VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÁ VERZE PROGRAMOVACÍHO JAZYKA BRAINFUCK A JEJÍ INTERPRET

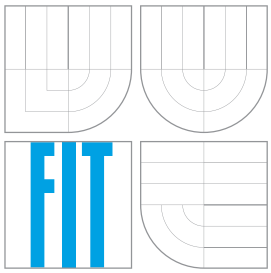
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARCEL FIALA

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÁ VERZE PROGRAMOVACÍHO JAZYKA BRAINFUCK A JEJÍ INTERPRET

EXTENDED VERSION OF THE BRAINFUCK PROGRAMMING LANGUAGE AND ITS INTERPRET

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MARCEL FIALA
AUTHOR

VEDOUCÍ PRÁCE — Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Autor v této práci analyzuje nedostatky v návrhu programovacího jazyka Brainfuck v kontextu jeho použitelnosti a udržovatelnosti, pro které dále navrhuje řešení v podobě rozšíření původního jazyka. Toto rozšíření následně formálně definuje a implementuje pro něj interpret a debugger.

Abstract

In this thesis, author discusses and analyzes design flaws of experimental programming language Brainfuck, for which he suggests solution in form of extension of original language. Then he formaly defines this extension and implements its interpret and debugger.

Klíčová slova

Brainfuck, Cleverer Brainfuck, CBF, interpret

Keywords

Brainfuck, Cleverer Brainfuck, CBF, interpret

Citace

Marcel Fiala: Rozšířená verze programovacího jazyka Brainfuck a její interpret, bakalářská práce, Brno, FIT VUT v Brně, 2015

Rozšířená verze programovacího jazyka Brainfuck a její interpret

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením profesora Alexandera Meduny. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Marcel Fiala
20. května 2015

Poděkování

Rád bych poděkoval Prof. RNDr. Alexanderovi Medunovi, CSc. za odborné vedení, trpělivost a ochotu, kterou mi v průběhu zpracování bakalářské práce věnoval.

Obsah

1 Úvod **3**
- 1.1 Cíl práce . . . 3
- 1.2 Struktura práce . . . 3

2 Brainfuck **4**
- 2.1 Historie . . . 4
- 2.2 Definice jazyka . . . 4
  - 2.2.1 Instrukce . . . 4
  - 2.2.2 Syntaxe . . . 5
- 2.3 Nedostatky v návrhu . . . 5
  - 2.3.1 Jednoduchost instrukcí . . . 6
  - 2.3.2 Arita instrukcí . . . 6
  - 2.3.3 Nedostatečné řídicí konstrukce a absence negace . . . 6
  - 2.3.4 Podpora práce s řetězci . . . 7
  - 2.3.5 Podprogramy . . . 7
  - 2.3.6 Další nedostatky . . . 8

3 Návrh rozšíření Cleverer Brainfuck **9**
- 3.1 Doplnění nejasné definice původního jazyka . . . 9
  - 3.1.1 Paměť . . . 9
  - 3.1.2 Chování při EOL a EOF . . . 10
  - 3.1.3 Komentáře ve zdrojovém souboru . . . 11
- 3.2 Návrh rozšíření jako řešení nedostatků původního jazyka . . . 11
  - 3.2.1 Nové „základní“instrukce . . . 11
  - 3.2.2 Abstraktní datový typ (ADT) zásobník . . . 12
  - 3.2.3 Řídicí konstrukce a negace . . . 13
  - 3.2.4 Definice vlastních procedur . . . 14
  - 3.2.5 Instrukce pro práci s řetězci, tisk čísel . . . 15
  - 3.2.6 Další rozšíření . . . 15
  - 3.2.7 Formální definice rozšíření . . . 16

4 Interpret pro Cleverer Brainfuck **18**
- 4.1 Implementace . . . 18
  - 4.1.1 Lexikální analyzátor . . . 19
  - 4.1.2 Syntaktický analyzátor . . . 20
  - 4.1.3 Generátor vnitřního kódu . . . 20
  - 4.1.4 Interpret vnitřního kódu . . . 20
- 4.2 Uživatelské rozhraní . . . 20

4.2.1 Vstup a výstup interpretovaného programu 20
4.2.2 Chyby 21
4.2.3 Varování 21
4.3 Přepínače interpretu 21

5 Debugger pro Cleverer Brainfuck **23**
5.1 Motivace 23
5.2 Implementace 23
5.3 Použití debbugeru 23
5.4 Příkazy 24

6 Závěr **25**
6.1 Splnění cílů práce 25
6.2 Požadovaná rozšíření jazyka Brainfuck 25
6.3 Možnosti dalšího vývoje 26
6.4 Shrnutí 26

A Obsah CD **28**
A.1 LL gramatika jazyka CBF 28
A.2 LL parsovací tabulka 28
A.3 Interpret `cbfrun` 28
A.4 Demo programy 28

Kapitola 1

Úvod

V roce 1993 vytvořil Urban Müler programovací jazyk Brainfuck s do té doby nejmenším kompilátorem. Obsahuje pouze 8 jednoduchých instrukcí, ovšem i přes svoji triviálnost splňuje podmínky Turingovské úplnosti[1].

Navzdory nepoužitelnosti pro reálné projekty se velmi krátce poté, co Müller publikoval svoji práci, stal Brainfuck mezi programátory velmi oblíbeným. Bylo v něm napsáno mnoho programů, dokonce i interprety Brainfucku samotného[2].

1.1 Cíl práce

Vzniklo mnoho různých rozšíření jazyka Brainfuck[3]. Téměř vždy se ale jednalo o drobné změny neřešící jeho hlavní „nedostatky“ — většinou pouze přidávala další možnosti pro experimentování. Pod pojmem „nedostatky“ se v tomto kontextu rozumí absence možností jazyka, které by usnadnily jeho reálné použití. Je nutno mít na paměti, že toto nebyl původní účel Brainfucku. Jedná se spíše o reakci na velkou oblibou Brainfucku, kdy s ním začalo mnoho programátorů experimentovat, ale jeho původní verze byla příliš minimalistická.

Autor se v rámci této bakalářské práce snažil vyřešit všechny tyto „nedostatky“ návrhem rozšíření původního jazyka, které nazval **Cleverer Brainfuck (CBF)**. Poté implementoval v jazyku C jeho interpret. Tento interpret zahrnuje i debugger, který výrazně zvyšuje komfort při vývoji programů v tomto novém jazyku.

1.2 Struktura práce

Ve druhé kapitole se autor seznamuje s původním jazykem Brainfuck, snaží se identifikovat jeho největší nedostatky a diskutuje míru jejich vlivu na použitelnost a udržovatelnost jazyka. Pro nalezené nedostatky navrhuje v rámci třetí kapitoly řešení ve formě rozšíření CBF. Ve třetí kapitole rovněž definuje toto rozšíření LL gramatikou.

Čtvrtá a pátá kapitola jsou věnovány implementačním detailům a ovládání interpretu pro CBF, respektive debuggeru, který je jeho součástí.

V poslední kapitole autor hodnotí dosažené výsledky a navrhuje další možný postup v tomto projektu.

[1] Viz `<http://www.iwriteiam.nl/Ha_bf_Turing.html>`

[2] Viz `<http://esolangs.org/wiki/Brainfuck\#Self-interpreters>`

[3] Přehled většiny rozšíření Brainfucku k nalezení na webu Esolangs, věnovaném esoterickým a experimentálním programovacím jazykům, viz `<http://esolangs.org/wiki/Category:Brainfuck_derivatives>`

Kapitola 2

Brainfuck

2.1 Historie

V roce 1993 vytvořil Urban Müler minimalistivký programovací jazyk Brainfuck[1], který měl do té doby nejmenší kompilátor (na Amiga OS 2.0 zabíral pouze 240B). Po uveřejnění práce však další autoři vytvořili i menší kompilátory, jeden z nich se dokonce vešel do limitu 100B[2].

Návrh Brainfucku byl ispirován jazykem **FALSE**[3] Woutera van Oortmerssena rovněž z roku 1993, který odstartoval éru esoterických programovacích jazyků.

