# Podstawy programowania skryptów Sh

**Streszczenie**

Artykuł jest wprowadzeniem do tworzenia skryptów w języku powłoki Sh. Przedstawione są podstawowe konstrukcje języka Sh. Najważniejsze mechanizmy wykorzystywane przy tworzeniu skryptów są zademonstrowane na przykładach. Pokazany jest również tryb edycyjny edytora GNU Emacs, przydatny przy pisaniu skryptów.

## Spis treści

# 1. Wstęp

Prędzej czy później, praktycznie każdy administrator systemu uniksowego zaczyna potrzebować narzędzia programistycznego, które ułatwiłoby jego pracę. Jego wybór często pada na język skryptowy powłoki sh.

Podstawowymi cechami takiego narzędzia programistycznego powinny być z jednej strony prostota i łatwość wykorzystania dla osoby, która nie musi być programistą, a z drugiej uniwersalność i duże możliwości. Języki skryptowe są naturalnym wyborem, gdyż nie wymagają dodatkowych narzędzi takich jak na przykład kompilator. Ponieważ są one interpretowane to kod źródłowy programu jest równocześnie kodem wykonywalnym. Pozwala to na stosunkowo łatwe i szybkie pisanie oraz testowanie programów.

Aktualnie dostępnych jest wiele zaawansowanych języków skryptowych. Najbardziej znany i często uznawany za najpotężniejszy to zapewne Perl. Popularność zdobywa również nowoczesny, zorientowany obiektowo, Python. Rozpowszechniony jest Tcl wraz ze swoimi rozszerzeniami do tworzenia interfejsów graficznych, takich jak Tk.

Język skryptowy powłoki sh nie jest „pełnowartościowym" językiem w takim sensie w jakim są nimi wyżej wymienione języki. Ma najprostsze typy danych i stosunkowo ograniczony zbiór instrukcji. Zapewnia za to rozbudowane mechanizmy komunikacji, które rekompensują brak dodatkowych bibliotek czy modułów. Zaletą sh jest więc w praktyce jego prostota i uniwersalność. Dlatego służy najczęściej do efektywnego i łatwego łączenia pracy standardowych narzędzi uniksowych. Tego typy języki określa się często po angielsku *glue languages*, czyli językami „sklejającymi" działanie innych narzędzi.

# 2. Co to jest skrypt sh?

Skrypt w języku sh to plik tekstowy zawierający dowolne polecenia wykonywalne oraz wyrażenia języka sh. Tak więc w praktyce nawet prosta sekwencja poleceń, normalnie wydawanych w powłoce sh, zapisana w pliku staje się skryptem. Aby taki skrypt można było wykonywać bezpośrednio poprzez podanie nazwy należy nadać mu atrybut wykonywalności.

Wykonywalny plik tekstowy zawierający polecenia jest uruchamiany najczęściej przy pomocy powłoki logowania użytkownika. Aby mieć pewność, że skrypt zostanie uruchomiony przy pomocy odpowiedniego interpretera trzeba to w nim zapisać. Pierwsza linia każdego skryptu powinna mieć postać:

```
#!/pełna_ścieżka/nazwa_interpretera opcje
```

