RWSH - Really Weird Shell

Un shell alternativ pentru sisteme UNIX-like

Tudor-Ioan Roman

Cuprins

Introducere - shell-ul

Codul poate fi găsit pe https://git.sr.ht/~tudor/rwsh.

Issue tracker: https://todo.sr.ht/~tudor/rwsh.

Shell-ul, sau linia de comandă, este interfața textuală a unui sistem de operare. Prin acesta, utilizatorul poate să execute programe sub formă de *comenzi*, sau să execute *script-uri*. Aceste comenzi pot fi ori scrise în timpul execuției shell-ului (modul interactiv), sau pot fi înșiruite într-un fișier, numit *script*. În cazul cel din urmă, shell-ul are rolul de interpretor pentru un limbaj special de programare.

Programele executate de către shell pot fi înlănțuite. În mod normal, un program citește date de la tastatură (intrarea standard) și scrie date pe ecran (ieșirea standard). Într-un lanț de programe, primul program citește de la tastatură, iar rezultatul care ar trebui scris pe ecran este în schimb transmis ca date de intrare pentru următorul program, ca și cum ar fi fost scrise la tastatură. Acest lanț se numește *pipe*, iar operația de înlănțuire se numește *piping*.

Comenzile date shell-ului pot coopera astfel pentru a ajunge la un rezultat final. Utilizatorul poate să prelucreze date și să administreze sistemul cu o eficiență ridicată.

Exemple de astfel de shell-uri sunt GNU `bash` (Bourne Again Shell), `csh` (C Shell), `ksh` (Korn Shell), `zsh` (Z Shell), `fish` (Friendly Interactive Shell) etc.

Publicul țintă al acestor programe este format din administratori de sistem, ingineri software și power useri.

> În blocurile de cod care urmează, textul precedat de simbolul '#' formează comentariile.

Prelucrarea textului

Pe platformele descendente din UNIX, precum Linux și MacOS, programele care operează în modul text se bazează pe schimbul și prelucrarea informației de tip *text simplu*, fără alte formate binare (ca pe Windows). Fișierele de configurație pentru programe sunt exprimate în text simplu, cât și documentele cum ar fi manualul sistemului, care sunt exprimate într-un limbaj special, spre

deosebire de programe precum *Microsoft Word* care stochează documentele într-un format binar.

Exemplu de cooperare între programe: afișarea tuturor fișierelor dintr-un folder care conțin `infoeducatie` în nume, în ordine inversă:

```
ls | grep infoeducatie | sort -r
```

Această linie de comandă conține trei comenzi: `ls`, `grep infoeducatie` și `sort -r`. Cele trei comenzi sunt legate între ele prin operatorul `|` (pipe). Operatorul pipe capturează rezultatul comenzii din stânga, și, în loc să îl afișeze, îl oferă ca date de intrare programului din dreapta, ca și cum ar fi datele date de la tastatură.

Comanda `ls` afișează fișierele din directorul curent. `grep infoeducatie` afișează șirurile de caractere de la citire care conțin subșirul "infoeducatie", iar `sort -r` ordonează în ordine lexicografică inversă. Când comanda `ls` "afișează" fișierele, textul este dat ca intrare comenzii `grep infoeducatie`, iar aceasta la rândul ei furnizează comenzii `sort -r` ca date de intrare fișierele care in denumirea lor conțin subșirul "infoeducatie". La final, rezultatul comenzii `sort -r` este afișat pe ecran.

Acest mod de funcționare al shell-ului (*piping*) se bazează pe faptul ca majoritatea programelor operează pe text, furnizând, filtrând și prelucrând text. `ls` furnizează text, `grep` filtrează, iar `sort` prelucrează (ordonează). Programele care operează pe text includ și uneltele de administrare a sistemului. Prin metoda *piping*-ului se pot realiza programe (*script-uri*) eficiente.

