Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé DP a prohlašuji, že **s o u h l a s í m** s případným užitím mé diplomové práce (prodej, zapůjčení apod.).

Jsem si vědom(a) toho, že užít své diplomové práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Diplomovou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Poděkování

Na tomto místě bych chtěl poděkovat RNDr. Ladislavu Mečířovi, za odborné vedení, pomoc při zpracování diplomové práce a poskytnuté informace. Dále chci poděkovat Doc. Ing. Janu Cvejnovi, Ph.D. a RNDr. Kláře Císařové za podporu při tvorbě diplomové práce. Děkuji všem ostatním, kteří mi poskytli pomoc po dobu studia.

Anotace

Diplomová práce se zabývá implementací interpretu programovacího jazyka REBOL. REBOL představil jeho tvůrce Carl Sassenrath v roce 1997. Dnes je to moderní interpret s velkým množstvím funkcí. Uživatelé volí tento jazyk kvůli jeho jednoduchosti, velikosti interpretu a množství funkcí, které nabízí. Vývoj na tomto programu je však z velké části uzavřený a kompletní zdrojový kód není k dispozici. Nepřekvapí nás tedy, že snaha dát světu implementaci, která bude mít otevřený kód, je velká. Práce ukazuje způsob, jakým lze tohoto cíle dosáhnout. Neklade si za cíl kompletní funkční interpret, ale nápaditou implementaci některých jeho částí. Interpret jazyka REBOL byl naprogramován v jazyce C. Pozornost se obrací především na práci s daty v jazycích s dynamickými typy dat a způsoby analýzy textu. Snaha je o kompaktní program, který poskytuje vysoký výkon a nízkou redundanci kódu. Využívá prostředí GNU, které zaručuje jak vysokou přenositelnost, tak řadu komponent, které jsou pro vývoj nezbytné. Primárně je program vyvíjen v operačním systému GNU Linux, čímž však program na přenositelnosti neztrácí.

Abstract

Diploma thesis engages in implementation of an interpreter of REBOL programming language. REBOL was introduced in 1997 by its creator, Carl Sassenrath. Nowadays, it is considered a modern interpreter which has numerous functions. Users are using it due to its simplicity, interpreter dimension and various powerful functions. However, major part of the languages development is closed to public; therefore, complete code is not at disposal. It is clear, that there is a great endeavour to provide a free source implementation. My work shows a way how to reach this goal. The aim is not to provide a working interpreter, but a resourceful implementation of some of its parts. REBOL language interpreter was programmed in C language. The main stress is put on data operations in languages with dynamical types of data and ways of text analyses. The outcome is a compact programme which provides good performance as well as low code redundancy. The programme uses GNU environment that provides excellent portability and a number of essential components. GNU Linux is the primary operating system used for the development, but the programme is nowise restricted by this fact.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 0. Úvod

Programovcí jazyk REBOL[1] představil jeho tvůrce Carl Sassenrath, návrhář operačních systémů, mezi jehož úspěchy patří například *Amiga Multitasking Operating System* uvolněný roku 1985. První implementace REBOLU byla dána k testování malé skupině uživatelů roku 1997. Tato první verze podporovala 3 platformy. Ke dnešnímu dni REBOL používá více jak milión uživatelů na více než 40 platformách.

V současné době je interpret složen z několika přímo podporovaných balíčků.

REBOL/Core
: Jádro interpretu. Ke komunikaci je požit příkazový řádek a souborový systém.

REBOL/View
: Grafické rozhraní. Využívá dialektu jazyka REBOL k popisu vzhledu.

REBOL/Command
: Rozšíření pro jazyk. Možnost přístupu k databázím, rozšíření pro web, zvuky a další.

REBOL/SDK
: Pro vývojáře, kteří vyžadují větší kontrolu nad funkcí a nastavením svých programů.

REBOL/IOS
: Vzdálený operační systém.

V dalším textu se budu věnovat pouze jádru interpretu.

REBOL/Core je interpret vyššího programovacího jazyka. Zkratka REBOL v originále znamená *Relative Expression Based Object Language*. Kompaktní a malé jádro přichází s velkým množstvím inovativních postupů a vlastností.

Mým úkolem je částečně implementovat interpret jazyka REBOL v programovacím jazyce C [2]. Výsledek by měl ukazovat možnosti implementace takového interpretu. Může také sloužit jako základ kompletního interpretu jazyka REBOL s otevřeným kódem. Nejvíce úsilí je věnováno práci s daty v jazycích s dynamickými datovými typy a analýze textu, s využitím již existujících funkcí jazyka REBOL. Práce řeší otázku přenostelnosti a volby vývojového prostředí.

Vzhledem k faktu, že vzorový interpret jazyka REBOL není projekt s otevřeným kódem, bude nutno použít metody *zpětného inženýrství*[2]. Dalším zdrojem informací bude manuál jazyka [13].

---

[1] © 2005 REBOL Technologies.

[2] Z anglického *reverse engineering*. Způsob jak dosáhnout představy o principech mechanismu pomocí analýzy jeho struktury a funkcí. Více například tento článek: [1].

Kapitola 1. Analýza

Předem bych chtěl upozornit na to, že s ohledem na inovativnost jazyka REBOL, je možné, že význam některých pojmů, vykládaných dále v textu, se může lišit od jiných programovacích jazyků.

Informace o analýze interpretu a jeho chování jsou ve většině případů čerpány nebo jsou přímo citací z manuálu a dokumentace jazyka REBOL [13] a článků z webové encyklopedie *wikipedia* [16].

1.1. Slovník

Pravopis slova, anglicky *spelling*, je způsob, kterým vytvoříme slovo z písmen zvolené abecedy tak, že respektujeme jejich pořadí. Pro příklad, spelling slova `zbytek` je řetězec "zbytek". *Identifikátor slova* je číselná hodnota, která vznikne překladem *pravopisu slova*.

Slovo je jednoznačně určeno svým *pravopisem*, *identifikátorem* a *kontextem*[1], do kterého patří. Musí existovat způsob, jak novému slovu přidělíme nový *identifikátor*, stejně jako k existujícímu slovu nalezneme již existující. Překlad *pravopisu* na *identifikátor* je řešen pomocí překladové tabulky. Překlad *pravopisu slova* na *identifikátor* nám přinese především výkonový rozdíl, protože je výhodnější porovnávat dvojici čísel, než dva řetězce. Systém by měl obsahovat alespoň jednu tuto tabulku, kterou nazýváme globální slovník.

1.2. Kontext

Kontext, v interpretru *context*, je seznam slov. *Kontext* může být tabulka, jejíž řádek obsahuje *identifikátor slova* a datové pole, které představuje hodnotu, kterou slovo popisuje. Musíme umět určit, zda tabulka obsahuje hodnotu s daným *identifikátorem* a pro nový *identifikátor* založit řádek s výchozí hodnotou slova.

Předem bych chtěl upozornit na to, že co je v části Analýza (kapitola 1) myšleno pod pojmem *kontext*, je v části Implementace (kapitola 2) nazýváno jmenný prostor.

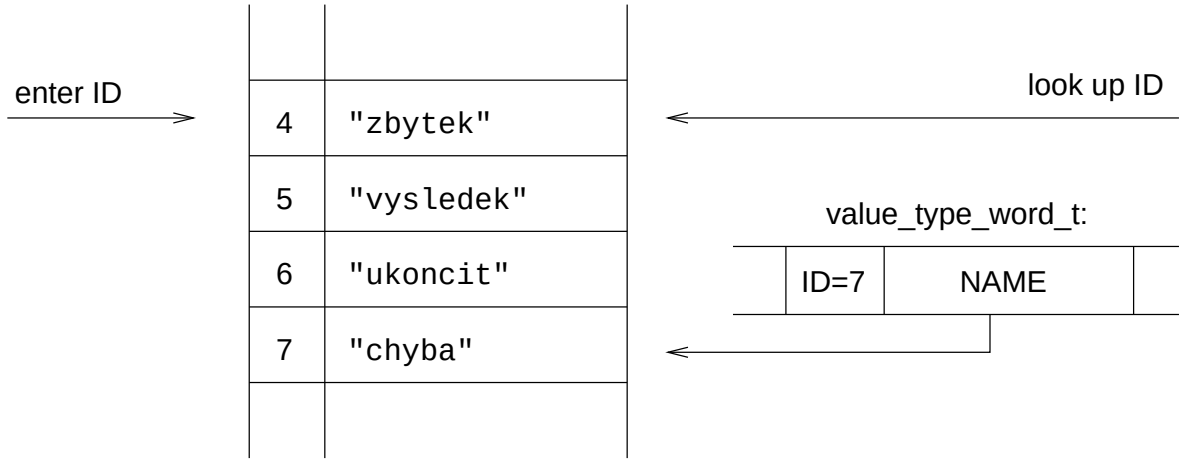
1.3. Interpretace

Interpretace kódu jazyka REBOL se skládá ze tří fází. Pro každou tuto fázi existuje v interpretu nativní funkce, která provádí odpovídající činnost.