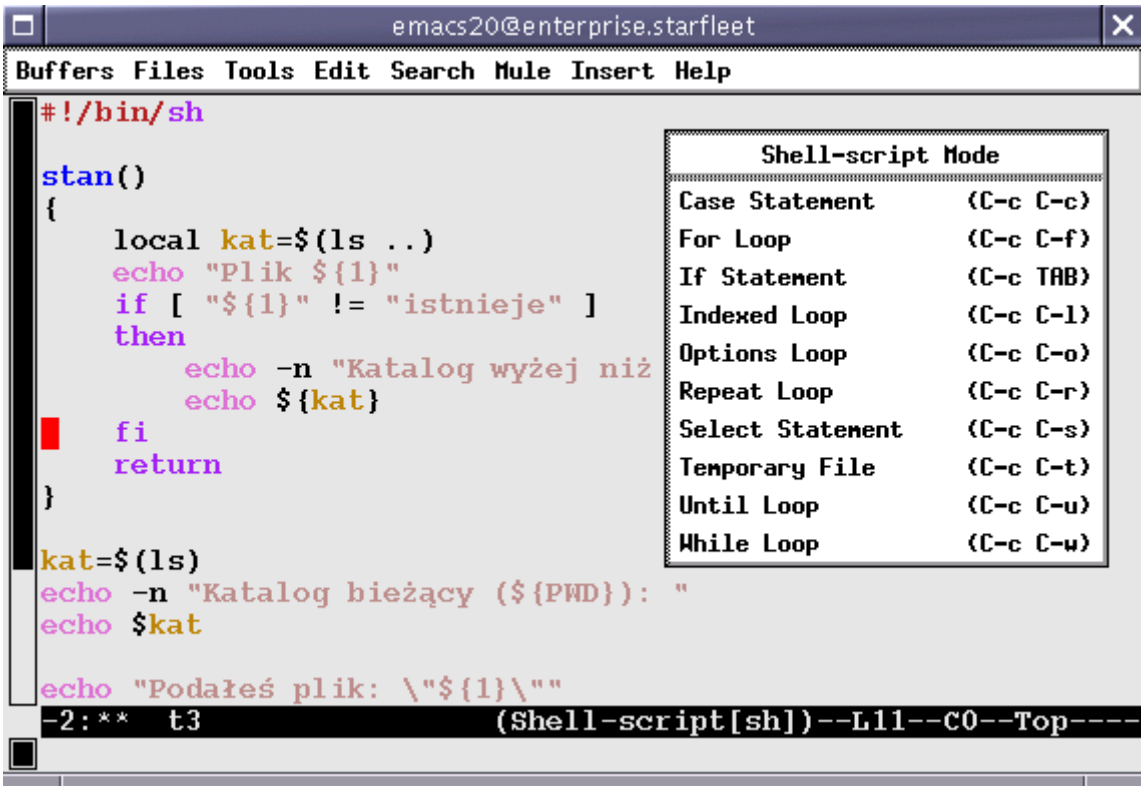
Co w przypadku skryptów sh wygląda następująco:

```
#!/bin/sh
```

Od znaku hash (`#`) zaczyna się również komentarze w skryptach.

Ponieważ skrypty są zwykłymi plikami tekstowymi, do ich pisania może być użyty dowolny edytor. Warto jednak zauważyć, że niektóre edytory mają rozbudowane tryby edycji skryptów sh. Na przykład GNU Emacs ma wbudowany tryb *sh-mode*, pozwalający na kolorowanie składni skryptu i udostępniający makra do tworzenia najczęściej wykorzystywanych konstrukcji. Przykład pracy z tym trybem jest pokazany na Rysunku 1. Również edytory Vim i mcedit mają tryby edycyjne sh.

Do pisania złożonych skryptów potrzebna jest znajomość podstawowych elementów języka sh, takich jak zmienne, metody komunikacji z otoczeniem skryptu, instrukcje sterujące sh i funkcje. Wszystkie te elementy zostaną omówione poniżej.




Rysunek 1: Tryb sh GNU Emacsa

Omawiane przykłady są uruchamiane w środowisku powłoki GNU Bash, będącej prawdopodobnie najlepszą i jedną z najpopularniejszych implementacji sh. Tym niemniej większość z nich powinna działać w innych powłokach zgodnych z sh.

# 3. Zmienne

Podstawowym typem zmiennych używanych przez sh są napisy. Zmienne definiuje się przy pomocy operatora przypisania:

```
zmienna=wartość
```

Wartość zmiennej jest napisem. Nazwa zmiennej może zawierać wielkie i małe litery, z tym że tradycyjnie zmienne środowiskowe powłoki są pisane wielkimi literami. Pomiędzy nazwą, operatorem przypisania, a wartością, nie może być odstępów.

