TARTALOMJEGYZÉK

1. BEVEZETÉS

A szakdolgozat célja kettős: a feladat első része egy programozási nyelv, második fele a hozzá tartozó értelmezőprogram leírásáról illetve tervezéséről szól.

Az értelmezett nyelvhez – melyet Webtongue v1 névre kereszteltem – kliens oldalon működő, internet-böngésző programokba ágyazható interpretert készítettem. Azért született ez a döntés, mert a nyelv alább vázolt profilja elsősorban minél szélesebb körű elérhetőséget követel meg, háttérbe szorítva a sebesség illetve felhasználói alkalmazásba ágyazhatóság egyébként lényeges követelményét.

A nyelv jelenlegi állapotában kísérleti és oktatási célokat szolgál. Tervezésekor nem a bizonyítottan legproduktívabb, hanem régóta meglévő de indokolatlanul perifériára szorult, illetve újszerű, de még meg nem szilárdult módszereket kombináltam a lehető legcélravezetőbb módon. Ezek az eljárások önmagukban is igen hatékonyak, de megfelelően együttműködve még egyszerűbb és kényelmesebb funkcionalitást eredményeznek. Mindezeket egyéni ötletekkel is kiegészítettem, így valóban sajátos és újszerű keverékét kaptuk a különböző módszereknek.

Sokrétű természete és generikus felépítése miatt a Webtongue közel bármely programozási nyelvnek megfelelő működésre képes, legyen az funkcionális elvű vagy egyszerű verem-automata. Alprogramokkal könnyen bővíthető szerkezete lehetőséget biztosít arra, hogy nyelvi elemeit kombinálva újakat hozzunk létre, illetve megváltoztassuk, befolyásoljuk a nyelv bizonyos alapvető mechanizmusainak működését. Mindezen tulajdonságok miatt hasznos segítséggé válhat a programozási nyelvek elméletében felmerülő számos probléma és lehetőség szemléltetésében akár az oktatás területén, akár új eljárások, alapelvek kidolgozásakor.

(Megjegyzés: A dolgozat egyes leíró részeiben többes szám első személyben fogalmaztam, ennek oka kizárólag stilisztikai, a munkát teljesen egyedül végeztem.)

2. A WEBTONGUE V1 NYELV

2.1. Rövid áttekintés

A Webtongue alapötletének megszületésekor a fő inspirálók azok a létező programozási nyelvek voltak, amik egyéni szerkezeteikkel és új – vagy épp régi de hatékony – gondolataikkal valamilyen módon kitűntek az értelmezett vagy lefordított nyelvek sokaságából. A Forth (2.) nyelv verem központú működése, a Lisp (9.) funkcionális felépítése illetve a REBOL (3.) univerzális blokk szerkezetei voltak azok az alapkövek, amikre a Webtongue koncepcióját építettem, a továbbiak már csak azon múltak, hogyan illesztem össze ezeket az elemeket. A tervezés során a legjelentősebb korlátozó tényező a feladatra jutó idő volt, mindössze fél év állt rendelkezésre ahhoz, hogy az alapötlettől a jól dokumentált és implementált értelmezőig eljussak.

A nyelv specifikációja sokat változott az első néhány hónapban, eleinte a többszöri újragondolások, később az implementáció során felmerülő friss elképzelések miatt, amik az elméleti tervezés időszakában elkerülték a figyelmemet. Az idő rövidsége olyan megoldásra sarkallt, amely a legegyszerűbb szerkezettel biztosítja az alapötletnek megfelelő nyelvi dinamizmust. Ebben a fázisban az assembly nyelv egyszerűsége adta a mércét, olyan scriptelőt szerettem volna készíteni, ami hasonlóan minimális felépítéssel biztosít az emberi gondolkodás számára kényelmes működést.

2.2. Alapelvek

A Webtongue tervezése során négy alapvető elvhez tartottam magam. Hogy miért épp e négy gondolat adta a tervezés alapját, az részletesebb magyarázatuk után olvasható:

a.) a nyelv szerkezete maradjon egyszerű és uniformizált

b.) felépítése tükrözze a metacirkuláris nyelvek sajátosságait (1.) (3.)

c.) az elkészült nyelv ne legyen szigorúan típusos

d.) a szintaxis ne legyen kötött

Az alábbiakban a fenti alapgondolatok kifejtése, illetve azok Webtongue v1-beli gyakorlati megvalósítása következik. Látni fogjuk, hogy sikerült tartani magunkat az eredeti elképzelésekhez:

a.) Ideális esetben az elkészült nyelv valamennyi nyelvi eleme, nyelvtani szerkezete a programozó szemszögéből ugyanúgy működjön. Ne legyen szintaktikai eltérés a beépített operátorok, alprogramok, vezérlési szerkezetek használati módja között, azok tetszőleges kombinációban egymásba ágyazhatóak legyenek. Az uniformizmus terjedjen ki a Webtongue szemantikájára is, azonos logikával lehessen valamennyi rendelkezésre álló eszközt felhasználni, ne csak a felírás módja egyezzen meg mindegyik esetben. Újabb követelmény, hogy a programozó minél kevesebb adattípussal dolgozzon (ennek részletesebb kifejtése a c.) pontban olvasható). Szintén fontos elvárás az egyes elemek minél sokrétűbb felhasználhatósága.

   A megoldás:

   - Az azonos logikával illetve szintaktikával működő elemek problémájának megoldását a kizárólag szubrutinhívások egymásutánjából álló Webtongue programszerkezet adja. Minden egyes nyelvi konstrukció egy-egy szubrutinnak felel meg. A beépített illetve felhasználói alprogramok indítása közt pedig az az egyedüli különbség, hogy utóbbiaknak a speciális `run` függvény paramétereként kell szerepelniük.

- Az egyes elemek sokrétű felhasználására irányuló törekvésünk legszembetűnőbb megvalósulása a speciális blokk szerkezet (3.). Webtongue-ban minden egyes programblokk egyúttal karakterláncként illetve tömb adatszerkezetként is képes funkcionálni. Természetesen mindez vice versa működik, bármelyik konstrukció alkalmas a többi szerepének ellátására.

b.) Esetünkben a metacirkuláris felépítés azt jelenti, hogy a programozási nyelv képes önmaga értelmezésére. Természetesen nem a nyelv öntudatra ébredéséről beszélünk, hanem arról, lehetséges-e kizárólag Webtongue-ban írt programmal Webtongue forráskódot helyesen értelmezni és végrehajtani, illetve van-e arra mód, hogy a nyelv szerkezeti elemeit kizárólag Webtongue-ban írt alprogramokkal kibővítsük. Jelen fogalomértelmezés az első felében teljesen megegyezik a REBOL fejlesztői által elfogadottal (3.), második része viszont újabb fontos egyéni tulajdonsággal bővíti elvárásaink sorát.

A megoldás:

- A beépített `eval` függvény segítségével tetszőleges, paraméterben megadott karakterlánc Webtongue programblokként értelmezhető. Ez a szubrutin teszi lehetővé a Webtongue programok egyszerű weboldalba ágyazását is.
- A `var` függvény funkciója a paraméterben megadott nevű Webtongue változó értékének visszaadása (ha a változó már létezik, vagyis van ilyen nevű beépített vagy felhasználói definíció).
  (Megjegyzés: Webtongue-ban nem lehetséges elődeklarációt végezni, az új változót azonnal, kezdőértéket megadva definiálni is kell.)

- A Webtongue programblokkok hármas természete miatt tetszőleges karakterlánc futtatható Webtongue kódként a `run` beépített funkció segítségével.

c.) Véleményem szerint egy értelmezett nyelv vagy scriptnyelv esetében fontosabb az egyszerű, átlátható, könnyen és gyorsan elkészíthető, módosítható programszerkezet kialakítása, mint a programozói hibák minimalizálást és a teljesítmény javítását célzó, de adott esetben bonyolultabb konstrukció elkészítése. Természetesen az általános produktivitást növelő megoldásoknak is megvan a helye, de az semmi esetre sem egy scriptnyelv.

A megoldás:

- Ennek szellemében a Webtongue valamennyi változója típus nélküli lett.
- Az egyes változótípusok közti konverzió automatizált.

d.) A c.) pontban vázolt érveléshez hasonló okokból a Webtongue szintaxisa kötetlen maradt. Olyannyira igaz ez, hogy még a nyelv karakterlánc változói sem merev szintaxisúak, a HTML leírónyelv szövegrészleteivel analóg szerkezetűek. Többek közt ez tette lehetővé a programblokk, tömb és karakterlánc konstrukciók összevonását is. (A nyelv szintaxisáról bővebben a 2.3. fejezetben olvashatunk.)

