tworzone w momencie inicjowania tej procedury, funkcji lub bloku, a usuwane w momencie jego zakończenia.

Funkcje anonimowe

Oprócz powyższej funkcji TinyScheme pozwala również tworzyć i używać form tzw. funkcji anonimowych. Zmienne są tworzone w specjalnym formacie, z wykorzystaniem *słowa kluczowego* **`lambda`** , którego nazwa pochodzi od słynnej teorii rachunku lambda (A. Church), który jest bardzo ważny zarówno w teoretycznej informatyce oraz w innych dziedzinach informatyki.
Właściwy zapis funkcji anonimowej sam w sobie nie bardzo różni się od zapisu powyższej funkcji nazwanej - jedyną różnicą jest to, że w składni podczas pisania w specjalnej formie ***lambda*** nigdzie nie zobaczymy nazwy funkcji, tylko listę (-y) formalnych parametrów, a następnie ciało funkcji.
Funkcję definiuje się za pomocą konstrukcji:

```
(lambda (lista argumentów) (ciało funkcji)),
```

```
(lambda (param1 param2 ...) wyr1 wyr2 ...)
```

lambda - ma postać kombinacji trzech elementów.
Pierwszy z nich to słowo kluczowe `lambda`. Drugi, to lista argumentów - parametrów formalnych ujęta w nawiasy. Natomiast trzeci, pozostałe wyrażenia, na podstawie którego/ych wartościowana jest funkcja (wartością zwracaną jest wartość ostatniego obliczonego wyrażenia) - opisują działanie tej procedury – to treść `lambdy`.
Jak widać `lambdę` zapisujemy podobnie do *define*, tylko z pominięciem nazwy procedury (jako anonimową) czyli używając innego słowa kluczowego.
Funkcja identycznościowa to `(lambda (x) x)`,
a funkcja podnosząca liczbę do kwadratu – `(lambda (x) (* x x))`,
x jest parametrem formalnym (zmienną związaną) tych funkcji.

```
((lambda (x) (* x x)) 6)
```

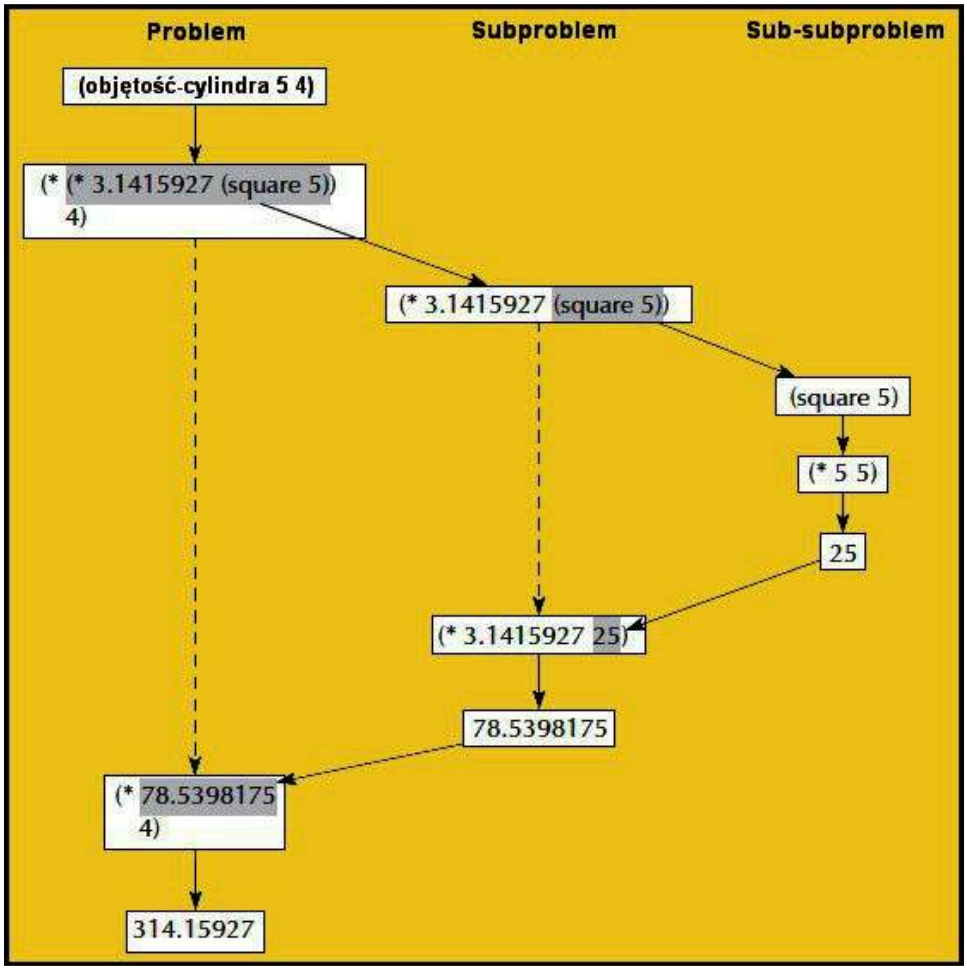
Wywołanie powyższego wyświetli wynik 36:
Wynik funkcji jest po prostu wartością wyrażenia występującego w `lambda`.

```
> ((lambda (i) (* i 2)) 2)
4
```

Przy ocenie pierwszy element przekształca się w funkcję, która akceptuje jeden parametr i wymnoży go przez dwa. Druga części listy, zostanie do niej przekazana jako parametr. Nowo utworzona funkcja się wtedy niezwłocznie wykona, zwróci wynik ***i jest usuwana z pamięci.*** Aby zdefiniować funkcję do wielu zastosowań (co niewątpliwie będziemy chcieli, musimy jednak użyć formy *define*.

Dalsze przykłady:

```
(define double (lambda (x) (* x 2)))
(define abs (lambda (x) (if (> x 0) x (- x))))
> abs
((lambda (x) (+ x 2)) 5)
> 7
(define square(lambda (x) (* x x)))
Teraz kiedy chcemy podnieść liczbę do kwadratu używamy:
(square 3)
> 9

((lambda (x) (+ x x)) 4)  >  8
((lambda (x) (* x x))(+ 3 5)) > 64
((lambda (x y) (+ x y)) 3 5) > 8
(eqv? (lambda () 1) (lambda () 2))  > #f
(let* ((p (lambda (x) x)))(eqv? p p)) > #t

(define objętość-cylindra
(lambda (promień wysokość)
(* (* 3.1415927 (square promień))
wysokość)))
(objętość-cylindra 5 4)
314.15927
(objętość-cylindra 5 4)
(* (* 3.1415927 (square 5)) 4)
(* (* 3.1415927 (* 5 5)) 4)
(* (* 3.1415927 25) 4)
(* 78.5398175 4)
314.15927
```




```
> (define powiel
(lambda (skalowanie)
(lambda (x)(* x skalowanie))))
(define podwoić (powiel 2))
(define potroić (powiel 3))
```

teraz Enter

```
> powielpodwoićpotroić
> (podwoić 7)
14
> (potroić 10)
30
```

Kiedy oceniamy definicję *powiel*, zewnętrzne wyrażenie lambda jest natychmiast oceniane, i jego wartość o nazwie procedury *powiel*.
Ta procedura będzie czekać na to, co przekaże *skalowanie*.
Kiedy oceniamy

```
(define podwoić ( powiel 2))
```

ciało procedury o nazwie *powiel* jest oceniane, a wartość 2 zastępuje *skalowanie*.
Innymi słowy, wyrażenie `(lambda (x) (* x skalowanie))` jest oceniane, z wartością 2 zastępującą *skalowanie*.
Efektem tej oceny jest procedura o nazwie *podwoić*, tak jakby definicja była:
`(define podwoić (lambda (x) (* x 2)))`.
Kiedy do *podwoić* zastosujemy 7, to zastosujemy 7 do procedury `(lambda (x) (* x 2))`, a wynikiem oczywiście jest 14.

Między funkcjami nazwanymi i anonimowymi, istnieje bardzo ścisły związek, który można opisać za pomocą prostego przykładu. Rozważmy funkcję użytkownika o nazwie **dodaj**, która dodaje swoje dwa parametry (dla uproszczenia, przyjmiemy te parametry jako cyfry) i zwraca sumę wartości obu parametrów.
Definicja takiej funkcji jest bardzo prosta:

```
> (define (dodaj x y) (+ x y))
; sprawdzam
(dodaj 1 2)
3
```

Pokazany powyżej zapis jest ekwiwalentny z poniższym zapisem, w którym tworzymy zmienną nazwaną **plus**, która jako swoją wartość zawiera funkcję (anonimową).

Funkcja może być wykorzystana w tych samych miejscach co inne typy wartości, tak, że prawidłowy będzie zapis:

```
> (define dodaj (lambda (x y) (+ x y)))
; sprawdzam
(dodaj 1 2)
3
```

Oceńmy wyrażenia `((lambda (x y) (+ (* x x) (* y y))) 3 4)`.
Rozwiązanie. Dane jest wyrażenie w postaci listy trzech wyrażeń:
Pierwszym z nich jest wyrażenie lambda
`(lambda (x y) (+ (* x x) (* y y)))`, którego formalne parametry tworzą listę (x y), a ciałem procedury jest wyrażenie `(+ (* x x) (* y y)`.
Dalsze dwa wyrażenia to cyfry `3` i `4`, które są również argumentami wywołania procedury. Z argumentów wywołania powstaje lista (`3 4`) a ta unifikuje się z listą formalnych parametrów procedury (`x y`).
Wynikiem jest substytucja (podstawienie) x = 3, y = 4. Zastosowanie tego podstawienia na ciało procedury prowadzi do wyrażenia `(+ (* 3 3) (* 4 4))`.
Jako ocenę dostaniemy liczbę 25, jako wartość wyrażenia w zadaniu.
Po zastosowaniu uzyskanego podstawienia na <ciało> wyrażenia lambda, ocenia się indywidualne wyrażenia ciała procedury w kolejności od lewej do prawej.

Oceńmy wywołanie procedury `((lambda x (cdr x)) 1 2 3 4)`.
Rozwiązanie. Lista argumentów wywołania (1 2 3 4) unifikuje się z parametrami formalnymi procedury tj. ze zmienną x. Dostaniemy podstawienie x = (1 2 3 4). Po zastosowaniu tego podstawienia na ciało procedury (cdr x) dostaniemy wartość (2 3 4).

Ze względu na to, że specjalny format ***`define`*** łatwo można zawsze zapisać za pomocą specjalnej formy ***`lambda`*** jest w TinyScheme możliwe określenie funkcji, która akceptuje zmienną ilość parametrów (np. w skrajnych przypadkach nawet zero), z których jest przy wywołaniu funkcji automatycznie tworzona lista, którą można w ciele funkcji manipulować w dowolny sposób.
Składniowo definicja funkcji wygląda w następujący sposób:

```
(define (nazwa funkcji . parametr) [ciało funkcji])
```

Co jest ekwiwalentem zapisu:

```
(define nazwa funkcji (lambda parametr [ciało funkcji]))
```

Przykład zastosowania:

```
; funkcja zwracająca liczbę faktycznie przekazanych parametrów
(define (foo . parametry) (length parametry))
>(foo 1 2 3 4)
4
; alternatywna forma zapisu
(define foo (lambda parametry (length parametry)))
; wywołanie funkcji z trzema parametrami (chodzi tutaj o trójkę symboli)
>(foo 'a 'b 'c)
3
; wywołanie funkcji bez parametrów
>(foo)
0
```

Uwaga: przy pisaniu Script-Fu forma anonimowa *lambda* jest stosowana bardzo rzadko, spotkałem tyko kilka takich przypadków, ale warto o niej coś wiedzieć.

Zmienne lokalne

Zmienne lokalne - to zmienne widoczne tylko w obrębie danej procedury, funkcji lub bloku (instrukcji sekwencji), tworzone w momencie inicjowania tej procedury, funkcji lub bloku, a usuwane w momencie jego zakończenia. **Inaczej** każda *zmienna lokalna* funkcji, zaczyna żyć dopiero w chwili wywołania funkcji, a zanika po jej zakończeniu.
Script-Fu oparty na TinyScheme zezwala na poprawne deklarowanie i ustawianie zmiennych (definicja listy zmiennych lokalnych i zakresu, w jakim są one aktywne) za pomocą użycie procedury **`let*`**.
Uwaga: nie mylić z ***`let`***.
Składnia wyrażenia `let*` ma postać kombinacji złożonej z trzech elementów.

```
  (let* ( variables )
         expressions )
```

lub zapis
`(let* ( zmienne )wyrażenie` które zwraca inną wartość `)`
Pierwszy element to *słowo kluczowe* `let*`. Drugi element, to lista par definiujących lokalnie wprowadzane zmienne, ujęte w nawiasy. Natomiast trzeci element, to wyrażenie (ciało), którego wartość stanowi wartość całego wyrażenia. Para definiująca lokalnie wprowadzaną zmienną składa się z nazwy tej zmiennej oraz wyrażenia określającego jej wartość, ujętych w nawiasy.

Wygląda to jak w poniższym przykładzie, gdzie deklarujmy dwie zmienne o nazwach **a** i **b**, oraz odpowiednio ich wartościach 5 i 14:

```
(let* ( ( a 5 )
( b 14 ))
(* a b )
)
```

lub zapis w jednej linii

```
(let* ( (a 5) (b 14) ) (* a b) )
```

```
> (let* ( ( a 5 )
( b 14 ))
( * a b )
)

70
```

TinyScheme zwraca wymnożenie dwóch zmiennych **70**.

W naszym przykładzie zmienne **a** i **b** obowiązują tylko w obrębie nawiasów okalających funkcję `let*`

Uwaga:

W zapisie składni `let*`. **najpierw** podajemy zmienne, a **potem** wyrażenie – np. mnożenia `(* a b)` - a nie najpierw mnożenia po `let*`.

To dlatego, że słowo kluczowe `let*` określa obszar, w skrypcie, w którym są zadeklarowano zmienne użytkowe; po wpisaniu **`(* a b)`** po słowie kluczowym **`(let* …)`**, otrzymamy błąd, ponieważ zadeklarowane zmienne są ważne tylko w kontekście dyrektywy `let*` (co nazywamy zmiennymi lokalnymi). `let*` wiąże zmienne od lewej do prawej. Wcześniejsze powiązania mogą być używane w nowych wiązaniach bardziej na prawo (lub w dół).

Przykłady:

```
(let*((a 1)(b (+ a 2)))(+ a b)) > 4
(let* ( (a 1) (b 2) (c 3))(list a b c 4 5)) > (1 2 3 4 5)
(let* ((a 3)(b 4))(/ a b)) > 0.75
```

`(let* ((+ *)) (+ 3 8))` wartość zwrócona `24`, ponieważ wewnątrz `let*` `+` oznacza `*`.

Operatory działań mogą być przesłaniane podobnie jak inne zmienne.

```
(let*( (x 9) )(sqrt x)) > 3
```

Jak w powyższym przykładzie zauważymy, nawet jeśli zdefiniujemy tylko jedną zmienną, zewnętrznych nawiasów dla listy zmiennych **nie opuszczamy**!

Wszystkie zmienne występujące w bloku `let*` "**muszą**" zawierać wartość początkową.
Nie ma znaczenia jaka wartość jest używana. Najprostszym sposobem rozwiązania problemu jest dodanie 0 (lub -1 dla obrazu i drawable ID) do deklaracji dla zmiennych numerycznych lub "" dla zmiennych typu string. Każda wartość w deklaracji będzie działać.
Przykład:

```
(let* ((x 10) (y 20))(+ x y)) > 30
```

```
> (let* ((x) (y 20))(+ x y))
Error: Bad syntax of binding spec in let*: ((x) (y 20))

> (let* ((x 0) (y 20))(+ x y))
20
```

`let*` jest podobna do `let`, ale `let*` ocenia deklaracje zmiennych sekwencyjnie, tak że mogą korzystać z wartości wykazanych w tych samych deklaracjach (wiązaniach). `let*` wiąże zmienne od lewej do prawej. Wcześniejsze wiązania mogą być używane w nowych wiązaniach bardziej na prawo (lub w dół).
`let*` - jest w rzeczywistości wygodnym skrótem zagnieżdżonych `let`.

```
> (let* ((a 5) (b (+ a a)) (c (+ a b)))
    (list a b c))
(5 10 15)
```

powyższe wyrażenie `let*` może być wyrażone jako zagnieżdżone `let`:

```
(5 10 15)
> (let ((a 5))
    (let ((b (+ a a)))
        (let ((c (+ a b)))
          (list a b c))))
(5 10 15)
```

```
> (let* ((x (list "Malach")])(y (cons "Janusz" x))(z (cons "Zbigniew" y)))
(list x y z))
(("Malach") ("Janusz" "Malach") ("Zbigniew" "Janusz" "Malach"))
>(let* ((x 3)
       (y (+ x 2))
       (z (+ x y 5)))
       (* x z))
39
```

Wiązania wyrażeń są ocenione razem z zewnętrznym kontekstowym otoczeniem zmiennych.
Jeśli jedna z lokalnej zmiennej ma tą samą nazwę jak zmienna w otaczającym środowisku, to *przysłania* zmienną zewnętrzną.

```
> (define a 100)
a
> (let ((a 10) (b (* 2 a))) (+ a b))
210
> (let* ((a 10) (b (* 2 a))) (+ a b))
30

>(let ((x 2) (y 3))
   (let ((x 7)
         (z (+ x y)))
     (* z x)))
35

>(let ((x 2) (y 3))
   (let* ((x 7)
          (z (+ x y)))
     (* z x)))
70
```

Bardziej szczegółowo o **`let`**

składnia : `(let (( zm war ) ...) wyr 1 wyr 2 ...)`
`let` ustawia lokalne wiązania zmiennej.
Każda zmienna *zm* jest związana z wartością odpowiedniego wyrażenia *war* .
`let` wiąże *zm*1 do *war*1 i *zm*2 do *war*2 (i tak dalej); potem wykonuje instrukcje swojego ciała
Mówimy, że zmienne są *wiązane* z wartościami przez `let`.
Ciało `let` z którym zmienne są wiązane, jest ciągiem wyrażeń *`wyr 1 wyr 2 ... .`*
Wyrażenia let mają postać kombinacji złożonej z trzech elementów.
Pierwszy z nich to słowo kluczowe `let`.
Drugi z nich, to lista par definiujących lokalnie wprowadzane stałe, ujęta w nawiasy.
Natomiast trzeci element, to wyrażenie, którego wartość stanowi wartość całego wyrażenia.
Para definiująca lokalnie wprowadzaną stałą składa się z nazwy tej stałej oraz wyrażenia określającego jej wartość, ujętych w nawiasy.
Zmienne lokalne przestają być widoczne po wyjściu z konstrukcji bloku **`let`**.

Zmienne wiązane przez `let` są widoczne tylko w ciele `let`.

```
(let ((+ *))
  (+ 2 3)) > 6
```

Możliwe jest zagnieżdżenie wyrażenia `let`.
Kiedy zagnieżdżone wyrażenie `let` wiąże tą samą zmienną, to tylko wiązanie tworzone przez wewnętrzną `let` jest widoczne w jej ciele.
Powtórzymy: jeśli jedna z lokalnej zmiennej ma tą samą nazwę jak zmienna w otaczającym środowisku, to *przysłania* zmienną zewnętrzną lub inaczej wewnętrzne wiązanie dla tej samej zmiennej, *przysłania* zewnętrzne wiązanie do tej zmiennej.

```
>(let ((x 1))
   (let ((x (+ x 1)))
     (+ x x)))
4
```

Zewnętrzne wyrażenie `let` wiąże x z 1 wewnątrz jej ciała, *którym jest drugie wyrażenie* `let`. Wewnętrzne wyrażenie `let` wiąże **x** z `(+ x 1)` wewnątrz jego ciała, którym jest wyrażenie `(+ x x)`.
Co będzie wartością `(+ x 1)`?
Skoro `(+ x 1)` pojawia się wewnątrz ciała zewnętrznego `let`, ale nie w ciele wewnętrznego `let`, wartość `x` musi być 1, a tym samym wartość `(+ x 1)` to 2.
Co natomiast będzie wartością `(+ x x)`?

To okazuje się ciałem obydwu wyrażeń `let`. Tylko wewnętrzne wiązywanie dla x jest widoczne, więc x jest 2 i wartość `(+ x x)` wynosi 4.

Jak podano wewnętrzne wiązanie dla x, *przysłania* zewnętrzne wiązanie. Wiązanie zmiennej `let` jest widoczne wszędzie w granicach treści wyrażenia `let` z wyjątkiem gdzie jest przysłaniane. Region, w którym wiązanie zmiennej jest widoczne nazywane jest *zakresem*. Zakresem pierwszego x w powyższym przykładzie jest ciało zewnętrznego wyrażenia `let` minus ciało wewnętrznego wyrażenia `let`, gdzie jest ono przysłonięte przez drugie x. Ta forma zakresu to jest *zakres leksykalny*.
Zakres każdego wiązania może być określony przez prostą analizę tekstu programu.
Przysłaniania można uniknąć, wybierając różne nazwy dla zmiennych. Powyższe wyrażenie może być przepisany tak, że zmienna związana przez wewnętrzną `let` jest `new-x`.

```
>(let ((x 1))
   (let ((new-x (+ x 1)))
     (+ new-x new-x)))
 4
```

Chociaż wybieranie różnych nazw może niekiedy zapobiegać bałaganowi, to przysłanianie może pomóc zapobiec przypadkowemu użyciu "starych" wartości.

`let` może być uważany za **lukier składniowy** - dla prostych abstrakcji lambda.
(Komentarze a także opcjonalne białe znaki zwiększające czytelność programu to również lukier składniowy – w żaden sposób semantycznie nie wpływają na przebieg kompilacji i tworzony kod wynikowy programu.)

Słowo kluczowe *set!*

Niekiedy po kilka razy musimy używać tych samych danych, ponadto dane wejściowe mogą różnić się od siebie. Po zainicjowaniu zmiennej, być może trzeba będzie zmienić jej wartość w dalszej części skryptu.
Każdej ***nazwanej*** stałej i zmiennej można *przypisać* nową wartość za pomocą operatora przypisania czyli słowa kluczowego **`set!`**.
Ma ono dwa argumenty. Pierwszym jest nazwa zmiennej, a drugim nowa przypisywana wartość.

```
(set! <variable> <expression>)
   (set! zmienna wyrażenie)
```

`set!` zmienia wartość zmiennej na nową, najczęściej policzoną w jakimś wyrażeniu.
Zmienna może być zadeklarowana (np. przez `define` lub `let*` albo być parametrem funkcji tworzonej przez `lambda`).
Przykład:

```
> (set!  x 5) ; zachowaj liczbę jako wartość symbolu x
5
> x         ; pobierz wartość symbolu x
5
> (let ((x 6)) x) ; tymczasowe związanie symbolu a do wartości 6
6
> x         ; zmienna lokalna a przestaje być widoczna po zakończeniu bloku let,
pobierając x jego wartość wraca do początkowej 5
5
> (+ x 6)   ;użyjemy wartości symbolu jako argumentu procedury
11

> (set! x 7)
7
> (let* ((x 1)
(y (+ x 1)))
  y
 )
2

> (set! c 4)
4
> (let ((c 5)) c)
5
> c
4

(let*( (a 42) (b 21) (x 0) )(set! x (/ a b))) zwraca 2
(set! xyz #\c) zwraca #\c

((define x 2) (+ x 1) (set! x 4)))
(define x 2)zdefiniowano że x = 2
```

```
(+ x 1)> 3
po czym (set! x 4) > x = 4
teraz (+ x 1)      > 5
```

Uwaga:
Zmienną może być tekst (oczywiście jako "Tekst"), aby być precyzyjnym zmiennymi mogą być inne "*rzeczy*" przypisane wektory, znaki, listy, itp…

Można stosować procedury `set-car!` i `set-cdr!` zmieniające odpowiednio pierwszą i drugą współrzędną w danej parze.
procedura: `(set-car! pair obi)`
`set-car!` changes the car of *`pair`* to *`obi`*.
procedura: (set-cdr! *pair obi*)
`set-cdr! changes the cdr of pair to obi.`
Przykład
```
(let* ((x '(a b c)))(set-car! x 1)x) > (1 b c)
(define p '(1 2 3))(set-car! (cdr p) 4) > p(1 4 3)
```

Stosowane określenia:
Zakres leksykalny "zakres widzialności" zmiennej nazywany jest jej *zasięgiem* (ang. lexical scope)
Określenie własności języków programowania, zgodnie z którą zakres definiowanej nazwy jest ograniczony do sąsiadującego z tą definicją tekstu programu (modułu, procedury, funkcji).
Np. zakres zmiennej definiowanej wewnątrz pewnej procedury jest ograniczony do tekstu tej procedury; zmienna ta jest niewidoczna na zewnątrz.
Jak sama nazwa wskazuje, ***zmienna lokalna*** jest "prywatną własnością" funkcji lub procedury, wewnątrz której została zdefiniowana. *Zmienna lokalna nie jest widoczna na zewnątrz funkcji.* We wnętrzu funkcji zmienna jest znana od momentu jej deklaracji do klamry zamykającej blok, w którym zmienna jest zadeklarowana!.
Czas życia zmiennych lokalnych ograniczony jest do czasu wykonywania posiadającej je funkcji, tak więc "lokalność" ma charakter nie tylko przestrzenny, ale i czasowy - zmienne lokalne są tworzone w momencie rozpoczęcia wykonywania funkcji i znikają po jej zakończeniu.
Procedura (funkcja) przypomina strukturą mały program, toteż deklaracje zmiennych lokalnych umieszcza się zwykle na jej początku, po nagłówku.
Zmienne globalne: ich zakres jest cały tekst programu, który jest z pewnością rozległy. Deklarujemy je przed zdefiniowaniem wszystkich funkcji. Są one dostępne dla wszystkich.
Wiązanie to zawężone leksykalnie przypisanie. Zachodzi na zmiennych w liście parametrów funkcji za każdym wywołaniem funkcji: formalne parametry są wiązane do rzeczywistych parametrów na czas wywołania funkcji. Zmienne można wiązać w dowolnym miejscu programu, używając formy **let**, lub let*.
Przesłanianie nazw (ang. *name shadowing*), pojawiło się wraz ze wprowadzeniem zasięgu zmiennych. Konflikt nazw zmiennych o różnym zasięgu jest rozstrzygany zawsze na korzyść zmiennej o **węższym** zakresie, czyli zazwyczaj oznacza to zmienną lokalną.
Blokiem w programie nazywamy jego pewną logiczną część, która jest traktowana jako pojedynczy, spójny fragment. Znaczy to w przybliżeniu tyle, że pojedynczy blok można traktować w pewnym sensie jak jedną instrukcję.
Wyrażenia algebraiczne - powstają przez połączenie symboli literowych oraz liczb znakami działań i nawiasów. Wyrażenia w których występują liczby i litery połączone znakami działań i nawiasami nazywamy wyrażeniami algebraicznymi. Wyrażenia algebraiczne przyjmują nazwę od ostatniego wykonywanego działania zgodnie z kolejnością wykonywania działań.
Rekurencja, zwana także ***rekursją*** (ang. *recursion*, z łac. *recurrere*, przybiec z powrotem) to w programowaniu odwoływanie się np. funkcji lub definicji do samej siebie.

Przykład wyjaśniający różnice w spotykanych określeniach:
Uruchamiamy GIMP-a i z menu **Filtry => Script-Fu => Konsola** (otwarta do końca przykładu - "lokalność" nie tylko przestrzenna, ale i czasowa)
```
> (define x 9) ; związanie wartości 9 z symbolem x
x
> x ; pobieramy wartość symbolu
9 wartość zwrócona
```
W ten sposób wprowadzono do przestrzeni interpretera wiązanie symbolu x z wartością 9.
Teraz stworzymy własną funkcję anonimową, która dodaje *2* do jej argumentów:
```
(lambda (x) (+ x 2)))
```
Po czym, możemy użyć innej zmiennej do przechowywania wartości tej funkcji:
```
> (define add2 (lambda (x) (+ x 2)))
add2 ; przywiązanie wartość symbolu x do add2
> (add2 3)
5
```

```
> (add2 x)
11
> x
9
```
Widzimy, że istnieje **x** globalne, oraz mamy także **x** lokalne, to ostatnie wprowadzone procedurą **add2**.
Globalny **x** (w danej sesji otwartego interpretera TinyScheme) będzie ciągle 9.
Lokalny **x** został związany z **3** w ramach pierwszego wywołania **add2** (w tej procedurze x lokalne przysłoniło x globalne) natomiast w drugim wywołaniu **add2** zostało związane z wartością globalnego **x**, tj. 9.
set! modyfikuje leksykalne wiązania zmiennych (patrz poniżej), a więc gdy:
```
> (set! x 20) ;zachowaj liczbę 20 jako wartość symbolu x
20
```
zmodyfikujemy globalne wiązanie x z 9 na 20, ponieważ jest to teraz wiązanie **x,** które jest widoczne przy `set!`.
Jeśli `set!` byłoby w środku ciała **add2**, byłaby to modyfikacja lokalnego **x**:
```
>(define add2
   (lambda (x)
     (set! x (+ x 2))
     x))
add2
```
tutaj `set!` dodaje 2 do zmiennej lokalnej **x**, i zwraca tę wartość jako nowe **add2**.
(Chodziło tu o efekt, ta procedura jest nie do odróżnienia od poprzedniej **add2**.)
Możemy nazwać **add2** globalnym x, jak poprzednio:
```
> (add2 x)
22
```
(Pamiętamy, że globalnym **x** jest teraz **20**, a nie 9)
`set!` wewnątrz **add2** dotyczy tylko zmiennej lokalnej używanej przez **add2**. Mimo że zmienna lokalna **x** jest jeszcze związana z globalnym **x**, to ostatnie jest bez wpływu na `set!` i teraz dla lokalnej x.
```
> x
20
```
Należy pamiętać, że w całej tej dyskusji, używaliśmy tego samego identyfikatora zmiennej lokalnej i zmiennej globalnej. W każdym tekście identyfikator o nazwie **x** odnosi się do najbliższej leksykalnie nazwy zmiennej **x**.
Jest to przysłanianie jakiegoś zewnętrznego lub globalnych x-ów ". np. w **add2**, parametr **x** w cieniu globalnego **x**. Procedura ciało ma dostęp i może modyfikować zmienne w jej zakresie i okolicach, pod warunkiem że parametry procedury ich nie przysłaniają.

Zmienne lokalne mogą być wprowadzone bez wyraźnego tworzenie procedury.
Najpierw użyjemy możliwego również do stosowania wyrażenia ***let***.
Mają one postać kombinacji złożonej z trzech elementów. Pierwszy z nich to słowo kluczowe `let`.
Drugi z nich, to lista par definiujących lokalnie wprowadzane stałe, ujęta w nawiasy.
Natomiast trzeci element, to wyrażenie, którego wartość stanowi wartość całego wyrażenia.
Para definiująca lokalnie wprowadzaną stałą składa się z nazwy tej stałej oraz wyrażenia określającego jej wartość, ujętych w nawiasy.
Forma `let` wprowadza listę zmiennych lokalnych na używane swoim ciele (body):
```
>(let ((x 1)
       (y 2)
       (z 3))
   (list x y z))
(1 2 3)
```
Podobnie jak w przypadku `lambda`, w ciele `let`, lokalne **x** (związane z *1*) przysłania globalny **x** (który jest związany z *20*).
```
> (let ((x 1)
       (y x))
   (+ x y))
21
```
To dlatego, że lokalny **x** jest związany z *1, a* **y** jest związany z *globalnym* **x**, czyli teraz *20*.
Jeżeli nadając wartość zmiennej lokalnej, chcemy wykorzystać tą wartość zdefiniowanej wcześniej zmiennej w obrębie tej samej konstrukcji **`let`**, wówczas należy użyć opisanej już bliźniaczej konstrukcji **`let*`**, zastosujemy więc formę `let*` i otrzymamy:
```
> (let* ((x 1)
        (y x))
   (+ x y))
2
```
Tutaj **x** uruchamiane w **y** odnosi się do **x** tuż powyżej.
Poniżej przykład z równoważnych zagnieżdżonych `let` (let* jest wygodnym skrótem):
```
> (let ((x 1))
   (let ((y x))
```

```
    (+ x y)))
2

> (let ((x 2))
    (let* ((x 3)
           (y (+ x 4)))
      (+ x y)))
10
> (let ((x 2))
    (let ((x 3))
      (let ((y (+ x 4)))
        (+ x y))))
10
```

Instrukcje sterujące (warunkowe)

słowo kluczowe **if**

TinyScheme posiada (co prawda mocno okrojone, ale za to bardzo funkcjonalne) instrukcje sterujące.
Instrukcja warunkowa ***if*** używana jest wszędzie tam, gdzie występuje konieczność podjęcia decyzji o wyborze dalszej drogi algorytmu.
Składnia `if` przedstawia się następująco:

```
(if (Predykat)
(wyrażenie_jesli_predykat_spelniony_T)
(alternatywne-wyrażenie_jesli_predykat_nie_spelniony_F))
```

Wyrażenie warunkowe **if** (*jeżeli*) czyli wybór „jeden z dwóch”, ma postać kombinacji czterech elementów, z których pierwszy to *słowo kluczowe* **if**.
Drugi element kombinacji to Predykat (warunek - mówiąc po ludzku, jest to zwykłe porównanie).
Trzeci element to wyrażenie (lub instrukcja), które jest obliczane gdy predykat jest prawdziwy,
a czwarty to wyrażenie, które jest obliczane gdy predykat nie jest spełniony - fałszywy.
Stosowane typy predykatów omówiono szczegółowo powyżej.

Tą formę specjalną możemy przetestować w ciągu następujących interakcji

```
> (if (> 1 2) (+ 3 4) (- 2 5))
-3
> (if (even? 4) (+ 4 1) (- 4 1))
5
```

W pierwszym przypadku, TinyScheme zwrócił -3, ponieważ predykat zakładał że jeden jest większe niż dwa, co oczywiście nie jest prawdą i dlatego obliczył wyrażenie `(- 2 5)`.
W drugim przypadku zwrócił wartość 5, bo cztery jest liczbą parzystą.
Korzystanie z tej szczególnej formy programu można również zademonstrować na procedurze obliczania wartości bezwzględnej liczby i procedurze ustalania, większej z dwóch liczb.

```
> (define (abs x)
(if (> x 0) x
(- x)))
abs
> (abs -4)
4
> (define (maximum x y)
(if (> x y) x y))
maximum
> (maximum 3 -8)
3
> (define (mnoż x y)
(if (= x 0) 0 (+ y (mnoż (- x 1) y))))
mnoż
> (mnoż 2 3)
6

(define x 42); definiuję że x ma wartość 42
; teraz użyjmy tylko jednego wyrażenia (gałęzi) słowa kluczowego if
; które będzie wykonane w przypadku , gdy wynik predykatu jest #t
> (if (number? x) (display "parameter jest typu cyfra"))
parametr jest typu cyfra#t

; teraz użyjmy obydwa wyrażenia (gałęzie) słowa kluczowego if
> (if (positive? x) (display "wartość dodatnia") (display "wartość ujemna"))
wartość dodatnia#t
```

```
; bardziej czytelny zapis poprzedniego
> (if (positive? x)              ; predykat (warunek)
    (display "wartość dodatnia") ; wyrażenie (gałąź) "then"
    (display "wartość ujemna")   ; wyrażenie (gałąź) "else" (ekwiwalent true)
)
wartość dodatnia#t
```

Dalsze przykłady:

```
>(if (> 3 2) 'tak 'nie)        >  tak
>(if (> 2 3) 'tak 'nie)        >  nie
>(if (> 3 2)(- 3 2)(+ 3 2))    >  1
>(define abs(lambda (n)(if (< n 0)(- 0 n)n)))
>(abs 77)
77
>(abs -77)
77
> ((if #f + *) 3 4)
12
>((if #t + *) 3 4)
7
```

Możemy również mieć do czynienia z łańcuchem wywołań instrukcji **`if`** (instrukcje zagnieżdżone).

```
; Sprawdzanie, czy wartość należy do listy
> (define (in-list? itm lst)
(let* (
    (found 0))
  (for-each (lambda (x)
    (if ((if (string? x) string=? =) x itm) (set! found (+ found 1))))
    lst)
  (if (> found 0) #t #f)))
in-list?
> (in-list? 3 '(1 2 3 4 5 6))
#t
> (in-list? 8 '(1 2 3 4 5 6))
#f
```

Funkcja `fact` wylicza silnię liczby naturalnej *n* (factorials):

```
5! = 5 x 4 x 3 x 2 x 1 = 120
1! = 1
 > (define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
fact ;zwrócone
(fact 5) jest obliczane zgodnie z:
>(fact 5)
⇒ 5 * (fact 4)
⇒ 5 * 4 * (fact 3)
⇒ 5 * 4 * 3 * (fact 2)
⇒ 5 * 4 * 3 * 2 * (fact 1)
⇒ 5 * 4 * 3 * 2 * 1
⇒ 5 * 4 * 3 * 2
⇒ 5 * 4 * 6
⇒ 5 * 24
⇒ 120
```

Słowo kluczowe ***cond***

Wymienimy jeszcze jedną użyteczną ogólniejszą instrukcję warunkową TinySCheme, która pozwala podać kilka warunków i akcji.
Jest nią *słowo kluczowe* ***cond*** "*jeden z wielu*" (skrót od słowa „condition“).
Składnia `cond` przedstawia się następująco:

```
(cond (Predykat_warunek1 wyrażenie_co-zrobić1)
      (Predykat_warunek2 wyrażenie_co-zrobić2)
...)
```

Kolejne warunki (predykaty) sprawdzane są po kolei, aż do napotkania prawdziwego, wtedy wykonywane jest odpowiednie `wyrażenie_co-zrobić`.