Poprzedzając nazwę zmiennej słowem kluczowym `export` wymusza się jej dziedziczenie przez procesy potomne danej powłoki. Innymi słowy, jeżeli zmienna zostanie zdefiniowana w ten sposób, to będzie ona dostępna dla wszystkich programów uruchomionych z danej powłoki (lub skryptu).

Przy definiowaniu i odwoływaniu się do zmiennych przydają się znaki cytowania czyli apostrofy i cudzysłowy. Pozwalają one na przypisywanie napisów składających się z kilku słów. Różnica między nimi polega na tym, że apostrofy wyłączają mechanizm podstawiania nazw zmiennych i znaków specjalnych powłoki, na przykład:

```
$ echo "To jest shell:" "$SHELL"
To jest shell: /bin/bash
$ echo "To jest shell:" '$SHELL'
```

```
To jest shell: $SHELL
```

Przykład pokazuje również jak odwoływać się do wartości zmiennych – poprzez znak dolara. Użyte w przykładzie polecenie powłoki *echo* służy do wypisywania tekstów.

Poprzedzając nazwę zmiennej znakiem dolara powinno się ją otaczać nawiasami klamrowymi. Pozwala to na bezpieczne włączanie wartości zmiennych w napisy:

```
$ echo "To jest shell:${SHELL}-tu dlaszy ciąg"
To jest shell:/bin/bash-tu dlaszy ciąg
```

Jednym z podstawowych mechanizmów umożliwiających łączenie skryptów sh z innymi poleceniami jest mechanizm wstawiania wyniku działania innych poleceń do wartości zmiennych.

```
$ DATE=‘date‘ ; echo "Aktulna data: ${DATE}"
Aktulna data: Sun Nov 26 21:57:37 CET 2000
$ DATE=$(date) ; echo "Aktulna data: ${DATE}"
Aktulna data: Sun Nov 26 21:57:51 CET 2000
```

W tym przypadku wartości zmiennej `DATE` jest przypisywany wynik działania polecenia *date*. Drugi z podanych przykładów działa wyłącznie w powłoce GNU Bash.

Mówiąc o zmiennych należy wspomnieć o tym, że skrypt sh ma, podobnie jak inne programy uruchamiane z poziomu powłoki, dostęp do zmiennych środowiskowych:

```
$ echo "Katalog domowy to: ${HOME}"
Katalog domowy to: /home/gjn
```

W każdym skrypcie dodatkowo jest definiowany szereg zmiennych związanych z parametrami jego wywołania. Zmienne w postaci `$N`, gdzie `N` jest liczbą, przechowują argumenty wywołania skryptu, czyli są odpowiednikami napisów `argv[N]` w języki ANSI C. Zmienna `$0` przechowuje nazwę z jaką został wywołany skrypt, `$#` liczbę wszystkich argumentów, a `$@` wszystkie argumenty. Poniższy przykład prezentuje ich użycie:

```
#!/bin/sh
echo "Skrypt ${0} został wywołany z ${#} argumentami."
echo "Z czego 1. to: ${1}, a 3. to: ${3}"
```

Uruchomienie tego prostego skryptu daje wynik:

```
$ ./t1 alfa beta gamma
Skrypt ./t1 został wywołany z 3 argumentami.
Z czego 1. to: alfa, a 3. to: gamma
```

# 4. Instrukcje warunkowe

Język sh oferuje podstawowe konstrukcje sterujące, takie jak *if* i *case*.