Az első lépés a fenti elvek kialakításakor a cél pontos kitűzése és megértése volt. Mivel nem általános, összetett programcsomagok fejlesztésére használható programozási nyelvet akartam készíteni, hanem kísérleti célú, kisebb volumenű problémák megoldására alkalmas scriptelőt, ennek megfelelően kellett kiválasztani a nyelv leendő meghatározó tulajdonságait is. Az utóbbi években figyelemmel kísértem néhány számítógépes nyelvészettel foglalkozó kutatást – amilyen például az Aardappel (7.) vagy az ANTLR (8.) – valamint megismerkedtem olyan régi és új fejlesztésekkel,

mint a Forth (2.), Lua (4.), Python, vagy akár False (7.). A szakirodalom és a programozói társadalom ajánlásain és tapasztalatain túlmenően ezek az ismeretek adták a Webtongue alapjait. Megpróbáltam kiválogatni és ötvözni a legkedvezőbb tulajdonságaikat, elvetve az egyszerűség és generikus felépítés útjában álló jellemzőket.

2.3. A Webtongue programok szerkezete

2.3.1. Egy programblokk felépítése

(Ahogyan eddig is tettük, az alprogram, szubrutin illetve függvény szavakat szinonimaként fogjuk használni. Bármelyik szerepeljen is a szövegben, a Webtongue valamely beépített vagy felhasználói alprogramját értjük alatta.)

Tetszőleges Webtongue programblokk a következő modell szerint épül fel. A programblokkok tetszőleges mélységben egymásba ágyazhatóak, ahogy a függvényhívások is egymás paramétereként:

```
szubrutinhívás szubrutinhívás szubrutinhívás...
szubrutinhívás szubrutinhívás...
...
```

A programblokk utasításai sorrendben, balról jobbra kerülnek kiértékelésre. A szubrutinhívások pontos működését a 2.3.4. részben taglaljuk részletesen. Egy szubrutinhívás vázlatos szerkezete a következő:

```
szubrutin_változó_neve paraméter1 paraméter2...
```

A rekurzív függvényhívás teljes mértékben támogatott. Vigyázzunk azonban rekurzió alkalmazásakor: a változók globálisan tárolódnak! Amikor csak tehetjük,

használjunk vermet a helyi változók alkalmazása helyett. (A változók láthatósági szabályairól illetve élettartamáról a 2.3.3. részben olvashatunk bővebben.)

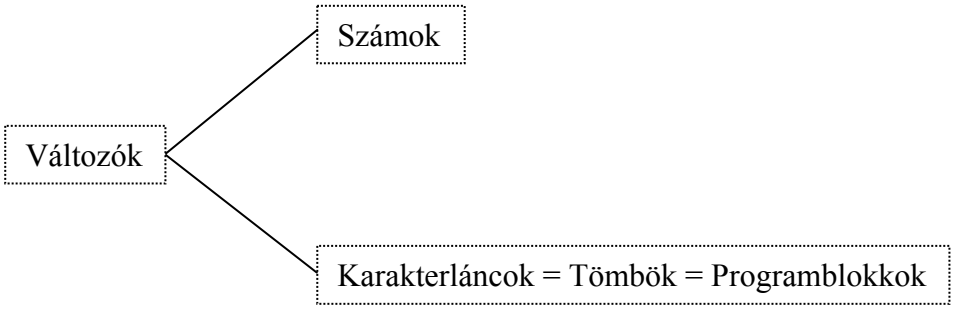
Szintaxis:

- A legfontosabb szintaktikai megkötés az, hogy az egymást követő nyelvi elemek közt minden esetben legalább egy üres helyet kell hagyni (szóköz, tabulátor vagy újsor karaktert).
- A főprogramtól különböző blokkokat a `{ }` nyitó illetve záró karakterek között adhatjuk meg. (Figyeljünk az üres helyek kihagyására!)
- A programblokkokban szerepelhetnek megjegyzések, ezeket `** **` párok közé kell írni. (Pl.: `** ez egy megjegyzés **`)

2.3.2. A Webtongue változói

Komplex adatstruktúrái miatt a Webtongue valamennyi nyelvi eleme egyenrangú változóként tárolódik. Nincs szükség az adatváltozók illetve szubrutinok önálló nyilvántartásra – egészen pontosan a nyelv szerkezetéből adóan erre mód sem kínálkozik. A beépített függvények kivételével ezért valamennyi alprogram hozzáférhető tömbként vagy karakterláncként is. (Az egyszerű `print dup` programot lefuttatva például kilistázhatjuk a `dup` Mezzanine függvény programkódját. A Mezzanine szubrutinokról bővebben a 2.5. fejezetben olvashatunk.)

Az egyes felhasználói változótípusok a következőképpen csoportosíthatók:



Láthatjuk, hogy a differenciálódás minimális, csak annyi adattípust különböztetünk meg az értelmező szempontjából, amennyit okvetlenül szükséges. Megtehettük volna, hogy a szám típust is összeolvasztjuk a komplex blokk struktúrával, és mint speciális egyetlen szóból álló karakterláncot, vagy egy elemű tömböt kezeljük. Ez a megoldás azonban feleslegesen bonyolította volna az automatikus konvertálás funkcióját az olyan beépített szubrutinok esetén, amelyek különböző módon kezelik a szám illetve karakterlánc argumentumokat. Arról nem is szólva, hogy így jóval egyszerűbb szintaxissal adhatunk meg közvetlen számértékeket a programunkban, nincs szükség a blokkot jelölő karakterekre.

Szintaxis:

- A változók nevében tetszőleges karakter szerepelhet, de kerüljük a kizárólag számokból álló vagy számjegyekkel kezdődő azonosítók alkalmazását. Hasznos lehetőség az alapvető függvények elnevezésére a különböző matematikai műveleti jelek illetve írásjelek használata. (Pl.: + - ’ ")
- A változónevek nem érzékenyek a kis- illetve nagybetűkre.

2.3.3. A változók élettartama és láthatósági szabályai

Minden egyes változó a definiálásának pillanatától kezdve a program futásának befejezéséig él. A program végrehajtása közben tetszőlegesen sokszor változhat az értéke (felüldefiniálással), de semmilyen módon nem szabadulhat fel, nem kerülhet ki a változók listájából. (Erre egy értelmezett nyelvben nincs is szükség, hiszen az adott nyelven elkészített programok valószínűleg sosem nőnek hatalmassá.)

Minden változó a definiálásának pillanatától kezdve a program egészében láthatóvá válik. Bárhová kerüljön is a vezérlés a forráskód kiértékelése során, a változó biztosan elérhető onnan. Részletesebben a 3. részben térünk ki ennek gyakorlati megvalósítására.

Ez talán a változók tárgyalása során felmerülő leglényegesebb kérdés, így mindenképp mélyebb indoklást igényel. A Webtongue fejlődése során több

láthatósággal kapcsolatos koncepció is gyökeret vert a specifikációban, volt amelyik hosszabb, volt amelyik rövidebb ideig tartotta magát. Az eredeti elképzelés szerint a C nyelv ilyen irányú szabályozásának megfelelő rendszer érvényesült volna a Webtongue-ban is, de az adattípusok számának fokozatos csökkenésével, illetve a megmaradó adatstruktúrák összetettségének növekedésével a helyzet fokozatosan változott. Ahogyan a nyelv szerkezete egyre egyszerűbbé, a megmaradt alkotóelemek egyre alapvetőbbé és sokrétűbbé váltak, a láthatóság koncepciója is a lehető legegyszerűbb modellt kezdte követni. A 2.2. részben vázolt első alapgondolathoz igazodva végül a globális láthatóság szabálya maradt érvényben.