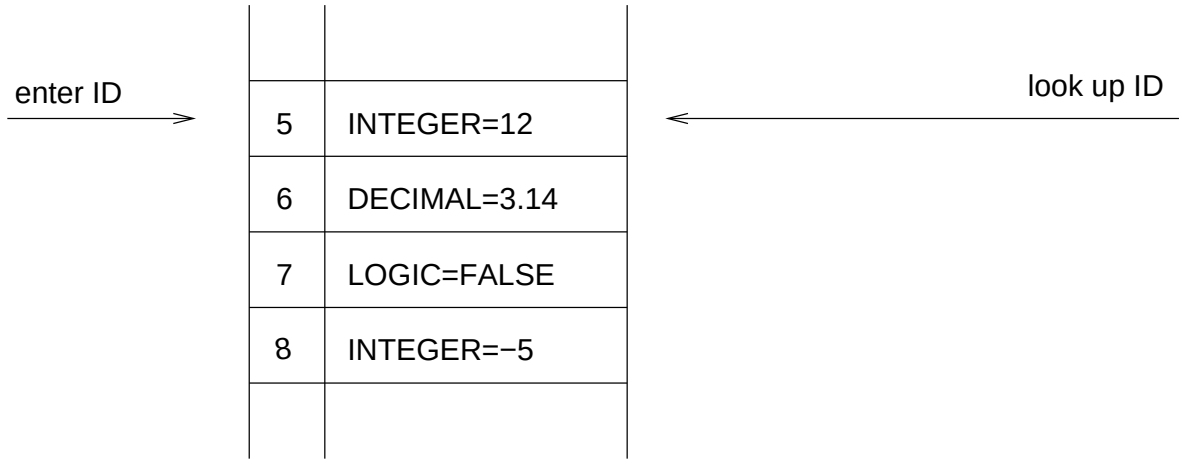
Fáze make

V této fázi interpret vytváří datový typ *block*. Blok je výsledkem práce funkce parse, které se jako parametr předá vstupní řetězec a blok, obsahující pravidla pro jeho parso-

[1] Definice pojmu *kontext* je provedena v sekci 1.2.



Obrázek 1.1. Představa uspořádání paměti slovníku.



Obrázek 1.2. Představa uspořádání paměti pro kontext

vání. Funkce rozpoznává jednotlivé datatypy REBOLu, vytváří je, odpovídajícím způsobem nastavuje jejich hodnoty a vkládá je do nového bloku. Žádné slovo neobsahuje informaci o *kontextu.*

Fáze load

Fáze má za úkol spojit slova, která blok obsahuje, s místem v paměti, kde bude nebo již je uložena jejich hodnota. Dělá to tak pro všechny datové typy odvozené od pseudotypu *any-word* s výjimkou typu *refinement*, který nenese žádnou další hodnotu. Interpret se v globálním *kontextu* pokusí najít *identifikátor* slova, který obdržel při požadavku uložení slova do globálního slovníku. Pokud ho najde, opraví referenci slova tak, že bude ukazovat na nalezené místo. Pokud nenajde, vytvoří nový záznam s tímto identifikátorem a výchozí hodnotou a referenci na toto místo zapíše do slova.

Fáze do

Jak bude dále podrobněji vysvětleno, není podstatné, jakým způsobem byl vytvořen blok, který se stal argumentem funkce *do.* Blok musí pouze nutně projít fází *load*, aby

byl interpretován tak, jak uživatel očekává.

Funkce zpracovává hodnoty uložené v bloku jednu po druhé. Pro některé datové typy nepředstavuje zpracování hodnoty žádnou akci, tedy jejím výsledkem je nezměněná hodnota. Vzhledem k tomu, že budou jednotlivé akce nad hodnotami do podrobna probrány v sekci Implementace (kapitola 1), nebudu se tím zabývat na tomto místě.

Všechny tyto zásady platí i pro datový typ *paren*. Více informací se lze dozvědět v manuálu jazyka REBOL [13] a v článku [9] nebo [4].

1.4. Parsing

Parsing je mechanismus, jak analyzovat data. Využívá se tam, kde je nutné interpretovat jazyk nebo data, která podléhají definovaným pravidlům za účelem dalšího zpracování.

Funkce *parse* jazyka REBOL je jednou z jeho nejsilnějších vlastností. Nahrazuje tradiční *regulární výrazy* [15] a v jistých ohledech jejich vlastnosti převyšuje. Umožňuje vytvořit řadu nových jazyků, zvaných *dialekty*. Současný interpret jazyka REBOL jich pro různé účely několik používá.

Syntax funkce *parse* je popsána takto.

```
PARSE input rules /all /case
```

Argument *input* musí být typu *series*. Argument *rules* pak jedním z typů *block*, *string* nebo *none*. Pokud je argument *rules* typu *string* nebo *none*, funkce pracuje v režimu, který jsem nazval **analýza pomocí oddělovačů**. Režim funkce v případě, že je argument *rules* typu *block*, jsem nazval **analýza podle pravidel**. Činnost funkce dále upravují příznaky zvané *refinements*. Refinement *case* zajistí porovnání znaků abecedy s přihlédnutím na to, zda jde o malé nebo velké písmeno. Bez jeho použití nebere na velikost písmen ohled. Refinement *all* vypíná automatické přeskakování tak zvaných bílých míst[1].

V režimu **analýza pomocí oddělovačů** funkce hledá ve vstupu *input* začátek a délku *tokenu*[2]. Jednotlivé tokeny jsou od sebe odděleny určenou sadou znaků. Rozpoznané tokeny vkládá po jednom do datového typu *block*, který je před návratem z funkce vrácen jako výsledek. V manuálu jádra REBOLu [13, sekce 15: Parsing] jsou popisovány dvě sady možných oddělovačů. Označil jsem je jako *měkké oddělovače* a *tvrdé oddělovače*. Jejich definice následuje. Použití typů oddělovačů v obou režimech je patrné z tabulky číslo 1.1.

V režimu **analýza podle pravidel** argument *rules* popisuje vstupní řetězec *input*. Pokud vstup pravidlům vyhoví, funkce vrátí datový typ *logic* s hodnotou true. V opačném případě je výstupem datový typ *logic* s hodnotou false. Rámec pravidel umožňuje provádět akce.

Jazyk je v zásadě rozdělen na přímé porovnávání typů a volání funkcí, kde je k rozhodnutí, zda-li vstup vyhovuje, potřeba složitější akce. Stručný přehled funkcí jazyka použitého pro pravidla poskytuje tabulka číslo 1.2.

---

[1] Z anglického *white spaces*.

[2] Slovo nebo atomický element v rámci řetězce.

	none	*string*	**typ argumentu**
-	S, H*	S, A	
all	H	A	
refinement			

S měkké oddělovače: mezera, tabulátor, nový řádek

H tvrdé oddělovače: čárka, středník

A *rules*: argument funkce

***** řetězec nulové délky mezi **S** a **H** je považován za token

Tabulka 1.1. Použití typů oddělovačů v jednotlivých režimech funkce parse

General Forms	|	alternate rule
	[block]	sub-rule
	(paren)	evaluate a REBOL expression
Specifying Quantity	none	match nothing
	opt	zero or one time
	some	one or more times
	any	zero or more times
	12	repeat pattern 12 times
	1 12	repeat pattern 1 to 12 times
	0 12	repeat pattern 0 to 12 times
Skipping Values	skip	skip a value (or multiple if repeat given)
	to	advance input to a value or datatype
	thru	advance input thru a value or datatype
Getting Values	set	set the next value to a variable
	copy	copy the next match sequence to a variable
Using Words	word	look-up value of a word
	word:	mark the current input series position
	:word	set the current input series position
	'word	matches the word literally (parse block)
Value Matches (block parsing only)	fred	matches the string fred
	%data	matches the file name %data
	10:30	matches the time 10:30
	1.2.3	matches the tuple 1.2.3
Datatype Words	type!	matches anything of a given datatype

Tabulka 1.2. Stručný přehled jazyka použitého pro pravidla funkce parse

Kompletní dokumentaci funkce *parse* najdeme v manuálu interpretu REBOL [13, sekce 15: Parsing]. Další informace lze čerpat například z těchto webových stránek [14].

Kapitola 2. Implementace

Zdrojový kód interpretu je zapsán ve vyšším programovacím jazyku C. Vývoj probíhal na operačním systému GNU/Linux. Jako překladač jsem použil GCC, tedy GNU Compiler Collection, pro jeho snadnou dostupnost a dobré vlastnosti. Konkrétní distribucí byl Slackware® Linux, přeložený pro architekturu i386.

Jedním z cílů bylo vyvinout interpret, který bude možné přeložit na více platformách. Problém přenostitelnosti je složen z přenositelnosti zdrojového kódu intepretu a přenositelnosti systémů, které jsou pro překlad nezbytné.

Překlad pro jinou architekturu je podmíněný existencí překladače GCC pro tuto platformu. Pro systémy typu UNIX, na kterých GNU dozrála, je tato podmínka splněna. Nutno podotknout, že do této kategorie dnes již patří i oblíbený operační systém Mac OS X firmy Apple Computer. Pro Platformu Windows firmy Microsoft je k dispozici prostředí MinGW [10], které představuje minimalistickou implementaci GNU prostředí pro tento operační systém.

Přenostitelnost jednotlivých použitých systémů je popsána pro každý systém zvlášť v následujícím textu.

2.1. Použité systémy a knihovny

Při návrhu interpretu bylo použito několik hotových systémů, ať již knihoven nebo nástrojů, které řeší specifický úkol. Bylo by velice pracné je implementovat ve vlastní režii a konečně jejich vývoj nebyl předmětem zadání. Všechny jsou volně dostupné a uvolněné pod některou z licencí *svobodného software.* V mém případě již byly některé přímo součástí operačního systému. Jedním z cílů bylo vyvinout interpret, který bude možno přeložit na více platformách.

2.1.1. Projekt GNU

Než začnu popisovat konkrétní systémy, rád bych vysvětlil pojmy použité dále v textu.

GNU je projekt, jehož kořeny sahají do roku 1983. V této době nebyl k dispozici použitelný operační systém, který by splňoval následující pravidla, *svobodného software.*

(1) Spouštět program za jakýmkoliv účelem.

