Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Petra Hvala

WebAssembly: zbirnik za spletne brskalnike

diplomsko delo

interdisciplinarni univerzitetni
študijski program prve stopnje
računalništvo in matematika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana 2020

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

WebAssembly: zbirnik za spletne brskalnike

Tematika naloge:

WebAssembly predstavlja nov način za pisanje spletnih aplikacij. Ker ga podpirajo vsi vodilni brskalniki, se čedalje bolj uveljavlja. Preučite standard, ki opisuje trenutno izvedbo WebAssemblyja, in orodja, ki že omogočajo njegovo uporabo. Izberite nek majhen programski jezik in sestavite preprost prevajalnik za prevajanje tega jezika v WebAssembly. Pokažite, kako lahko prevedene programe izvedemo v brskalniku.

Mojim čebelam.

Kazalo

Povzetek

Naslov: WebAssembly: zbirnik za spletne brskalnike

Avtorica: Petra Hvala

V diplomskem delu je predstavljen prevajalnik iz programskega jezika Brainf*ck v WebAssembly. Za izdelavo prevajalnika smo najprej morali spoznati strukturo WebAssemblyja, tako v tekstovni kot binarni obliki, JavaScript vmesnik ter preizkusiti delovanje že obstoječih prevajalnikov za programske jezike C/C++ ter Rust in zbirke programskih orodij za WebAssembly. Prevajalnik smo implementirali v WebAssembly tekstovni obliki in ga prevedli v binarno obliko s pomočjo orodja WABT. Sestavni del prevajalnika sta tudi JavaScript vmesnik, ki WebAssemblyjev modul naloži s strežnika in z njim komunicira, ter html stran za spletni prikaz. Prevajalnik smo uspešno implementirali in brez težav prevede Brainf*ck v izvedljive programe.

Ključne besede: WebAssembly, Brainf*ck, prevajalnik.

Abstract

Title: WebAssembly: assembler for web browsers

Author: Petra Hvala

In this work is represented a Brainf*ck to WebAssembly compiler. To build a compiler we had to learn about WebAssembly binary and text format, JavaScript API and tried out already existing compilers for C/C++ and Rust programing languages and WebAssembly toolkits. Compiler was implemented in WebAssembly text format and with toolkit WABT compiled to WebAssembly binary format. JavaScript API, which loads WebAssembly module from server and communicates with it, and html page for web display are the other two components of compiler. Compiler was successfully implemented. It compiles Brainf*ck into executable programs without any issues.

Keywords: WebAssembly, Brainf*ck, compiler.

Poglavje 1

Uvod

Vse odkar je človek izumil računalnik, je razvoj tega naraščal. Najprej smo z računalnikom komunicirali le z ničlami in enkami. Komunikacija se je poenostavila z zbirnikom. Nato smo napredovali na nizkonivojske programske jezike, kot so Fortran, C in drugi. Z računalnikom smo se povezali z drugim računalnikom v omrežju in nastal je internet. Internet je prinesel storitve, kot sta e-pošta in svetovni splet. Na svetovnem spletu so bile sprva le statične strani, toda novi programski jeziki so omogočili tudi interakcijo z uporabnikom.

Leta 1995 je nastal JavaScript, edini visokonivojski programski jezik, ki ga je podpiral svetovni splet. „Predviden je bil kot majhen skriptni jezik za interakcijo s HTML DOM.“ [8] Ta majhen jezik pa je postal večji in ga z nešteto izboljšavami uporabljamo še danes. Čeprav vsaj zaenkrat potreben programski jezik pa se je JavaScript izkazal kot občasno nezanesljiv in počasen za današnje standarde. Rešitev za te probleme smo iskali z izboljšavami, kot je asm.js, ki je nizkonivojska podkategorija JavaScripta, ki omogoča, da se lahko vanjo prevede programski jezik C in drugi.

Druga rešitev se je pojavila leta 2017 v obliki nove nizkonivojske kode, imenovane WebAssembly. „WebAssembly naslovi problem varne, hitre, prenosljive nizkonivojske kode na spletu.“ [1] To so ravno rešitve za pereče težave JavaScripta. Poleg tega je WebAssembly predvsem ciljni jezik prevajanja iz

drugih programskih jezikov, ki je že dosežen s programskima jezikoma C in C++. Podprt je „s strani vseh modernih spletnih brskalnikov.“ [8]

To novo tehnologijo smo spoznali in jo v praktičnem delu uporabili za prevod drugih programskih jezikov za uporabo na spletu.

Najprej si v tej nalogi ogledamo strukturo WebAssemblyja (tekstovno in binarno obliko), predstavimo WebAssembly JavaScript vmesnik, ki je zaenkrat ključen za uporabo WebAssemblyja na spletu, ter predstavimo že implementirane prevajalnike v WebAssembly: Emscripten, Rust in druga orodja.

V praktičnem delu predstavimo prevajalnik, zapisan v WebAssemblyju, ki prevaja programski jezik Brainf*ck v WebAssembly.

Poglavje 2

WebAssembly

WebAssembly (kratko Wasm) je varna in prenosljiva oblika nizkonivojske kode, ki je zasnovana tako, da ima kompaktno predstavitev in učinkovito izvedbo [9]. Ta nov odprti standard so razvili na mednarodnem inštitutu W3C (World Wide Web Consortium) v skupini WebAssembly Working Group, v osnovi pa je namenjen za splet. Pri njegovem oblikovanju so bili glavni cilji hitrost, varnost, dobra definiranost, neodvisnost od strojne opreme, neodvisnost od jezika, neodvisnost od računalniškega okolja in odprtost.

Na spletu je ciljni operacijski sistem in procesor neznan, zato so pri snovanju vzeli skupne značilnosti vseh pogosteje uporabljenih arhitektur centralno procesnih enot. Prenosljivost WebAssemblyja predpostavlja, da okolje izvajanja ponuja karakteristike, kot so:

- bajtno naslavljanje pomnilnika,
- dvojiški komplement predznačenih celih števil, zapisan z 32-bitnim ali 64-bitnim številom,
- 32-bitna in 64-bitna števila s plavajočo vejico, zapisana po standardu IEEE 754-2008 [5],
- bajtni vrstni red po pravilu tankega konca,

- pomnilniške regije, ki so lahko učinkovito naslovljive z 32-bitnimi kazalci,

- varna ločitev med WebAssemblyjevim modulom in drugimi moduli oziroma procesi, ki se izvajajo na istem stroju.

WebAssembly lahko deluje v kateremkoli računalniškem okolju, ki zanj definira programski vmesnik. Na spletu je definiran WebAssembly JavaScript vmesnik, ki ga bomo spoznali v poglavju 3.

WebAssembly je predvsem ciljni jezik prevajalnikov iz drugih programskih jezikov. Tako je programerjev stik z njim posreden. Prevajalnikov cilj je prevesti v binarno obliko WebAssemblyja, za lažje razumevanje kode in razhroščevanje pa so ustvarili tudi tekstovno obliko, ki je ekvivalentna binarni.

Program WebAssemblyja je imenovan modul. Moduli so razdeljeni v več segmentov oziroma sekcij. Vsaka sekcija je opcijska, zato je najmanjši veljaven program brez definiranih sekcij. Za program, ki opravi nekaj operacij, pa so nujno potrebne tri sekcije: sekcija tipov, funkcij in kode.

Ukaze, ki se bodo izvajali, zapišemo v sekcijo kode. Ločimo jih na navadne ukaze, ki operirajo z vrednostmi, in strukturirane ukaze, ki vodijo in spreminjajo potek izvajanja. Ukazi operande za izvajanje jemljejo s sklada – WebAssembly je torej skladovni stroj. Ukazi so podobni tistim, ki jih imajo arhitekture dejanskih strojev. Glavna razlika je operiranje s skladom in ne z registri.

Vsakega od elementov v programu definiramo s tipom: poznamo tip vrednosti, tip funkcije, pomnilniški tip in tip tabel. V WebAssemblyju imamo zgolj 4 tipe vrednosti – 32- in 64-bitna cela števila ter 32- in 64-bitna števila s plavajočo vejico, ki so tudi vhodne in izhodne vrednosti WebAssemblyjevih funkcij. Pri številih s plavajočo vejico se WebAssembly opira na že omenjeni standard IEEE 754-2008 [5]. Standard Unicode [11] se uporablja za zapis tekstovne oblike WebAssemblyja ter imena uvoženih in izvoženih elementov, katerih znaki so kodirani z UTF-8.

2.1 Struktura

WebAssembly je predstavljen z dvema ekvivalentnima oblikama – binarno in tekstovno. Binarna oblika se uporablja v aplikacijah in se zapiše v datoteko s priporočeno končnico `wasm`. Tekstovno obliko zapišemo v datoteko s priporočeno končnico `wat`, uporablja pa se za lažje razhroščevanje ali direktno programiranje v WebAssemblyju.

Notacija za predstavljeni obliki je sledeča:

- Glavni simboli binarne oblike so bajti, ki so predstavljeni s šestnajstiškim zapisom: `0x0F`.
- Glavni simboli tekstovne oblike so zaporedja znakov, ki so zapisana med enojnimi narekovaji: `'module'`.
- Definicije so zapisane kot: *symbol* ::= $B_1 \Rightarrow A_1 | ... | B_n \Rightarrow A_n$, kjer je *symbol* definiran s spremenljivkami v B_i in ga predstavlja A_i. A_i je abstraktna sintaksa za *symbol*, izražena s spremenljivkami dejanske sintakse v B_i.
- A^n je zaporedje A-jev, kjer je $n \geq 0$.
- A^* je lahko prazno zaporedje A-jev, kjer število iteracij ni pomembno.
- A^+ je neprazno zaporedje A-jev.
- $A^?$ je opcijska pojavitev A-ja.
- $concat(s^*)$ je zaporedje, tvorjeno s stikom vseh s^i v s^*.

Vsako spoznavanje novega jezika se navadno prične s programom, ki izpiše „Hello, world!“ Program hello.wat je zapisan v primeru 2.1 (tekstovna oblika) ter ekvivalenten program hello.wasm v primeru 2.2 (binarna oblika). Do izpisa pridemo tako, da v ustvarjeni pomnilnik zapišemo „Hello, world!“, nato pa s funkcijo $hello kličemo zunanjo funkcijo $print, ki ji podamo začetno pozicijo in dolžino zapisa. Zunanjemu okolju sta na voljo pomnilnik in funkcija

$hello, hkrati pa mora zunanje okolje definirati funkcijo $print, ki poskrbi za izpis. Ko je vse povezano, v zunanjem okolju samo kličemo funkcijo $hello. Kako pripravimo zunanje okolje, si bomo ogledali v poglavju 3 preko Javascript vmesnika.

```
(module
  (import "imports" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 0) "Hello, world!")
  (func $hello (export "hello")
    i32.const 0
    i32.const 13
    call $print
  )
```

Primer 2.1: Modul z nekaterimi segmenti, program hello.wat.

```
0| 00 61 73 6D 01 00 00 00
1| 01 09 02 60 02 7F 7F 00 60 00 00
2| 02 11 01 07 69 6D 70 6F 72 74 73 05 70 72 69 6E 74 00 00
3| 03 02 01 01
5| 05 03 01 00 01
7| 07 12 02 06 6D 65 6D 6F 72 79 02 00 05 68 65 6C 6C 6F 00 01
10| 0A 0A 01 08 00 41 00 41 0D 10 00 0B
11| 0B 13 01 00 41 00 0B 0D 48 65 6C 6C 6F 2C 20 77 6F 72 6C
    64 21
```

Primer 2.2: Program hello.wasm, šestnajstiški zapis binarne oblike primera 2.1.

2.1.1 Moduli

Programi so v WebAssemblyju organizirani v module. Modul lahko vsebuje definicije pomnilnika, tabele, globalnih spremenljivk, funkcij in tipov. Poleg tega lahko deklarira uvoze in izvoze, pri inicializaciji pa lahko začetne vrednosti pomnilnika in tabele definiramo s pomnilniškimi oziroma elementnimi segmenti. Segment start nam omogoča deklarirati funkcijo, ki se požene takoj po inicializaciji.

V binarni obliki se modul najprej začne s čarobnim številom `magic` in verzijo WebAssemblyja `version`, nato pa je organiziran v sekcije. Sekcije

morajo biti zapisane po vrsti glede na njihov identifikator (1–11). Izjema so sekcije po meri, ki se lahko vrinejo kjerkoli v vrsti drugih sekcij. Modul je torej sestavljen tako:

```
magic   ::= 0x00 0x61 0x73 0x6d
version ::= 0x01 0x00 0x00 0x00
module  ::= magic version
            customsec* typesec customsec* importsec
            customsec* funcsec customsec* tablesec
            customsec* memsec customsec* globalsec
            customsec* exportsec customsec* startsec
            customsec* elemsec customsec* codesec
            customsec* datasec customsec*
```

Sekcije predstavljajo vsebino *cont*, katere zapis se razlikuje od sekcije do sekcije. Vsem sekcijam pa je skupno, da se najprej začnejo z identifikacijsko številko sekcije `N`, ki zaseda 1 bajt, ter dolžino vsebine sekcije, v bajtih zapisano z `u32`. Tem bajtom sledi vsebina *cont*:

$$\begin{array}{lll} \texttt{section}_{\texttt{N}}(\texttt{B}) & ::= N{:}\texttt{byte}\ \mathit{size}{:}\texttt{u32}\ \mathit{cont}{:}\texttt{B} & \Rightarrow \mathit{cont} \\ & \mid \quad \epsilon & \Rightarrow \epsilon \end{array}$$

Sekcija je lahko tudi prazna, zato najmanjši veljaven program sestoji iz čarobnega števila in verzije ali v tekstovni obliki `(module)`. Vsaka od sekcij se praviloma enači s segmenti modula, izjema so zgolj funkcije, saj za njihovo definicijo potrebujemo dve sekciji.

Za razliko od binarne oblike se lahko v tekstovni obliki segmenti modula pojavijo v kateremkoli vrstnem redu. Modulu pa lahko pripišemo tudi identifikator. Rezervirana beseda za modul je `module`, njegovo definicijo, kot tudi vse segmente modula, pa zapišemo v oklepajih:

```
module      ::= '(' 'module' id? (m:modulefield)* ')'
modulefield ::= type | import | func | table
                mem | global | export | start
                elem | data
```

Indeksi

Vsaka od definicij nosi svojo indeksno številko x, zapisano z `u32`. Indeksi tipov `typeidx`, funkcij `funcidx`, pomnilnika `memidx`, tabele `tableidx` in globalnih spremenljivk `globalidx` so dosegljivi povsod v modulu. Indeksi pripadajo tudi uvoženim elementom, ki dobijo svojo indeksno številko pred drugimi, definiranimi v modulu.

Indeksni prostor imajo tudi lokalne spremenljivke `localidx`, ki so dosegljive zgolj v funkciji, ki jih definira.

Svoje indekse prejmejo tudi strukturirani ukazi `block`, `loop` in `if`, ki pa so dosegljivi zgolj znotraj teh ukazov. Ko so ti ukazi ugnezdeni, ima najbolj notranji ukaz indeks 0, ukaz, ki ga objema, ima indeks 1, in vsak naslednji ukaz ima indeksno število za ena večje od prejšnjega.

V tekstovni obliki lahko namesto indeksnih številk pripišemo simbolični identifikator v, ki kaže na indeksno številko x. Simbolični identifikator prepreči morebitno zmedo pri naslavljanju pravega strukturiranega ukaza.

V binarni obliki je torej indeks določen s številom x, v tekstovni obliki pa lahko število nadomestimo s simboličnim identifikatorjem v:

$$
\begin{array}{lll}
\texttt{typeidx} & ::= x{:}\texttt{u32} \Rightarrow x \\
& \mid \; v{:}\texttt{id} \Rightarrow x \\
\texttt{funcidx} & ::= x{:}\texttt{u32} \Rightarrow x \\
& \mid \; v{:}\texttt{id} \Rightarrow x \\
\texttt{tableidx} & ::= x{:}\texttt{u32} \Rightarrow x \\
& \mid \; v{:}\texttt{id} \Rightarrow x \\
\texttt{memidx} & ::= x{:}\texttt{u32} \Rightarrow x \\
& \mid \; v{:}\texttt{id} \Rightarrow x \\
\texttt{globalidx} & ::= x{:}\texttt{u32} \Rightarrow x \\
& \mid \; v{:}\texttt{id} \Rightarrow x \\
\texttt{localidx} & ::= x{:}\texttt{u32} \Rightarrow x \\
& \mid \; v{:}\texttt{id} \Rightarrow x \\
\texttt{labelidx} & ::= l{:}\texttt{u32} \Rightarrow l \\
& \mid \; v{:}\texttt{id} \Rightarrow l
\end{array}
$$

Sekcija po meri

Sekcija po meri ima identifikator 0. V modulu se lahko pojavi večkrat, a WebAssemblyjeva semantika jih ignorira. Sekcija po meri največkrat nosi podatke za razhroščevanje ali pa je namenjena za druge vtičnike.

Največkrat uporabljena sekcija po meri je sekcija imen, ki lahko stoji samo za zadnjo sekcijo – sekcijo podatkov. V njej so zapisani simbolični identifikatorji, ki so vidni v tekstovni obliki, za lažje povezovanje podatkov med razhroščevanjem.

Sekcija po meri `customsec` sestoji iz imena sekcije `name` in zaporedja podatkov `byte`*:

$$\begin{aligned} \texttt{customsec} &::= \texttt{section}_0(\texttt{custom}) \\ \texttt{custom} &::= \texttt{name}\ \texttt{byte}^* \end{aligned}$$

Tipi

Ta komponenta definira tipe funkcij. Definirani morajo biti tipi tako lokalnih kot uvoženih funkcij.

V binarni obliki tej komponenti pripada sekcija tipov `typesec`, ki ima identifikator 1, njena vsebina pa je vektor funkcijskih tipov `functype`:

$$\texttt{typesec} ::= ft^*{:}\texttt{section}_1(\texttt{vec}(\texttt{functype})) \Rightarrow ft^*$$

Primer 2.2 prikaže zapis sekcije tipov za dva tipa. Prvi tip funkcije ima dva parametra tipa `i32` in nič rezultatov, drugi pa nima ne parametrov ne rezultatov.