Van ezen kívül még egy igen lényeges indok, ami a globális megoldás mellett szól. Számos összetett algoritmus esetében (pl. rekurzív szubrutinok) az utóbbi évek (kis híján évtizedek) általános programozói rutinja a lokális változók használatát helyezi előtérbe. Ez a gyakorlat természetesen helyes, mégis sok esetben üdvös lehet, ha a kissé erősebb odafigyelést igénylő, de jóval hatékonyabb és szebb veremtáras megoldással oldjuk meg a problémát. Mindez különösen kisebb alkalmazások esetében igaz, amelyek készítésére a Webtongue-ot is szántam. A hibalehetőségek száma kissé megnő ugyan, de megfelelő gyakorlatot szerezve számos helyzetben igen hasznos eszközzé válhat a kezünkben ez az eljárás. Példaként tekintsük a következő, Webtongue-ban írt algoritmust. A programban egyetlen segédváltozót sem használunk:

```
let fac
{
   if not eq peek top 1
      { push dec peek top run fac push mul pop pop }
      nop
}

push 6
run fac
print pop   ** 6! = 720 **
```

A Tisztelt Olvasó mostanára biztosan azt gondolja, egyszerűen elsiklunk a puszta tény felett, mely szerint napjainkban a globális adattárolás módszere elavuló

félben van és többféle veszélyt is rejt magában. Természetesen indokoltak ezek az aggályok, viszont kísérleti illetve oktatási célú nyelvként ez a hiányosság még a Webtongue előnyére is válhat. Léteznek áthidaló megoldások a probléma orvoslására, a különböző lehetőségek megvalósítása közben pedig még jobban elmélyülnek a felhasznált módszerekre vonatkozó ismereteink. Igen hasznos oktatási példa lehet a Webtongue ilyen vagy olyan hiányzó tulajdonsággal történő felruházása pusztán a meglévő eszközök segítségével. Érdekes és megvalósítható feladat például olyan szubrutin készítése, ami a paraméterben megadott műveleteket nem prefix, hanem infix módon értelmezi.

Egyszerű példa a láthatósági szabályok működésére:

```
let newSub
{
   let a 1
   let b 2
}

print add a b    ** hibát okoz, mert a és b
                    még nem definiált **

run newSub       ** mostantól kezdve a és b
                    bárhol elérhető a programban **

print add a b    ** a kimenet: 3 **
```

2.3.4. Adattípusok

a.) A 2.3.3. részben láttuk, hogy a Webtongue változói két nagy csoportra oszthatók. Az első ilyen csoportot a számokat reprezentáló változók alkotják:

A Webtongue számai tízes számrendszerben értelmezett lebegőpontos értékek. A legtöbb JavaScript implementációban 32 bites IEEE-754 lebegőpontos számként tárolódnak, ezért értékük viszonylag pontatlan. (Érdekes feladat lenne Webtongue-ban tetszőleges pontosságú lebegőpontos aritmetikai műveleteket megvalósító függvényeket írni.)

Érvényes közvetlen megadás pl.: `123 12.3 12e-3 -123`

b.) A másik, jóval szerteágazóbb osztály az univerzális blokk struktúráké:

Ahogy arról már szóltunk, a Webtongue blokkjai hármas természetűek: egyaránt képesek karakterláncként, tömbként illetve programblokként működni. Most részletesen megvizsgáljuk mindhárom esetet:

I.) A blokk struktúra, mint karakterlánc:

A Webtongue értelmező minden egyes karakterláncot HTML stringként kezel. A szavak közt lévő üres hely mérete nem számít, nem is kerül tárolásra. A szövegben használhatunk HTML tageket és szabványos escape szekvenciákat. Ha üzenetünket weboldalon jelenítjük meg, éppúgy működnek mintha csak közvetlenül írtuk volna be őket a weblap kódjába. (Az online JavaScript értelmező csupán egy szövegdobozban jeleníti meg a kimenetet, ott a felhasznált HTML kódok úgy jelennek meg, ahogyan beírtuk őket.)

Példa:

```
print { Hello   World! }   ** a kimenet: Hello World! **
```

II.) A blokk struktúra, mint tömb:

Ha a Webtongue értelmezőprogramja tömbargumentumot talál, annak tartalmát az a.) pontban leírt módon értelmezi. A tömbelemek a karakterlánc egyes szavai lesznek, nullától a szavak száma mínusz egyig indexelve. Az egyes elemek értékének típusa automatikusan konvertálódik. Az alábbi példa jól illusztrálja a konstrukció működését:

```
let s run string 0 10
   ** megadhatnánk let a { 0 0 0 0 0 0 0 0 0 0 }
      alakban is – létrehozzuk a tömböt **

let i 10             ** iterátor a tömb bejárásához **

let j { sub 10 i }   ** fordított iterátor – 10-től
                        számlál lefelé **

** az alábbi programrészlet feltölti a tömböt nullától
   kilencig egyesével növekvő elemekkel **

loop i
{
   let s run write s run j run j
   let i dec i
}

print s   ** a kimenet: 0 1 2 3 4 5 6 7 8 9 **
```

Látható, miként inicializálhatunk egy tömböt a `string` Mezzanine szubrutin segítségével, ahogyan az is, hogyan adhatjuk meg a tömb kezdőértékeit közvetlen definícióval. A fent létrehozott `s` tömb természetesen egyúttal karakterláncként is használható, tartalma bármikor kiírható a `print` utasítás segítségével, vagy végezhetünk vele tetszőleges egyéb karakterlánc-műveletet. A legtöbb esetben a blokk adattípus sokrétű tulajdonságait együttesen alkalmazva érhetjük el a legegyszerűbb eredményt, hol tömbként, hol karakterláncként használva azt.

A tömbmanipuláló beépített és Mezzanine szubrutinok részletes leírását a 2.5. részben olvashatjuk.

III.) A blokk struktúra, mint alprogram:

Az alábbi egyszerű példa végigvezet a Webtongue-beli függvényhívás folyamatán:

```
let newSub
{
   get p1
   get p2
   get p3

   run { add p1 add p2 p3 }

}

print run newSub 1 2 3   ** a kimenet: 6 **
```

Elsőként `newSub`-ot mint általános blokkot definiáljuk. Az interpreter ekkor még nem tudja, miként értelmezze az univerzális blokkunkat, csak akkor dől el, milyen felhasználásra szántuk, mikor a `run` speciális beépített függvény paramétereként meghívásra kerül. Mielőtt a függvényhívás mechanizmusának taglalásába kezdenénk, tudnunk kell, hogy a Webtongue az alprogramok paraméterit egy speciális, csak erre az egy műveletre használható sor struktúrában adja át a szubrutinnak. A `get` függvény feladata ezen paraméterek maradéktalan kiürítése a sorból, elemeket a sorba helyező szubrutin pedig egyáltalán nem is létezik – ez a művelet az értelmezőre hárul. A `get` függvény a `let` hívással egyetemben speciálisnak minősül, amennyiben ezek az egyedüliek a beépített szubrutinok sorában, amik új változó létrehozására alkalmasak. Amikor `get`-et használva argumentumokat emelünk ki a hívási sorból, nem kell előzőleg változókat definiálnunk számukra, a `get` elvégzi ezt nekünk. Fontos még tudnunk, hogy `get` szubrutinhivatkozás kizárólag programblokk elején szerepelhet, de ott tetszőlegesen sok.

Visszatérve a hívás menetéhez: mikor a vezérlés egy `run` szubrutinhoz ér, a beépített függvény beolvassa a végrehajtandó blokk paramétert (legyen az változóban tárolt vagy közvetlen megadású), majd megszámlálja, hány darab `get` utasítás szerepel a blokk elején. Ezután annyi paramétert olvas még be a forráskódból és helyez el a paramétersorban, ahány hivatkozást talált, majd átadja a vezérlést az első argumentumként kapott utasításblokknak. (A beépített szubrutinok rögzített számú paramétert olvasnak be a programkódból, az erre vonatkozó információk a 2.4. alfejezetben találhatóak.) Az argumentumok beolvasása során természetesen az azokban

elhelyezett függvényhívások végrehajtódnak, és a függvények visszatérési értéke lesz a tulajdonképpeni argumentum:

```
let a { get p1 get p2 add p1 p2 }
print run a add 1 2 add 3 4   ** a kimenet: 10 **
```

A függvényblokkok visszatérési értéke a legutoljára végrehajtott utasítás kimenete. A fenti példában az `a` blokk visszatérési értéke az utolsó `add p1 p2` összeadás eredménye.

A legtöbb Webtongue függvény rendelkezik visszatérési értékkel, hiszen ennek a számnak/blokknak a meghatározása automatizált. Amennyiben egy programblokk nem tartalmaz egyetlen eredményt visszaadó hívást sem, a visszatérési érték a legutóbbi ilyen végeredmény lesz. A beépített Webtongue hívások úgy kerültek kialakításra, hogy egymásba ágyazásuk ne jelentsen gondot. Ezzel a szerkezettel lehetséges közel értelmes, beszélt nyelvű mondatra emlékeztető programkódot készíteni. Általában egyikük sem képes a paraméterként átadott argumentum értékének megváltoztatására, a legtöbb esetben a függvény kimenete az, ami a végrehajtott változást tükrözi:

```
let a 1 inc a print a          ** a kimenet: 1 **

let a 1 print inc a            ** a kimenet: 2 **
print a                        ** a kimenet: 1 **

let a 1 let a inc a print a    ** a kimenet: 2 **
```

Azok a beépített hívások, amelyeknek nincs visszatérési értéke, a „N0pe.” karakterlánccal térnek vissza. Ugyanez a karaktersorozat szolgál az esetleges hibás végeredmény jelzésére is.