Ostatnim z warunków *może być Słowo kluczowe* `else` (jeżeli inne badanie się udało), który to warunek jest zawsze prawdziwy *true*.
Funkcja warunkowa ***cond*** upraszcza tworzenie zagnieżdżonych funkcji **`if`**.
Tak więc formę:

```
 (if (char<? c #\c) -1
    (if (char=? c #\c) 0
        1))
```

może być zapisana jako:

```
(cond ((char<? c #\c) -1)
      ((char=? c #\c) 0)
      (else 1))

> (cond ((< x 0) -1) ((= x 0) 0) (else 1))
-1

(define (sprawdz x)
(cond ((< x 0) "liczba mniejsza od 0")
((and (>= x 0) (<= x 10)) "liczba z przedziału od 0 do 10")
((> x 10) "liczba wieksza od 10")))
>(sprawdz -1)
"liczba mniejsza od 0"
>(sprawdz 5)
"liczba z przedziału od 0 do 10"
>(sprawdz 12)
"liczba wieksza od 10"

> ( cond
((< 3 2) 1)
((< 4 3) 2)
(else 3))
3
```

W poniższym przykładzie zwrócony zostanie jeden z trzech łańcuchów w zależności od tego, jaka wartość liczbowa jest przekazana cond:

```
> (define x -42)
x
> (cond
    ((negative? x) "wartość ujemna")
    ((positive? x) "wartość dodatnia")
    (else "zero")
)
"wartość ujemna"

> (define x 0)
x
> (cond
    ((negative? x) "wartość ujemna")
    ((positive? x) "wartość dodatnia")
    (else "zero")
)
"zero"

> (define a 3)
a
> (define b (+ a 1))
b
> (cond ((= a 4) 6)
          ((= b 4) (+ 6 7 a))
          (else    25))
16
```

Słowo kluczowe: ***`else`***.
Słowo kluczowe `else` oznacza "w przeciwnym wypadku", czyli jeśli warunek nie zostanie spełniony, wykonaj inny kod.
Ogólny zapis instrukcji warunkowej, wykorzystujący *słowo kluczowe* `else` będzie więc wyglądał tak:
`if(...)...else...`
Jedyną niewiadomą w tym zapisie są ostatnie trzy kropki po `else` (nie podajemy żadnego wyrażenia). Oznaczają one instrukcję/blok instrukcji, która zostanie wykonana tylko w przypadku, gdy wszystkie

poprzednie wyrażenia **if nie** będą prawdziwe. Użycie słowa kluczowego `else` jest opcjonalne. Zamiast jednak konkretnej operacji możemy wywołać kolejną instrukcję **if**, aby sprawdzić następne wyrażenie. Słowo kluczowe **if** musi być zawsze pierwszym wyrażeniem w ciągu warunkowym, a *else* ostatnim. Dzięki temu interpreter jest w stanie odróżnić, gdzie zaczyna się i kończy dany warunek.

Pętla

Słowo kluczowe ***while***

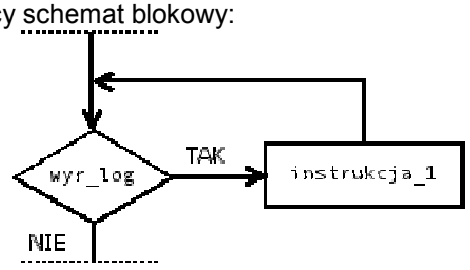
Pętla to po prostu instrukcja wykonania czegoś wiele razy. Oczywiście, można to zrobić po prostu wprowadzić instrukcje jedna po drugiej (ręcznie). Jest to jednak niepraktyczne i nieefektywne.
Najpierw tworzymy zmienną, która będzie użyta do liczenia **iteracji** (powtórzeń) pętli. Potem słowo kluczowe *while* do tworzenia pętli. Potem podajemy warunek do sprawdzenia.
Pętla `while` jest najprostszym typem pętli. Podobnie jak instrukcja `if`, zależy ona od warunku. Różnica pomiędzy pętlą `while` a instrukcją `if` polega na tym, że instrukcja `if` wykonuje następujący po niej blok kodu jednokrotnie, jeżeli jej warunek ma wartość `true`. Pętla `while` będzie powtarzać wykonywanie bloku tak długo, jak jej warunek będzie miał wartość `true`. Wniosek nasuwa się jeden - aby pętla kiedyś się skończyła, w jej ciele należy zadbać o to, żeby warunek w końcu prowadził do fałszu (czyli zakończenia pętli)!. `while` jest jedyną pętlą warunkową obecną w script-fu.
Słowo kluczowe ***while*** "*tak długo, jak*" lub "*dopóki*" służy do tworzenia pętli o tej samej nazwie.
Oto jej składnia:

```
( while ( warunek ;test logiczny)
( dzialanie1 ) ; kod wykonujący się DOPÓKI warunek logiczny jest spełniony
( dzialanie2 )
)
```

W trakcie wykonywania programu, jeżeli napotkana jest instrukcja **while** to wtedy sprawdzany jest *warunek* (`logiczny` boolean? – jest wartością 1 true lub 0 false) umieszczony w nawiasach.
W pętli **while(...)** warunek jest sprawdzany przez interpreter **zanim** wykona się *blok instrukcji*. Jeśli wynik jest różny od zera (*true*), to zostaje wykonana instrukcja i interpreter rozpoczyna nowy obieg od ponownego obliczenia wartości *warunku*. *Instrukcje* są wykonywane dopóki *warunek* jest prawdziwy (zwraca wartość różną od zera). Jeśli *warunek* ma wartość równą zero, instrukcja w pętli nie zostaje wykonana i pętla kończy działanie.
Ponieważ *warunek* jest sprawdzany na początku każdego obiegu pętli przed wykonaniem instrukcji zawartej w pętli, to jeśli jest on już od początku fałszywy, pętla nie wykona ani jednego obiegu.
Jeżeli treścią pętli **while** jest nie jedna, a więcej instrukcji, wtedy cały blok kodu należącego do pętli **while** umieszczamy pomiędzy parą nawiasów.
Ze względu na taką ogólną konstrukcję **while** przydaje się tam, gdzie musimy coś powtarzać do czasu osiągnięcia pewnego stanu.
A więc pamiętamy, że `while` oznacza mniej więcej tyle, co słowo "dopóki". Zatem, dopóki jakiś warunek jest spełniony, wykonywać się będą jakieś polecenia.
Przykład:
Wpisujemy do konsoli:

```
>(define x 0)           ; zmienna (globalna)- potrzebna do liczenia iteracji
x
>(while (< x 5)                 ; warunek zakończenia realizacji pętli
        (print x)               ; drukuj wynik - ciało pętli
        (set! x (+ x 1))        ;zwiększenie wartości zmiennej (globalnej)
) ; użycie tutaj nowego define prowadziło by do związania nowej zmiennej
0
1
2
3
4
();Przypominam - to jest pusta lista, może być zdefiniowane jako '() lub ().
```

Najpierw tworzymy zmienną, która będzie użyta do liczenia **iteracji** (powtórzeń) pętli. Linia druga to definicja pętli. Podajemy tu warunek do sprawdzenia. Obowiązują dokładnie te same zasady, które omówiłem przy okazji instrukcji if.
Pętli `while` odpowiada następujący schemat blokowy:



```
> (define b 7)
> b
7
> (while (> b 0) (write b) (set! b (- b 1)))
7654321(); Pusta lista
```
Tutaj warunek to (> b 0) - tak długo, jak b będzie większe od zera wynik działania zostanie wykonany. (set! b (- b 1)) odejmuje jeden od wartości b.

```
>(let ((x 1))
   (while (< x 10)
     (print x)
     (set! x (+ x 1))))
1
2
3
4
5
6
7
8
9
()      => ; Pusta lista
```
lub

`print` – jest wymieniona przez **`write`** (mająca za sobą **`newline`** zastosowana w dalszym przykładzie)

```
(let* ((i 1))
(while (< i 10)
(write i)
(display "\t") ; horizontal tab
(set! i (+ i 1))))
1       2       3       4       5       6       7       8       9       ()
```
Polecenie `let*` przypisuje wartość do zmiennej, której nazwą jest **i**, a wyrażeniem prosta wartość, liczba 1.
Słowo kluczowe `while` zaczyna pętlę.
Wyrażenie `write` jest wykonywane tak długo jak warunek jest prawdziwy. Warunkiem w naszym przykładzie jest wyrażenie (< i 10). Jest ono prawdziwe dopóki zmienna **i** jest mniejsza od dziesięciu.
Polecenie `write` wypisuje na ekranie argumenty. W tym wypadku wartość zmiennej **i**. Ponieważ **i** to 1, napisze:1 Później jest inne polecenie set!. Wartością jest wyrażenie (+ **i** 1). Dodaje ono jeden do zmiennej i przypisuje nową wartość do tej samej zmiennej.

Obliczenie tabeli podzielenia dwóch liczb całkowitych, przy zastosowaniu makra `while` możemy napisać następująco:

```
> (define x 1)          ;zmienne dla obydwu pętli należy wyraźnie określić
(define y 1)
xy
>(while (<= y 5)        ; warunek zakończenia pętli zewnętrznej
    (set! x 1)
    (while (<= x 5)     ; warunek zakończenia pętli wewnętrznej
        (write (/ x y)) ; wypisz wynik
        (display "\t")  ; horizontal tab
        (set! x (+ x 1))
    )
    (set! y (+ y 1))
(newline)               ; lub (display "\n")
)

1     2     3     4     5
0.5   1     1.5   2     2.5
0.3333333333      0.6666666667      1     1.333333333 1.666666667
0.25  0.5   0.75  1     1.25
0.2   0.4   0.6   0.8   1
()
```

Przykład do wykorzystania:

```
; Wycięcie nazwy pliku z pełnej ścieżki
> (define (hl-filename path)
(let* (
    (f-len (string-length path))
    (s-len 1))
```

```
  (while (and
      (<= s-len f-len)
      (not (re-match "/" (substring path (- f-len s-len)))))
    (set! s-len (+ s-len 1)))
  (substring path (+ (- f-len s-len) 1))))
hl-filename ;wartość zwrócona
> (hl-filename "C:/Program Files/GIMP-2.6. /GIMP-
2.0/share/gimp/2.0/scripts/script-fu.init")
"script-fu.init"
```

Specjalna forma - `begin`

Czasami konieczne jest do wykonywania czynności jedna po drugiej w narzuconym odgórnie porządku. Mówi się, że działania te są wykonywane sekwencyjnie - w *kolejności séquence*. Aby połączyć kilka poleceń w jednym bloku.
W TinyScheme jest możliwe użycie specjalnej formy `begin` (rozpocząć), której argumenty wyrażenia należy ocenić po kolei, od pierwszego z lewej do ostatniego z prawej. Wynik oceny ostatniego wyrażenia (główny efekt) jest zwracany.
Kolejność składni jest następująca:

(begin *expression_1... expression_n*)

Procedura ma działania uboczne!.
Dla niektórych wyrażeń, ważny jest nie zwrócony wynik, ale podejmowane działania.

W poniższym przykładzie,
możemy zrobić, cztery rzeczy: wyświetlić tekst, cyfry, nową linię i rekurencyjnie wywołać procedurę.
Kod będzie wyglądać tak:

```
> (define (suma-od-do a b)
 (if (> a b)
(begin
 (display "Gotowe")
 (display #\ ); preferowane jest jednak #\space
0)
 (begin
 (display "Dodaję liczbę");Procedura akceptuje jeden parametr i wpisze go na
ekranie (dodano to w celach diagnostycznych).
 (display #\ )
 (display a)
 (display "\n"); lub (newline)
 (+ a (suma-od-do (+ a 1) b)))))
suma-od-do ; zwrócone
> (suma-od-do 2 10)
Dodaję liczbę 2
Dodaję liczbę 3
Dodaję liczbę 4
Dodaję liczbę 5
Dodaję liczbę 6
Dodaję liczbę 7
Dodaję liczbę 8
Dodaję liczbę 9
Dodaję liczbę 10
Gotowe 54
```

Jak widać wartość zwrócona (czyli główny efekt) jest wynikiem oceny ostatniego wyrażenia.

```
; #\space wprowadzono żeby ładniej wyglądały wpisane wyniki.
; gotowe i zwrócone 0, jeśli warunek nie jest spełniony.
> (suma-od-do 10 2)
Gotowe 0

>(define x 0)
(begin (set! x 5)(+ x 1))
6

>(let countdown ((i 10))
(if (= i 0) 'liftoff
(begin
(display i)
(newline)
(countdown (- i 1)))))
```

```
10
9
8
7
6
5
4
3
2
1
Liftoff

> (begin (display "One") (display "\n") (display "Two")(display "\n"))
One
Two
#t

> (begin
  (display 'a)
  (display 'b)
  (display 'c))
abc#t

> (define x 3)
> (begin
(set! x (+ x 1))
(+ x x) )
8 ; zwrócone

> (begin
       (display "Odpowiedź:")
       (display #\ )
       (display (+ 2 4 (* 4 5)))
     )
Odpowiedź: 26#t
```

Ponownie inaczej zapisana definicja silni:

```
5! = 5 x 4 x 3 x 2 x 1 = 120
> (define (fact n)
  (if (= n 1)
      1
(begin
 (display "Wprowadzenie silni dla n = ")
(display n)
(newline)
      (* n (fact (- n 1))))))
fact
> (fact 5)
Wprowadzenie silni dla n = 5
Wprowadzenie silni dla n = 4
Wprowadzenie silni dla n = 3
Wprowadzenie silni dla n = 2
120
> (define próba
  (lambda ()
    (begin
      (display "GIMP to jest to")
        (display #\ )
      (display (sqrt 25))
        (display "!")
     (newline))))
próba
> (próba)
GIMP to jest to 5!
#t
```

Praktycznie każdy język programowania opiera się na języku angielskim. Mam na myśli chociażby nazwy funkcji, które stworzono ze skróconych nazw określonych operacji.
Głównym powodem stosowania się do tej zasady jest uniwersalność, dlatego każdy piszący skrypty powinien pisać je po angielsku.

Dzięki powyższym krótkim wprowadzeniom, możemy poćwiczyć tworzenie własnych funkcji, obliczających podstawowe zadania matematyczne, lub operacje na zmiennych.

Przestarzałe funkcje

Wymienione poniżej funkcje i stałe, które były określone w interpreterze systemu SIOD, nie są częścią interpretera TinyScheme.
W GIMP 2.4 załadowano plik (script-fu-compat.scm), który zawiera pewną liczbę funkcji i stałych zgodnych z SIOD. Plik ten jest automatycznie ładowany kiedy plug-in Script-Fu startuje.

W poniższej liście są funkcje i stałe dostępne w pliku zgodności z SIOD i równoważne TinyScheme (jeśli istnieje). Wymienione **nie** są zalecane do stosowania w nowych skryptach.

- aset – wymieniona przez **vector-set!**
- aref - wymieniona przez **vector-ref**
- butlast
- cons-array - wymieniona przez **make-vector** (except for string arrays)
- fmod
- fopen - wymieniona przez **open-input-file**
- fread
- last
- mapcar - wymieniona przez **map**
- nil - wymieniona przez **'()**
- nreverse - wymieniona przez **reverse**
- (nth k list) - wymieniona przez **(list-ref list k)**
- *pi* - jest zdefiniowane jako (* 4 (atan 1.0)) w procedurach kompatybilnych SIOD
- pow - wymieniona przez **expt**
- prin1 - wymieniona przez **write**
- print - wymieniona przez **write** mająca za sobą **newline**
- prog1
- strcat - wymieniona przez **string-append**
- strcmp
- string-lessp - wymieniona przez **string<?**
- symbol-bound? - wymieniona przez **defined?** (To wydaje się być poza R5RS ale interpreter pozwala to stosować)
- the-environment - wymieniona przez **current-environment**
- trunc - wymieniona przez **truncate**
- verbose

I/O

```
As per R5RS, plus String Ports (see below).
current-input-port, current-output-port,
close-input-port, close-output-port, input-port?, output-port?,
open-input-file, open-output-file.
read, write, display, newline, write-char, read-char, peek-char.
char-ready? returns #t only for string ports, because there is no
portable way in stdio to determine if a character is available.
Also open-input-output-file, set-input-port, set-output-port (not R5RS)
Library: call-with-input-file, call-with-output-file,
with-input-from-file, with-output-from-file and
with-input-output-from-to-files, close-port and input-output-port?
(not R5RS).
String Ports: open-input-string, open-output-string,
open-input-output-string. Strings can be used with I/O routines.
```

Control features

```
Apart from procedure?, also macro? and closure?
map, for-each, force, delay, call-with-current-continuation (or call/cc),
eval, apply. 'Forcing' a value that is not a promise produces the value.
There is no call-with-values, values, nor dynamic-wind. Dynamic-wind in
```

```
the presence of continuations would require support from the abstract
machine itself.
```
Tych rzeczy nie będę opisywał!.

Określenia

INSTRUKCJE
umożliwiają wydawanie poleceń w danym języku programowania.
Rozróżniamy następujące instrukcje: procedury, funkcje, przypisania, skoku, sekwencji, warunkowe, iteracji, pętli.

Procedury i funkcje
to podprogramy, stanowiące pewną całość, posiadające jednoznaczną nazwę i ustalony sposób wymiany informacji z pozostałymi częściami programu.
Są stosowane do wykonania czynności wielokrotnie powtarzanych przez dany program.

Różnice między funkcją a procedurą:
- sposób przekazywania wartości
- odmienne sposoby wywołania

➢ zadaniem procedury (podprogramu) jest wykonanie pewnej sekwencji czynności (poleceń), polegających zwykle na obliczaniu jednej lub wielu wartości
Liczba, kolejność i typy parametrów muszą być zgodne z definicją procedury!
➢ zadaniem funkcji jest przekazywać pewne parametry jej wykonania, jak również może zwracać wynik jej działania.
Liczba, kolejność i typy parametrów oraz typ wartości zwracanej muszą być zgodne z definicją funkcji!

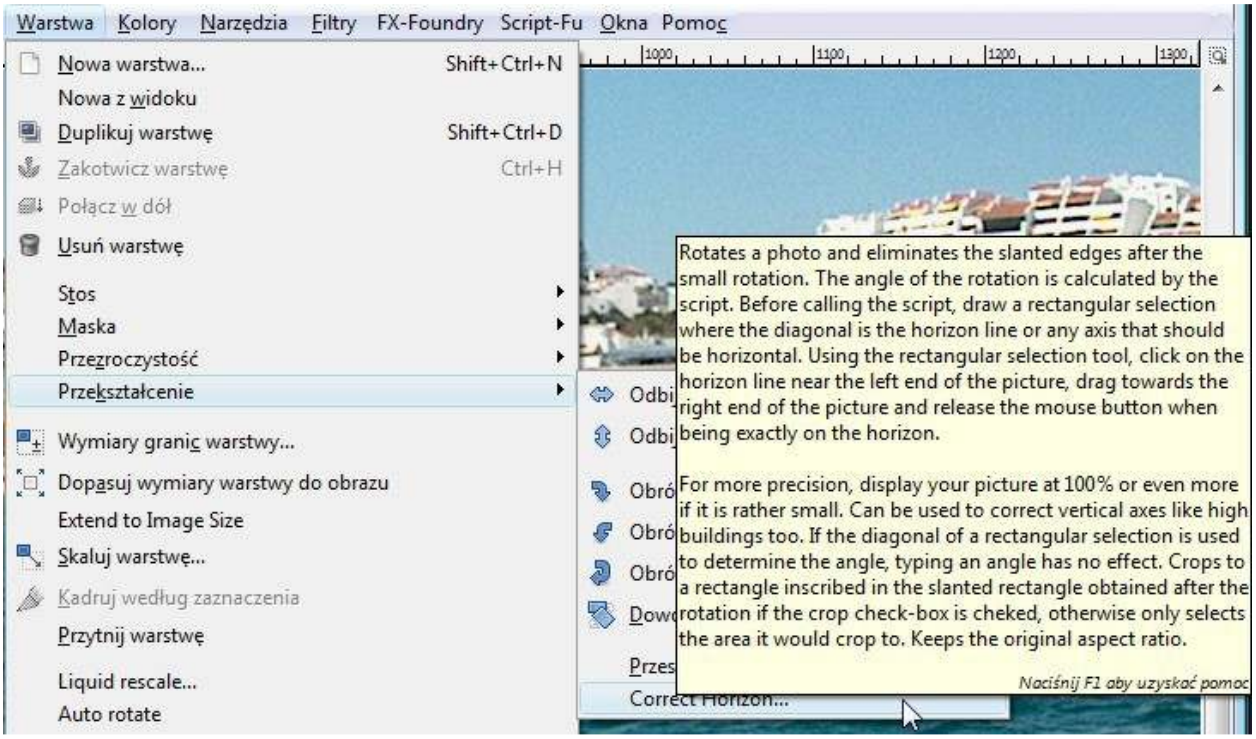
Parametry procedury wymienione w nawiasie na początku jej ciała nazywamy parametrami (argumentami) formalnymi,
wartości dla parametrów formalnych podane w wywołaniu procedury nazywamy parametrami aktualnymi (argumentami wywołania) procedura może być także bezparametrowa.
Nazwa procedury musi być identyfikatorem *(w przypadku funkcji wyjątkiem są nazwy operatorów)* procedura nie zwraca wartości *funkcja zwraca dokładnie jedną wartość.*

Procedura może mieć swoje własne zmienne (zmienne lokalne), deklarowane w jej części deklaracyjnej zmienna lokalna jest widziana tylko wewnątrz procedury z tego powodu zmienne lokalne różnych procedur mogą mieć takie same nazwy **zmienne lokalne procedury to nie to samo co jej parametry.**

Komentarze

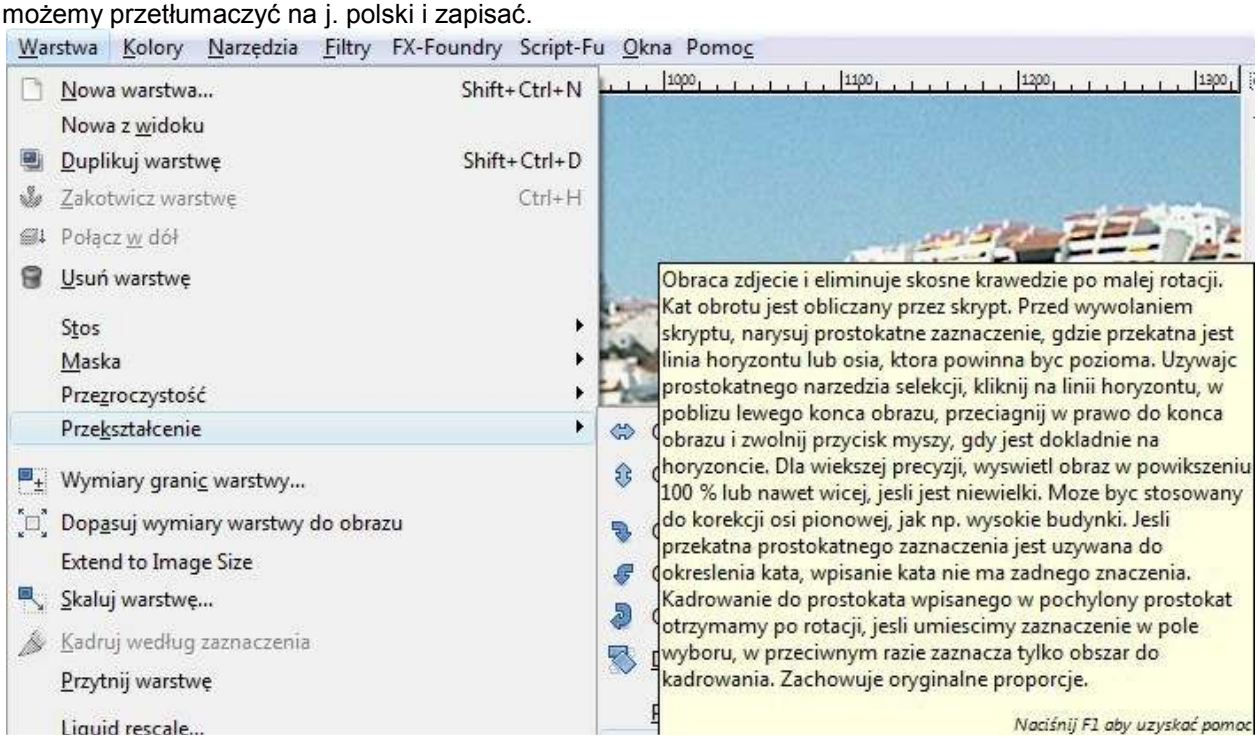
Komentarze zaczynają się od znaku średnika (**;**)
Pierwszą rzeczą, jaką zauważymy w skryptach na dole to wiele, wiele linii jako komentarze.
Komentarze są czymś, co pomoże Nam zrozumieć skrypt, ale nie mają wpływu na faktyczną realizację.
Innymi słowy, komentarze nie są w ogóle oprogramowaniem, to jest czysty tekst.
Komentarze są bardzo pomocne. Szukamy średników, aby następnie przeczytać wszystkie komentarze, które występują po prawej za średnikiem.
Uwaga: Jak wspomniano powyżej, jedną z głównych przyczyn przejścia na TinyScheme jest wspieranie międzynarodowych języków i czcionek (kod UTF-8). A więc możemy otworzyć dowolny Script-Fu w **Notatnik**-u (NotePad) http://www.7tutorials.com/beginners-guide-notepad
Najlepiej ustawić, aby pliki zawarte w folderze **scripts** były zawsze otwierane w **Notatniku**
Klikamy PPM na dowolnym pliku w folderze i wybieramy:
Otwórz za pomocą => Wybierz program domyślny… => Notatnik od tej chwili dwuklik na dowolnym pliku otworzy go w notatniku.
W notatniku możemy ustalić rodzaj kodowania przy zapisie:

Bo jeśli dokument zawiera tekst w kilku językach, to należy użyć kodu UTF-8.
Opisy skryptów istotne dla Nas przy podglądzie (po wskazaniu wskaźnikiem myszki Nazwy skryptu):



Podgląd opisu skryptu

możemy przetłumaczyć na j. polski i zapisać.



Dla polskich znaków obowiązuje norma kodowania zaakceptowana przez ISO – ISO-8859-2 lub Unicode UTF-8. Jeżeli nasz edytor tego nie potrafi, powinniśmy zmienić edytor lub... po każdej edycji dokumentu przekonwertować go do standardu kodowania ISO-8859-2 albo UTF-8.
Nie pisać tekstów w **WordPad** nie posiada kodowania polskich znaków.

Ten komentarz znajdziemy w Script-Fu

```
(script-fu-register "script-fu-correct-horizon"
                    "<Image>/Layer/Transform/Correct Horizon..."
                    "Rotates a photo and eliminates the slanted edges after the
small rotation. The angle of the rotation is calculated by the script. Before
calling the script, draw a rectangular selection where the diagonal is the
horizon line or any axis that should be horizontal. Using the rectangular
selection tool, click on the horizon line near the left end of the picture, drag
towards the right end of the picture and release the mouse button when being
exactly on the horizon.\n\nFor more precision, display your picture at 100% or
even more if it is rather small. Can be used to correct vertical axes like high
```

```
buildings too. If the diagonal of a rectangular selection is used to determine
the angle, typing an angle has no effect. Crops to a rectangle inscribed in the
slanted rectangle obtained after the rotation if the crop check-box is cheked,
otherwise only selects the area it would crop to. Keeps the original aspect
ratio."
```

Podstawową zasadą w tłumaczeniu kodu źródłowego jest nie ruszać słów kluczowych. Słowa kluczowe są rozpoznawane przez interpreter i muszą pozostać w dokładnie takiej postaci, w jakiej napisał je Autor (programista).

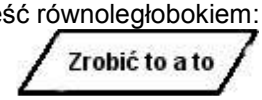
Również nazwy zmiennych, funkcji itp. konstrukcji zdefiniowanych w programie należy pozostawiać bez zmian. Po pierwsze dlatego, że są one zawsze pisane w języku angielskim i jest to swego rodzaju przyjęty nieformalnie standard. Nazwy w jakimkolwiek innym języku w kodzie programu będą wyglądać po prostu dziwnie. Z drugiej strony, jeśli zmienimy nazwę zmiennej lub funkcji w jednym miejscu, musimy znaleźć wszystkie jej wystąpienia w innych miejscach programu, co powoduje niepotrzebne komplikacje i bardzo łatwo przy tym o pomyłkę.
Dla przejrzystości i ładnego układu tekstu komentarza, możemy stosować znak nowej linii **\n**,
ale w stringu musimy wstawiać go po podwójnym apostrofie "\nOpis"

Programowe manipulacje z obrazami.

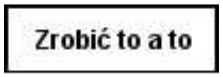
W jaki sposób wskażemy GIMP-owi, jak utworzyć obraz?

korzystano m.in. z:
http://www.seul.org/~grumbel/gimp/script-fu/script-fu-tut.html
http://docs.gimp.org/2.6/en/gimp-using-script-fu-tutorial.html
http://docs.gimp.org/2.6/en/plug-in-dbbrowser.html

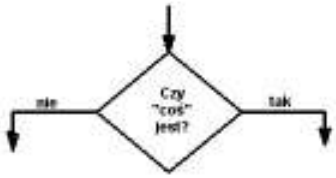
Zamierzając utworzyć obraz, musimy zastanowić się, co faktycznie kolejno ma mieć miejsce, ponieważ pisanie skryptów, czyli programowanie – to zdolność myślenia algorytmicznego. Algorytm to nic innego, jak po prostu przepis na wykonanie pewnych czynności dokładnie w określonej kolejności, aby za każdym razem, gdy je wykonamy osiągnąć pożądany rezultat. Programowanie oznacza zapisywanie algorytmicznie ujętych zadań (komputerowych przepisów kucharskich) w języku zrozumiałym dla GIMP-a.
Dobre programowanie polega na tym, żeby swego "przepisu" nie rozgadać, by zawierał tylko tyle i jednocześnie aż tyle poleceń, ile trzeba i ile wystarczy do uzyskania wyniku. Program robi dokładnie to co mu przepisano, potrafi podać czy w przepisie nie ma błędu w sformułowaniu, ale nie potrafi wykryć czy w programie nie ma błędnych zaleceń.
Czyli ujawni błąd ortograficzny w przepisie, ale nie poda, że wsypanie pieprzu do ciastka jest błędem. Żeby program (skrypt) precyzyjnie zapisać najlepiej rozrysować sobie najpierw zadanie w postaci graficznej, potem bardzo łatwo napisać program (skrypt) z schematu.
Widać wtedy "strukturę logiczną" zadania. Schemat lub zadania rysujemy z góry w dół, czyli treść pierwszej czynności musi znaleźć się najwyżej, a kolejne stopniowo coraz niżej itd.
Czynność pobierania czegoś, należy obwieść równoległobokiem:



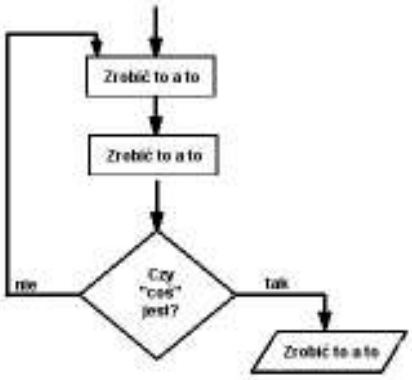
Jeśli czynność polega na obróbce czegoś, przetwarzaniu - należy obwieść prostokątem.



W każdym równoległoboku lub prostokącie może mieścić się co najwyżej jedna czynność.
Od poprzedniej do następnej czynności prowadzimy strzałkę.
Jeśli w trakcie wyk. Przepisu trzeba podjąć jakąś decyzję, której wynikiem będzie wykonanie następnej jednej z dwóch czynności, (by odp. brzmiała tak lub nie), treść tego pytania obwodzimy rombem postawionym na sztorc:



Jeśli w trakcie wykonywania "przepisu" trzeba po podjęciu decyzji wrócić do jakiejś czynności i wykonać ją jeszcze raz (co nazywa się "pętlą") rysujemy tak:



Oczywiście taki tok postępowania zależy od indywidualnych preferencji, na ogół przy krótkich skryptach dużo programistów pomija taki tok postępowania i czyni jak poniżej metodą uproszczoną.

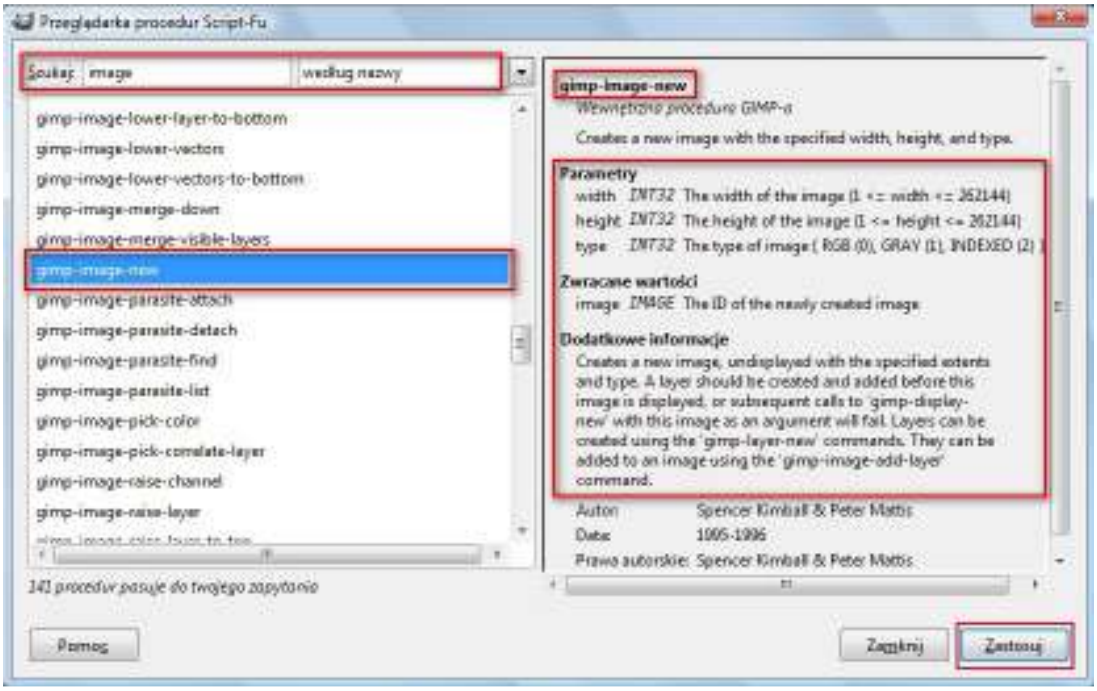
W celu zapoznania się z metodą spróbujemy najprostszej rzeczy: utworzymy nowy obraz o zadanych wymiarach, wypełniony określonym kolorem.