Šest z osmi instrukcí Brainfucku odpovídá symbolům jazyka **P"**, který Corrado Böhm použil v roce 1964 pro popis Turingova stroje[2]. Müler však popřel, že by se při návrhu Brainfucku inspiroval Böhmovou prací.

2.2 Definice jazyka

Brainfuck je extrémně minimalistický imperativní programovací jazyk. Přestože vznikl jako příklad jazyka s nejmenším kompilátorem, v dnešní době se pro spouštění programů v Brainfucku využívá interpretů, kterých od roku 1993 vzniklo nespočet (ovšem neúplná definice jazyka způsobuje problémy, viz oddíl 2.3.6).

Návrh jazyka zahrnuje 8 jednoduchých instrukcí, doprava nekonečnou pásku (angl. *tape*) buněk (angl. *cells*) o velikosti 1B inicializovaných na nulu, které mohou nabývat pouze kladných hodnot (0 - 255), a datový ukazatel (angl. *data pointer*), který může v jeden okamžik ukazovat pouze na jednu z buněk. Návrh Brainfucku dále počítá se dvěma datovými proudy (angl. *data streams*) — vstupem a výstupem (angl. *input, output*).

2.2.1 Instrukce

Brainfuck obsahuje pouze 8 jednoduchých instrukcí, které pracují s pamětí popsanou výše. Jejich seznam a význam naleznete v tabulce 2.1.

[1] Viz dokument o Brainfucku v databázi experimentálních programovacích jazyků [3].
[2] Zdrojový kód a další informace na `<http://pferrie.host22.com/misc/brainfck.htm>`.
[3] Viz `<https://esolangs.org/wiki/False>`.

instrukce	význam
+	inkrementuj hodnotu v aktuální buňce
-	dekrementuj hodnotu v aktuální buňce
>	posuň ukazatel o jednu buňku doprava
<	posuň ukazatel o jednu buňku doleva
.	vytiskni znak s ordinální hodnotou z aktuální buňky
,	načti znak a ulož jeho ordinální hodnotu do aktuální buňky
[	pokud je hodnota v aktuální buňce nenulová, přejdi na další znak; jinak skoč za odpovídající uzavírací hranatou závorku
]	skoč zpět na odpovídající otevírací hranatou závorku

Tabulka 2.1: Instrukce původního jazyka Brainfuck včetně slovního popisu. Za aktuální buňku se považuje buňka, na kterou odkazuje datový ukazatel.

2.2.2 Syntaxe

Syntaxe jazyka Brainfuck je vskutku triviální. Všechny instrukce kromě hranatých závorek ([a]) se mohou v syntakticky správném kódu vyskytovat v libovolném pořadí.

Hranaté závorky reprezentující řídicí konstrukci **IF** se musí vždy vyskytovat ve dvojici v libovolném počtu zanoření, nikdy se však dvě konstrukce **IF** nesmí překrývat. Formální zápis těchto pravidel ve formě LL gramatiky se nachází na obr. 2.1.

$$
\begin{array}{lcl}
\text{PROGRAM} & \longrightarrow & \text{INSTRUCTION\_LIST} \quad \$ \\
\text{INSTRUCTION\_LIST} & \longrightarrow & \text{INSTRUCTION} \quad \text{INSTRUCTION\_LIST} \\
\text{INSTRUCTION\_LIST} & \longrightarrow & \text{CYCLE} \\
\text{INSTRUCTION\_LIST} & \longrightarrow & \epsilon \\
\text{CYCLE} & \longrightarrow & [\quad \text{INSTRUCTION\_LIST} \quad] \\
\text{INSTRUCTION} & \longrightarrow & + \\
\text{INSTRUCTION} & \longrightarrow & - \\
\text{INSTRUCTION} & \longrightarrow & > \\
\text{INSTRUCTION} & \longrightarrow & < \\
\text{INSTRUCTION} & \longrightarrow & . \\
\text{INSTRUCTION} & \longrightarrow & ,
\end{array}
$$

Obrázek 2.1: LL gramatika originálního jazyka Brainfuck. Neterminály jsou zapsány tiskacími písmeny, $ značí konec zdrojového souboru a symbol ϵ reprezentuje prázdný řetězec.

2.3 Nedostatky v návrhu

Protože Brainfuck nebyl vyvíjen s ohledem na použitelnost ani udržovatelnost, jeho návrh má mnoho „nedostatků“, které vylučují jeho reálné použití.

2.3.1 Jednoduchost instrukcí

Obě instrukce pro změnu hodnoty v buňce (+ a -), stejně jako instrukce pro posun datového ukazatele (< a >), změní hodnotu, resp. buňku, na kterou datový ukazatel odkazuje, pouze o jedničku.

To v praxi znamená, že pro přičtení např. hodnoty 42 k aktuální hodnotě v buňce, na kterou odkazuje datový ukazatel, musí programátor do zdrojového souboru napsat za sebe 42 instrukcí pro inkrementaci, případně napsat cyklus, který tuto instrukci 42krát použije za něj. Podobný problém nastává při posunu adresového ukazatele do větší vzdálenosti od aktuální buňky.

V důsledku je pak takový zdrojový soubor zbytečně rozsáhlý, nepřehledný, a tím pádem i velmi těžko udržovatelný. Je sice možné situaci o trochu zlepšit dodržováním konzistentního odsazování, např. pětice instrukcí stejného typu separovat mezerou, to ale programátora nezbaví nutnosti tyto n-tice instrukcí při úpravách kódu ručně přepočítávat.

Problém jednoduchosti původních instrukcí se neprojevuje nejen u triviálních matematických operací, ale i při přiřazování hodnot do buněk. Jakákoliv instrukce přiřazení hodnoty totiž zcela chybí. Pro nastavení libovolné hodnoty je tedy třeba n-krát použít inkrementaci nebo dekrementaci.

Druhotným důsledkem, který se projeví až při behu programu, je snížená efektivita. Přestože závisí na implementaci interpretu nebo překladače, provedení velkého množství inkrementací bude ve většině případů trvat déle než jedno přičtení dané hodnoty.

2.3.2 Arita instrukcí

Všechny instrukce Brainfucku jsou unární, tzn. mají pouze jeden operátor — buňku, na kterou odkazuje datový ukazatel. Toto eliminuje mnoho možností, jako jsou:

- aritmetické operace (sčítání, odčítání, násobení, dělení, ...);
- přímé přiřazování hodnot do buněk a jejich kopírování z jedné buňky do druhé;
- efektivnější práce s datovým ukazatelem — rychlý skok na velkou vzdálenost od aktuální buňky.

Toto má opět negativní vliv na použitelost Brainfucku, protože je třeba této funkcionality dosáhnout použitím jedoduchých unárních instrukcí, viz tabulka 2.1.

2.3.3 Nedostatečné řídicí konstrukce a absence negace

Pro řízení toku programu napsaném v Brainfucku je možno použít pouze jednu řídicí konstrukci — podmíněný vstup do bloku ohraničeného hranatými závorkami, resp. opakovaný skok na jeho začátek — tedy obdobu **WHILE** cyklu.

Přestože tuto konstrukci lze při umožnění pouze jednoho průchodu použít jako **IF** blok, není toto řešení ideální. Implementace řízení toku ve stylu **IF-ELSE** je pak ještě o mnoho komplikovanější a značně neefektivní. Podobné řízení toku programu je však naprosto běžná programátorská praxe a jednoduché používání konstrukce **IF-ELSE** je esenciální pro dobrou použitelnost libovolného programovacího jazyka.

Při diskusi návrhu řídicích kontrukcí je také nutno zmínit způsob vyhodnocování podmínek. Brainfuck se v tomto případě chová podobně jako většina programovacích jazyků, tedy nenulová hodnota znamená *true*, nulová pak *false*. Co ale Brainfucku naprosto chybí,

je možnost negace, což se často odkazuje jako poměrně zásadní problém při použití jediné řídicí kontrukce, kterou původní Brainfuck obsahuje.

2.3.4 Podpora práce s řetězci

Práce s textovými řetězci, minimálně co se týká načítání ze vstupu a tisk na výstup, není v Braifucku nikterak složitá. Obě tyto funkcionality lze realizovat nekolika málo instrukcemi, příklad viz obr. 2.2.

```
,[.,]
```

Obrázek 2.2: Implementace programu `cat` v originálním Brainfucku (EOF vrací 0).