(2) Studovat, jak program pracuje a přizpůsobit ho svým potřebám. Předpokladem k tomu je přístup ke zdrojovému kódu.

(3) Redistribuovat kopie dle svobodné vůle.

(4) Vylepšovat program a zveřejňovat zlepšení, aby z nich mohla mít prospěch celá komunita. Předpokladem je opět přístup ke zdrojovému kódu.

Cílem tedy bylo vytvořit klon operačního systému UNIX, který by byl šířen jako *svobodný software*. Vzorem se staly systémy UNIX System V. Projekt založil Richard Stallman[1], který má dnes klíčovou roli ve vývoji jádra operačního systému Linux a je jedním z hlavních protagonistů modelu *svobodného software*. Pro nový systém dlouhou dobu neexistovalo jádro. Problém vyřešil Linus Torvalds, který pro tyto účely poskytnul svůj Linux. Projekt GNU pro svůj software ustanovil několik licencí, z nichž nejznamější je *The GNU General Public License*, pod kterou je vydáno například jádro systému – Linux. Slovo GNU je rekurzivní zkratka pro "GNU's Not Unix". Projekt GNU je zaštítěn Nadací pro svobodný software (Free Software Foundation, Inc.). Více informací lze nalézt na stránkách [8].

2.1.2. Boehm-Demers-Weiser conservative garbage collector

Garbage collector je obecně systém, který je alternativou ke stadardní dynamické správě paměti.

Knihovna je vysoce portabilní a v době psaní tohoto textu je známo, že je schopná provozu na následujících platformách: Linux, *BSD, aktuální verze Windows, MacOS X, HP/UX, Solaris, Tru64, Irix a několik dalších.

Sestavení vyžaduje knihovnu *libpthread*, která implementuje operace nad thready. Na systému GNU/Linux® je také nutno připojit knihovnu *libdl*, která obsahuje volání *dlopen* a implementuje dynamické linkování knihoven a relokaci kódu vynuceně pomocí volání.

Následuje popis systému.

Lze jím nahradit knihovní funkce *malloc*, *realloc* a *free*. Jeho výhodou je skutečnost, že se aplikace sama nemusí starat o uvolňování paměti, která jí byla přidělena funkcí *GC_MALLOC*. Systém zaměstnává kód (zvaný "garbage collecting routine"), který si drží informace o rozložení, obsahu a typu adresového prostoru procesu a je schopen poznat, zda je daný úsek paměti dostupný nebo je možné jej uvolnit. Knihovna může pracovat ve dvou režimech.

Neinkrementální režim

Do tohoto režimu je knihovna nastavena, pokud se nepoužije volání *GC_enable_incremetal*. Režim používá metodu zvanou *stop-the-world*, která spočívá v pozastavení všech threadů v systému,[2] prozkoumání registrů, provedení velké většiny cyklu "garbage collectingu" a opětovného spuštění všech threadů.

Knihovna je v interpretu používána v tomto režimu.

Inkrementální režim

Do tohoto režimu se musí knihovna přepnout voláním *GC_enable_incremetal*. V tomto režimu se v každém cyklu provede jen malý kus práce a řízení se vrátí opět aplikaci. Dosahuje se tím lepší odezvy procesu, přestože může být do jisté míry více neoptimální z důvodu konzumace více strojového času. To proto, že může dojít ke změně situace, kterou se snaží knihovna v několika bězích popsat a v dalším cyklu

[1] http://www.stallman.org/

[2] Toto není jednoduchý úkol, protože knihovna používá některou z variant knihovny *libptread*, která je v podstatě jedinou přenositelnou knihovnou implementovanou do takové míry, aby vůbec mohla být použita. Na každé z architektur se zastavení threadů dosahuje jiným, skoro bez výjimky nepřenositelným způsobem. Vyvarovat se musí také možnosti vytvoření nového threadu.

potřebuje čas na nové hledání.

Knihovna používá tak zvaného *modifikovaného mark-sweep* algoritmu. V principu se dělí na čtyři fáze:

Preparation

Vynuluje příznak dostupnosti na všech objektech.

Mark phase

Označí všechny objekty, které jsou přístupné přes ukazatele (i přes vícenásobné odkazy). Za normálních okolností nemá knihovna žádné informace, kde v adresovém prostoru mohou ukazatele ležet. Proto hledá ve všech statických datových oblastech, hromadě, zásobníku i registrech procesorů. Způsob jak předat knihovně informaci o rozložení adresového prostoru, především hromady a tím jí usnadnit práci a omezit tak strojový čas, knihovnou spotřebovaný, lze pomocí rozhraní *typed*.

Sweep phase

Hledá v hromadě objekty, nedostupné od té doby, co byly neoznačeny a řadí je do seznamu objektů k uvolnění. Tato část probíhá odděleně pouze v inkrementálním režimu, když se dostane na dno seznamu objektů k uvolnění.

Finalization phase

Nedostupné objekty, které dříve zaregistrovali svoji proceduru pro ukončení jsou zařazeny do fronty pro ukončení.

Zajímavou stránkou činnosti je především fáze označování, tedy *mark phase*. Ve zjednodušení ji lze rozdělit do následujících kroků:

(1) Ukazatel je testován, zda se trefí do hrubých hranic hromady. Všechny objekty hromady tak musí do těchto hranic padnout. Většina objektů neprojde tímto testem.

(2) Adresa ukazatel je rozdělena na dvě části. Na část s více významnými bity, která udává číslo stránky (na všech achitekturách se velikost stránky rovná 2^n) a část s méně významnými bity, která udává offset do stránky.

(3) Zkusí vyhledat číslo stránky v tabulce (dvouúrovňová stromová struktura pro rychlé vyhledávání ukazatelů), která může obsahovat buď 0 nebo malé číslo N nebo nebo ukazatel na popisovač stránky. V případě, kdy je nalezena 0 víme, že stránka není zahrnuta do paměti určené ke "garbage collectingu". Malé číslo N znamená, že stránka je součástí velkého objektu, začínajícího o N stránek zpět. I v tomto případě je knihovna dopočítat ukazatel na popisovač stránky, který obsahuje začátek objektu.

(4) Je spočítána adresa začátku objektu. Popisovač stránky obsahuje velikosti objektů v této stránce, typy objektů a důležité příznaky pro označení. Velikost objektu je použita ke zjištění správného začátku objektu. Ke zrychlení tohoto procesu hlavička stránky obsahuje ukazatel na předpočítanou mapu offsetů náhradou za počátek objektu. Toto je optimalizace proti potenciálně pomalé operaci odčítání.

(5) Příznak pro označení se otestuje a označí. V případě, že byl příznak neoznačený, uloží se

na zásobník označených objektů.

(6) Příznaky pro označení všech zbylých objektů se vynulují a pro případ, že algoritmus označil nesprávný objekt.

Slovo *konzervativní* ve spojení "conservative garbage collector" znamená, že systém má jen částečné informace o umístění ukazatelů a proto je nucen se chovat ke všem bitovým typům, jako by to byly ukazatele. Více informací se lze dozvědět v materiálu [3].

V principu se algoritmus chová jako tak zvaný "mostly parallel garbage collector". Tento termín se vztahuje k problematice, která řeší, které části kódu "garbage collectoru" běží paralelně ve vlastním threadu a také na jak dlouho a jak často přeruší činnost aplikace. Hlavní změna proti "mostly parallel garbage collectoru" spočívá ve skutečnosti, že kód "boehm-demers-weiserova garbage collectoru" běží ve threadu klienta, který žádá o přidělení paměti. Neexistuje zde žádný oddělený thread "garbage collectoru". Tvůrci se snažili o jednoduchou přenositelnost na architektury, kde není možné provozovat více jak jeden thread. Více informací se lze dozvědět v dokumentu [11].

Knihovnu lze úspěšně použít i jako *memory leak detektor*. Detekování paměťových úniků se používá pro lokalizování částí kódu, který je chybně napsaný a vyznačuje se tím, ze paměť od systému požaduje, ale již ji neuvolňuje. Tato knihovna ke zjištění tohoto typu chyby používá podobných technik, jako při "garbage collection".

Existují dva doporučované způsoby, jak knihovnu používat. V obou případech je nutné k binárnímu souboru připojit knihovnu pojmenovanaou jako *libgc*.

(1) Aplikace již je v notném stadiu rozpracovanosti a bylo by velmi pracné měnit stávající kód. Používají se pouze direktivy překladače (a to konkrétně `-DREDIRECT_MALLOC=GC_malloc -DIGNORE_FREE`), která způsobí, že veškerá volání funkce *malloc* se přesměrují na funkci *GC_MALLOC*, která je součástí knihovny *libgc* a všechna volání funkce *free* se neprovedou.

(2) Aplikaci vyvíjíme se zaintegrovaným prostředím knihovny a používáme přímo její aplikační rozhraní, které je popsáno níže. V tomto případě stačí jen připojit požadovanou knihovnu. Není nutné používat funkci na uvolňování paměti.

Aplikační rozhraní se skládá z následujících volání překladače jazyka C:

`void * GC_malloc(size_t nbytes)`
Přidělí a vynuluje *nbytes* bytů paměti. Spotřebuje čas úměrný *nbytes*. Vrácený blok paměti bude automaticky uvolněn v okamžiku, kdy stane nedostupným. *GC_MALLOC* je makro jazyka C, které v případě, kdy je nastaven *GC_DEBUG*, zapne testovací kód. V obou případech však volá *GC_malloc*.