RWSH - prin ce diferă

RWSH include propriile facilități de prelucrare a textului, care operează într-un mod inedit, diferit de oricare alt shell sau program de prelucrare al textului, facilități inspirate din limbajul de prelucrare al textului folosit de `sam`, editorul de texte al sistemului de operare experimental *Plan 9*, dezvoltat în anii '80 de Laboratoarele Bell.

În mod tradițional, marea majoritate a programelor care operează pe text prelucrează datele linie cu linie. În unele cazuri, aceasta abordare poate fi ineficientă și pentru procesor, dar și pentru programator.

De asemenea, sintaxa RWSH este una curată, nouă, lipsită de neclaritățile din limbajele uzuale de shell-scripting, care au apărut din cauza nevoii de a păstra compatibilitatea cu versiuni mai vechi ale acestora. RWSH nu respectă standardul POSIX. Acesta poate fi un lucru bun, pentru că permite dezvoltarea unei sintaxe complet noi, dar și un lucru rău, atunci când compatibilitatea POSIX este dorită.

Expresiile regulate structurale (structural regular expressions)

RWSH folosește un sub-limbaj prin care poate fi exprimată structura textului pe care dorim să operăm. Un alt mecanism foarte important este cel al *expresiilor regulate* (regular expressions, pe scurt *regex*). Acestea sunt șiruri de caractere, exprimate într-un limbaj special, care definesc un *șablon de căutare*. Aceste expresii regulate, extinse cu facilități de descriere a structurii, dau naștere *expresiilor regulate structurale* (structural regular expressions).

Acest sub-limbaj include operații de prelucrare a textului, care pot fi combinate cu programele convenționale.

Exemplu: înlocuirea numelui "Tudor" cu "Ioan" într-un document.

```
cat document.txt |> ,x/Tudor/ c/Ioan/ |> ,p
```

Modul de funcționare

O comandă *pizza* este formată dintr-o *adresă* și o *operație*. Adresa este o expresie regulată structurală, iar operația este identificată printr-o literă și poate avea parametri. Aceste comenzi sunt înlănțuite prin operatorul `|>`, numit *operatorul pizza* (pentru că seamănă cu o felie de pizza). Adresa poate fi omisă, fiind folosită adresa ultimei comenzi, numită *dot*. Intern, adresa este o pereche de numere: poziția de început, și poziția de sfârșit, în caractere de la începutul fișierului. Dot este setată atunci când se specifică în mod explicit adresa unei comenzi, și la finalul execuției comenzii. De exemplu, comanda `c`, care înlocuiește textul situat la adresa *dot* cu textul dat ca parametru, setează la final *dot* ca adresa la care se află textul cel nou.

În continuare, voi ilustra exemplul de mai sus:

`cat document.txt` invocă programul `cat`, care afișează pe ecran conținutul

fișierului `document.txt`.

Rezultatul comenzii, fiind urmat de operatorul pizza (`|>`), conținutul fișierului, în loc să fie afișat pe ecran, va fi pasat comenzilor pizza care urmează.

Prima comandă, `,x/Tudor/ c/Ioan/` are adresa `,`, care se referă la datele de intrare în întregime. Operația este operația buclă, notată prin `x`. Ea primește doi parametri: primul este o expresie regulată, al doilea este o altă comanda. Comanda `x` executa comanda transmisă ca parametru pentru fiecare subșir care se potrivește cu expresia regulată dată. Comanda `c/Ioan/` schimbă subșirul cu textul "Ioan".

Următoarea comandă din șir este `,p`, care afișează textul dat în întregime.

Pe lângă operația `c`, mai există operațiile `a`, `d` și `i`, care adaugă după dot, șterge textul din dot și respectiv inserează înaintea dot-ului.

Inversul operației `x` este `y`, care execută o comandă pe subșirurile care nu se potrivesc cu expresia regulată.

O altă abilitate specială este cea de a executa comenzi *pizza* în paralel, adică se execută mai multe comenzi pe același text, iar rezultatele fiecăreia se aplică cumulat.