Daleko komplikovanější je tisk řetězců, které se nenačítají za běhu programu ze vstupu. Do buněk je třeba uložit ordinální hodnoty jednotlivých znaků, což je vzhledem k neefektivitě původních instrukcí Brainfucku (viz oddíl 2.3.1) velmi zdlouhavé na implementaci a neefektivní při běhu programu. Příklad komplikovanosti tisku řetězců demonstruje fragment kódu na obr. 2.3.

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

Obrázek 2.3: Uložení a tisk loginu autora (`XFIALA47`) v originálním Brainfucku.

Dalším závažným nedostatkem v tisku na výstup je absence instrukce pro tisk hodnoty buňky jako čísla. Konverze hodnoty na řetezec pro tisk je netriviální funkcionalita, jejíž implementace je nejen neefektivní, ale i velmi náročná.

2.3.5 Podprogramy

Jedním z nejzásadnější nedostatků návrhu Brainfucku v kontextu použitelnosti a udržovatelnosti je nemožnost tvorby vlastních podprogramů (funkce, procedury), které lze volat v případě potřeby. To v kombinaci s absencí (podmíněných i nepodmíněných) skoků v rámci řízení toku programu znamená, že stejná funkcionalita potřebná v několika místech zdrojového souboru musí být n-krát rozkopírovaná v nezměnené podobě. Z toho vyplývají dva významné problémy:

- pokud je v rozkopírované části kódu chyba, je nutné ji najít a opravit na mnoha místech (hrozí riziko přehlédnutí chyby v jednom či více výskytech);
- délka zdrojového souboru se může až násobně zvětšovat, kód je méně přehledný, mnoho pasáží je reduntantních.

Oba tyto problémy zásadním způsobem ovliňují použitelnost jazyka. Rozkopírování jednotlivých částí kódu může vést ke mnoha těžko nalezitelným chybám, nehledě na tristní udržovatelnost v případě potřeby funkcionalitu upravit.

2.3.6 Další nedostatky

Brainfuck trpí celou řadou dalších drobných nedostatků, pro které jsou dva hlavní důvody:

- jazyk nebyl navržen pro praktické použití;
- definice jazyka je neúplná, proto je hodně implementačních detailů závislých na vůli programátora konkrétního interpretu.

Velikost buněk

Původní definice jazyka zmiňuje doprava nekonečnou pásku 8bitových nezáporných celých čísel. Originální překladač pak počítal s konečným počtem 30 000 buněk. Zatímco by toto množství proměnných, reps. buněk pro mnoho programů stačilo, paměťový prostor o velikosti 1B pro ukládání jednotlivých hodnot je značně omezující. Různé interprety/překladače tedy pracují s rozdílně velkými buňkami, což může způsobovat problémy u programů, kde se nějakým způsobem využívá vlastnosti přetečení hodnot.

Přetečení hodnot, posunutí datového ukazatele „za hranice“

Tyto vlastnosti jsou také nedostatečně definovány. Chování buňky při přetečení/podtečení hodnot (angl. *overflow, underflow*) se liší u každé implementace. V některých případech nelze překročit hraniční hodnotu, jindy dojde ke klasickému posunu na druhou stranu číselného intervalu (angl. *wrapping*).

Podobný problém nastává u implementace posunu datového ukazatele za danou hranici nebo před první buňku. Chování se velmi liší v každé implementaci, od resetu ukazatele až po ukončení interpretace chybou.

Nekonzistence konce řádku (EOL) na různých operačních systémech

Různé operační systémy používají odlišný způsop pro ukončování řádků. Microsoft Windows používá CRLF (ordinální hodnoty 13 a 10), UNIX-based operační systémy pak pouze ASCII znak s ordininální hodnotou 10.

Většina interpretů Brainfucku počítá se druhým způsobem ukončování řádků, je však nutno mít na paměti tento možný zdroj problémů s portabilitou.

Chování při čtení konce souboru (EOF)

Implementace této vlastnosti má zásadní vliv na práci s řetezci načtenými ze vstupu. Existují dvě možnosti:

- po načtení EOF se do buňky uloží hodnota *-1*, tedy největší možná hodnota — pak lze za všech okolností odlišit EOF od ostatních znaků;
- po načtení EOF se do buňky uloží hodnota *0* — toto umožňuje snazší implementaci tisku řetězce na výstup pomocí jednoduchého cyklu, který tiskne obsah všech buněk, dokud nenarazí na nulu.

Kapitola 3

Návrh rozšíření Cleverer Brainfuck

V této kapitole se autor věnuje návrhu řešení „nedostatků“ původního jazyku Brainfuck, které diskutoval v předchozí kapitole. Toto řešení ve formě rozšíření původního jazyka nazval **Cleverer Brainfuck (CBF)**.

3.1 Doplnění nejasné definice původního jazyka

Tento oddíl je vyhrazen pro dodatečnou definici, resp. redefinici některých vlastností jazyka Brainfuck, od kterých se pak bude odvíjet chování některých instrukcí.

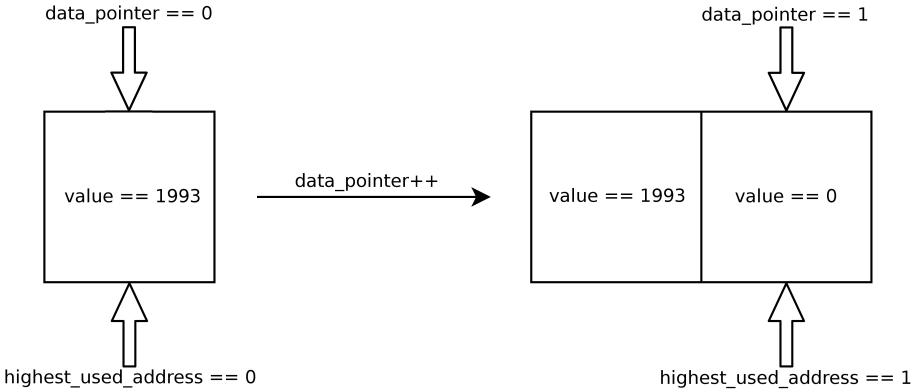
Už při návrhu rozšíření plánoval autor implementaci interpretu v jazyku C, některé případy definovaného chování jsou tedy závislé na vlastnostech tohoto programovacího jazyka. Jedná se především o velikost celočíselných proměnných.

3.1.1 Paměť

Paměť programu v jazyku CBF má tyto parametry:

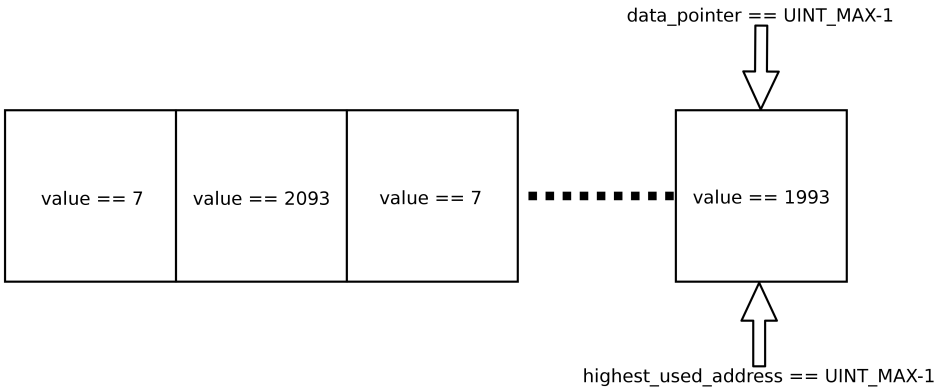
- základní paměťovou jednotkou je buňka s defaultní hodnotou 0;
- buňka má velikost `sizeof(unsigned int)` — závislá na OS, maximální hodnota `UINT_MAX`;
- přetečení hodnoty buňky funguje stejně jako v jazyku C;
- paměť programu v CBF je tvořena páskou buněk;
- páska buněk má defaultní délku 1 (adresa první buňky je 0);
- existuje jeden datový ukazatel inicializovaný na adresu 0;
- při posunu datového ukazatele doprava se dynamicky alokují další buňky až do délky pásky `UINT_MAX`;
- při posunu datového ukazatele doprava z poslední buňky (adresa `UINT_MAX-1`) se datový ukazatel přesune na adresu 0;
- při posunu datového ukazatele doleva se žádné nové buňky nealokují; v případě posunu doleva z první buňky (adresy 0) se datový ukazatel změní na adresu nejpravější použité buňky.