`void * GC_malloc_atomic(size_t nbytes)`
V principu funkce *GC_malloc*. Rozdíl je pouze ten, že přidělí paměť, o které volající slíbil, že se v ní nebudou vyskytovat ukazatele. Paměť není nulována. Toto je doporučovaný způsob, jak získat paměť pro řetězce, bitmapy, binární data a podobné.

`void * GC_malloc_uncollectable(size_t nbytes)`
V principu funkce *GC_malloc*. Rozdíl je pouze v tom, že přidělená paměť bude vždy

prohledávána na řetězce, protože bude vždy považována za dostupnou.

`void * GC_realloc(void *old, size_t new_size)`
: Přidělí nový objekt *new_size* veliký a zkopíruje původní na počátek nového. Zde má knihovna prostor pro optimalizaci použitím stránkovacích technik nebo použitím původního místa v případě, že je za ním dostatečně velký prostor. Vlastnosti oblasti paměti zůstávají nezměněny ve srovnání s původní.

`void GC_free(void *dead)`
: Uvolní přidělenou paměť. Nepoužívá se na malé objekty a nedoporučuje se volat často.

`void * GC_malloc_ignore_off_page(size_t nbytes)`
: V principu *GC_malloc* a *GC_malloc_atomic*. Volající garantuje, že dokud je objekt používán, bude udržován ukazatel na nějaké místo v rámci jeho prvních 512 bytů. Tento ukazatel by měl být deklarován jako *volatile* aby se předešlo problémům s optimalizačními technikami překladače. Toto je doporučovaný způsob, jak získat paměť pro objekty větší, jak 100kB. Významně snižuje pravděpodobnost neuvolnění objektu v případě, že již není používán.

`void * GC_malloc_atomic_ignore_off_page(size_t nbytes)`
: Analogické ke *GC_malloc_ignore_off_page*.

`void GC_gcollect(void)`
: Vynutí proběhnutí celého cyklu "garbage collection".

`void GC_enable_incremental(void)`
: Vynutí proběhnutí jedné části cyklu každých několik volání funkce *GC_malloc* místo toho, aby proběhnul celý cyklus najednou. Spotřebuje více výkonu, ale zlepší odezvu na platformách, které mají odpovídající podporu v knihovně.

`GC_warn_proc GC_set_warn_proc(GC_warn_proc p)`
: Dovolí nastavit funkci, která vypisuje varování knihovny. Původní funkce zapisuje do *stderr* nejčastěji v případech, kdy by použití *GC_malloc_ignore_off_page* bylo vhodnější.

`void GC_register_finalizer(...)`
: Zaregistruje funkci, která bude volána v okamžiku, kdy se objekt stane nepřístupným. Toto je často jediný rozumný způsob, jak dosáhnout uvolnění systémových prostředků, které jsou nějakým způsobem spjaty s objektem (například zavírání souborů). Tato metoda není vhodná pro operace, které se mají provést neprodleně. Ve správně napsaném kódu by se mělo toto volání vyskytovat co nejméně.

Existuje i rozhraní pro jazyk C++, to však pro tento projekt nemá význam, proto se jím zde zabývat nebudu. Je možnost použít část knihovny zvanou *cords*, která optimalizuje přístupy a práci se řetězcemi. Konkrétně pak spojování řetězců a práce s dlouhými řetězci. Jednou z posledních zajímavých částí je rozhraní, nazývané *typed*, pomocí kterého jsme schopni komunikovat do knihovny typový popis oblasti paměti. Výhodou je, že se zvětší výkon v případech, že takto označíme paměť, o které by si knihovna původně myslela, že se v ní mohou vykytovat ukazatele a mohla

by se jí snažit prohledávat, na druhou stranu docílíme bezchybnosti algoritmu v případě, kdy se snažíme uchovávat ukazatele v paměti, o které jsme slíbili, že ukazatele obsahovat nebude (získané například pomocí *GC_malloc_atomic*).

Více informací o celém systému lze získat na stránkách [5].

2.1.3. GNU autoconf

GNU Autoconf je nástroj pro generování skriptů jazyka UNIX shell, které mají za úkol automaticky nastavit prostředí pro překlad zdrojového kódu balíčků aplikací. Tento úkon je nezbytný z toho důvodu, že každý systém typu UNIX nemusí obsahovat stejné komponenty, verze nebo dokonce implementace systémů nutných pro sestavení aplikace. Autoconf poskytuje metody, jak zjistit, zda-li je daná komponenta přítomna, je-li přítomna ve správné verzi a pokusí se přeložit příklad. Autorovi aplikace je dána možnost umístit doplňující shell skript kód pro nastalé výjimky. Uživateli je pak nabídnut přehledný systém voleb, kterými může ovlivnit, které z volitelných částí má výsledek obsahovat, případně jaké vlastnosti má mít. Výsledný kód, který je výstupem z Autoconf, není na sobě samém závislý, tedy běží i na systému, který instalaci Autoconf postrádá. Běh těchto skriptů při konfiguraci nevyžaduje uživatelovu interakci a v ideálním případě by měly být sestavené tak, aby nebylo potřeba žádné další volby.

Autoconf se obvykle používá společně s dalšímy nástroji. GNU Automake, popsaný v následujícím odstavci, generuje přenositelné předpisy pro GNU Make, zvané *Makefile*. Make v tomto předpise najde kompletní informace o cílech, způsobu překladu a jeho nastavení. Autoconf pro svůj běh vyžaduje tradiční UNIX makro processor s názvem *GNU m4*.

Další text bude věnován konfiguraci nástroje Autoconf použitém v této práci. Soubor *configure.in* obsahuje následující.

```
AC_PREREQ([2.57])
AC_INIT([src/main.c])
AM_INIT_AUTOMAKE([rbl], [0.2])
AM_CONFIG_HEADER([include/config.h])
```

V této části definujeme, pro které nejnižší verzi systému Autoconf je tato konfigurace určena a kde má výsledný skript hledat zdrojový kód. Pro Automake je tu nastavené jméno aplikace, jeho verze a kam má uložit vygenerovaný hlavičkový soubor jazyka C, který bude obsahovat konfiguraci aplikace.

```
AC_PROG_CC
```

Zkontroluje dostupnost překladače a jeho nezbytných součástí.

```
AC_CHECK_LIB([dl], [dlopen])
AC_CHECK_LIB([gc], [GC_malloc], , [echo "*** libgc not found." && exit 1],
  [-pthread])
```

Kontroluje dostupnost vyžadovaných knihoven. První argument je jméno knihovny, druhý je jméno volání, které se použije při pokusu přeložit příklad. Třetí, volitelný, je akce, která se má provést v případě, že test selže. Důležitý je příkaz *exit* jazyka shell, který způsobý okamžité zastavení konfigurace. Dělá tak z této knihovny komponentu povinnou. Posledním, čtvrtým

argumentem je seznam knihoven, nezbytných pro přeložení testovacího programu.

```
AC_STDC_HEADERS
AC_CHECK_HEADERS([assert.h errno.h getopt.h search.h stdio.h stdlib.h string.h])
```

Zjistí, zda-li jsou dostupné hlavičkové soubory standardních knihoven a otestuje přítomnost knihoven volitelných.

```
AC_C_CONST
```

Ověří předpokládané vlastnosti překladače.

```
AC_CHECK_FUNCS([strdup])
```

Ujistí se, že jsou přítomny všechny užité funkce.

```
AC_OUTPUT([Makefile
 src/Makefile])
```

Zapíše výsledek do tohoto předpisu pro nástroj Make.

2.1.4. GNU automake

GNU Automake je nástroj, který automaticky generuje předlohu pro *Makefile*, kterou dále využívá GNU Autoconf. Nástroj z relativně jednoduché konfigurace, která je běžně uložena v souboru *Makefile.am*, vygeneruje běžné cíle pro nástroj *GNU Make*. Mezi ně patří nejen cíl pro sestavení celé aplikace, ale také promazání stromu od produktů překladu, generování archivu balíku nebo pravidla pro jeho instalaci. Automake vyžaduje programovací jazyk *perl*, o kterém se lze dozvědět více informací v sekci dokumentace na této internetové adrese: [12].

Tento odstavec věnuji popisu použitých konfigurací nástroje Automake. V kořenovém adresáři projektu najdeme následující.

```
SUBDIRS = src
```

Ukazuje na všechny podřízené adresáře, které obsahují samostatnou konfiguraci.

```
DISTCLEANFILES = *~ stamp-h* include/stamp-h*
MAINTAINERCLEANFILES = *~ stamp-h* include/stamp-h*
```

Tyto definice přidávají k cílům, sloužícím pro úklid stromu zdrojového kódu, další soubory.

```
EXTRA_DIST = include/*.h
```

Pokud zvolíme cíl pro generování archivu k distribuci, tato definice zaručí, že bude obsahovat i všechny hlavičkové soubory.

Adresář *src* obsahuje tuto konfiguraci.

```
bin_PROGRAMS = rbl
```

Definuje cíle, které budou obnovovány. V tomto případě je to spustitelný sobor *rbl*.

```
rbl_SOURCES = dt.c error.c exec.c jump_table.c main.c ns.c parse.c rebol.c
test.c value.c value_function.c value_number.c value_other.c value_serie.c
value_word.c
```

Definuje základní zdroje pro cíl *rbl*, které mohou podlehnout změně. Seznam dalších souborů, na kterých jsou tyto závislé, vygeneruje překladač na základě odkazů v uvedených zdrojích. Za tuto práci je však zodpovědný nástroj GNU Autoconf.

```
AM_CFLAGS = -g -Wall -DREDIRECT_MALLOC=GC_malloc -DIGNORE_FREE
AM_LDFLAGS = -pthread -L /usr/local/lib/pth -L /usr/lib/pth
rbl_LDADD = ../r/data.o
```

První řádek definuje seznam doplňujících argumentů pro překladač, druhý pak seznam doplňujících argumentů pro linker. Poslední řádek definuje dodatečné knihovny, které mají být přidány do cíle *rbl*.