V tekstovni obliki ima rezervirano besedo `type`, zapis ilustrira primer 2.3, zapišemo pa ga tako:

$$\texttt{type} ::= \texttt{'('}\ \texttt{'type'}\ \texttt{id}^?\ ft{:}\texttt{functype}\ \texttt{')'} \Rightarrow ft$$

Uporaba tipov

Uporaba tipov `typeuse` je referenca na definicijo tipa. Uporablja se v tekstovni obliki pri definiciji funkcij, definiciji uvoznih funkcij in pri ukazu

`call_indirect`. Njen tip definiramo v oklepajih z rezervirano besedo `type` ter z indeksom tipa x ali z indeksom tipa x in vektorjem parametrov t_1 in rezultatov t_2, ki se morajo ujemati s tipom x:

$$\texttt{typeuse} ::= \texttt{'('}\ \texttt{'type'}\ x{:}\texttt{typeidx}_\texttt{I}\ \texttt{')'} \Rightarrow x$$
$$\mid\ \texttt{'('}\ \texttt{'type'}\ x{:}\texttt{typeidx}_\texttt{I}\ \texttt{')'}\ (t_1{:}\texttt{param})^*\ (t_2{:}\texttt{result})^* \Rightarrow x$$

```
(module
  (type $mytype (func (param i32 i32) (result i32)))
  (func (export "addTwo") (type $mytype)
    local.get 0
    local.get 1
    i32.add)
)
```

Primer 2.3: Definicija tipa in njegova uporaba v definiciji funkcije.

V definiciji funkcije se lahko uporaba tipov opusti z vstavljenim vektorjem parametrov in rezultatov. V tem primeru se indeks tipa vstavi samodejno. Tako bi pri primeru 2.3 lahko izpustili definicijo tipa in na mestu definicije funkcije zapisali:
`(func (export "addtwo") (param i32 i32) (result i32) ...)`.
Binarna oblika je, ne glede na tekstovni zapis, enaka.

Uvozi

Uvozi so elementi, ki jih uvozimo iz drugega modula ali zunanjega okolja, ki komunicira z WebAssemblyjevim modulom, v našem primeru je to JavaScript vmesnik. Uvozi so lahko funkcije, pomnilnik, tabela in globalne spremenljivke, vsakega od teh pa definiramo z dvema imenoma in tipom. Kombinacija obeh imen se lahko pojavi tudi večkrat, v kolikor je tip uvoza različen.

Binarna oblika uvoze zapiše v sekciji uvozov `importsec`, ki nosi identifikator 2. Imena *mod* in *nm*, ki definirajo posamezen uvoz `import`, so zapisana z UTF-8 kodiranjem Unicode znakov zaporedja. Tip uvoza pa označimo z `0x00`, če je to funkcija in ji pripišemo indeks tipa x, `0x01` za tabelo s tipom tabele *tt*, `0x02` za pomnilnik, ki mu pripišemo pomnilniški tip *mt*, ter `0x03` za

globalno spremeljivko, ki ji pripada globalni tip gt. Sekcijo uvozov zapišemo tako:

```
importsec  ::= im*:section2(vec(import))          ⇒ im*
import     ::= mod:name nm:name d:importdesc
                 ⇒ {module mod, name nm, desc d}
importdesc ::= 0x00 x:typeidx                     ⇒ func x
             | 0x01 tt:tabletype                  ⇒ table tt
             | 0x02 mt:memtype                    ⇒ mem mt
             | 0x03 gt:globaltype                 ⇒ global gt
```

Tekstovna oblika segment uvozov označuje z rezervirano besedo `import`, ki mu sledi kombinacija dveh imen mod ter nm in tip uvoza `importdesc`. Celotna definicija je zapisana v oklepajih. Tip uvoza prav tako zapišemo v oklepajih z rezervirano besedo, opcijskim simboličnim identifikatorjem `id` in ustreznim indeksom tipa. Uvozi so tako funkcije z rezervirano besedo `func` in tipom x, tabele z rezervirano besedo `table` in tipom tabel tt, pomnilnik z rezervirano besedo `memory` in pomnilniškim tipom mt ter globalne spremenljivke z rezervirano besedo `global` ter globalnim tipom gt. Segment uvozov zapišemo tako:

```
import     ::= '(' 'import' mod:name nm:name d:importdesc ')'
                 ⇒ {module mod, name nm, desc d}
importdesc ::= '(' 'func' id? x:typeuse ')'       ⇒ func x
             | '(' 'table' id? tt:tabletype ')'   ⇒ table tt
             | '(' 'memory' id? mt:memtype ')'    ⇒ mem mt
             | '(' 'global' id? gt:globaltype ')' ⇒ global gt
```

Primer 2.1 prikaže uvoz JavaScript funkcije.

Uvozi so lahko deklarirani tudi znotraj definicije funkcije oziroma ostalih elementov, in sicer takoj po rezerviranem imenu komponente in opcijskem identifikatorju.

```
'(' 'func' id? '(' 'import' name1 name2 ')' typeuse ')'
```

Funkcije

Ta komponenta definira vektor funkcij, ki so zapisane najprej z njihovim tipom, nato z vektorjem lokalnih spremenljivk, ki jim sledi telo funkcije.

V binarni obliki se ta komponenta razdeli v dve sekciji. Prva sekcija je sekcija funkcij `funcsec`, ki ima identifikator 3. Definira prvi del funkcij – to je njihov tip `typeidx`, zapišemo pa jo tako:

$$\texttt{funcsec} ::= x^*\texttt{:section}_3(\texttt{vec}(\texttt{typeidx})) \Rightarrow x^*$$

Druga sekcija je sekcija kode `codesec` z identifikatorjem 10, v kateri zapišemo lokalne spremenljivke in telo funkcije. Sekcija sestoji iz vektorja kod `code`, ki najprej definira dolžino drugega dela funkcije *size* z `u32`, sledi pa ji vektor lokalnih spremenljivk t^* ter sosledje ukazov e^*. Lokalne spremenljivke so definirane s številom n ter vrednostnim tipom t. Sekcija kode je torej zapisana:

$$\begin{array}{lll}
\texttt{codesec} & ::= \mathit{code}^*\texttt{:section}_{10}(\texttt{vec}(\texttt{code})) & \Rightarrow \mathit{code}^* \\
\texttt{code} & ::= \mathit{size}\texttt{:u32}\ \mathit{code}\texttt{:func} & \Rightarrow \mathit{code} \\
\texttt{func} & ::= (t^*)^*\texttt{:vec(locals)}\ e^*\texttt{:expr} & \Rightarrow \mathit{concat}((t^*)^*),\ e^* \\
\texttt{locals} & ::= n\texttt{:u32}\ t\texttt{:valtype} & \Rightarrow t^n
\end{array}$$

V modulu, kjer je definiranih več funkcij, se vrstni red funkcij, zapisanih najprej v sekciji uvozov in nato v sekciji funkcij, ujema z vrstnim redom njihovih teles, zapisanih v sekciji kode. Primer zapisa sekcij najdemo v primeru kode 2.2.

V tekstovni obliki je definicija funkcije `func`, njen tip x, lokalne spremenljivke t^* in ukazi in^*, zapisana z enim segmentom, katerega rezervirana beseda je `func`, lokalne spremenljivke pa imajo rezervirano besedo `local`. Tako funkciji kot lokalnim spremenljivkam lahko pripišemo identifikator `id`, ki ga zapišemo takoj za rezervirano besedo. Primer zapisa funkcije najdemo v primeru 2.1, zapišemo pa ga tako:

$$\begin{array}{lll}
\texttt{func} & ::= \texttt{'('}\ \texttt{'func'}\ \texttt{id}^?\ x\texttt{:typeuse}\ (t\texttt{:local})^*\ (\mathit{in}\texttt{:instr})^*\ \texttt{')'} & \\
 & \Rightarrow \{\texttt{type}\ x,\ \texttt{locals}\ t^*,\ \texttt{body}\ \mathit{in}^*\ \texttt{end}\} & \\
\texttt{local} & ::= \texttt{'('}\ \texttt{'local'}\ \texttt{id}^?\ t\texttt{:valtype}\ \texttt{')'} & \Rightarrow t
\end{array}$$

Tabele

Tabela je vektor referenc na elemente. Definirana je z limito minimalne in opcijsko maksimalne vrednosti števila njenih elementov. Trenutno je edini

možni element v tabeli `anyfunc`, kar je referenca na funkcijo kateregakoli tipa. Prav tako imamo lahko v modulu zgolj eno tabelo. Inicializiramo jo lahko z elementnimi segmenti.

Tabele nimajo enake vloge, kot smo je navajeni iz ostalih programskih jezikov. Njihovo uporabnost lahko najbolje primerjamo s C/C++ kazalci na funkcije. Tovrstni podatki so zelo ranljivi, če bi jih zapisali v pomnilnik, zato jih v WebAssemblyju zapišemo v tabelo.

Sekcija tabel `tablesec` ima identifikator 4, njena vsebina pa je vektor tabel `table`, ki je trenutno omejen zgolj na eno tabelo, ta pa je definirana s tipom tabele *tt*. Sekcijo tabel zapišemo tako:

$$\begin{array}{lll} \texttt{tablesec} ::= tab^*\texttt{:section}_4(\texttt{vec}(\texttt{table})) & \Rightarrow & tab^* \\ \texttt{table} ::= tt\texttt{:tabletype} & \Rightarrow & \{\texttt{type}\ tt\} \end{array}$$

V tekstovni obliki ima tabela rezervirano besedo `table`, ki ji sledi opcijski identifikator `id` ter tip tabele *tt*. Primer 2.4 prikaže njeno definiranje in uporabo, definicijo pa zapišemo v oklepajih tako:

$$\texttt{table} ::= \texttt{'('}\ \texttt{'table'}\ \texttt{id}^?\ tt\texttt{:tabletype}\ \texttt{')'} \Rightarrow \{\texttt{type}\ tt\}$$

```
(module
  (table 2 anyfunc)
  (elem (i32.const 0) $f1 $f2)
  (func $f1 (result i32)
    i32.const 42)
  (func $f2 (result i32)
    i32.const 13)
  (type $return_i32 (func (result i32)))
  (func (export "callByIndex") (param $i i32) (result i32)
    get_local $i
    call_indirect (type $return_i32))
)
```

Primer 2.4: Definicija tabele z dvema poljema, njena inicializacija s funkcijama `f1` in `f2`.

Pomnilniki

Pomnilnik je vektor bajtov. Definiran je z limito minimalne in opcijsko maksimalne vrednosti, do katere lahko raste. V trenutni verziji WebAssemblyja

je modul lahko vezan le na en pomnilnik, medtem ko je en pomnilnik lahko vezan na več modulov. Inicializiramo ga lahko s podatkovnimi segmenti.

Sekcija pomnilnika `memsec` ima identifikator 5, njena vsebina pa je vektor pomnilnikov `mem`, ki je omejen na eno vrednost. Pomnilnik `mem` je predstavljen s tipom pomnilnika mt. Sekcijo pomnilnika zapišemo tako:

$$\begin{array}{lll} \texttt{memsec} & ::= mem^*\texttt{:section}_5(\texttt{vec}(\texttt{mem})) & \Rightarrow\ mem^* \\ \texttt{mem} & ::= mt\texttt{:memtype} & \Rightarrow\ \{\texttt{type}\ mt\} \end{array}$$

Primer 2.2 ima definiran pomnilnik z minimalno vrednostjo 1 (enota WebAssemblyjeve strani), kjer maksimalna vrednost ni definirana.

V tekstovni obliki je rezervirana beseda za pomnilnik `memory`, za katero pripišemo opcijski identifikator `id` ter pomnilniški tip mt. Primer 2.1 prikaže definicijo pomnilnika, ki ga tudi izvozimo, definicijo pa zapišemo v oklepajih tako:

$$\texttt{mem} ::= \texttt{'('}\ \texttt{'memory'}\ \texttt{id}^?\ mt\texttt{:memtype}\ \texttt{')'} \Rightarrow \{\texttt{type}\ mt\}$$

Globalne spremenljivke

Ta segment definira vektor globalnih spremenljivk. Vsaka globalna spremenljivka nosi eno vrednost določenega tipa, ki je bodisi spremenljiva ali konstantna. Poleg tega jo znotraj definicije tudi inicializiramo z ukazom za konstante.

Sekcija globalnih spremenljivk `globalsec` ima identifikator 6. Globalne spremenljivke so definirane z globalnim tipom gt, ki mu sledi inicializacija z izrazom e:

$$\begin{array}{lll} \texttt{globalsec} & ::= glob^*\texttt{:section}_6(\texttt{vec}(\texttt{global})) & \Rightarrow\ glob^* \\ \texttt{global} & ::= gt\texttt{:globaltype}\ e\texttt{:expr} & \Rightarrow\ \{\texttt{type}\ gt,\ \texttt{init}\ e\} \end{array}$$

V tekstovni obliki se definicija globalnih spremenljivk začne z rezervirano besedo `global`, sledi opcijski identifikator `id`, globalni tip gt ter izraz e, ki spremenljivko inicializira, zapišemo pa jo tako:

$$\begin{array}{l} \texttt{global} ::= \texttt{'('}\ \texttt{'global'}\ \texttt{id}^?\ gt\texttt{:globaltype}\ e\texttt{:expr}\ \texttt{')'} \\ \qquad\qquad \Rightarrow \{\texttt{type}\ gt,\ \texttt{init}\ e\} \end{array}$$

Primer 2.5 ilustrira zapis globalne spremenljivke v tekstovni obliki.

```
(module
  (global (mut i32) (i32.const 0))
)
```

Primer 2.5: Definicija globalne spremenljivke s tipom `i32` inicializirana na 0, katere vrednost lahko kasneje spremenimo.

Izvozi

Izvozi modula so prav tako kot pri uvozih lahko definicije pomnilnika, tabele, funkcij in globalnih spremenljivk. V gostujočem okolju postanejo dosegljivi po opredelitvi modula. Vsak od izvozov pa mora imeti edinstveno ime, da jih lahko v okolju nedvoumno ločimo.

Sekcija izvozov `exportsec` ima identifikator 7. Vsebuje vektor izvozov `export`, ki so definirani z imenom nm ter vrsto izvoza `exportdesc`. Vrsta izvoza je definirana z bajtom `0x00` in funkcijskim indeksom x za funkcije, z bajtom `0x01` in indeksom tabele x za tabele, z bajtom `0x02` in indeksom pomnilnika x za pomnilnik ter za globalne spremenljivke z bajtom `0x03` in globalnim indeksom x. Sekcijo zapišemo tako:

$$
\begin{array}{lll}
\texttt{exportsec} & ::= ex^*{:}\texttt{section}_7(\texttt{vec}(\texttt{export})) & \Rightarrow\ ex^* \\
\texttt{export} & ::= nm{:}\texttt{name}\ d{:}\texttt{exportdesc} & \Rightarrow\ \{\texttt{name}\ nm,\ \texttt{desc}\ d\} \\
\texttt{exportdesc} & ::= \texttt{0x00}\ x{:}\texttt{funcidx} & \Rightarrow\ \texttt{func}\ x \\
 & \mid\ \texttt{0x01}\ x{:}\texttt{tableidx} & \Rightarrow\ \texttt{table}\ x \\
 & \mid\ \texttt{0x02}\ x{:}\texttt{memidx} & \Rightarrow\ \texttt{mem}\ x \\
 & \mid\ \texttt{0x03}\ x{:}\texttt{globalidx} & \Rightarrow\ \texttt{global}\ x
\end{array}
$$

Primer 2.2 ima v sekciji izvozov dva vpisa. Prvi je pomnilnik z indeksom 0 in izvoznim imenom „memory“, drugi je funkcija z indeksom 1 in izvoznim imenom „hello“.

V tekstovni obliki imajo izvozi rezervirano besedo `export`. Definicijo zapišemo v oklepajih, po rezervirani besedi pa zapišemo ime izvoza nm ter vrsto izvoza `exportdesc`. Funkcije imajo rezervirano besedo `func`, ki ji sledi funkcijski indeks x, tabele imajo rezervirano besedo `table`, ki ji sledi indeks

tabele x, pomnilniki imajo rezervirano besedo `memory`, ki ji sledi indeks pomnilnika x ter globalne spremenljivke z rezervirano besedo `global`, ki ji sledi globalni indeks x. Definicijo izvozov napišemo torej tako:

```
export     ::= '(' 'export' nm:name d:exportdesc ')'
               ⇒ {name nm, desc d}
exportdesc ::= '(' 'func' x:funcidx ')'          ⇒ func x
             | '(' 'table' x:tableidx ')'        ⇒ table x
             | '(' 'memory' x:memidx ')'         ⇒ mem x
             | '(' 'global' x:globalidx ')'      ⇒ global x
```

Prav tako jih lahko deklariramo znotraj definicij, ki jih bomo izvozili. Enako kot pri uvozih jih zapišemo takoj po rezervirani besedi komponente in opcijskem identifikatorju, tako kot v primeru 2.1.

```
'(' 'func' id? '(' 'export' name ')' ... ')'
```

Funkcija start

Začetna funkcija je deklarirana z indeksom funkcije, ki se začne izvajati takoj po opredelitvi modula. Začetna funkcija je lahko le tip funkcije, ki ne zahteva parametrov in tudi ne vrne rezultata. V primeru 2.1 in 2.2 bi kot začetno funkcijo lahko definirali funkcijo `$hello`.

Sekcija začetne funkcije `startsec` ima številko 8, njena vsebina pa je zgolj funkcijski indeks x funkcije, ki jo določimo za začetno funkcijo:

```
startsec ::= st?:section8(start) ⇒ st?
start    ::= x:funcidx           ⇒ {func x}
```

V primeru 2.2 bi začetna funkcija lahko definirala funkcijo z indeksno številko 1. Tako bi sekcija začetne funkcije imela bajte `08 01`.

V tekstovni obliki ima začetna funkcija rezervirano besedo `start`. Definicijo zapišemo v oklepajih z rezervirano besedo ter funkcijskim indeksom x tako:

```
start ::= '(' 'start' x:funcidx ')' ⇒ {func x}
```

V primeru 2.1 bi tako lahko zapisali `(start $hello)`.

Segment elementov

S segmentom elementov lahko inicializiramo tabelo, ki je na začetku prazna. Komponenta definira vektor fukcijskih indeksov na določenem odmiku v tabeli.

Elementi so zapisani v sekciji elementov `elemsec`, ki nosi identifikator 9. Elementi v tabeli so definirani z indeksom tabele x, sledi mu izraz e, ki pove, kam v tabelo naj zapiše elemente, ter vektor funkcijskih indeksov y^*:

$$\begin{aligned}
&\texttt{elemsec} ::= seg^*{:}\texttt{section}_9(\texttt{vec}(\texttt{elem})) && \Rightarrow\ seg\\
&\texttt{elem} ::= x{:}\texttt{tableidx}\ e{:}\texttt{expr}\ y^*{:}\texttt{vec}(\texttt{funcidx})\\
&\qquad \Rightarrow\ \{\texttt{table}\ x,\ \texttt{offset}\ e,\ \texttt{init}\ y^*\}
\end{aligned}$$

V primeru 2.6 najdemo zapis sekcije elementov primera 2.4, ki ima en element, ki ga zapišemo na mesto 0, element pa nosi dve referenci na funkcije, in sicer na funkcijo z indeksom 0 in z indeksom 1.