Příklad alokace nových buněk naleznete na obr. 3.1. V tomto konkrétním případě se jedná o alokaci druhé buňky při posunu datového ukazatele z první buňky směrem doprava.



Obrázek 3.1: Alokace nové buňky při posunu datového ukazatele doprava.

Nová alokace buněk probíhá až do délky pásky `UINT_MAX`. Tento hraniční stav je zachycen na obr. 3.2. Další alokace již neprobíhá, místo toho je při pokusu o posun datového ukazatele doprava nastavena jeho adresa na hodnotu 0, viz obr. 3.3.



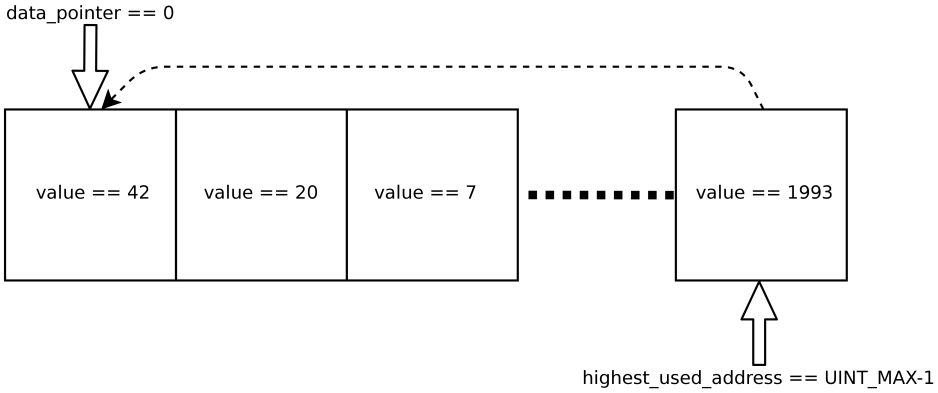
Obrázek 3.2: Stav paměti programu (pásky), kdy je alokováno maximální množství buněk.

Při pokusu o posun datového ukazatele z první buňky doleva je jeho hodnota nastavena na nejvyšší dosud použitou adresu, tzn. adresu nejlevější použité buňky, viz. obr. 3.4.

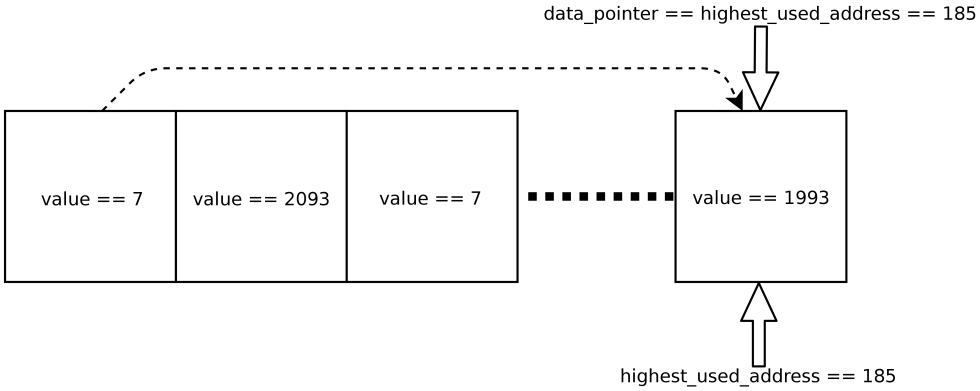
3.1.2 Chování při EOL a EOF

V kontextu použití CBF je konec řádku (EOL) po vzoru UNIX-based operačních systémů reprezentován jako *newline*, tedy ASCII znak s ordinální hodnotou 10.

Při čtení konce souboru ze vstupu je buňka nastavena na hodnotu 0.



Obrázek 3.3: Při pokusu o posunutí datového ukazatele doprava z buňky s adresou `UINT_MAX-1` se jeho hodnota nastaví na 0.



Obrázek 3.4: Při pokusu o posunutí datového ukazatele doleva z buňky s adresou se jeho hodnota nastaví na adresu nejlevější použité buňky.

3.1.3 Komentáře ve zdrojovém souboru

V Brainfucku komentáře jako takové neexistují. Interpret pracuje jenom s platnými znaky, všechny ostatní ignoruje. Komentář tedy může být naprosto kdekoliv ve zdrojovém souboru.

V Cleverer Brainfuck je již situace složitější. Protože bylo přidáno mnoho instrukcí a za platné je považováno daleko více znaků, bylo třeba přesně definovat pravidla pro vkládání komentářů. Autor vybral do té doby nepoužitý znak (`=`), který označuje začátek řádkového komentáře. Od tohoto znaku až po konec řádku se všechny znaky považují za komentář a jsou zcela ignorovány.

3.2 Návrh rozšíření jako řešení nedostatků původního jazyka

3.2.1 Nové „základní“ instrukce

Velkým problémem původního jazyka Brainfuck byla jednoduchost původních instrukcí (viz oddíl 2.3.1). Toto autor vyřešil přidáním několika nových instrukcí. Pro nastavení hodnoty aktuální buňky se jedná konkrétně o dvě, viz tabulka 3.1.

instrukce	význam
`'char'`	nastav hodnotu aktuální buňky na ordinální hodnotu znaku
`'int'`	nastav hodnotu aktuální buňky na danou hodnotu

Tabulka 3.1: Nové instrukce pro nastavení hodnoty aktuální buňky.

Místo `char` může být dosazen libovolný ASCII znak, případně jedna z těchto escape sekvencí: `\n`,`\'`.

Místo `int` lze dosadit libovolné číslo v desítkové nebo dvojkové soustavě. Pro jednoznačné rozlišení soustavy se používá jednoznakový prefix `d`, resp. `b`, viz tabulka 3.2.

literál	hodnota v desítkové soustavě
`d0` nebo `b0`	0
`d1` nebo `b1`	1
`d42`	42
`b1000010010010`	4242
`b222`	syntax error — binární číslo nemůže obsahovat 2ku

Tabulka 3.2: Příklady zápisu celých čísel v jazyku CBF.

Změna adresy v datovém ukazateli byl o trochu komplikovanější problém. Autor ze zkušenosti s programováním v Brainfucku ví, že se posun po pásce může rozdělit na dva druhy:

- posun o několik málo buňek (výběr jiné proměnné);
- posun o několik (desítek) buněk (skok do „odkládací“ části pásky, kam programátoři umisťují počítadla cyklů, pomocé proměnné, ...).

Proto se autor rozhodl umožnit změnu adresy v datovém ukazateli dvěma způsoby, kterými jsou krátký skok (angl. *short jump*) a dlouhý skok (angl. *long jump*). Dlouhý skok vyžaduje použití ADT zásobník, bude tedy detailně vysvětlen v oddílu 3.2.2 věnovaném právě zásobníku.

Krátký skok je pak rozšíření původního posunu datového ukazatele, realizované přidáním jedné číslice přímo za instrukci pro posun datového ukazatele. Krátký skok může mít délku 1-10 (číslice `0` za instrukcí reprezentuje skok délky 10). Příklady použití krátkého skoku naleznete v tabulce 3.3.

Často potřebnou instrukcí, která v Brainfucku chybí, je skok na první buňku na pásce (reset datového ukazatele). Instrukci pro tento speciální případ skoku rovněž naleznete v tabulce 3.3.

3.2.2 Abstraktní datový typ (ADT) zásobník

Zásobník je jeden z nejznámějších abstraktních datových typů, který má mnoho různých použití. Na zásobník u CBF se ukládají hodnoty stejného datového typu jako do standardních buněk na pásce a jeho kapacita je omezena pouze strojem, na kterém je program interpretován.

instrukce	význam
`>3`	posun o 3 buňky doprava
`<7`	posun o 7 buněk doleva
`>0`	posun o 10 buněk doprava
`_`	skok na první buňku (adresa 0)

Tabulka 3.3: Příklady krátkých skoků a resetu datového ukazatele.