2.1.5. GNU make

GNU Automake automaticky rozezná, které části aplikce je nutné na základě změny zdroje obnovit a obnovu provede. Veškeré informace o své činnosti hledá v konfiguraci aplikace, která popisuje rozložení zdrojového kódu, výsledné části aplikace a způsob, jak z něj tento výsledek vyrobit. Konfigurace je standardně uložená v souboru, který se jmenuje *Makefile*. Jsou složené z jednotlivých cílů aplikace a k němu je definován seznam příkazů, které je nutno provést, aby se cíl obnovil. Posouzení nutnosti cíl obnovit se děje na základě času změny zdrojového a cílového souboru. Pokud je čas změny na zdroji menší než na cíly, je nutné cíl obnovit.

Make umí spustit více úloh paralelně, pomáhá tedy využít většího množství oddělených výpočetních jednotek. Použití tohoto nástroje není omezeno na oblast programování, ale lze ho využít kdekoliv, kde je nutná automatická aktualizace souborů.

Pro větší projekty je ruční udržování konfigurace velice pracné. Další problém, který Make neřeší, je přenositelnost konfigurace. Proto je v dnešní době pro kompletní správu aplikace doplňován o další GNU nástroje, především GNU Automake a GNU Autoconf.

2.1.6. GNU objcopy

Gnu objcopy je součást balíku *GNU binutils*, který obsahuje řadu nástrojů pro vývoj aplikací pro systém GNU a je standardně instalovaný v rámci vývojového prostředí operačního systému Linux. Kopíruje obsah vstupního souboru na soubor výstupní a nad přenášenými daty provádí požadované operace. Program přistupuje k mnoha různým formátům dat a proto používá knihovnu GNU BFD, která s nimi umožňuje pracovat. Primární účel programu je kopírování objektu pro linker[1] z jednoho formátu do formátu jiného. Jedna z dalších funkcí, kterou využívá i tento projekt, je vytvoření objektu z obecného souboru dat. V rámci tohoto módu program vygeneruje řadu symbolů, které popisují binární objekt, ukládaný do výstupu. V mém případě tuto možnost využívám při vkládání zdrojového kódu rebolu do spustitelného souboru interpretu.

Pro doplnění představy o fázích překladu překladače GNU GCC připojuji zjednodušené schéma.

[1] Sestavuje výsledný spustitelný soubor z více souborů, které jsou výsledkem předchozí fáze kompilace.

Preprocess: .c → .c Fázi provádí preprocessor jazyka c, v GCC nazvaný *cpp*. Do zdrojových kódů se vkládají hlavičkové soubory a rozvíjejí se makra.

Compile: .c → .s Tato fáze zaměstnává překladač jazyka c, nazvaný *cc1*. Výsledkem překladu je zdrojový kód pro GNU assembler.

Compile: .s → .o Překlad jazyka GNU assembler je skrytý v souboru jménem *as*. Výsledkem je binarní kód pro zvolenou architekturu uložený v některém z formátů knihovny BFD. Tento soubor je nazýván objektem pro linker.

Link: .o → out Konečnou fázi překladu má na starosti tak zvnaý linker, pojmenovaný *ld*, který ze vstupních objektů sestaví spustitelný soubor.

2.2. Datové typy

Datové typy jsou jednou z nejdůležitějších částí REBOLU. Na jejich optimálním navržení je závislá spotřeba paměti a celkový výkon interpretu. Hodnoty jazyka REBOL lze rozdělit na část pohotovostní a na část dodatkovou. Pohotovostní část musí splňovat následující podmínky:

(1) pro všechny datové typy musí mít konstantní velikost

(2) musí obsahovat identifikátor, který jednoznačně identifikuje typ

Měla by také obsahovat co nejdelší část hodnoty, případně hodnotu celou, dodatková část pak bude obsahovat zbytek hodnoty. První podmínka zaručuje snadné indexování, protože offset N-tého uzlu dat bude vždy *N.S*, kde *S* je velikost pohotovostní části hodnoty. Volba velikosti byla kompromisem mezi výkonem a spotřebou paměti. Současný interpret používá 16 bytů. Druhá podmínka byla řešena zavedením proměnné o velikosti 1 byte, která má v rámci struktury ve všech případech offset 0. Její obsah je definován následující výčtem jazyka C.

```
enum datatype_t {
  VT_ANY_TYPE       = 0x00,
  VT_UNSET          = 0x00,

  VT_NONE           = 0x01,
  VT_DATATYPE       = 0x02,

  VT_LOGIC          = 0x03,
  VT_CHAR           = 0x04,
  VT_BITSET         = 0x05,

  VT_NUMBER         = 0x08,
    VT_INTEGER      = 0x09,
    VT_DECIMAL      = 0x0A,

  VT_ANY_FUNCTION   = 0x10,
```

```
    VT_NATIVE       = 0x11,
    VT_FUNCTION     = 0x12,

  VT_ANY_WORD       = 0x20,
    VT_WORD         = 0x21,
    VT_SET_WORD     = 0x22,
    VT_GET_WORD     = 0x23,
    VT_REFINEMENT   = 0x24,

  VT_ANY_BLOCK      = 0x40,
    VT_BLOCK        = 0x41,
    VT_PAREN        = 0x42,
    VT_PATH         = 0x43,

  VT_ANY_STRING     = 0x80,
    VT_STRING       = 0x81,
    VT_FILE         = 0x82,
    VT_BINARY       = 0x83,
};
```

Hodnota 0x00 je vyhrazena pro pseudotyp, který zahrnuje všechny vyjmenované typy. Používá se také k detekci nedefinovaných proměnných. Rozsah 0x08 až 0x0F je rezervován pro číselné typy. Jim je nadřazený pseudotyp VT_NUMBER. Datovým typům, chovajícím se jako spustitelné bloky hodnot se specifikací rozhraní, je nadřazen pseudotyp VT_ANY_FUNCTION a spadají do rozsahu 0x10 až 0x1F. Následuje rozsah 0x20 až 0x2F, jehož nadřazený pseudotyp je VT_ANY_WORD a zahrnuje hodnoty, které jsou odvozeny od slova. Následují dvě skupiny, které obě mají charakter vektoru a mají nadřazený pseudotyp VT_SERIES. První je umístěna do rozsahu 0x40 až 0x4F a nadřazenému pseudotypu říkám VT_ANY_BLOCK. Poslední skupina má rozsah 0x80 až 0x8F a je odvozená od řetězce. Zbylé datové typy jsou obecného charakteru a nespadají do žádné jiné skupiny, než VT_ANY_TYPE.

2.2.1. None

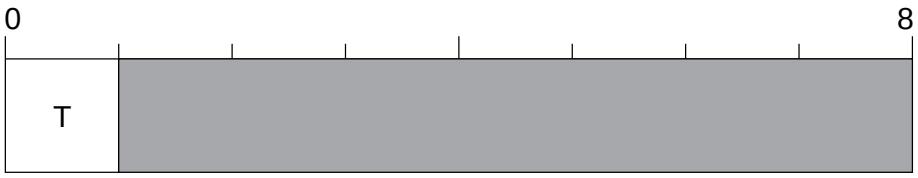
Implementace datového typu *none* je triviální. Nemá žádný datový obsah, tedy je použit pouze první byte. Popisuje ho datový typ jazyka C *value_type_none_t*. Rozložení instance v paměti ukazuje obrázek 2.1.

2.2.2. Datatype

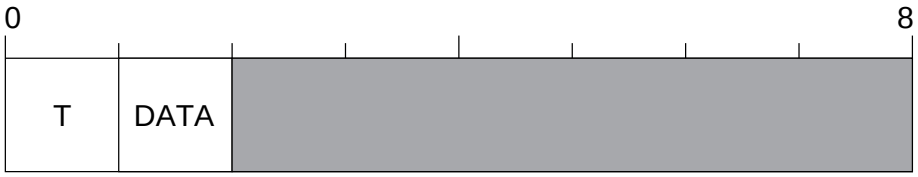
Hodnotu datového typu *datatype* tvoří identifikátor kteréhokoliv z vyjmenovaných typů. Tento má velikost jeden byte. Datový typ popisuje typ jazyka C *value_type_datatype_t* a jeho rozložení v paměti je znázorněno na obrázku 2.2.

2.2.3. Logic

Hodnota datového typu *logic* je definována výčtovým typem a je vyjádřením možných hodnot Booleovské funkce. Uložení datového typu interpretu v paměti popisuje datový typ jazyka C



Obrázek 2.1. none



Obrázek 2.2. datatype

value_type_logic_t a znázorňuje obrázek 2.3. Reálná velikost hodnoty je dána optimalizačními technikami překladače. Empirickou cestou bylo zjištěno, že se použije stejný počet bytů, jako u *int* typu jazyka C, tedy 4.

2.2.4. Char

Hodnota datového typu *char* je jeden byte s neznaménkovou aritmetikou. Popisuje ho datový typ jazyka C *value_type_char_t* a rozmístění v paměti znázorňuje obrázek 2.4.

