NHF specifikáció

Uszkay Balázs

2025. április 22.

Stacc (és a számítástudomány két legnehezebb problémája) [1]

A *Stacc* egy kreatívan elnevezett, verem alapú konkatenatív, interpretált, általános célú programozási nyelv. Az inspirációt a Forth és a Factor nyelvek adták.

Nyelvi elemek

Egy *Stacc* program egymástól whitespace-szel elválasztott *szavakból* áll. Ezek jobbról balra kerülnek végrehajtásra. Szónak számít minden karaktercsoport, amelyek közt fehér tér van, kivétel a string konstans, amely `"`-től (`U+0022`) `"`-ig tart. Ezek a szavak a központi vermen operálnak. Kommentek `--` -től a sor végéig tartanak.

A veremre helyezéshez (`push`) nem tartozik külön szó, ez automatikusan megtörténik, amikor konstansba fut a program.

Adattípusok

A program futás közben az adatokat a veremben tárolja. Ezek az adatok az alábbi típusúak lehetnek:

- **integer:** 64-bites előjeles egész.
  Pl.: `-1 10 9223372036854775807`
- **float:** 64-bites lebegőpontos szám (double).
  Pl.: `27 -3.2 6.022e+23`
- **list:** Heterogén lista, más elemeket tartalmaz. `{` szótól (`U+007B`) `}` szóig (`U+007D`) tart.
  Pl.: `{ 1 2 3 4 } { { "alma" 213.3 } { } }`
- **string:** Karakterek listája, `"`-től (`U+0022`) `"`-ig tart. A listák műveletei ugyanúgy érvényesek.
  Pl.: `"Tudja, hol szeret a cápa" "kecske" ""`
- **block:** Futtatható blokkok, egyfajta anonim függvények. Ezek segítségével lehet például if-else, ciklus stb. control flow-t megvalósítani. `‘[‘` szótól (`U+005B`) `‘]‘` szóig (`U+005D`) tart.
  Pl.: `0 < [ sqrt 2 * ] [ dup * ] if --` ez a program negatív számokon máshogy operál, mint pozitívakon
- **ident:** Olyan szó, amely valamilyen függvényre hivatkozik. Létrehozni a `’` (`U+0027`) karakterrel lehet, azaz a szó első karaktere `’` kell, hogy legyen.
  Pl.: `[ 2 < [ drop 1 ] [ dup -1 + ’fac call * ] if ] ’fac :`
  Megjegyzés. A fenti kódrészletben szereplő `:` szó névhez (**’ident**) köt egy blokkot (**{block}**), azaz függvényt definiál.

[1] Phil Karlton szerint két nehéz dolog van a számítástudományban: a gyorsítótár-érvénytelenítés és dolgok elnevezése.

Beépített primitívek

Ez a szakasz nagyon megváltozhat a végleges beadásig!

Az itt felsorolt primitív szavak egyből használhatók, ezeket definiálni nem kell. A `( -- )` jelölés a Forth és a Factor által is használt (az előbbi által csak konvencióként) *veremjelölés* (stack notation), ahol a `--` bal és jobb oldala sorrendben a szó végrehajtása előtti és utáni verem állapotot jellemzi. Ilyenkor csak a legfelső n elemet mutatjuk (hiszen csak ezek számítanak), a fent említett nyelvekhez képest annyi kiegészítéssel, hogy én a blokkokat és a neveket kiemelten jelölöm. Ezért például a `:` szó veremjelölése:

`({block} 'ident -- )`,

hiszen ez a szó elnyeli a verem tetején lévő blokkot és nevet. A verem teteje jobbra van, így a `rot` szó jelölése,

`(a b c -- b c a)`,

azt mutatja, hogy felülről harmadik elem kerül a stack tetejére.

I/O:

- `. (a -- )`

  Kiírja a stack legfelső elemét és ejti azt.

- `S. (... -- ...)`

  (Debug jellegű kiírás) kiírja a stack nagyságát és összes elemét.
  Példa:

```
1 2 'nev [ dup * 2 + ] "árvíztűrő tükörfúrógép" { 1 2 "alma" 4 } S.
<5>
1
2
'nev
[<block>]
"árvíztűrő tükörfúrógép"
{<4-list>}
```

Aritmetika, műveletek:

- `+ (n1 n2 -- sum)`
  `- (n1 n2 -- n1-n2)`
  `* (n1 n2 -- prod)`
  `/ (n1 n2 -- n1/n2)`
  `% (n1 n2 -- n1%n2)`
  `divmod (n1 n2 -- n1/n2 n1%n2)`
  `pow (n1 n2 -- n1^n2)`

  Bináris aritmetikai műveletek. `%` a modulus operátor. Stringeken a `+` konkatenálásként működik.

- `sqrt (n -- r)`
  `sin (n -- r)`
  `cos (n -- r)`
  `tan (n -- r)`
  `arcsin (n -- r)`
  `arccos (n -- r)`
  `arctan (n -- r)`

  Unáris műveletek.

- `inc (a -- a')`
  `dec (a -- a')`

  Eggyel növeli, ill. csökkenti a verem legfelső elemét.

- `and (bool1 bool2 -- bool)`
  `or (bool1 bool2 -- bool)`
  `xor (bool1 bool2 -- bool)`
  `not (bool -- bool)`

  Boole-algebrai műveletek. A `false`-t a `0` egész, a `true`-t pedig a `-1` (avagy nem `0`) jelöli (emiatt jelenleg a `not` egészekre a negálás).

Listaműveletek

- `len ({list} -- n)`
  Visszaadja egy lista hosszát.