Instrukcja warunkowa *if* ma następującą składnię:

```
if [ warunek ]
then
   instrukcje
elif [ warunek ]
then
   instrukcje
else
   instrukcje
fi
```

Do konstruowania warunków najczęściej wykorzystuje się instrukcję *test(1)*. Powłoka pozwala na jej uproszczone wywołanie, to znaczy z pominięciem samej nazwy polecenia *test*, a jedynie z podaniem odpowiedniego operatora porównania. Operatory polecenia *test* są przedstawione w Tabeli 1. Przy porównywaniu napisów trzeba pamiętać o konieczności otaczania wartości

| Operator | Opis |
|---|---|
| Operacje na plikach | prawdziwe jeżeli `plik` |
| `-e plik` | istnieje |
| `-d plik` | jest katalogiem |
| `-f plik` | jest zwykłym plikiem |
| `-L plik` | jest linkiem symbolicznym |
| `-r plik` | można czytać |
| `-w plik` | można zapisywać |
| `-x plik` | można wykonywać |
| `plik1 -nt plik2` | jeżeli `plik1` jest nowszy niż `plik2` |
| `plik1 -ot plik2` | jeżeli `plik1` jest starszy niż `plik2` |
| Porównywanie napisów | prawdziwe jeżeli |
| `-z napis` | `napis` ma długość 0 |
| `-n napis` | `napis` ma długość niezerową |
| `napis1 = napis2` | `napis1` jest identyczny z `napisem2` |
| `napis1 != napis2` | `napis1` jest różny od `napisu2` |
| Operatory arytmetyczne | prawdziwe jeżeli wartości są |
| `wart1 -eq wart2` | są sobie równe |
| `wart1 -ne wart2` | różne |
| `wart1 -lt wart2` | 1. mniejsza od 2. |
| `wart1 -le wart2` | 1. mniejsza lub równa 2. |
| `wart1 -gt wart2` | 1. większa od 2. |
| `wart1 -ge wart2` | 1. większa lub równa 2. |

Tablica 1: Operatory polecenia *test*.

zmiennych znakami cudzysłowu.

Instrukcja *case* jest złożoną instrukcją warunkową. Ma następującą składnię:

```
case $ZMIENNA in
    WARTOŚĆ1)
        instrukcja
        ;;
    WARTOŚĆ2)
        instrukcja
        ;;
     ...
    WARTOŚĆN)
        instrukcja
        ;;
    *)
        instrukcja
        ;;
esac
```

Gwiazdka w miejscu ostatniej wartości oznacza wszystkie pozostałe wartości i odpowiada instrukcji *default* w ANSI C.

# 5. Pętle

Podczas pisania skryptów w sh można wykorzystywać 3 podstawowe typy pętli.

Pierwszym z nich jest pętla *for*, której składnia ma postać:

```
for ZMIENNA in LISTA_WARTOŚCI
do
    instrukcje
done
```

W kolejnych cyklach wykonywania pętli `ZMIENNEJ` są przypisywane kolejne elementy z `LISTY_WARTOŚCI`. Poniższe przykłady pokazują różne możliwości wykorzystania pętli:

```
$ for i in 1 2 3 ; do echo "Zmienna i ma wart.: ${i}" ; done
Zmienna i ma wart.: 1
Zmienna i ma wart.: 2
Zmienna i ma wart.: 3
$ for i in /etc/hosts* ; do echo "Plik: ${i}" ; done
Plik: /etc/hosts
Plik: /etc/hosts.allow
Plik: /etc/hosts.deny
Plik: /etc/hosts.equiv
```

Dwa kolejne typy pętli to *while* i *until*:

```
while [ warunek ]
do
    instrukcje
done

until [ warunek ]
do
    instrukcje
done
```

We wszystkich 3 pętlach można używać poleceń *break* i *continue* w celu przerwania pętli, lub kontynuowania kolejnego cyklu pętli.