Tetszőleges blokk futtatható Webtongue kódként a `run` beépített függvény segítségével:

```
run { print { Hello World! } }

run { get a get b add a b } 3 5
```

A szubrutin ilyen jellegű felhasználása analóg a λ függvény működésével (1.):

„… Then our `pi-sum` *procedure can be expressed without defining any auxiliary procedures as*

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b)) …"
```

Mindez Webtongue kódban:

```
let pi-sum
{
      get a get b

      add run { get p1 div 1 mul p1 add p1 2 } a
          run { get p2 add p2 4 } b
}
```

2.3.5. A Webtongue nyelvtana

Az alábbiakban a Webtongue nyelvet generáló nyelvtani szabályok következnek. A szabályrendszer megadásakor igazodunk a Lua teljes szintaxisát definiáló leíráshoz (4.). A definícióban nem térünk ki a szavak és számok felépítésére, valamint az egyes nyelvi elemek közti szóközök elhelyezésére – azokat értelemszerűnek illetve automatikusnak tekintjük.

```
// Start
(1) S ::= BPS | üres
// Beépített függvények
(2) B ::= run P | let | var | if | eq | gt | gteq | and |
      or | not | loop | exit | error | debug | ws | tab |
      inc | dec | add | sub | mul | div | mod | pow | sin |
      cos | tan | asin | acos | atan | ln | e | pi | abs |
      ceil | floor | round | rand | push | pop | lift |
      drop | peek | poke | top | dump | tostack | tostring
      | melt | merge | get | date | print | println | input
// Paraméterek
(3) P ::= BP | PP | üres | Q
(4) Q ::= QQ | szó | szám | '{' Q '}' | '{' S '}'
```

Példaként tekintsük az előző oldali `pi-sum` függvény programblokkját:

```
get a get b
add run { get p1 div 1 mul p1 add p1 2 } a
    run { get p2 add p2 4 } b
```

Ez a blokk generálható a fenti szabályok alkalmazásával, a következő sorrendben:

(1) (1) (1) (1) => BPBPBP
(2) (2) (2) (3) (3) (3) => get Q get Q add PP
(4) (4) (3) (3) (2) (2) => get a get b add run PP run PP
(3) (3) (3) (3) (4) (4) (4) (4) => get a get b add run { S } a run { S } b

Hasonló módszerrel a maradék S nemterminálisokat is feloldjuk és megkapjuk a kívánt eredményt.

2.4. Beépített szubrutinok

A Webtongue v1 szubrutinjai (a Mezzanine függvények kivételével, melyeket a következő alfejezetben tárgyalunk) négy nagy csoportra oszthatók. Ezek a nyelv szerkezetét adó alapfüggvények, a matematikai alprogramok, a verem és sor adatstruktúrákat kezelő függvények illetve az I/O-ért és egyéb funkciókért felelős szubrutinok. Csoportonként külön fogjuk tárgyalni őket, de ne felejtsük, hogy szerves egészet alkotnak, a különböző lehetőségek kombinációja adja a nyelv igazi erejét. Mivel a beépített alprogramok száma viszonylag kevés, majd mindegyikük funkcionalitása igen jelentős. A Webtongue számos sajátossága csak beépített szubrutinjainak alapos tanulmányozásával ismerhető meg, néhányuk csak itt kap említést!

A beépített függvények nevei, a blokkot nyitó-záró karakterek, valamint a megjegyzések jelölésére szolgáló dupla csillag lefoglalt szavaknak számítanak Webtongue-ban. Az értelmezőprogram ugyan nem okvetlenül ellenőrzi a felüldefiniálásukat, de ezen elnevezések használata erősen ellenjavallt programjainkban.

2.4.1. Alapfüggvények

Ebben a szekcióban a programszerkezetet meghatározó és a működéshez elengedhetetlen alprogramokat tárgyaljuk. A beépített alprogramok leírásáról szóló részekben a szubrutinok visszatérési értékét „→ visszaadott_érték” formában adjuk meg.

(Megjegyzés: Webtongue v1-ben bármely nem nulla számérték logikai egynek, a nulla pedig logikai nullának számít.)

- *let arg1 arg2*: arg1 := arg2; visszatérési értéke arg2; ha arg1 már létező változó, értéke felüldefiniálódik
- *var arg*: ha arg definiált → arg, egyébként → „N0pe.”

- *if arg1 arg2 arg3*: ha (arg1) akkor run arg2 egyebkent run arg3; arg2 és arg3 blokkok; csak a ténylegesen végrehajtott blokkot ellenőrzi az értelmező
- *eq arg1 arg2*: ha arg1 = arg2 → 1, egyébként → 0
- *gt arg1 arg2*: ha arg1 > arg2 → 1, egyébként → 0
- *gteq arg1 arg2*: ha arg1 >= arg2 → 1, egyébként → 0
- *and arg1 arg2*: ha arg1 && arg2 → 1, egyébként → 0
- *or arg1 arg2*: ha arg1 || arg2 → 1, egyébként → 0
- *not arg*: ha arg → 0, egyébként → 1
- *loop arg1 arg2*: addig ismétli arg2-t, amíg arg1 igaz; az ellenőrzés folyamatos arg2 értelmezése során! A *loop* segítségével valamennyi ismert ismétlő szerkezet elkészíthető, elől-hátul tesztelő ciklusok, számláló ciklus, stb.
- *run arg1 arg2 ... arg n*: a *run* használatának részletes kifejtése 2.3.4. III.) szekciójában olvasható
- *exit*: befejezi a program futását
- *error arg*: befejezi a futást az előbukkanó arg üzenet megjelenítése után
- *debug*: ki/be kapcsolja a hibakereső üzemmódot
- *ws*: szóköz értékű karakterláncot ad vissza
- *tab*: tabulátor értékű karakterláncot ad vissza

eq, *gt* és *gteq* elsősorban numerikus, másodsorban alfabetikus összehasonlítást végez.

2.4.2. Aritmetikai műveletek

Valamennyi aritmetikai művelet numerikus argumentumokkal dolgozik. A karakterlánc típusú paraméterek automatikusan konvertálódnak és számokként értelmeződnek.

- *inc arg*: → arg + 1
- *dec arg*: → arg - 1
- *add arg1 arg2*: → arg1 + arg2
- *sub arg1 arg2*: → arg1 - arg2
- *mul arg1 arg2*: → arg1 * arg2
- *div arg1 arg2*: → arg1 / arg2
- *mod arg1 arg2*: → arg1 % arg2
- *pow arg1 arg2*: → $arg1^{arg2}$
- *sin arg*: → sin arg; a trigonometrikus függvények radiánban számolnak; a fokban illetve radiánban megadott értékek közti konverzió a *todeg* és *torad* Mezzanine szubrutinokkal lehetséges
- *cos arg*: → cos arg
- *tan arg*: → tan arg
- *asin arg*: → asin arg
- *acos arg*: → acos arg
- *atan arg*: → atan arg
- *ln arg*: → logaritmus naturalis arg
- *e*: → Euler konstans
- *pi*: → π
- *abs arg*: → | arg |
- *ceil arg*: → ceil(arg)
- *floor arg*: → floor(arg)
- *round arg*: → round(arg)
- *rand arg*: → véletlen szám a [0..arg) intervallumból

2.4.3. Verem- és sormanipuláló függvények:

A verembéli elemek címzése úgy történik, hogy mindig a legfelső elem rendelkezik a legmagasabb címmel. A verem illetve szubrutinhívási sor mérete nincs korlátozva, annak csak a fizikai és virtuális memória szab határt.

- *push arg*: elhelyezi arg értékét a veremben
- *pop*: → és kiemeli a verem legfelső elemét
- *lift arg*: az arg. pozícióban levő elemet a verem tetejére emeli
- *drop arg*: a verem legfelső elemét az arg. pozícióba ejti
- *peek arg*: → az arg. helyen levő elem értékét
- *poke arg1 arg2*: verem[arg1] := arg2
- *top*: → a legfelső veremelem címét; üres verem esetén ez -1
- *dump*: kilistázza a verem illetve a paramétersor tartalmát
- *tostrack arg*: arg karakterlánc szavait egyesével a verembe helyezi
- *tostring arg*: → a verem arg. pozíciótól számított tartalmából karakterláncot képez, oly módon, hogy az egyes szavak a verem korábbi elemei lesznek; a feldolgozott elemeket eltávolítja a veremből
- *melt arg*: „feloldja" arg karakterláncot a veremben, vagyis annak minden egyes karakterét (a szeparáló szóközöket is beleértve) balról jobbra egyesével a verembe helyezi
- *merge arg*: a verem arg. pozíciójától kezdve egyetlen szót formál a veremelemekből, amit aztán →; a feldolgozott elemeket eltávolítja a veremből
- *get*: kiemeli és → a legelső elemet a szubrutinhívási sorból; kizárólag programblokk elején szerepelhet

2.4.4. Be-/kimeneti függvények, vegyes szubrutinok

- *date*: → szöveges formában az aktuális helyi időt és dátumot
- *print arg*: kiírja arg-ot az alapértelmezett kimenetre; a beágyazott `evalWebtongue()` JavaScript hívást alkalmazva ez a függvény visszatérési karakterlánca
- *println arg*: mint *print*, de soremeléssel
- *input arg*: előbukkanó ablakban bemeneti szövegdobozt jelenít meg, melynek alapértelmezett tartalma arg

2.5. Mezzanine függvények

A Mezzanaine szubrutinok elve a REBOL-ból származik (3.). Az elgondolás lényege az, hogy lehetőség szerint valósítsunk meg minimális számú, kizárólag életfontosságú funkciókat ellátó zárt kódú alprogramot, a nyelv szolgáltatásainak magját adó szubrutinok pedig már az adott programozási nyelven realizálódjanak. Mindezen túlmenően pedig – és ez a legfontosabb, – az így elkészített függvények legyenek elérhetőek a felhasználói programokban, valamilyen karakterlánc formájában. Azon kívül, hogy az így hozzáadott részek biztosan maximálisan igazodnak a nyelv logikájához és könnyedén elérhető a forráskódjuk, számos más előnyt is tartogat ez a megoldás. Ilyen például az a lehetőség, mely szerint bármely Mezzanine funkció kódja akár a programunk futása közben módosítható.

Mivel a Webtongue Mezzanine függvényei általában rövidek és beszédesek, ezért magyarázatuk helyett álljon itt az egyszerűsített forrásuk:

- *nop*: let nop { }
- *string arg1 arg2*: let string { get p get p2 if not gt p2 0 { error { invalid parameter for string } } nop let addr inc top loop p2 { push p let p2 dec p2 } tostring addr }

- *size arg*: let size { get p let addr inc top tostack p let ret sub top dec addr tostring addr let ret ret }
- *read arg1 arg2*: let read { get p get p2 let addr inc top tostack p let ret peek add addr p2 tostring addr let ret ret }
- *write arg1 arg2 arg3*: let write { get p get p2 get p3 let addr inc top tostack p lift add addr p2 pop push p3 drop add addr p2 tostring addr }
- *ins arg1 arg2 arg3*: let ins { get p get p2 get p3 let addr inc top tostack p push p3 drop add addr p2 tostring addr }
- *del arg1 arg2*: let del { get p get p2 let addr inc top tostack p lift add addr p2 pop tostring addr }
- *sqrt arg*: let sqrt { get p pow p 0.5 }
- *exp arg*: let exp { get p pow e p }
- *log arg*: let log { get p div ln p ln 10 }
- *torad arg*: let torad { get p div mul p pi 180 }
- *todeg arg*: let todeg { get p div mul p 180 pi }
- *dup*: let dup { if not gteq top 0 { error { cannot dup with empty stack } } nop push peek top }
- *swp*: let swp { if not gteq top 1 { error { cannot swp with one or less element in stack } } nop drop dec top }
- *rot*: let rot { if not gteq top 2 { error { cannot rot with two or less elements in stack } } nop drop sub top 2 }
- *clr*: let clr { if gteq top 0 { tostring 0 } nop }

Az 55 beépített függvény és 16 Mezzanine szubrutin megfelelő alapot ad a kísérletezésre illetve kisebb alkalmazások készítésére. Kialakításuk lehetővé teszi, hogy ne csak újabb funkcionalitással, hanem akár újabb szerkezeti elemekkel bővítsük a nyelvet, próbálkozzunk hát bátran!

3. AZ ÉRTELMEZŐPROGRAM

3.1. Áttekintés

Most hogy pontosan ismerjük a célnyelvet és értjük törekvéseinket, áttérhetünk a megvalósítás kérdésére.

Az első és legfontosabb feladat ezzel kapcsolatban lehetőségeink áttekintése, és az implementációs platform illetve nyelv kiválasztása. Ha igazán hordozható és széles körben elérhető programot szeretnénk készíteni, olyan célkörnyezetet kell választanunk, ami majd minden számítógépen rendelkezésre áll. Manapság az Internet jelenti az összekötő kapcsot a különböző hardverplatformok és operációs rendszerek között, annak hivatalosan bejegyzett (és sokszor önkényesen elferdített) szabványait minden rendszer támogatja valamilyen formában. (Sajnálatos, hogy jelenleg épp a piacvezető cégek azok, akik kevésbé elfogadható módon teszik ezt.)

Nyilvánvalóan adódik a felismerés: olyan programot kell kifejleszteni, ami Internetes böngészőkben fut. Ez újabb dilemmát vet fel, nevezetesen: szerver vagy kliens oldali megoldást lenne-e célszerű alkalmazni? Programunk felhasználói táborát semmiképp sem akarjuk csak az internetes eléréssel rendelkezőkére szűkíteni, így a kliens oldali megoldás mellet döntünk. Itt két fő alternatíva akad, megvalósíthatjuk az értelmezőprogramot Javaban illetve JavaScriptben. Egy Javaban implementált program futtatásához szükség lenne megfelelő futtatási környezet telepítésére, ez pedig nem minden esetben lehetséges (például egy nyilvános terminálon). Gyengén típusos felépítésével a JavaScript egyébként is jobban igazodik a Webtongue szerkezetéhez, JavaScriptben fogunk tehát programozni.

Természetesen néhány hátulütője is van választásunknak: a kiszemelt nyelv lassúbb és szűkebb lehetőségekkel bír, mint a Java. Szerencsére a sebesség nem döntő követelmény esetünkben, a funkcionalitás kibővítésére pedig több lehetőség is kínálkozik. A legkézenfekvőbb ezek közül a Java és JavaScript átjárhatóságának kihasználása a Java objektumkészletén keresztül. (A Webtongue eddigi fejlesztése során nem volt szükség rá, hogy éljünk is ezzel a lehetőségekkel.) Újabb negatívum a

JavaScript csökkentett objektumtámogatása. Nincs mód például osztályok öröklöttetésére vagy hozzáférési jogosultságok beállítására. Továbbá a rendelkezésre álló szövegértelmező generátorok – melyek közül talán az ANTLR (9.) a legjobb – sem támogatnak JavaScript formátumú kimenetet.

Mint minden döntésnek, ennek is megvannak az előnyei és árnyoldalai, elsődleges céljainkat szem előtt tartva azonban nem okoz gondot a választás: valószínűleg többet kell dolgoznunk és kevésbé lesz objektum-orientált a programunk, de a lehető legszélesebb körben használható eszköz lesz a végeredmény.

Az alábbi fejezeteknek nem az a célja, hogy pontosan, lépésről lépésre végigvezessen az implementáció menetén, hanem hogy általános képet adjon az értelmezőprogram működéséről, hogy bemutassa milyen nehézségekkel, illetve döntési szituációkkal kellett megbirkózni a fejlesztés során.

3.2. Az alapvető szerkezet kialakítása

Programunk számára olyan felépítés lenne ideális, melyben minden alkotóelemet és funkcionális egységet egy-egy objektum reprezentál. Egy későbbi, C++ vagy Java nyelven elkészített implementáció biztosan így is fog felépülni. Most azonban meg kell maradnunk lehetőségeink határain belül, és moduláris eszközökkel jó minőségű, átlátható programot írni.

Mivel a program Webtongue értelmező, nem pedig Webtongue fordító lesz, a nyelv feldolgozásának két legnagyobb részfeladatát nem tudjuk teljes mértékben szétbontani. A szintaktikai és szemantikai elemzésért felelős programrészeket bizonyos mértékig integrálni kell, hogy a folyamatos feldolgozás zavartalan lehessen. Ez a megközelítés módot ad arra is, hogy konzol-szerűen működő, parancssoros Webtongue feldolgozót készítsünk, ugyanezen értelmezőmag felhasználásával. (A Python hasonló konzoljának mintájára.) Az interaktív program, melyet e fejezet későbbi részeiben tárgyalunk ugyan nem e megközelítést követi, de kis átalakítással így is képes lenne működni.

Első lépésként létrehozzuk a változók bejegyzéseit tároló adattípust:

```
// variable class
function variable(name, argumentCount, isBuiltIn, value)
{
      this.name = name
      this.argumentCount = argumentCount
      this.isBuiltIn = isBuiltIn
      this.value = value
}
```

Definiáljuk a szükséges globális adatstruktúrákat, melyek az értelmező valamennyi újratöltődésekor frissülnek, visszaállnak kezdeti értékükre:

```
function initGlobals()
{
      wsre = /\s+/

      builtIn = true // variable types
      user = false   //

      // Error messages

      invalidBlockValue = "Invalid block value."
      invalidImmediateStringValue = "Invalid immediate
      string value."
      ...
      divisionByZero = "Division by zero."