Exemplu practic: înlocuirea tuturor aparițiilor numelui "Tudor" cu "Andrei" și "Andrei" cu "Tudor":

```
cat text.txt |> ,x/Tudor|Andrei/ {
    g/Tudor/ c/Andrei/
    g/Andrei/ c/Tudor/
} |> ,p
```

Unde `text.txt` conține:

```
Tudor este prietenul lui Andrei. Tudor îi oferă
lui Andrei o bomboană. Alex vrea și el una, dar Tudor a rămas fără.
Andrei îi este recunoscător.
```

Programul va afișa pe ecran:

```
Andrei este prietenul lui Tudor. Andrei îi oferă
lui Tudor o bomboană. Alex vrea și el una, dar Andrei a rămas fără.
Tudor îi este recunoscător.
```

În cuvinte, programul de mai sus poate fi descris în următorul mod:

- Pentru fiecare apariție a cuvântului “Tudor” sau “Andrei”...
  - ... dacă este “Tudor”, atunci schimbă-l cu “Andrei”.
  - ... dacă este “Andrei”, atunci schimbă-l cu “Tudor”.
- Afișează rezultatul.

Comenzile aflate intre acolade sunt cele executate *în paralel*. Efectul fiecărei comenzi este înregistrat într-un jurnal sub formă de vector. Acestea sunt interclasate și la final efectele sunt aplicate.

Un alt exemplu este să vedem de câte ori și unde apare cuvântul “linux” în jurnalul sistemului:

```
dmesg |> ,x/linux/ {
    =
    +-p
}
```

Date de ieșire:

```
#183,#188
[    0.000000] Command line: \\vmlinuz-linux-zen rw root=UUID=dba92c4c-35bc-4b73-9
#288,#293
[    0.000000] Command line: \\vmlinuz-linux-zen rw root=UUID=dba92c4c-35bc-4b73-9
#16485,#16490
[    0.000000] Kernel command line: \\vmlinuz-linux-zen rw root=UUID=dba92c4c-35bc
#16590,#16595
[    0.000000] Kernel command line: \\vmlinuz-linux-zen rw root=UUID=dba92c4c-35bc
#32278,#32283
[    0.540171] pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giomett
```

Operația `=` afișează valoarea percepută a lui *dot*. Mai concret, va afișa poziția cuvântului “linux”. Adresa `+-` a comenzii de afișare `p` înseamnă să meargă o linie în față și una în spate față de adresa subșirului găsit. Această tehnică este folosită pentru a afișa toată linia pe care am găsit subșirul. Altfel, s-ar fi afișat doar cuvântul “linux”.

Exemplu mai complex: afișarea liniilor care conțin cuvântul “linux”, dar fără timpul evenimentelor (textul dintre paranteze pătrate de la începutul fiecărei linii):

```
dmesg |> ,x/^\[.*\] /d |> ,x/linux/ {
    =
    +-p
} | lolcat
```

Rezultatul va fi pasat comenzii `lolcat` pentru a fi afișat în culorile curcubeului.

Detalii legate de comenzile disponibile pot fi găsite in anexă.

Avantajele abordării RWSH

Integrarea uneltelor de prelucrare a textului în cadrul shell-ului este inevitabilă. Uneltele convenționale, precum `grep`, `sed`, `cut`, `tr` etc. sunt folosite în aproape orice *shell script* din cauza funcțiilor elementare pe care le prestează. Majoritatea shell-urilor moderne prezintă unele astfel de funcționalități tocmai pentru că sunt indispensabile și sunt prea lungi de scris. Priviți care este diferența dintre eliminarea sufixului numelui unui fișier în mod tradițional vs. cu ajutorul sintaxei speciale din cel mai popular shell, GNU *bash*:

```
# Avem funcția proceseaza_fisier, care primeste numele fara extensie,
# si variabila filename, care tine minte numele complet al fisierului.

# Acestea sunt cele trei modalitati de apelare a functiei cu numele fisierului
# fara extensie.

proceseaza_fisier "$(echo "$filename" | cut -d'.' -f1-)" # in mod traditional

proceseaza_fisier "$(cut -d'.' -f1- <<< "$filename")" # modul traditional,
                                                      # dar folosind o scurtatura
                                                      # specifica bash


proceseaza_fisier "${filename%.in}" # folosind sintaxa speciala de eliminare
                                    # a sufixului din bash
```