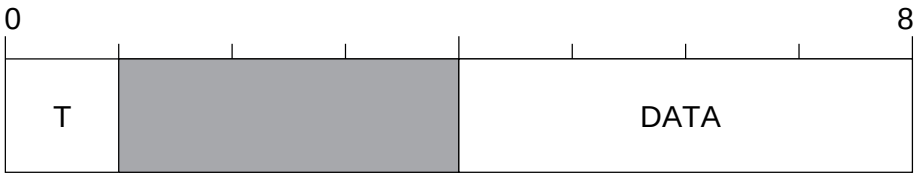
2.2.5. Bitset

Obsahem datového typu *bitset* je reference na strukturu jazyka C *stub_series_t*. Dodatková část, popsaná touto strukturou, se chová jako dodatková část datového typu *block* jazyka rebol. Její celková délka je vždy 3 hodnoty. První hodnota obsahuje slovo *make* a druhá datatyp *bitset*. Třetí prvek, který je typu *binary*, představuje neindexovatelné bitové pole dlouhé 256 bitů, tedy 32 bytů a je stěžejní hodnotou tohoto datového typu. Reference obsažená v pohotovostní části datového typu *bitset* je u zvolené *architektury*[1] dlouhá 4 byty. Uspořádání v paměti definuje datový typ jazyka C *value_type_bitset_t* a je znázorněno na obrázku 2.5. Tento datový typ využívá především funkce interpretu *parse*, která interpretuje *bitset* jako booleovskou funkci a popisuje takto výskyt všech možných znaků z používané znakové sady.

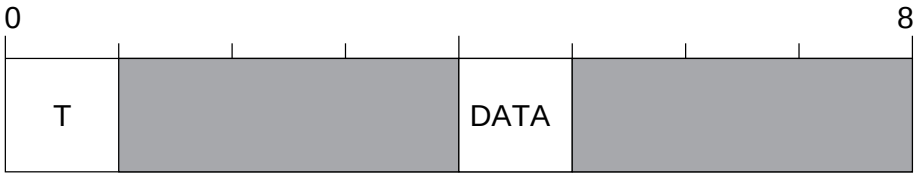
2.2.6. Integer

Číselný datový typ *integer* charakterizují 4 byty se znaménkovou aritmetikou. V jazyce C je tento datový typ interpretu definován typem *value_type_integer_t* a rozmístění v paměti znázorňuje obrázek 2.6.

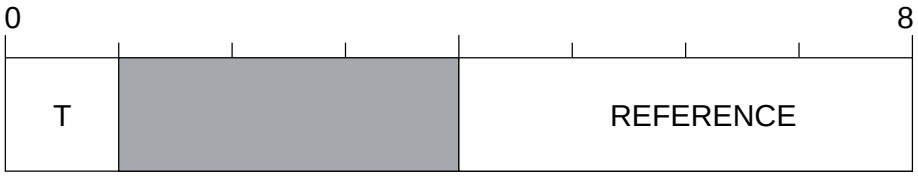
[1] Sekce 2.



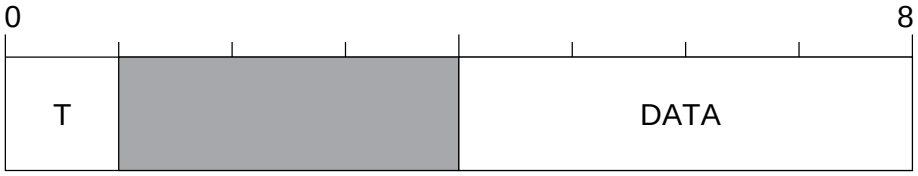
Obrázek 2.3. logic



Obrázek 2.4. char



Obrázek 2.5. bitset



Obrázek 2.6. integer

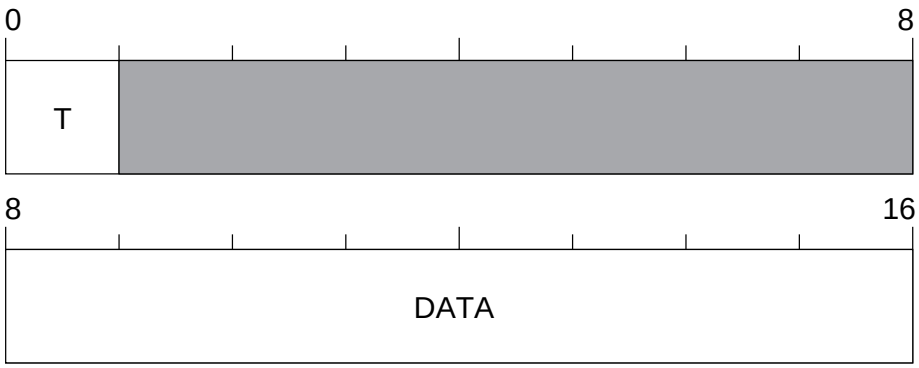
2.2.7. Decimal

Délka hodnoty datového typu *decimal* je 8 bytů a odpovídá datovému typu jazyka C *double.* Vlastnosti tohoto datového typu jsou definovány mezinárodním standardem IEEE-754[1] a ve vztahu ke zvolenému překladači jsou blíže popsány například v této publikaci [6]. Datový typ *decimal* interpretu je definován strukturou jazyka C *value_type_decimal_t* a její struktura v paměti je znázorněna na obrázku číslo 2.7.

2.2.8. Native, funkce

Datový typ *native* umožňuje v interpretu REBOLU volat funkci, která je přeložena překladačem jazyka C a je vestavěna do interpretu. Pohotovostní část hodnoty je složena z reference na specifikaci rozhraní a paměťovou referenci na vstupní bod funkce jazyka C. Specifikace roz-

[1]Standard for Binary Floating-Point Arithmetic



Obrázek 2.7. decimal

hraní je nezbytná pro správné předání parametrů a dekódování požadovaných příznaků funkce, zvaných *refinements.* Specifikace rozhraní je, stejně jako u REBOLU, realizována pomocí datového typu *block*, zřejmě bez možnosti ovlivňovat jeho parametr *offset*. Z tohoto důvodu ukazuje reference na dodatkovou část datového typu *block*. Podrobnosti o tomto typu budou vysvětleny v sekci 2.2.14. Rozložení paměťového místa typu *native* je znázorněno na obrázku číslo 2.8.

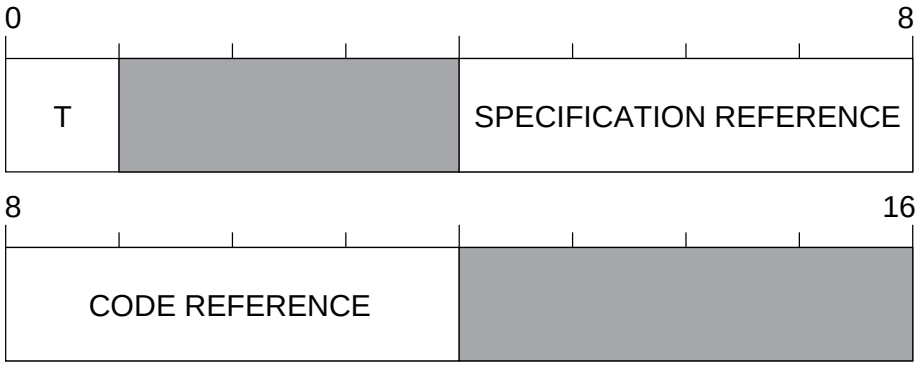
Z důvodu časové náročnosti a přehlednosti kódu jsem se snažil implementovat minimální počet takto volaných funkcí. Funkce, u kterých jsem se obešel bez možnosti přistupovat k interním strukturám interpretu, jsou implementovány v jazyce REBOL a po inicializaci interpretu jsou uloženy do datového typu *function.* Podrobnosti o něm naleznete v sekci 2.9.

2.2.9. Function

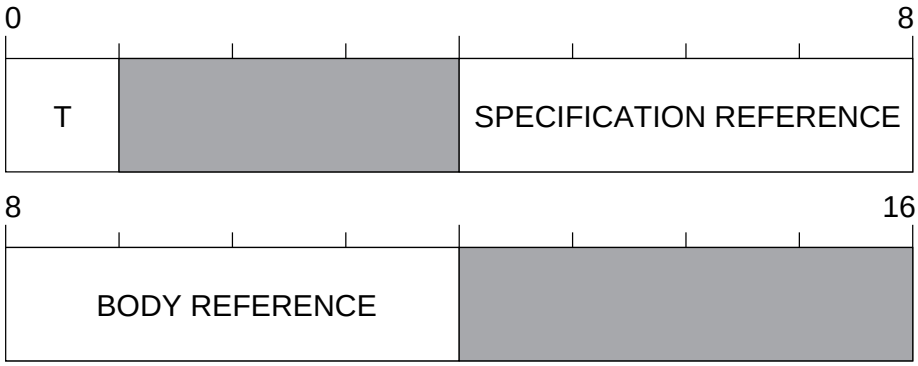
Datový typ *function* umožňuje uložit funkci definovanou v interpretu REBOLU včetně specifikace rozhraní. Vlastnosti specifikace rozhraní jsou stejné jako u typu *native.* Tělo funkce zapsané v jazyce REBOL je *block* a ani zde nemá význam nastavovat její parametr *offset*. Její součástí jsou tedy také dvě hodnoty. Obě dvě jsou referencemi na dodatkovou část typu *block*. Obrázek číslo 2.9 znázorňuje uložení v paměti.