Tworzyć obraz, musimy się zastanowić, jakie czynności powinny być wykonanie i w jakiej kolejności.
1. Chcemy mieć obraz (i ewent. okno: warstwy, kanały, ścieżki itp.).
Co musimy wiedzieć o naszym obrazie? Ważne będą Wymiary i typ (RGB, skala szarości, indeksowany)
2. Następnie, w poniższym przykładzie, będziemy potrzebować warstwę, więc możemy ją dodać.
Co może być ważne w warstwie? Wymiar, krycie, tryby mieszania, itp. (omówimy to później, ale należy o tym pamiętać!)
3. Utworzyliśmy nasz obraz i warstwę, a co ma być w tej warstwie? Jeszcze nic. Musimy powiedzieć, GIMP, czy ma coś dodać. W naszym przykładzie chcemy wypełnić obraz kolorem. Dlatego musimy wskazać w **Konsoli Script-Fu** jak ma to zrobić.
4. Czy jeśli stworzymy nasz obraz i dodamy do niego warstwę to automatycznie pojawi się na ekranie? Jeśli robimy to w interfejsie GIMP-a, odpowiedź brzmi "tak", ale z **Konsoli Script-Fu**, odpowiedź brzmi "**nie**", obraz będzie tylko w pamięci. Musimy podpowiedzieć GIMP, aby pokazał nam wynik.

Określone powyżej zadania zrobimy w kilka minut. Będzie to kilka kroków, które zrobimy, aby osiągnąć nasz cel, przy czym nie będziemy robić ich w identycznej kolejności, jak opisane powyżej. Przede wszystkim chodzi o to aby pokazać, co, jak i dlaczego tak robimy.
A więc rozpoczynamy.
GIMP-a mamy uruchomionego, okno Edytor obrazów jest puste, zaczniemy proces tworzenia obrazu. Cały czas będziemy korzystać z **Konsoli Script-Fu** oraz **Przeglądarki Procedur.**

W pierwszym kroku rozpoczynającym tworzenie obrazu jest nam potrzebna procedura tworzenia obrazu.
`gimp-image-new`
W tym celu w oknie **Edytor obrazów GIMP**-a wybieramy z menu:
Filtry => Script-Fu => Konsola.
Po chwili otworzy się okno **Konsola Script-Fu** (dalej będę stosował skrót nazwy **KSF**). Najpierw klikamy "**Wyczyść**" co usunie opis z okna.
Na dole tego okna po prawej znajduje się ***Pole wprowadzania bieżącego polecenia***. Tutaj będą pojawiać się wywoływane procedury, których przykłady zostaną podane poniżej.
Okno Konsoli pozostawiamy cały czas otwarte!.
Teraz klikamy na przycisk "**Przeglądaj**", w celu otwarcia okna **Przeglądarka Procedur** (dalej skrót **PP**). Procedur (z Biblioteki procedur) lub funkcji w **PP** (możemy poszukiwać w polu wyszukiwania wg:

W polu wyszukiwania wpisujemy słowo "**image**", jak pokazano poniżej:

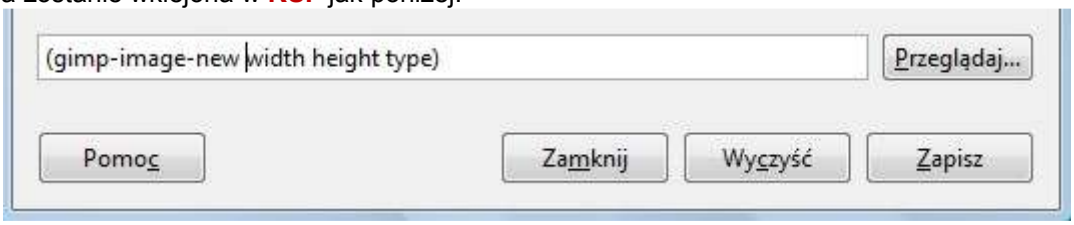


Jeżeli nazwa funkcji zaczyna się od słowa "`gimp`" - jest "rodowitą" procedurą GIMP. Nazwy zawierają odniesienie do obiektu, z którym funkcja działa. Na przykład, jeśli chcemy coś zrobić z warstwą, powinna zawierać, frazy wskazujące na ścieżkę poszukiwania "layer", aby znaleźć funkcje, które działają na wybranych obszarach – "select", itp.

Większość procedur w oknie *Przeglądarki procedur* ma nazwę odpowiadającą nazwie w GIMP. Przesuwamy suwak w dół, aż po chwili gdy dojdziemy do procedury o nazwie "**gimp-image-new**", czyli tego co jest nam w tej chwili potrzebne.
W prawej części okna, widzimy kilku bardzo ważnych informacji, które musimy znać i stosować.
Widzimy, że "**`gimp-image-new`**" tworzy nowy obraz o określonej szerokości, wysokości i typie
(Np. .. 400 x 300 RGB, itd.).
W sekcji **Parametry**, widzimy:
width *INT32* (szerokość): 1 <= width <= 262144,
height *INT32* (wysokość): 1 <= width <= 262144 w pikselach
INT32: oznacza liczbę całkowitą ***integer*** ze znakiem (bez miejsc po przecinku) o długości 32 bitów. Zmienne typu integer nie mogą pamiętać dowolnie dużych liczb całkowitych.
i **typy** obrazów mogą być:
RGB (**0**) – tworzy kolorowy obraz trybu RGB,
GRAY (**1**) – tworzy obraz w gradacji szarości,
INDEXED (**2**) – tworzy obraz indeksowany
To są parametry, których GIMP oczekuje, aby je poprawne podać. Jeśli ich nie podamy lub wpiszemy je nieprawidłowo, dostaniemy zwrócony błąd.
W sekcji **Zwracane wartości** widzimy, że jeśli podamy poprawne parametry, GIMP zwróci **ID** nowo tworzonego obrazu (jak podano już powyżej, będzie on tylko w pamięci, do chwili, aż powiadomimy **KSF**, aby nam go pokazała).
Sekcja **Dodatkowe informacje** również podaje nam bardzo przydatne informacje. I nie obejmuje to tylko omawianych tutaj, ale trzeba mieć tego świadomość i czytać dla każdej procedury.

Tworzymy nasz obraz!

Po podświetleniu procedury (**`gimp-image-new`**), klikamy przycisk "**Zastosuj**" w dolnej części okna **PP**.
Procedura zostanie wklejona w **KSF** jak poniżej:



Zauważymy, że kursor został od razu ustawiony w odpowiednim miejscu. Jest to bardzo ładna funkcja!

Zanim uczynimy cokolwiek innego, teraz musimy wypełnić wymagane parametry.

Powyżej powiedziałem, że musimy mieć parametry, które musimy koniecznie teraz włączyć?
Mamy zaznaczone miejsce, gdzie będziemy je wprowadzali.

W naszym przykładzie mamy zamiar utworzyć obraz, który ma 800 pikseli szerokości, 600 pikseli wysokości i typ RGB.
Widać, że **PP** podpowiada nam, iż dla tej procedury należy wpisać *nasze parametry* (width, height, type) **szerokość i wysokość obrazu** oraz jego **typ**.

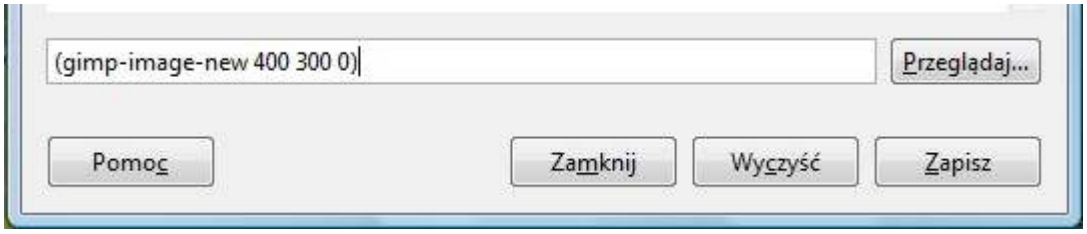
Ponadto, jak już wcześniej wielokrotnie podano **KSF** oczekuje, że całość będzie zawarta w nawiasach.

Tak więc, aby stworzyć obraz RGB 400 x 300 pikseli, musimy zmodyfikować procedurę, po prostu wkleić do **KSF** nasze parametry, umieszczając je w przestrzeni gdzie w odpowiednim miejscu był ustawiony kursor, czyli tak:

```
(gimp-image-new width height type)
(gimp-image-new 400 300 RGB)
```

Przy czym wpisując typ obrazu, możemy podać parametr korzystając z zapisu np.: RGB lub 0 itp. j/w.
ale RGB jest bardziej opisowy, gdy patrzymy na kod

Oto zrzut ekranu, jak będzie to wyglądać w **KSF**:



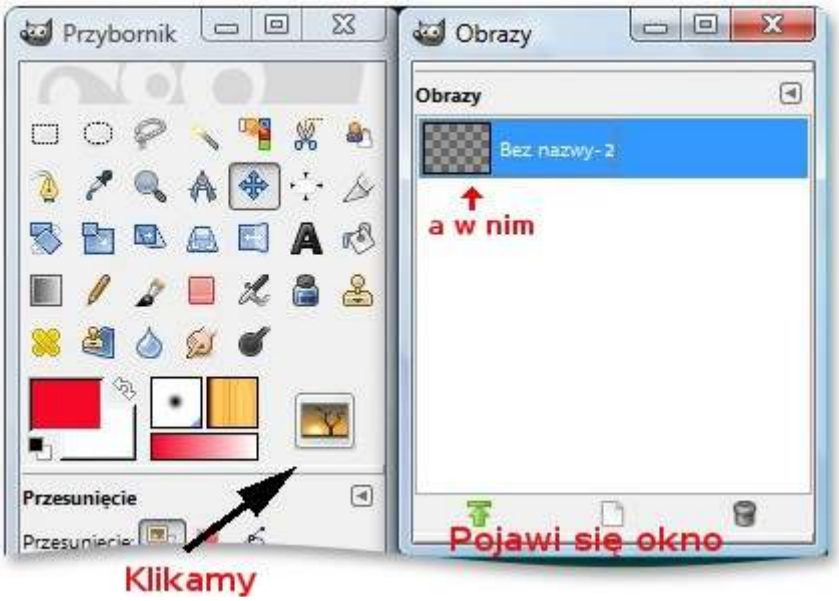
Teraz klikamy **Enter** i KSF zwróci coś takiego:

(Jeśli wpiszemy coś niewłaściwie, pojawi się błąd i trzeba będzie ponownie wprowadzić parametry)
Dlatego przypomnienie:
Gdy przy wpisaniu w ***Pole wprowadzania bieżącego polecenia*** w oknie **Konsola Script-Fu** popełnimy błąd, możemy użyć na klawiaturze strzałki w górę, aby w Polu wprowadzania ponownie pojawiło się ostatnie polecenie. Co umożliwi wyedytować procedurę ponownie i wprowadzić poprawki. To oszczędza czas!.

Pierwszy wiersz w powyższym zrzucie – to procedura, która została wykonana.
Drugi wiersz to wartość, którą zwróciła procedura.
Jak widać na powyższym zrzucie, zwrócona przez **KSF** wartość, to:
jednoelementowa lista (bo jest w nawiasach),
którą interpreter przypisał do naszego nowego obrazu .... w naszym przypadku, to liczba **(2)**.
Nie martwimy się, jeśli cyfra będzie inna (zależy to od tego, czy nie robiliśmy jakichś innych wcześniejszych prób po otwarciu KSF i nie wyłączono GIMP-a), zapisujemy ją na kartce jako identyfikator **ID** przed chwilą utworzonego obrazu .... za chwilę będziemy potrzebować tego **ID**, w charakterze argumentu do kolejnej procedury w KSF.!
Uwaga:
To co napisano powyżej, jest bardzo ważne dla dalszych naszych działań, ponieważ w trakcie omawiania kolejnych kroków, będziemy manualnie wpisywać identyfikator ID do dalszych procedur . Natomiast kiedy potem będziemy pisać już nasz pierwszy prawdziwy script-fu musimy podać interpreterowi instrukcję, aby sam wyciągnął z listy "głowę" przy pomocy funkcji **car**.
Np.: `(image (car (gimp-image-new 400 300 RGB)`

Dotychczas wszystko niby jasne, ale gdzie znajduje się właśnie stworzony obraz ?
Powtórzę - odpowiedź jest bardzo prosta: nie na Ekranie, jest tylko w pamięci.
Można powiedzieć, że Script-Fu zakłada, iż może jeszcze będziemy robić kilka dalszych rzeczy z obrazem przed pokazaniem na ekranie.

Jeśli nie wierzymy można sprawdzić w oknie Przybornik:

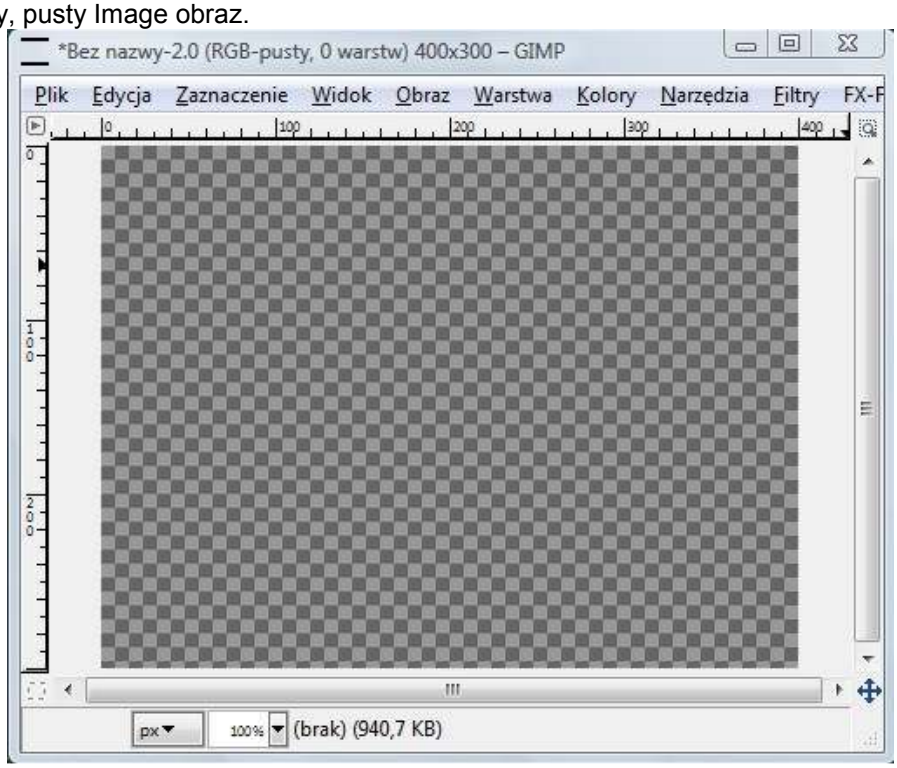


będzie minimum jeden wpis o nazwie "Bez nazwy- " i zobaczymy ID.

Oczywiście tylko dla zabawy, ale możemy również, owoc naszej pracy, aż do tego momentu, zobaczyć wyświetlony w oknie **Edytor obrazów GIMP**-a wpisując procedurę:

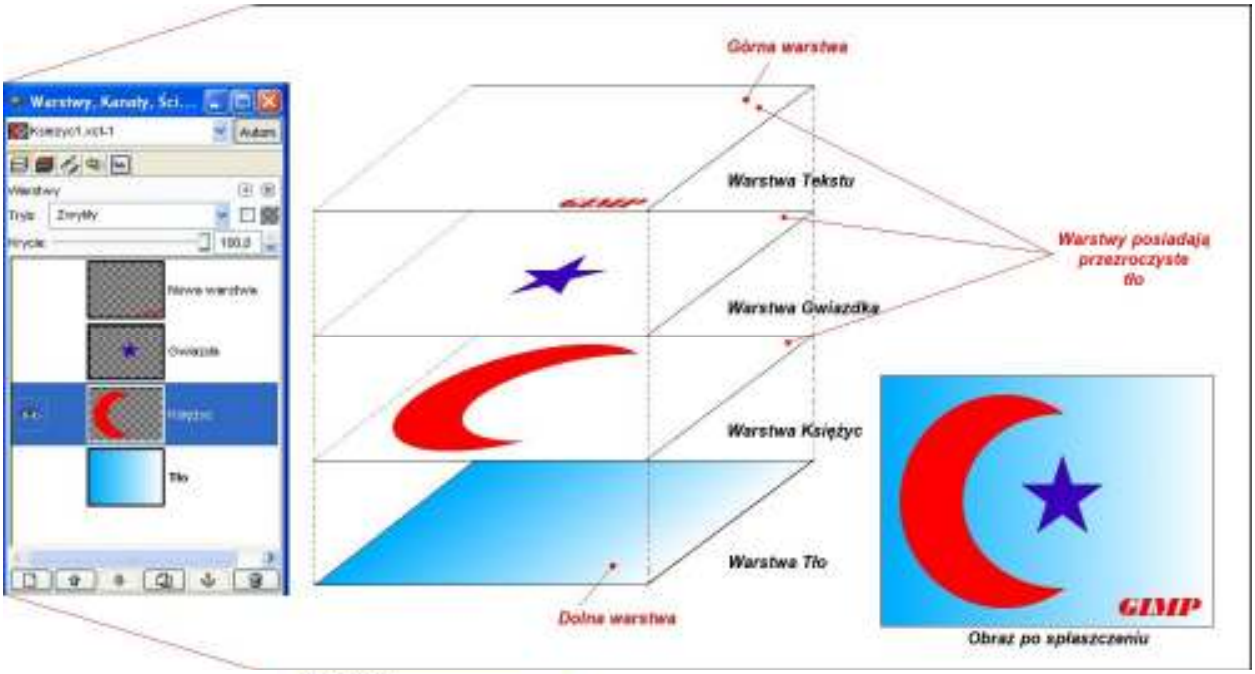
```
(gimp-display-new 2)
```

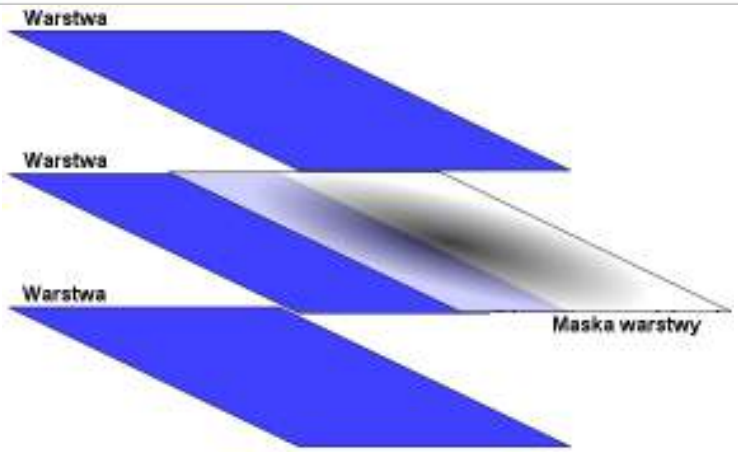
zobaczymy nowy, pusty Image obraz.



(Warto zwrócić uwagę, że wokół "obrazu" nie ma jeszcze charakterystycznych mrówek.)
Jeśli obraz nie ma warstw, nie da się używać żadnych narzędzi z **Przybornika** (np. do malowania lub selekcji).
Dlaczego tak się dzieje, musimy przypomnieć pewne ogólne stwierdzenia dotyczące pracy w edytorze graficznym.

Co to jest w tym momencie **Image** – jest to, mówiąc obrazowo Ramka lub przestrzeń o określonych wymiarach i proporcjach z możliwością wypełnienia – wstawienia obiektu (papieru, folii, płótna…) na którym będzie można malować.
Malowanie tradycyjnie, to oczywiście zastosowanie płótna w sztalugach czyli ramach. Można także rysować na kartach papieru, folii itp.
A więc musimy zawsze mieć "coś" jako podłoże - materiał (różne rodzaje płótna, tekturę, papier, folię...itp), na czym będziemy malowali przy pomocy narzędzi - funkcji malarskich.
Płótno to jest coś, na czym można rysować, to warstwy, ale także kanały, maski warstw, selekcje, itd.





Określenia:
Drawable`s – to generalnie obiekty, na których można rysować, "coś" na czym można wstawiać piksele.
Drawable`s - to kategoria obiektów, do których możemy się dostać i czasem zmienić dane pierwotne.
Tak więc: "drawables" jest w GIMP-ie pojęciem, które obejmuje warstwy, ale także wiele innych rodzajów rzeczy związanych z obrazem, takich jak kanały, maski warstw i maski zaznaczenia, wszystkie są drawable`s.
Drawables, jak sama nazwa sugeruje, są rzeczami, które umożliwiają rysowanie na nich, są obiektami Pixmap.
Drawable – to funkcje rysujące (do rysowania) którymi możemy malować, wypełniać, itp.

Jeżeli warstwa jest aktualnie aktywna, to jest podświetlona w oknie dialogowym Warstwy, a jej nazwa jest wyświetlana w *obszarze stanu* okna obrazu.

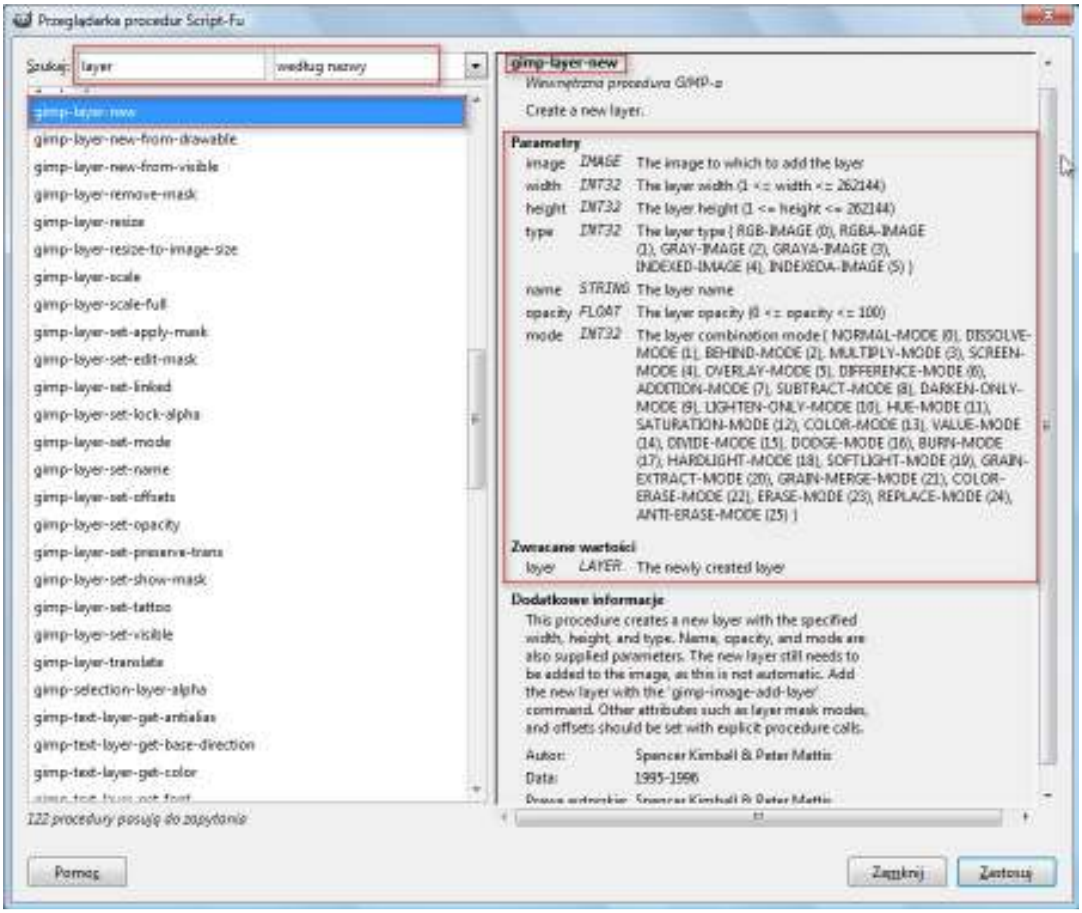
Każdy otwarty obraz musi mieć w każdej chwili co najmniej pojedyncze *active drawable*.

Dla wyprowadzenia obrazu, potrzebujemy więc co najmniej jednej warstwy.

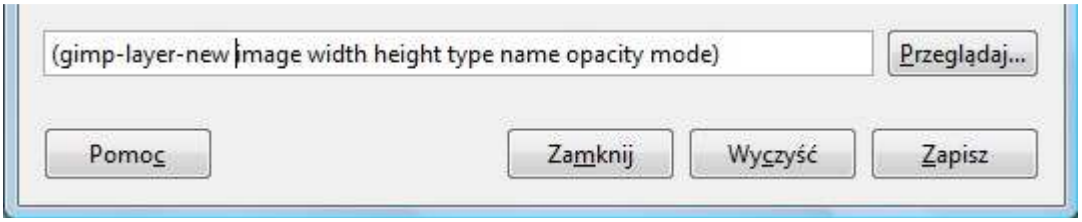
Dlatego, następnym krokiem będzie, stworzenie warstwy.
Zanim zaczniemy, muszę dodać, że nie będziemy omawiać każdego drobiazgu w szczegółach tak jak powyżej. Wszystko, co robiliśmy będzie po prostu dalej podobne jak dotąd.
Będą podpowiedzi, co napisać i czego oczekiwać, ale bez potrzeby tak szczegółowych kroków.

Tworzymy warstwę

W tym celu wracamy do **PP** i wpisujemy w **Szukaj** „**layer”**, podobnie jak zrobiliśmy powyżej dla „**image**”, przewijamy suwak w dół, aż dojdziemy do "**gimp-layer-new**".
Jak widać, w sekcji parametrów, ta procedura oczekuje, aby określić aż 7 parametrów.



Podobnie jak to było w przypadku tworzenia obrazu, klikając przycisk "**Zastosuj**", kopiujemy i wklejamy "**gimp-layer-new**" do **KSF**:
Oto zrzut ekranu, jak będzie to wyglądać w **KSF**:



Zrzut pokazuje, że procedura utworzenia warstwy w GIMP, oczekuje określenia 7 parametrów.
Parametry **kolejno** są następujące:
Image ID - to otrzymaliśmy gdy tworzyliśmy obraz
Width - jaką chcemy szerokość naszej warstwy. Warstwa może być większa lub mniejsza niż wymiary obrazu, ale przyjmiemy, że są równe.
Height - takie same informacje jak dla szerokości.
Type - typ warstwy { RGB-IMAGE (**0**), RGBA-IMAGE (**1**), GRAY-IMAGE (**2**), GRAYA-IMAGE (**3**), INDEXED-IMAGE (**4**), INDEXEDA-IMAGE (**5**) }
Name – nazwa warstwy, która GIMP doda. Nazwa to łańcuch, musimy dołączyć tą informacje w cudzysłowie.
Opacity – krycie warstwy, przezroczystość warstwy, jako procent. To jest wartość float, dzięki czemu można wprowadzić cyfry z miejscem po przecinku. Waha się od 0% krycia do 100%, a ponieważ chcemy, aby naszą warstwę było widać ustawiamy na 100.
Mode - tryby mieszania warstw mogą być: { NORMAL-MODE (**0**), DISSOLVE-MODE (1), BEHIND-MODE (2), MULTIPLY-MODE (3), SCREEN-MODE (4), OVERLAY-MODE (5), DIFFERENCE-MODE (6), ADDITION-MODE (7), SUBTRACT-MODE (8), DARKEN-ONLY-MODE (9), LIGHTEN-ONLY-MODE (10), HUE-MODE (11), SATURATION-MODE (12), COLOR-MODE (13), VALUE-MODE (14), DIVIDE-MODE (15), DODGE-MODE (16), BURN-MODE (17), HARDLIGHT-MODE (18), SOFTLIGHT-MODE (19), GRAIN-EXTRACT-MODE (20), GRAIN-MERGE-MODE (21), COLOR-ERASE-MODE (22), ERASE-MODE (23), REPLACE-MODE (24), ANTI-ERASE-MODE (25) }.

Przygotujmy sobie wpis:
2 400 300 RGBA-IMAGE "Zbyma Layer" 100 NORMAL-MODE
lub
2 400 300 1 "Zbyma Layer" 100 0

Sprawdzamy czy mamy wykonany poprawny zapis RGBA-IMAGE i NORMAL-MODE;
Poczym Ctrl+C i Ctrl+V

Pamiętamy:

Do zapisu stałych **Type** i **Mode**: RGBA-IMAGE; NORMAL-MODE; FOREGROUND-FILL; używamy bezwzględnie wielkich liter – *wersaliki*, separatory rozdzielające ciąg znaków nie stosujemy dolnego podkreślenia "_" tylko minus "– "; nie stosujemy spacji - odstępu pomiędzy dwoma wyrazami.

Ponadto, musimy dać unikalną nazwę warstwy, upewniamy się, że jest ***wpisana w cudzysłowie prostym*** (należy pamiętać, że jest to **łańcuch**).

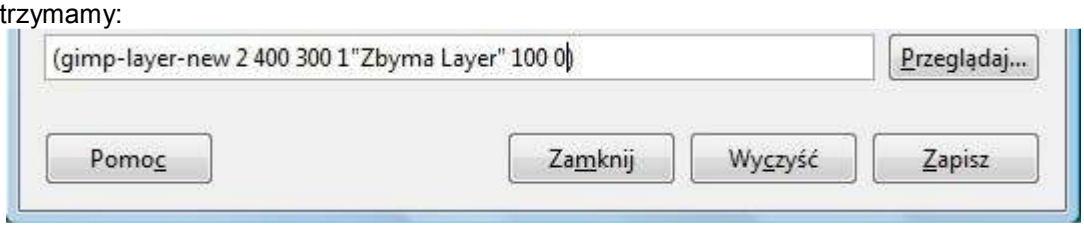
Przypominam:

W TinyScheme musimy stosować **White Sapce** "białe znaki" pomiędzy operatorem matematycznym (lub inną nazwą funkcji lub identyfikatorem), aby był on prawidłowo interpretowany przez Konsola Script-Fu, podobnie pomiędzy argumentami.

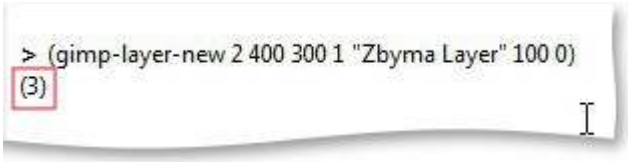
"Białe znaki" (takie jak spacje czy tabulatory) np. pomiędzy argumentami nie mają znaczenia, są ignorowane przez interpreter programu *TinyScheme,* a więc mogą być stosowane obficie i przyczynić się do wyjaśnienia i organizowania kodu w skrypcie.

W oknie **Konsoli TinyScript-Fu**, musimy wpisać całe polecenie w jednej linii, to jest **wszystkie wyrażenia między nawiasem otwarcia i zamknięcia muszą być w jednej linii** w oknie Konsoli.

Oto co otrzymamy:

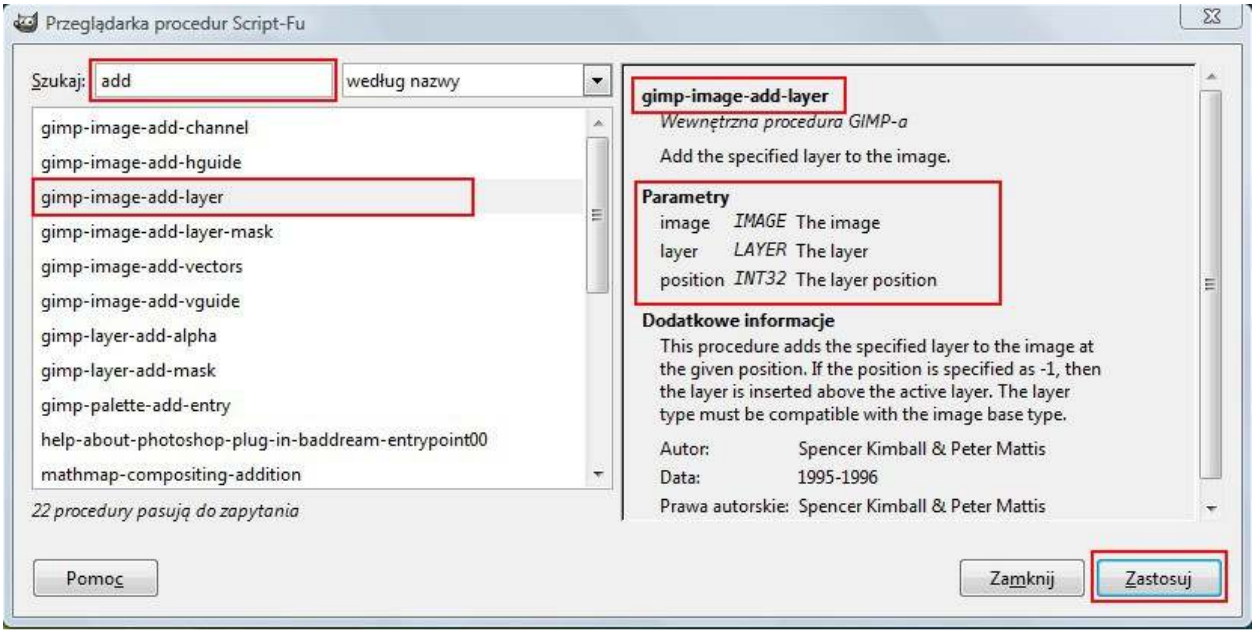


Gdy wkleimy, klikamy **Enter**, jeśli zrobiono wszystko poprawnie, **KSF** zwróci inną wartość, jak pokazano poniżej. Jest to wartość przypisana do nowej warstwy. Ponownie zapisujemy tą cyfrę, ponieważ będzie ona potrzebne później. Poniżej *Zwrócona wartość* otrzymana w KSF, ID naszej warstwy to (lista) "**3**".



Dodanie warstwy do obrazu

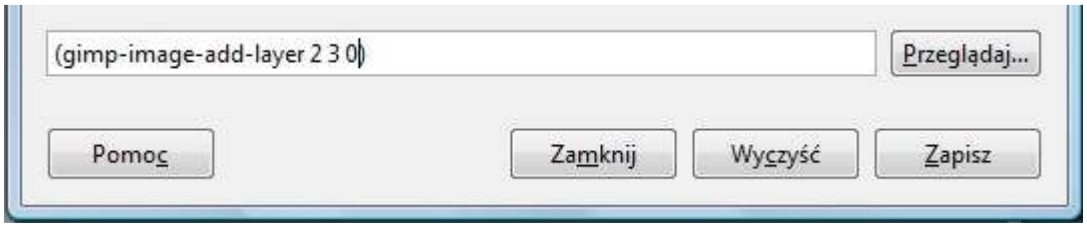
Stworzyliśmy warstwę, ale ta warstwa istnieje tylko w pamięci GIMP. Musimy dodać ją do obrazu. Wracamy do **PP** i wpisujemy „**add**” – (dodaj) i wybieramy procedurę "**gimp-image-add-layer**"



Z **dodatkowych informacji** dowiadujemy się, że procedura dodaje warstwę do obrazu w wskazaną pozycję: wartość position -1, prowadzi do tego, że nowa warstwa będzie wstawiona w górze stosu warstw, to znaczy będzie się znajdować powyżej wszystkich.
Oprócz tego*, warstwy, nie mające kanału- alfa* (przezroczystości) należy zawsze wstawiać w pozycji 0 (zero).
Typ warstwy musi być zgodny z typem obrazu bazowego, innymi słowy nie należy próbować w obraz w odcieniach szarości wstawiać kolorową warstwę.