În prima metodă, metoda pur tradițională, trebuie să folosim două procese de sistem pentru a afla denumirea fără extensie (un proces pentru `echo` și

încă unul pentru `cut`). În cea de a doua, avem numai un singur proces, cel pentru `cut`, deoarece shell-ul are o sintaxa speciala pentru pasarea automata a unor date de intrare ca date de la tastatură (`<<<`).

Cea de a treia metodă nu folosește niciun proces.

Folosirea proceselor nu doar că încetinește mult programul, dar și este costisitoare pentru programatorul care scrie codul. Mai multă complexitate a codului poate duce la mai multe erori.

Un alt avantaj al abordării RWSH este că niciun alt program nu utilizează tehnica expresiilor regulate *structurale*, care fac procesarea textului mai eficientă și mai citibilă.

Funcționalitatea de limbaj de programare

Pentru moment, în limita timpului disponibil, am reușit să implementez variabilele, șirurile de caractere, operațiile matematice, blocurile de cod, blocurile decizionale (`if`), buclele (`while`) și două blocuri speciale pentru pattern matching: `switch` și `match`.

Variabilele

Valorile sunt atribuite variabilelor cu comanda `let`. Variabilele sunt declarate automat la atribuire.

```
let nume = Tudor
echo "Salut, $nume!" # Va afisa "Salut, Tudor!"

let nume = Andrei
echo "Salut, $nume!" # Va afisa "Salut, Andrei!"
```

Pentru a folosi o variabilă, numele ei va fi precedat de simbolul `$`.

Pentru a șterge variabila, se va folosi `let` cu flag-ul `-e` (erase): `let -e name`.

Notă: există o variabilă specială, numită `?`. Ea ține minte "exit code"-ul ultimei comenzi executate. Comanda precedentă se consideră executată cu succes dacă `?` va fi egal cu 0.

Variabilele pot avea un *scope* asemănător limbajelor moderne de programare, spre deosebire de POSIX shell. Când o variabilă este creată, ea va fi atribuită blocului de cod în care se află. Atunci când atribuim o valoare nouă unei variabile în interiorul unui bloc inferior de cod, variabila va rămâne în blocul superior. Putem crea o nouă variabilă cu același nume în blocul inferior de cod cu flag-ul `-l` (local).

Exemple:

Păstrarea scope-ului:

```
let name = Tudor
echo $name # Tudor
{
    echo $name # Tudor
    let name = Ioan
    echo $name # Ioan
}
echo $name # Ioan
```

Crearea unui nou scope local:

```
let name = Tudor
echo $name # Tudor
{
    echo $name # Tudor
    let -l name = Ioan
    echo $name # Ioan
}
echo $name # Tudor
```

Atribuirea unui scope inferior atunci când variabila nu există:

```
echo $name # nimic
{
    let name = Tudor
    echo $name # Tudor
}
echo $name # nimic
```

Vectorii (array)

Toate variabilele sunt stocate drept vectori. Când declarăm o variabilă simplă, se declară de fapt un vector cu un element.

Putem declara un vector cu mai multe elemente:

```
let fructe = [ mere rosii prune ]
echo mie îmi place să mănânc $fructe[2]
```

Dacă folosim un vector fără un index, fiecare element va fi tratat ca un argument separat (array expansion).

```
echo $fructe # echo primeste trei parametrii
```

Putem folosi asta pentru a crea vectori pe baza altora:

```
let numere = [ 1 2 3 ]
let numere_speciale = [ 5 $numere 8 9 4 ] # elementele vor fi 5 1 2 3 8 9 4
```

Expresiile glob, precum `*.mp4`, sunt tratate și ele ca argumente diferite:

```
let list_of_movies = [ *.mp4 ]
let list_of_music = [ *.ogg *.flac ]
```