V tekstovni obliki ima segment elementov rezervirano besedo `elem`. Indeks tabele x lahko izpustimo, saj je njegova privzeta vrednost enaka 0, ki je indeks edine tabele, ki je lahko definirana v enem modulu. Prav tako lahko na mestu odmika opustimo oklepaje in rezervirano besedo `offset` ter zapišemo zgolj izraz e za inicializacijo. Po odmiku e zapišemo še vektor funkcijskih indeksov. Zapis z opustitvijo teh delov najdemo v primeru 2.4, celotno definicijo pa zapišemo tako:

$$\begin{aligned}
&\texttt{elem} ::= \texttt{'('}\ \texttt{'elem'}\ x{:}\texttt{tableidx}\ \texttt{'('}\ \texttt{'offset'}\ e{:}\texttt{expr}\ \texttt{')'}\\
&\qquad y^*{:}\texttt{vec(funcidx)}\ \texttt{')'}\ \Rightarrow\ \{\texttt{table}\ x,\ \texttt{offset}\ e,\ \texttt{init}\ y^*\}
\end{aligned}$$

```
09 08 01 00 41 00 0B 02 00 01
```

Primer 2.6: Sekcija elementov primera 2.4

Ta segment lahko zapišemo tudi znotraj definicije tabele, s tem da se vektor indeksov funkcij zapiše na odmik 0, velikost tabele pa je enaka velikosti vektorja.

$$\texttt{'('}\ \texttt{'table'}\ \texttt{id}^?\ \texttt{elemtype}\ \texttt{'('}\ \texttt{'elem'}\ x^n{:}\texttt{vec(funcidx)}\ \texttt{')'}\ \texttt{')'}$$

Segment podatkov

Tako kot s segmentom elementov inicializiramo tabelo, lahko podobno inicializiramo tudi pomnilnik s podatkovnim segmentom. Ta definira vektor zaporedja bajtov, ki ga zapiše na podani odmik v pomnilnik.

Sekcija podatkov `datasec` ima identifikator 11. Njena vsebina je vektor podatkov `data`, ki jo zapišemo najprej z indeksom pomnilnika x, nato z odmikom e ter z vektorjem bajtov b^*:

$$\begin{aligned}
&\texttt{datasec} ::= seg^*{:}\texttt{section}_{11}(\texttt{vec}(\texttt{data})) && \Rightarrow\ seg\\
&\texttt{data} ::= x{:}\texttt{memidx}\ e{:}\texttt{expr}\ b^*{:}\texttt{vec}(\texttt{byte})\\
&\qquad\qquad \Rightarrow\ \{\texttt{data}\ x,\ \texttt{offset}\ e,\ \texttt{init}\ b^*\}
\end{aligned}$$

Primer 2.2 ima sekcijo podatkov, ki v pomnilnik vpiše UTF-8 kodiranje zaporedja znakov `Hello, world!`

V tekstovni obliki ima segment podatkov rezervirano besedo `data`, sledi ji indeks pomnilnika x, izraz za inicializacijo odmika e, naznanjen z rezervirano besedo `offset`, ki ju skupaj zapišemo v oklepajih, sledijo še podatki y^*. Segment podatkov zapišemo torej tako:

$$\begin{aligned}
&\texttt{data} ::= \texttt{'('}\ \texttt{'data'}\ x{:}\texttt{memidx}\ \texttt{'('}\ \texttt{'offset'}\ e{:}\texttt{expr}\ \texttt{')'}\\
&\qquad y^*{:}\texttt{datastring}\ \texttt{')'} && \Rightarrow\ \{\texttt{table}\ x,\ \texttt{offset}\ e,\ \texttt{init}\ y^*\}\\
&\texttt{datastring} ::= (b^*{:}\texttt{string})^* && \Rightarrow\ concat((b^*)^*)
\end{aligned}$$

V tekstovni obliki lahko enako kot pri tabelah izpustimo indeks pomnilnika, saj je njegova privzeta vrednost enaka 0. Enako lahko na mestu odmika opustimo rezervirano besedo `offset` in zapišemo zgolj izraz za inicializacijo. Primer 2.1 prikazuje zapis z opuščenimi prej omenjenimi elementi.

Prav tako lahko segment podatkov zapišemo znotraj definicije pomnilnika. Podatki so tako zapisani na odmik 0, velikost tabele pa se prilagodi velikosti podatkov n zaokroženo navzgor na velikost strani.

$$\texttt{'('}\ \texttt{'memory'}\ \texttt{id}^?\ \texttt{'('}\ \texttt{'data'}\ b^n{:}\texttt{datastring}\ \texttt{')'}\ \texttt{')'}$$

2.1.2 Ukazi

WebAssembly je osnovan na skladovnem stroju, kjer sosledja ukazov operirajo z vrednostmi, tako da s sklada snamejo argumente in nanj vrnejo rezultat. Nekateri ukazi poleg vrednosti s sklada operirajo tudi z vrednostmi, ki jih podamo skupaj z ukazom. Delimo jih na navadne in strukturirane ukaze.

V binarni obliki ukaze predstavljajo bitne kode, tekstovna oblika pa jih predstavi z besedno obliko, kjer najdemo vse že znane izraze iz drugih zbirnikov. V tekstovni obliki jih lahko zapišemo tudi kot simbolične izraze. Sprememba zapisa je zgolj vizualni pripomoček.

Numerični ukazi

To so osnovni ukazi, ki operirajo z numeričnimi vrednostmi določenega tipa. Potrebne operande vzamejo s sklada ter rezutat vrnejo nazaj na sklad. Za vsak tip vrednosti obstaja svoja različica istega ukaza, le-te pa ločimo v naslednje kategorije:

- Konstante: Dano konstanto vrnejo na sklad.
- Enočleni operatorji: Vzamejo en operand, vrnejo en rezultat.
- Dvočleni operatorji: Vzamejo dva operanda, vrnejo en rezultat.
- Test: Vzame en operand, vrne Boolov številski rezultat.
- Primerjava: Vzame dva operanda, vrne Boolov številski rezultat.
- Pretvorba: Vzame en operand določenega vrednostnega tipa in ga pretvori v rezultat drugega vrednostnega tipa.

Nekateri ukazi imajo dve različici zaradi predznačenosti števil. V kolikor celoštevilski ukaz podatka o predznačenosti nima, pomeni, da predznačenost števila ne vpliva na rezultat.

V binarni obliki so posamezni ukazi zapisani s svojimi bitnimi kodami. Večina ukazov je zapisana zgolj z bitno kodo, saj operande vzamejo s sklada.

Izjema je skupina ukazov za konstante, kjer konstanto zapišemo skupaj z ukazom. Konstanto za vrednostni tip `i32` zapišemo z bitno kodo `0x41`, ki ji sledi vrednost n. Podoben zapis imajo ostali trije ukazi za konstante:

```
instr ::= 0x41 n:i32 ⇒ i32.const n
        | 0x42 n:i64 ⇒ i64.const n
        | 0x43 z:f32 ⇒ f32.const z
        | 0x44 z:f64 ⇒ f64.const z
```

V tekstovni obliki so bitne kode zamenjane z besedno obliko, ki se začne z zapisom vrednostnega tipa, temu sledi pika in nato ime ukaza. Ukazi, pri katerih je pomembna predznačenost števila, se nadaljujejo s podčrtajem in ustreznim `u` ali `s`. Ukazi, ki izvedejo pretvorbo, pa za poševnico navedejo še rezultatski tip. Takšni ukazi so zapisani tako:

```
instr ::= 'i32.const' n:i32    ⇒ i32.const n
        | 'i64.add'            ⇒ i64.add
        | 'i32.div_s'          ⇒ i32.div_s
        | 'f32.convert_s/i64'  ⇒ f32.convert_s/i64
          ...
```

Primer 2.1 prikaže uporabo ukaza za konstante tipa `i32` pri podajanju odmika v definiciji segmenta podatkov ter v kodi funkcije `$hello`, kjer s tem ukazom na sklad naložimo argumente za funkcijo `$print`, ki jo pokličemo.

Primer 2.3 prikaže uporabo ukaza za seštevanje `i32.add`, ki predhodno potrebuje na skladu dve vrednosti tipa `i32` za seštevanje, rezultat pa vrne nazaj na sklad.

Parametrični ukazi

Ti ukazi lahko operirajo z operandi kateregakoli vrednostnega tipa. Ukaz `drop` odvrže en operand in ima bitno kodo `0x1A`, ukaz `select` pa izbere enega od prvih dveh operandov, glede na to, ali je tretji nič ali različen od nič ter ima bitno kodo `0x1B`, kot vidimo spodaj:

```
instr ::= 0x1A ≡ 'drop'   ⇒ drop
        | 0x1B ≡ 'select' ⇒ select
```

Ukazi spremenljivk

Ti ukazi nam omogočajo dostop do lokalnih in globalnih spremenljivk. Ukazi `get` vrednost spremenljivke naložijo na sklad, ukazi `set` pa vrednost na skladu pripišejo določeni spremenljivki. Poleg teh dveh ukazov imamo pri lokalnih spremenljivkah tudi ukaz `tee`, ki je enak ukazu `set`, le da tudi vrne argument na sklad. Ti ukazi imajo v binarni obliki bitne kode od `0x20` do vključno `0x24`, ki jim sledi indeks spremenljivke x. V tekstovni obliki je zapis enak, le da bitne kode zamenjamo z imeni ukazov, kot vidimo tu:

`instr` ::= `0x20` x:`localidx` ≡ `'get_local'` x:`localidx` ⇒ `get_local` x
`instr` ::= `0x21` x:`localidx` ≡ `'set_local'` x:`localidx` ⇒ `set_local` x
`instr` ::= `0x22` x:`localidx` ≡ `'tee_local'` x:`localidx` ⇒ `tee_local` x
`instr` ::= `0x23` x:`localidx` ≡ `'get_global'` x:`localidx` ⇒ `get_global` x
`instr` ::= `0x24` x:`localidx` ≡ `'set_global'` x:`localidx` ⇒ `set_global` x

Primer 2.3 v definirani funkciji na sklad naloži vrednosti dveh lokalnih spremenljivk, ki sta v funkciji podani kot parametra.

Pomnilniški ukazi

Za dostop do pomnilnika potrebujemo ukaze za shranjevanje `store` in nalaganje `load`. Ti se razlikujejo glede na vrednostni tip, s katerim operirajo, ter tudi glede na število bitov, ki jih shranijo ali naložijo. Pri nekaterih ukazih za nalaganje `load` pa je potrebno specificirati tudi predznačenost `u` ali `s`. Z ukazom moramo v binarni obliki zapisati tudi pomnilniški argument `memarg`, ki vsebuje dve vrednosti, odmik o in poravnavo a. Pred pomnilniški

argument m zapišemo bitno kodo za posamezen ukaz tako:

```
memarg ::= a:u32 o:u32      ⇒ {align a, offset o}
instr  ::= 0x28 m:memarg    ⇒ i32.load m
         | 0x29 m:memarg    ⇒ i64.load m
         | 0x2A m:memarg    ⇒ f32.load m
         | 0x2B m:memarg    ⇒ f64.load m
         | 0x2D m:memarg    ⇒ i32.load8_u m
         | 0x2E m:memarg    ⇒ i32.load16_s m
         | 0x36 m:memarg    ⇒ i32.store m
         | 0x3E m:memarg    ⇒ i64.store32 m
         | ...
```

V tekstovni obliki pomnilniškega argumenta ni potrebno določiti. Primer 2.7 prikaže uporabo `load` in `store` ukazov, ki operirajo z bajti, ki so nepredznačena števila. `Load` ukazi potrebujejo na skladu odmik, ki jim pove, od kod naj naložijo neko vrednost na sklad. `Store` ukazi pa potrebujejo na skladu dve vrednosti, prva je odmik, druga pa vrednost, ki jo shranijo na podani odmik. Pomnilniške ukaze torej najprej zapišemo z besedno obliko, ki ji sledi pomnilniški argument m. Pomnilniški argument ima za odmik o rezervirano besedo z enačajem `offset=` ter za poravnavo a rezervirano besedo z enačajem `align=`, kot je zapisano tu:

```
memarg_N ::= o:offset a:align_N          ⇒ {align n, offset o}
             (if a = 2^n)
offset   ::= 'offset='o:u32              ⇒ o
           | ϵ                           ⇒ 0
align_N  ::= 'align='a:u32               ⇒ a
           | ϵ                           ⇒ N
instr    ::= 'i32.load' m:memarg_4       ⇒ i32.load m
           | 'i64.load16_s' m:memarg_2   ⇒ i64.load16_s m
           | 'f64.store' m:memarg_8      ⇒ f64.store m
           | ...
```

Odmik je dodan dinamičnemu naslovu tako, da dobimo 33-bitni naslov, s katerim dostopamo do pomnilnika. Vse vrednosti so pisane po pravilu tankega konca. V kolikor pa skušamo dostopati do pomnilnika, katerega naslov leži zunaj določene velikosti, se sproži past.

```
(module
 (memory (import "memory" "mem") 1)
 (func (export "average") (param $offset i32)(param $len i32)
  (param $stOffset i32)
  (local $sum i32) (local $n i32)
  (set_local $sum (i32.const 0))(set_local $n (get_local $len))
  (if (i32.gt_u (get_local $n)(i32.const 0))
   (then (loop $loop
    (i32.load8_u (get_local $offset))
    get_local $sum
    i32.add
    set_local $sum
    (set_local $offset
    (i32.add (get_local $offset)(i32.const 1)))
    (set_local $n (i32.sub (get_local $n)(i32.const 1)))
    (if (i32.gt_u (get_local $n)(i32.const 0))
     (then br $loop)
    ))
    get_local $stOffset
    (i32.div_u (get_local $sum)(get_local $len))
    i32.store8
   )
   (else
    (i32.store8 (get_local $stOffset)(get_local $sum))
))))
```

Primer 2.7: Program, ki izračuna povprečje bajtov v pomnilniku na danem odmiku v dolžini `len`.

V to skupino sodita tudi ukaza `memory.size` z bitno kodo `0x3f` in `memory.grow` z bitno kodo `0x40`. Prvi vrne trenutno velikost pomnilnika, drugi pa poveča pomnilnik za dano vrednost in vrne prejšnjo velikost ali -1, če ne more biti dodeljenega dovolj pomnilnika. V binarni obliki je za bitno kodo ukaza zapisan še bajt `0x00`, ki označuje pomnilnik z indeksom 0. Zapis je torej tak:

```
instr ::= 0x3f 0x00 ≡ 'memory.size' ⇒ memory.size
        | 0x40 0x00 ≡ 'memory.grow' ⇒ memory.grow
```

Krmilni ukazi

Ti ukazi nadzirajo potek programa. Ukaz `unreachable` sproži past, ukaz `nop` pa ne naredi ničesar.

Ukazi `block`, `loop` in `if` so strukturirani ukazi, ki imajo ugnezdena sosledja drugih ukazov *in* ločenih oziroma končanih z rezerviranimi besedami

`else` oziroma `end`, prav tako pa lahko zapišejo rezultat v podani `resulttype`.

Ukazi `br`, `br_if` in `br_table` so ukazi, ki v strukturiranih ukazih spremenijo potek ukazov, lahko brezpogojno ali pogojno. Če se s podanim indeksom l nanašajo na bločne in pogojne ukaze, se obnašajo kot znani ukaz `break`, medtem ko se z indeksom l nanašajo na zančne ukaze, se obnašajo kot znani ukaz `continue`. Podobno funkcijo ima ukaz `return`, ki predstavlja brezpogojni skok do najbolj zunanjega bloka, kar je telo trenutne funkcije.

Ukaza `call` in `call_indirect` pokličeta drugo funkcijo. Prvi argumente za klicano funkcijo x vzame s sklada in nanj tudi zapiše njen rezultat, drugi pa kliče funkcijo posredno preko indeksa v tabeli x.

V binarni obliki so za krmilne ukaze rezervirane operacijske kode med `0x00` in `0x11`. Vsak od strukturiranih ukazov se konča z rezervirano besedo `end`, ki ji pripada koda `0x0B`. Vsi opisani krmilni ukazi se zapišejo tako:

$$
\begin{array}{lll}
\texttt{instr} ::= & \texttt{0x00} & \Rightarrow \texttt{unreachable} \\
| & \texttt{0x01} & \Rightarrow \texttt{nop} \\
| & \texttt{0x02}\ rt{:}\texttt{blocktype}\ (in{:}\texttt{instr}^*)\ \texttt{0x0B} & \Rightarrow \texttt{block}\ rt\ in^*\ \texttt{end} \\
| & \texttt{0x03}\ rt{:}\texttt{blocktype}\ (in{:}\texttt{instr})^*\ \texttt{0x0B} & \Rightarrow \texttt{loop}\ rt\ in^*\ \texttt{end} \\
| & \texttt{0x04}\ rt{:}\texttt{blocktype}\ (in{:}\texttt{instr})^*\ \texttt{0x0B} & \Rightarrow \texttt{if}\ rt\ in^*\ \texttt{end} \\
| & \texttt{0x04}\ rt{:}\texttt{blocktype}\ (in_1{:}\texttt{instr})^*\ \texttt{0x05}\ (in_2{:}\texttt{instrn}^*)\ \texttt{0x0B} & \\
& \quad \Rightarrow \texttt{if}\ rt\ in_1^*\ \texttt{else}\ in_2 * \ \texttt{end} & \\
| & \texttt{0x0C}\ l{:}\texttt{labelidx} & \Rightarrow \texttt{br}\ l \\
| & \texttt{0x0D}\ l{:}\texttt{labelidx} & \Rightarrow \texttt{br\_if}\ l \\
| & \texttt{0x0E}\ l^*{:}\texttt{vec(labelidx)}\ l_N{:}\texttt{labelidx} & \Rightarrow \texttt{br\_table}\ l^*\ l_N \\
| & \texttt{0x0F} & \Rightarrow \texttt{return} \\
| & \texttt{0x10}\ x{:}\texttt{funcidx} & \Rightarrow \texttt{call}\ x \\
| & \texttt{0x11}\ x{:}\texttt{typeidx}\ \texttt{0x00} & \Rightarrow \texttt{call\_indirect}\ x
\end{array}
$$

V tekstovni obliki imajo krmilni ukazi rezervirane besede, ki smo jih že opisali. Vsakemu strukturiranemu ukazu lahko pripišemo simbolični identifikator l, da se lažje sklicujemo na pravi strukturirani ukaz pri ukazih `br`, sledi rezultat rt, sosledje ukazov in ter rezervirana beseda `end`. Pri ukazu `if` se lahko pojavi tudi rezervirana beseda `else`. Ukazi `br` za rezervirano besedo potrebujejo indeks l strukturiranega ukaza, na katerega se ukaz sklicuje. Ukazom `call` pa po rezervirani besedi pripišemo še funkcijski indeks oziroma uporabo tipov x. Zapišemo jih tako:

```
instr ::= 'unreachable'                          ⇒ unreachable
        | 'nop'                                  ⇒ nop
        | 'block' l:id? rt:resulttype (i n:instr)* 'end'
          ⇒ block rt in* end
        | 'loop' l:id? rt:resulttype (in :instr)* 'end'
          ⇒ loop rt in* end
        | 'if' l:id? rt:resulttype (in_1: instr)* 'else'
          (in_2:instr)* 'end' ⇒ if rt in_1* else in_2* end
        | 'br' l:labelidx                        ⇒ br l
        | 'br_if' l : labelidx                   ⇒ br_if l
        | 'br_table' l* : vec(labelidx) l_N : labelidx
          ⇒ br_table l* l_N
        | 'return'                               ⇒ return
        | 'call' x : funcidx                     ⇒ call x
        | 'call_indirect' x : typeuse            ⇒ call_indirect x
```

Primer 2.7 vsebuje tri krmilne ukaze, in sicer `if`, `loop` ter `br`. V danem programu `if` stavek preverja, ali je vrednost `len` večja od 0, ki je pogoj za nadaljevanje zanke. Zanka gre čez vrednosti v pomnilniku od podanega odmika `offset` v dolžini `len`. `Br` ukaz pa poskrbi, da se zanka lahko nadaljuje, tako da se sklicuje na zanko ob izpolnjenem pogoju. V kolikor pogoj ni izpolnjen, se zanka zaključi. Zanki smo pripisali simbolični identifikator `$loop`, ki ga pri ukazu `br` tudi uporabimo. V binarni obliki bi na tem mestu zapisali 1, saj ukaz `br` najprej objema ukaz `if`, ki ima indeks 0, nato pa ukaz `loop`, ki ima indeks 1. Kombinacijo `if` stavka in `br` ukaza bi lahko zamenjali z ukazom `br_if`. Primer 2.7 je sicer zapisan z zloženo obliko ukazov `if` in `loop`, njuno obliko brez zloženih ukazov pa najdemo v primeru 2.8.

