



ZSK23

Ezoteryczny język programowania i jego interpreter

STRESZCZENIE

Dokumentacja projektu na zaliczenie przedmiotu „Programowanie” w Technikum Komunikacji im. Hipolita Cegielskiego w Poznaniu

Krzysztof Łuczka

Spis treści

Obsługa interpretatora	2
Struktura	2
Operacje na stosie.....	2
Operacje na ciągach znaków i interakcja z użytkownikiem	4
Struktura ciągu znaków	4
Operacje wejścia.....	5
Operacje matematyczne	6
Odwrotna notacja polska.....	6
<i>calc</i> oraz <i>~calc</i>	7
Operatory matematyczne.....	7
Instrukcje warunkowe i pętle	8
Instrukcje warunkowe	8
Pętla warunkowa	9
Pętla stała.....	10
Pętla mieszana	10
Pozostałe polecenia	11
Przyszłościowe plany rozwoju projektu.....	11

Obsługa interpretatora

Interpretator obsługuje pliki tekstowe z kodem ZSK23. Rozszerzenie pliku nie ma najmniejszego znaczenia, jednakże preferowanym jest *.zsk. Podstawowym poleceniem jest *help*, które przedstawi krótki opis wszystkich możliwych komend obsługiwanych przez interpretator.

Struktura

Po wykonaniu polecenia *load <ścieżka>* interpretator załaduje plik tekstowy z kodem. Cały kod jest analizowany przez preprocesor, który w celach optymalizacyjnych zamienia linijki na wykonywalne komendy wewnątrz interpretatora. Dzięki temu program może być wykonywany znacznie szybciej, gdyż analizowanie tekstu i wykonywanie poleceń na bieżąco jest o wiele bardziej kosztowne dla komputera.

Polecenie *debug* pozwala zmienić flagę debugowania. Po włączeniu tej opcji program w trakcie interpretacji będzie zwracał informacje na temat błędu oraz jego pozycji w kodzie. Przed uruchomieniem wyświetli ile pamięci interpretatora wykorzystuje załadowany program. W trakcie interpretacji co linijkę będzie wyświetlał ile pamięci zajmuje stos.

Aby uruchomić załadowany program, należy wpisać polecenie *run*.

Operacje na stosie

Język ZSK23 opiera wszystkie działania na stosie 4 bajtowych liczb całkowitych. Polecenie *push* pozwala wrzucić liczbę na stos, a polecenie *pop* pozwala na zdjęcie jej ze stosu.

```
1 push 10
2 push 11
3 push 12
4 pop
5 push 13
6 stack
```

Powyższy kod wyświetli na ekranie wynik *13 11 10*. Użyta komenda *stack* pozwala na wyświetlenie zawartości stosu oddzielonego spacjami. Jeżeli zaraz po poleceniu *stack* podamy liczbę, wyświetli ona daną ilość liczb ze stosu.

```
1 push 10
2 push 11
3 push 12
4 pop
5 push 13
6 stack 2
```

W tym przypadku zostanie wyświetlone *13 11*.

Polecenie *copy* pozwala skopiować dowolną ilość kolejnych elementów ze stosu i wrzucenie ich z tą samą kolejnością z powrotem na stos.

```
1 push 1
2 push 2
3 push 3
4 push 4
5 push 5
6 copy 3
7 stack
```

Powyższy kod wyświetli *5 4 3 5 4 3 2 1*. Jeżeli nie podalibyśmy liczby przy *copy*, to domyślnie przyjmie wartość 1.

Kolejnym poleceniem manipulującym stos jest *reverse*. Odwraca on stos – pierwszy element jest ostatnim itd.

```
1 push 1
2 push 2
3 push 3
4 stack
5 reverse
6 stack
```

Powyższy kod wyświetli *3 2 1 1 2 3*. Jeżeli po *reverse* podamy liczbę, tylko część stosu zostanie odwrócona.

```

1 push 1
2 push 2
3 push 3
4 push 4
5 push 5
6 reverse 3
7 stack

```

Wynikiem tego programu będzie 3 4 5 2 1.