Dacă folosim vectorul fără un index, dar în interiorul unui șir de caractere cu ghilimele, elementele lui vor fi tratate ca un singur parametru, cu spații între ele:

```
echo "$fructe" # echo primeste un parametru
```

Dacă numele vectorului are sufixul `PATH` (exemple: `PATH`, `MANPATH`, `LD_LOAD_PATH` etc.), în scrierea lui ca șir de caractere, elementele nu vor fi delimitate de spații, ci de doua puncte (“:”):

```
echo "$PATH" # afișează /bin:/usr/bin:/usr/local/bin etc
echo $PATH   # afișează /bin /usr/bin /usr/local/bin,
             # dar fiecare path va fi perceput ca un
             # argument separat de catre echo
```

Variabilele de mediu (environment variable) care au sufixul “PATH”, precum cele enumerate mai sus, vor fi automat convertite în vectori.

Comanda `let`

Pe lângă atribuirea de valori și crearea variabilelor în scope-uri noi, `let` știe și să creeze variabile de mediu (environment variables):

```
let -x variabila_de_mediu = "o valoare" # -x vine de la eXport

# echivalentul bash:
export variabila_de_mediu="o valoare"
```

Ștergerea variabilelor:

```
let -e variabila # sterge variabila normala
let -xe variabila # sterge variabila de mediu
```

Pe lângă `=`, `let` are și alți operatori pentru diverse scurtături:

```
let i += numar # echivalent cu let i = $(calc $i + numar)
# exista si -=, *=, /=, %=

let array ++= element # adauga "element" in array
# echivalent cu let array = [ $array element ]

let array ::= element # adauga "element" la inceput in array
# echivalent cu let array = [ element $array ]

let array ++= [ element1 element2 ] # adauga mai multe elemente
let array ::= [ element1 element2 ] # adauga mai multe elemente la inceput
```

Șirurile de caractere

Parametrii comenzilor date sunt exprimați ca șiruri de caractere separate prin spații. Pentru a putea folosi șiruri de caractere cu caractere speciale și spații, acestea vor fi înconjurate de ghilimele (`"`) sau apostrofuri (`'`).

Apostrofurile diferă de ghilimele prin faptul că în șirurile de caractere cu apostrofuri, cuvintele precedate de `$` nu vor fi tratate ca variabile.

```
let nume = Tudor
echo Salut, $nume! # Va afisa "Salut, Tudor!"
                   # Comanda echo primeste doi parametri: "Salut," si "Tudor!"

echo "Salut, $nume!" # Va afisa tot "Salut, Tudor!"
                     # De data asta, echo primeste un singur parametru:
                     # "Salut, Tudor!"

echo 'Salut, $nume!' # Va afisa "Salut, $nume!"
                     # echo primeste un singur parametru.
```

Șirurile de caractere pot fi lipite pentru a forma unul singur, respectând regulile fiecăruia:

```
let nume = Tudor

echo Salut", $nume"'!' # Va afisa tot "Salut, Tudor!"
                       # Comanda echo primeste un singur parametru
```

Șirurile de caractere simple și cele între ghilimele pot de asemenea să conțină rezultatul unei comenzi:

```
echo Este ora $(date +%H:%M) # Va afisa "Este ora 11:27"
                             # Comanda echo primeste 3 parametri
```

Important: valorile variabilelor / elementelor vectorilor, indiferent de cate spații conțin, vor fi tratate întotdeauna ca un singur parametru. RWSH nu necesită "quoting"-ul variabilelor.

```
# bash:
filename="video haios.mp4"
rm $filename
# rm: cannot remove 'video': No such file or directory
# rm: cannot remove 'haios.mp4': No such file or directory

# rwsh:
let filename = "video haios.mp4"
rm $filename # merge fara nicio problema, cum ne-am aștepta
rm $(echo video haios.mp4) # merge și așa
```

În cazul în care vreți ca o variabilă sa fie "extinsă" ca în bash, aveți două opțiuni:

1. Declarați variabila ca vector: `let filename = [ video haios.mp4 ]`
2. Folosiți comanda `eval`: `eval rm $filename`

Notă: dacă un șir de caractere simplu (fără ghilimele sau apostrofuri) conține la început o cale de fișier care începe cu caracterul `~`, tilda va fi înlocuită de calea către directorul utilizatorului. Exemplu:

```
ls ~/src # Afiseaza conținutul folder-ului /home/tudor/src
ls ~altuser/dir # Afișează conținutul folder-ului /home/altuser/dir
```

Blocurile de cod și blocurile `if` și `while`

Sintaxa pentru blocurile `if` și `while` este `if (condiție) comandă_de_executat`, respectiv `while (condiție) comandă_de_executat`.

Condiția este o comandă. Dacă "exit code"-ul comenzii din condiție este 0, condiția este validă, iar comanda se va executa. În cazul lui `while`, comanda se va executa cât timp condiția se evaluează cu codul 0.

Dacă vrem să executăm mai multe comenzi, vom folosi blocul de cod, scris între acolade:

```
if (condiție) {
    o_comanda
    a_doua_comanda
}
```

Putem și să executăm cod dacă condiția nu se verifică, folosind blocul `else`, și să punem condiții în plus cu `else if`:

```
if (condiție) fa_ceva
else if (alta_condiție) fa_altceva
else nu_avem_incotro
```

Cum condiția este o comandă, putem sa folosim pipe-uri, operatori pizza, etc.

Condiția poate fi negată cu operatorul `!`: `if (! condiție) fa_ceva`.

Mai pot fi folosiți și operatorii logici `||` și `&&`, chiar și în afara condiției, ca in bash.

Operațiile matematice

Operațiile matematice se fac cu comanda `calc`. Putem stoca rezultatul într-o variabila astfel:

```
let a = 2
let b = 3
let c = $(calc $a + $b)

echo "Rezultatul este $c" # Va afisa "Rezultatul este 5"
```

Incrementarea și decrementarea variabilelor se poate face direct cu `let`:

```
let i = 0
let n = 10
while ([ $i -lt $n ]) {
    fa_ceva
    let i += 1
}
```

Pattern matching: `switch` si `match`

Blocul `switch` arată în felul următor:

```
switch $valoare
    /pattern_1/ fa_ceva
    /pattern_2/ fa_altceva
    ...
    /pattern_n/ nu_stiu_ce_sa_mai_fac
    // fallthrough
end
```

`switch` evaluează pattern-urile (care sunt regex-uri) în ordine și excută comanda asociată primului regex care se potrivește cu valoarea dată.

Notă: `switch` ar putea fi îmbunătățit cu condiții care nu țin doar de natura șirului de caractere. Momentan nu există nicio modalitate de a folosi `switch` pe intervale numerice, de exemplu.

Exemplu:

```
switch $status
    /\[ERROR\] (.*)/ echo s-a petrecut o eroare: $1
    /\[WARNING\] (.*)/ echo avertizare: $1
end
```

Putem folosi `switch` și ca să împărțim un text în mai multe câmpuri:

```
switch $status
    /\[(?P<type>.*)\] (.*)/ echo got event type \"$type\" with message: $2
end
```

`match` este similar comenzii SRE `x`: citește de la `stdin` și pentru fiecare subșir care se potrivește cu un regex, execută comanda asociată. Dacă un subșir se potrivește cu mai multe regex-uri, se execută comenzile asociate tuturor regex-urilor cu care se potrivește. `match` este echivalent-ul pattern-urilor din `awk`. Totuși, `awk` execută comenzile pe întreaga linie care se potrivește cu regex-ul, în timp ce RWSH ia strict textul potrivit.

Exemplu: afișează textul dintre ghilimele

```
echo 'un text cu "ghilimele" in el. si '"'apostrofuri'" |
    awk '/".*"/ { print "ghilimele", $1 }'"
         /'.*'/"' { print "apostrofuri", $1 }'
```

Va afișa "ghilimele un apostrofuri un", deoarece `print $1` afișează primul cuvânt de pe linia cu ghilimele ("un"), nu textul dintre ghilimele.