Izrazi

Izrazi `expr` so sosledja ukazov *in* končana z oznako `end`. Kot izraz so predstavljena funkcijska telesa, inicializacije globalnih spremenljivk in odmiki pri definiranju elementnih ali podatkovnih segmentov. V binarni obliki najprej zapišemo sosledje ukazov *in*, ki jih zaključimo z bitno kodo `0x0B` tako:

$$\texttt{expr} ::= (in{:}\texttt{instr})^* \texttt{ 0x0B} \Rightarrow in^* \texttt{ end}$$

V tekstovni obliki po sosledju ukazov oznake `end` ne zapišemo:

$$\texttt{expr} ::= (in{:}\texttt{instr})^* \Rightarrow in^* \texttt{ end}$$

Zloženi ukazi

Ukazi so lahko v tekstovni obliki zapisani kot S-izrazi. S-izraz oz. simbolični izraz je način predstavitve ugnezdenih podatkov. Ukaze zapišemo v S-izraz tako, da jih zapišemo v oklepajih, vsak pa lahko vsebuje tudi ugnezdene zložene ukaze, ki predstavljajo operand ukaza. Pri strukturiranih ukazih zložena oblika ovrže oznako `end`, pri `if` izrazih pa morata biti obe veji ukaza grupirani v svoj S-izraz, s tem da prva veja dobi rezervirano besedo `then`, ki je drugače nimamo:

$$\begin{aligned}
&\texttt{'(' instr foldedinstr}^* \texttt{ ')'} \equiv \texttt{foldedinstr}^* \texttt{ instr} \\
&\texttt{'(' 'block' label resulttype instr}^* \texttt{ ')'} \\
&\equiv \texttt{ 'block' label resulttype instr}^* \texttt{ 'end'} \\
&\texttt{'(' 'loop' label resulttype instr}* \texttt{ ')'} \\
&\equiv \texttt{ 'loop' label resulttype instr}^* \texttt{ 'end'} \\
&\texttt{'(' 'if' label resulttype foldedinstr}^* \texttt{ '(' 'then' instr}_1^* \texttt{ ')'} \\
&\texttt{'(' 'else' instr}_2^* \texttt{ ')'}^? \texttt{ ')'} \\
&\equiv \texttt{foldedinstr}^* \texttt{ 'if' label resulttype instr}_1^* \texttt{ 'else' } (\texttt{instr}_2^*)^? \texttt{ 'end'}
\end{aligned}$$

Pri pisanju programov moramo razumeti delovanje sklada. Zložena oblika nam pomaga zgolj vizualno, saj je sintaksa programa tako bolj podobna visokonivojskim programskim jezikom, ki so nam bližje. Večji del programa primera 2.7 je zapisan z zloženimi ukazi. V podanem primeru je začetek prvega ukaza `if` sestavljen iz zloženih ukazov. V kolikor ne bi uporabili zloženih ukazov, bi tam koda izgledala tako, kot kaže primer 2.8.

```
get_local $n
i32.const 0
i32.gt_u
if
  loop $loop
    get_local $offset
    i32.load8_u
    ...
  end
  get_local $stOffset
  get_local $sum
  get_local $len
  i32.div_u
  i32.store8
else
  get_local $stOffset
  get_local $sum
  i32.store8
end
```

Primer 2.8: Del programa iz primera 2.7, zapisan s tekstovno obliko, ki ni sestavljena iz zloženih ukazov.

2.1.3 Tipi

Vrednostni tipi

WebAssembly definira 4 različne vrednostne tipe. `i32` in `i64` predstavljata 32- in 64-bitna cela števila, medtem ko `f32` in `f64` predstavljata 32- in 64-bitna števila s plavajočo vejico. Spodaj vidimo vrednostne tipe, zapisane v binarni obliki z njihovimi bitnimi kodami ter v tekstovni obliki z njihovimi rezerviranimi besedami:

```
valtype ::= 0x7F ::= 'i32' ⇒ i32
          | 0x7E ::= 'i64' ⇒ i64
          | 0x7D ::= 'f32' ⇒ f32
          | 0x7C ::= 'f64' ⇒ f64
```

V primeru 2.7 operiramo z vrednostnim tipom `i32`. Tega tipa so tako parametri kot lokalne spremenljivke, posledično pa uporabljamo tudi ukaze, ki operirajo s tem tipom.

Rezultatski tipi

Rezultat zaporedja ukazov najdemo v rezultatskem tipu, ki je lahko prazen ali katerikoli izmed vrednostnih tipov. Trenutna verzija WebAssemblya nam omogoča imeti le eno vrednost v rezultatu. V binarni obliki so rezultati predstavljeni s tipom bloka `blocktype`. Operacijska koda `0x40` označuje odsotnost rezultata, če pa rezultat obstaja, zapišemo njegov vrednostni tip t tako:

$$\begin{array}{rcl} \texttt{blocktype} ::= \texttt{0x40} & \Rightarrow & [] \\ \mid \quad t{:}\texttt{valtype} & \Rightarrow & [t] \end{array}$$

V tekstovni obliki rezultat `resulttype` predstavlja vrednostni tip t, v odsotnosti rezultata pa se zapis rezultata opusti.

$$\texttt{resulttype} ::= t{:}\texttt{result} \Rightarrow [t]$$

V primeru 2.4 najdemo štiri uporabe rezultatskega tipa, tri v definiciji funkcije in enega v definiciji tipa.

Funkcijski tipi

Funkcijski tip `functype` deklarira funkcije, ki preslikajo vektor parametrov ${t^*}_1$ v vektor rezultatov ${t^*}_2$. Vektor rezultatov je omejen na eno vrednost, medtem ko je parametrov lahko več. V binarni obliki je za funkcijski tip rezervirana operacijska koda `0x60`:

$$\texttt{functype} ::= \texttt{0x60}\ t_1^*{:}\texttt{vec}(\texttt{valtype})\ t_2^*{:}\texttt{vec}(\texttt{valtype}) \Rightarrow [t_1^*] \to [t_2^*]$$

V tekstovni obliki je za funkcijski tip rezervirana beseda `func`, po kateri zapišemo vektor parametrov ${t^*}_1$, ki imajo rezervirano besedo `param`, ter vektor rezultatov ${t^*}_2$ z rezervirano besedo `result`. Parametrom in rezultatom moramo določiti vrednostni tip t, parametrom pa lahko določimo tudi simbolični identifikator `id` tako:

$$\begin{array}{lcll} \texttt{functype} & ::= & \texttt{'('}\ \texttt{'func'}\ t_1^*{:}\texttt{vec}(\texttt{param})\ t_2^*{:}\texttt{vec}(\texttt{result})\ \texttt{')'} & \Rightarrow\ [t_1^*] \to [t_2^*] \\ \texttt{param} & ::= & \texttt{'('}\ \texttt{'param'}\ \texttt{id}^?\ t{:}\texttt{valtype}\ \texttt{')'} & \Rightarrow\ t \\ \texttt{result} & ::= & \texttt{'('}\ \texttt{'result'}\ t{:}\texttt{valtype}\ \texttt{')'} & \Rightarrow\ t \end{array}$$

Primer 2.9 prikazuje uporabo funkcijskega tipa v definiciji tipa `$first`.

```
(module
  (type $first (func (param i32 i32)(param i64)(result i32)))
)

00 61 73 6D 01 00 00 00 01 08 01 60 03 7F 7F 7E 01 7F
```

Primer 2.9: Program, ki definira en tip `$first` s funkcijskim tipom, ki ima 3 parametre in en rezultat. Tekstovna in binarna oblika programa.

Limite

Limite deklarirajo začetno velikost in morebitno omejitev pomnilnika, povezanega s pomnilniškimi tipi in tipi tabel. Limita `limits` določa minimalno vrednost n, ki je začetna velikost pomnilnika ali tabele, ter opcijsko tudi maksimalno vrednost m, ki določa največjo velikost, do katere lahko pomnilnik ali tabelo povečujemo. Če maksimalna vrednost ni določena, lahko velikost pomnilnika ali tabele povečujemo do vrednosti, ki jo omogoča okolje izvajanja. V binarni obliki odsotnost maksimalne vrednosti označimo z bajtom `0x00`, ki mu pripišemo minimalno vrednost n, zapisano z `u32`. Prisotnost maksimalne vrednosti označimo z bajtom `0x01`, ki mu pripišemo minimalno n in nato še maksimalno vrednost m z `u32` tako:

$$\begin{array}{rcll} \texttt{limits} & ::= & \texttt{0x00}\ n{:}\texttt{u32} & \Rightarrow\ \{\texttt{min}\ n,\ \texttt{max}\ \epsilon\} \\ & | & \texttt{0x01}\ n{:}\texttt{u32}\ m{:}\texttt{u32} & \Rightarrow\ \{\texttt{min}\ n,\ \texttt{max}\ m\} \end{array}$$

V tekstovni obliki ob odsotnosti maksimalne vrednosti m zapišemo zgolj minimalno n, prisotnost pa označujeta dve vrednosti v `u32`, kot zapisano spodaj.

$$\begin{array}{rcll} \texttt{limits} & ::= & n{:}\texttt{u32} & \Rightarrow\ \{\texttt{min}\ n,\ \texttt{max}\ \epsilon\} \\ & | & n{:}\texttt{u32}\ m{:}\texttt{u32} & \Rightarrow\ \{\texttt{min}\ n,\ \texttt{max}\ m\} \end{array}$$

Pomnilniški tipi

Pomnilniški tipi `memtype` deklarirajo linearni pomnilnik in njegovo velikost zapišejo z limito *lim*, ki pa je zapisana v enotah WebAssemblyjeve velikosti

strani. Ena WebAssemblyjeva stran je 64 kilobajtov.

$$\texttt{memtype} ::= \textit{lim}\texttt{:limits} \Rightarrow \textit{lim}$$

Primer 2.10 prikazuje program, ki definira pomnilnik s pomnilniškim tipom oziroma limito `1 5`. Minimalna vrednost je 1 WebAssemblyjeva stran, povečuje pa se lahko do 5 WebAssemblyjevih strani. V binarni obliki v istem primeru so pomnilniški tip zadnji trije bajti. Prvi od teh treh bajtov označuje prisotnost maksimalne vrednosti, naslednja dva pa sta minimalna in maksimalna vrednost.

```
(module
  (memory 1 5)
)

00 61 73 6D 01 00 00 00 05 04 01 01 03
```

Primer 2.10: Tekstovna in binarna oblika programa, ki definira pomnilnik z limito `1 5`.

Tipi tabel

Tipi tabel `tabletype` deklarirajo tabele z elementi v velikosti, definirani z limito *lim*. Elementi so lahko zgolj tipa `anyfunc`, ki predstavlja unijo katerihkoli funkcijskih tipov. V binarnem zapisu je tip elementa `anyfunc` zapisan z bajtom `0x70`.

$$\begin{aligned} \texttt{tabletype} &::= \textit{et}\texttt{:elemtype}\ \textit{lim}\texttt{:limits} \Rightarrow \textit{lim}\ \textit{et} \\ \texttt{elemtype} &::= \texttt{0x70} \Rightarrow \texttt{anyfunc} \end{aligned}$$

V tekstovni obliki tip tabele zapišemo najprej z limito *lim*, nato pa z rezervirano besedo `anyfunc`, ki označuje trenutno edini veljaven tip elementa, tako:

$$\begin{aligned} \texttt{tabletype} &::= \textit{lim}\texttt{:limits}\ \textit{et}\texttt{:elemtype} \Rightarrow \textit{lim}\ \textit{et} \\ \texttt{elemtype} &::= \texttt{'anyfunc'} \Rightarrow \texttt{anyfunc} \end{aligned}$$

Primer 2.11 prikazuje uporabo tipa tabel v definiciji tabele. Tip tabele predstavljata limita `3` in `anyfunc`. V binarni obliki predstavljajo zadnji trije

bajti tip tabele. Prvi od teh označuje tip elementa `anyfunc`, drugi označuje odsotnost maksimalne vrednosti, tretji bajt pa je začetna velikost tabele.

```
(module
  (table 3 anyfunc)
)

00 61 73 6D 01 00 00 00 04 04 01 70 00 03
```

Primer 2.11: Tekstovna in binarna oblika programa, ki definira tabelo z limito 3.

Globalni tipi

Globalni tipi `globaltype` deklarirajo vrednost t in tip globalne spremenljivke m, ki je lahko spremenljiva ali konstantna. V binarnem zapisu, poleg vrednostnega tipa, spremenljivost označimo z bajtom `0x00`, če je vrednost konstantna in je kasneje ne moremo več spreminjati, ali z bajtom `0x01`, če je vrednost spremenljiva.

```
globaltype ::= t:valtype m:mut  ⇒ m t
mut        ::= 0x00             ⇒ const
             | 0x01             ⇒ mut
```

V tekstovnem zapisu globalni tip zapišemo z vrednostnim tipom t. Rezervirana beseda `mut` se pojavi le pri spreminjajoči se spremenljivki, ki jo zapišemo pred vrednostni tip t in v oklepajih tako:

```
globaltype ::= t:valtype                ⇒ const t
             | '(' 'mut' t:valtype ')'  ⇒ var t
```

Primer 2.5 prikazuje definicijo globalne spremenljivke z globalnim tipom `(mut i32)`, ki označuje spremenljivo globalno spremenljivko tipa `i32`. V binarni obliki bi ta globalni tip zapisali z bajtoma `7F 01`.

2.1.4 Vrednosti

Bajti

Bajti so osnovne vrednosti v binarni obliki in kodirajo sami sebe. V tekstovni obliki jih prav tako lahko zapišemo kot celo število. Čeprav nimajo svojega vrednostnega tipa, obstajajo ukazi vrednostnih tipov celih števil, ki operirajo z bajti. Bajti so torej:

$$\begin{array}{rcl} \texttt{byte} ::= \texttt{0x00} & \Rightarrow & \texttt{0x00} \\ \quad | & \ldots & \\ \quad | \quad \texttt{0xFF} & \Rightarrow & \texttt{0xFF} \end{array}$$

Cela števila

Cela števila se delijo na nepredznačena `u(N)` in predznačena `s(N)`. V binarni obliki so predstavljena z LEB128 [15]. Na predstavitev števila poleg predznačenosti vpliva tudi dolžina zapisa celega števila v bitih N.

Nepredznačena števila so predstavljena z nepredznačeno obliko LEB128. V kolikor je število manjše od 2^7, je število zapisano v enem bajtu. Večja števila se razdelijo na več bajtov. Prvi bajt dobimo tako, da vzamemo prvih 7 najmanj pomembnih bitov ter jim dodamo enko na najpomembnejši bit ($1a_6...a_1a_0$). Preostanek bitov obravnavamo po enaki formuli, zapisani spodaj, vedoč da N zmanjšamo za 7.

$$\begin{array}{rcll} \texttt{u(N)} ::= n{:}\texttt{byte} & \Rightarrow & n & (if\ n\ <\ 2^7) \\ \quad | \quad n{:}\texttt{byte}(\texttt{1a}_6\texttt{...a}_1\texttt{a}_0)\ m{:}\texttt{u}(\texttt{N}-7) & \Rightarrow & 2^7 \cdot m + (n - 2^7) & (if\ n\ \geq\ 2^7) \end{array}$$

Predznačena števila so predstavljena s predznačeno obliko LEB128, za predstavitev negativnih števil pa se uporablja dvojiški komplement.

$$\begin{array}{rcll} \texttt{s(N)} ::= n{:}\texttt{byte} & \Rightarrow & n & (if\ n\ <\ 2^6) \\ \quad | \quad n{:}\texttt{byte} & \Rightarrow & n - 2^7 & (if\ 2^6 \leq n\ <\ 2^7) \\ \quad | \quad n{:}\texttt{byte}(\texttt{1a}_6\texttt{...a}_1\texttt{a}_0)\ m{:}\texttt{s}(\texttt{N}-7) & \Rightarrow & 2^7 \cdot m + (n - 2^7) & (if\ n \geq 2^7) \end{array}$$

Tekstovna oblika zapisa celih števil je podobna drugim programskim jezikom. Razlikuje se po tem, da so lahko številke z desetiškim `num` ali

šestnajstiškim `hexnum` načinom zapisanega števila ločene s podčrtajem. Pri zapisu je seveda potrebno izbrati primeren vrednostni tip, katerega omejitve omogočajo zapis vrednosti števila. Pred šestnajstiško število postavimo rezerviran `0x`, pri predznačenih številih pa zapišemo tudi znak *pm* za + ali -, kot vidimo spodaj.

$$
\begin{array}{lll}
\texttt{u(N)} ::= n\texttt{:num} & \Rightarrow\ n & (if\ n < 2^N) \\
\quad | \quad \texttt{'0x'}\ n\texttt{:hexnum} & \Rightarrow\ n & (if\ n < 2^N) \\
\texttt{s(N)} ::= \pm\texttt{:sign}\ n\texttt{:num} & \Rightarrow\ \pm n & (if\ -2^{N-1} \leq \pm n < 2^{N-1}) \\
\quad | \quad \pm\texttt{:sign '0x'}\ n\texttt{:hexnum} & \Rightarrow\ \pm n & (if\ -2^{N-1} \leq \pm n < 2^{N-1})
\end{array}
$$

Primer 2.12 prikazuje zapis celega števila 2019. Vrednost je zapisana v desetiški obliki, kjer so posamezne števke ločene s podčrtajem. Ukaz bi bil enak, če bi zapisali `i32.const 2019` ali v šestnajstiški obliki `i32.const 0x7E3` ali `i32.const 0x7_E_3`. Binarni zapis števila je po zgornji formuli za nepredznačena števila enak `E3 0F`.

```
(module
  (func (param i32) (result i32)
    get_local 0
    i32.const 2_0_1_9
    i32.add
  )
)

00 61 73 6D 01 00 00 00 01 06 01 60 01 7F 01 7F
03 02 01 00 0A 0A 01 08 00 20 00 41 E3 0F 6A 0B
```

Primer 2.12: Program, ki sešteje vrednost 2019 s podanim parametrom.