      // Global variables

      wordArray = []
      wordCount = 0xdeadbeef
      wordCurrent = 0xdeadbeef

      variableArray = []
      variableCount = 0xdeadbeef
      variableLastLookup = 0xdeadbeef

      stop = false
      output = ""

      programQueue = []
      programStack = []

      workStack = [] // general stack
```

```
        lastReturnValue = "N0pe."

        subroutineLevel = 0
        loopLevel = 0

        currentSubroutine = "N0pe." // debug var
        debug = false

        delimiterBlockBegin = "{"
        delimiterBlockEnd   = "}"
        delimiterComment    = "**"

        mezzanineLength = 0xdeadbeef
}
```

Látható, hogy a beépített és felhasználói változók, valamint a különböző változótípusok ugyanabban a tömbben tárolódnak. A megfelelő jelzőváltozók lehetővé teszik egyértelmű megkülönböztetésüket az értelmezőprogram számára. A program vermének, paramétersorának, visszatérési értékeinek és egyáltalán valamennyi funkciójának helyes működéséhez szükséges segédváltozók mind az `initGlobals()` hívás során inicializálódnak.

A Webtongue forrásprogramok feldolgozása a kódban elhelyezett megjegyzések eltávolításával kezdődik, felhasználva a `wsre` reguláris kifejezést a forrásprogram szavakra bontásához, majd tömbbe szervezéséhez:

```
_wordArray = code.split(wsre)
wordArray = stripComments(_wordArray)
```

Ezután a beépített változók táblájának feltöltése következik, az alábbi formában:

```
variableArray.push(new variable("let", 2, builtIn,
     subLet))
variableArray.push(new variable("print", 1, builtIn,
     subPrint))
...
```

Végül az utasításonként történő feldolgozásért felelős főciklus szerepel, melynek futása az értelmezett forráskód befejeződéséig, vagy az első fellépő hibáig tart:

```
while (!stop)
{
   currentSubroutine = "main loop"
   if (wordCurrent >= wordCount)
   {
      stop = true
   }
   else
   {
      if ( variableIsDefined(parseNextWord()) )
      {
         if ( variableArray[variableLastLookup].isBuiltIn
            == user )
         {
            error(builtInSubroutineCallRequired, true)
            break
         }
         variableArray[variableLastLookup].value()
      }
      else
      {
         // Firefox bugfix
         if (wordArray[wordCurrent-1] == "") stop = true
         else error(invalidSubroutineReference, true)
      }
   }
}
```

Észrevehetjük, hogy a szövegfeldolgozást és értelmezést néhány általunk készített JavaScript szubrutin segíti. Sok más változatuk mellett ilyen a `lookAhead()` és a `parseNextWord()` hívás. Előbbi egy elem mélységű előreolvasást végez, másikuk a `wordArray[]` soron következő elemét olvassa be. Ezek csak kiragadott példák a nagyszámú hasonló profilú függvény közül, de mindenképp az interpreter működésének alapját képező megoldások.

3.3. A fejlesztés sarkalatos pontjai

A Webtongue beépített szubrutinjai hagyományos JavaScript alprogramok. Függvénymutatókként tárolódnak a `variableArray[]` tömbben, meghívásukkor az értelmező egyszerűen lefuttatja a megfelelő szubrutinokat. Szükséges argumentumaikat automatikusan beolvassák, azok típusellenőrzését is maguk végzik.

```
function subSin()
{
      currentSubroutine = "subSin()"

      var _value

      _value = readNextNumber()
      if (stop) return "N0pe."

      _value = Math.sin(_value)
      lastReturnValue = _value // +++ lastReturnValue +++
      return _value
}
```

Elsőként beállítjuk a végrehajtás pillanatnyi helyét jelző `currentSubroutine` változót, majd beolvassuk a szükséges argumentumokat (`_value = readNextNumber()` - itt egy specializált forrásszöveg kezelő függvényt használunk, amely paraméterét mindenképp számként értelmezve olvassa be). Ha nincs megfelelő argumentum, a futás leáll. Végül a globális `lastReturnValue` beállítását követően a kiszámított értékkel visszatérünk, a vezérlés pedig visszakerül a hívó ciklushoz.

Más a helyzet viszont a felhasználói függvényekkel. Feldolgozásuk jóval bonyolultabban történik. Az első nehézséget az okozza, hogy a végrehajtani kívánt programkód vagy egy változóban, egyszerű karakterláncként tárolódik, vagy közvetlen megadású blokként a forráskódban. Mindkét esetben ki kell lépnünk a forrásprogram eddig megszokott teréből, és az interpretert másik, különálló kódrészleten futtatni. Objektumorientált megoldással valamennyi végrehajtani kívánt programblokk saját értelmezővel rendelkezhetne, mely az adott osztály részét képezné. A tagfüggvények és metódusok hozzáférési jogosultságainak megfelelő megadásával a hibák lehetősége jelentősen lecsökkenne, tulajdonképpen meg is szűnne. JavaScript-ben ezek az eszközök nem állnak rendelkezésünkre, így olyan hagyományos megoldást kell találnunk, amivel hasonló eredmény érthető el.

A legrövidebb de legösszetettebb programkódot, egyúttal pedig a legtöbb hibalehetőséget olyan szerkezettel kapnánk, amelyben tetszőleges alprogram feldolgozását az interpreter főciklusa végezné. Ekkor rengeteg állapotjelző változót és számlálót kellene elmentenünk és időben visszaállítanunk az amúgy is mindenképp szükségesek mellett. Hogy a program ne legyen túlságosan bonyolult, továbbá a

szükséges védelemről is gondoskodjunk, jó döntés bizonyos funkciókhoz saját, önállóan implementált értelmezőt készíteni, a főciklust úgyszólván tehermentesíteni. A szubrutinhívások tetszőleges mélységben és kombinációban egymásba ágyazhatósága igen bonyolult helyzetet teremt, ezzel a módszerrel viszont elkerüljük a végrehajtás egymástól különböző szintjeiből eredő kavarodást. Az online Webtongue értelmezőprogram esetében ezek a szubrutinok az `evalWebtongue()` és a `subLoop()`. A megoldás a fenti nehézségek ellenére áttekinthető maradt, de hogy összetettségét érzékeltessük, álljon itt a `subLoop()` forrása (az egyes ciklusok értelmezése során pontosan ugyanezzel a feladattal szembesülünk ugyanis):

```
function subLoop()
{
   currentSubroutine = "subLoop()"

   var iterator
   var iteratorPtr
   var la
   var block
   var stop3

   iterator = parseNextWord()
   if (stop) return "N0pe." // +++ STOP +++

   if (variableIsDefined(iterator))
   {
   if (variableArray[variableLastLookup].isBuiltIn == builtIn)
      {
         error(invalidIteratorVariableForLoop, true)
         return "N0pe."
      }
      else
      {
      if ( _isNaN(variableArray[variableLastLookup].value) )
         {
            error(invalidIteratorVariableForLoop, true)
            return "N0pe."
         }
         else // number
         {
            // iterator ok
            iteratorPtr = variableLastLookup

            la = lookAhead()
            if (stop) return "N0pe."
```

```
if (la == "defined")
{
   parseNextWord()
   if (variableArray[variableLastLookup].isBuiltIn ==
      builtIn)
   {
      error(invalidBlockArgumentForLoop, true)
      return "N0pe."
   }
   else // user
   {
      if ( _isNaN(
         variableArray[variableLastLookup].value) )
      {
         block =
            variableArray[variableLastLookup].value
      }
      else // number
      {
         error(invalidBlockArgumentForLoop, true)
         return "N0pe."
      }
   }
}
else if (la == "string")
{
   block = parseNextString()
   if (stop) return "N0pe."
}
else
{
   parseNextWord()
   error(invalidBlockArgumentForLoop, true)
   return "N0pe."
}