Următoarea este varianta corectă:

```
echo 'un text cu "ghilimele" in el. si '"'apostrofuri'" | awk '{
    for (i = 1; i <= NF; i++) {
        if (match($i, /".*"/)) {
            print "ghilimele", $i
        } else if (match($i, '"/'.*'/"')) {
            print "apostrofuri", $i
        }
    }
}'
```

Codul acesta nici măcar nu mai folosește pattern-urile din `awk`.

Varianta RWSH este mai simplă:

```
echo 'un text cu "ghilimele" in el. si '"'apostrofuri'" |
    match
        /"(.*)"/ echo ghilimele $1
        /'(.*)'/ echo apostrofuri $1
    end
```

Această variantă nu doar că este mai ușoară de înțeles, ba chiar extrage textul dintre ghilimele / apostrofuri.

Detalii tehnice

Singura cerință de sistem este un sistem de operare UNIX-like, precum Linux, MacOS, FreeBSD etc.

Programul este scris în limbajul de programare Rust, un limbaj similar cu C++ care pune accent pe corectitudinea programului și a modelului de memorie. Cum shell-ul este un program cheie în orice sistem de calcul, acesta nu trebuie să aibă erori de memorie sau probleme de securitate (a se vedea: Shellshock). Rust de asemenea vine cu sintaxă și librărie standard modernă, făcând experiența de programare mai apropiată de un limbaj de programare "ușor", precum Python sau Go.

Pentru a asigura siguranța codului și sănătatea minții, folosesc **teste automate** pentru a detecta bug-uri în cod. Acestea se execută cu comanda `cargo test` din Rust și cu script-ul `run_examples.sh` din folder-ul `examples`.

Aceste teste sunt executate automat la fiecare `git push` într-un sistem tip CI (continous integration) la adresa https://builds.sr.ht/~tudor/rwsh. La fiecare push, serviciul compilează codul, verifică ca fiecare fișier să conțină antetul pentru licența GPL și execută testele. Dacă testele merg bine, codul este împins automat pe GitHub. Dacă testele merg bine, codul este împins automat pe GitHub.

Librăriile folosite includ `nix` pentru funcțiile de bibliotecă pentru sistemul de operare, `regex` pentru expresiile regulate *simple*, și `calculate` pentru funcția de calculator.

To do

Am implementat funcțiile cele mai importante importante pentru a obține o soluție consistentă pentru prezentare. Funcționalități care vor fi implementate:

- Buclă cu iterator (`for`)
- Funcții
- Variabile hash map
- Execuția comenzilor din SRE
- Job control (foarte complex)
- Ticket-ul #4 (pe `https://todo.sr.ht/~tudor/rwsh`)

Arhitectura aplicației

Codul este împărțit în mai multe module de tip "crate", specifice limbajului de programare Rust. Codul executabilului este în modulul executabil, care se folosește de modulul "lib". Modulul "lib" este mai departe împarțit în:

Modul	Descriere
`builtin`	Conține codul fiecărui program "builtin", precum `calc`, `let`, `eval`, `exit` etc.
`parser`	Se ocupă de transformarea textului în arbore de sintaxă.
`shell`	Conține cod specific funcționalității de shell, precum execuția codului, stocarea variabilelor.
`sre`	Codul din spatele expresiilor SRE.
`task`	Fiecare activitate pe care o duce shell-ul, fie ea execuția comenzilor, procesarea șirurilor de caractere, execuția blocurilor de cod, blocurilor `if`, `while`, `switch`, `match`, pipe-uri etc.
`tests`	Cod comun testelor.
`util`	Conține codul responsabil cu citirea propriu-zisă a liniilor de cod de la tastatură sau din fișier. Conține o interfață abstractă pentru asta.