Základní instrukce pro práci se zásobníkem najdete v tabulce 3.4. Samozřejmě, jeho přidání nebylo samoúčelné. Prvek na vrcholu zásobníku se považuje za druhý operand nově přidaných aritmetických instrukcí (viz tabuka 3.5).

Dalším využitím zásobníku je dlouhý skok datového ukazatele. Pro jeho realizaci bylo nutné definovat dvě nové instrukce (viz tabulka 3.6).

instrukce	význam
`/`	*push* — ulož hodnotu z aktuální buňky na vrchol zásobníku
`\`	*pop* — upokud není zásobník prázdný, přesuň hodnotu z vrcholu zásobníku do aktuální buňky
`?`	*top* — pokud není zásobník prázdný, zkopíruj hodnotu z vrcholu zásobníku do aktuální buňky
`!`	*remove* — pokud není zásobník prázdný, odstraň prvek z jeho vrcholu
`E`	pokud je zásobník prázdný, nastav hodnotu aktuální buňky na 1, jinak na 0 (dotaz na prázdnost zásobníku)
`H`	pokud není zásobník prázdný, inkrementuj hodnotu na jeho vrcholu
`L`	pokud není zásobník prázdný, dekrementuj hodnotu na jeho vrcholu

Tabulka 3.4: Základní sada instrukcí pro práci se zásobníkem.

3.2.3 Řídicí konstrukce a negace

Negace

Absence možnosti negace „výrazů“ je závažný nedostatek, který v Brainfucku značně omezuje možnosti použití cyklu. Z tohod úvodu autor zavedl instrukci, která umožňuje negaci terminální podmínky cyklu — viz příklad v tabulce 3.7.

Cykly s pevným počtem iterací (FOR)

Cyklus **FOR** je často používaná konstrukce, díky které umožňuje určit počet iterací cyklu. V jazyku Cleverer Brainfuck je jeho zápis opravdu jednoduchý, využívá se v něm dříve definované reprezentace číslených hodnot (viz oddíl 3.2.1). Několik příkladů v tabulce 3.8.

instrukce	význam
`A`	pokud není zásobník prázdný, přičti hodnotu z jeho vrcholu k hodnotě v aktuální buňce
`S`	pokud není zásobník prázdný, odečti hodnotu z jeho vrcholu od hodnoty v aktuální buňce
`M`	pokud není zásobník prázdný, vynásob hodnotou z jeho vrcholu hodnotu v aktuální buňce
`D`	pokud není zásobník prázdný, proveď celočíselné dělení hodnoty v aktuální tabulce hodnotou z jeho vrcholu

Tabulka 3.5: Instrukce pro aritmetické operace.

instrukce	význam
`&`	ulož aktuální adresu na vrchol zásobníku
`*`	pokud není zásobník prázdný, změň adresu v datovém ukazateli na hodnotu z vrcholu zásobníku; pokud je tato hodnota vyšší než aktuálně nejvyšší použitá adresa, doalokuj potřebné buňky

Tabulka 3.6: Instrukce pro realizaci dlouhého skoku datového ukazatele.

Řízení toku programu

Pro rozhodování mezi dvěma bloky kódu (větvemi) ve většině moderních programovacích jazyků slouží řídicí konstrukce **IF-ELSE**. Původní Brainfuck nic takového neumožňuje, což výrazným způsobem zhoršuje jeho použitelnost. Proto autor přidal do CBF řídicí konstrukci **IF** — kulaté závorky se stejnou syntaxí jako má originální cyklus (viz tabulka 3.9).

Samotná konstrukce **IF** ovšem nestačí. Když se ale použije vkombinaci s negací, lze kromě **IF** vytvořit i řídicí konstrukci **IF NOT** a **IF-ELSE** (viz tabulka 3.10).

3.2.4 Definice vlastních procedur

Možnost definice vlastních procedur je největším přínosem oproti původnímu jazyku Brainfuck. Eliminuje nutnost rozkopírování celých bloků kódu, címž zvyšuje přehlednost s udržovatelností a zároveň až násobně zkracuje celkovou délku zdrojových souborů.

CBF podporuje definici až 26 vlastních procedur s jednopísmennými lower-case indentifikátory. Tyto procedury musí být definovány ve speciání bloku v úvodu zdrojového souboru.

Každá procedura má unikátní identifikátor a složenými závorkami jasně ohraničené tělo. Volání procedur se provádí pouhým vložením indentifikátoru.

Procedury v CBF samozřejmě podporují nejen vzájemné volání, ale i rekurzi. Stejně jako ve všech programovacích jazycích ale platí, že se použitím rekurze vystavujeme nebezpečí přetečení zásobníku.

Příklad definice a volání procedur včetně použití rekurze je demonstrováno na obr. 3.5.

instrukce	význam
`[...]`	originální cyklus Brainfucku — ekvivalent konstrukce **WHILE** v jazyku C
`@[...]`	stejný cyklus, ovšem s negovanou podmínkou — lze nazvat **WHILE NOT**

Tabulka 3.7: Názorná ukázka negace podmínky u cyklu.

instrukce	význam
`#'b1'[...]`	tělo cyklu se provede jednou
`#'d42'[...]`	tělo cyklu se provede 42krát
`#'d0'[...]`	tělo cyklu se neprovede ani jednou

Tabulka 3.8: Příklady FOR cyklu.

3.2.5 Instrukce pro práci s řetězci, tisk čísel

Jak již autor zmínil v přechozí kapitole 2.3.4, podpora řetězců a tisk číselných hodnot je v Brainfucku na velmi nízké úrovni. Cleverer Brainfuck proto zahrnuje dvě další instrukce, které tento nedostatek řeší (viz tabulka 3.11). ***Pozor na použití symbolu pro negaci, v tomto kontextu má úplně jiný význam!*** Pro srovnání — kód v CBF na obr. 3.6 je ekvivalentní kódu v Brainfucku na obr. 2.3.

3.2.6 Další rozšíření

Vestavěná funkce *sort*

Řazení je funkcionalita, kterou pro svoj práci potřebuje mnoho dalších algoritmů. Protože CBF umožňuje pracovat pouze s číselnými hodnotami, vylučuje se možnost použití jakéhokoliv jiného klíče pro řazení. Proto přímo CBF obsahuje vestavěnou funkci, která vzestupně seřadí všechny hodnoty v daném intervalu adres, aby si ji programátor nemusel sám složitě vytvářet.

Řazení se spouští symbolem „`%`“. Pokud jsou ve chvíli, kdy dojde k volání funkce, na zásobníku alespoň dvě hodnoty, jsou tyto použity jako hranice intervalu pro řazení. V opačném případě je vypsáno varování (viz oddíl 4.2.3) na standardní chybový výstup a interpretace pokračuje u další instrukce.

Pokud by chtěl programátor seřadit hodnoty sestupně, může po funkci *sort* použít zásobník pro inverzi seřazeného intervalu.

Exit instrukce

Interpretace prograu v Brainfucku může skončit pouze dvěma způsoby — dosažením konce zdrojového souboru nebo pádem interpretu. Cleverer Brainfuck umožňuje třetí možnost — exit instrukci, která okamžitě uvolní všechny alokované zdroje a ukončí interpretaci programu. Symbol této instrukce je „`^`“.

instrukce	význam
`(`	pokud je hodnota v aktuální buňce nenulová, proveď blok; jinak skoč na ukončovací závorku
`)`	ukončovací závorka — konec jednoho **IF** bloku

Tabulka 3.9: Konstrukce IF.

instrukce	význam
`(...)`	řídicí konstrukce **IF**
`@(...)`	řídicí konstrukce **IF NOT**
`(...)@(...)`	řídicí konstrukce **IF-ELSE**

Tabulka 3.10: Možnosti použití řídicí konstrukce IF a negace.

3.2.7 Formální definice rozšíření

Pro další postup v projektu (implementace interpretu) bylo třeba rozšíření formálně definovat. Autor k tomu použil LL gramatiku.