2.2.10. Word, slova

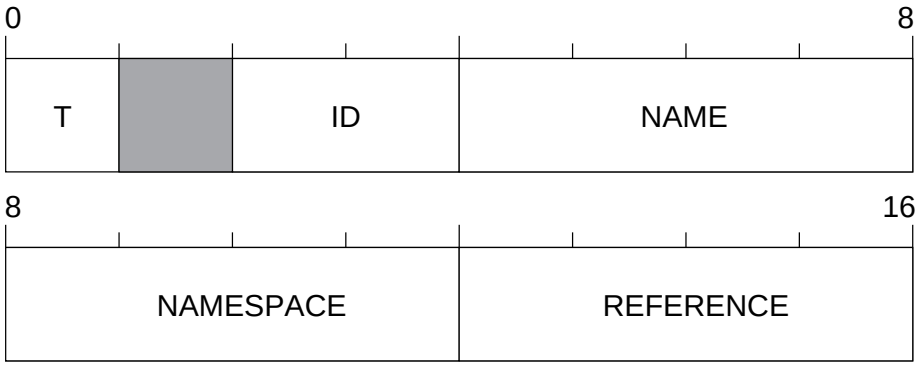
Nezbytnou součástí interpretu REBOLU je datový typ *word*, který je popsaný datovým typem jazyka C *value_type_word_t*. Plní funkci *proměnné* běžného programovacího jazyka, ale může do ní být uložena hodnota kteréhokoliv datového typu REBOLU. Příkladem může být kód v podobě bloku nebo odkaz na jiné slovo. Slovo je jednoznačně určeno pomocí *identifikátoru,* jeho *pravopisu* a jeho *jmenného prostoru.* Identifikátor je definován jako neznaménkový číselný datový typ o velikosti dvou bytů, který je ve struktuře zarovnaný na sudou adresu. Jeho hodnota je potom klíčem jak do slovníku slov, tak do jmenného prostoru. *Name,* tedy pravopis slova, je jedinou hodnotou s výjimkou identifikace datového typu, která musí být vyplněna v každém stavu slova. Nezbytnou součástí je ukazatel na *jmenný prostor*. *Reference* je ukazatel na další datový typ REBOLU, což jsou data, která jsou k danému slovu přiřazena. Všechny výše uvedené ukazatele jsou na zvolené architektuře dlouhé čtyři byty a mají adresu dělitelnou čtyřmi z důvodu optimalizace. Rozdělení paměťového místa je znázorněno na obrázku číslo 2.10.



Obrázek 2.8. native



Obrázek 2.9. function



Obrázek 2.10. word, set-word, get-word

2.2.11. Set-word

Pomocí datového typu *set-word*, který má stejné uspořádání v paměti jako *word*, můžeme přiřadit slovu hodnotu. Hodnotou tohoto datového typu je pak přiřazená hodnota.

2.2.12. Get-word

Hodnotou datového typu *get-word* je slovo, které má identifikátor, pravopis i jmenný prostor shodný s těmi hodnotami, které jsou v tomto typu uloženy. Rozložení v paměti je shodné

s datovým typem *word*. Používá se v případech, kdy je potřeba vrátit odkaz na slovo místo jeho hodnoty.

2.2.13. Refinement

Datový typ *refinement*, odvozený od typu *word*, se používá jako příznak při volání funkcí a mění jejich chování. Jedinou odlišností je nepřítomnost odkazu na hodnotu. Je definovaný typem jazyka C *value_type_refinement* a jeho uspořádání v paměti znázorňuje obrázek číslo 2.11.

2.2.14. Block, série

Všechny datové typy odvozené od *series* mají podobné vlastnosti. Jsou to datové typy s rychlým indexovým přístupem a lze u nich nastavit počátek, v REBOLU zvaný *offset*. Jednou z vlastností REBOLU je, že několik sérií lišících se *offsetem* může odkazovat na stejná data. Při modifikaci jednoho z nich se změna musí projevit ve všech ostatních. Proto je zvolena dvouúrovňová struktura znázorněná na obrázku číslo 2.12. Tyto typy obsahují pouze položku *offset*, jejíž velikost je čtyři byty a udává pozici počátku série a položku *reference*, která ukazuje na dodatkovou část slova. Dodatková část je popsána strukturou *stub_series_t*. Blok paměti přidělený k uchování dat v dodatkové části slova může být obecně větší, čímž je zabráněno příliš častému volání knihovní funkce *GC_REALLOC*. Její celková délka je šestnáct bytů a obsahuje čtyři položky o délce čtyři byty. *Memory pointer* ukazuje na přidělený paměťový blok. Jeho velikost obsahuje položka *memory length*. Na vlastní data ukazuje položka *data pointer*, o které víme, že je *data length* dlouhá. Znázornění obou datových typů je na obrázku číslo 2.12. Znázornění bloku vlastních dat je vidět na obrázku číslo 2.13.

Zde bych chtěl poukázat na nesporně čistější pojetí *series* v REBOLU, ve srovnání s ukazateli v ostatních programovacích jazycích. Jednou z hlavních výhod může být nemožnost adresace místa mimo rozsah platné paměti. To je možné proto, že mechanismy adresace v REBOLU se snaží upravit offset pole, tak aby se vešel do dovolených mezí.

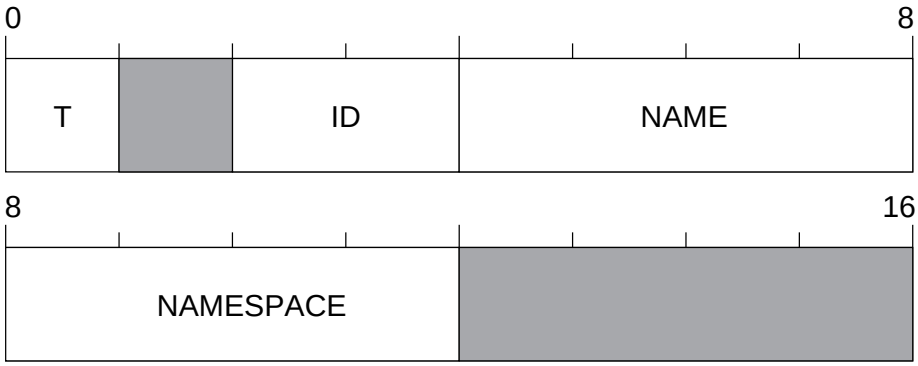
Odvozený typ *block* je charakterizovaný typem jazyka C *value_type_block_t*. Velikost jeho indexovatelné buňky je velikost pohotovostní části hodnot všech zmiňovaných datových typů, tedy 16 bytů. Jeho využití v rámci interpretu nemá hranice, neboť může obsahovat jakýkoliv jiný popisovaný typ a tento typ může být pro jednotlivé buňky různý. Obvykle se používá jako úložiště spustitelného kódu.

2.2.15. Paren

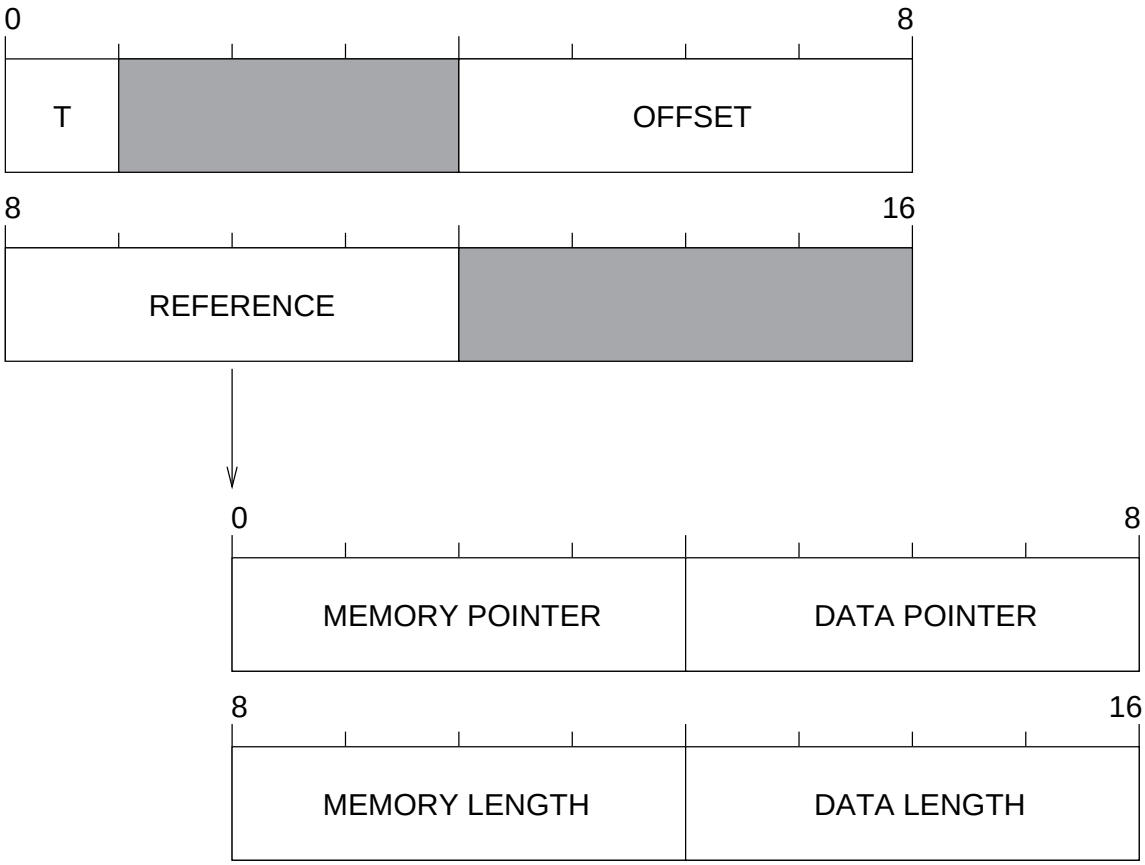
Datový typ *paren* je charakterizovaný typem jazyka C *value_type_paren_t*. Vlastnosti jsou shodné s typem *block*, liší se pouze ve způsobu interpretace. Hodnota typu *block* se při interpretaci nezmění, kdežto hodnota typu *paren* se vyhodnotí. K tomuto účelu je se také využívá.