Ghid de instalare

- Instalați Rust folosind `rustup`.
- Clonați repo-ul: `git clone https://git.sr.ht/~tudor/rwsh && cd rwsh`
- Compilați: `cargo build --release`
- Executați: `cargo run --release`

Anexă - comenzi disponibile

p

Afișează conținutul de la adresa *dot*. Nu acceptă parametri.

Exemplu: Afișează tot conținutul.

```
|> ,p
```

a, c, i și d

Adaugă după, înlocuiește, inserează înaintea sau șterge conținutul *dot*.

Exemple:

```
|> 2a/O nouă linie\n/ # adaugă o nouă linie între liniile 2 și 3

|> /To Do/ c/Done/ # înlocuiește statutul unei notițe

|> 2i/O nouă linie\n/ # adaugă o nouă linie între liniile 1 și 2

echo "text de șters" |> /de șters/d |> ,p # afișează "text"
```

g și v

Execută o comandă pe textul de la adresa *dot* dacă respectivul text se potrivește cu un regex.

`v` este opusul comenzii `g`: Execută comanda dacă textul **nu** se potrivește.

Exemplu:

Dacă primul cuvânt de la poziția *dot* este "Tudor", afișează poziția.

Comenzile `g` și `v` sunt folosite în general împreună cu `x` și `y`, nu pe cont propriu.

```
|> /\b.+\b/ g/Tudor/ =
```

Înlocuiește toate aparițiile cuvântului "vi" cu "emacs". Dacă un cuvânt conține "vi", el nu va fi alterat. Alternativ, se poate folosi metacaracterul `\b` în regex.

```
|> ,x/vi/ v/.../ c/Emacs/
# echivalent cu
|> ,x/\bvi\b/ c/Emacs/
```

x și y

Execută o comandă pentru fiecare subșir care se potrivește cu un regex în cadrul textului de la adresa *dot*.

Comanda `y` este opusul comenzii `x`: execută comanda pe textul situat **între** subșirurile care se potrivesc cu regex-ul, în cadrul textului de la adresa *dot*.

Exemplu:

Înlocuiește toate aparițiile lui "Tudor" cu "Ioan".

```
|> ,x/Tudor/ c/Ioan/
```

Înlocuiește toți identificatorii numiți `n` într-un cod sursă, cu `num`, fără să atingă șirurile de caractere, aflate între ghilimele sau apostrofuri.

```
|> ,y/".*"|'.*'/ x/\bn\b/ c/num/
```

=

Afișează poziția adresei *dot* în caractere de la începutul fișierului.

Exemplu:

Va afișa `#8,#13`.

```
echo "eu sunt Tudor" |> /Tudor/=
```

Adrese

Extras din manualul editorului `sam(1)`:

Addresses

An address identifies a substring in a file. In the following, 'character n' means the null string after the n-th character in the file, with 1 the first character in the file. 'Line n' means the n-th match, starting at the beginning of the file, of the regular expression `.*\n?`. All files always have a current substring, called dot, that is the default address.

Simple Addresses

`#n` The empty string after character n; `#0` is the beginning of the file.

`n` Line n; 0 is the beginning of the file.

`/regexp/`; `?regexp?` The substring that matches the regular expression, found by looking toward the end (/) or beginning (?) of the file, and if necessary continuing the search from the other end to the starting point of the search. The matched substring may straddle the starting point. When entering a pattern containing a literal question mark for a backward search, the question mark should be specified as a member of a class.

`0` The string before the first full line.

`$` The null string at the end of the file.

`.` Dot.

Compound Addresses

In the following, `a1` and `a2` are addresses.

`a1+a2` The address `a2` evaluated starting at the end of `a1`. `a1-a2` The address `a2` evaluated looking in the reverse direction starting at the beginning of `a1`. `a1,a2` The substring from the beginning of `a1` to the end of `a2`. If `a1` is missing, `0` is substituted. If `a2` is missing, `$` is substituted. `a1;a2` Like `a1,a2`, but with `a2` evaluated at the end of, and dot set to, `a1`.

The operators `+` and `-` are high precedence, while `,` and `;` are low precedence.