Števila s plavajočo vejico

Števila s plavajočo vejico so predstavljena z enojno ali dvojno natančnostjo standarda IEEE 754-2008. V binarni obliki so zapisana z enakim bitnim zaporedjem kot določa standard (poglavje 3.4) po pravilu tankega konca.

$$\texttt{fN} ::= b^*\texttt{:byte}^{N/8} \Rightarrow \texttt{bytes}_{fN}^{-1}(b^*)$$

V tekstovni obliki lahko števila s plavajočo vejico predstavimo z desetiškim `float` ali šestnajstiškim `hexfloat` zapisom. Zapis za decimalno

vejico zapišemo po formuli za `frac`, podobno pri šestanjstiški predstavitvi z `hexfrac` tako:

```
frac    ::= ϵ                                          ⇒ 0
          | d:digit q:frac                             ⇒ (d + q)/10
          | d:digit '_' p:digit q:frac                 ⇒ (d + (p + q)/10)/10
hexfrac ::= ϵ                                          ⇒ 0
          | h:hexdigit q:hexfrac                       ⇒ (h + q)/16
          | h:hexdigit '_' p:hexdigit q:hexfrac        ⇒ (h + (p + q)/16)/16
```

V desetiškem zapisu je eksponent predstavljen s črko `E`, baza eksponenta pa je 10, medtem ko je v šestnajstiškem zapisu eksponent predstavljen s črko `P`, baza eksponenta pa je 2. Formula za zapis je taka:

```
float    ::= p:num '.' q:frac                                  ⇒ p + q
           | p:num ('E' | 'e') ± :sign e:num                   ⇒ p · 10^±e
           | p:num '.' q:frac ('E' | 'e') ± :sign e:num
               ⇒ (p + q) · 10^±e
hexfloat ::= '0x' p:hexnum '.' q:hexfrac                      ⇒ p + q
           | '0x' p:hexnum ('P' | 'p') ± :sign e:num          ⇒ p · 2^±e
           | '0x' p:hexnum '.' q:hexfrac ('P' | 'p') ± :sign e:num
               ⇒ (p + q) · 2^±e
```

Primer 2.13 prikazuje zapis števila $-1,23 \cdot 10^{-12}$ v desetiški obliki v vrednostnem tipu `f32`. Zapis v binarni obliki po standardu IEEE 754-2008 je `6C 1B AD AB`.

```
(module
  (func (param f32) (result f32)
    local.get 0
    f32.const -1.23E-12
    f32.add
  )
)

00 61 73 6D 01 00 00 00 01 60 01 60 01 7D 01 7D
03 02 01 00 0A 0C 01 0A 00 20 00 43 6C 1B AD AB 92 0B
```

Primer 2.13: Program, ki sešteje dve števili s plavajočo vejico.

Zapišemo lahko tudi NaN in neskončnost.

$$
\begin{array}{lll}
\texttt{fN} & ::= \pm\,\texttt{:sign}\ z\texttt{:fNmag} & \Rightarrow \pm z \\
\texttt{fNmag} & ::= z\texttt{:float} & \Rightarrow float_N(z) \\
& \mid\ z\texttt{:hexfloat} & \Rightarrow float_N(z) \\
& \mid\ \texttt{'inf'} & \Rightarrow \infty \\
& \mid\ \texttt{'nan'} & \Rightarrow nan(2^{signif(N)-1}) \\
& \quad (signif(32) = 23,\ signif(64) = 52) & \\
& \mid\ \texttt{'nan:0x'}\ n\texttt{:hexnum} & \Rightarrow nan(n) \quad (if 1 \leq n \leq 2^{signif(N)})
\end{array}
$$

Imena

Imena so zaporedja numeričnih predstavitev znakov, definiranih z Unicode. V binarni obliki so kodirana kot vektor bajtov UTF-8 [12] kodiranja le-teh. Vektorji pa so kodirani najprej z njihovo dolžino v u32, ki ji sledi kodiranje zaporedja elementov x^{n}.

$$
\begin{array}{lll}
\texttt{name} & ::= b^*\texttt{:vec(byte)} & \Rightarrow \texttt{name} \quad (if\ utf8(\texttt{name}) = b^*) \\
\texttt{vec(B)} & ::= n\texttt{:u32(x : B)}^{\mathtt{n}} & \Rightarrow x^n
\end{array}
$$

V tekstovni obliki so imena znakovni nizi.

$$\texttt{name} ::= b^*\texttt{:}string \Rightarrow c^*\ (if\ b^* = utf8(c^*))$$

Znakovni nizi lahko vsebujejo tekstovne ali binarne podatke, ki jih zapišemo v dvojnih narekovajih.

Niz ne sme vsebovati dvojnih narekovajev ('"'), ASCII kontrolnih znakov in poševnice ('\'), ki predstavlja ubežno kodo. Z ubežno kodo lahko zapišemo dvojne narekovaje, enojne narekovaje, poševnico in nekatere druge znake, če poznamo njihovo Unicodovo kodo, prav tako pa tudi binarne podatke, zapisane z bajtom v šestanjstiški obliki.

$$
\begin{array}{lll}
\texttt{string} & :== \texttt{'\'}\ (b^*\texttt{:stringelem})^*\ \texttt{'\'} & \Rightarrow concat((b^*)^*) \\
\texttt{stringelem} & ::= c\texttt{:stringchar} & \Rightarrow utf8(c) \\
& \mid\ \texttt{'\'}\ n\texttt{:hexdigit}\ m\texttt{:hexdigit} & \Rightarrow 16 \cdot n + m \\
\texttt{stringchar} & ::= c\texttt{:char} \mid \texttt{'\t'} \mid \texttt{'\\'} \ldots & \\
& \mid\ \texttt{'\u \{'}\ n\texttt{:hexnum '}\}\texttt{'} & \Rightarrow U + (n)
\end{array}
$$

Primer 2.1 prikazuje uporabo imen in znakovnih nizov v vseh možnih primerih. Imena `"imports"` in `"print"` so uporabljena pri uvozu funkcije `$print`, ime `"memory"` pa pri izvozu pomnilnika. Poleg imen pa imamo zapisan tudi znakovni niz `"Hello, world!"` pri definiranju podatkov, ki se zapišejo v pomnilnik.

Identifikatorji

Identifikatorje `id` v tekstovni obliki zapišemo z znakom dolar `$`, ki mu sledi niz ASCII znakov. Ta ne sme vsebovati presledka, dvojnih narekovajev, pike, podpičja in oklepajev.

```
id     ::= '$' idchar+
idchar ::= '0' | ... | '9'
         |   'A' | ... | 'Z' | 'a' | ... | 'z'
         |   '!' | '#' | '<' | ...
```

Identifikatorje lahko pripišemo modulu, pomnilniku, tabeli, tipom, globalnim spremenljivkam in funkcijam. Znotraj funkcij lahko identifikator pripišemo parametrom, lokalnim spremenljivkam in strukturiranim ukazom. V binarni obliki pa identifikatorje nadomestijo številčni indeksi.

Poglavje 3

WebAssembly JavaScript vmesnik

WebAssembly JavaScript vmesnik potrebujemo, da lahko na spletu prevedemo WebAssemblyjevo binarno kodo v nabor ukazov ciljne arhitekture in dobimo primerek modula, ki ga nato uporabljamo. WebAssembly pri večini ukazov vrednosti pridobi s sklada, medtem ko ciljne arhitekture vrednosti pridobijo iz registrov. Ta razlika prevajalnikom omogoča, da lahko kar najbolje izkoristijo uporabo registrov. Prevod se zgodi v navideznem stroju brskalnika, ki ga uporabljamo, zato se lahko prevodi v različnih brskalnikih razlikujejo. Prevod v vsakem primeru pridobimo v objekt modula `Module`. Ko k temu modulu povežemo še vse uvoze in izvoze ter pripišemo pomnilnik ter vse inicializacije, dobimo izvedljivo kodo v objekt primerka modula `Instance`.

Hitrost je ena ključnih lastnosti WebAssemblyja in ko govorimo o hitrosti, je to predvsem primerjava hitrosti med JavaScriptom in WebAssemblyjem – primerjava od pridobivanja datoteke s strežnika do vključno izvajanja programa. Ker je WebAssembly ciljni jezik prevajalnikov, lahko prevajalniki kar najbolj optimalno prevedejo ukaze v WebAssembly. „Ker programerjem WebAssemblyja ni potrebno neposredno programirati, lahko WebAssembly priskrbi niz ukazov, ki so primernejši za stroje. Glede na to kaj opravlja vaša

koda, se ti ukazi izvajajo od 10% do 800% hitreje.“(Clark, 2017) [6] Ravno hitrejše izvajanje ukazov pa ponuja aplikacijam, ki so računsko zahtevnejše, možnost postavitve na splet.

Uporabo nekaterih elementov vmesnika si bomo ogledali na primeru izpisa ”Hello, world!”

Za prikaz delovanja programa bomo uporabili html stran v primeru 3.1.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <title>Hello, world!</title>
</head>
<body>
  <p id='text'></p>
  <script src='hello.js'></script>
</body>
</html>
```

Primer 3.1: Html stran pri uporabi programa za izpis „Hello, world!“

Za izpis besedila „Hello, world!“ napišemo `hello.wat` (primer 3.2), ki ga nato s pomočjo orodja `wabt` prevedemo v binarno kodo `hello.wasm`.

```
(module
  (import "imports" "print" (func $print (param i32 i32)))
  (memory (export "memory") 1)
  (data (i32.const 0) "Hello, world!")
  (func $hello (export "hello")
    i32.const 0
    i32.const 13
    call $print
  )
)
```

Primer 3.2: Program hello.wat, ki izpiše „Hello, world!“

V zapisanem modulu najprej uvozimo funkcijo `$print`, nato definiramo pomnilnik in vanj zapišemo „Hello, world!“ Definiramo še funkcijo `$hello`, ki na sklad porine začetno pozicijo in dolžino besedila v pomnilniku ter kliče funkcijo `$print`. S komponento `export` označimo še izvoza, pomnilnik `memory` in funkcijo `$hello`, ki ju bomo po inicializaciji modula lahko klicali v JavaScriptu.

Obstaja več možnih načinov, kako pridobiti binarno kodo na splet. Priporočajo uporabo `fetch` vmesnika, ki odgovor vrne v objektu `Promise`. Ko postane odgovor dostopen, ga pretvorimo v `ArrayBuffer`.

Objekt WebAssemblya v JavaScriptu deluje kot imenski prostor za vse funkcionalnosti, povezane z WebAssemblyjevim JavaScript vmesnikom. Ta objekt nam nudi 4 metode:

1. `boolean validate(BufferSource bytes)` – potrjevanje pravilnosti kode,
2. `Promise<Module>compile(BufferSource bytes)` – prevod kode v objekt modula,
3. `Promise<Instance>instantiate(Module moduleObject, object importObject)` – opredelitev modula skupaj z uvoznim objektom v primerek le-tega,
4. `Promise<WebAssemblyInstantiatedSource>instantiate (BufferSource bytes, object importObject)` – opredelitev kode z uvoznim objektom v modul in njegov primerek.

Torej, po pridobitvi kode naprej le-to prevedemo z metodo `compile`, nato pa jo opredelimo v primerek z metodo `instantiate`, ki kot parameter vzame objekt modula. Četrta metoda `instantiate` je sestavljena iz prejšnjih dveh in nam vrne objekt modula in objekt primerka.

V hello.js moramo najprej zapisati:

```
fetch('hello.wasm')
.then(response => response.arrayBuffer())
.then(buffer => WebAssembly.instantiate(buffer, importObject))
.then(result => {});
```

Primer 3.3: Pridobitev WebAssemblyjeve binarne kode in njen prevod.

Preko primerka modula lahko kličemo izvozne objekte. V našem primeru sta to pomnilnik `memory` in funkcija `hello`. Vse izvozne objekte dobimo v primerku v objektu `exports`. Naš primer potrebuje dostop do pomnilnika,

zato ga bomo shranili v prej deklarirano spremenljivko `memory`, nato pa bomo klicali funkcijo `hello`, ki nam bo izpisala besedilo „Hello, world!“ Tako zapisu v primeru 3.3 dodamo:

```
.then(result => {
  memory = result.instance.exports.memory;
  result.instance.exports.hello()});
```

Primer 3.4: Pridobitev pomnilnika in klicanje izvožene funkcije `hello`.

Manjka nam definicija uvoznega objekta. Ta objekt je sicer lahko prazen, vendar bi nam v našem primeru metoda `instantiate` vrnila napako.

Uvozimo lahko katerokoli funkcijo, ki je na voljo v JavaScriptu, poleg tega pa tudi pomnilnik, ki ga ustvarimo s konstruktorjem `WebAssembly.Memory`, tabelo – `WebAssembly.Table` in globalne spremenljivke – `WebAssembly.Global`. Velja tudi obratno, pomnilnik, tabelo in globalne spremenljivke lahko izvozimo iz WebAssemblyja in jih z JavaScriptom spreminjamo.

Napisali bomo svojo funkcijo `print`, ki prejme dva parametra – pozicijo in dolžino. Medpomnilnik, ki vsebuje `memory`, naprej pretvorimo v prikaz medpomnilnika znakov s funkcijo Uint8Array, ki ji podamo tudi pozicijo in dolžino. Ker so tekstovni segmenti v WebAssemblyju zapisani z UTF-8, moramo pridobljene bite še dekodirati. Pridobljeno pa zapišemo v html.

```
function print(offset, length) {
  var bytes = new Uint8Array(memory.buffer, offset, length)
  var s = new TextDecoder('utf8').decode(bytes);
  document.getElementById("text").innerHTML=s;
}
```

Primer 3.5: Funkcija print.

Sedaj lahko definiramo še uvozni objekt. Kot vidimo v WebAssemblyjevi kodi, imajo uvozni elementi imenski prostor na dveh ravneh. Funkcija `$print` je tako dostopna na `imports.print`. Uvozni objekt v JavaScriptu mora odražati enak imenski prostor na dveh ravneh, po katerem zapišemo ime funkcije, ki jo uvažamo.

```
var importObject = {
  imports: {print: print}
}
```

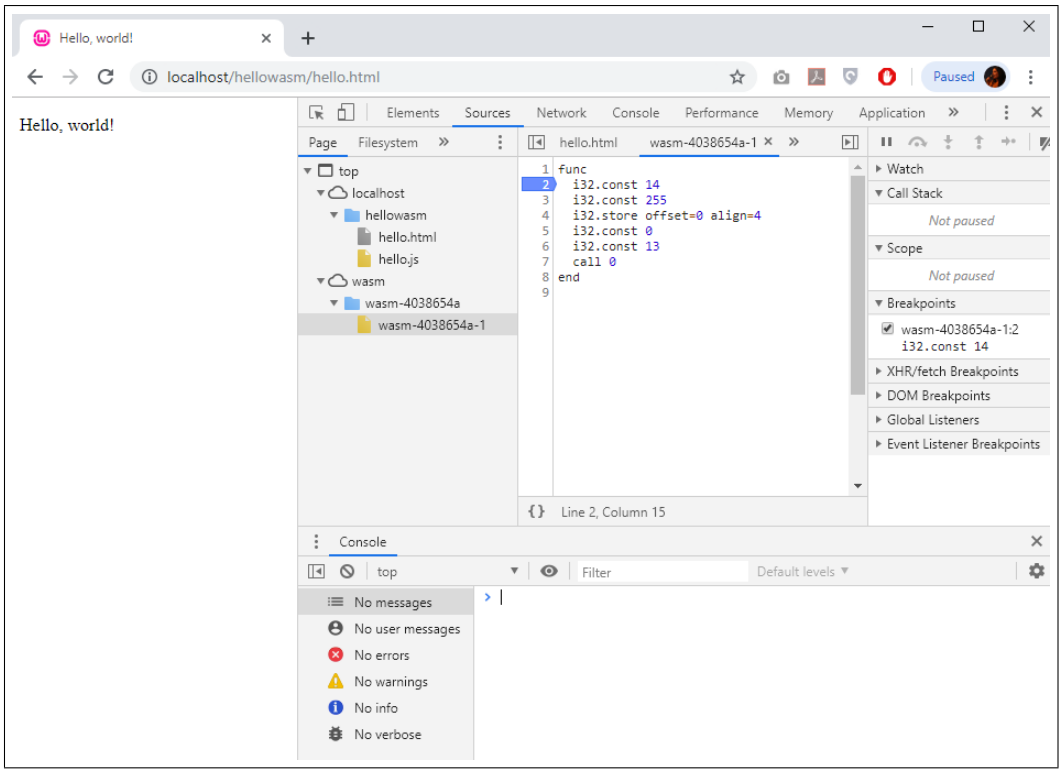
Primer 3.6: Uvozni objekt, ki uvaža funkcijo `print`.

Dokument in elementi html strani so dostopni šele po tem, ko se naložijo na splet, zato funkcijo `fetch` zavijemo v funkcijo `window.onload`. Pri izvajanju fetch vmesnika se lahko pojavijo tudi napake, zato nanje ne smemo pozabiti. Celotna koda JavaScript datoteke je zapisana v primeru 3.7. Sledi zgolj