Polecenie *size* wrzuci wartość liczbową reprezentującą wielkość stosu na stos. Liczba ta jest ilością innych wartości na stosie z wyłączeniem jej samej.

```

1 push 32
2 push 16
3 push 8
4 push 4
5 push 2
6 size
7 stack 1

```

Powyższy kod wyświetli liczbę 5 pomimo faktu, że na stosie jest 6 elementów.

Operacje na ciągach znaków i interakcja z użytkownikiem

Struktura ciągu znaków

Ciągi znaków są przechowywane na stosie. Język ZSK23 przewiduje następującą metodę przechowywania ciągów znaków:

- Najmłodszy element stosu – długość całego ciągu znaków (z wyłączeniem owego elementu)
- Pozostałe – dowolne znaki zapisane w kodzie ASCII

Polecenie *print* właśnie w taki sposób wyświetla znaki na ekranie. Pobiera najmłodszy element stosu i wykorzystuje go jako ilość kolejnych znaków do wyświetlenia.

Polecenie *string* natomiast wrzuca na stos dowolny ciąg znaków – wszystko co znajdzie się w tej samej linii zostanie zinterpretowane jako jeden ciąg. *string* zawiera jedną bardzo ważną funkcję, którą wizualizuje ten przykład: założmy że ręcznie wpiszemy *abc* na stos.

```

1 push 97    = 'a'
2 push 98    = 'b'
3 push 99    = 'c'
4 push 3      długość
5 print

```

Powyższy kod wyświetliłby ciąg *cba*. Wszystko dzięki właściwościom stosów. To co zostało wrzucone jako pierwsze, zostanie zdjęte jako ostatnie; to co zostanie wrzucone jako ostatnie, zostanie zdjęte jako pierwsze. Komenda *string* automatycznie odwraca ciąg znaków tak, aby polecenie *print* wyświetliło go poprawnie.

Jeżeli zaraz po *print* podamy liczbę, to zostanie wyświetlona dana ilość znaków w kodzie ASCII. Ważne jest, że tak jak poprzednio najmłodszy element służył jako długość ciągu, tak teraz jest po prostu pomijany.

```

1 string abcdefgh
2 print 4

```

Powyższy kod wyświetli ciąg *abcd*.

Aby stworzyć nową linię, wykorzystamy znak tyldy. Oto przykład użycia.

```

1 string Witaj w klubie!~Mamy dla ciebie niespodzianke!
2 print

```

ZSK23 Interpreter

```

Witaj w klubie!
Mamy dla ciebie niespodzianke!

```

Operacje wejścia

Programista ma dostęp do dwóch poleceń, *getstring* oraz *getinteger*. To pierwsze działa identycznie jak *string*, z tym że program będzie czekał aż użytkownik wpisze dowolny ciąg. Tak samo *getinteger* – program będzie czekał aż użytkownik wpisze dowolną liczbę całkowitą. Analogicznie, obydwa wejścia przechowywane są na stosie.

ZSK23 Interpreter

```

Podaj swoje imie > Krzysztof
Czesc Krzysztof! Podaj ile masz lat > 100
Masz 100 lat.

```

```

1 string Podaj swoje imie >
2 print
3 string !
4 getstring
5 string Czesc
6 print
7 print
8 print
9 string Podaj ile masz lat >
10 print
11 string lat.
12 getstring
13 string Masz
14 print
15 print
16 print

```

Operacje matematyczne

Odwrotna notacja polska

Język ZSK23 w celu ułatwienia pracy na stosie korzysta z RPN – tytułowej odwrotnej notacji polskiej. Normalnie gdybyśmy chcieli policzyć obwód trójkąta (przykładowo egipskiego) użylibyśmy wyrażenia $3 + 4 + 5$. Aby policzyć obwód wykorzystując RPN, użylibyśmy wyrażenia $3\ 4\ 5\ ++$ lub $3\ 4\ +\ 5\ +$. Podawane liczby wrzucane są na stos, a operatory inicjują działanie matematyczne na kolejnych liczbach ze stosu. Poleceniem do obliczania wyrażeń w ZSK23 jest *calc*.