Informace, jak tuto gramatiku sestavit autor čerpal z knihy *Handbook of Formal Languages*[4]. Z důvodu velkého množství terminálů (z jazykového pohledu lexémy, kterými jsou podle expanzních pravidel gramatiky nahrazovány neterminální symboly) a řídicích instrukcí je pravidel značné množství a gramatika je tudíž velmi rozsáhlá (viz příloha A.1).

instrukce	význam
`"Hello World!"`	vytiskne řetězec
`@"Hello world!"`	uloží řetězec jako posloupnost ordinálních hodnot zakončenou buňkou hodnotou 0; původní hodnoty buněk jsou ztraceny, pozice datového ukazatele zachována
`;`	vytiskni řetězec — postupně tiskne všechny znaky uložené na pásce, dokud nenarazí na ukončovací nulu (konec řetězce)
`:`	vytiskni hodnotu v aktuální buňce jako číslo v desítkové soustavě

Tabulka 3.11: Nové instrukce pro práci s řetězci, tisk čísel.

```
$ = začáček bloku pro definice procedur

= definice procedury, která vytiskne znak a volá
= sama sebe, dokud nenarazí na konec řetězce
r {.>(r)}

= definice procedury, která pouze volá následující procedury
a {hsw}

= definice trojice procedur volané procedurou 'a'
h {"Hello"}
s {" "}
w {"world!\n"}

= redefinice procedury 'a'
= pokud zůstane nazakomentováno, způsobí SYNTAX ERROR
a {hsw}

$ = konec bloku pro definice procedur

= uložení řetězce do paměti
@"Hello world!\n "

= volání rekurzivní procedury pro tisk řetězce
r

= volání procedury, která tiskne Hello wolrd!
= voláním dalších procedur
a
```

Obrázek 3.5: Definice a volání procedur v jazyku Cleverer Brainfuck

```
"XFIALA47";
```

Obrázek 3.6: Uložení a tisk loginu autora (`XFIALA47`) v jazyku CBF

Kapitola 4

Interpret pro Cleverer Brainfuck

Po definici jazyka CBF pomocí bezkontextové gramatiky bylo třeba vyvinout interpret, pomocí kterého je možné programy napsané v tomto jazyku spouštět. Proto autor navrhnul a poté v jazyku C implementovval interpret `cbfrun`.

Tento interpret přijímá zdrojové texty s příponou `.cbf`. Interpret je case-sensitive[1].

4.1 Implementace

Vzhledem k tomu, že jazyk CBF obsahuje velmé množství instrukcí, komplikované řídicí konstrukce a definice vlastních procedur, není možné, narozdíl od původního Brainfucku, implementovat jeho interpret jednoduchým konečným automatem.

Z tohoto důvodu byl autor nucen vydat se standardní cestou[1] pro implementaci interpretu, resp. kompilátoru[2]. To znamená implementaci čtyř jasně daných vzájemně spolupracujících modulů:

- lexikální analyzátor (angl. *scanner*);
- syntaktický analyzátor (angl. *parser*);
- generátor vnitřního kódu;
- interpret vnitřního kódu.

Lexikální analyzátor (LA, viz oddíl 4.1.1) přijímá jako vstup zdrojový text jazyka CBF a prostřednictvím lexikální analýzy popsané v následujícím oddílu v něm identifikuje lexémy[3], které ve formě seznamu tokenů předává syntaktickému analyzátoru (SA, viz oddíl 4.1.2).

SA pomocí prediktivního algoritmu využívajícím LL tabulku kontroluje syntaktickou správnost zdrojového kódu, tedy správnost pořadí lexémů.

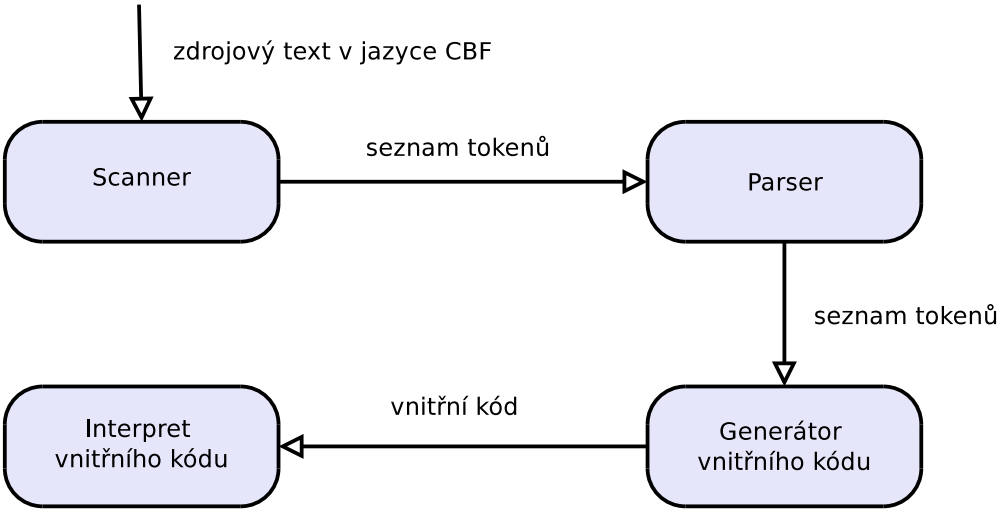
[1] Case-sensitivity znamená rozlišování malých a velkých písmen abecedy.

[2] Kompilátor (překladač) a interpret si jsou implementačně velmi podobné. Rozdíl je až v poslední části zpracování zdrojového textu — tam, kde u interpretu nastává samotná interpretace vnitřního kódu, kompilátor provádí konverzi vnitřního kódu na cílový kód, většinou binární.

[3] Lexém je základní lexikální (jazyková) jednotka, v kontextu formálních jazyků se může jednat např. o operátor, identifikátor, literál atd.

Pokud tyto dvě fáze analýzy zdrojového textu proběhnou v pořádku, je seznam tokenů předán[4] generátoru vnitřního kódu (viz 4.1.3), který na jeho základě vytvoří abstraktní datovou strukturu označovanou jako tzv. vnitřní kód.

Tato struktura je pak předána interpretu vnitřního kódu (viz. 4.1.4), který provede samotnou interpretaci instrukcí ve zdrojovém souboru. Schéma celého procesu interpretace je na obr. 4.1.



Obrázek 4.1: Schéma spolupráce jednotlivých modulů interpretu jazyka CBF.

4.1.1 Lexikální analyzátor

Jak již bylo zmíněno, vstupem lexikálního analyzátoru (LA) je zdrojový soubor jazyka CBF. LA, který je implementovaný jako konečný automat, načítá jednotlivé znaky zdrojového souboru a testuje, zda-li jejich pořadí odpovídá některému z lexémů.

Test každého znaku může skončit jednou z následujících možností:

- začátek nového lexému;
- pokračování lexému;
- konec lexému — úspěšně zpracování lexému;
- lexikální chyba způsobená načtením neočekáváného znaku.

Po každém úspěšném zpracování lexému je tento přidán ve formě tokenu na konec seznamu tokenů, který je výstupem LA. V případě lexikální chyby je navrácen chybový kód reprezentující lexikální chybu. Tento chybový stav vylučuje další správné zpracování a zdrojového souboru a jeho následnou interpretaci.

[4]Obvykle parser vytváří abstraktní syntaktický strom (ASS), ovšem jazyk CBF není natolik komplikovaný, aby tento mezikrok potřeboval a generátor vnitřního kódu může zpracováváat přímo seznam tokenů.

4.1.2 Syntaktický analyzátor

Syntaktický analyzátor (SA) přijímá seznam tokenů a testuje, zda-li jejich pořadí odpovídá gramatice jazyka. SA interpretu jazyka CBF používá syntaktickou analýzu shora dolů, která využívá prediktivního algoritmu.

Prediktivní algoritmus, v tomto případě implementovaný zásobníkovým automatem, má několik vstupů:

- seznam tokenů získaného ze zdrojového souboru;
- seznam expanzních pravidel LL gramatiky;
- LL tabulku, která slouží k výběru expanzního pravidla.