```
var memory;

function print(offset, length) {
  var bytes = new Uint8Array(memory.buffer, offset, length)
  var s = new TextDecoder('utf8').decode(bytes);
  document.getElementById("t").innerHTML=s;
}

var importObject = {
  imports: {print: print}
}

window.onload = function() {
  fetch('hello.wasm')
  .then(response => response.arrayBuffer())
  .then(buffer => WebAssembly.instantiate(buffer,
          importObject))
  .then(result => {
    memory = result.instance.exports.memory;
    result.instance.exports.hello();
  })
        .catch(e => console.log(e));
}
```

Primer 3.7: Celotna koda JavaScript datoteke `hello.js`.

še izvajanje napisane kode na lokalnem strežniku, kjer je prikaz v spletnem brskalniku Chrome tak, kot kaže slika 3.1. V spletnem brskalniku Chrome je odprto okno s pregledom WebAssemblyjeve kode v tekstovni obliki. Ta pregled nam omogoča razhroščevanje tako JavaScripta kot WebAssemblyja.



Slika 3.1: Izvajanje programa `hello.wat` v spletnem brskalniku Chrome.

Poglavje 4

Prevajalniki v WebAssembly in druga orodja

Wasm je oblikovan kot prenosni cilj za prevajanje visokonivojskih jezikov, kot so C, C++ in Rust, ki na spletu omogočajo razvoj odjemalskih in strežniških aplikacij [14]. Tako se z WebAssemblyjem programer sreča le posredno. Prevajalnik navadno vrne WebAssemblyjevo binarno kodo in JavaScript povezovalno kodo, saj WebAssembly trenutno še ne more delovati brez JavaScript okolja, drugi vmesniki pa še niso implementirani. Binarna koda WebAssemblya se zapiše v datoteko s predlagano končnico wasm.

4.1 Emscripten

Emscripten je orodje za prevajanje v asm.js in WebAssemblyju, zgrajeno z uporabo LLVM, ki omogoča zaganjanje C in C++ v spletu [4].

WebAssembly je bil zasnovan tako, da se lahko vanj brez večjih težav prevede programski jezik C in C++.

Delovanje Emscriptna bomo preizkusili v operacijskem sistemu Windows 10. Pred namestitvijo Emscriptna potrebujemo Git za prenos prevajalnika ter Python 2.7.x ali novejši, saj je Emscripten Pythonova skripta. Po namestitvi preverimo, da imajo novonameščeni programi svoj naslov zapisan v

spremenljivki PATH.

Ukazno vrstico odpremo v mapi, kjer želimo imeti prevajalnik, ter ga prenesemo in namestimo z ukazi:

```
$ git clone https://github.com/juj/emsdk.git
$ cd emsdk
$ emsdk install latest
$ emsdk activate latest
```

Za nadaljevanje potrebujemo program v C ali C++. Uporabili bomo sledeči program, ki izpiše Hello, world! in smo ga zapisali v datoteko hello.c.

```
#include <stdio.h>

int main(int argc, char ** argv) {
  printf("Hello, world!\n");
  return 0;
}
```

V mapi emsdk poiščemo Windows paketno datoteko `emcmdprompt.bat` in jo poženemo. Ta zažene ukazno vrstico in požene ukaz `emsdk_env.bat`, ki nastavi spremenljivko PATH za trenutno sejo, da lahko dostopamo do ukazov prevajalnika.

Prevajalnik, do katerega dostopamo z ukazom `emcc`, nam ob predložitvi datoteke, ki jo prevajamo, vrne WebAssemblyjevo binarno kodo `hello.wasm` ter JavaScript povezovalno kodo `hello.js`, saj je WebAssembly že v osnovi omogočen.

```
$ emcc hello.c
```

V kolikor nam prevajalnik ne vrne WebAssemblyjeve binarne kode, ukazu dodamo stikalo `-s WASM=1`. Da lahko vidimo izpis v spletnem brskalniku, potrebujemo še html datoteko. Pridobimo jo lahko z definiranjem izhodne datoteke `-o hello.html`. Dodano stikalo -g pa nam vrne tekstovno obliko WebAssemblyja.

```
$ emcc hello.c -s WASM=1 -g -o hello.html
```

Ker brez strežniškega okolja WebAssembly ne bo deloval, lahko z ukazom `emrun` poženemo spletni strežnik, ki odpre spletni brskalnik in prikaže rezultat našega prevoda:

```
$ emrun --port 8080 hello.html
```

V spletnem brskalniku se pojavi stran, kot kaže slika 4.1.

Slika 4.1: Spletni prikaz emscriptenovega prevoda.

Emscripten prevede zgolj funkcijo `main`, ostale funkcije preskoči oz. jih vstavi v program, če jih glavna funkcija kliče. Če želimo imeti ostale funkcije dosegljive, jih moramo dodati kot izvožene funkcije.

```
$ emcc -s "EXPORTED_FUNCTIONS=['_main', '_my_func']" ...
```

Alternativno lahko spremenimo c program tako, da pred imenom druge funkcije zapišemo `EMSCRIPTEN_KEEPALIVE`. Poleg tega pa na začetku programa dodamo knjižnico `emscripten/emscripten.h`.

```
#include <stdio.h>
#include <emscripten/emscripten.h>

void EMSCRIPTEN_KEEPALIVE printHello() {
  printf("Hello, world!\n");
}

int main(int argc, char ** argv) {
  printHello();
  return 0;
}
```

4.2 Rust

Drugi podprti jezik, ki ga lahko prevajamo v WebAssembly, je Rust. Vse informacije za začetek prevajanja najdemo na strani Rust and WebAssembly [10].

Na operacijskem sistemu Windows 10 najprej potrebujemo `Visual Studio` ali `Visual C++ build tools`, da lahko namestimo orodje za `Rust 1.30` ali novejši. Poleg tega potrebujemo še paket `wasm-pack`, ki je ključen za prevod Rusta v WebAssembly, ter `npm`, ki je prevajalnik paketov za JavaScript. Po namestitvi preverimo, da lahko prek ukazne vrstice dostopamo do programov.

Tudi tu bo naš cilj izpis "Hello, World!"Najprej ustvarimo nov projekt.

```
$ cargo new --lib hello
```

Ta ukaz ustvari mapo hello, ki vsebuje `Cargo.toml` ter `src/lib.rs`. Najprej spremenimo rust datoteko, da ima sledečo vsebino:

```
extern crate wasm_bindgen;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern {
  pub fn alert(s: &str);
}
#[wasm_bindgen]
pub fn greet() {
  alert(&format!("Hello, World!"));
}
```

Na začetku Rust programa uvozimo knjižnico `wasm-bindgen`, ki je del paketa `wasm-pack`. Ta knjižnica poskrbi za komunikacijo med Rustom in

JavaScriptom. Pri prvi uporabi knjižnice iz Rusta kličemo zunanjo funkcijo `alert`, ki pripada JavaScriptu. Pri drugi uporabi nato kreiramo Rustovo funkcijo `greet` z uporabo zunanje funkcije, ki bo kasneje na voljo za klicanje v JavaScriptu.

Spremeniti moramo tudi datoteko `Cargo.toml`, ki nosi podatke o odvisnostih in metapodatke za cargo, program za upravljanje paketov in orodij za gradnjo. V datoteko dodamo:

```
[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2"
```

Sedaj lahko zgradimo projekt z ukazom, ki ga poženemo znotraj mape projekta `hello`. Ta ukaz s cargo programom prevede Rustovo kodo v WebAssemblyjevo binarno kodo, nato pa z uporabo `wasm-bindgen` generira JavaScript povezovalno kodo.

```
$ wasm-pack build
```

Rezultat grajenja je v podmapi `hello/pkg`, kjer najdemo `hello.wasm` WebAssembly binarno kodo, `hello.js` JavaScript kodo ter `package.json`, ki vsebuje metapodatke o grajenju teh kod.

Potrebujemo še spletno stran, kjer lahko uporabimo ustvarjeno kodo. Pomagamo si lahko s predlogo `create-wasm-app`. Ukaz poženemo v mapi projekta `hello`.

```
$ npm init wasm-app www
```

Datoteke predloge najdemo v podmapi `hello/www`.

V nadaljevanju moramo še povezati naš projekt s predlogo. Najprej namestimo lokalni strežnik z ustreznimi odvisnostmi, tako da v mapi `hello/www` poženemo:

```
$ npm install
```

Nato moramo ustvariti povezavo med našim projektom in predlogo. Premaknemo se v podmapo `hello/pkg` in poženemo:

```
$ npm link
```

V podmapi `hello/www` pa poženemo:

```
$ npm link hello
```

Po povezovanju moramo spremeniti še `index.js` v podmapi `hello/www`, tako da uvozimo naš paket `hello`. Po potrebi spremenimo tudi `index.html`. `Index.js` spremenimo v sledečo vsebino:

```
import * as wasm from "hello";
wasm.greet();
```

Sedaj lahko naš projekt preizkusimo na lokalnem strežniku. V podmapi `hello/www` poženemo:

```
$ npm run start
```

Ta ukaz požene lokalni strežnik in naš projekt naloži na `http://localhost:8080/`.

4.3 WABT

WebAssemblyjev binarni komplet programskih orodij najdemo na GitHubovi strani Wabt [13]. Orodja vključujejo pretvorbo tekstovne oblike WebAssemblyja v binarno ter obratno, interpreter, pretvorbo binarne kode WebAssemblyja v C program in drugo. Za nas je najbolj pomembno orodje `wat2wasm`, ki WebAssembly pretvori iz tekstovne v binarno obliko.

Za njegovo namestitev na operacijskem sistemu Windows 10 potrebujemo `Cmake`, `Visual Studio` in `Git`.

Orodja najdemo v mapi `wabt/bin`. Potrebujemo program `wat2wasm`, za program `hello.wat`, ki smo ga napisali v poglavju 3. Program nam vrne datoteko `hello.wasm`.

```
$wat2wasm \path-to-file\hello.wat
```

```
$git clone --recursive https:://github.com/WebAssembly/wabt
$cd wabt
$mkdir build
$cd build
$cmake .. -DCMAKE_BUILD_TYPE=DEBUG -DCMAKE_INSTALL_PREFIX=..
\bin -G "Visual Studio 15 2017"
$msbuild INSTALL.vcxproj
```

4.4 Binaryen

Binaryen je prevajalska in orodna knjižnica za WebAssembly, napisana v programskem jeziku C++ [2]. Komplet orodij se uporablja kot ozadje prevajalnikov, ki prevajajo v WebAssembly. Vključuje interpreter, orodja, ki optimizirajo kodo, prevajalnik WebAssembly binarne kode v JavaScript in drugo. Binaryen je vključen tudi v Emscripten.

4.5 Ostali prevajalniki

Na spletu najdemo že veliko prevodov programskih jezikov v WebAssembly, ki pa niso uradno podprti. Obširen seznam trenutno znanih prevedenih jezikov je zapisan na GitHubovi strani `Awesome WebAssembly Languages` [7].

Poglavje 5

Brainf*ck

Brainf*ck je ezoterični programski jezik, ki ga je ustvaril Urban Müller v letu 1993. Čeprav je sestavljen iz zgolj osmih ukazov, je njegova zmogljivost enakovredna Turingovemu stroju. Zaradi drugega dela imena, ki v angleškem jeziku predstavlja žaljivko, ga v literaturi najdemo pod raznimi imeni, ki to besedo ali del nje zamenjajo z * ali drugimi znaki.

Ukazi Brainf*cka so sledeči:

>	S kazalcem se premakne za ena desno.
<	S kazalcem se premakne za ena levo.
+	Bajt, na katerega kaže kazalec, poveča za 1.
-	Bajt, na katerega kaže kazalec, zmanjša za 1.
.	Vrednost bajta, na katerega kaže kazalec, izpiše na standardni izhod.
,	Vrednost iz standardnega vhoda zapiše v celico, na katero kaže kazalec.
[	Če je vrednost pod kazalcem enaka 0, skoči do naslednjega ukaza po ujemajočem]ukazu, v nasprotnem primeru pa nadaljuje z ukazi.
]	Če je vrednost pod kazalcem različna od 0, skoči do naslednjega ukaza po ujemajočem ukazu [, v nasprotnem primeru pa nadlajuje z ukazi.

Tabela 5.1: 8 ukazov Brainf*cka.

Drugih značilnosti Brainf*cka njegov stvaritelj ni natančno specificiral. V glavnem veljajo naslednje zakonitosti, ki pa jih mnogi priredijo po svoje.

- Pomnilnik je sestavljen iz 8-bitnih celic, ki so na začetku programa inicializirane na 0. Velikost pomnilnika je omejena na 30.000 celic. V celice se zapisuje nepredznačena števila, kjer se pri prelivanju uporabi dvojiški komplement števila.

- Kazalec na začetku programa kaže na prvo celico v pomnilniku, zato negativni pomnilnik v programih ne sme obstajati.

- Za prikaz znakov se največkrat uporablja UTF-8 kodiranje Unicode znakov z dodanim ANSI kodiranjem.

- Vhod naj se ne procesira, dokler ga uporabnik ne potrdi. Program nato prejme celoten niz znakov, ki ga navadno prebere z ukazi '>+[>,]'.

- Vsi znaki, različni od ukazov Brainf*cka, se tretirajo kot komentarji in so zato med izvajanjem ignorirani.

- Do sintaktične napake lahko pride zgolj pri pojavitvi oglatega zaklepaja pred ujemajočim oglatim uklepajem ter pri neujemajočem številu levih in desnih oglatih oklepajev.

Poglejmo dva primera programa Hello, world!, zapisana v Brainf*cku. Obema primeroma je skupno, da si najprej pripravita celice pomnilnika z željenimi vrednostmi. Te vrednosti so čim bližje vrednostim UTF-8 kodiranja Unicode znakov H, e, l, o, w, r, d, klicaja, vejice in presledka. Nato s prištevanjem in odštevanjem pridobi prave vrednosti in jih izpiše uporabniku. Primer 5.1 upošteva vse omenjene značilnosti Brainf*cka brez prelivanja in deluje v vsakem interpreterju ali prevajalniku. Primer 5.2 pri implementaciji upošteva prelivanje celic, torej 0 se z odštevanjem spremeni v 255, in negativni pomnilnik. Predvsem zaradi druge značilnosti programa bo ta deloval

v redkih interpreterjih in prevajalnikih. Oba primera in tudi druge najdemo na strani esolangs.org [3] pod iskanjem `Brainf*ck`.

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]
>>.>---.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

Primer 5.1: Program, ki izpiše `Hello, world!`

```
--<-<<+[+[<+>--->->->-<<<]>]
<<--.<++++++.<<-..<<.<+.>>.>>.<<<.+++.>>.>>-.<<<+.
```

Primer 5.2: Program, ki izpiše `Hello, world!`, a upošteva negativni pomnilnik.

Poglavje 6

Prevajalnik programskega jezika Brainf*ck v WebAssembly

V praktičnem delu bomo predstavili prevajalnik programskega jezika Brainf*ck v WebAssembly.

WebAssembly je v prvi vrsti namenjen za uporabo na spletu, zato smo prevajalnik postavili v spletnem okolju. Za implementacijo smo uporabili označevalni jezik html, programski jezik JavaScript ter WebAssembly v tekstovni in binarni obliki. Sestavljen je iz 3 datotek:

- Html stran za spletni prikaz, kamor uporabnik zapiše Brainf*ck program in prejme njegov prevod.
- JavaScript vmesnik, ki skrbi za pridobitev prevajalnikove kode ter komunikacijo med spletnim prikazom in prevajalnikom.
- WebAssemblyjev modul, ki prejme Brainf*ck program in ga prevede v WebAssemblyjevo binarno kodo.

.

Prevajalnik uporabniku vrne 3 datoteke:

- Html stran za spletni prikaz,
- JavaScript vmesnik,
- WebAssemblyjevo binarno kodo – prevod programa,

Html in JavaScript datoteki sta kot predlogi naloženi s strežnika in ponujeni uporabniku, medtem ko binarna koda nastane med prevajanjem na spletu. Vse tri datoteke, skupaj postavljene na strežnik, tvorijo izvedljiv prevedeni program.

Za potrebe izvajanja smo uporabili orodje WABT, s katerim smo prevedli WebAssemblyjevo tekstovno obliko v binarno, WampServer – spletni strežnik za operacijski sistem Windows, na katerega smo postavili prevajalnik ter ga testirali in razhroščevali v Google Chrome spletnem brskalniku.

V nadaljevanju si bomo ogledali sestavo prevajalnika in njegovo delovanje.

6.1 Spletni prikaz

Za prejem uporabnikovega programa potrebujemo html stran. Napišemo preprosto stran, prikazano v primeru 6.1, ki ima polje za vnos teksta ter gumb `compile`, na katerega uporabnik pritisne po vnesenem Brainf*ck programu, da sproži prevajanje. Po prevajanju mu prevajalnik vrne 3 datoteke v sekcijo izhoda, kjer JavaScript dinamično spremeni prikaz strani s povezavami na datoteke.

6.2 JavaScript vmesnik

Poleg html strani potrebujemo še JavaScript vmesnik. Ta poskrbi, da se WebAssemblyjeva koda pridobi s strežnika, opravi predpripravo na inicializacijo modula ter ga inicializira in požene. Po prevajanju uporabniku vrne rezultat prevajanja.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <title>Brainf*ck to WebAssembly compiler</title>
  <script src='compiler.js'></script>
</head>
<body>
  <h1>Brainf*ck to WebAssembly compiler in WebAssembly</h1>
  <div id='brainf*ck'>
    <p>Your brainf*ck code:</p>
    <textarea id='brainCode'></textarea>
    <button id='compile'>Compile!</button>
  </div>
  <div id='output'>
        </div>
</body>
</html>
```

Primer 6.1: Html stran – spletni prikaz prevajalnika.

Predpriprava zajema prenos uporabnikovega programa v pomnilnik WebAssemblya ter definiranje uvoznih objektov, ki so poleg pomnilnika še funkcije JavaScripta, ki jih potrebujemo med prevajanjem.

6.2.1 Pridobitev WebAssemblyjeve kode

Binarno kodo WebAssemblya pridobimo z ukazom `fetch`, kot prikazano v primeru 6.2. Kodo prevedemo in jo shranimo v spremenljivko `module`, tako da se pri večkratni uporabi prevajalnika binarna koda prenese samo enkrat ob nalaganju strani.

```
var module;

fetch('compile.wasm')
.then(response => response.arrayBuffer())
.then(buffer => WebAssembly.compile(buffer))
.then(result => module = result);
```

Primer 6.2: Pridobitev prevajalnika z ukazom `fetch`.