Ponownie, po prawej stronie widzimy, że teraz procedura potrzebuje **3** parametry: ID obrazu, identyfikator warstwy i pozycję warstwy. Znamy już, ID obrazu, oraz identyfikator warstwy, ale musimy dodać jej pozycję.
Tak jak poprzednio, klikamy przycisk "**Zastosuj**", aby skopiować i wkleić procedurę "**gimp-image-add-layer**" do **KSF**:

Procedurę musimy wypełnić wymaganymi parametrami (tzn.: ID image obrazu, ID layer warstwy, **position**).
Będzie to (gimp-image-add-layer 2 3 0) 2 jest identyfikatorem obrazu, 3 jest identyfikatorem warstwy i ponieważ warstwa, nie ma kanału-alfa wstawiamy ją w pozycji 0 (zero) w tym przypadku najwyższa pozycja.
Wklejamy nasze parametry ID image / warstwy, oraz pozycję.
Oto co otrzymamy:

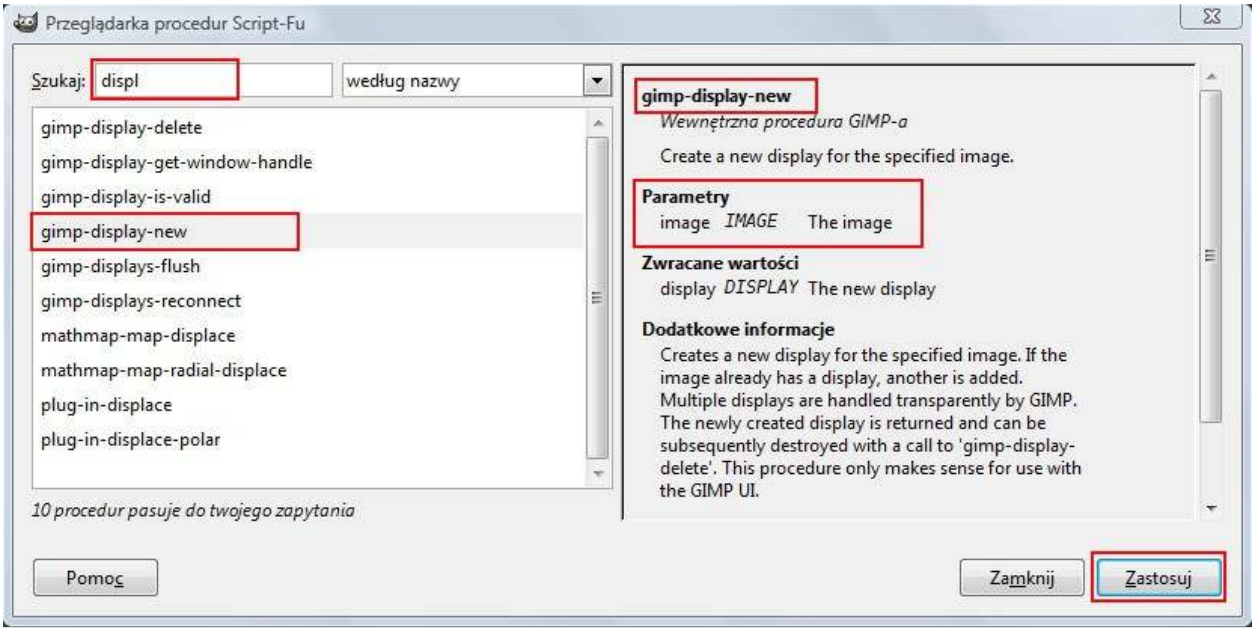


Klikamy klawisz **Enter**, jeśli zrobiono wszystko poprawnie, w oknie KSF powinno pojawić się następujące:

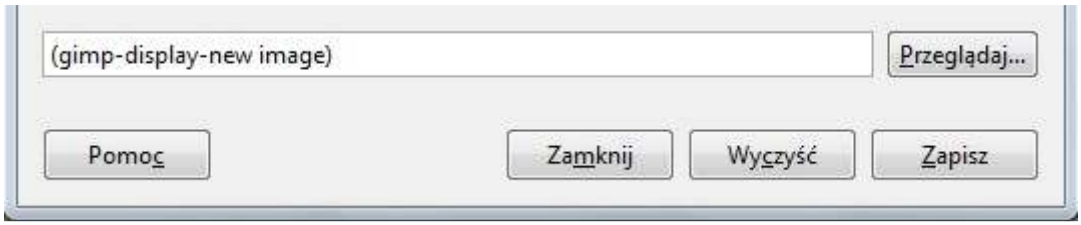
Wartości logiczne **true** *prawda* i **false** *fałsz* są reprezentowane odpowiednio symbolami **`#t`** i **`#f`**.
Widzimy że, GIMP zaakceptował procedurę, czyli nie ma błędów. W tym przypadku oznacza to, że warstwa została dodana do naszego obrazu.

Teraz możemy wreszcie zobaczyć obraz

Przeszliśmy przez te wszystkie kroki i musimy sprawdzić, czy nasz obraz naprawdę istnieje.
Chcemy dowodu! Dowód jest poniżej….
W KSF klikamy na przycisk "**Przeglądaj**" w otwartej **PP** wpisujemy w "*Szukaj*" **display** i przewijamy w dół do "**gimp-display-new**".

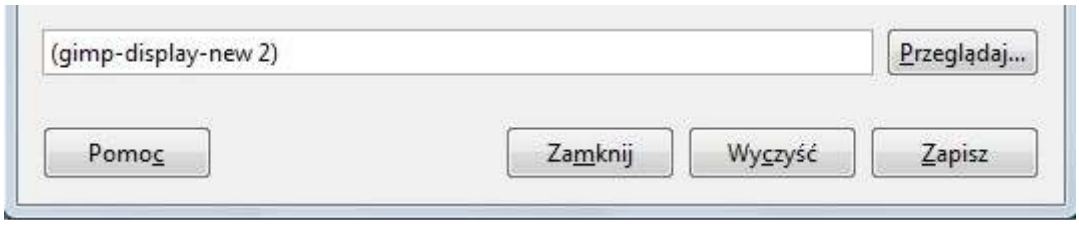


Ta Procedura wymaga jednego parametru, **ID** obrazu.
Klikamy przycisk "**Zastosuj**", aby skopiować procedurę z **PP** i wkleić ją jak wcześniej do KSF.



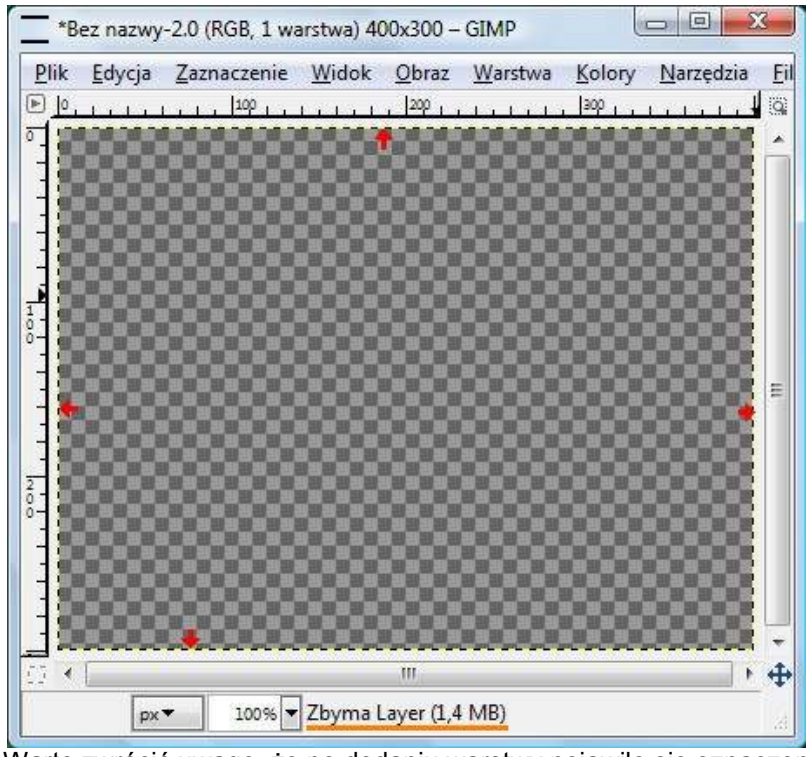
Teraz, jak poprzednio, dodajemy **ID** obrazu. Nasze **ID** obrazu było **2**, czyli zastępujemy powyższe *image* identyfikatorem ID obrazu **2**.

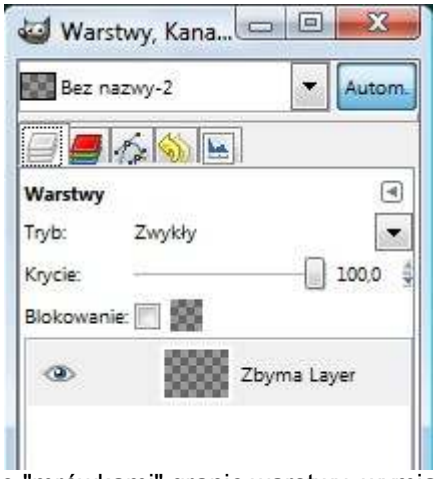
Ostatecznie procedura wygląda tak:



Po kliknięciu klawisza **Enter** powinno pojawić się:

oraz



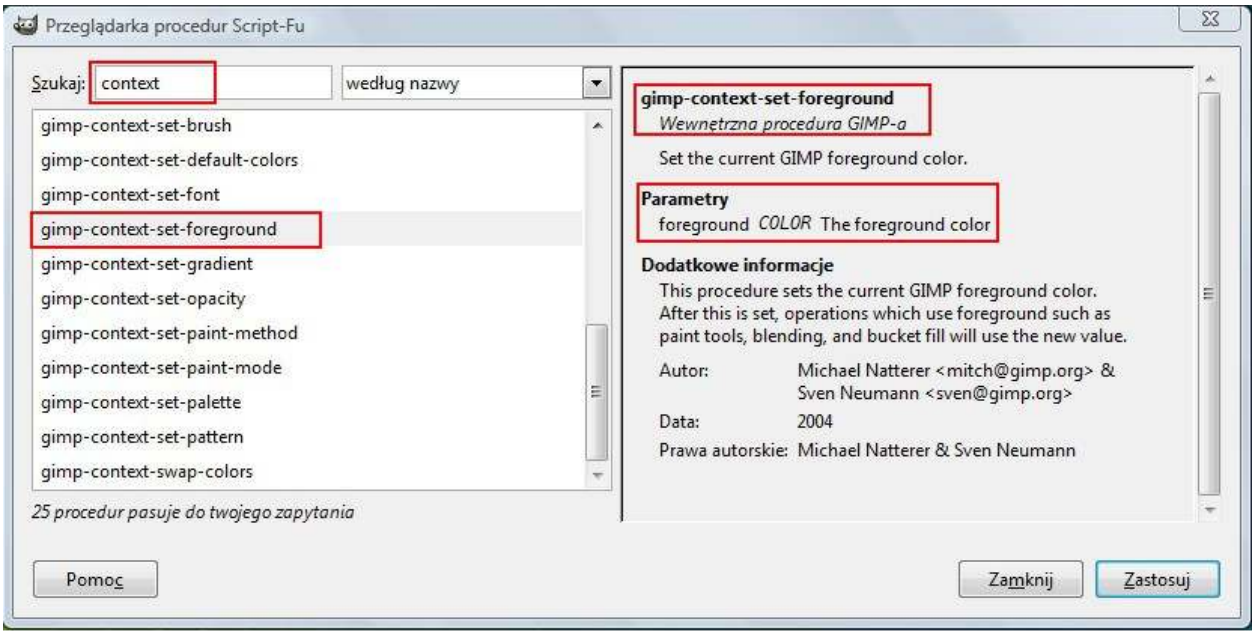


Warto zwrócić uwagę, że po dodaniu warstwy pojawiło się oznaczenie "mrówkami" granic warstwy, wymiar warstwy zgodny z image, oraz jej nazwa jest wyświetlana na dole w *obszarze stanu* okna obrazu.
Obraz **może** (ale nie musi) składać się z kilku przypadkowych kolorów, najczęściej pojawia się przezroczysty tak jak powyżej. Powodem tego jest to, że polecono GIMP, aby utworzył warstwę i dodał ją do obrazu, ale aż do teraz, nie została niczym wypełniona.
GIMP czasami wypełnia ją losowymi kolorowymi śmieciami (to dlatego, że Script-Fu alokuje pewien obszar pamięci dla obrazu, bez zawracania sobie głowy aby pozbyć się starej zawartości tej pamięci), dopóki nie jest poinformowany inaczej.
Jeśli tego chcemy (nie będziemy zmieniać tego w tym kroku, ponieważ będziemy to naprawić w następnym kroku), można wpisać w KSF: (**gimp-edit-clear drawable**) w naszym przypadku będzie to (**gimp-edit-clear 3**), aby wyczyścić ewentualną zawartość warstwy.

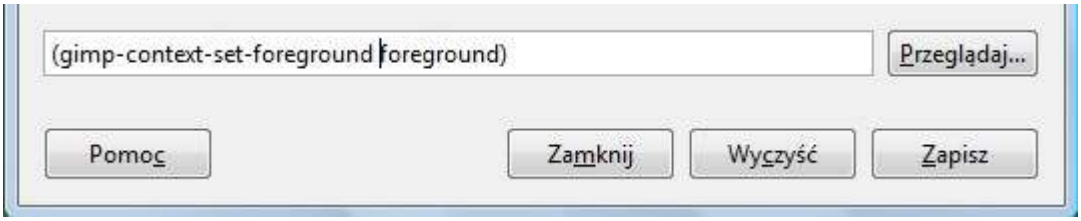
W oknie **Warstwy**, widzimy również nazwę warstwy (Zbyma Layer), którą nadano, kiedy ją tworzyliśmy.

Teraz ustalimy jakiś pożądany kolor warstwy.

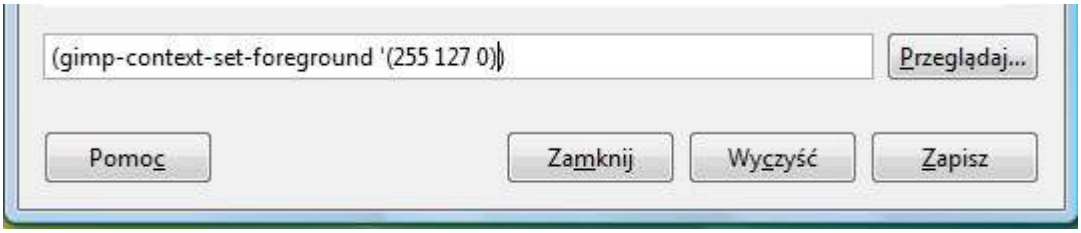
To już jest bardzo proste.
Znowu wywołujemy **PP** i wpisujemy słowo **context**. Procedurę context możemy ustawić na "defaults".
Możemy również wpisać " Foreground ".
Wybieramy procedurę o nazwie "**gimp-context-set-foreground**".



Zauważymy, że procedura oczekuje jednego parametru: ***Color***. Aby wypełnić ten parametr, musimy to zrobić w KSF. A więc, skopiujemy i wkleimy procedurę z **PP**, klikając przycisk "**Zastosuj**".



Aby wypełnić ją kolorem potrzebujemy **pojedynczej wartości,** która składa się z czerwieni (R), zieleni (G) i niebieskiego (B). Ta pojedyncza wartość składa się z listy 3 wartości (R, G i B). Jak już wiemy, do reprezentowania listy w KSF, używamy znaku apostrofu przed listą wartości, które są zawarte w nawiasach.
Oto jak będzie wyglądać nasza lista: **'(RGB)**
Przyjmijmy, że warstwa ma być koloru pomarańczowego.
Pomarańczowy jest reprezentowany w trybie RGB jako: **255 127 000**.
Podstawiając do naszej listy powyższy "*Szablon*", otrzymamy:
'(**255 127 000**) – zgodnie z tym co napisano na początku, GIMP odczyta tą listę jako jedną wartość.
Musimy połączyć listę z naszą procedurą, wstawiając ją zamiast *foreground,* otrzymamy następującą procedurę:
(gimp-context-set-foreground '(**255 127 000**))
– upewniamy się, czy apostrof i nawiasy są poprawne albo dostaniemy błąd.



Klikamy klawisz Enter i KSF zwróci:

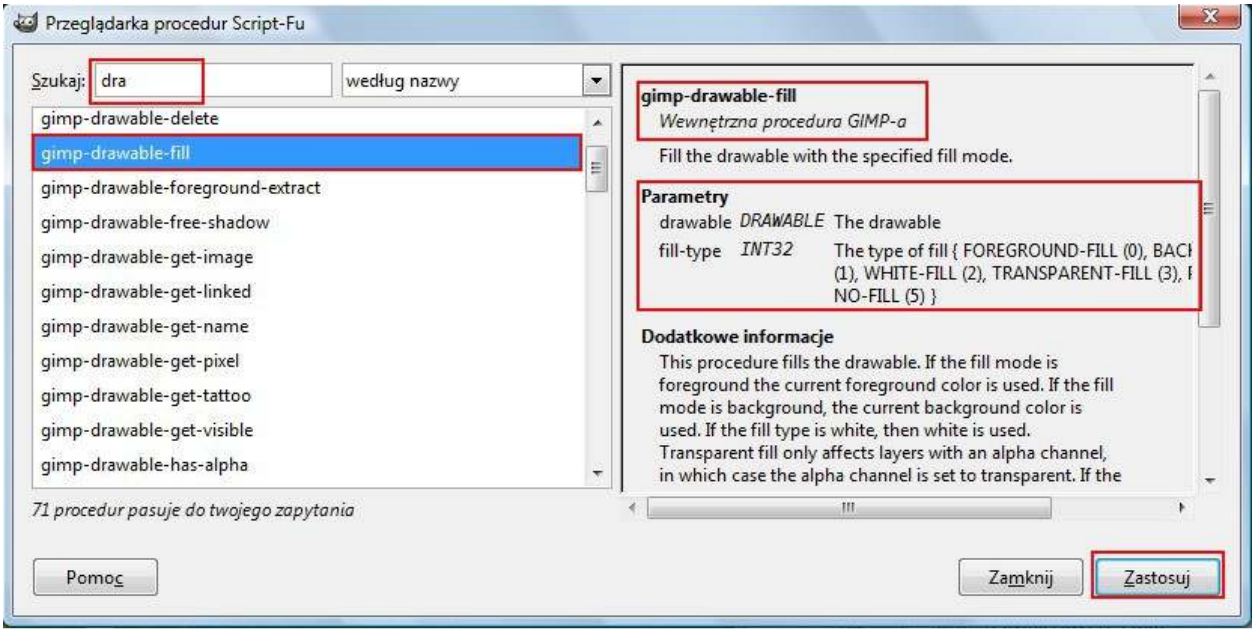
Sprawdzamy, czy nasz pierwszoplanowy kolor pomarańczowy, pojawił się w głównym oknie GIMP-a.
Poniżej widać że tak jest:

OK.
A więc jesteśmy już prawie u celu! Możemy wypełnić naszą warstwę kolorem pierwszoplanowym.

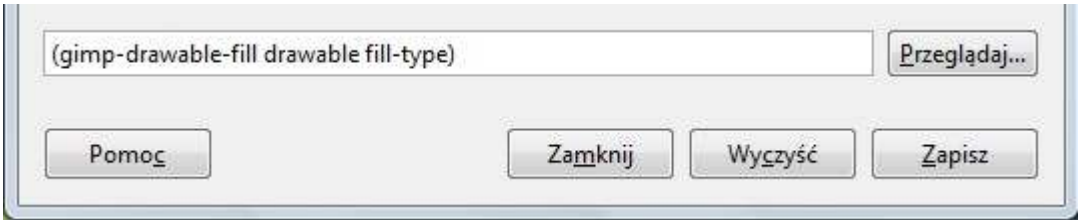
Przechodzimy do **PP** wpisujemy `gimp-layer-` i stwierdzamy, że mamy 42 takie procedury, ale nie ma wśród nich `gimp-layer-`fill.
Wpisujemy do **PP** `gimp-drawable-` i znajdujemy 60 procedur w tym "**gimp-drawable-fill**".
Funkcja *wypełnić warstwę* nazywa się `gimp-drawable-fill`, a nie `gimp-layer-fill` jak ktoś może się spodziewał, a to dlatego, że warstwa jest drawables, czyli obiektem do rysowania.

Więc musimy szukać funkcji służącej do manipulowania warstwą, funkcji, która umożliwia manipulacje na obiekcie do rysowania (na nim) – czyli w tym przypadku funkcji drawable-fill.
Co w GIMP-ie jest drawable można sprawdzić:
http://developer.gimp.org/api/2.0/libgimp/libgimp-gimpdrawable.html
Tak więc: warstwy, maski warstw, selekcje wszystkie są "drawables".
Drawables, jak sama nazwa sugeruje, są rzeczami, które wspierają rysowanie na nich, są obiektami Pixmap.

Image - obrazy nie są drawables, ale struktury danych które zawierają są drawables (można wciągnąć połączyć z obrazem).
Maski (albo dokładniej maski warstw) i kanały obydwie bez kanału alfa i w skali szarości są drawables; natomiast warstwy mogą być RGB, indeksowane, albo w skali szarości i mogą mieć albo nie mieć kanał alfa.
Kanały zawsze mają te sam wymiary i położenie jak obraz - jeśli zmieniamy wielkość obrazu wtedy wszystkie kanały także mają zmienioną wielkość.
Warstwy mogą być dowolnego wymiaru (minimum 1x1 piksela) i mogą zostać umieszczone, dowolnie w stosunku do obrazu (regiony warstwy mogą być poza granicami obrazu).
Maska (warstwy) powinna mieć zawsze wymiary i położenie warstwy, do której jest podłączona.
Znajomość różnic między rodzajami drawables może być bardzo pomocna dla piszących script-fu muszą starać się zrozumieć, jakie operacje mogą być stosowane (np. nie można zmienić offsetu kanału lub maski warstwy), jest również kilka rzeczy, które nie zachowują się jak można się spodziewać ("gimp-drawable-set-visible" nie działa z maskami warstwy).



Ta procedura wymaga dwóch parametrów: **drawable** i **fill-type** - typ wypełnienia.
Klikamy przycisk "**Zastosuj**", aby skopiować i wkleić procedurę do KSF.



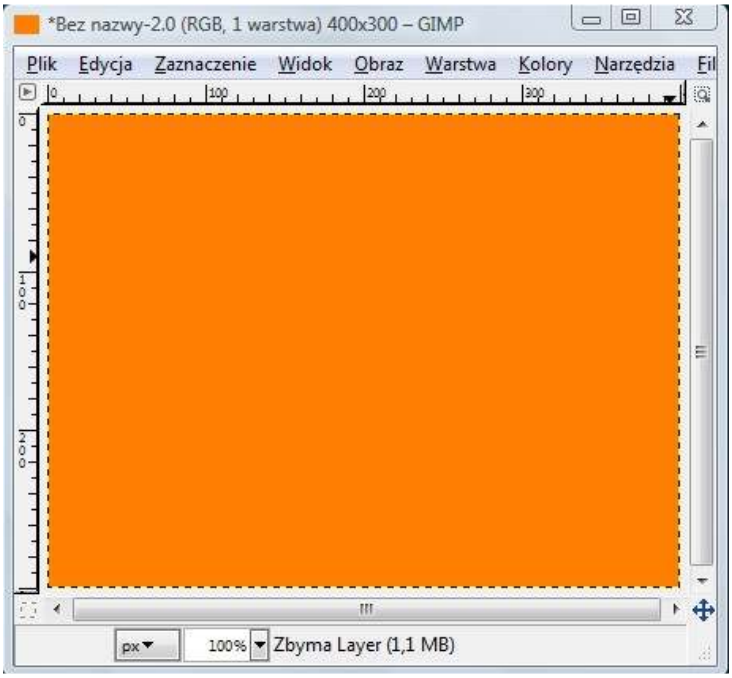
Teraz wpisujemy parametry identyfikator ID **drawable** warstwy (pamiętamy że jest to "**3**") oraz typ wypełnienia (używamy zapis stałej FOREGROUND-FILL (jak już wspomniano, nazwę stałej wpisujemy bezwzględnie wielkimi literami i nie stosujemy dolnego podkreślenia).
W **PP** były podane Typy wypełniania (*fill*) możliwe do zastosowania:
FOREGROUND-FILL lub (0),
BACKGROUND-FILL (1),
WHITE-FILL (2),
TRANSPARENT-FILL (3),
PATTERN-FILL (4),
NO-FILL (5) }

Powinno to ostatecznie wyglądać tak:

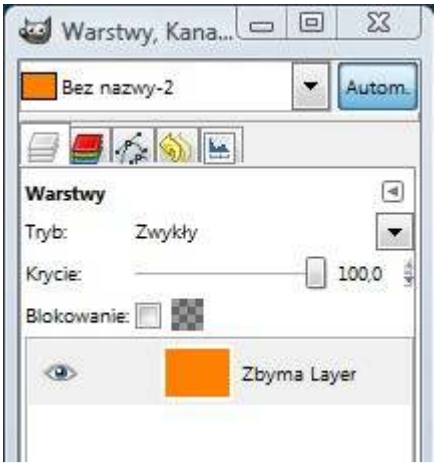
Klikamy klawisz **Enter** i zwrócona wartość:

```
> (gimp-drawable-fill 3 FOREGROUND-FILL)
(#t)
```

Pojawiający się obraz powinien wyglądać tak:



a w oknie dialogowym warstw zobaczymy to co pokazano poniżej:



No to jakoś dobrnęliśmy do końca !. Obraz można zapisać, usunąć lub zrobić z nim cokolwiek co nam przyjdzie do głowy.
Myślę, że w ten sposób nauczyliśmy się posługiwać **KSF** i **PP**.

```
> (gimp-image-new 400 300 RGB)
(2)
> (gimp-display-new 2)
(3)
> (gimp-layer-new 2 400 300 1 "Zbyma Layer" 100 0 )
(3)
> (gimp-image-add-layer 2 3 0)
(#t)
> (gimp-context-set-foreground '(255 127 000))
(#t)
> (gimp-drawable-fill 3 FOREGROUND-FILL)
(#t)
```

Pojawią się na pewno pytania:
Czy jest łatwiejszy sposób tworzenia Script-Fu?.
Odpowiedź – Nie ma! (lub bardziej precyzyjnie nie znam)
Czy można bardziej profesjonalnie i efektywnie wykorzystać KSF w pracy ?.
Odpowiedź - Oczywiście że tak.
Dlatego poniżej postaram się to pokazać.

Powyżej mogliśmy zauważyć, że wiele z wymienionych parametrów jest używane w kilku procedurach.
Możemy więc zdefiniować zmienne przy użyciu tej samej nazwy, jak te w procedurze i nadać zmiennym wartości.
Za każdym razem, kiedy klikniemy w **PP** na **Zastosuj** nasze parametry będą już wypełnione potrzebnymi danymi.

Przykładowo powyżej: procedura (`gimp-image-new`) wymagała parametrów: szerokość wysokość i typ.
Jeśli więc zdefiniujemy zmienne "`width`", "height" i "typ" i przypiszemy im pożądane przez Nas wartości, to podamy te informacje tylko raz i **KSF** zrobi resztę i w taki sposób możemy pracować bardziej wydajnie w KSF.

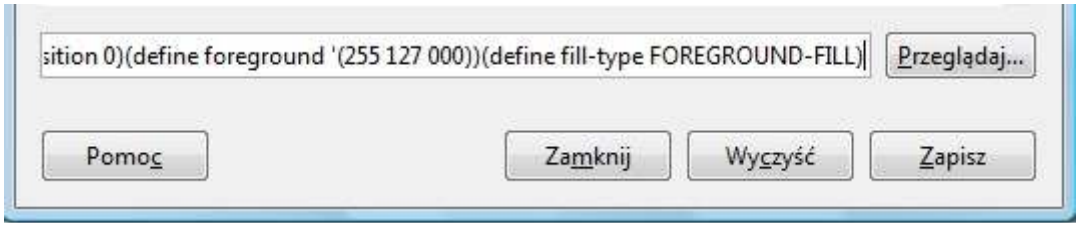
Jak podano w TinyScheme możemy definiować funkcje i zmienne z pomocą słowa kluczowego **`define`**.
W **Podstawy TinyScheme** mamy więcej szczegółowych informacji na ten temat.
Aby zdefiniować jakąś zmienną, używamy następującego formatu:

```
(define "Nazwa-zmiennej" par).
```

Przygotujemy i wpiszmy w KSF następujące polecenia:

```
(define width 400)
(define height 300)
(define type RGB)
(define name "ZbymaLayer")
(define opacity 100)
(define mode NORMAL-MODE)
(define position 0)
(define foreground '(255 127 000))
(define fill-type FOREGROUND-FILL)
(define image)
(define layer)
(define drawable)
```

Definicje możemy wprowadzać do KSF w ***Pole wprowadzania bieżącego polecenia***, pojedynczo lub jednorazowo.
Jeśli wpiszemy je razem (w jednej linii), będzie to wyglądało tak:



Po kliknięciu Enter, oto co pojawi się w oknie KSF:

```
> (define width 400)
(define height 300)
(define type RGB)
(define name "ZbymaLayer")
(define opacity 100)
(define mode NORMAL-MODE)
(define position 0)
(define foreground '(255 127 000))
(define fill-type FOREGROUND-FILL)
Widthheighttypenameopacitymodepositionforegroundfill-type
```

Upewnimy się czy GIMP ma przypisane wartości zmiennych.
W tym celu wystarczy wpisać nazwę jednej z nich width; height; opacity; i name (bez cudzysłowu lub nawiasu) do **KSF** i "Enter" po każdej z nich i oglądamy, jaka wartość zostanie zwrócona.
Możemy wpisać nazwy zmiennych, jedna po drugiej (ze spacją pomiędzy nimi) i też Enter.
Jakie wartości zostaną zwrócone?
Oto zrzut z KSF:

```
> width
400
> height
300
> opacity
100
> name
"ZbymaLayer"
```
W tym momencie pewnie pojawi się pytanie, dlaczego nie zdefiniowano również trzech ostatnich funkcji ("image", "layer", "drawable").
W poprzednich wersjach GIMP, można było je prezentować jako `(define image)` i GIMP to rozpoznawał.
Od ver. 2.6 GIMP-a ten sposób już nie działa i wartość musi być przypisana, dlatego aby GIMP zrozumiał, jaki mamy zamiar, definiujemy tymczasowo w taki sposób:
`(define image '())` gdzie `'()` [spotykane `nil`- pusta lista] lub wpisać 0.

W skrypcie można więc zapisać to jako:
```
(define image '())
(define layer '())
(define drawable '())
```

Konieczność użycia tego w GIMP od ver. 2.6 wymaga jeszcze dodatkowych wyjaśnień.

Jeśli spojrzymy wstecz, na obszar gdzie, już opisano procedury, parametry i zwracane wartości, mamy wszystkie potrzebne parametry do szybkiego tworzenia obrazu w KSF bez zmiany wartości.

Na początku, wpisywaliśmy procedury dla obrazu lub warstwy bezpośrednio do **KSF** np.:
```
(gimp-image-new 400 300 RGB)
```
i otrzymaliśmy zwróconą wartość ID dla Obrazu lub Warstwy, które potem manualnie wpisywano do dalszych procedur.

Wiemy już, jak przypisać zmienne, ponieważ robiliśmy to już powyżej dla width, height, type, itp.
Pojawia się jednak pytanie, jak możemy przypisać wartość z listy do zmiennej?
W **Podstawach** podano już jak otrzymać wartość z listy.
W TinyScheme istnieją dwie funkcji o nazwie "car" i "cdr" (i wiele ich kombinacji), które są używane do wyodrębniania "*głowy*" i "*ogona*" z list.
Jeśli chcemy pierwszej pozycji zawartej w liście czyli "*głowy*", stosujemy "car", jeśli chcemy wszystko pozostałe w liście, czyli "*ogona*" możemy użyć "cdr".
Łączenie car i cdr, pozwala do pobierania elementów z każdej pozycji na liście.
W **Podstawy TinyScheme** mamy więcej szczegółowych informacji na ten i inne tematy związane z programowaniem Script-fu.
Chcemy przypisać wartość pozycji z różnych list zmiennych, które tworzą nasze image, layer i drawable.
Jak podano w Podstawach, każdej nazwanej stałej i zmiennej można przypisać nową wartość.
Służy do tego specjalny (assignment operator) operator przypisania, *słowo kluczowe* **`set!`**.
Ma ono dwa argumenty. Pierwszym jest nazwa zmiennej, a drugim nowa przypisywana wartość.
```
(set! <variable> <expression>)
(set! zmienna wyrażenie)
```
`set!` zmienia wartość zmiennej na nową, najczęściej policzoną już w jakimś wyrażeniu.
A więc użyjmy słowa kluczowego **`set!`** aby przypisać wartości zmiennych.
Przykład:
```
(set! image (car (gimp-image-new width height type)))
(set! layer (car (gimp-layer-new image width height type name opacity mode)))
(set! drawable (car (gimp-image-get-active-drawable image))).
```
Podajmy GIMP definicję zmiennej obrazu, korzystając z wyniku uzyskanego za pomocą procedury `(gimp-image-new)`, która ma parametry `width, height, type` (mamy już zdefiniowane te zmienne).
Jak wspomniano, *poprzednio*, po prostu wpisywano procedurę bezpośrednio do KSF i zwracała ona jednoelementową listę zawierającą tylko cyfrę, czyli GIMP, podawał pierwszą pozycję na tej liście za pomocą funkcji "car", przypisaną jako wartość zmiennej "image". To samo dla layer i drawable.

Wpiszmy kolejno do KSF:
```
(set! image (car (gimp-image-new width height type)))
(set! layer (car (gimp-layer-new image width height type name opacity mode)))
```
(jako osobne polecenia i Enter po każdym poleceniu):

Co KSF zwróci?
W moim przypadku otrzymałem: 3 (reprezentuje ID obrazu) i 6 (ID warstwy).

```
> (set! image (car (gimp-image-new width height type)))
3
> (set! layer (car (gimp-layer-new image width height type name opacity mode)))
6
```
Jeśli pojawi się Error, coś zostało wpisane niepoprawnie.

Do tego momentu nie podaliśmy do KSF zmiennych drawable, ponieważ te zmienne będą wynikiem włączenia **`get-active-drawable...`**, a to będzie możliwe dopiero gdy warstwa faktycznie zostanie dodana do obrazu.

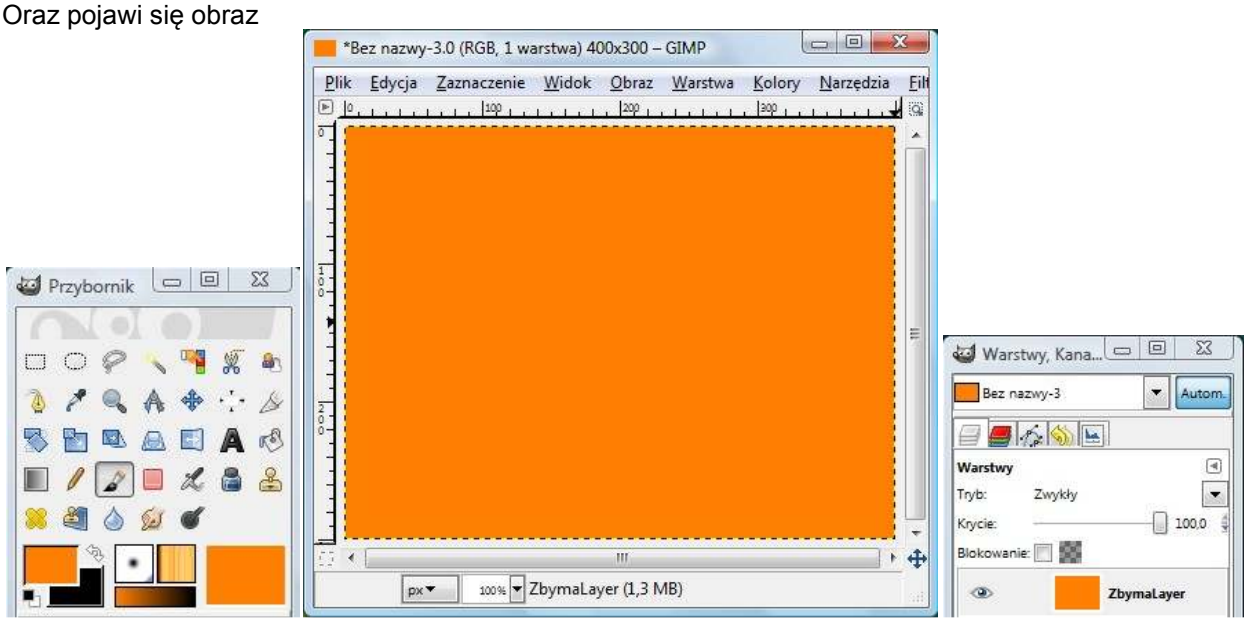
Wszystkie, drawable zostały poprzednio określone:

```
(gimp-image-add-layer image layer position)
(set! drawable (car (gimp-image-get-active-drawable image)))
(gimp-context-set-foreground foreground)
(gimp-drawable-fill drawable fill-type)
(gimp-display-new image)
```

Teraz powyższe zaznaczamy i Ctrl+C => Kopiuj => Ctrl+V => Wklej do KSF i **Enter**

okno KSF będzie wyglądać następująco:
```
> (gimp-image-add-layer image layer position)
(set! drawable (car (gimp-image-get-active-drawable image)))
(gimp-context-set-foreground foreground)
 (gimp-drawable-fill drawable fill-type)
 (gimp-display-new image)

(#t)6(#t)(#t)(1)
```

Oraz pojawi się obraz



Jeśli otrzymamy komunikat Error, popełniono jakiś błąd.