# 6. Funkcje

Przy bardziej złożonych skryptach bardzo przydaje się dodatkowy mechanizm strukturalizacji kodu, a mianowicie funkcje. Definicja funkcji może mieć dwojaką postać:

```
function nazwa_funkcji()
{
    instrukcje
}
```

```
nazwa_funkcji()
{
    instrukcje
}
```

Funkcje są uruchamiane przez tę samą powłokę co sam skrypt, w przeciwieństwie do zewnętrznych poleceń uruchamianych przez mechanizm *$()*. Podczas uruchamiania funkcji zmienne `$N` wskazują na argumenty z jakimi została wywołana funkcja, zmienna `$#` wskazuje na liczbę tych argumentów, a zmienna `$0` nie zmienia swojej wartości.

Wewnątrz funkcji można definiować zmienne lokalne, to znaczy takie, których zasięg nazw obowiązuje tylko w funkcji. Reszta zmiennych jest dzielona pomiędzy główną częścią skryptu a funkcją. Powrót z funkcji można wymusić poprzez instrukcję *return*.

Opisane powyżej mechanizmy można pokazać na przykładzie skryptu:

```
#!/bin/sh

stan()
{
    local kat=$(ls ..)
    echo "Plik ${1}"
    if [ "${1}" != "istnieje" ]
    then
        echo -n "Katalog wyżej niż ${PWD}: "
        echo ${kat}
    fi
    return
}

kat=$(ls)
echo -n "Katalog bieżący (${PWD}): "
echo $kat

echo "Podałeś plik: \"${1}\""
if [ -e "$1" ] ; then
    stan "istnieje."
else
    stan "nie znaleziony!"
fi
```

Skrypt demonstruje przekazywanie argumentów do funkcji, zmienne lokalne i globalne, a także instrukcję warunkową *if*.

```
$ ./t3 h
Katalog bieżący (/tmp/a/a): e f g t3
Podałeś plik: "h"
Plik nie znaleziony!
Katalog wyżej niż /tmp/a/a: a b c d
```

# 7. Przydatne polecenia

Ponieważ sh nie ma dodatkowych bibliotek funkcji, jak to ma na przykład miejsce w przypadku Perla, intensywnie wykorzystuje zewnętrzne polecenia systemowe. Te polecenia są przykładowo

używane do operowania na napisach. W Tabeli 2 są pokazane niektóre z najczęściej wykorzystywanych tego typu narzędzi.

| Polecenie | Opis | Przykład |
|---|---|---|
| *basename* | podaje nazwę pliku bez nazwy katalogu i ew. bez rozszerzenia | `basename /tmp/a.png .png` |
| *dirname* | podaje ścieżkę dostępu do pliku | `dirname /tmp/a.png` |
| *grep* | umożliwia zaawansowane wyszukiwanie wzorców (wyrażeń regularnych) w pliku | `grep wyrażenie plik` |
| *cut* | pozwala na wycinanie podanego zakresu znaków, lub pól rozdzielonych zdefiniowanym znakiem | `echo "Users $(cut -d: -f1 /etc/passwd)"` |
| *head* | wypisuje zadaną liczbę linii z początku pliku | `head -n 1` |
| *tail* | wypisuje zadaną liczbę linii z końca pliku | `tail -n 1` |
| *sort* | sortuje w porządku alfabetycznym plik | `sort plik` |
| *sed* | zaawansowany edytor operujący na strumieniu danych wejściowych; pozwala na przykład zamianę jednego napisu na inny | `sed s/stary napis/nowy napis/` |

Tablica 2: Narzędzia operujące na napisach

# 8. Podsumowanie

Na koniec warto zauważyć, że możliwości programowaia skryptów sh są nieco większe w przypadku powłoki GNU bash. Jest ona wyposażona w zawansowane mechanizmy i polecenia wbudowane nie występujące w innych powłokach. Zainteresowani mogą się z nimi zaznajomić w podręczniku systemowym *bash(1)*.

Artykuł prezentuje na przykładach najważniejsze elementy i mechanizmy języka sh. Ich umiejętne łączenie i wykorzystanie wymaga rzecz jasna nieco praktyki, lecz pozwala na pisanie dość złożonych skryptów, będących nieodzowną pomocą w codziennej pracy administratora.