6.2.2 Prejem programa in opredelitev modula

V primeru 6.3 imamo kodo, ki prejme program in opredeli modul, tako da modulu določi uvozne objekte in kot rezultat vrne izvedljivo kodo v objektu primerka modula. Gumbu za prevajanje, ki je sicer dostopen šele, ko se stran naloži, dodamo upravitelja dogodkov, ki ob kliku poskrbi za prenos uporabnikovega programa v pomnilnik prevajalnika in prične s prevajanjem. Ker WebAssembly ne podpira string podatkovnih tipov, temveč operira samo s številskimi vrednostmi, Brainf*ckov program kodiramo z UTF-8, da pridobimo številske vrednosti posameznih znakov.

Do pripravljenega WebAssemblyjevega pomnilnika, kreiranega v JavaScriptu, dostopamo s prikazom medpomnilnika nizov, da lahko vanj zapišemo program. Pomnilnik, poleg funkcij `printWast` ter `moveCode`, deklariramo kot uvozni objekt.

Primerek modula pridobimo z metodo `WebAssembly.instantiate` in poženemo fukcijo `compile`, ki prejme dolžino kode programa v bajtih kot argument. Poskrbimo tudi za vse napake, ki se lahko zgodijo med povezovanjem uvoznih objektov z modulom ter med samim izvajanjem prevajalnika.

Funkcija moveCode

Ta funkcija je potrebna pri zapisu sekcije 10. Njen namen je premik določenega odseka v pomnilniku za ena v desno. To storimo z zanko, ki bajt za bajtom od desne proti levi premakne v desno. Omenjeno kodo najdemo v primeru 6.4.

Funkcija printWast

Primer 6.5 vsebuje funkcijo, ki je potrebna za izpis prevoda programa uporabniku, skupaj s predlogama JavaScript vmesnika in spletnega prikaza. Binarno kodo prevoda v podani dolžini od začetne pozicije pridobimo v prikaz podatkov, iz katerega lahko nato ustvarimo datoteko. Datoteko ustvarimo s pomočjo vmesnika `Blob`, kateremu določimo tip `application/x-binary`.

```
var memory = new WebAssembly.Memory({initial:10});

var importObject = {
        memory: {mem: memory}
  imports: {printwast : printWast, movecode: moveCode}
};

window.onload = function () {
  document.getElementById("compile").addEventListener("click",
        function(){
    code = document.getElementById("brainCode").value;

    if (code != null && code.length != 0) {
      var memView = new Uint8Array(memory.buffer);
      var codeView = new TextEncoder('utf-8').encode(code);
      memView.set(codeView);

      WebAssembly.instantiate(module, importObject)
      .then(result => result.exports.compile(codeView.length))
      .catch(e => console.error(e));
    }
    else {
      //your program is empty.
    }
  });
}
```

Primer 6.3: Prejem programa in opredelitev modula.

```
function moveCode(firstPos, lastPos) {
  var memView = new Uint8Array(memory.buffer);
  for (var i=lastPos; i > firstPos; i--) {
    memView[i+1] = memView[i];
  }
}
```

Primer 6.4: Funkcija `moveCode`.

Datoteko skupaj s predlogama, ki ju pridobimo s strežnika, predložimo uporabniku z nadpovezavo.

6.3 WebAssembly modul

Modul smo zapisali v tekstovni obliki WebAssemblyja, prevod uporabnikovega programa pa med prevajanjem zapišemo v pomnilnik v binarni obliki, saj moramo uporabniku ponuditi program, ki je takoj izvedljiv na spletu.

```
var brainfuckJS = window.document.createElement('a');
brainfuckJS.href = 'brainfuck.js';
brainfuckJS.download = 'brainfuck.js';
brainfuckJS.innerHTML = 'brainfuck.js';
var brainfuckHTML = window.document.createElement('a');

function printWast(offset, length) {
  output = document.getElementById('output');
  output.removeChild(output.childNodes[0]);

  var wasm = new DataView(memory.buffer, offset, length);

  var a = window.document.createElement('a');
  a.href = window.URL.createObjectURL(new Blob([wasm],
    {type: 'application/x-binary'}));
  a.download = 'brainfuck.wasm';
  a.innerHTML = 'brainfuck.wasm';
  output.appendChild(a);
  output.appendChild(brainfuckHTML);
  output.appendChild(brainfuckJS);
}
```

Primer 6.5: Funkcija `printWast`.

Modul najprej pripravi vse uvoze, ki jih definiramo pred ostalimi definicijami modula. Nato definiramo funkcijo `compile`, ki prične v pomnilnik zapisovati vse potrebne sekcije prevedenega programa. Dejanski prevod programa v ustrezne ukaze se zgodi pri zapisovanju sekcije kode. Po uspešnem prevodu programa kličemo še zunanjo funkcijo, ki iz pomnilnika izloči prevedeni program in ga ponudi uporabniku.

6.3.1 Prevod ukazov

Pričenjamo z opisom postopka prevajanja ukazov Brainf*cka, saj tu ustvarimo jedro programa in ugotovimo, katere lokalne spremenljivke in uvoze potrebujemo za sam prevajalnik in za prevedeni program. Sosledje ukazov prevoda bomo zapisali tako:

$$\texttt{get\_local 0} \;=\; \texttt{0x20 0}$$

Leva stran enačaja predstavlja tekstovni zapis ukazov, desna stran pa ekvivalentne ukaze v binarni obliki. Slednja se uporabi v prevajalniku.

Preko ukazov gremo z zanko, ki bajt za bajtom preverja, ali se ujema s katerim od ukazov Brainf*cka ali ne. Prepoznane ukaze obravnavamo, ostale znake pa ignoriramo. To preverjamo do konca programa, katerega dolžina je parameter funkcije `compile`.

Pri obravnavi znakov poleg dolžine programa potrebujemo še 3 lokalne spremenljivke. Prva, `pointer`, je kazalec na začetek uporabnikovega programa v pomnilniku, s katerim gremo preko ukazov. Druga, `position`, je prav tako kazalec na pomnilnik, z njim pa označujemo mesto, kamor lahko pišemo prevod programa. Tretja spremenljivka, `lcount`, je števec števila zank, s katerim si pomagamo pri določitvi sintaktične pravilnosti programa.

Zanko po vsakem obravnavanem bajtu nadaljujemo z ukazom `br`, ki se sklicuje na zanko. To velja tudi za bajt, ki ni prepoznan kot znan ukaz, pri katerem moramo prav tako povečati kazalec na program.

Da zanko lahko prekinemo, jo zavijemo v strukturo bloka, nanj pa se z ukazom `br` sklicujemo pri preverjanju na začetku zanke, kot kaže primer 6.6.

```
(block $overexit
  (loop $over
    (if (i32.eq (get_local $pointer)(get_local $len))
      (then br $overexit)
    )
    ;;instructions check
    ;;...
    (set_local $pointer
     (i32.add (get_local $pointer)(i32.const 1)))
    br $over
  )
)
```

Primer 6.6: Zanka, s katero pregledamo ukaze Brainf*cka.

Ukaza < in >

Ta dva ukaza kazalec na pomnilnik premakneta za ena v levo (<) oziroma za ena v desno (>). Kazalec na pomnilnik je edina lokalna spremenljivka Brainf*ck programa, zato ima indeks 0 in jo naložimo na sklad z ukazom

`get_local 0`. Za njo naložimo na sklad še število 1 z ukazom `i32.const 1`, ki ga nato z ustreznim ukazom prištejemo `i32.add` oziroma odštejemo `i32.sub`. Rezultat zapišemo nazaj v lokalno spremenljivko z ukazom `set_local 0`. Ukazi so torej:

```
get_local 0 = 0x20 0
i32.const 1 = 0x41 1
i32.add     = 0x6a | i32.sub = 0x6b
set_local 0 = 0x21 0
```

Te ukaze po vrsti shranimo v pomnilnik v bloku sekcije kode. Znak < prepoznamo po bajtu 0x3c, znak > pa po bajtu 0x3e. Po vsakem zapisu v pomnilnik povečamo lokalno spremenljivko `position` za ustrezno število bajtov, zato da vemo, na katero pozicijo zapišemo naslednje bajte. Tudi lokalno spremenljivko `pointer` pomaknemo za ena v desno in nato nadaljujemo z zanko, kot kaže primer 6.7.

Ukaza + in -

+ in - spreminjata vrednost v pomnilniku, na katero kaže kazalec. Ukaz `i32.load8_u` potrebuje na skladu pozicijo, s katere neko vrednost naloži iz pomnilnika. Ta pozicija je shranjena v lokalni spremenljivki, ki kaže na pomnilnik, zato jo pridobimo z ukazom `get_local 0`. Nato pridobljeni vrednosti prišteje `i32.add` oziroma odšteje `i32.sub` število 1 `i32.const 1`, rezultat pa shrani `i32.store8` na isto pozicijo v pomnilniku. Ukaz `store` potrebuje dve vrednosti na skladu, druga je vrednost, ki jo pridobimo z računanjem, prva pa pozicija, zato moramo le-to naložiti na sklad `get_local 0` pred vsemi drugimi ukazi. Brainf*ck program operira z bajtnimi vrednostmi, zato smo

```
(if $right
 (i32.eq (i32.load8_u (get_local $pointer))(i32.const 0x3e))
  (then
    ;; get_local 0
    (i32.store16 (get_local $position)(i32.const 0x0020))
    (set_local $position
     (i32.add (get_local $position)(i32.const 2)))
    ;; i32.const 1
    (i32.store16 (get_local $position)(i32.const 0x0141))
    (set_local $position
     (i32.add (get_local $position)(i32.const 2)))
    ;; i32.add
    (i32.store8 (get_local $position)(i32.const 0x6a))
    (set_local $position
     (i32.add (get_local $position)(i32.const 1)))
    ;; set_local 0
    (i32.store16 (get_local $position)(i32.const 0x0021))
    (set_local $position
     (i32.add (get_local $position)(i32.const 2)))
    (set_local $pointer
     (i32.add (get_local $pointer)(i32.const 1)))
    br $over
  )
)
```

Primer 6.7: Obravnava ukaza >, zapis ukazov prevoda v pomnilnik ter ustrezno povečanje lokalnih spremenljivk za ustrezno nadaljevanje zanke.

uporabili ukaza `i32load8_u` in `i32.store8`. Ukazi si sledijo tako:

```
get_local 0  = 0x20 0
get_local 0  = 0x20 0
i32.load8_u = 0x2d 0 0
i32.const 1  = 0x41 1
i32.add      = 0x6a | i32.sub = 0x6b
i32.store8   = 0x3a 0 0
```

Znak + prepoznamo po bajtu 0x2b, znak – po bajtu 0x2d. Ukaza obravnavamo podobno kot kaže primer 6.7, seveda z zapisom ustreznih prevedenih ukazov v pomnilnik.

Ukaza [in]

Ukaza [in] tvorita zanko. Vsak od njiju ima svoj pogoj, s katerim odloča o nadaljevanju zanke.

Znak [naznanja začetek zanke, v kolikor pa je vrednost pod kazalcem enaka 0, se ta preskoči. Pogoj preverimo tako, da najprej na sklad naložimo pozicijo kazalca `get_local 0` in iz pomnilnika pridobimo vrednost pod kazalcem z ukazom `i32.load8_u`. Pridobljeno vrednost lahko primerjamo z 0 z ukazom `i32.eqz`. Da lahko zanko preskočimo, jo zavijemo v blok `block`, na katerega se sklicujemo pri izpolnjenem pogoju z ukazom `br_if 0`, v nasprotnem primeru pa pričnemo z zanko `loop`. Ukazi si sledijo tako:

```
block       = 0x02 0x40
get_local 0 = 0x20 0
i32.load8_u = 0x2d 0 0
i32.eqz     = 0x45
br_if 0     = 0x0d 0
loop        = 0x03 0x40
```

Znak] predstavlja konec zanke. Pred zaključkom zanke preverimo, če je vrednost pod kazalcem enaka 0. Preverjanje pogoja naredimo z enakimi ukazi kot pri znaku [`get_local 0`, `i32.load8_u` ter `i32.eqz`. Če je pogoj izpolnjen, zanko zaključimo `br_if 1`, v nasprotnem primeru pa z zanko nadaljujemo `br 0`. Indeks 1 pri ukazu textttbr_if se sklicuje na blok, indeks 0 pri ukazu `br` pa na zanko. Bajt 0x0b se na koncu pojavi dvakrat, saj moramo zaključiti tako zanko kot blok:

```
get_local 0 = 0x20 0
i32.load8_u = 0x2d 0 0
i32.eqz     = 0x45
br_if 1     = 0x0d 1
br 0        = 0x0c 0
end         = 0x0b
end         = 0x0b
```

Zanka je tudi mesto, kjer lahko pride do sintaktične nepravilnosti programa. Tretjo lokalno spremenljivko `lcount` spreminjamo tako, da jo ob znaku [povečamo za ena, ob znaku] pa zmanjšamo za ena. Prva nepravilnost se preverja pri ukazu], kot kaže primer 6.8. Prevajanje se zaključi, v kolikor ta ukaz pred njim nima ujemajočega [ukaza. Druga nepravilnost se preverja po zaključenem prevajanju ukazov, kot prikazano v primeru 6.9.

Prevajalnik zaključi z izvajanjem, v kolikor je spremenljivka `lcount` različna od 0, kar označuje, da se ena od zank ni zaključila.

Znak [prepoznamo po bajtu 0x5b, znak] pa po bajtu 0x5d.

```
(if $loopr (i32.eq (i32.load8_u (get_local $pointer))
(i32.const 0x5d))
  (then
    (if (i32.eqz (get_local $lcount))
      (then unreachable) ;;trap
    )
    ;;count loop
    (set_local $lcount (i32.sub (get_local $lcount)
    (i32.const 1)))
    ;;instructions
    ;; ...
    br $over
  )
)
```

Primer 6.8: Obravnava ukaza] ter ugotavljanje sintaktične pravilnosti programa.

```
(block $overexit
  ;;loop ...
)
(if (i32.eqz (get_local $lcount))
  (then br 0) ;;continue
  (else unreachable) ;;trap
)
```

Primer 6.9: Ugotavljanje sintaktične pravilnosti programa pri koncu prevoda.

Ukaza . in ,

Ta dva ukaza predstavljata komunikacijo z uporabnikom. Komunikacija poteka preko JavaScript vmesnika, zato sta tu potrebni dve uvoženi funkciji JavaScripta.

Ukaz . vrednost pod kazalcem izpiše na izhod, zato to vrednost naložimo na sklad z ukazoma `get_local 0` in `i32.load8_u` ter kličemo funkcijo `output` z ukazom `call $output`, ki s sklada vzame naloženi parameter. Ukazi so

torej:

```
get_local 0   = 0x20 0
i32.load8_u   = 0x2d 0 0
call $output  = 0x10 0
```

Ukaz , iz vhoda prepiše vrednost v pomnilnik na mesto, kamor kaže kazalec. Funkcija `input` po klicanju z ukazom `call $input` vrne vrednost, ki jo je uporabnik vnesel in jo postavi na sklad. Zato da vrednost zapišemo na pravo mesto z ukazom `i32.store8`, moramo še pred klicanjem zunanje funkcije na sklad naložiti vrednost kazalca na pomnilnik `get_local 0` tako:

```
get_local 0  = 0x20 0
call $input  = 0x10 1
i32.store8   = 0x3a 0 0
```

Znak . prepoznamo po bajtu 0x2e, znak , pa po bajtu 0x2c.

6.3.2 Uvozi

Uvoze definiramo v začetku modula, kot kaže primer 6.10. Imamo 3 uvozne objekte, že omenjene pri JavaScript vmesniku. Prvi je pomnilnik, ki ima na začetku vpisan Brainf*ck program, vanj pa zapišemo njegov prevod. Druga dva uvoza sta funkciji. Prvo funkcijo `printwast` potrebujemo za posredovanje prevoda programa uporabniku. Druga funkcija `movecode` pa premakne odsek kode v pomnilniku za ena v desno. Njihova definicija vsebuje kombinacijo enakih imen, kot so deklarirana v JavaScript vmesniku. Če bi se ta razlikovala, bi sprožili LinkError.

```
(memory (import "memory" "mem") 10)
(import "imports" "printwast" (func $printwast (param i32 i32)
))
(import "imports" "movecode" (func $movecode (param i32 i32)))
```

Primer 6.10: Uvoženi pomnilnik ter funkciji, potrebni pri prevodu.

6.3.3 Funkcija compile

To je glavna funkcija prevajalnika, ki prejme parameter `len` – dolžino Brainf*ck programa. Najprej definira vse potrebne lokalne spremenljivke. Spoznali smo že tri, kazalec `pointer`, ki kaže na začetek uporabnikovega programa v pomnilniku in gremo z njim preko ukazov uporabnikovega programa, `position`, ki kaže na pomnilnik, kamor zapišemo prevod programa, ter `lcount`, s katerim štejemo število zank. Poleg teh tu potrebujemo še 3. Dve od teh sta namenjeni za shranitev vrednosti kazalca `position`, kjer zapišemo dolžino kode telesa funkcije Brainf*cka `funlenpos` ter dolžino kode sekcije 10 `remembpos`, ki postaneta znani šele, ko zaključimo s prevajanjem ukazov. Zadnjo lokalno spremenljivko `value` potrebujemo za hranjenje začasne vrednosti.

Nato v pomnilnik zapiše sekcije prevoda. Potrebne sekcije so sekcija tipov, sekcija uvozov, sekcija funkcij, sekcija pomnilnika, sekcija začetne funkcije ter sekcija kode. Pri vsaki sekciji predstavimo bajte, ki so zapisani v šestnajstiškem številskem sistemu.

Potrebujemo tudi 3 uvozne objekte. Prvi je pomnilnik z uporabnikovim programom. Druga dva pa sta funkciji. Prva funkcija je potrebna za prestavitev kode v desno, v kolikor je velikost dolžine večja od 2^7, druga funkcija pa nam pomaga pri izpisu prevoda uporabniku.

Lokalne spremenljivke in čarobno število

Funkcijo `compile` pričnemo, kot kaže primer 6.11. Lokalne spremenljivke morajo biti definirane na začetku funkcije, inicializiramo pa jih z ukazom `set_local`. Večino inicializiramo z 0, le spremenljivka `position` prejme vrednost `len`, saj ne smemo prepisati Brainf*ck programa.