- `append ({list} item -- {list'})`
  Végére illesztés

- `prepend ({list} item -- {list'})`
  Elejére illesztés

- `insert ({list} index item -- {list'})`
  Pozícióba illesztés.

- `each ({list} [block] -- ...)`
  Művelet elemenként.

- `map ({list} [block] -- {list'})`
  Művelet végrehajtása minden elemen.

- `filter ({list} [block] -- {list'})`
  Elemek szűrése feltétel alapján.

- `reduce ({list} init [block] -- result)`
  Redukció (fold).

- `reduce1 ({list} [block] -- result)`
  Kezdőérték nélküli redukció.

- `scan ({list} init [block] -- {result})`
  Folytonos redukció (scan).

- `scan1 ({list} [block] -- {result})`
  Kezdőérték nélküli `scan`.

- `first ({list} -- item)`
  Első elem.

- `last ({list} -- item)`
  Utolsó elem.

- `take ({list} n -- {list'})`
  Első `n` elem.

- `drop ({list} n -- {list'})`
  Lista az első `n` elem nélkül.

- `iota (n -- {list})`
  Számok listája 1-től `n`-ig.

Stringműveletek

- `upper ("str" -- "STR")`
  Nagybetűssé tétel.
- `lower ("STR" -- "str")`
  Kisbetűssé tétel.
- `find ("str" "sub" -- pos)`
  Megkeresi `"sub"` első pozícióját `"str"`-ben.
- `count ("str" "sub" -- n)`
  Visszaadja `"sub"` darabszámát `"str"`-ben.

Összehasonlítás:

- `= (a b -- bool)`
  `< (a b -- bool)`
  `<= (a b -- bool)`
  `> (a b -- bool)`
  `>= (a b -- bool)`
  `!= (a b -- bool)`

  Összehasonlítások. Irreflexív összehasonlításoknál a verem tetején levő kerül „jobbra”. Példa:

  ```
  2 3 < .
  -1 -- 2 < 3, igaz
  4 10 >= .
  0 -- 4 >= 10, hamis
  ```

Verem kezelés:

- *<konstans>* `( -- a)`
  Konstans felrakása a veremre.
- `dup (a -- a a)`

  Duplicate, a legfelső érték megduplázása.
- `2dup (a b -- a b a b)`

  2-Duplicate, a felső elempár duplázása.
- `swap (a b -- b a)`

  A felső két érték cseréje.
- `2swap (a b c d -- c d a b)`

  A felső két elempár cseréje.
- `over (a b -- a b a)`

  A második érték megduplázása a verem tetejére.
- `drop (a -- )`

  A legfelső érték eldobása.
- `rot (a b c -- b c a)`

  Rotate, a harmadik érték felhozása a verem tetejére.

Control flow

- `[` és `] ( -- [blokk] )`
  Új blokk felrakása a veremre.
- `call ('fv -- )`
  Függvény hívása.
- `: ([blokk] 'nev -- )`
  Névhez rendelés.
- `if (bool [then] [else] -- )`
  Feltételes futás.
- `curry (item [block] -- [item block])`
  Currying.

A feladat

Írjon interpretert a fent leírt *Stacc* nyelvhez! A program legyen képes fájlokat bemenetként venni, illetve fájlmegadás hiányakor viselkedjen REPL-ként

Konzultálandók

Kérdéses, hogy legyen-e modulrendszer, és hogy milyen szavakkal érdemes bővíteni a beépítettek listáját. Eddig a nyelv nem tud bemenetet venni a standard bemenetről (`stdin`), vagy fájlt megnyitni stb.

Az is kérdéses, hogy az .exe mit tudjon (ez a másfél mondatos feladatkitűzés erősen bővíthető).

Terv

1. ábra. Tervezett osztálydiagram. Ezeken az osztályokon függvények operálnak.

2. ábra. Egy program lefuttatásának pszeudokódja.

```
def Futtatas(szoveg: string) -> bool:
    tokenek: Maybe<vector<Token*>> = Tokenize(szoveg).
    if (!tokenek.has_value())
        return true.
    # megjegyzés. ezt a részt vsz. closure-rel oldanám meg.
    objektumok: Maybe<vector<Object>> = Parse(*tokenek).
    if (!tokenek.has_value())
        return true.
    success: bool = Interpret(*objektumok).
    return success.

def Tokenize(szoveg: string) -> Maybe<vector<Token*>>:
    tokenek = empty vector.
    while (szoveg.not_empty()):
        szoveg.skip_ws()
        if (szoveg.next() == '"'):
            olvasd sztringkent, amig ujra nem '"'.
            tokenek.append(uj String(...)).
        olvass be egy szót -> word: string
        ha csak numerikus karakter:
            tokenek.append(uj Int(word))
        ha lehet valos:
            tokenek.append(uj Float(word))

def Parse(tokenek: vector<Token*>, block: bool=false, list: bool=false) -> Maybe<vector<Object*>>:
    objektumok = empty vector.
    for token of tokenek:
        if block és token == Word(']'):
            return Maybe(objektumok).
        if list és token == Word('}'):
            return Maybe(objektumok).
        if token == Word('['):
            objektumok.append(Parse(tokenek, true, false)).
        if token == Word('{'):
            objektumok.append(Parse(tokenek, false, true))
        objektumok.append(Token típusának megfelelő Object).
    return objektumok.

def Interpret(objektumok: vector<Object*>) -> bool:
    for objektum of objektumok:
        switch objektum:
        | Word('call') > ...
        | Int > ...
        ...
    Ha bárhol hiba van, jelezzük a visszatérési értékkel.
```