Przypomnienie: jeśli po wpisaniu polecenia w KSF popełnimy błąd, możemy użyć strzałki w górę na klawiaturze, aby wprowadzić ostatnie polecenie, które zostanie wpisane ponownie. Teraz możemy poprawić polecenie.

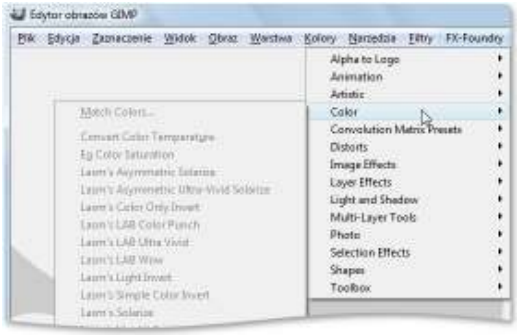
Tworzenie skryptu z dotychczasowych informacji

Możemy tworzyć dwa rodzaje Script-Fu:
Pierwszy typ to skrypty, wykonujące operacje na obrazie, które można wywołać tylko wtedy, gdy użytkownik otworzy plik obrazu, czyli **Image-dependent - podległe obrazowi**, wszystkie skrypty, które działają na kolory, dostosowują odcień, nasycenie, jasność itd. ...
Drugi rodzaj skryptów – to **autonomiczne**, które wręcz przeciwnie tworzą obraz na podstawie danych dostarczonych przez użytkowników w trybie interaktywnym, czyli Script-Fu **Standalone**.

Pierwszy typ skryptów powinien rozpocząć się w ścieżce od prefiksu "<**Image**>/"a w autonomicznych np. od "<**Utwórz**>/ "(w rzeczywistości mamy jeszcze bardziej prostą sytuację, ponieważ dotyczy to prefiksu, a zatem może istnieć wiele rodzajów skryptów takich jak "<Layers>" "<Channels>" czy "<Patterns>).
Prefiks to nazwa pozycji w menu głównym, czyli jest to już istniejąca nazwa (np. "Plik", "Obraz", "Filtry", itp.) lub nowa pozycja w menu np. Script-Fu.
Jeśli skrypt ma znajdować się w podmenu (np. Filtry / Renderowanie / Chmury), potrzebna jest nazwa lub nazwy poziomów menu oddzielone / (ukośnikiem), więc ścieżka wynikająca z polecenia podobna do zwykłej metody wprowadzania ścieżki do pliku. Gdzie będziemy umieszczać swoje skrypty to kwestia własnych preferencji.
Przykład:

```
(script-fu-menu-register "script-fu-poradnik-obraz"
                         "<Image>/Script-Fu/Poradnik…" )
"<Toolbox>/Xtns/Misc/")z takim zapisem skrypt znajdziemy w <Plik/Utwórz.../Misc.
```

Wskazówka:
Jeśli próbujemy wywołać Script-Fu podległe obrazowi (**Image-dependent**), gdy użytkownik nie otworzy pliku obrazu, będą one wyszarzone w menu:

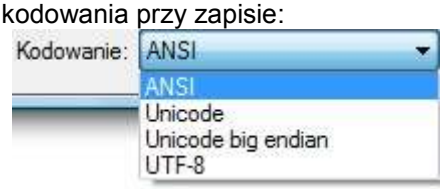


Niektóre Script-Fu pozostaną w menu wyszarzone, pomimo otwarcia pliku obrazu, jeśli wymagają dodanie do obrazu "kanału – **alfa**".

Spróbujmy pokazać jak tworzymy prosty skrypt typu **Standalone** z wszystkich informacji podanych powyżej.

```
(define (script-fu-poradnik-obraz)
(let*
(
(width 400)
(height 300)
(type RGB)
(name "ZbymaLayer")
(opacity 100)
(mode NORMAL-MODE)
(position 0)
(foreground '(255 127 000))
(fill-type FOREGROUND-FILL)
(image '())
(layer '())
(drawable '())
)
(set! image (car (gimp-image-new width height type)))
(set! layer (car (gimp-layer-new image width height type name opacity mode)))
(gimp-image-add-layer image layer position)
(set! drawable (car (gimp-image-get-active-drawable image)))
(gimp-context-set-foreground foreground)
(gimp-drawable-fill drawable fill-type)
(gimp-display-new image)
)

)
(script-fu-register "script-fu-poradnik-obraz"
"Poradnik skrypt obrazu"
"Skrypt utworzy nowy obraz w GIMP według naszych ustwien."
"Zbyma"
"Zbyma"
"Grudzień 2010"
""
)
(script-fu-menu-register "script-fu-poradnik-obraz"
                         "<Image>/Script-Fu/Poradnik…" )
```

Powyżej zaznaczamy cały tekst po czym **Ctrl + C**. Teraz klikamy Windows => **Start -=> akcesoria =>** otwieramy **Notatnik** (NotePad) http://www.7tutorials.com/beginners-guide-notepad
W notatniku możemy ustalić rodzaj kodowania przy zapisie:

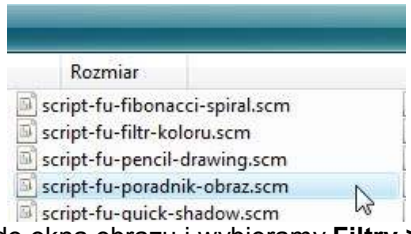


Możemy oczywiście najpierw sprawdzić, czy gdzieś nie brakuje domknięcia nawiasu, lub tych domknięć jest za dużo – wykorzystując do tego zainstalowany edytor, Notepad++ http://notepad-plus-plus.org/ , który prawidłowo rozpoznaje i koloruje składnię języka TinyScheme. Ale trzeba doinstalować plik gimp_scm_cmds_1.txt wg. podanych wskazówek http://registry.gimp.org/node/24223 . Pomaga w przeglądaniu i edytowaniu skryptów GIMP. Przykład:
http://perso.ensg.ign.fr/try/Web_prog/pspad454en/Context/Script-fu.def

```
(define (script-fu-poradnik-obraz)
(let*
(
(width 400)
(height 300)
(type RGB)
(name "ZbymaLayer")
(opacity 100)
(mode NORMAL-MODE)
(position 0)
(foreground '(255 127 000))
(fill-type FOREGROUND-FILL)
(image '())
(layer '())
(drawable '())
)
(set! image (car (gimp-image-new width height type)))
(set! layer (car (gimp-layer-new image width height type name opacity mode)))
(gimp-image-add-layer image layer position)
(set! drawable (car (gimp-image-get-active-drawable image)))
(gimp-context-set-foreground foreground)
(gimp-drawable-fill drawable fill-type)
(gimp-display-new image)
)

)
(script-fu-register "script-fu-poradnik-obraz"
"Poradnik skrypt obrazu"
"Skrypt utworzy nowy obraz w GIMP według naszych ustwien."
"Zbyma"
"Zbyma"
"Grudzień 2010"
""
)
(script-fu-menu-register "script-fu-poradnik-obraz"
"<Image>/Script-Fu/Poradnik..."
)

```

Teraz mamy podświetlane nazwy funkcji TinyScheme

Nasz skrypt możemy zapisać do **naszego** folderu skryptów:
C:\ Użytkownicy\NazwaUżytkownika\.gimp-2.6\scripts
C:\ Documents and Settings \ <Użytkownik> \ .gimp-2.2 \ scripts
a nie do:
C: \ Program Files \ GIMP-2.0 \ share \ gimp \ 2.0 \ scripts , na końcu dodajemy rozszerzenie "**scm**".
Zapiszemy jako: **script-fu-poradnik-obraz.scm**



Następnie przechodzimy w GIMP-ie do okna obrazu i wybieramy **Filtry > Script-Fu > Odśwież Skrypty**, aby sprawdzić, czy mamy poprawnie zainstalowany skrypt.

Jak powyżej zauważymy, każdy skrypt ma następujący podstawowy format:
Zbiór procedur, wykonujących ściśle określone operacje - które obejmują rzeczywiste działania skryptu oraz części kodu w
script-fu-register - rejestracja skryptu do bazy danych proceduralnych Procedural Database (PDB), do których GIMP może uzyskać dostęp, zawiera informacje na temat skryptu, autora i instrukcje, gdzie umieścić skrypt. Może również zawierać wartości danych wprowadzanych przez użytkownika (zostaną opisane szczegółowo poniżej).
Kiedy GIMP czyta skrypt, będzie wykonywał te funkcje - procedury, które skrypt zarejestrował w bazie danych proceduralnych - Procedural Database (PDB).
Wywołanie funkcji **script-fu-register** możemy umieścić gdziekolwiek chcemy w swoim skrypcie, ale zwykle podaje się ją na końcu, po wszystkich innych kodach.

W GIMP od ver. 2.4 script-fu zostały rozrzucone w menu **filtry**. Teraz filtry i Skrypt-Fu są pogrupowane w menu i zorganizowane zgodnie z nowymi kategoriami. Jeśli jakiś plugin i skrypt działa podobnie, są one w podobnym menu.
Oryginalnie nie ma w ogóle menu **script-fu**. Będzie ono widoczne tylko w wypadku jeśli doinstalujemy dodatkowe skrypty, które będą się lokowały w menu **script-fu,** i tylko te dodatkowe skrypty tam będą.
W katalogu GIMP-a, folder **<Użytkownik> \ .gimp-2.6 \ scripts** instalujemy skrypty dostępne dla danego użytkownika. Domyślnie folder ten jest pusty (skrypty dostępne zaraz po instalacji programu GIMP znajdują się w katalogu programu - dostępne dla wszystkich użytkowników).

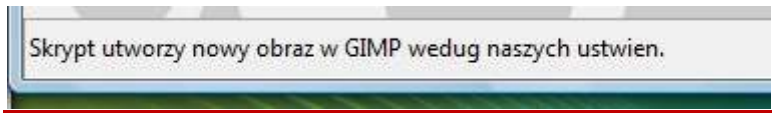
O procedurach wymienionych w powyższym skrypcie już co nieco wiemy, natomiast jak na razie nic nie wiemy na temat jego rejestracji w **script-fu-register,** omówimy więc:

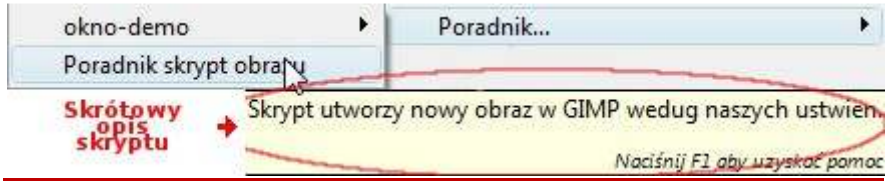
Etapy rejestracji Skryptu

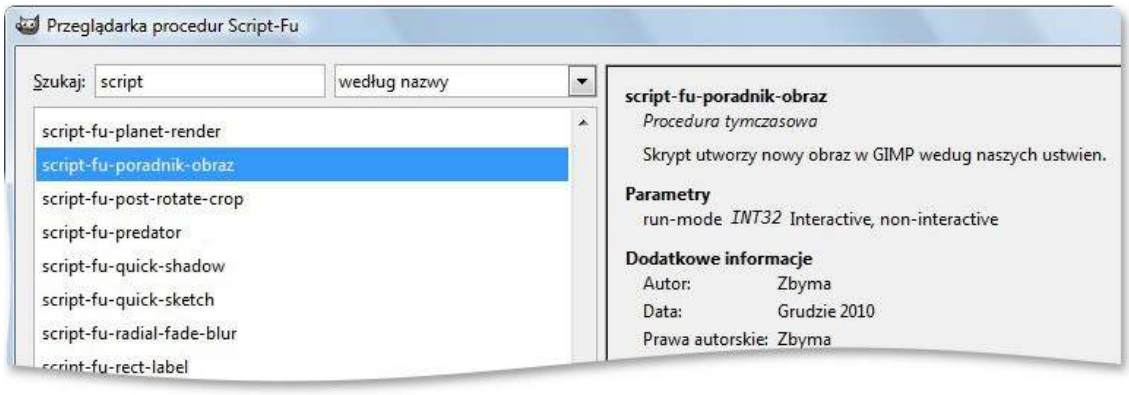
Aby zarejestrować skrypt w GIMP, wywołujemy funkcję `script-fu-register`, w której należy podać min. 7 wymaganych parametrów i ewentualnie dodać do skryptu własne parametry, wraz z opisem i domyślną wartością dla każdego parametru.
Skrypty, które mogą być stosowane do kolorowych obrazów, obrazów w skali szarości i obrazów z paletą kolorów, które w dodatku nie mają regulowanych przez użytkownika parametrów mogą być rejestrowane za pomocą tylko tych 7 poniższych parametrów.

Poniżej w tabeli podano schemat kolejności wymaganych parametrów z opisem.

Numer parametru	Znaczenie
1	*name* - nazwa funkcji, w której jest zapisane ciało skryptu, funkcja jest wywoływana przy uruchomieniu skryptu. Jest to konieczne, ponieważ możemy zdefiniować inne funkcje w tym samym pliku, a GIMP musi wiedzieć, którą z funkcji ma wywołać. Znajdziemy ją w Przeglądarce procedur.
2	*location* – etykieta skryptu w formie, jaka ma się pojawić w menu i opis okna dialogowego
3	*description* – ***skrótowy*** opis skryptu, który pojawia się na pasku stanu w dole okna Edytor obrazów GIMP oraz obok kursora myszki po umieszczeniu go na nazwie skryptu w menu /podmenu. Skrócony opis znajdziemy także w prawej części okna Przeglądarki procedur.
4	*Your name* **-** *nazwisko autora* skryptu
5	***Copyright*** - informacja o prawach autorskich
6	***Copyright*** - informacja o prawach autorskich
7	***Type*** - lista typów obrazów, na których można uruchomić skrypt np. RGB, RGBA, GRAY, GRAYA, INDEXED, INDEXEDA lub dla oznaczenia z kanałem alfa `RGB*; GRAY*; INDEXED*;` i wszystkie typy `"*"`







Uwaga:
Jeśli skrypt ma pracować na wszystkich typach obrazów, nie stosować dwóch cudzysłowów "", stosować tylko gwiazdkę (symbol wieloznacznika) otoczoną cudzysłowami "*" jest wieloznacznikiem rozwijanym do listy nazw wszystkich zbiorów w bieżącym katalogu.
Przy stosowaniu dwóch cudzysłowów "", skrypty działają dobrze, ale jeśli cofnąć operację skryptu po uruchomieniu, nie będzie można wykonać w menu polecenia **Filtry / Powtórz** (Ctrl + F) lub **Filtry / Wyświetl ponownie** (Shift + Ctrl + F), musimy wrócić do dokładnej lokalizacji skryptu w menu, aby go uruchomić ponownie lub Edycja => Ponów operację: (np.: *Script-Fu* => **Fake HDR look**).
Występują jeszcze inne problemy do których może to doprowadzić, więc upewniamy się że umieszczono odpowiedni typ obrazu w cudzysłowie.

Kilka dodatkowych słów dotyczących konwencji nazewnictwa:
Wskazane jest stosować w nazwach skryptów ***name*** małych liter z łącznikami z nazwą funkcji.

```
script-fu-poradnik-obraz
```

Dobra konwencja podawania nazw skryptów w GIMP to **script-fu-abc**, bo wtedy wszystkie pojawią się w bazie danych proceduralnych [Przeglądarka procedur (PDB)], wymienione gdzieś **w liście script-fu**.
Ale nie wszyscy tego przestrzegają. Pomaga to również potem w odróżnienie ich od plugin-ów.
Trzeba również zwracać uwagę aby nadana nazwa była unikalna dla uruchomienia w PP. Jeśli istnieje inna procedura zarejestrowana w PP, która ma taką samą nazwę, jedna z nich nie będzie działać.
Sugeruję aby przed wybraniem ostatecznej nazwy procedury, wpisać ją w PP i sprawdzić, czy nie pojawi się na liście. Jeśli tak, to można np. dodać swoje inicjały na końcu w celu odróżnienia jej od tej drugiej w PP – lub wybierać inną unikalną nazwę. Jeśli jakiś skrypt nie działa, istnieje duże prawdopodobieństwo (z wyjątkiem błędów w skryptcie), że mamy zarejestrowane w PP dwa egzemplarze procedury o tej samej nazwie.
Przy zapisie gotowego skryptu część autorów stosuje **Zapisz jako…** odrzucając część `scripts-fu-`.
Niektórzy z autorów zalecają *bardziej opisowe* nazwy dla parametrów i zmiennych, czyli dodanie prefiksu "**in**" do parametrów, aby można było szybko zobaczyć, że są to wartości przekazywane do skryptu, a nie stworzone w jego obrębie, oraz stosowanie prefiksu "**the**" dla zmiennych zdefiniowanych w skrypcie
(np. skrypty: `selective-coloring.scm ; photo-stack.scm...`)

location – etykieta skryptu w formie np. `"Okno demo"`, można przed pierwszą literą (lub inną dowolną) wprowadzić znak dolnego podkreślenia "_", wtedy ta litera będzie w etykiecie podkreślona.

Teraz jeszcze funkcja **`script-fu-menu-register`** , którą wywołujemy z dwoma parametrami typu string. Pierwszym parametrem jest nazwa funkcji, która wykonuje własny skrypt (jest to ta sama funkcja, która została wprowadzona w skrypcie **script-fu-register**), drugim parametrem jest ścieżka do skryptu w menu:

```
Np.(script-fu-menu-register "script-fu-poradnik-obraz"
                            "<Image>/Script-Fu/Poradnik..."
```

Przy czym oznaczenie z trzema kropkami **Poradnik…** sugeruje, że jest to utworzony podkatalog, w którym może być umieszczone kilka*naście* skryptów tematycznych.

Do rejestracji oprócz powyższych 7 parametrów, są potrzebne dodatkowe parametry *script-fu-register*:
Najpierw specyfikujemy "SF-parametry" dla ustawień nieinteraktywnych

- *param-type* , typ parametru (może być SF-IMAGE, SF-DRAWABLE, SF-COLOR, SF-VALUE, SF-STRING, SF-TOGGLE);
- *deafult value* , wartości domyślne.

SF-IMAGE

Przydatne tylko w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który jest rozwijanym polem wyboru, który pozwala użytkownikowi wybrać otwarty obraz. Używamy by otrzymać ID obrazu.
"Widget label" – etykieta przypisana do pola wyboru,

Uwaga: do etykiet nie dodaje się dwukropka!.
Zastosowanie:
SF-IMAGE "Widget label" 0
Wartość zwracana, gdy skrypt zostanie wywołany odpowiada numerowi ID otwartego obrazu.
ID - liczba całkowita – Integer.
Przykład:
SF-IMAGE "Image" 0 lub " Input Image" 0; lub "inImage" 0

SF-DRAWABLE

Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który jest rozwijanym polem wyboru, który pozwala użytkownikowi wybrać `DRAWABLE` (np. warstwę **Tło**) z otwartego pliku obrazu.
"Widget label" – etykieta przypisana do pola wyboru.
Zastosowanie:
`SF-DRAWABLE "Widget label" 0`
Wartość zwracana, gdy skrypt zostanie wywołany odpowiada numerowi ID wybranego drawable: (obrazu, kanału lub warstwy). ID - liczba całkowita – Integer.
Przykład:
SF-DRAWABLE "Drawable" 0 lub "Input Drawable" 0; lub "The Drawable" 0; "The Layer" 0

SF-COLOR

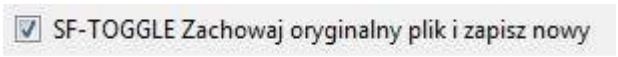
Tworzy to widget sterowania dialogowego, który jest przyciskiem zawierającym podgląd wybranego koloru. Jeśli przycisk zostanie wciśnięty pojawi się wyskakujące okno wyboru koloru.
Akceptuje listę trzech liczb całkowitych, wartości koloru czerwonego, zielonego i niebieskiego lub nazwę koloru w notacji CSS.
Arkusz stylów CSS jest sekwencją znaków z zestawu Universal Character Set (zobacz [ISO10646]). Dla celów transmisji i przechowywania znaki te muszą zostać zakodowane w jakimś kodowaniu znaków, które obsługuje znaki z zestawu US-ASCII (np. UTF-8, ISO 8859-x, SHIFT JIS itp.). Dobry wstęp do zestawów znaków i kodowania znaków znajduje się w specyfikacji języka HTML 4 ([HTML4], rozdział 5). Zobacz również specyfikację XML 1.0 ([XML10], podrozdziały 2.2 i 4.3.3 oraz dodatek F).
lista argumentów przycisku:
`SF-COLOR "label" -` etykieta, tekst opisu przed klawiszem.
`'(red green blue)-` lista trzech wartości dla składników czerwonego, zielonego i niebieskiego.
`lub`
`SF-COLOR "label" "color" gdzie "color" -` nazwa koloru w notacji CSS.
Przykład:
`SF-COLOR "Kolor" '(255 127 000)`

SF-STRING

Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, którym będzie pole z jedną linią tekstu. Wartość zwracana, gdy skrypt zostanie wywołany zwraca łańcuch zawierający wprowadzony tekst.
Zastosowanie:
`SF-STRING "Text" "A single line of text."`
`SF-STRING "Tekst" "Okno demonstracyjne"`

SF-VALUE
Akceptuje cyfry i łańcuchy. Wartość domyślna powinna być w cudzysłowie, nawet jeśli jest to cyfra. Należy pamiętać, że cytowanie musi być dodane do domyślnego tekstu, jest to lepsze wykorzystanie niż SF-STRING.
Przykład:
`SF-VALUE "size" "100"` Każda wartość integer lub zmiennoprzecinkowa.

SF-TOGGLE



Wyświetla pole, aby zaznaczyć/odznaczyć wartość logiczną Prawda lub Fałsz. (TRUE lub FALSE)
Np.:

```
SF-TOGGLE  "Preserve aspect ratio" TRUE lub
SF-TOGGLE "Keep original files and save new files with prefix \' _ \'" TRUE
```

Dla skryptów, wykonujących operacje na obrazie, które można wywołać tylko wtedy, gdy użytkownik otworzy plik obrazu, czyli **- podległych obrazowi**, należy podawać jako pierwsze argumenty zmiennych typu:
SF-IMAGE i **SF-DRAWABLE**.

Dla **SF-IMAGE** i **SF-DRAWABLE**, powinna być przez nas podana wartość **0**;
GIMP wypełni je ID bieżącego obrazu lub warstwy...

Parametry API script-Fu
wszystko na temat okna dialogowego uruchamianego przed wywołaniem skryptu

API - Application Programming Interface, interfejs programowania aplikacji, interfejs programu użytkownika – zbiór gotowych procedur i funkcji umożliwiających komunikację z biblioteką, których używamy pisząc skrypty, GIMP to nam oferuje, nie musimy pisać od podstaw procedur obsługi okien, tylko wywołujemy te procedury.
Przydatne materiały źródłowe:

http://developer.gimp.org/api/2.0/libgimp/index.html
http://doc.gnu-darwin.org/libgimpwidgets/
http://doc.gnu-darwin.org/libgimpbase/
http://gimp-plug-ins.sourceforge.net/doc/libgimp/html/libgimp-gimp.html
http://ftp.sai.msu.su/~megera/gimp/bless/gimp/prb/gimp.prb.0.03.html#toc8

JAK KORZYSTAĆ Z NAZW STAŁYCH W BLOKU REJESTRACJI.

Wykaz typów argumentów, które można wykorzystać w bloku rejestru skryptu Script-Fu, dla trybu interaktywnego (wtedy skrypt jest do użytku w trybie wsadowym, ponieważ wymaga otwarcia okna deklaracji parametrów), gdzie każdy z nich będzie tworzyć Widget jako okno sterowania dialogowego.
Opiszemy dość szczegółowo parametry typów argumentów i ich format, które mogą być użyte w bloku rejestru Script-Fu.

Okno dialogowe wywoływane przy uruchomieniu skryptu.

Funkcja **script-fu-register** akceptuje oprócz siedmiu obowiązkowych parametrów, również dowolną liczbę innych parametrów, określenie wartości których, jest możliwe w wyświetlonym oknie dialogowym przed wykonaniem skryptu.
To okno dialogowe może zawierać, praktycznie dowolną kombinację elementów kontrolnych, w tym np. do wprowadzania tekstu, wartości numerycznych, przyciski wywołujące dalsze specjalizowane okienka dialogowe wyboru czcionki, koloru, warstw (poziomów), suwaki itp. Każdy element sterujący jest opisany trójką parametrów – typem elementu sterującego (chodzi o stałą np. **SF-ADJUSTMENT),** dowolnym łańcuchem, który jest wyświetlany w oknie dialogowym elementu (chodzi o opis etykietę - *label*) i trzecim parametrem, którego znaczenie zmienia się w zależności od tego o jaki element sterujący chodzi – może np. chodzić o implementację koloru, nazwę czcionki, nazwę pędzla, zawartość pola list, itd. Wartości podane przez użytkownika są przekazywane do funkcji przedstawiającej ciało skryptu jako jej parametry – musimy zwrócić uwagę na to, aby ilość parametrów odpowiadała ilości elementów w oknie dialogowym a także na ich prawidłową kolejność.
Nazwy (stałe) wszystkich typów elementów sterujących, odpowiadają typom danych w GIMP, ich opis jest poniżej.

Poniżej pokazano okno demonstracyjne, na bazie którego zaobserwować można zastosowanie wielu typów dostępnych elementów wejściowych. Ten skrypt nic nie robi (wywołuje **Okno-demo**), po wywołaniu z menu Script-Fu pojawi się Okno dialogowe, a w nim są zawarte przykładowe elementy sterujące, które można wykorzystać w GIMP. Akceptuje dokładnie taką ilość parametrów, jaka odpowiada ilości elementów sterujących pokazanych w oknie.
Po wywołaniu tego okna demonstracyjnego staramy się zrozumieć jak działają poszczególne elementy sterujące (widet), które omówiono poniżej. Zobaczymy, że Aktywacja niektórych elementów (np. klikając na nim) wywołuje specjalizowane okna dialogowe GIMP-a, w których łatwo wybrać np. kolor, gradient itd.

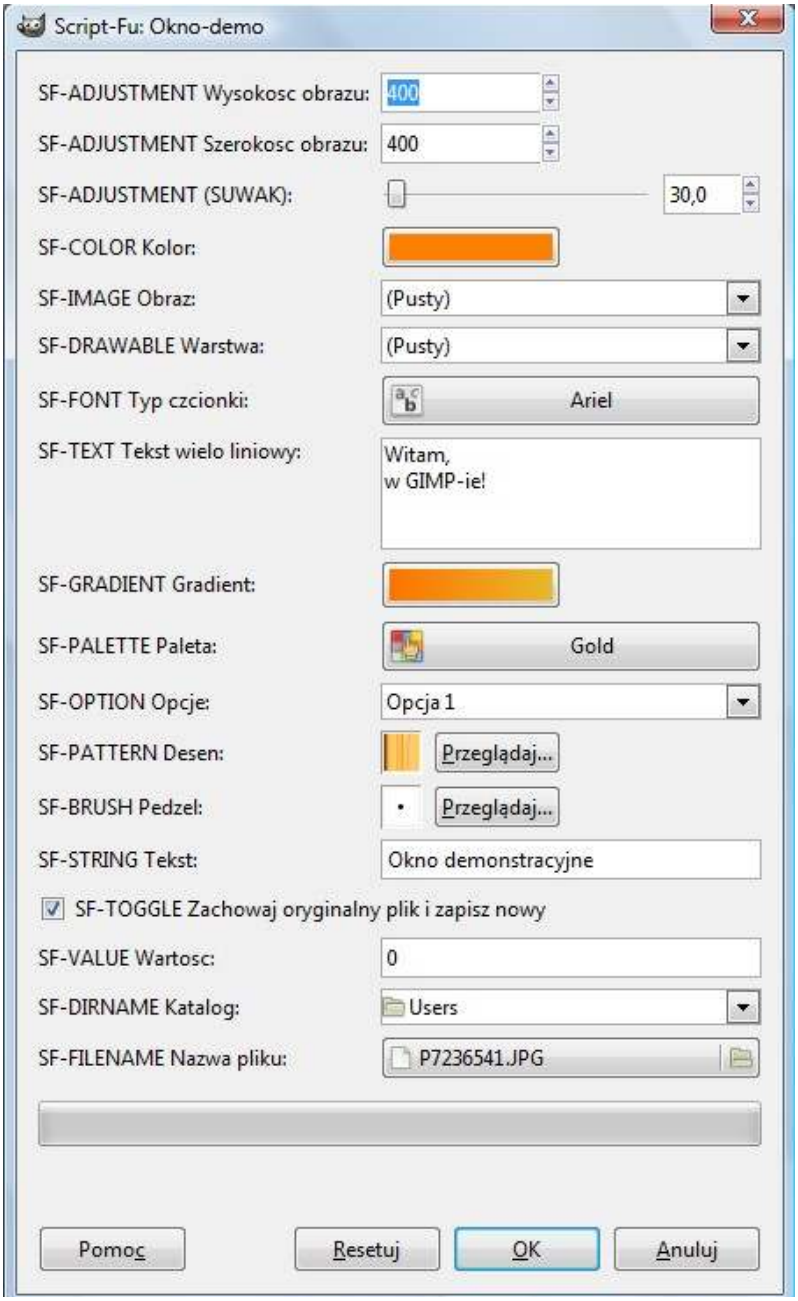
Omówienie okna demonstracyjnego skryptu
Zapoznaj się następujące informacje w skrypcie (Zwrócę uwagę na pewne cechy, które opiszę poniżej):

```
SF-ADJUSTMENT "SF-ADJUSTMENT Wysokosc obrazu"      '(400 50 1000 1 10 0 1)
SF-ADJUSTMENT "SF-ADJUSTMENT Szerokosc obrazu"     '(400 50 1000 1 10 0 1)
SF-ADJUSTMENT "SF-ADJUSTMENT suwak"                '( 30 1 2000 1 10 1 0)
SF-ADJUSTMENT "Cyan-Red color balance"             '(0 -100 100 1 5 0 SF-SLIDER)
[SF-SLIDER zamiast 0 ]
```

Określamy części skryptu, gdzie definiujemy naszą główną funkcją o nazwie "okno-demo" i określamy parametry funkcji, którymi np. będą : `"Wysokosc", "Szerokosc", itd.`
Parametry są "nośnikami" informacji powierzanych im przez użytkownika w oknie dialogowym, które pojawią się po pierwszym uruchomieniu skryptu.
W celu stworzenia okna dialogowego, musimy podpowiedzieć GIMP jakie parametry potrzebujemy oraz zdecydować, jak powinny pojawiać się w oknie dialogowym (kolejność w oknie).
Korzystając z powyższych informacji, stwórzmy demonstracyjne okno dialogowe `okno-demo.scm` – które będzie wyglądać tak:

```
; Przyklad demonstracyjny
; Funkcja musí akceptowac taka ilosc parametrow,
; aby odpowiadala ilosci elementow sterujacych pokazanych
; w oknie dialogowym.
(define (okno-demo
        value
        adj1
        adj2
        image
        drawable
        toggle
        pattern
        string
        font
        color
        option
        gradient)
(gimp-context-push)
(gimp-message "Nic nie robi, tylko pokaz !")
(gimp-context-pop)
(gimp-displays-flush) )
; podstawowe informacje o skrypcie a takze definicje tresci okna dialogowego

(script-fu-register "okno-demo" ;nazwa skryptu zarejestrowana w PDB
                    "Okno-demo" ;nazwa skryptu jaka ma sie pojawic w menu oraz
jako opis Okna dialogowego
                    "Nic nie robi, tylko pokaze wszystkie wzory wejsciowe"
                    "Zbyma"
                    "GPL"
                    "Marzec 2011"
                    ""
                   SF-ADJUSTMENT "SF-ADJUSTMENT Wysokosc obrazu"     '(400 50
1000 1 10 0 1)
                   SF-ADJUSTMENT "SF-ADJUSTMENT Szerokosc obrazu"         '(400
50 1000 1 10 0 1)
                    SF-ADJUSTMENT "SF-ADJUSTMENT (SUWAK)" '( 30 1 2000 1 10 1 0)
                      SF-COLOR      "SF-COLOR Kolor" '(255 127 000)
                      SF-IMAGE      "SF-IMAGE Obraz" 0
                      SF-DRAWABLE   "SF-DRAWABLE Warstwa" 0
                      SF-FONT       "SF-FONT Typ czcionki" "Ariel"
                      SF-TEXT       "SF-TEXT Tekst wielo liniowy" "Witam,\nw GIMP-
ie!"
                      SF-GRADIENT   "SF-GRADIENT Gradient"  "Yellow Orange"
                      SF-PALETTE    "SF-PALETTE Paleta" "Gold"
                      SF-OPTION     "SF-OPTION Opcje" '("Opcja 1" "Opcja 2" "Opcja
3")
                      SF-PATTERN    "SF-PATTERN Desen" "Pine"
                  SF-BRUSH "SF-BRUSH Pedzel" '("Circle (03)" 100 44 0)
                      SF-STRING     "SF-STRING Tekst" "Okno demonstracyjne"
                      SF-TOGGLE     "SF-TOGGLE Zachowaj oryginalny plik i zapisz
nowy" TRUE
                      SF-VALUE      "SF-VALUE Wartosc" "0"
                      SF-DIRNAME    "SF-DIRNAME Katalog" "C:/Users/"
                      SF-FILENAME   "SF-FILENAME Nazwa pliku" "D:/Pictures/Do
prob/P7236541.JPG"
)
; rejestracja skryptu w menu
(script-fu-menu-register "okno-demo"
                         "<Image>/Script-Fu/Poradnik.../okno-demo")
```

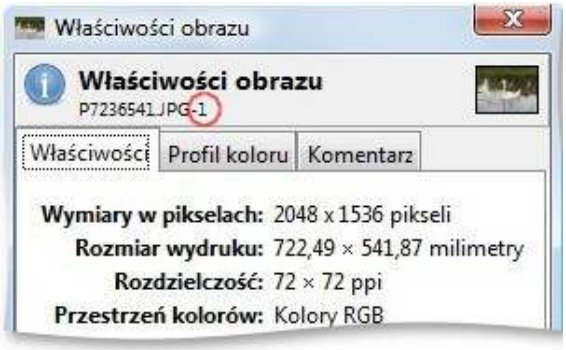
Na podstawie: http://pingus.seul.org/~grumbel/gimp/script-fu/script-fu-tut.html
Otwieramy jakiś obraz, a następnie skrypt:



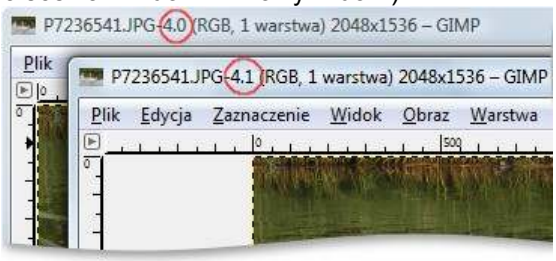
Okno Dialogowe skryptu demonstracyjnego.