Vsak WebAssembly program se začne s čarobnim številom in verzijo WebAssemblyja. Čarobno število zapišemo v obrnjenem vrstnem redu, saj se števila zapisujejo po pravilu tankega konca. Po vsakem zapisu v pomnilnik povečamo spremenljivko `position` za ustrezno število bajtov.

```
(func (export "compile") (param $len i32)
  (local $pointer i32)
  (local $position i32)
  (local $remembpos i32)
  (local $funlenpos i32)
  (local $value i32)
  (local $lcount i32)
  (set_local $pointer (i32.const 0))
  (set_local $position (get_local $len))
  (set_local $remembpos (i32.const 0))
  (set_local $funlenpos (i32.const 0))
  (set_local $lcount (i32.const 0))
  (block $magic
    (i32.store (get_local $position)(i32.const 0x6d736100))
    (set_local $position (i32.add (get_local $position)
     (i32.const 4)))
    (i32.store (get_local $position)(i32.const 1))
    (set_local $position (i32.add (get_local $position)
     (i32.const 4)))
  )
        ;; ...
)
```

Primer 6.11: Pričetek funkcije `compile`.

Sekcija 1 – tipi

Sekcija 1 definira vse tipe funkcij, ki se pojavijo v modulu, definirane in uvožene funkcije. Sekcija ima najprej bajt z identifikacijsko številko `01` in nato dolžino kode sekcije, zapisano z `u32`, v našem primeru je dolžina kode `0c`. V sekciji tipov nato zapišemo število različnih tipov, v našem primeru so to trije `03`. Tipi so trenutno zgolj funkcijski, zato za vsako funkcijo zapišemo bajt `60`. Temu sledita število parametrov in število rezultatov z njihovimi vrednostnimi tipi.

Prva funkcija je funkcija za zapis izhoda. Ta ima en parameter tipa `i32` in nič rezultatov, kar zapišemo z bajti `60 01 7f 00`. Druga funkcija je pridobitev vhoda, ki nima parametrov, ima pa en rezultat tipa `i32`, kar zapišemo z bajti `60 00 01 7f` . Sledi še glavna funkcija, ki izvaja vse ukaze, ta pa nima niti parametrov niti rezultatov, kar zapišemo z bajti `60 00 00`. Ta sekcija je zapisana v primeru 6.12.

```
01 0c 03 60 01 7f 00 60 00 01 7f 60 00 00
```

Primer 6.12: Sekcija tipov prevedenega programa.

Sekcija 2 – uvozi

Sekcija 2 definira uvoze, kot kaže primer 6.13. Uvozi so definirani z dvema imenoma in tipom uvoza.

Po indeksu sekcije `02` in dolžini sekcije `22` sledi število uvozov. Brainf*ck program ima dva uvoza `02`, in sicer že prej omenjeni funkciji za izhod in vhod. Za vsako od imen uvozov pred samim imenom, zapisanim z UTF-8 kodiranimi znaki Unicode-a, zapišemo dolžino imena v bajtih. Prvi uvoz za izhod je definiran z imeni `'imports'`, kar znaša sedem znakov `07`, ki so zapisani z bajti `69 6d 70 6f 72 74 73`, in `'output'`, ki ima šest znakov `06` in so predstavljeni z bajti `6f 75 74 70 75 74`. Drugi uvoz za vhod pa je definiran z imeni `'import2'`, kar znaša sedem znakov `07`, ki so zapisani z bajti `69 6d 70 6f 72 74 32`, ter `'input'`, ki ima 5 znakov `05` in so zapisani z bajti `69 6e 70 75 74`. Oba uvoza sta funkciji, zato po imenih prejmeta bajt `00` ter ustrezni indeks tipa, definiranega v sekciji 1, prvi uvoz ima indeks 0, drugi pa indeks 1. Ta sekcija je zapisana v primeru 6.13.

```
02 22 02
07 69 6d 70 6f 72 74 73 06 6f 75 74 70 75 74 00 00
07 69 6d 70 6f 72 74 32 05 69 6e 70 75 74 00 01
```

Primer 6.13: Sekcija uvozov prevedenega programa.

Sekcija 3 – funkcija

Sekcija 3 definira vse funkcije v modulu, ki niso uvozi.

Indeks sekcije je prvi bajt `03`, drugi bajt je dolžina sekcije `02`. V Brainf*ck modulu imamo zgolj eno funkcijo `01`, ki je tretji definirani tip v sekciji 1 z indeksom 2, zapisanem v četrem bajtu `02`. Zapis te sekcije najdemo v primeru 6.14.

```
03 02 01 02
```

Primer 6.14: Sekcija funkcij prevedenega programa.

Sekcija 5 – pomnilnik

Sekcija 5 definira pomnilnik, kot kaže primer 6.15. Prvi bajt je indeks sekcije `05`, ki mu sledi dolžina sekcije `03`. Sledi število pomnilnikov, ki je v trenutni verziji WebAssemblyja lahko zgolj en in ima indeks 0, kar je zapisano z bajti `01 00`. Brainf*ck program naj bi imel pomnilnik velik 30.000 bajtov. V WebAssemblyju tako natančne velikosti ni mogoče specificirati, zato definiramo pomnilnik velikosti `01`, kar znaša 65536 bajtov, ki so ob inicializaciji vsi nastavljeni na 0. Zapis te sekcije najdemo v primeru 6.15.

```
05 03 01 00 01
```

Primer 6.15: Sekcija pomnilnika prevedenega programa.

Sekcija 8 – start

Sekcija 8 definira start funkcijo, ki se požene takoj po opredelitvi modula. Sekcijo najedemo v primeru 6.16. Prva dva bajta sta indeks sekcije in njena dolžina `08 01`. Našo funkcijo lahko določimo kot start funkcijo, ker ta nima niti parametrov niti rezultata, ima pa indeks `02`. Tako se tudi izognemo potrebi po izvozu funkcije.

```
08 01 02
```

Primer 6.16: Sekcija začetne funkcije prevedenega programa.

Sekcija 10 – telo funkcije

Sekcija 10 definira vsa telesa funkcij, definiranih v modulu. Sekcija je zapisana v primeru 6.17.

Prvi bajt je indeks sekcije `0a`, a dolžina sekcije, ki mora biti zapisana v naslednjem bajtu ali bajtih, je na začetku neznana, saj se bo prej opisani

prevod ukazov šele začel izvajati. Zato tu pustimo prazen bajt `00`, vendar si zapomnimo njegovo pozicijo v spremenljivko `remembpos` za kasnejše spreminjanje.

Dolžini sekcije sledi vektor teles funkcij. Imamo zgolj 1 funkcijo, ki je zapisana z bajtom `01`, katere dolžina telesa je prav tako neznana. Enako tu pustimo prazen bajt `00` in si v drugi spremenljivki `funlenpos` zapomnimo mesto, kamor bomo kasneje vpisali pravo dolžino.

Sledi definicija lokalnih spremenljivk. V vektorju imamo zgolj eno spremenljivko `01`, ta spremenljivka pa je tipa `i32`, kar zapišemo z bajtoma `01 7f`, do katere bomo dostopali z indeksom 0. Po definiranju te jo še inicializiramo na 0, saj mora kazalec kazati na prvi bajt v pomnilniku. Inicializiramo jo z ukazoma `i32.const 0 set_local 0`. V binarni obliki bi ta dva ukaza zapisali z bajti `41 0 21 0`.

Zatem se prične v pomnilnik zapisovati prevod ukazov Brainf*ck programa. Po končanju zanke na konec pripišemo bajt `0b`, ki označuje konec funkcije.

```
0a 00 01 00 01 01 7f 41 0 21 0 ... instructions ... 0b
```

Primer 6.17: Sekcija kode prevedenega programa, kjer so bajti prevedenih ukazov na mestu, kjer je zapisano `instructions`.

Dve vrednosti sekcije 10 na tej točki ostajata neopredeljeni – dolžina telesa funkcije in dolžina sekcije. Pridobitev dolžine je trivialna – od trenutne pozicije odštejemo pozicijo, kamor moramo zapisati dolžino, in jo shranimo v začasno spremenljivko `value`. Zaplete se pri zapisu te vrednosti.

Vsaka sekcija predvideva zapis dolžine z nepredznačenim celim številom, ki je največ 32-bitno, enako velja za dolžino telesa funkcije. Ta števila so v WebAssemblyju kodirana z LEB128.

Če je dolžina kode manjša od 2^7, težave ni, v nasprotnem primeru moramo najprej kodo, od pozicije zapisa dolžine pa do konca trenutno zapisane, premakniti za ena v desno. To storimo z uvoženo funkcijo `movecode`, kot kaže primer 6.19. Nato moramo pridobiti pravi bajt, ki ga zapišemo na pozicijo

zapisa dolžine. Pri tem upoštevamo, da se vrednosti v pomnilnik zapišejo po pravilu tankega konca. Pomagamo si s pomožno funkcijo v primeru 6.18, ki prejme dolžino, izračuna ostanek pri deljenju z 2^7 ter temu ostanku prišteje 2^7. Tako pridobimo 7 najmanj pomembnih bitov števila z dodano enko na mestu najpomembnejšega bita.

```
(func $leb (param $n i32)(result i32)
  get_local $n
  i32.const 128
  i32.rem_u
  i32.const 128
  i32.add
)
```

Primer 6.18: Pomožna funkcija za pridobitev pravega zapisa števila za zapis dolžine.

Pridobljeno vrednost zapišemo v pomnilnik, dolžino pa zmanjšamo za 2^7. Ustrezno povečamo tudi spremenljivki `position` in `funlenpos` oziroma `remembpos` za 1 ter nadaljujemo z zanko. Dolžino kode funkcije moramo zapisati pred dolžino sekcije.

Izpis programa

Izpis programa se zgodi po končanem zapisu sekcije 10, saj je to zadnja sekcija, ki jo potrebujemo. Za izpis poskrbi uvožena funkcija `printwast`, ki potrebuje dva argumenta. Prvi je pozicija začetka prevedenega programa, ki je zapisan v spremenljivki `len`, drugi je dolžina prevedenega programa, ki jo izračunamo, kot kaže primer 6.20.

6.4 Html in JavaScript predlogi

Ti dve datoteki sta sestavni del prevedenega programa. Z njima pričnemo z izvajanjem prevedenega programa in komuniciramo z uporabnikom.

```
(set_local $value (i32.sub (get_local $position)
 (get_local $funlenpos)))
(loop $setlength
  (if (i32.ge_u(get_local $value)(i32.const 128))
    (then
      get_local $funlenpos
      get_local $position
      call $movecode
      get_local $funlenpos
      get_local $value
      call $leb
      i32.store8
      (set_local $value (i32.div_u (get_local $value)
       (i32.const 128) ))
      (set_local $position (i32.add (get_local $position)
       (i32.const 1)))
      (set_local $funlenpos (i32.add (get_local $funlenpos)
       (i32.const 1)))
      br $setlength
    )
    (else
      get_local $funlenpos
      get_local $value
      i32.store8
    )
  )
)
```

Primer 6.19: Zanka, s katero v pomnilnik zapišemo dolžino kode funkcije.

```
(set_local $position (i32.sub (get_local $position)
 (get_local $len)))
get_local $len
get_local $position
call $printwast
```

Primer 6.20: Izpis programa s klicem funkcije `printWast`.

6.4.1 Spletni prikaz

Preprost spletni prikaz s tekstovnim poljem, kjer se izpisuje izhod programa, prikazuje primer 6.21.

6.4.2 JavaScript vmesnik

JavaScript vmesnik najprej pridobi Brainf*ck program s strežnika ter modul opredeli, tako da mu določi uvozne objekte in vrne izvedljivo kodo v

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <title>Brainfuck</title>
  <script src='brainfuck.js'></script>
</head>
<body>
  <div>
    <h1>Brainfuck program</h1>
    <textarea id='output'></textarea>
  </div>
</body>
</html>
```

Primer 6.21: Spletni prikaz prevedenega programa.

objektu primerka modula. Nato avtomatično požene začetno funkcijo, kot kaže primer 6.22.

```
var importObject = {
  imports: {output: output},
  import2: {input: input}
};

window.onload = function () {
  output = document.getElementById("output");
  fetch('brainfork.wasm')
  .then(response => response.arrayBuffer())
  .then(buffer => WebAssembly.instantiate(buffer,
          importObject));
};
```

Primer 6.22: JavaScript vmesnik prevedenega programa.

Funkcija `output`, ki izpiše izhod, najprej pretvori znakovno kodo v znak, nato ga izpiše uporabniku, kot v primeru 6.23.

```
var output;

function output(val) {
  var s = String.fromCharCode(val);
  output.innerHTML += s;
}
```

Primer 6.23: Funkcija `output`.

Funkcija `input`, primer 6.24, prejme vhod uporabnika. Za pridobitev

vhoda najprej sproži pozivnik, v katerega lahko uporabnik vnese vhod. Pozivnik zaustavi delovanje JavaScripta, dokler uporabnik ne zapre okna s pozivnikom. To je ključno pri izvajanju programa, saj bi v nasprotnem primeru program nadaljeval z izvajanjem, vhoda uporabnika pa ne bi prejel ob pravem času.

V funkciji poskrbimo, da se zaporedje znakov vhoda in njegovo UTF-8 kodiranje shrani, da ob ponovnem klicu funkcije ne sprožimo pozivnika, v kolikor nismo zaključili s prejšnjim zaporedjem znakov.

```
var input;
var flag = 0;
var pos = 0;
var value;
function input() {
  if (flag == 0) {
    input = prompt("Your input:");
    if (input != null) {
      value = new TextEncoder('utf-8').encode(input);
      var temp = value[0];
      pos++;
      flag = 1;
      return temp;
    }
    else return 0;
  }
  else {
    if (pos == value.length) {
      flag = 0;
      return 0;
    }
    else {
      var temp = value[pos];
      pos++;
      return temp;
    }
  }
}
```

Primer 6.24: Funkcija `input`.

Poglavje 7

Sklepne ugotovitve

O WebAssemblyju najdemo na spletu vse več vsebin. Večinoma prejema pozitivne odzive in mnogi svoje aplikacije z njim nadgrajujejo. Med bolj znanimi aplikacijami je AutoCAD. Ustvarjalci so si kot cilj zadali prenosno obliko AutoCADa na spletu in to dosegajo z WebAssemblyjem. Uporabnikom so že na voljo beta različice aplikacije na spletu.

Stvaritelji WebAssemblyja si neprenehoma prizadevajo nadgraditi osnovno verzijo. Tudi specifikacija se nenehno spreminja in jo citiramo kot delo, ki je v procesu. Tekom izdelave diplomskega dela so se na primer spremenili zapisi nekaterih ukazov, med njimi je v diplomskem delu uporabljen ukaz `get_local`, katerega zapis v tekstovni obliki so spremenili v `local.get`. Zato priporočamo ogled zadnje izdaje specifikacije.

V prihodnosti bodo WebAssemblyju dodali mnoge funkcionalnosti. Velja omeniti dve. Prva je upravljanje pomnilnika (angl. garbage collection), ki bo omogočil boljši prevod programskih jezikov, ki ga uporabljajo (npr. Java). Druga funkcionalnost so niti. Zanje so že pripravili predlog, ki je bil na voljo za testiranje v Google Chrome brskalniku s pridobljenim žetonom za preizkušanje.

Sprva je bil cilj diplomskega dela v praktičnem delu napraviti prevajalnik iz Brainforka v WebAssembly, to je nadgrajena različica Brainf*cka z dodanim ukazom Y, ki ustvari novo nit programa. Niti bi lahko ustvarili s

spletnimi agenti (angl. WebWorker), vendar se ustavi pri dokončni izvedbi ukaza. Pri novoustvarjeni niti bi morali skočiti do sekcije programa, kjer smo ustvarili novo nit, vendar WebAssembly nima ukaza, ki bi skočil do tega dela programa, niti nima dostopnega kazalca na ukaz, ki se bo izvedel naslednji (angl. instruction pointer). Poleg tega ne bi mogli zagotoviti pravilnega izvajanja, kjer se niti izvajajo izmenično z enim ukazom naenkrat. WebAssembly zaenkrat še ne omogoča čakanja na določenem naslovu med izvajanjem programa.

Prevajalniki in orodja, ki so podprti s strani WebAssemblya, so sicer dobri in funkcionalni, vendar je za uporabnike Windows operacijskega sistema njihova namestitev zahtevnejša, saj so orodja najverjetneje bolj namenjena uporabnikom Linux operacijskih sistemov. Pri namestitvi programskega orodja Binaryen smo naleteli na napako med prevajanjem kode orodja v izvedljive programe. Po odpravi napake pa nismo mogli zagotoviti pravilnega delovanja programov orodja.

WebAssembly čaka še svetla prihodnost. Odpira nove možnosti za aplikacije na spletu, ki jih do zdaj z JavaScriptom niso mogli uresničiti. Uporabljajo ga že številne precej zahtevne spletne aplikacije (že prej omenjeni AutoCAD, SketchUp, Google Earth itd.), nedvomno pa bo napravil velike spremembe tudi na področju spletnih iger.

Literatura

[1] Andreas Rossberg Andreas Haas. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[2] Binaryen. Dosegljivo: `https://github.com/WebAssembly/binaryen`. [Dostopano: 6. 12. 2018].

[3] Brainf*ck. Dosegljivo: `https://esolangs.org/wiki/Brainfuck`. [Dostopano: 11. 12. 2018].

[4] emscripten. Dosegljivo: `http://emscripten.org`. [Dostopano: 4. 12. 2018].

[5] 754-2008 - ieee standard for floating-point arithmetic. Dosegljivo: `https://ieeexplore.ieee.org/document/4610935/`. [Dostopano: 11. 9. 2018].

[6] Clark Lin. A cartoon intro to webassembly. In *JSConf EU*, 2017.

[7] Awesome webassembly languages. Dosegljivo: `https://github.com/appcypher/awesome-wasm-langs`. [Dostopano: 4. 12. 2018].

[8] Anders Moller. Technical perspective: Webassembly: A quiet revolution of the web. *Communications of the ACM*, 61(12):106, 2018.

[9] Andreas Rossberg. Webassembly specification, release 1.0. Dosegljivo: https://webassembly.github.io/spec/core/bikeshed/, 2018. [Dostopano: 27. 8. 2018], Dokument je citiran kot delo v procesu.

[10] Rust and webassembly. Dosegljivo: https://rustwasm.github.io/. [Dostopano: 4. 12. 2018].

[11] Unicode® 11.0.0. Dosegljivo: http://www.unicode.org/versions/Unicode11.0.0/. [Dostopano: 11. 9. 2018].

[12] Utf-8. Dosegljivo: https://en.wikipedia.org/wiki/UTF-8. [Dostopano: 8. 11. 2018].

[13] Wabt: The webassembly binary toolkit. Dosegljivo: https://github.com/WebAssembly/wabt. [Dostopano: 6. 12. 2018].

[14] Webassembly. Dosegljivo: https://webassembly.org/. [Dostopano: 4. 12. 2018].

[15] Leb128. Dosegljivo: https://en.wikipedia.org/wiki/LEB128. [Dostopano: 26. 9. 2018].