// block ok
// ------------------------------------------------

workStack.push(wordArray)
workStack.push(wordCount)
workStack.push(wordCurrent)
wordArray = block.split(wsre)
wordCount = wordArray.length
if (wordArray[0] == "") wordCount = 0
wordCurrent = 0
loopLevel++

stop3 = false

while (!stop3)
{
```

```
   currentSubroutine = "subLoop()"

   // check iterator
   if ( _isNaN(variableArray[iteratorPtr].value) )
   {
      error(iteratorTypeMustNotBeChanged, true)
      break
   }
   else // iterator ok
   {
      if ( variableArray[iteratorPtr].value == 0 )
      {
         stop3 = true
      }
      else // iterator != 0
      {
         if (wordCurrent >= wordCount)
         {
            wordCurrent = 0 // loop
         }
         else
         {
            if (variableIsDefined(parseNextWord()))
            {
               if ( variableArray[
                  variableLastLookup].isBuiltIn == user
                  )
               {
                  error(builtInSubroutineCallRequired,
                     true)
                  break
               }
               variableArray[
                  variableLastLookup].value()
            }
            else
            {
               // Firefox bugfix
               if (wordArray[wordCurrent-1] == "")
                  stop3 = true
               else error(invalidSubroutineReference,
                  true)
            }
         }
      }
   }

   if (stop) stop3 = true
}

loopLevel--
wordCurrent = workStack.pop()
```

```
            wordCount = workStack.pop()
            wordArray = workStack.pop()
         }
      }
   }
   else
   {
      0error(invalidIteratorVariableForLoop, true)
   }

   return "N0pe."
}
```

A fejlesztés során igen kényelmes eszköznek bizonyultak a JavaScript maguktól konvertálódó, látszólag típus nélküli változói. Számos Webtongue funkció szinte magától működni kezdett a program növekedésével. Egy kritikus határon túl azonban, amikortól az egyes programhibák felderítése nehezebbé vált, nem várt nehézségek ütötték fel a fejüket. Kiderült, hogy a különböző JavaScript implementációk kizárólag egyvalamiben jelentenek közös platformot: valamennyi tartalmaz több-kevesebb hibát. A legtöbb hiba és inkompatibilitás a változók értékének automatikus konverziója, illetve értékük különböző értelmezése körül adódott. Ezeket a szabványtalan vagy egyszerűen rossz működésből adódó anomáliákat igen nehéz volt felderíteni, némelyik hiba egy-egy napot is hozzáadott a megvalósítás idejéhez.

3.4. A tervezés áttekintése

Az online értelmezőprogram forrását áttanulmányozva, valamint az eddigi fejezetek elolvasását követően jó rálátásunk lehet a Webtongue interpreter szerkezetére. Észrevehetjük, hogy a különböző szubrutinok három nagy funkciócsoportra oszlanak, ezek a kódszöveget kezelő és beolvasó függvények, a beépített szubrutinokat kezelő alprogramok és az értelmező keretét adó függvények illetve ciklusok. E három csoport egymástól függetlenül fejlődhetett, ebből a szempontból az értelmező teljesen modulárisnak tekinthető. Implementációjuk időben sem esett egybe, illetve csak részben, amikor egyikük bővítése megkövetelt bizonyos szolgáltatásokat a többi csoport függvényeitől. A három osztályra vonatkozó követelmények is különbözőek voltak, de az implementáció során adaptívan változtak és formálódtak, így az eredmény teljesen

egységesé vált. A 2.2. fejezet egyszerű felépítésre vonatkozó alapgondolata a program megvalósítása idején is meghatározó volt, megpróbáltam a forráskódot illetve koncepciót olyan áttekinthetőnek megőrizni, amennyire csak lehetséges.

A tervezés menetében felfedezhető némi ciklikusság. Egy-egy új funkció hozzáadásakor, mikor az a már meglévőkkel elkezdett együttműködni, időnként új gondolatok és elvek merültek fel a specifikációban leírtakkal szemben. Az ilyen esetekben mindig sor került a tervezet felülvizsgálatára, és ha szükséges volt, módosítására.

Jó példa erre a `probe` függvény, mely az eredeti Webtongue specifikációnak szerves részét képezte, és kis híján helyet is kapott a nyelvben. Funkciója az lett volna, hogy ha segítségével deklarálunk egy változót, az értelmező a változó értékét minden hivatkozáskor az eredeti definíciós formulának megfelelően újraszámlálja. Egy példán keresztül jobban illusztrálható ez a működés:

```
let i 10
let j sub 10 probe i

loop i
{
   print i print j
   let i dec i
}
```

A fenti programrészlet kimenete 10 0 9 1 8 2 7 3… stb. lehetett volna. Nem lett volna szükség rá, hogy a `j` értékét meghatározó `sub 10 i` kódot külön blokkban tároljuk, a `probe` pedig jelezte volna az értelmezőprogramnak, hogy minden alkalommal értékelje újra az így kapott inverz iterátort. Mikor azonban a fejlesztésben a `run` függvény implementációjáig értem, világos volt, hogy ez a konstrukció felesleges, hiszen az alábbi megoldás tökéletesen helyettesíti:

```
let i 10
let j { sub 10 i }

loop i
{
   print i print run j
   let i dec i
}
```

Fontos még kitérnünk a Mezzanine szubrutinok kialakítására. Ezek tökéletesen elkülöníthetők a fejlesztés többi részétől, hiszen csupán Webtongue nyelven írt függvényekről van szó, amelyeket felhasználói Webtongue programunk elején az értelmező automatikusan létrehoz (`let` hívások futtatásával). Azért jelentenek mégis fontos tényezőt a program tervezésével kapcsolatban, mert végig az volt a cél, hogy a funkcionalitásnak olyan nagy részét vegyék át ezek a szubrutinok, amennyit csak lehetséges. Ezt a törekvést siker koronázta, hiszen sok valóban fontos és alapvető alprogram született meg tisztán Webtongue implementációban, jóval rövidebb és áttekinthetőbb kóddal, mint JavaScriptben (`string`, `size`, `read`, `write`…)

3.5. Az interaktív online értelmezőprogram

Az elkészült interpreter igen egyszerűen HTML oldalba ágyazható. Működéséhez mindössze egy bemeneti karakterláncra van szükség, kimenetét szintén karakterlánc formájában adja (`outstring = evalWebtongue(instring);`). Ezt a lehetőséget kihasználva készítettem el az interaktív értelmezőprogramot is a Webtongue nyelvhez. Kezelése nagyon egyszerű, mindössze három gomb segítségével vezérelhető, egyikük a nyelv elektronikus, angol nyelvű leírását adja (Help), a következő új programot kezd (New), az utolsó pedig végrehajtja a forráskódnak megfelelő szövegdobozba gépelt vagy másolt Webtongue kódot (Execute!). A kimenet szintén egy szövegdobozban jelenik meg. A program a CD melléklet webtongue_v1 könyvtárában található index.html állomány megtekintésével indítható. A helyes működéshez alapkövetelmény JavaScript 1.5-öt támogató böngészőprogram használata. Ha beágyazott formában szeretnénk használni a Webtongue értelmezőt, szükséges a routines.js file importálása. A file végén található `startup()`, `handleNew()` és

`handleExecute()` függvények lehetőséget biztosít új interaktív interpreter készítésére is, feltéve, hogy a bennük hivatkozott HTML form elemek léteznek a céldokumentumban.

4. TESZTELÉS

4.1. A tesztelés módszere

A tesztelés szempontjából a JavaScript alábbi tulajdonságai a leglényegesebbek:

- A JavaScript a Webtongue-hoz hasonlóan szintén értelmezett nyelv. Egy JavaScript program futása alkalmával csak az éppen végrehajtott sorok kerülnek kiértékelésre, így hosszabb, kiterjedt funkcionalitású programrészlet tesztelése során roppant körültekintően kell eljárnunk. A futtatás ilyen tulajdonsága miatt nem csak a szemantikai, de a szintaktikai hibák sem derülnek ki, amíg az érintett rész végre nem hajtódik - ez még tovább nehezíti a dolgunkat. Az egyetlen célravezető út, ha módszeresen, az összes lehetőséget végigjárva kipróbáljuk az adott szakaszt. Ez meglehetősen sok időt és átgondolt tesztelési tervet igényel, aminek a kidolgozása adott esetben jóval tovább tart, mint a programtervezés többi fázisa együttvéve. Jelen feladatban megpróbáltam a lehetőségekhez képest legsokoldalúbb tesztprogramokat írni, de természetesen rejtett programhibák maradhattak az implementációban.
- Tovább bonyolítja a helyzetet, hogy az egyes hibák, pontatlanságok kölcsönösen hatást gyakorolhatnak egymásra, van, hogy több hiba együttes eredménye lesz csak látható a felhasználó számára. Ilyen eset fordult elő a visszatérési értékkel rendelkező függvényhívás fejlesztése során. A hiba a ciklus végét jelző változók helytelen kezeléséből adódott, ráadásul összekapcsolódott a JavaScript egy implementációs hibájával is. A megoldás megtalálása három napba telt.
- Az egyes JavaScript megvalósítások inkompatibilitásai jelentik a harmadik, és talán legveszélyesebb hibaforrást. Néhány napnyi használat után az ember megszokja, hogy az eszköz, amit használ, csaknem olyan megbízhatatlan,

mint ő maga. Ez igazán nem jó érzés, mondjuk az ANSI C stabil hátteréhez képest.

Ezek azok a tényezők, amik a tesztelés általános elvein kívül legjobban rányomják bélyegüket a jelenlegi munkára. Mindezeknek köszönhetően a program ellenőrzése folyamatosan zajlott a fejlesztéssel párhuzamosan. Amint új elemmel bővült az értelmezőprogram, annak minden aspektusát szükséges volt ellenőrizni először önmagában, aztán a meglévő elemekkel együttműködve. Ahogy a program nőtt, ez a feladat egyre nagyobbá vált, a későbbiekben a rendelkezésre álló idő hiányában már nem is volt teljes körű, de természetesen végig törekedtem rá, hogy a példaprogramok és különböző tesztek a lehető legtöbb területre kiterjedjenek. Tovább bővítették a tesztfutások sorát azon esetek, amikor kívánatos cél volt a hibás működés. Az egyes anomáliák helyes kezelését vizsgálta ez a fajta teszt.

Az implementáció befejeződésével a kész programot is ellenőrizni kellett, ekkor, az utólagos dokumentálással és próbákkal együtt már csak ez volt a feladat, remélem, hogy a végeredmény a lehetőségekhez képest maximálisan hibamentes lett.

A tesztelés két platformon zajlott, egyik böngésző a Mozilla Firefox 0.8 volt, amely ma talán a vonatkozó szabványokat leghívebben megvalósító szoftver, másik pedig elterjedtsége miatt az Internet Explorer 6.0-ás változata.

4.2. Tesztelési példák

Az egyes függvények működésének módszeres vizsgálata nem volna túlságosan olvasmányos leírás, még ha ez tette is ki a tesztelésre szánt idő nagyobb részét. (Pl. a `tostack` és `tostring` szubrutinok esetén módszeres műveletek üres veremmel; különböző mélységben, különböző típusú elemekkel feltöltött veremmel; külön-külön, egymással együttműködve; közvetlen megadású argumentummal vagy változó argumentummal; számokkal és blokkokkal, stb.) Helyette álljon itt két összetettebb példa, az elsőben sok függvényhívás és ciklikus feladat végrehajtás látható, második célja a rekurzió helyes kezelésének szemléltetése:

```
let fibo
{
   get fibo_n

   let oldfib -1
   let fib 1

   loop fibo_n
   {
      let newfib add oldfib fib
      let oldfib fib
      let fib newfib
      let fibo_n dec fibo_n
   }
   let fib fib
}

let printfibo
{
   get n

   push add n 2
   let i { sub peek top n }
   loop n { print run fibo run i print ws let n dec n }
   pop
}

run printfibo 10   ** a kimenet: 1 1 2 3 5 8 13 21 34 55 **
```

```
let fac
{
   get n

   if not gteq n 0
      { let ret 0 }
      { if eq n 0
         { let ret 1 }
         { let ret mul n run fac dec n }
      }
   let ret ret
}

print run fac 6   ** a kimenet: 720 **
```

5. A TOVÁBBFEJLESZTÉS LEHETŐSÉGEI

A Webtongue számos jelenlegi megoldása tovább alakítható. Úgy az elméleti alapok, mint a gyakrolati megvalósítás területén kínálkoznak bővítési lehetőségek.

Elsőként kezdjük a nyelv specifikációjának tárgyalásával. A Webtongue használata során biztosan kialakul majd, melyek azok a részei, amelyek legjobban megfelelnek eredeti céljainknak, és melyek azok, amiket vagy át kell alakítani, vagy ki kell cserélni. Az is megeshet, hogy lesznek területek, amelyek bővítést igényelnek. A nyelv felépítéséből adódóan igen könnyű új funkcionalitással, esetleg az eddigiektől gyökeresen eltérő tulajdonságokkal felruházni. Sok olyan pozitív tulajdonságot ötvöz, amik önmagukban is ütőképes eszközt jelentenek, együtt azonban kifejezetten hatékonyak. Jelenlegi egyszerű szerkezete elképzelhető, hogy későbbi fejlődése útjába áll, akkor majd szükségszerűen kötöttebb szintaxisúvá fog alakulni. Még valószínűbb az a jövőbeli megoldás, amely szerint a Webtongue egy köré épült keretrendszerrel együtt fog létezni, és lehetőség lesz értelmezett szintaxisának igen könnyű módosítására. Még ideálisabb esetben magából a nyelvből lenne lehetséges saját szintaxisának befolyásolása. A láthatóság koncepciója majdnem biztosan változni fog, amint a szerkezet bármennyivel is bonyolódni kezd. Összességében: a most még kísérleti nyelv kis átalakításokkal egészen hatékony eszközzé tehető, amit jelenleg olyan elegyként képzelek, ami a Lisp és REBOL házasságából születne.

Az implementáció terén szintén sok javítanivaló akad, bizonyos algoritmusok korántsem a leghatékonyabbak, továbbá az egész megvalósítás átültethető lenne egy olyan programozási nyelvre, amely jóval gyorsabb és megbízhatóbb futást eredményez. Egy például C++-ban íródott implementáció lehetővé tenné a Webtongue felhasználói programokba ágyazását, amelyek vezérlésére tökéletesen alkalmas volna.

Hogy kiléphessen elméleti/kísérleti státuszából, vagy ha úgy tetszik gyermekcipőjéből, funkcionalitásának drasztikus bővítésére lenne szükség. Némely programozási nyelv úgy oldja meg ezt a problémát, hogy egyszerűen egy másik, hatalmas eszközkészlettel bíró nyelv lehetőségeivel él (pl. a Java objektumkészletét használja fel). Efféle megoldás a Webtongue esetében is elképzelő lenne.

Meggyőződésem, hogy a programozási nyelvek elmélete tartogat még olyan módszert, amely a jelenleg elterjedt megoldásoknál sokkal jobban igazodik az emberi gondolkodáshoz. Az is bizonyos, hogy egy ilyen nyelv kidolgozása sokévnyi kutatómunkát, és még több tapasztalatot igényel, ráadásul nem csupán egyetlen emberét, de mindenképpen lehetséges. A Webtongue természetesen messze nem tör ilyen babérokra, de ahhoz hasznos eszköz lehet, hogy a különböző elveket megismerjük és kipróbáljuk, új elképzelésekkel bővítsük ki egyébként igen egyszerű szerkezetét, vagy épp régen gyökeret vert gondolatokat ismerjünk meg behatóbban a segítségével.

IRODALOMJEGYZÉK

[1.] Abelson, H., Sussman, G. J. and Sussman, J. (1996). *Structure and Interpretation of Computer Programs.* MIT Press.

[2.] ANSI Technical Committee X3J14. (1994). *American National Standard of Information Systems/Programming Languages/Forth, last draft.* American National Standards Institute, Inc.

[3.] Goldman, E., Blanton, J. and Renaldi, W. (2000). *REBOL: The Official Guide.* McGraw-Hill Osborne Media.

[4.] Ierusalimschy, R. (2003). *Programming in Lua.* Lua.org

[5.] Koopman, P. J., Jr. (1993). *A Brief Introduction to Forth.* ACM, Second History of Programming Languages Conference (HOPL-II), Boston MA.

[6.] Moncur, M. (2002). *Tanuljuk meg a JavaScript használatát 24 óra alatt.* Kiskapu Kft.

[7.] Oortmerssen, Wouter van (1993, 2000). *The FALSE Programming Language, The Aardappel Programming Language.* http://wouter.fov120.com/proglang/index.html

[8.] Parr, Terence (2004). *ANTLR Reference Manual.* antlr.org

[9.] Winston, P. and Horn, B. (1989). *Lisp.* Addison-Wesley Pub Co.