Zauważ, że poniższe przykłady zwrócą ten sam wynik.

```

1 push 3
2 push 4
3 push 5
4 calc + +
5 stack 1

```

```

1  push 3
2  calc 4 5 + +
3  stack 1

```

Ich celem było ukazanie, że liczba w poleceniu *calc* działa jak wywołanie polecenia *push*.

calc oraz *~calc*

ZSK23 oprócz *calc* udostępnia jeszcze polecenie *~calc*. Postawienie tyldy przed poleceniem różni się tylko tym, że *calc* pracuje na prawdziwym stosie, a *~calc* na jego kopii.

Aby zrozumieć różnice między *~calc* oraz *calc*, policzmy obwody trójkąta egipskiego i sprawdźmy jak wygląda stos.

```

1  push 3
2  push 4
3  calc 5 + +
4  stack

```

Powyższy kod wyświetli 12.

```

1  push 3
2  push 4
3  ~calc 5 + +
4  stack

```

Natomiast ten wyświetli 12 4 3. Różnica polega na tym, że wewnątrz polecenia *~calc* domyślnie tworzona jest identyczna kopia stosu, a wynik działania jest wrzucany na główny stos. W związku z tym liczby 3 i 4 wrzucony na główny stos pozostają nieruszone. Gdybyśmy dołożyli polecenie *push 5*, a następnie *~calc + +*, to program wyświetliłby 12 5 4 3.

Operatory matematyczne

Poniższa tabelka ukazuje matematyczne operatory i wyniki ich działania.

operator	działanie
+	<i>starszy element + młodszy element</i>
-	<i>starszy element - młodszy element</i>
/	<i>starszy element / młodszy element (dzielenie całkowite)</i>
*	<i>starszy element * młodszy element</i>

%	<i>starszy element mod młodszy element (reszta z dzielenia)</i>
----------	---

Instrukcje warunkowe i pętle

Instrukcje warunkowe

ZSK23 udostępnia prostą metodę używania instrukcji warunkowych. Uogólniony zapis przedstawia się tak: *instrukcja ilość*. W zależności od instrukcji, jeżeli warunek zostanie spełniony to wykona się wskazana *ilość* linii. W przypadku niespełnienia warunku, interpreter pominię wskazaną *ilość* linii.

instrukcja	warunek
if=	<i>starszy element = młodszy element</i>
if>	<i>starszy element > młodszy element</i>
if<	<i>starszy element < młodszy element</i>
if>=	<i>starszy element >= młodszy element</i>
if<=	<i>starszy element <= młodszy element</i>
if!	<i>starszy element != młodszy element</i>

Poniższy kod sprawdza, czy masz rocznikowo skończone 18 lat.

```

1 string Podaj obecny rok >
2 print
3 getinteger
4 string Podaj rok urodzenia >
5 print
6 getinteger
7 calc -
8 push 18
9 if>= 2
10 string Masz conajmniej 18 lat!
11 print
```

W przypadku poprzedzenia instrukcji warunkowej tyldą, polecenie nie będzie zdejmować porównywanych liczb ze stosu.

```

1 string Podaj obecny rok >
2 print
3 getinteger
4 string Podaj rok urodzenia >
5 print
6 getinteger
7 calc -
8 push 18
9 ~if>= 2
10 string Masz conajmniej 18 lat!
11 print
12 ~if< 2
13 string Nie masz 18 lat...
14 print

```

Powyższy przykład nie mógłby zadziałać gdyby przed instrukcjami nie była postawiona tylda. Wynika to z faktu, że pierwsza instrukcja warunkowa zdjęłaby ze stosu dwie liczby, więc druga instrukcja warunkowa zastałaby pusty stos – nie byłoby czego porównywać.