Seznam tokenů získá parser od scanneru, seznam expanzních pravidel gramatiky jazyka autor definoval v kapitole XXX. LL tabulku sloužící k výběru pravidel bylo třeba vytvořit na základě pravidel LL gramatiky. LL tabulku generovanou z gramatiky jazyka CBF naleznete v přílohách (viz A.2).

V případě, že syntaktická analýza skončí chybou, je navrácen chybový kód upřesňující typ syntaktické chyby a stejně jako v prápadě chyby lexikální je interpretace programu ukončena.

4.1.3 Generátor vnitřního kódu

Vstupem generátoru kódu je syntakticky správný seznam tokenů sestavený ze zdrojového souboru. Generátor vnitřního kódu tento seznam načítá po jednotlivých tokenech a vytváří abstraktní datovou strukturu zvanou vnitřní kód. Obecně se jedná o jednosměrně svázaný lineární seznam prvků reprezentujících instrukce.

Každý prvek obsahuje kód instrukce, případně její parametr (např. délku krátkého skoku, dekadickou hodnotu pro uložení do buňky, ...). Teprve v této fázi dochází k interpretaci rozpoznaných lexémů, do této doby se pracovalo pouze s jejich typy.

Na konec vnitřního kódu je implicitně umístěna terminální instrukce.

4.1.4 Interpret vnitřního kódu

Interpret vnitřního kódu prochází od začátku seznam instrukcí (vnitřní kód) a provádí je, dokud nenarazí na terminální podmínku, nebo nedojde k neočekávané chybě, případně varování se zapnutým přepínačem `-pedantic` (více viz oddíl 4.3).

4.2 Uživatelské rozhraní

Interpret `cbfrun` je implementován jako konzolová aplikace. Zdrojový kód v jazyku CBF je interpretu předán jménem souboru (včetně přípony `.cbf`) na pozici prvního agrumentu. Příklad spuštění interpretace souboru *hello.cbf* viz obr. 4.2.

4.2.1 Vstup a výstup interpretovaného programu

Jazyk CBF umožňuje čtení i tisk znaků prostřednictvím standardního vstupu (`stdin`) a výstupu (`stdout`). Tyto datové toky lze přesměrovat jako u všech ostatních konzolových aplikací.

```
$ cbfrun hello.cbf
```

Obrázek 4.2: Spuštění interpretace zdrojového souboru *hello.cbf*.

Chybová hlášení (viz oddíl 4.2.2) a varování (viz oddíl 4.2.3) jsou stejně jako případný výstup debuggeru (viz kapitola 5) tisknuty na standardní chybový výstup (`stderr`).

4.2.2 Chyby

Chyba je kritickou situací, které nastala při běhu interpretu. Pokud nastane chyba, není možné v interpretaci zdrojového kódu pokračovat. Interpret `cbfrun` rozlišuje několik základních typů chyb:

- chyba v argumentech intrepretu (neplatný přepínač, neexistující zdrojový soubor, ...);
- lexikální chyba (neplatný znak při načítání lexému);
- syntaktická chyba (neplatné pořadí tokenů, redefinice procedury, ...);
- interní chyba interpretu (selhání alokace paměti, ...).

V případě vzniku chyby je ukončena interpretace, na stadardní chybový výstup je odeslán popis chyby a interpret při ukončení vrátí kód chyby. Kódy jednotlivých chyb naleznete v tabulce 4.1.

4.2.3 Varování

Varování je způsob, kterým dává interpret užvateli najevo, že pravděpodobně nastala neplánovaná situace (např. použití hodnoty z vrcholu prázdného zásobníku, ...).

Varování jsou stejně jako chyby odesílány na stadardní chybový výstup, ale nezpůsobují ukončení interpretace programu. Pomocí přepínače `-pedantic` lze však toto defaultní chování změnit. Pro tento případ má i každé varování svůj kód, který je interpretem navrácen, pokud je s varováním nakládáno jako s chybou. Z tohoto důvodu začínají kódy varování hodntou 42, aby nechocházelo k jejich záměně za kódy chyb.

Seznam všech varování i s jejich kódy je v tabulce 4.2.

4.3 Přepínače interpretu

Interpret `cbfrun` rozpoznává několik přepínačů, které modifikují jeho defaultní chování, viz tabulka 4.3.

Až na výjimku prvního přepínače pro nápovědu (`--help` nebo `-help`), který lze použít jako první a jediný přepínač, musí všechny ostatní následovat za jménem interpretovaného zdrojového `.cbf` souboru. Žádný z přepínačů nelze použít vícekrát.

Už z podstaty funkce přepínačů `-pedantic` a `-suppress-warnings` vyplývá, že je nelze použít společně.

kód chyby	význam
1	neplatný počet argumentů při spuštění intepretu (příliš mnoho přepínačů)
2	neplatný přepínač intepretu
3	násobné použití jednoho či více přepínačů
4	nedovolená kombinace přepínačů interpretu
5	`--help` a `-h` smí být použito pouze samostatně
6	neplatná nebo chybějící přípona zdrojového souboru
7	neexistující zdrojový soubor (nebo nedostatečná práva ke čtení)
8	interní chyba — neúspěšný pokus o alokaci paměti
9	lexikální chyba — neočekávaný znak
10	lexikální chyba — neplatný znak
11	lexikální chyba — EOF před koncem řetězce
12	lexikální chyba — literál obsahuje neplatnou číslici
13	literál obsahuje příliš velké číslo
14	syntaktická chyba — chybné pořadí tokenů
15	redefinice procedury
16	volání nedefinované procedury
17	dělení nulou

Tabulka 4.1: Chybové kódy interpretu jazyka CBF.

kód varování	význam
42	dlouhý skok s prázdným zásobníkem (neexistuje cíl skoku)
43	volání instukce *pop* nebo *top* na prázdný zásobník
44	pokus o použití hodnoty ze zásobníku (např. aritmetickou operací) v okamžiku, kdy je zásobník prázdný
45	volání instrukce *remove top* s prázdným zásobníkem
46	pokus o inkrementaci nebo dekrementaci hodnoty na zásobníku v okamžiku, kdy je zásobník prázdný
47	volání fuknce *sort* s méně než dvěma prvky na zásobníku
48	volání fuknce *sort* nad zatím nepoužitou částí paměti (pásky)

Tabulka 4.2: Kódy varování interpretu jazyka CBF.

přepínač	funkce
`--help` nebo `-h`	vytiskne nápovědu pro použití interpretu
`-pedantic`	varování — stejně jako chyby — způsobí ukončení interpretace programu
`-suppress-warnings`	potlačí zobrazení varování
`-debug`	debug mód (viz kapitola 5)

Tabulka 4.3: Přepínače interpretu `cbfrun`

Kapitola 5

Debugger pro Cleverer Brainfuck

5.1 Motivace

Přestože rozšíření popsaná v sekci 3.2 výrazně zlepšují orientaci v kódu a snižují riziko vzniku chyb, stejtě jako při programování v jiných jazycích, ani v CBF se programátor chybám zcela vyhnout nemůže. Jejich hledání je často velmi pracné, vede ke snížené efektivitě při používání jazyka a k frustraci uživatele — programátora.

Z důvodů v předchozím odstavci autor, navíc inspirován několika on-line debuggery pro původní jazyk Brainfuck, v rámci této práce rovněž implementoval debugger pro CBF, který poskytuje základní funcionalitu pro krokování programu, analýzu použité paměti a případné zásahy do ní.

5.2 Implementace

Debugger CBF je modulem interpretu `cbfrun` a také používá terminál jako uživatelské rozhraní.

Z toho vyplývá, že veškerý vstup i výstup debuggeru je pouze v textové, resp. pseudografické podobě. Jedná se sice o jisté omezení, které má negativní vliv především na názornost zobrazených dat, ovšem pro potřeby debuggingu v CBF je tato forma uživatelského rozhraní dostačující.

Vývoj případné grafické nadstavby uživatelského rozhraní pro debugger autor navrhuje v oddílu 6.3.