Uwaga: w oknie demonstracyjnym celowo pozostawiono w etykietach wyświetlanie nazw stałych SF, które pokazuje, jaki format ma widget.
Jak widać skrypt demonstracyjny wyświetla okno z pytaniem o wiele parametrów, ale nic nie robi, po kliknięciu OK. pojawi się okno z komunikatem:

Uwaga: najprostszym sposobem określenia ID zdjęcia na którym otwarto jakiś skrypt, jest otwarcie okna "Obraz => Właściwości obrazu", w którym znajdziemy numer ID zdjęcia, na końcu nazwy pliku w górnej części okna.



Jeśli spojrzeć na pasek tytułowy otwartego obrazu, zobaczymy nazwę obrazu w postaci pliku. Cyfry po myślniku są cyframi ID obrazu i numerem ID widoku (można mieć więcej niż jeden widok tego samego obrazu, otwartych za pomocą polecenia "Widok => Nowy widok").



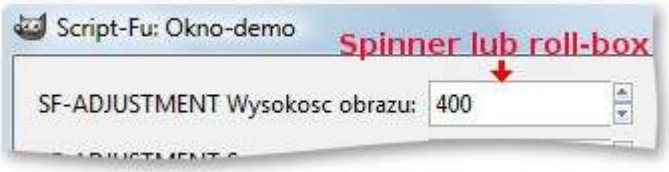
ID widoku możemy zazwyczaj ignorować, ale cyfra ID obraz, jest tą która przechodzi do skryptu.

GIMP wymaga, aby każdy parametr podać w określonym formacie, przykładowo dla "Wysokosc obrazu":

```
SF-ADJUSTMENT       "Wysokosc obrazu"      '(400 50 1000 1 10 0 1)
```

W powyższym kodzie skryptu, GIMP ma użyć kolejno parametrów o nazwie:

SF-ADJUSTMENT
Tworzy widget sterowania dialogowego w formie pola tekstowego lub suwaka:



Widget (podstawowy element GUI np. okno, pole edycji - wyboru, suwak, przycisk.
`Spinner, SpinButton` `lub` **`roll-box`** – pozwala użytkownikowi na wybór wartości z zakresu wartości numerycznych. Składa się z pola tekstowego oraz strzałek w górę i dół z lewej strony. Wartości może mieć regulowaną liczbę miejsc po przecinku, a wielkość kroku jest konfigurowalna. Wybór jednego z przycisków powoduje, zmianę wartość w górę i w dół z zakres możliwych wartości. Pole wpisu może również być edytowane bezpośrednio do określonej wartości.

```
'(value lower upper step_inc page_inc digits type)
```

Dostosowanie parametrów **`SF-ADJUSTMENT`** wymaga 7 wartości argumentów okna.
Wyjaśnię je kolejno poniżej:
lista argumentów Okna (Widget) - podstawowy element GUI.
`"Label"` – Etykieta, czyli tekst pokazany przed: pole edycji, suwak, przycisk
`value` – Domyślna wartość startowa
`lower / upper` – Min-value max-value dolna i górna wartość (zakres regulacji).
`step_inc` – Przyrost / zmniejszenie wartości.
`page_inc` – Przyrost / zmniejszenie wartości za pomocą klawiszy.
`digits` – Cyfry po przecinku (dziesiętne części). [int=0 lub float=1]
`type` – SF-SLIDER lub **0**, SF-SPINNER lub **1**
Przykładowa lista:

```
'(400 50 1000 1 10 0 1)
```

400 – reprezentuje na liście parametrów **`SF-ADJUSTMENT`** domyślną wartością startową. Mówi GIMP, że domyślna wysokość obrazu ma wynosić 400 pikseli. Oczywiście potem możemy zastosować, jakąś inną ale

w granicach podanych poniżej.
50 - reprezentuje najmniejszą wartość, jaką użytkownik może wprowadzić.
1000 - jest to największa wartość, jaką użytkownik może wprowadzić.
1 - jest to przyrost o ile **spinner** przesunie się po każdym kliknięciu. Pisząc skrypt mamy zestaw możliwości, aby przejść w krokach co 1, ale można ustawić na 10, 25, 50, itd. Wybór należy do Nas.
10 - jest to przyrost o ile przesunie się **slider - suwak** z każdym kliknięciem. Suwak jest zawsze przedstawiany jako suwak i spinner - pole tekstowe). Jeśli klikniesz przyciski obok pola tekstowego suwak, będzie się poruszał w odstępach określonych powyżej. Ale spinner tarczy będzie przesuwał się w odstępach zestaw z poprzednią wartość. Przy pisaniu skryptu wartość zostaną użyte według własnego uznania.
0 - wartość ta może być tylko 0 (zero) lub 1 (jeden). Zero wskazuje GIMP, że wartości wprowadzone przez użytkownika mogą być liczbami całkowitymi (cyfry bez miejsc po przecinku). Jedynka wskazuje GIMP, że wartości wpisane przez użytkownika mogą mieć miejsca po przecinku. We moich parametrach, używam tylko liczb całkowitych. Dlaczego? Parametry których używam nie korzystają z wartości miejsc po przecinku. Przykładowo: czy ma sens, określać szerokość obrazu jako 802,5 a wysokości, 625,75. Ale przykładowo, aby określić ustawienie, rozmycia Gaussa, można użyć liczb dziesiętnych. Czyli typ liczb zależy do projektanta skryptu, i dlatego musimy wiedzieć, jakie liczby mogą być użyte.
1 - wartość ta może być tylko **0** (zero lub `SF-SLIDER`) lub **1** (jeden lub `SF-SPINNER`).
Zero wskazuje użycie suwaka, jedynka wskazuje GIMP, że chcemy używać spinner (spinner w literaturze opisowej widet, można spotkać się również z określeniami roll-box lub spinbutton).

SF-FONT

Tworzy widget sterowania dialogowego, jako przycisk z etykietą "..." domyślnie ustalonej czcionki.
Po kliknięciu przycisku pojawi się wyskakujące okno, w którym możemy wybierać dowolne czcinki i każda z cech może być modyfikowana.
Zwraca fontname jako ciąg znaków.
Istnieją dwie nowe procedury ułatwiające używanie zwracanego parametru:

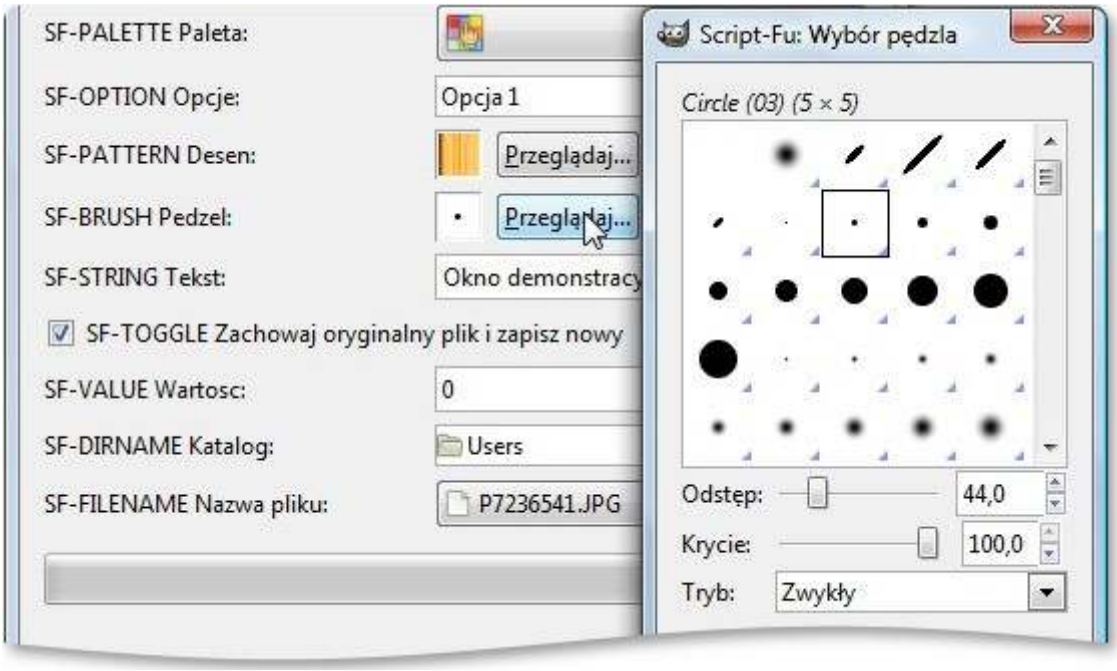
```
(gimp-text-fontname image drawable x-pos y-pos text border antialias size unit
font)
(gimp-text-get-extents-fontname text size unit font)
```

gdzie otrzymana czcionka jest fontname.
Rozmiar podany w fontname jest ignorowany. Jest wykorzystywany tylko w font-selektor.
lista argumentów Spinbutton
`"Label"` – Etykieta, tekst opisu podanego na przycisku.
`"Fontname"` – Nazwa domyślnej czcionki.
Przykład:

```
SF-FONT       "SF-FONT" "Ariel"
```

Pozwala nam wybrać czcionkę.

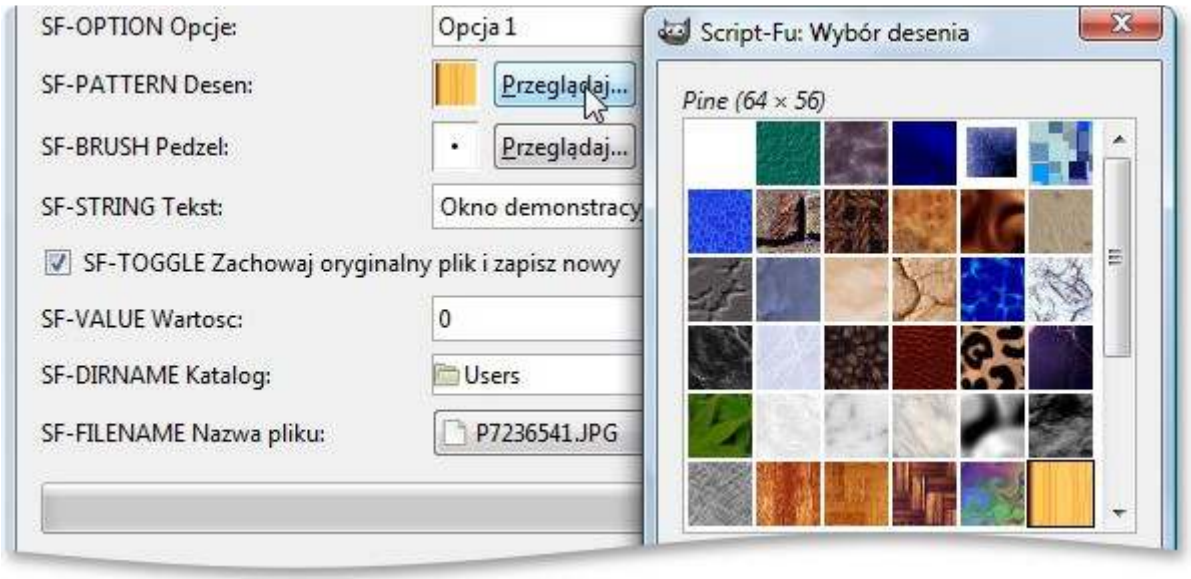
SF-BRUSH



Przydaje się w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który obejmuje obszar podglądu oraz przycisk, po naciśnięciu którego, uzyskamy podgląd w postaci wyskakującego okna, w którym pędzle mogą być wybierane i każda z cech pędzla może być modyfikowana.

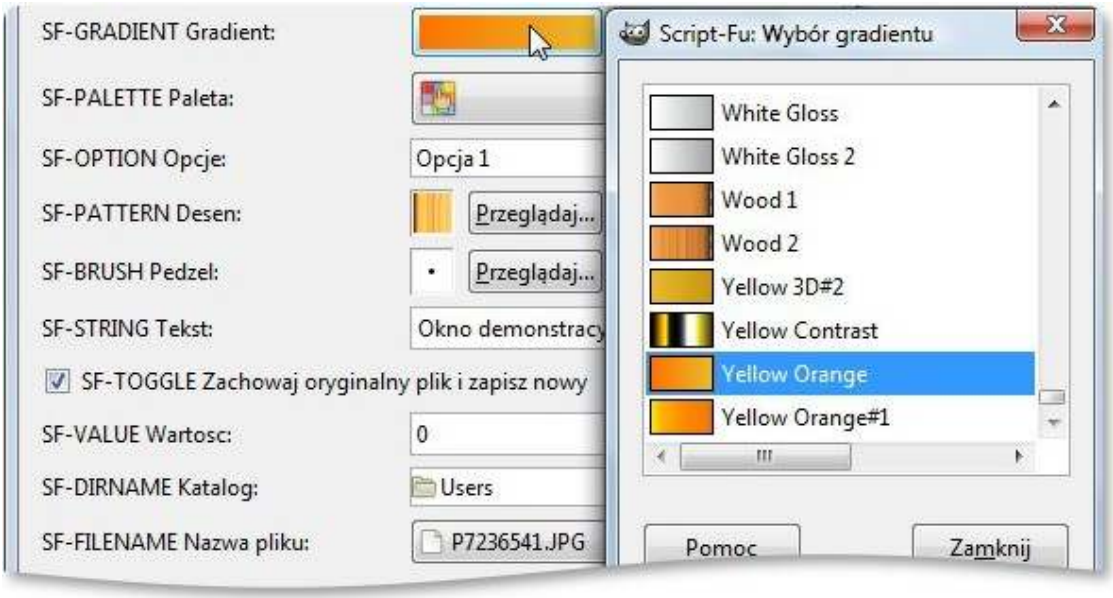
Rzeczywista wartość zwracana, gdy skrypt zostanie wywołany lista składająca się z nazwy pędzla, odstępów, krycia, i trybu pędzla w tych samych jednostkach jaką wartość domyślną podano.
Zastosowanie:
`SF-BRUSH "Brush" '("Circle (03)" 100 44 0)`
`"BRUSH"` – etykieta, czyli tekst, który będzie pokazany przed polem podglądu
Tutaj dialog pędzla pojawi się komunikat z pędzlem domyślnie Circle (03) odstęp 44, krycie 100 i tryb Zwykły Normal (wartość 0). Jeżeli tych opcji nie zmienimy wartość przekazywana do funkcji jako parametr zostanie `'("Circle (03)"100 44 0)`.

SF-PATTERN



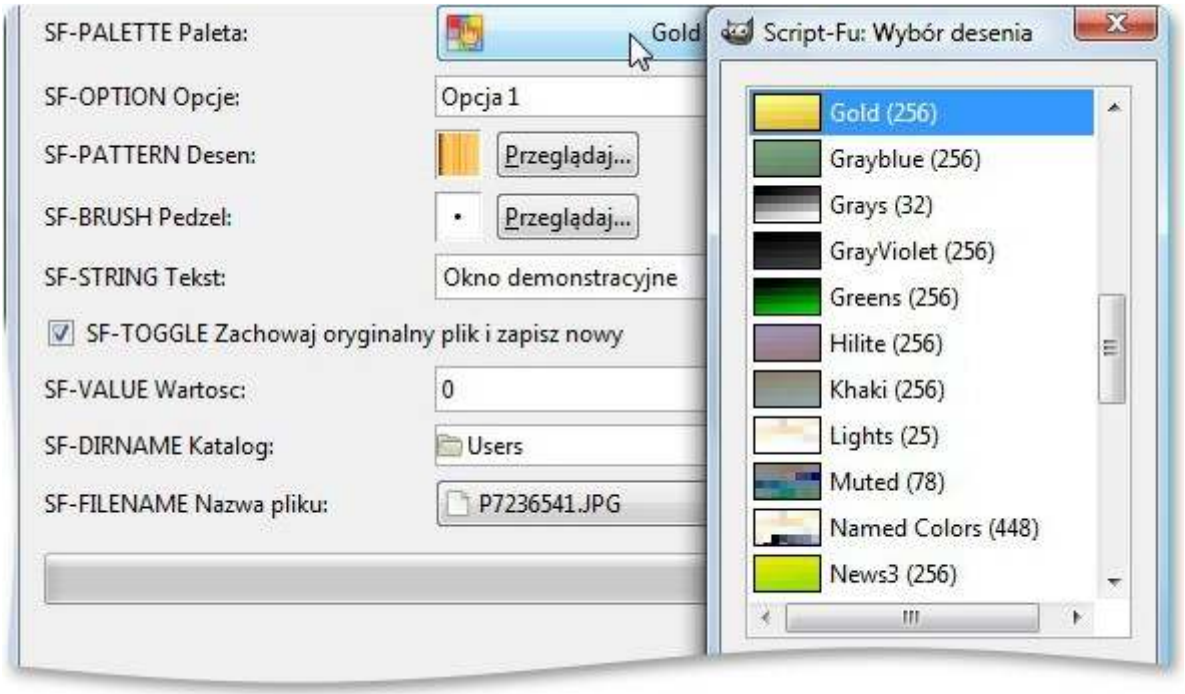
Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który składa się z obszaru podglądu wybranego desenia i przycisku (po kliknięciu którego pojawi się wyskakujące okno "Wybór desenia"). Pozwala nam wybrać inny deseń.
Zastosowanie:
SF-PATTERN "Pattern" "Pine"
"Pattern" - etykieta, czyli tekst, który będzie pokazany przed polem podglądu
"Pine" – deseń wybrany pierwotnie w skrypcie.
Wartość zwracana, gdy skrypt zostanie wywołany łańcuch zawierający wzorzec nazwy. Jeżeli powyższego wyboru nie zmieniono to łańcuch będzie zawierał "Pine".

SF-GRADIENT



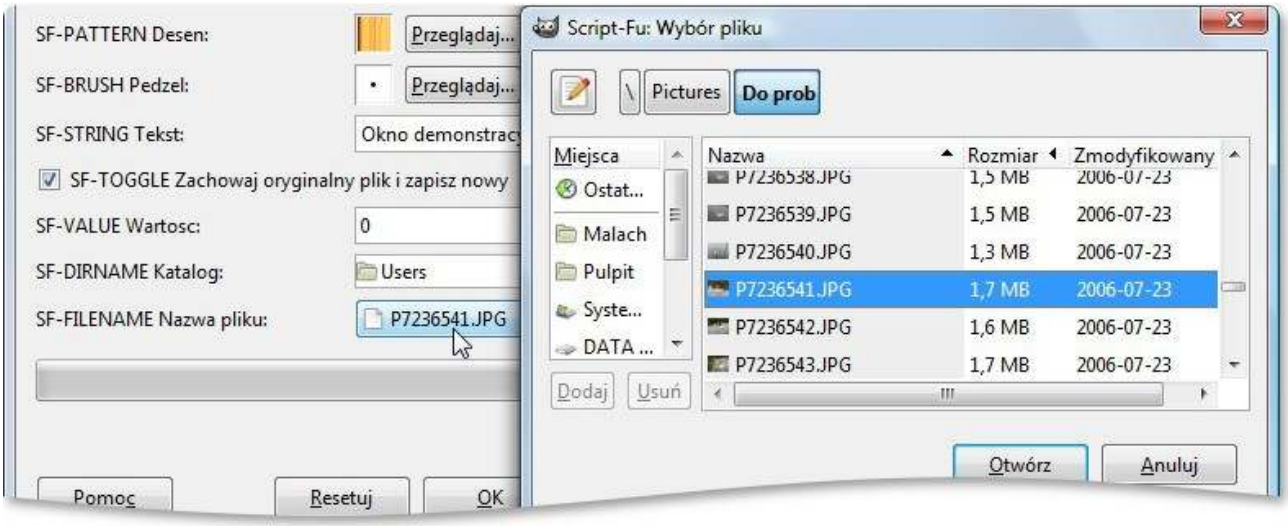
Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który jest przyciskiem zawierającym podgląd wybranego gradientu. Jeśli przycisk zostanie wciśnięty pojawi się wyskakujące okno wyboru gradientu. Pozwala nam wybrać inny gradient.
Zastosowanie:
SF-GRADIENT "Gradient" nazwa gradientu np. "Yellow Orange"
"Gradient" - etykieta, czyli tekst, który będzie pokazany przed polem podglądu
"Yellow Orange" – gradient wybrany pierwotnie w skrypcie.

SF-PALETTE



Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który składa się z przycisku z "..." etykietą (po kliknięciu którego pojawi się wyskakujące okno "Wybór palety"). Pozwala nam wybrać inną paletę.
Zastosowanie:
SF-PALETTE "Palette" "Gold "
"Palette" - etykieta, czyli tekst, który będzie pokazany przed przyciskiem
"Gold" – deseń wybrany pierwotnie w skrypcie.
Wartość zwracana, gdy skrypt zostanie wywołany łańcuch zawierający wzorzec nazwy. Jeżeli powyższego wyboru nie zmieniono łańcuch będzie zawierać "Gold".

SF-FILENAME

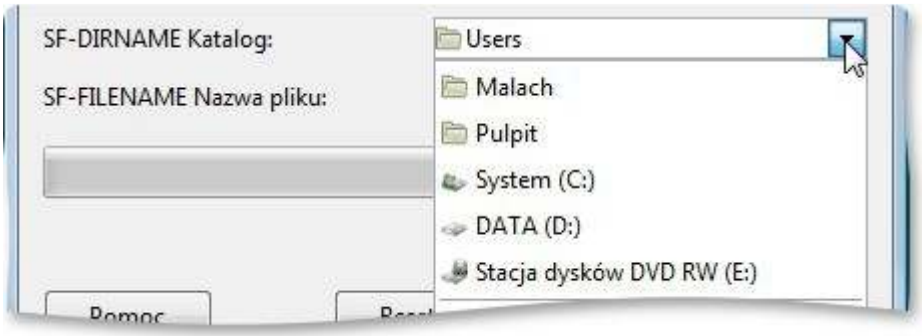


http://library.gnome.org/devel/gtk/2.10/GtkFileChooserButton.html
Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który składa się z przycisku, *zawierającego nazwę pliku*. Jeśli przycisk zostanie naciśnięty pojawi się wyskakujące okno "Wybór pliku", gdzie użytkownik może zmienić plik związany z tym przyciskiem.
Zastosowanie:

SF-FILENAME "Nazwa pliku" "/itd" czyli np.: "D:/Pictures/Do prob/P7236541.JPG"

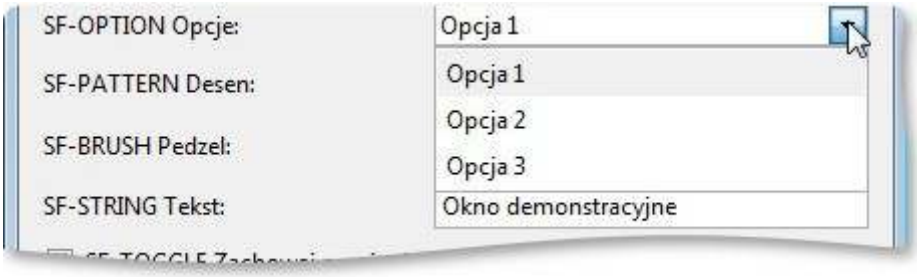
"*.roz" wyrażenie typu dowolna_nazwa_pliku (maska).rozszerzenie
Wartość zwracana, gdy skrypt zostanie wywołany łańcuch zawierający nazwę pliku.
`~/home/katalog1/katalog2/plik.jpg` **(UNIX)**

SF-DIRNAME



Przydatne w trybie interaktywnym. Bardzo podobne do SF-FILENAME, ale tworzony widet, sterowania dialogowego, który jest rozwijanym polem wyboru pokazującym Katalog (*Folder*). Jeśli przycisk zostanie naciśnięty pojawi się możliwość wyboru przez użytkownika innego katalogu.
Zastosowanie:
SF-DIRNAME "Katalog" można stosować sposób wskazania drogi **"C:/"** lub **"C:\\"**
SF-DIRNAME "LABEL" /HOME/KATALOG1/KATALOG2/ ***(UNIX)***
Windows ma inny sposób na wskazanie drogi (używa \ , ukośnik wsteczny – backslash, zamiast / , ukośnika - forward slash)
Wartość zwracana, gdy skrypt zostanie wywołany łańcuch zawierający dirname nazwę katalogu.
Uwagi:
Windows posługuje się konwencją nazewnictwa katalogów **UNC** (ang. *Universal Naming Convention*). Zgodnie z nią do identyfikowania nazwy komputera stosowany jest *podwójny* ukośnik *wsteczny* **** (ang. backslash), natomiast pojedynczy *ukośnik wsteczny* **** wskazuje katalog na dysku tego komputera np. "\\?\C:\katalog1\katalog2" co zostanie zinterpretowane jako "C:\katalog1\katalog2".

SF-OPTION



Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, który jest rozwijanym polem wyboru pokazującym wszystkie opcje, które były przekazane w postaci listy. Pierwsza opcja jest wyborem domyślnym. Pozwala nam wybrać element z listy.
Zastosowanie:
SF-OPTION "Orientation" '("Horizontal" "Vertical") lub
SF-OPTION "Size" '("Original Size" "Scale" "Crop")
Wartość zwracana, gdy skrypt zostanie wywołany to numer wybranej opcji, opcja pierwsza jest liczona jako **0**.

SF-ENUM
Przydatne w trybie interaktywnym. Tworzy to widget sterowania dialogowego, którym jest pole wyboru, wewnątrz którego zawierają się linijki z określonym tekstem do wyboru. W polu pokazują się wszystkie wartości enum dla danego typu enum. Musi to być nazwa zarejestrowanego enum (na przodzie bez prefix "GIMP"). Drugi parametr określa wartość domyślną (przy użyciu wartości enum nicku).
Zastosowanie:
`SF-ENUM "Interpolation" '("InterpolationType" "linear")`
`SF-ENUM "Interpolation (Scaling)" '("InterpolationType" "lanczos")`
Wartość zwracana, gdy skrypt zostanie wywołany odpowiada wartości wybranego enum.

SF-TEXT

Przydatne w trybie interaktywnym. Bardzo podobny do SF-STRING, ale w polu można tworzyć więcej niż jeden wiersz tekstu. Wewnątrz łańcucha na końcu linii (wiersza), dodajemy znak nowej linii (**\n**). Można dodać znak np. podwójnego odstępu (nowej linii) **\n\n (*escape* ucieczka).**

Jeśli za wstecznym ukośnikiem znajdującym się w łańcuchu znajduje się znak nowego wiersza, jest on ignorowany (tzn. zostaje uznane, że łańcuch nie zawiera ani ukośnika, ani znaku nowego wiersza).
Zastosowanie:
SF-TEXT "Multi-line text" "To jest tekst pierwszego wiersza 1 **\n**To jest tekst drugiego wiersza 2."
Wartość zwracana, gdy skrypt zostanie wywołany, łańcuch zawierający wszystkie wprowadzone teksty.

Jeśli ktoś chciałby spojrzeć na materiał źródłowy czyli skrypt opracowany przez "Spencer Kimball, Sven Neumann" w latach "1996, 1998" polecam do testów w pełni działający "script-fu-test-sphere":
http://www.koders.com/scheme/fid49E4B2301F0767B8D6E0958037D9A110EC4E84F8.aspx lub
http://fencepost.gimpdome.com/Images/Tutorials/GIMPScripting103a/SupportingDocuments/test-sphere.scm
Trzeba tylko w GIMP ver. 2.4 lub 2.6 zmienić:
(script-fu-register "script-fu-test-sphere"

```
"<Toolbox>/Xtns/Script-Fu/Test/Sphere..."
                    na
"<Image>/Script-Fu/Test/Sphere…"
```

Dobre rady, zawsze należy pamiętać o dodaniu do skryptu procedur:

Możemy nasz skrypt wskazać GIMP-owi do pracy w "statusie pracy tymczasowej", która pozwala na uruchamianie skryptu, zmienianie różnych ustawień, gdy skrypt jest uruchomiony, a następnie ustawić je z powrotem, takie jakie były przed uruchomieniem skryptu. Nie jest to funkcja wymagana, ale jest to miła rzecz. Aby to działało, musimy wprowadzić dwa polecenia, które działają w tandemie ze sobą, a mianowicie:
`(gimp-context-push) - Inicjuje tymczasowy stan pracy skryptu.`
`(gimp-context-pop) - Wyłącza stan tymczasowy i resetuje poprzednie ustawienia użytkownika.`
Możemy również stosować dwa polecenia, które działają w tandemie: **`(gimp-image-undo-group-start)`** jako początek procedury, która pozwoli nam anulować wszystkie działania naszego skryptu w GIMP-ie jako jedna operacja, czyli anulować cały skrypt, a nie tylko ostatnie działanie skryptu.
Musimy pamiętać o dodaniu funkcji zakończenia wywołanej procedury **`(gimp-image-undo-grupa-end)`.**
Ostateczny cel zostanie osiągnięty przy użyciu kombinacji klawiszy (Ctrl + Z) => lub **Cofnij**.
Jest to właściwe w przypadku, jeżeli przez użycie skryptu obraz został już dopracowany lub skrypt powinien być uruchamiany z innymi parametrami.
W trakcie testowania skryptu, można stosować czasowe wyłączanie powyższych dwóch procedur.
Wystarczy wtedy umieścić przed procedurą znak komentarza czyli średnik **";"** na czas trwania testów.
Możemy wtedy korzystać z historii wszystkich cofnięć.

Polecenie **`(gimp-displays-flush)`** stosujemy, aby odświeżyć przetwarzany obraz na ekranie.
Ta procedura ta ma zastosowanie tylko do skryptów z otwartymi komponentami zdjęcia.
Dla skryptów, które generują nowy obraz, powinna być stosowane procedura **`(gimp-display-new)`.**
`Interactive 1` lub `TRUE`,
`Non-interactive 0` lub `FALSE`

Stworzymy nasz pierwszy skrypt typu Image-dependent

Spróbujemy pokazać jak tworzymy taki skrypt z wszystkich podanych powyżej informacji, jednak teraz wyjaśnienia będą głównie w formie komentarzy.

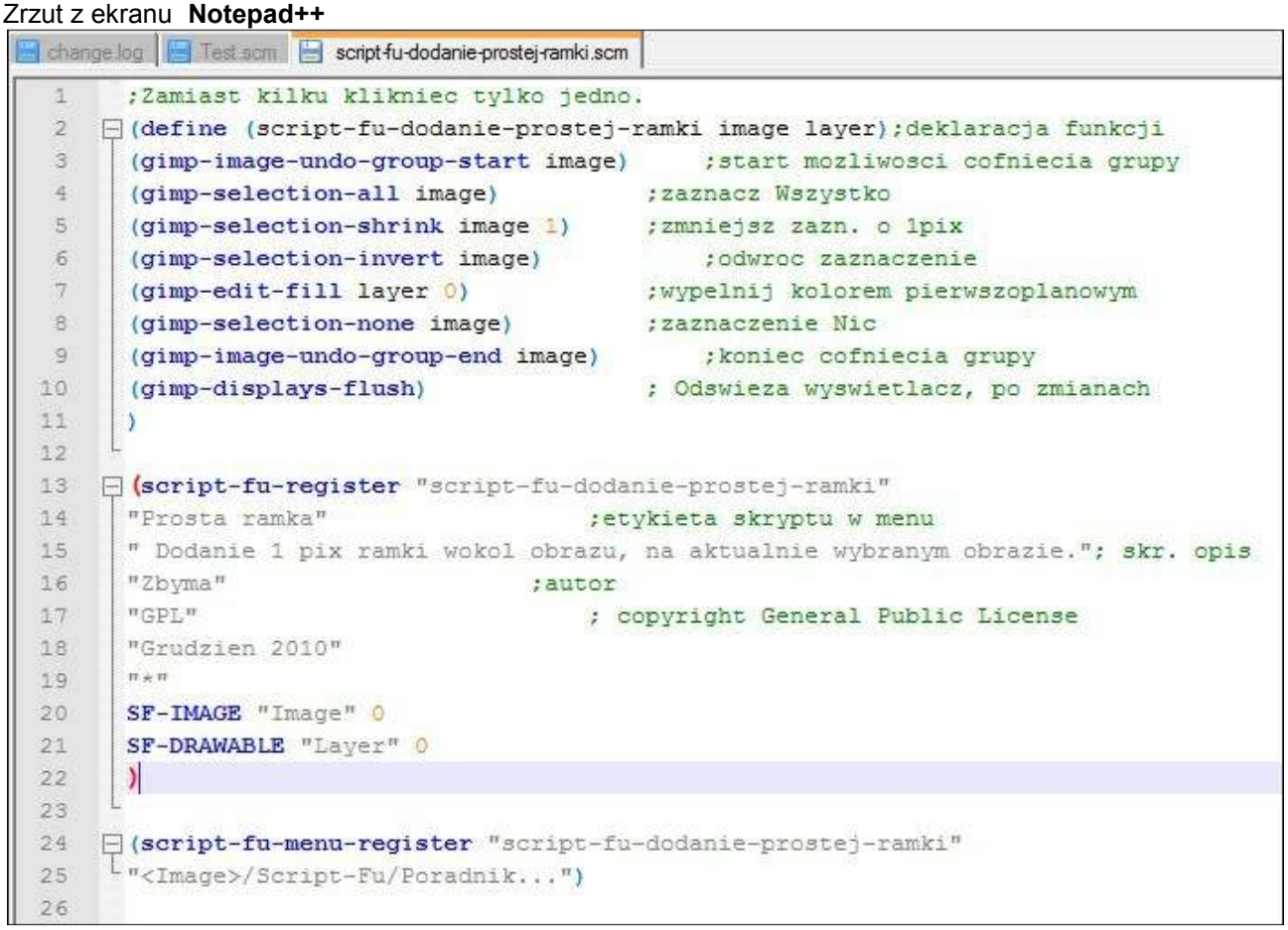
Przypomnienie:
Dla skryptów, wykonujących operacje na obrazie, które można wywołać tylko wtedy, gdy użytkownik otworzy plik obrazu, czyli **- podległych obrazowi**, w **script-fu-register** należy podać dwa najważniejsze argumenty zmiennych typu:

SF-IMAGE i **SF-DRAWABLE**.

Dla **SF-IMAGE** i **SF-DRAWABLE**, powinna być przez nas podana wartość **0**;
GIMP wypełni je ID bieżącego obrazu lub warstwy...