Pętla warunkowa

Przykładem pętli warunkowej może być połączenie instrukcji warunkowych z poleceniem *jump*, czyli skokiem bezwarunkowym. Poniższy program pobiera dowolną liczbę, a następnie wyświetla wszystkie liczby naturalne od 0 do podanego argumentu.

```

1 string Podaj dowolna liczbe >
2 print
3 getinteger
4 push 0
5 ~if> 4
6 pop
7 ~calc 1 -
8 push 0
9 jump 5
10 pop
11 stack

```

Powyższy kod pobiera liczbę, następnie wrzuca 0 na stos, do którego będzie porównywany podany argument. Następnie pozbywa się zera, wykonuje obliczenia (dekrementuje podany argument), wrzuca zero z powrotem i przeskakuje bezwarunkowo w miejsce instrukcji warunkowej. Jeżeli podany argument będzie równy zero, pętla przestanie się wykonywać i program przeskoczy do linii 10.

Pętla stała

ZSK23 udostępnia kilka schematów pętli. Jednym z nich jest pętla stała, wykonywana poleceniem *loop ilość*. Wykonuje ona wskazaną *ilość* linii (mechanika podobna jak w przypadku instrukcji warunkowych), tyle razy, ile wskazuje na to najmłodsza liczba stosu. Ten schemat pętli nie może być zagnieżdżany w sobie samej. Ważnym faktem jest to, że polecenie *loop* zdejmuje liczbę ze stosu.

Poniższy kod wygeneruje wskazaną ilość liczb ciągu Fibonacciego.

```
1 push 1
2 push 1
3 string Podaj ilosc liczb ciagu Fibonacciego do wygenerowania (min. 2) >
4 print
5 getinteger
6 calc 2 -
7 loop 1
8 ~calc +
9 stack
```

Kod z poprzedniego podrozdziału można przepisać w taki sposób, aby użyć tylko i wyłącznie pętli stałej.

```
1 string Podaj dowolna liczbe >
2 print
3 getinteger
4 copy
5 loop 1
6 ~calc 1 -
7 stack
```

Uwaga, pętle stałe nie pozwalają na wyjście z nich za pomocą polecenia *jump*.

Pętla mieszana

Możemy połączyć pętle warunkowe i stałe. Poniższy program wyświetli 10 razy *Jest parzysta!*, jeżeli reszta z dzielenia liczby przez 2 będzie równa 0, i analogicznie wyświetli 5 razy *Jest nieparzysta...*, gdy liczba będzie różna od 0.

```
1 string Podaj dowolna liczbe >
2 print
3 getinteger
4 ~calc 2 %
5 push 0
6 ~if= 4
7 push 10
8 loop 2
9 string Jest parzysta!~
10 print
11 ~if! 4
12 push 5
13 loop 2
14 string Jest nieparzysta...~
15 print
```

Pozostałe polecenia

Polecenie *random* wygeneruje losową liczbę z zakresu $\langle 0, x \rangle$, gdzie x to najmłodsza liczba na stosie. Jeżeli po komendzie wskażemy jakąś liczbę, to *random* nie skorzysta z stosu, a z ww. liczby.

Polecenie *kill* kończy interpretację kodu. Jest to pozostałość z próby przedwczesnego zaimplementowania wielowątkowości, więc w tej chwili pełni funkcję zaledwie forsownego wyłączenia programu.

Przyszłościowe plany rozwoju projektu

Poniżej przedstawiam listę rzeczy które miały zostać wprowadzone do ZSK23, jednakże zabrakło na nie czasu.

1. Dyrektywy preprocesora – podstawowe polecenia *define* *ifdef* *ifndef* oraz *limit* ograniczający wielkość stosu
2. Wielowątkowość – możliwość rozbicia programu na kilka niezależnych wątków
3. Funkcje – możliwość definiowania własnych funkcji
4. Działanie na liczbach zmiennoprzecinkowych – dodanie obsługi liczb zmiennoprzecinkowych