5.3 Použití debbugeru

Jako zarážka (angl. *breakpoint*) se v jazyku CBF používá svislá čára „|“. Tento symbol lze umístit na libovolná místa ve zdrojovém souboru (kromě prostoru mimo definici procedur v bloku pro definice procedur — zarážka je na takovém místě zbytečná, protože by při interpretaci programu nikdy nedošlo k jejímu zpracování).

Tyto zarážky jsou defaultně neaktivní a jsou zcela ignorovány. Pro aktivaci debug modulu interpretu a zpracování zarážek je nutné použít přepínač `-debug` (viz oddíl 4.3).

Použitím přepínače `-debug` se interpret spustí v debug módu, ve kterém při zpracování každé zarážky vypíše informaci na výstup (pro debugger je výstupem `stderr`). V tu chvíli je interpretace přerušena a čeká se na příkaz od programátora. Seznam všech příkazů naleznete v následujícím oddílu 5.4.

5.4 Příkazy

Uživateli jsou k dispozici základní příkazy pro analýzu a manipulaci paměti a řízení běhu programu, viz tabulka 5.1.

příkaz	popis funkce
`continue`	skok na další zarážku v programu; v případě skoku z poslední zarážky ukončí interpret i debugger
`terminate`	okamžité ukončení interpretu i debuggeru
`memory`	zobrazí aktuální stav paměti v pseudografickém režimu
`stack`	zobrazí aktuální stav zásobníku v pseudografickém režimu
`set address to X`	nastavení datového ukazatele na danou adresu X; X je desítková hodnota v rozmezí `0-UINT_MAX`
`set cell X to Y`	nastaví hodnotu v buňce s adresou X na Y; X a Y jsou desítkové hodnoty v rozmezí `0-UINT_MAX`
`set stacktop to X`	nastavení hodnoty prvku na vrcholu zásobníku na hodnotu X; X je desítková hodnota v rozmezí `0-UINT_MAX`
`push X`	vložení prvku s hodnotou X na vrchol zásobníku; X je desítková hodnota v rozmezí `0-UINT_MAX`
`pop`	odstranění prvku z vrcholu zásobníku, jeho hodnota se nikam neukládá
`help`	výpis nápovědy pro debugger

Tabulka 5.1: Uživatelské příkazy pro debbugger interpretu jazyka CBF

Kapitola 6

Závěr

6.1 Splnění cílů práce

Autor v této práci hledal a analyzoval největší nedostatky návrhu jazyka Brainfuck v kontextu použitelnosti a udržovatelnosti. Poté pro nalezené nedostatky navrhnul řešení v podobě rozšíření původního jazyka. Toto rozšíření pojmenoval **Cleverer Brainfuck (CBF)**.

6.2 Požadovaná rozšíření jazyka Brainfuck

Po analýze jazyka Brainfuck se autor zaměřil na návrh požadovaných rozšíření — byli jimi tyto:

- řídicí konstrukce **IF-ELSE**, viz oddíl 3.2.3;
- podpora počítaných cyklů **FOR**, viz oddíl 3.2.3;
- definice vlastních procedur s podporou rekurzivního volání, viz oddíl 3.2.4;
- podpora práce s řetězci, viz oddíl 3.2.5.

Kromě požadovaných rozšíření autor v rámci řešení nedostatků návrhu přidal i další vylepšení původního programovacího jazyka:

- definice nejasně vymezených vlastností původního jazyka a redefinice špatně navrhnutých vlastností původního jazyka, viz oddíl 3.1;
- přidání instrukcí umožnujících efektivnější práci s hodnotami v buňkách (přímé vkládání hodnot) a datovým ukazatelem (reset, krátké a dlouhé skoky), viz oddíl 3.2.1;
- podpora abstraktního datového typu zásobník, viz oddíl 3.2.2;
- symbol pro negaci, použitý pro **ELSE** větěv v konstrukci **IF-ELSE** lze použít pro modifikaci původního cyklu **WHILE** na cyklus **WHILE NOT**, viz oddíl 3.2.3;
- rozšířené možnosti tisku obsahu buněk na výstup (zejména tisk číselných hodnot), viz oddíl 3.2.5;
- přidání vestavěné funkce *sort*, viz oddíl 3.2.6.

Po dokončení návrhu rozšíření CBF jej autor formálně popsal bezkontextovou gramatikou (viz příloha A.1). Pro tuto gramatiku sestavil LL parsovací tabulku, kterou používá při implementaci syntaktického analyzátoru interpretu nově navrhnutého jazyka.

V další fázi autor navrhul a implementoval jako součást interpretu i debugger, který umožňuje snazší analýzu programů v CBF a rychlejší hledání a odstraňování chyb.

Na závěr autor implementoval 10 programů v jazyku CBF (viz příloha A.4), které demonstrují jeho možnosti v porovnání s původním jazykem Brainfuck. Tyto programy naleznete společně se zdrojovými texty interpretu (viz příloha A.3) na přiloženém CD .

6.3 Možnosti dalšího vývoje

Přestože je rozšíření CBF o mnoho použitelnější než původní minimalistický Brainfuck, lze i tento nový jazyk dále rozšiřovat o další možnosti, případně upravovat stávající vlastnosti:

- další instrukce pro práci s hodnotami — např. výměna dvou hodnot;
- aritmetické operace pro dva operandy bez využit zásobníku;
- podpora libovolně dlouhých identifikátorů procedur;
- podpora návratových hodnot podprogramů, tedy funkcí;
- podpora multithreadingu;
- optimalizace vnitřního kódu (eliminace mrtvého kódu, kontextové generování kódu);
- grafické uživatelské rozhraní (GUI) pro debugger.

6.4 Shrnutí

Přestože nebyl jazyk Brainfuck navržen pro reálné použití, mnoho programátorů na celém světě si jej velmi oblíbilo a začalo v něm psát programy menšího rozsahu.

Autor v této práci navrhnul řešení pro největší nedostatky v použitelnosti tohoto jazyka, toto rozšíření formálně popsal a implementoval pro něj interpret a debugger. Hlavními výhodami nového jazyka CBF v porovnání s původním Brainfuckem jsou:

- čitelnost kódu (na první pohled je vidět programátorův záměr);
- udržovatelnost kódu (zejména díky možnosti definice vlastních procedur);
- použitelnost jazyka (všechny standardně používané řídidích konstrukce, větší množství instrukcí);
- větší efektivita (není nutné používat velké množství primitivních separátně prováděných instrukcí);
- výrazně kratší zdrojové texty programů v CBF, které jsou ekvivalentní s programy v Brainfucku;
- jasně definované chování za všech podmínek.

Literatura

[1] Aho, A.; Ullman, J.; Lam, M.; aj.: *Compilers: Principles, Techniques, and Tools*. Pearson Education, druhé vydání, 2006, iSBN: 0-321-48681-1.

[2] Böhm, C.: On a family of Turing machines and the related programming languages. *ICC Bulletin*, , č. 3, July 1964: s. 185–194.

[3] Feeney, S.: Brainfuck. Duben 2005.
URL `<http://esolangs.org/wiki/Brainfuck>`

[4] Rosenberg, G.; Saloma, A.: *Handbook of Formal Languages*. Springer, 1997, iSBN: 3-540-60649-1.

Příloha A

Obsah CD

A.1 LL gramatika jazyka CBF

V adresáři *grammar* se nachází definice jazyka Cleverer Brainfuck ve formě LL gramatiky.

A.2 LL parsovací tabulka

Pro implementaci sytaktické analýzy s použitím prediktivního algoritmu bylo třeba sestavit parsovací LL tabulku, která slouží k výběru expanzních pravidel.

Tato tabulka je umístěna v adresáři *ll_table*.

A.3 Interpret `cbfrun`

V adresáři *interpret* se nachází okomentované zdrojové texty interpretu `cbfrun` v jazyku C.

Překlad pomocí GCC je možné spustit příkazem `make`. Použití interpretu je popsáno v oddílu 4.2. Interpret je testován v prostředí Windows XP, Vista, 7 a 8.1, Mac OS a Linux Mint 16 a 17 a na fakultním serveru *merlin* (CentOS).

A.4 Demo programy

Adresář *demos* obsahuje 10 programů implementovaných v nově navrženém jazyku Cleverer Brainfuck.

Všechny programy jsou okomentované a je možné je spustit interpretem `cbfrun`.