Gdy wiemy co, chcemy uzyskać, możemy zacząć pisanie skryptu.
Jedyne, co teraz będzie inaczej niż wcześniej, to że teraz będziemy pracować przede wszystkim w edytorze, a nie z Konsoli Script-Fu. Może to być **Notepad++** http://notepad-plus-plus.org/ lub **SciTE** (proste, lecz funkcjonalne edytory tekstowe), które prawidłowo rozpoznają i kolorują składnię języka TinyScheme.
Otwieram Notepad++ i zaczynam pisać skrypt umożliwiający dodanie do otwartego obrazu prostej jedno pikselowej ramki, co zmniejsza ilość potrzebnych operacji:

Zrzut z ekranu **Notepad++**



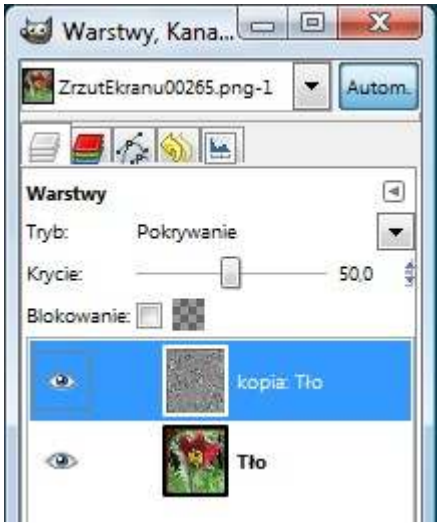
Skrypt zapisujemy jako `script-fu-dodanie-prostej-ramki.scm`.

```
;Zamiast kilku klikniec tylko jedno.
(define (script-fu-dodanie-prostej-ramki image layer);deklaracja funkcji
(gimp-image-undo-group-start image)       ;start mozliwosci cofniecia grupy
(gimp-selection-all image)                ;zaznacz Wszystko
(gimp-selection-shrink image 1)           ;zmniejsz zazn. o 1pix
(gimp-selection-invert image)             ;odwroc zaznaczenie
(gimp-edit-fill layer 0)                  ;wypelnij kolorem pierwszoplanowym
(gimp-selection-none image)               ;zaznaczenie Nic
(gimp-image-undo-group-end image)         ;koniec cofniecia grupy
(gimp-displays-flush)                     ; Odswieza wyswietlacz, po zmianach
)

(script-fu-register "script-fu-dodanie-prostej-ramki"
"Prosta ramka"                            ;etykieta skryptu w menu
" Dodanie 1 pix ramki wokol obrazu, na aktualnie wybranym obrazie."; skr. opis
"Zbyma"                                   ;autor
"GPL"                                     ; copyright General Public License
"Grudzien 2010"
"*"
SF-IMAGE "Image" 0
SF-DRAWABLE "Layer" 0
)

(script-fu-menu-register "script-fu-dodanie-prostej-ramki"
"<Image>/Script-Fu/Poradnik...")
```

Kiedyś napisałem poradnik
Poradnik - Zwiększamy ostrość zdjęcia z GIMP-ie
Po czym na jego podstawie napisałem dla siebie skrypt, który również można zastosować (proszę jak poprzednio prześledzić komentarze):

```
(define (script-fu-high-pass inImage
                                    inLayer)
; Funkcja musí akceptowac taka ilosc parametrow, aby
; odpowiadała ilosci elementow sterujacych pokazywanych
; w oknie dialogowym.

(gimp-image-undo-group-start inImage); start możliwości cofnięcia grupy
    ( let* (
        (new-layer (car (gimp-layer-copy inLayer TRUE)))
            (new-layer-2 (car (gimp-layer-copy inLayer TRUE)))
    )
; Tworzymy dwie warstwy z których stworzymy dalej filtr górnoprzepustowy
      (gimp-image-add-layer inImage new-layer -1)
      (gimp-image-add-layer inImage new-layer-2 -1)
; Zastosowanie na warstwie new-layer-2 plug-inu Rozmycie Gaussa typ RLE
; o wartości promienia 8 pix
(plug-in-gauss
        RUN-NONINTERACTIVE    ; można podać 0 lub FALSE
        inImage
        new-layer-2
        8 ; promień rozmycia poziomo
        8 ; promień rozmycia pionowo
      1                       ; typ rozmycia RLE można zmienić na 0 czyli IIR
oraz promienie rozmycia powiększyć o 10
); można zmienić na INTERACTIVE i dodać w script-fu-register SF-ADJUSTMENT
; zmiany promienia rozmycia.

  (gimp-invert new-layer-2) ; inwersja warstwy

  (gimp-layer-set-opacity new-layer-2 50); zmiana krycie warstwy na 50%
; obraz staje się szary
; teraz połączenie górnych warstw
(let* (
(SharpLayer (car (gimp-image-merge-down inImage new-layer-2 CLIP-TO-IMAGE)))
      )
      (gimp-levels SharpLayer HISTOGRAM-VALUE 100 150 1.0 0 255); modyfikacja
poziomów kolorów
      (gimp-layer-set-mode SharpLayer OVERLAY-MODE); tryb miesznia Pokrywanie
(5),
      ; można zmienić na HARDLIGHT-MODE (18)lub SOFTLIGHT-MODE (19) script-fu-
register
      ; o kolejny SF-ADJUSTMENT, ale szybko można to zrobić w oknie Warstwy,
kanały, sceżki
      (gimp-layer-set-opacity SharpLayer 50); początkowa wartość Krycie
)
)
(gimp-image-undo-group-end inImage); koniec cofnięcia grupy

  (gimp-displays-flush); odświerzanie ekranu po zmianach
  )
(script-fu-register "script-fu-high-pass"
                    "Poradnik High Pass" ;etykieta skryptu w menu
                    "High Pass a Layer"        ; skrótowy opis
                    "Zbyma"                          ; autor
                    "GPL"                            ; licencja
                    "03-07-2009"               ; data utworzenia lub ostatniej
modyfikacji
                    "RGB*, GRAY*"              ; typ obrazów na których pracuje
                    SF-IMAGE    "Image"         0
                    SF-DRAWABLE "Layer"        0
                  )
(script-fu-menu-register "script-fu-high-pass"
                         "<Image>/Script-Fu/Poradnik...")
```

```
(define (script-fu-high-pass inImage
                       inLayer)
; Funkcja musi akceptowac taka ilosc parametrow, aby
; odpowiadała ilosci elementow sterujacych pokazywanych
; w oknie dialogowym.

(gimp-image-undo-group-start inImage); start możliwości cofnięcia grupy
    ( let* (
        (new-layer (car (gimp-layer-copy inLayer TRUE)))
        (new-layer-2 (car (gimp-layer-copy inLayer TRUE)))
    )
; Tworzymy dwie warstwy z których stworzymy dalej filtr górnoprzepustowy
    (gimp-image-add-layer inImage new-layer -1)
    (gimp-image-add-layer inImage new-layer-2 -1)
; Zastosowanie na warstwie new-layer-2 plug-inu Rozmycie Gaussa typ RLE
; o wartości promienia 8 pix
(plug-in-gauss
        RUN-NONINTERACTIVE  ; można podać 0 lub FALSE
        inImage
        new-layer-2
        8 ; promień rozmycia poziomo
        8 ; promień rozmycia pionowo
    1               ; typ rozmycia RLE można zmienić na 0 czyli IIR oraz promienie rozmycia powiększyć o 10
); można zmienić na INTERACTIVE i dodać w script-fu-register SF-ADJUSTMENT
; zmiany promienia rozmycia.

  (gimp-invert new-layer-2) ; inwersja warstwy

  (gimp-layer-set-opacity new-layer-2 50); zmiana krycia warstwy na 50%
; obraz staje się szary
; teraz połączenia górnych warstw
(let*   (
(SharpLayer (car (gimp-image-merge-down inImage new-layer-2 CLIP-TO-IMAGE)))
    )
    (gimp-levels SharpLayer HISTOGRAM-VALUE 100 150 1.0 0 255); modyfikacja poziomów kolorów
    (gimp-layer-set-mode SharpLayer OVERLAY-MODE); tryb mieszania Pokrywanie (5),
    ; można zmienić na HARDLIGHT-MODE (18)lub SOFTLIGHT-MODE (19) script-fu-register
    ; o kolejny SF-ADJUSTMENT, ale szybko można to zrobić w oknie Warstwy, kanały, ścieżki
    (gimp-layer-set-opacity SharpLayer 50); początkowa wartość Krycia

)
)

(gimp-image-undo-group-end inImage); koniec cofnięcia grupy

  (gimp-displays-flush); odświerzanie ekranu po zmianach
  )

(script-fu-register "script-fu-high-pass"
                    "Poradnik High Pass"    ;etykieta skryptu w menu
                    "High Pass a Layer"     ; skrótowy opis
                    "Zbyna"                 ; autor
                    "GPL"                   ; licencja
                    "03-07-2008"            ; data utworzenia lub ostatniej modyfikacji
                    "RGB*, GRAY*"           ; typ obrazów na których pracuje
                    SF-IMAGE    "Image"         0
                    SF-DRAWABLE "Layer"         0
                )
(script-fu-menu-register "script-fu-high-pass"
                         "<Image>/Script-Fu/Poradnik...")
```



Kilka uwag:
Tryb mieszania warstw Pokrywanie stosujemy dla bardziej subtelnego ostrzenia, stosując "Twarde światło" uzyskamy mocniejszy efekt.
Mamy jeszcze procedury:

```
(plug-in-gauss-iir  x x x x  x)
(plug-in-gauss-iir2 x x x x  x)
(plug-in-gauss-rle  x x x x  x)
(plug-in-gauss-rle2 x x x x  x)
```

Można zastosować inną metodę realizacji skryptu z wykorzystaniem krzywej utworzonej procedurą :
Najpierw =>

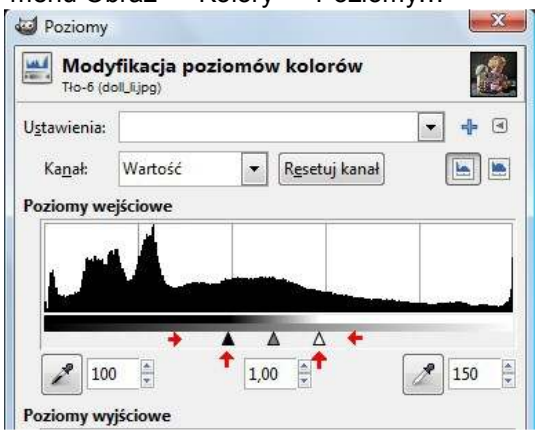
```
; zwiększenie kontrastu poprzez przeciągnięcie o tą samą wartość na niskim i wysokim końcu
```

Parametry

drawable	*DRAWABLE*	The drawable
channel	*INT32*	The channel to modify { HISTOGRAM-VALUE (0), HISTOGRAM-RED (1), HISTOGRAM-GREEN (2), HISTOGRAM-BLUE (3), HISTOGRAM-ALPHA (4), HISTOGRAM-RGB (5) }
low-input	*INT32*	Intensity of lowest input (0 <= low-input <= 255)
high-input	*INT32*	Intensity of highest input (0 <= high-input <= 255)
gamma	*FLOAT*	Gamma correction factor (0,1 <= gamma <= 10)
low-output	*INT32*	Intensity of lowest output (0 <= low-output <= 255)
high-output	*INT32*	Intensity of highest output (0 <= high-output <= 255)

```
(gimp-levels SharpLayer HISTOGRAM-VALUE 100 150 1 0 255)lub
```

Procedura `gimp-leveles` to w menu Obraz → Kolory → Poziomy…



lub zastosować:

Parametry

drawable	*DRAWABLE*	The drawable
channel	*INT32*	The channel to modify { HISTOGRAM-VALUE (0), HISTOGRAM-RED (1), HISTOGRAM-GREEN (2), HISTOGRAM-BLUE (3), HISTOGRAM-ALPHA (4), HISTOGRAM-RGB (5) }
num-points	*INT32*	The number of values in the control point array (4 <= num-points <= 34)
control-pts	*INT8ARRAY*	The spline control points: { cp1.x, cp1.y, cp2.x, cp2.y, ... }

```
(gimp-curves-spline SharpLayer HISTOGRAM-VALUE 8 #(0 0 100 60 150 220 255 255))
```

Jak już podano, wektory piszemy przy użyciu notacji procedury `#(elem ...)`, specjalny znak **#** poprzedza zwracany wector zawarty w nawiasach, mamy określone 8 współrzędnych punktów na krzywej (wskazanie punktu początkowego, punktów przegięcie i punktu końcowego, *spline* - polska nazwa to krzywa sklejana, krzywa interpolowana funkcjami sklejanymi).
Procedura `gimp-curves-spline` to w menu Obraz → Kolory → Krzywe…

W tym miejscu warto przypomnieć, że rozpoczynamy od zastosowania filtru rozmywania - Blur.
Jest to filtr przepuszczający tylko niskie częstotliwości „Low Pass”, który ze zdjęcia filtruje, czyli usuwa składowe wyższych częstotliwości w postaci drobnych **detali.**
(Gdy stosujemy inwersję otrzymamy dopiero jego odwrotność, czyli filtr górno przepustowy.)
Tak więc od skuteczności odfiltrowania drobnych detali zależy skuteczność filtru high-pass dlatego warto może skorzystać z możliwości utworzenia regulacji promienia rozmycia:

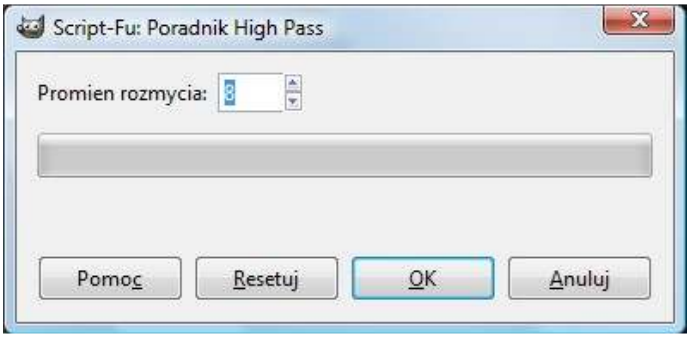
```
(plug-in-gauss 1 inImage new-layer-2 inBlur inBlur 1)
```

1 – wskazuje że skrypt będzie uruchamiany w trybie interaktywnym
Procedura `plug-in-gauss` to w menu Obraz → Filtry → Rozmycie → Rozmycie Gaussa…

Oczywiście wtedy należy dodać w definicji funkcji `define` parametr `inBlur,` a w `script-fu-register`:

```
SF-ADJUSTMENT   "Promien rozmycia" '(8 1 80 1 1 0 1)
```

Ponieważ teraz będziemy pracować w trybie `RUN-INTERACTIVE` (**1**) pojawi się okno wywołania które będzie wyglądać:



Uwaga:
Pasek postępu nie wnosi wartości funkcjonalnej, ale miło wiedzieć, jako użytkownik, jak daleko skrypt został już uruchomiony.
Więc jest to mini-rozdział o pasku postępu, czasem w dużych skryptach warto wiedzieć, czy ten pasek na pewno jest w ruchu, czy też skrypt się zawiesił.
GIMP oferuje dziewięć funkcji (szukaj "gimp-progress") dotyczących zmiany belki postępu, ale wiele z nich jest tylko do plug-inów, a nie tylko dla skryptów.
My jesteśmy zainteresowani tylko dwoma z nich:
(`gimp-progress-set-text`) i (`gimp-progress-update`)
Pierwsza funkcja pisze tekstowy pasek postępu, więc użytkownik można wiedzieć, co się teraz dzieje.
Druga funkcja wskazuje postęp belki. Mamy możliwość wpisać wartości procent postępu, które muszą być zawarte pomiędzy 0.0 i 1.0.
Po każdym kroku procedury wpisujemy przykładowo:
Krok 0

```
(gimp-progress-update 0.0) ;progress bar
```

Krok 1

```
(gimp-progress-update 0.1) ;progress bar
```

Krok 2

```
(gimp-progress-update 0.2) ;progress bar
```

itd… jako przykład `gimp_diving.scm`
Oczywiście można pominąć pasek jeśli Nasz skrypt zajmuje tylko kilka sekund, aby go uruchomić, ale w większych skryptach, dla użytkownika z pewnością nie jest złe, kiedy widzi, jak daleko jest już skrypt.
A więc możemy ustalić, jako autor czy pasek jest w ruchu i na jaki procent się przesunie.
Jak akcja jest wykonywana, zaktualizować np. na pasku postępu (gimp-progress-update 0.5) 50%.

Patrząc na to co podałem powyżej pomyślałem, że może trzeba przyjąć inną koncepcję wyjaśnienia kolejnych kroków tworzenia skryptu.
Zakładam, że chcemy stworzyć skrypt na podstawie kolejnych kroków koniecznych do wykonania w GIMP.

Po otwarciu jakiegoś zdjęcia chcemy kolejno wykonać następujące operacje:

- automatyczne **Przycięcie** obrazu
- **Przeskalowanie obrazu -** proporcjonalne zmniejszenie wymiarów obrazu do szerokości 500 pikseli
- **Spłaszczenie** obrazu,
- Dodanie **czarnego** 2 pikselowego **obramowania** czyli:
  - zaznaczenie obrazu,
  - zmniejszenie zaznaczenia o 2 piksele,
  - inwersja zaznaczenia
  - wypełnienie zaznaczenia czarnym kolorem,
  - rezygnacja z zaznaczenia,
- przejście na **kolor indeksowany** obrazu (32 kolorów)
- zmiana kolorów indeksowanych na **Paletę kolorów**

Opis kolejnych kroków

Aktualnie otwarty obraz jest przekazywany jako parametr do skryptu.

Krok 0.
Ustawienie koloru pierwszego planu i tła,
Chcąc wykonać to ustawienie klikamy w:

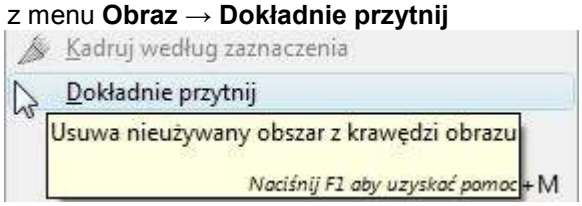
Kolor pierwszoplanowy / tło

Natomiast w skrypcie do tego celu prowadzą dwie procedury

```
(Gimp-context-set- foreground) '(0 0 0))
(Gimp-context-set-background '(255 255 255))
```

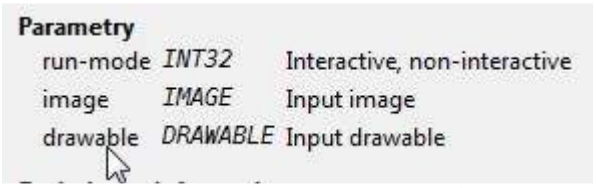
- Kolor pierwszoplanowy czarny (czerwony = 0, zielony = 0, niebieski = 0)
- Tło białe (czerwony = 255, zielny = 255 niebieski = 255)

Krok 1.
Otwarty obraz chcemy **Dokładnie przyciąć**, aby usunąć nieużywane obszary z krawędzi obrazu.
Polecenie to jest równoważne z menu **Obraz** → **Dokładnie przytnij**



Dokładnie przytnij

W skrypcie do tego celu wykorzystujemy wtyczkę `plug-in-autocrop` w której musimy określić 3 parametry:



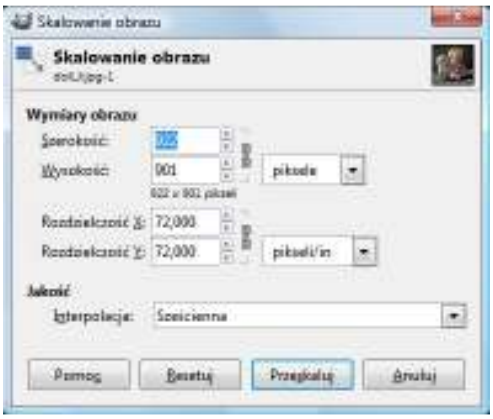

```
(plug-in-autocrop 0 image (car (gimp-image-get-active-drawable image)))
```

Kolejne parametry to:

- `0` : skrypt będziemy uruchomić w trybie NON-INTERACTIVE, brak widgetu zmiany parametrów
- **`image`** : nasz obraz, na którym będą wykonywane kolejne operacje,

- **(gimp-image-get-active-drawable image)**:
  Procedura ta zwraca identyfikator aktywnego obrazu drawable. Może to być warstwa, kanał, lub maski warstwy. Aktywny drawable jest określony przez aktywny kanał obrazu, jeśli to jest -1, a następnie przez warstwę aktywnego obrazu.
  Polecenia `gimp-image-get-active-drawable` zwraca listę, aby pobrać pierwszy element tej listy `car`.

Krok 2.
Przeskalowanie obrazu
Zmniejszenie obrazu, aby określić nową szerokość i wysokość obrazu, normalnie po prostu wywołujemy z menu **Obraz → Skaluj obraz ....**



Skalowanie wymiarów obrazu

W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-image-scale image new-width new-height)
```

Ale jak widać musimy najpierw określić dwa nowe parametry

Parametry

image	*IMAGE*	The image
new-width	*INT32*	New image width (1 <= new-width <= 262144)
new-height	*INT32*	New image height (1 <= new-height <= 262144)

które normalnie wpisujemy manualnie do widgetu. Do tego celu zdefiniujemy zmienne:

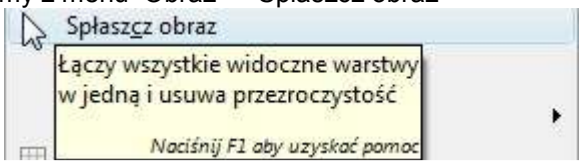
```
; Określam największe dopuszczalne wymiary zmniejszonego obrazu
      (define max-width 500)
      (define max-height 500)
; definiuję wymiary otwartego obrazu
   (define image-width (car (gimp-image-width image)))
   (define image-height (car (gimp-image-height image)))

   (if (< image-height image-width)

; instrukcja warunkowa podjęcie decyzji o wyborze dalszej drogi
;algorytmu jezeli image-height jest mniejsze od image-width zdefiniować
;jeden z ponizszych wspolczynnikow skalowania
   (define scale-factor (/ image-width max-width))
   (define scale-factor (/ image-height max-height)))
; definiowanie finalnych wymiarow obrazu
   (define final-width (/ image-width scale-factor))
   (define final-height (/ image-height scale-factor))
; okreslenie rozdzielczosci obrazu, jesli inna niz domyslna
   (gimp-image-set-resolution image 92 92)

; skalowanie
   (gimp-image-scale image final-width final-height)
```

Komentarze wyjaśniają co się dzieje

Uwaga: Procedura `gimp-image-scale` stosuje tryb interpolacji domyślnej - sześciennej.
Możliwe jest zastosowanie dowolnego trybu interpolacji, ale przy pomocy procedury gimp-image-scale-full.

Informacje dodatkowe: http://zbyma.republika.pl/pdf/Skalowanie%20obrazu%20w%20GIMP-ie.pdf

Krok 3.
Spłaszczenie obrazu
normalnie po prostu wywołujemy z menu Obraz → Spłaszcz obraz



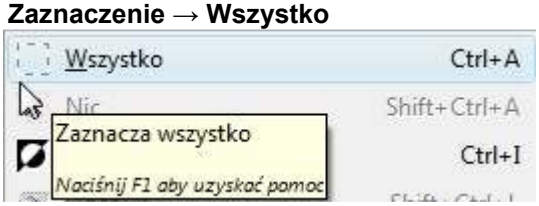
Obraz → Spłaszcz obraz

W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-image-flatten image)
```

Jedynym parametrem jest `image`.

Krok 4.
Zaznaczenie całego obrazu
W GIMP-ie wywołujemy z menu **Zaznaczenie → Wszystko**



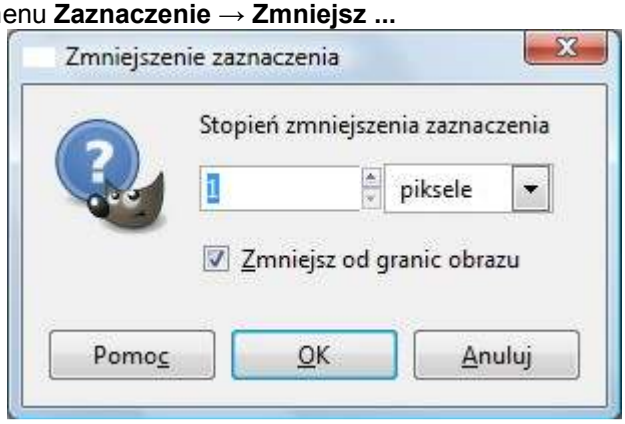
Zaznaczenie → Wszystko

W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-selection-all image)
```

Jedynym parametrem jest `image`.

Krok 5.
Zmniejszenie zaznaczenia o 2 piksele
W GIMP-ie wywołujemy z menu **Zaznaczenie → Zmniejsz ...**



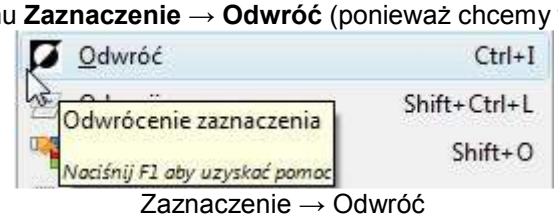
Zmniejszenie zaznaczenia

W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-selection-shrink image 2)
```

Parametrami są `image i liczba pikseli`.

Krok 6.
Odwrócenie zaznaczenia
W GIMP-ie wywołujemy z menu **Zaznaczenie → Odwróć** (ponieważ chcemy wypełnić tylko zanaczenie)



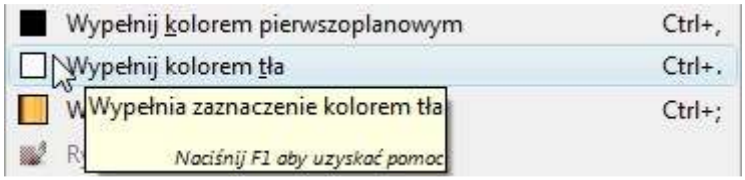
Zaznaczenie → Odwróć

W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-selection-invert image)
```

Jedynym parametrem jest `image`.

Krok 7.
Wypełnienie zaznaczenia czarnym kolorem
W GIMP-ie wywołujemy z menu **Edycja → Wypełnij kolorem pierwszoplanowym** lub (**Ctrl + ,**)
Lub Wypełnij kolorem Tła (**Ctrl + .**)



W skrypcie do tego celu możemy wykorzystać procedurę:

`(gimp-edit-fill drawable fill-type)`

Parametry

drawable	*DRAWABLE*	The drawable to fill to
fill-type	*INT32*	The type of fill { FOREGROUND-FILL (0), BACKGROUND-FILL (1), WHITE-FILL (2), TRANSPARENT-FILL (3), PATTERN-FILL (4), NO-FILL (5) }

Procedura ta wypełnia określony drawable (rysowalny) w trybie wypełnienia.
Wpisujemy typy wypełniania:
FOREGROUND-FILL lub 0 – kolorem pierwszoplanowym
BACKGROUND-FILL lub 1 - kolorem Tła
Ta procedura dotyczy tylko regionów zaznaczenia, jeśli istnieje aktywne zaznaczenie.
Inne sposoby wypełnienia nie powinien być stosowane.
Jeśli chcemy wypełnić cały drawable, bez względu na zaznaczenie, należy użyć "`gimp-drawable-fill`".

Możemy również zastosować narzędzie **Wiadro** z farbą, (przy manualnym wypełnianiu małego zaznaczenia jest konieczne duże powiększenie aby trafić Wiadrem w obszar zaznaczenia)

procedura jest bardziej rozbudowana:

```
 (let* ((drawable (car (gimp-image-active-drawable image))))
 (gimp-edit-bucket-fill drawable 0 0 100 0 0 0 0)
)
```

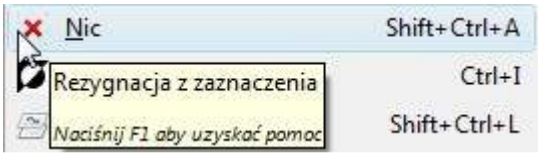
Parametry

drawable	*DRAWABLE*	The affected drawable
fill-mode	*INT32*	The type of fill { FG-BUCKET-FILL (0), BG-BUCKET-FILL (1), PATTERN-BUCKET-FILL (2) }
paint-mode	*INT32*	The paint application mode { NORMAL-MODE (0), DISSOLVE-MODE (1), BEHIND-MODE (2), MULTIPLY-MODE (3), SCREEN-MODE (4), OVERLAY-MODE (5), DIFFERENCE-MODE (6), ADDITION-MODE (7), SUBTRACT-MODE (8), DARKEN-ONLY-MODE (9), LIGHTEN-ONLY-MODE (10), HUE-MODE (11), SATURATION-MODE (12), COLOR-MODE (13), VALUE-MODE (14), DIVIDE-MODE (15), DODGE-MODE (16), BURN-MODE (17), HARDLIGHT-MODE (18), SOFTLIGHT-MODE (19), GRAIN-EXTRACT-MODE (20), GRAIN-MERGE-MODE (21), COLOR-ERASE-MODE (22), ERASE-MODE (23), REPLACE-MODE (24), ANTI-ERASE-MODE (25) }
opacity	*FLOAT*	The opacity of the final bucket fill (0 <= opacity <= 100)
threshold	*FLOAT*	The threshold determines how extensive the seed fill will be. It's value is specified in terms of intensity levels. This parameter is only valid when there is no selection in the specified image. (0 <= threshold <= 255)
sample-merged	*INT32*	Use the composite image, not the drawable (TRUE or FALSE)
x	*FLOAT*	The x coordinate of this bucket fill's application. This parameter is only valid when there is no selection in the specified image.
y	*FLOAT*	The y coordinate of this bucket fill's application. This parameter is only valid when there is no selection in the specified image.

Procedura `gimp-edit-bucket-fill` uaktualniająca narzędzie Wiadro z farbą. Wymaga `drawable` a nie `image`.
Ta procedura wymaga aż 8 parametrów:

- **drawable** : warstwa, na której wypełniamy,
- **0** : tryb wypełnienia, kolorem pierwszoplanowym = 0,
- **0** : tryb nakładania farby, 0 = NORMAL-MODE
- **100**%: Krycie
- **0** : Próg , ten parametr jest istotny tylko przy braku selekcji w obrazie
- **0** : opcji Composite używamy do złożonych obrazów, nie do drawable, dlatego 0 = FALSE,
- **0 0** : argument Współrzędne x i y mają zastosowanie, jeśli parametr composite jest TRUE, Jest to równoważne pobierania próbek kolorów po połączeniu wszystkich widocznych warstw.

Krok 8.
Odznacz wszystko
W GIMP-ie wywołujemy z menu **Zaznaczenie → Nic**

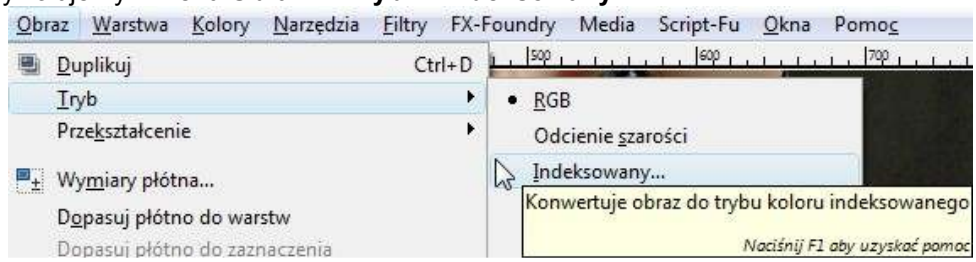


Zaznaczenie → Nic

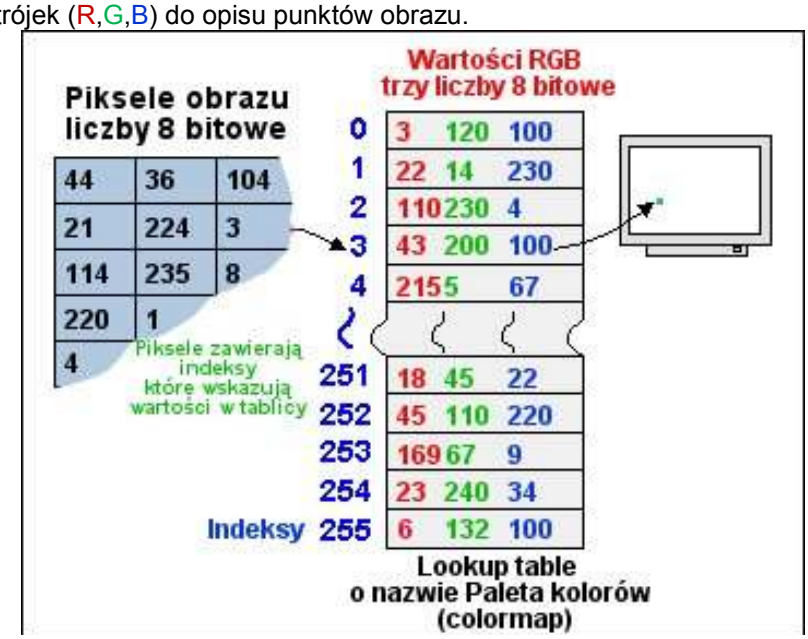
W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-selection-none image)
```

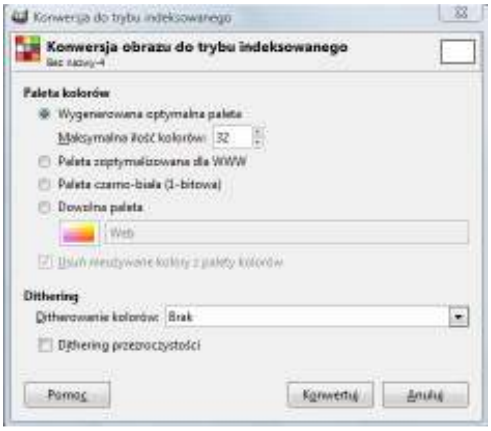
Krok 9.
Przekształcenie obrazu w tryb kolorów indeksowanych
W GIMP-ie wywołujemy z menu **Obraz → Tryb → Indeksowany…**



Konwersja na kolory indeksowane redukuje liczbę kolorów w obrazie do maksimum 256 — jest to standardowa liczba kolorów obsługiwana przez formaty GIF i PNG-8.
Aby przekonwertować obraz na tryb koloru indeksowanego, trzeba użyć obrazu w trybie RGB lub skali szarości i o 8 bitach na składową.
Opis punktu obrazu jest indeksem do palety kolorów. Ilość różnych kolorów jest równa ilości elementów w palecie. Maksimum może to być 256 różnych kolorów. To maksimum 256 różnych kolorów odpowiada 256-ciu wybranych trójek (R,G,B) do opisu punktów obrazu.



Jak widać ta technika określa 256 najczęściej używanych kolorów obrazu i z tych kolorów tworzy *tablicę*, zwaną "colormap" lub "Paleta kolorów", która jest przechowywana z obrazem. Zamiast każdego piksela obrazu, posiadającego wszystkie trzy kolory RGB 8 bitowe, każdy piksel zawiera jeden 8-bitowy numer, indeksu w 256-color lookup tabeli, która zawiera wartości RGB.
Czyli w indeksowanym obrazie, zamiast przypisania koloru bezpośrednio (jak to się dzieje w obrazach RGB i skali szarości), kolory są przypisane do pikseli metodą pośrednią, za pomocą wyszukiwania w **Lookup table,** o nazwie ***Paleta kolorów (colormap)***, która przechowuje określoną ilość przeliczonych wcześniej danych. Aby określić kolor, który należy wykazać dla jakiegoś piksela, GIMP sprawdza indeks Palety kolorów obrazu. Każdy indeksowany obraz ma swoją własną Paletę kolorów.
Paleta kolorów indeksowanego obrazu jest tablicą bajtów, trzy bajty na każdy kolor.
Ponieważ istnieje 256 możliwych 8 bitowych kolorów obrazu indeksowanego, maksymalny rozmiar listy (Paleta kolorów) ***colormap*** wynosi 768 bajtów (256 * 3). Jak pamiętamy Bajt to osiem bitów, jeden bit może mieć dwie wartości (0 lub 1), stąd 2^8 daje 256 (czyli od 0 - 255).
Paleta kolorów nie musi przedstawić wszystkich 256 kolorów, a indeks bajtu ostatnich kolorów w mapie kolorów jest mniejszy trzy razy od liczby kolorów. W trybie indeksowanym piksel może być albo całkowicie przezroczysty albo całkowicie nieprzezroczysty. A więc w wyniku indeksowania część częściowo przezroczystych pikseli zostanie usunięta, a część stanie się całkowicie nieprzezroczysta.



Konwersja do trybu Indeksowanego

W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-image-convert-indexed image 0 0 32 0 0 "")
```

gimp-image-convert-indexed

Wewnętrzna procedura GIMP-a

Convert specified image to and Indexed image

Parametry

image	*IMAGE*	The image
dither-type	*INT32*	The dither type to use { NO-DITHER (0), FS-DITHER (1), FSLOWBLEED-DITHER (2), FIXED-DITHER (3) }
palette-type	*INT32*	The type of palette to use { MAKE-PALETTE (0), WEB-PALETTE (2), MONO-PALETTE (3), CUSTOM-PALETTE (4) }
num-cols	*INT32*	The number of colors to quantize to, ignored unless (palette_type == GIMP_MAKE_PALETTE)
alpha-dither	*INT32*	Dither transparency to fake partial opacity (TRUE or FALSE)
remove-unused	*INT32*	Remove unused or duplicate color entries from final palette, ignored if (palette_type == GIMP_MAKE_PALETTE) (TRUE or FALSE)
palette	*STRING*	The name of the custom palette to use, ignored unless (palette_type == GIMP_CUSTOM_PALETTE)

Wymagane parametry to:

- **image** : obraz do konwersji,
- **0** : typ ditheringu, 0 = bez dithering, inne typy widać powyżej
- **0** : typ palety, 0 = generuje optymalną paletę,
- **32** : ilość kolorów, mała liczba kolorów może spowodować pasmowanie kolorów - banding
- **0** : dithering przejrzystości, 0 = FALSE, pozwala na poprawianie niepożądanych efektów
- **0** : usuwanie nieużywanych kolorów z palety, 0 = FALSE lub 1 =TRUE,
- **" "** : do stosowania - nazwa palety kolorów niestandardowych, można ignorować Paleta typu GIMP-CUSTOM-PALETTE.

Paleta kolorów obrazu indeksowanego (dalej będzie zmieniona)

Ważne jest, aby uświadomić sobie, że kolory na Palecie kolorów są to tylko kolory dostępne w indeksowanym obrazie.

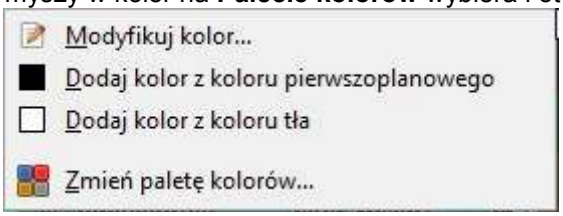
Okno Palety kolorów pozwala zmienić mapę kolorów dla obrazu, albo przez utworzenie nowej pozycji, lub zmieniając kolory istniejących wpisów. Jeśli zmiana koloru wiąże się z danym indeksem, zmiany widać odzwierciedlone w obrazie, jako zmiany koloru dla wszystkich pikseli, które są przypisane tego indeksu. Wpisy w oknie Palety kolorów są numerowane od 0 w lewym górnym rogu, co 1 dalej do prawej strony, itp., ostatni w naszym przypadku będzie 31. (Jak widać, w tej konkretnej palecie obrazu indeksowanego brak koloru białego!)
Colormap => Paleta kolorów (lepszą nazwą była by chyba Paleta indeksowana) dokowalne okno dialogowe pozwala na edycję Palety kolorów z indeksowanego obrazu co pokazano.

Jak korzystać z okna Palety kolorów

Szereg operacji które można wykonywać przy użyciu tego okna opisano:
http://docs.gimp.org/en/gimp-indexed-palette-dialog.html

Menu kontekstowe Palety kolorów

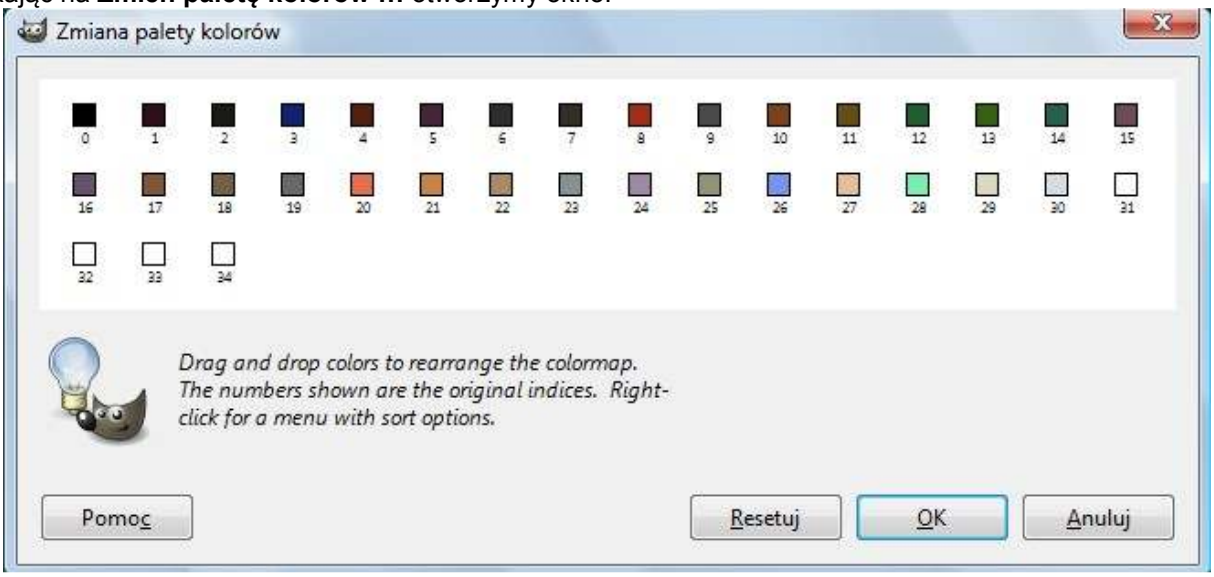
Kliknięcie prawym przyciskiem myszy w kolor na **Palecie kolorów** wybiera i otwiera menu:



Menu kontekstowe Palety kolorów

Okno Modyfikuj kolor…

w którym mamy możliwość odczytu wartości poszczególnych składowych RGB, dla danego koloru indeksowanego.
Klikając na **Zmień paletę kolorów …** otworzymy okno:



Więcej szczegółów opisano w: http://docs.gimp.org/en/plug-in-colormap-remap.html

Dodatkowa możliwość:

http://flashingtwelve.brickfilms.com/GIMP/Scripts/ skrypt **indexed-decompose.scm** dzieli obraz na N różnych kolorów i ustawia każdy kolor na innej warstwie. Jest możliwość konwersji obrazu wynikowego do trybu RGB. Każda z warstw ma nazywę przy użyciu jej kolorów składowych (np. "RGB = [128, 59, 34]". Aby można stosować skrypt w GIMP ver 2.6.11 trzeba w definicji zmiennych na początku podać wartości (muszą być przypisane), dla wszystkich wpisujemy **0** (zero).

Krok 10.
Zamiana kolorów indeksowanych na Paletę kolorów
Paletę kolorów tworzymy w oparciu o *Paletę kolorów obrazu indeksowanego* (nie jest to możliwe dla obrazu w trybie RGB lub szarości).
W GIMP-ie wywołujemy z menu **Kolory → Odwzorowania → Zmień paletę kolorów…**

W skrypcie do tego celu możemy wykorzystać procedurę:

```
(gimp-image-get-colormap image)
```

gimp-image-get-colormap
Wewnętrzna procedura GIMP-a
Returns the image's colormap

Parametry

image	*IMAGE*	The image

Zwracane wartości

num-bytes	*INT32*	Number of bytes in the colormap array (num-bytes >= 0)
colormap	*INT8ARRAY*	The image's colormap

```
Procedura ta zwraca aktualny indeks koloru do Palety kolorów obrazu, jak również
liczbę bajtów zawartych w Palecie kolorów. Rzeczywista liczba kolorów
przekazywanych do Palety kolorów jak już podano będzie 'num-bytes' / 3.
```

Jeśli spojrzymy na ostatni parametr tej procedury, stwierdzimy, że dane są typu **INT8ARRAY**. Tablica jest jedną z form wylistowania elementów. INT8ARRAY oznacza, że jest to tablica 8-bitowych liczb całkowitych. Każdy element tablicy jest liczbą od 0 do 255. Liczby całkowite dotyczące danych są typu INT32.
Jeśli w poprzednim kroku została utworzona Paleta i przypisana do zmiennej o nazwie ***colormap*** można indywidualne bajty (elementy) palety ustawić przy pomocy specjalnej formy **vector-set!**, której przekazuje się zmienną typu *tablicowego*, indeks ustawianego elementu i jego nowa wartość to:

```
(vector-set! colormap 0 r)
(vector-set! colormap 1 g)
(vector-set! colormap 2 b)
(vector-set! colormap 3 a); przezroczystość
```

Przypomnienie: `(vector-set! vec n k)`
Procedura zastępuje element o indeksie `n` w wektorze `vac` przez `k` (lub inaczej zapisuje obiekt *k* w miejsce o indeksie ***n*** wektora *vec*).

Pierwszy kolor w tablicy zaczyna się od byte-index "0", indeks drugiego bajtu zczyna się od "3", indeks trzeciego bajtu od "6", i tak dalej.

A więc rozpisujemy

```
(let* ((colormap (cadr (gimp-image-get-colormap image))))
(vector-set!  colormap 0 0);składowe RGB dla koloru w palecie o indeksie 0
(vector-set!  colormap 1 0)
(vector-set!  colormap 2 0)

(vector-set!  colormap 93 255);składowe RGB dla koloru w palecie o indeksie 32
(vector-set!  colormap 94 255)
(vector-set!  colormap 95 255)
  (gimp-image-set-colormap image (* 32 3) colormap)
)
```

Polecenie **let*** stworzy zmienną lokalną w bloku. Zmienna otrzyma wektor ***colormap*** pobrany z Palety kolorów wykorzystywanych przez aktualny obraz indeksowany.
Tablica (vector) zawiera 1 -ą wartość na każdą składową (R czerwony, G zielony, B niebieski).
Paleta kolorów została określona w poprzednim kroku na 32 kolory,dlatego też tablica zawiera 96 wartości:

- wartości o indeksie 0, 1, 2 odpowiadają składowym pierwszego koloru (odcieniom barwy),
- wartości o indeksie 93, 94, 95 odpowiadają składowym ostatniego koloru.

Przykład

Procedura **vector-set!** używana do ustawiania wartości w tabelicy, wymaga 3 wywołań, tego polecenia dla zmiany jednego koloru w Palecie kolorów.

No i wreszcie, procedura `gimp-image-set-colormap` pozwala na relokację utworzonej Palety kolorów na obraz.
Jej parametry to:

- **image** : zmieniany obraz,
- **(* 32 3)** : liczba wpisów w liście (32 kolory × 3 składowe koloru),
- **colormap** : nowa paleta kolorów.

Procedura `gimp-image-set-colormap` odpowiada za działania zmierzające do zmiany palety:

Regulacja palety

Uwaga! Nie ma potrzeby stosować (car), aby przypisać zmiennej wartość, tylko w *native* GIMP procedury zwracają wartości w postaci listy, **`vector-set!`** nie jest funkcją GIMP-a nie znajdziemy jej w PP !.

Krok 11.
Koniec cofnięć grupy
Procedura `(gimp-image-undo-group-end image)` koniec bloku cofnięć (od początku funkcji) oraz, `(gimp-displays-flush)` odświeża wyświetlacz, po zmianach.

Teraz możemy już przystąpić do pisania tego skryptu.
Tak jak poprzednio, pracować będziemy przede wszystkim w edytorze, a nie z Konsoli Script-Fu.
Może to być **Notepad++**, które prawidłowo rozpoznaje i koloruje składnię języka TinyScheme.
Otwieram Notepad++ i zaczynam pisać skrypt kolejnymi krokami, umożliwiający wprowadzenie zmian w obrazie:

```
(define (script-fu-zmiany image)
  ; początek cofania grupy
  (gimp-image-undo-group-start image)

  ; ustawienie kolor pierwszoplanowy (czarny) i tła (białe)
  (gimp-context-set-foreground '(0 0 0))
  (gimp-context-set-background '(255 255 255))
  ; dokładne przycięcie obrazu
  (plug-in-autocrop 0 image (car (gimp-image-get-active-drawable image)))
  ; Określam największe dopuszczalne wymiary zmniejszonego obrazu
    (define max-width 500)
    (define max-height 500)
; definiuję wymiary otwartego obrazu
   (define image-width (car (gimp-image-width image)))
   (define image-height (car (gimp-image-height image)))

   (if (< image-height image-width)
; instrukcja warunkowa podjęcie decyzji o wyborze dalszej drogi
;algorytmu jezeli image-height jest mniejsze od image-width zdefiniować
;jeden z ponizszych wspolczynnikow skalowania
   (define scale-factor (/ image-width max-width))
   (define scale-factor (/ image-height max-height)))
; definiowanie finalnych wymiarow obrazu
   (define final-width (/ image-width scale-factor))
   (define final-height (/ image-height scale-factor))
; okreslenie rozdzielczosci obrazu, jesli inna niz domyslna
   (gimp-image-set-resolution image 92 92)
; skalowanie
   (gimp-image-scale image final-width final-height)

  ; splaszczenie obrazu
  (gimp-image-flatten image)

  ; zaznaczenie calego obrazu
  (gimp-selection-all image)

  ; zmniejsz zaznaczenie o 2 pix
  (gimp-selection-shrink image 2)

  ; odwroc zaznaczenie
  (gimp-selection-invert image)

  ; wypełniamy zaznaczenie czarnym (kolor tła)
  (let* ((drawable (car (gimp-image-active-drawable image))))
    (gimp-edit-bucket-fill drawable 0 0 100 0 0 0 0)
  )

  ; odznacz wszystko
  (gimp-selection-none image)

  ; przełączenie trybu kolorów na indeksowane do 32 kolorów
  ; (gimp-image-convert-indexed image NO-DITHER MAKE-PALETTE 32 FALSE FALSE "")
  (gimp-image-convert-indexed image 0 0 32 0 0 "")

  ; Zmiana kolorow indeksowanych na Paletę kolorow
  (let* ((colormap (cadr (gimp-image-get-colormap image))))
  (vector-set!  colormap 0 0)
  (vector-set!  colormap 1 0)
  (vector-set!  colormap 2 0)

  (vector-set!  colormap 93 255)
  (vector-set!  colormap 94 255)
  (vector-set!  colormap 95 255)


    (gimp-image-set-colormap image (* 32 3) colormap)
  )

  ; koniec cofnęć grupy
  (gimp-image-undo-group-end image)
  (gimp-displays-flush) ; Odswieza wyswietlacz, po zmianach
)

(script-fu-register "script-fu-zmiany"
                    "<Image>/Script-Fu/Poradnik.../Wykonanie kolejno zmian"
                    "Wykonanie kolejno zmian w obrazie"
                    "Zbyma"
                    "GPL"
                    "2011"
                    "RGB*"
                    SF-IMAGE "Input Image" 0

)
```

```
(define (script-fu-zmiany image)
  ; początek cofania grupy
  (gimp-image-undo-group-start image)

  ; ustawienie kolor pierwszoplanowy (czarny) i tła (białe)
  (gimp-context-set-foreground '(0 0 0))
  (gimp-context-set-background '(255 255 255))
  ; dokładne przycięcie obrazu
  (plug-in-autocrop 0 image (car (gimp-image-get-active-drawable image)))
  ; Określam największe dopuszczalne wymiary zmniejszonego obrazu
      (define max-width 500)
      (define max-height 500)
; definiuję wymiary otwartego obrazu
   (define image-width (car (gimp-image-width image)))
   (define image-height (car (gimp-image-height image)))


   (if (< image-height image-width)

; instrukcja warunkowa podjęcie decyzji o wyborze dalszej drogi
;algorytmu jezeli image-height jest mniejsze od image-width zdefiniować
;jeden z ponizszych wspolczynnikow skalowania
   (define scale-factor (/ image-width max-width))
   (define scale-factor (/ image-height max-height)))
; definiowanie finalnych wymiarow obrazu
   (define final-width (/ image-width scale-factor))
   (define final-height (/ image-height scale-factor))
; okreslenie rozdzielczosci obrazu, jesli inna niz domyslna
   (gimp-image-set-resolution image 92 92)

; skalowanie
   (gimp-image-scale image final-width final-height)

  ; splaszczenie obrazu
  (gimp-image-flatten image)

  ; zaznaczenie calego obrazu
  (gimp-selection-all image)

  ; zmniejsz zaznaczenie o 2 pix
  (gimp-selection-shrink image 2)

  ; odwroc zaznaczenie
  (gimp-selection-invert image)

  ; wypełniamy zaznaczenie czarnym (kolor tła)
  (let* ((drawable (car (gimp-image-active-drawable image))))
    (gimp-edit-bucket-fill drawable 0 0 100 0 0 0 0)
  )

  ; odznacz wszystko
  (gimp-selection-none image)

  ; przełączenie trybu kolorów na 32 kolory indeksowane

      (gimp-image-convert-indexed image 0 0 32 0 0 "")

  ; Zamiana kolorow indeksowanych na Paletę kolorow
  (let* ((colormap (cadr (gimp-image-get-colormap image))))
  (vector-set!  colormap 0 0)
  (vector-set!  colormap 1 0)
  (vector-set!  colormap 2 0)

  (vector-set!  colormap 93 255)
  (vector-set!  colormap 94 255)
  (vector-set!  colormap 95 255)
```

```
    (gimp-image-set-colormap image (* 32 3) colormap)
  )

  ; koniec cofnęć grupy
  (gimp-image-undo-group-end image)
  (gimp-displays-flush) ; odswieza wyswietlacz, po zmianach
)

(script-fu-register "script-fu-zmiany"
                "<Image>/Script-Fu/Poradnik.../Wykonanie kolejno zmian"
                "Wykonanie kolejno zmian w obrazie"
                "Zbyma"
                "GPL"
                "2011"
                "RGB*"
                SF-IMAGE "Input Image" 0

)
```

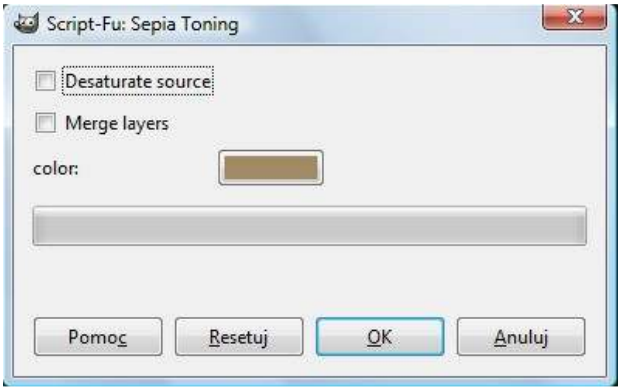
Po zastosowaniu `script-fu-zmiany` – obraz wygląda jak malowany.

Istnieje dużo ładnych Tutoriali do GIMP ver. 1.x, których autorem jest **Eric R. Jeschke (GIMP Guru).**
Na podstawie jednego z nich: http://www.gimp.org/tutorials/Sepia_Toning/ oraz dodatkowych wskazówek z http://www.retouchpro.com/tutorials/lum-mask-sepia.html skrypt opracował Jakub Klawiter http://registry.gimp.org/node/17159 postanowiłem opublikować go tutaj z uwagi na jego możliwości szkoleniowe:

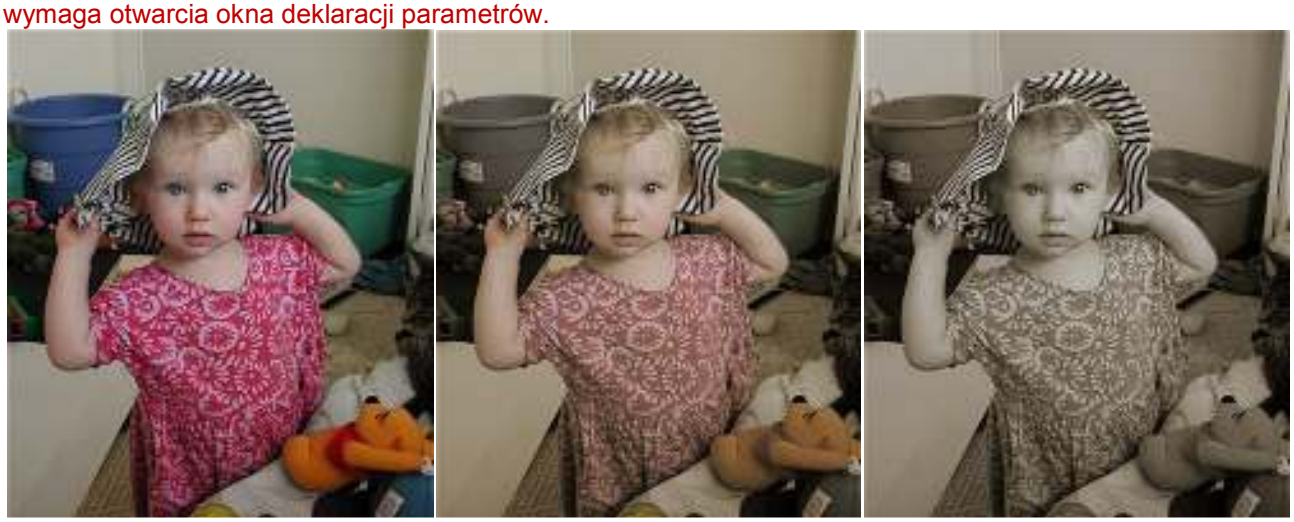
zobaczymy, po kolei jak wszystko dziala

```
; Sepia toning script for GIMP
; version 0.4 - GIMP 2.6.x
; just to learn how all it works ;-)
; by Jakub Klawiter 05.2007 - 11.2007
;
; this is a copy of Sepia Toning tutorial
; http://www.gimp.org/tutorials/Sepia_Toning/
; by Eric R. Jeschke
;
; copyleft GNU GPL v3 etc. etc.
;

(define
    (script-fu-Sepia_Toning
        img
        drawable
        desaturate
        merge-layers
        color
    )

; Start an undo group. Everything between the start and the end will
; be carried out if an undo command is issued.
    (gimp-image-undo-group-start img)
    (gimp-displays-flush)

    (let*
        (  ; definicja zmiennych konieczna dla GIMP 2.6
           (sepia-layer 0)
           (mask-layer  0)
           (mask 0)
        )

; STEP 2 - copy and desaturate (optional) source layer

    (set! sepia-layer
        (car
            (gimp-layer-copy
                drawable
                TRUE
            )
        )
    )
    (gimp-layer-set-name sepia-layer "Sepia")
    (gimp-image-add-layer img sepia-layer -1)

    (if (equal? desaturate TRUE)
        (gimp-desaturate sepia-layer)
        ()
    )

; STEP 3 Set foreground color
    (gimp-context-set-foreground color)

; STEP 4
; Create a new layer
    (set! mask-layer
        (car
            (gimp-layer-new
                img                                 ; image handle
```

```
                (car (gimp-image-width img))    ; width of layer
                (car (gimp-image-height img))   ; height
                1                               ; type (RGB, RGBA, etc.)
                "Sepia Mask"                    ; name of layer
                100                             ; opacity
                COLOR-MODE                      ; mode
            )
        )
    )

; Add the new layer to the image
    (gimp-image-add-layer img mask-layer -1)

    (gimp-drawable-fill mask-layer 0)

; STEP 5
    (set! mask
        (car
            (gimp-layer-create-mask mask-layer 0)
        )
    )
    (gimp-layer-add-mask mask-layer mask)

; STEP 6, 7 Copy image into Sepia Layer mask, and than invert it
    (gimp-layer-resize-to-image-size sepia-layer) ; workaround because i cannot
'paste in place' into mask
    (gimp-edit-copy sepia-layer)

    (let ((selection (car (gimp-edit-paste mask 0))))
        (gimp-floating-sel-anchor selection)
    )
    (gimp-invert mask)

; merge layer down
    (if (equal? merge-layers TRUE)
        (gimp-image-merge-down
            img             ; img
            mask-layer      ; upper layer
            0               ; merge type [0,1,2]
        )
        ()
    )

    ) ; let* variables definition

; Complete the undo group
    (gimp-image-undo-group-end img)
)

(script-fu-register "script-fu-Sepia_Toning"
      "Sepia Toning"
      " Jest to automatyzacja wspaniałego Sepia Toning tutorial
by Eric R. Jeschke http://www.gimp.org/tutorials/Sepia_Toning/"
      "Jakub Klawiter"
      " copyleft GNU GPL v3"
    "03.2007"
    "RGB RGBA"
      SF-IMAGE      "img"                 0
      SF-DRAWABLE   "drawable"            0
      SF-TOGGLE     "Desaturate source"   FALSE
    SF-TOGGLE     "Merge layers"        FALSE
    SF-COLOR      "color"               '(162 138 101))

(script-fu-menu-register
    "script-fu-Sepia_Toning"
    "<Image>/Script-Fu/Photo"
)
```



Jak widać:
Użycie interaktywnych parametrów powoduje, że skrypt jest do użytku w trybie wsadowym, ponieważ wymaga otwarcia okna deklaracji parametrów.



Oryginał | Sepia | Sepia desaturate

Kilka dodatkowych info:
Uruchamiamy GIMP-a, w oknie obrazu otwarte jedno zdjęcie. Otwieramy konsolę KSF.
Wpisujemy w konsoli proste polecenie, `(gimp-version)` i klikamy "**Zastosuj**", na ekranie pojawi się:

```
> (gimp-version)
("2.6.10");jako wartość zwróconą otrzymamy numer wersji stosowanego GIMP-a.
```

lub

```
> (string-append "Versions: " (car (gimp-version)))
"Versions: 2.6.10";inna możliwość
```

Teraz spróbujemy ustalić listę aktualnie otwartych obrazów, w KSF klikamy **Przeglądaj**, otwiera się okno PP w którym wpisujemy `(gimp-image-list)` i mamy wyświetlone co zwróci procedura,

gimp-image-list
Wewnętrzna procedura GIMP-a
Returns the list of images currently open.

Zwracane wartości

num-images	*INT32*	The number of images currently open (num-images >= 0)
image-ids	*INT32ARRAY*	The list of images currently open

klikamy **Zastosuj**, co przepisze procedurę do KSF, klikamy **Enter**, zostanie wyświetlona wartość zwrócona:

```
> (gimp-image-list)
(1 #( 1 ))
```

Jak należy tłumaczyć otrzymaną sekwencję zwróconych wartości widać z PP.

```
; otwarty jeden obraz, zwrócona lista zawiera dwa elementy,
;drugi element jest listą (tablicą) w rzeczywistości vector-em ID otwartych obrazów,
;pierwszy element jest liczbą, ile vector zawiera otwartych obrazów.
```

Jak sie nazywa otwarty obraz

```
> (gimp-image-get-name image)
Error: eval: unbound variable: image
```

Wybierz pierwszy parametr "image" następnie zastąpić go pierwszym numerem ID

```
> (gimp-image-get-name 1)
("doll_li.jpg"); konsola wyświetliła nazwę jako łańcuch w nawiasach
```

Możemy przejść do ID obrazu za pomocą funkcji odwołania do pierwszego elementu tablicy `( vector-ref )` z listy `(gimp-image-list)`.

```
> (car (gimp-image-get-name (vector-ref (cadr (gimp-image-list)) 0)))
"doll_li.jpg"
```
Spróbujmy dowiedzieć się jeszcze ile mamy warstw:
```
> (gimp-image-get-layers 1)
(1 #( 2 )); jedna warstwa o ID 2
```
Możemy uzyskać te dane inną procedurą:
```
> (gimp-image-get-active-layer 1)
(2);aktywna warstwa obrazu 1 ma ID 2
```
Aby uzyskać dostęp do wpisu w tablicy, należy użyć "`vector-ref`":
```
> (define all-images (gimp-image-list))
all-images
> all-images
(1 #( 1 ))
> (define first-image (vector-ref (cadr all-images) 0))
first-image
> first-image
1
```
Teraz zaczniemy jeszcze od innej procedury:
```
>(gimp-image-height image)
Error: eval: unbound variable: image
```
Ale jeśli wykorzystamy element wektora `#( 1 )`, z `(gimp-image-list)` jako parametr obrazu w procedurze:.
```
> (gimp-image-height 1)
(901); watość zwrócona to wysokość otwartego obrazu co już powyżej było podane
```
Inne możliwości:
```
> (gimp-image-list)
(0 #( )); nie otwarto żadnego obrazu
> (gimp-image-list)
(3 #( 3 2 1 )); otwarto 3 obrazy o kolejnych ID 1, 2, 3
```
ID 3, 2 lub 1 mogą się różnić w sesji, jeśli już otwierano i zamykano inne zdjęcia!.
```
> (define image 1)
Image
> (define layer (car (gimp-image-get-active-layer image)))
layer
```
Jak już podano wcześniej, wartość ID obrazu jest dołączana do belki okna obrazu, co sprawia, że interakcja z poziomu konsoli jest bardzo łatwa. W przypadku gdy mamy np. wyświetlone na pasku tytułowym "Untitled.jpg-1.0", to słowo "Untitled" oznacza, że jest to nowy obraz, którego ID jest "1" a "0" po kropce jest numerem ID widoku (można mieć więcej niż jeden widok tego samego obrazu, otwartych za pomocą polecenia "Widok => Nowy widok" oraz "1" w nawiasie oznacza jedną warstwą, chyba że mamy więcej niż jedną warstwę tego samego obrazu).

Co można dalej:

1. Otwieramy w GIMP-ie zdjęcie
2. Otwieramy w menu **Filtry** => **Script-Fu** => **Konsola**
3. Kopiujemy poniższy kod i wklejamy go w polu Script-Fu Konsoli

```
; kod wyświetla nazwę bieżącego obrazu jego szerokość i wysokość
(define (pictureInfo)
    (let* ; deklaracja zmiennych lokalnych
        (   (image      (vector-ref (cadr (gimp-image-list)) 0)) ; uchwyt
pierwszego zdjęcia
            (filename   (car (gimp-image-get-name image)))
            (width      (car (gimp-image-width image)))
            (height     (car (gimp-image-height image)))
        )
        (display filename)
        (newline)

        (display width)
        (display " x ")
        (display height)
        (newline)
    )
)
```
Klikamy Enter procedura zwróci
```
pictureInfo
```
Wywołujemy funkcję

```
> (pictureInfo)
```
Zwraca:
```
doll_li.jpg
922 x 901
#t
```
Nazwa, szerokość i wysokość zmienia się oczywiście w zależności od otwartego obrazu.

I to by było na tyle wprowadzenia, resztę trzeba opanować samemu!

Autor opracowania:
inż. Zbigniew Małach
Zbyma72age

Linki jako literatura uzupeniająca:
http://docs.gimp.org/pl/gimp-using-script-fu-tutorial.html
http://docs.gimp.org/en/gimp-scripting.html
http://docs.gimp.org/ru/gimp-filters-script-fu.html
http://www.gimp.org/docs/script-fu-update.html
ftp://ftp.fernuni-hagen.de/pub/mirrors/www.gimp.org/manual/GUM/write_scriptfu3.html#449771
http://pl.wikibooks.org/wiki/GIMP/Programowanie
http://www.ic.al.lg.ua/~ksv/gimpdoc-html/write_sc.html
http://www.ve3syb.ca/wiki/doku.php?id=software:sf:writing
http://www.ve3syb.ca/wiki/doku.php?id=software:sf:updating-scripts
http://www.ve3syb.ca/wiki/doku.php?id=software:sf:reg-block-args
http://www.ve3syb.ca/wiki/doku.php?id=software:sf:writing#the_re_extension
http://gug.criticalhit.dk/tutorials/titix1/
http://gug.sunsite.dk/tutorials/tomcat17/
http://gug.sunsite.dk/tutorials/thepeach1/
http://gug.criticalhit.dk/tutorials/tomcat3/
http://www.xgarreau.org/aide/devel/langtk/scm_gimp.php
http://www.cs.indiana.edu/scheme-repository/imp/siod.html SIOD szczegóły
http://www.home.unix-ag.org/simon/gimp/guadec2002/gimp-plugin/html/index.html
http://photolinux.freeforums.org/desaturation-partielle-t257.html dokładny opis tworzenia skryptu
http://chl.be/glmf/www.linuxmag-france.org/old/lm1/fu.html
http://www.developpez.net/forums/d917861/logiciels/autres-logiciels/imagerie/gimp-2-6-script-fu-traitement-lots/
http://www.cs.hut.fi/Studies/T-93.210/schemetutorial/node10.html poradnik
http://wiki.gimpforum.de/wiki/Skript-Fu
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Grundlagen
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Variablen
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Listen
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Funktionen
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Rezept
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Skript
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Benutzereingaben
http://wiki.gimpforum.de/wiki/Skript-Fu-Einstieg:_Fehlersuche
http://wiki.gimpforum.de/wiki/Skript-Fu_text-effekt
http://wiki.gimpforum.de/wiki/Skript-Fu:_Bilder_bearbeiten
http://wiki.gimpforum.de/wiki/Skript-Fu:_If_und_While
http://wiki.gimpforum.de/wiki/Skript-Fu:_Erweiterte_Abfragen
http://wiki.gimpforum.de/wiki/Skript-Fu:_Der_Fortschrittsbalken

Wstęp do pisania skryptów: http://www.gimptalk.com/forum/viewtopic.php?f=9&t=10447
76228 http://www.gimptalk.com/forum/topic/An-introduction-to-Script-fu-10447-1.html
http://imagehacking.blogspot.com/2010/02/getting-started-scripting-in-gimp.html
http://home.arcor.de/habr64/gimp/pdb/script_fu.html
http://gimpbook.com/scripting/gimp-script-templates/test-sphere.scm
http://imagic.weizmann.ac.il/~dov/gimp/scheme-tut.html
http://lists.xcf.berkeley.edu/lists/gimp-user/2007-January/009428.html
http://www.seul.org/~grumbel/gimp/script-fu/script-fu-tut.html
http://www.home.unix-ag.org/simon/gimp/guadec2002/gimp-plugin/html/index.html

http://www.cs.grinnell.edu/~rebelsky/Glimmer/ScriptFu/Labs/script-fu.html
http://www.cs.grinnell.edu/ ~ rebelsky / Glimmer / ScriptFu / Labs / script-fu-procs.html
http://www.math.grin.edu/~rebelsky/Courses/CS151/2006F/Readings/script-fu-procs.html
http://gja.frndz.pagesperso-orange.fr/grafisme/script-fu/SFops_01.htm
http://gja.frndz.pagesperso-orange.fr/grafisme/script-fu/SFops_02.htm
http://gja.frndz.pagesperso-orange.fr/grafisme/script-fu/SFops_03.htm
http://gja.frndz.pagesperso-orange.fr/grafisme/script-fu/SFops_03.htm Traceur mathématique pour le GIMP
http://abcdugimp.free.fr/gimp/apprendre/scheme/scheme.php
http://abcdugimp.free.fr/gimp/apprendre/scheme/script-fu-register.php
http://abcdugimp.free.fr/gimp/apprendre/scheme/debogage.php
http://abcdugimp.free.fr/gimp/apprendre/scheme/mode-de-calque.php
http://abcdugimp.free.fr/gimp/apprendre/scheme/tuto-avant-propos.php
http://www.latenightpc.com/blog/archives/2005/07/23/hue-rotation-using-the-gimp
http://beefchunk.com/documentation/lang/gimp/GIMP-Scripts-Fu.html
http://techdem.centerkey.com/2008/02/gimp-screenshot-processor-script-fu.html
http://benjisimon.blogspot.com/2009/04/2-helpful-script-fu-resources-and-more.html
http://benjisimon.blogspot.com/2009/04/more-script-fu-practice-makes-perfect.html
http://stackoverflow.com/questions/1386293/how-to-parse-out-base-file-name-using-script-fu/1497109#1497109
http://www.ehow.com/how_5715564_use-script-fu-python-fu-gimp.html How to Use Script-Fu and Python-Fu for **GIMP**
http://www.ehow.com/how_5053701_install-scriptfu-script-gimp-windows.html
http://www.ehow.com/video_5103423_make-vector-brushes-gimp.html
http://leoblog.ru/?Proekty:Pesni_pro_grafiku:I_snova_o_Script-Fu_v_GIMP
http://meinwords.wordpress.com/2009/01/31/script-fu-template-for-batch-processing-in-gimp/
http://old.nabble.com/Creating-unique-file-names-with-Script-Fu-td28974951.html
http://www.echolalie.org/gimp/script-fu-parametriccurves.html
http://www.gimpdome.com/index.php?board=33.0
http://kreisel.fam.cx/webmaster/file/script-fu/hp.vector.co.jp/authors/VA025935/script-fu/script-fu.html
http://imagic.weizmann.ac.il/~dov/gimp/scheme-tut.html
http://www.rru.com/~meo/gimp/faq-dev.html
http://tunes.org/wiki/scheme.html
http://hans.breuer.org/gimp/pdb/bymenu.html
http://files.slembcke.net/gimp-ruby/gimp-ruby-guide.html
http://ningelgen.eu/04_GIMP/GimpDateien/Kapitel13_Scriptfu.pdf