2.2.16. Path

Datový typ *path* je charakterizovaný typem jazyka C *value_type_path_t*. Velikost jeho indexovatelné buňky je velikost pohotovostní části hodnoty *word*, tedy 16 bytů. U typů odvozených od *any-function* se používá jako způsob, jak připojit k volání jeho příznaky, tak zvané *refinements*. U typů vektorového charakteru pak jako nástroj pro přímou indexaci. Využití je více.



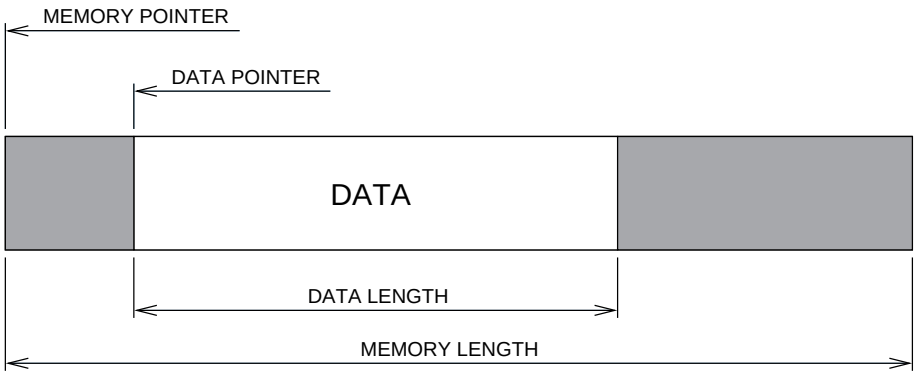
Obrázek 2.11. refinement



Obrázek 2.12. series: block, paren, path, string, file, binary

2.2.17. String

Datový typ *string*, jak už název napovídá, slouží jako řetězec. Velikost indexované buňky je tedy 1 neznaménkový byte. Je definovaný typem jazyka C *value_type_string_t*.



Obrázek 2.13. znázornění rozložení dat popisovaných *stub_series_t*

2.2.18. File

Datový typ *file* je charakterizovaný typem jazyka C *value_type_file_t*. Velikost jeho indexovatelné buňky je stejná jako u typu *string*, tedy 1 byte. Používá se pro práci se systémem souborů. Ukazuje na adresáře nebo soubory.

2.2.19. Binary

Datový typ *binary* je charakterizovaný typem jazyka C *value_type_binary_t*. Velikost jeho indexovatelné buňky je opět 1 byte. Může sloužit jako uložiště obecných dat.

2.3. Funkční celky, části interpretu

V této části práce bych chtěl rozebrat k čemu slouží a jakým způsobem jsou implementovány jednotlivé části interpretu.

2.3.1. Vstupní část programu, inicializace

Vstupním bodem programu je standardně pojmenovaná funkce *main*, přesně řečeno její verze, jíž se předají proměnné, ve kterých jsou uloženy argumenty programu. Funkce *main* se zabývá pouze voláními, která interpret zinicializují, spustí a dokáže ho ukončit v případě, že nastane chyba. Inicializace částí interpretu probíhá ve vhodně zvoleném pořadí. Nejprve se prozkoumají argumenty programu. Následuje inicializace části pro uchování hodnot. Poté část zodpovědná za běh celého interpretu a nakonec se provede test, který má ukázat správnou funkčnost důležitých částí interpretu. Ten lze zakomentováním makra *DEBUG_TEST* v souboru `include/rbl-config.h` a přeložením vypnout.

2.3.2. Volací konvence

Volací konvence je pravidlo, které vymezuje, jakým způsobem bude volána jistá skupina funkcí. V tomto případě popisuje kvalitativní a kvantitativní stránku argumentů a návratového typu. Volba volací konvence spočívala ve vyřešení následujících problémů.

- Přenos chybového stavu. Na libovolném místě a v libovolném stupni vnoření volání inter-

pretu může dojít k chybě a úkolem je dopravit informaci k místu, kde bude na chybu reagováno.

- Přenos argumentů volaným funkcím. Přenos výsledků funkcím volajícím.
- Časová náročnost volání.
- Složitost použití.

Nabízely se dvě použitelné varianty.

(1) `kod_chyby F(*vysledek,  argument1, argument2, ...);`

(2) `vysledek  F(*kod_chyby, argument1, argument2, ...);`

První varianta by vykazovala výrazně větší časové zatížení, kdyby se v každém volání žádalo o paměť *garbage collector*. Kód interpretu se však chová tak, že místo pro výsledek se zpravidla alokuje na velice nízkém stupni vnoření a používá se pro celý podstrom volání. V praxi to znamená, že je režie *garbage collectoru*, způsobená tímto jevem, zanedbatelná.

Dalším argumentem pro druhou variantu by byla možnost optimalizovat kód tak, že by se do registrů procesoru hodnota vešla celá. Manuál *GCC* [7, sekce 3.17.13: Intel 386 and AMD x86-64 Options, přepínač -mregparm=num] však říká, že maximum možných registrů určených pro předávání argumentů je na zvolené architektuře *i386* 3. Jejich velikost je tedy 12 bytů. Celá hodnota však v této verzi interpretu nemůže být menší než 16 bytů.

V obou případech je před voláním nutné uložit argumenty na zásobník. Fakt, že by výsledek předchozího volání zůstal, pokud by to bylo možné, v akumulátoru procesoru, jak je to u tohoto typu překladače běžné, výhodou není, protože pro další volání je potřeba tuto hodnotu opět uložit na zásobník.

Argumentovat se dá také tím, že specifické optimalizace pro určitou platformu by vedly ke snížení stupně přenositelnosti.

Časová režie mechanismu předávání argumentů se zdá být pro větší počet volání srovnatelná. Kvůli, dle mého názoru, jednoduššímu způsobu použití jsem nakonec zvolil první variantu. I přesto jsem byl časem nucen zavést makro, které celé volání ještě více zjednodušilo.

```
#define R(fn,...) \
  if( (eResult = fn(__VA_ARGS__) ).ecCode != E_NOERR) \
    return(eResult)
```

Tomuto typu makra se říká *variadic*. Umožňuje volání makra s proměnným počtem argumentů. Použité pak na volání *eVProbe* vypadá takto.

```
R(eVProbe, &vResult, vValue);
```

S jeho použitím již nemusíme při každém volání řešit chybový kód. Podmínka v makru *R* kód otestuje a v případě výskytu chyby vyskočí z funkce na nižší úroveň.

2.3.3. Chybové stavy interpretu

Cílem práce na mechanismu oznamování chyb bylo dosáhnout dostatek informací o chybě v místě, kde jsou potřeba a snadné použití mechanismu v programu. Chybový stav je v interpretu popsán typem jazyka C *err_t*.

```
typedef struct{
  errcode_t ecCode;
  const char* sFileName;
  const char* sFunction;
  void *vpArg[3];
} err_t;
```

Je složen z identifikátoru chyby, který pomocí dalších agregovaných typů jednoznačně určuje povahu chyby. Obsahuje ukazatel na statický text se jménem zdrojového souboru a funkce, kde byla chyba detekována a pole neotypovaných ukazatelů, které jsou použity jako rozšiřující informace.

Povaha chyby je charakterizována tabulkou, jejímž prvkem je struktura jazyka C *err_node_t*.

```
typedef struct{
  int iArgCount,
  iTerminate,
  iExitCode;
  const char *sSysMsg,
  *sFormatString;
} err_node_t;
```

Každý řádek informuje o počtu povinných argumentů pole *vpArg* typu *err_t*. Velice důležitý je příznak, zda-li je možné se z chyby zotavit nebo zda je nutné interpret ukončit. Součástí řádku je pak i kód, který se má vrátit do operačního systému a především dva řetězce, které chybu popisují slovně. První se vytiskne nezměněný, druhým je formátovací řetězec pro funkci *printf* jazyka C, do kterého se při výpisu dosadí doplňující informace.

Podstatný je i způsob, jakým se vyskakuje z funkce v okamžiku, kdy chybu detekujeme. Protože by bylo velice pracné při každém možném výskytu chyby v programu sestavovat a vyplňovat datový typ *err_t*, zavedl jsem pro tento účel sadu maker. Každému z nich je společná část, která vyplní informaci o souboru a funkci, kde chyba nastala.

```
#define RET_STORE_INFO(res,ecCd)\
  res ecCode = ecCd;\
  res sFileName = __FILE__;\
  res sFunction = __PRETTY_FUNCTION__
```

Pro každý různý počet předávaných rozšiřujících argumentů z místa výskytu chyby musí existovat makro očekávající správný počet argumentů. V praxi jsem neměl potřebu přenášet více jak tři argumenty, takže existují 4 různá makra pro žádný až 3 argumenty. Takto vypadá makro pro jeden argument.

```
#define RETURN1(ecCd,arg1) {\
  RET_STORE_INFO(eResult.,ecCd);\
  eResult.vpArg[0] = (void *) arg1;\
  return(eResult);\
}
```

Šablonu jednoduché funkce potom naznačuje další příklad.

```
err_t eFn(value_t *vpResult, value_t vValue){
  err_t eResult;

  assert(vpResult);

  /* misto pro pozadovanou operaci */

  RETURN(E_NOERR);
}
```

Díky volací konvenci je možné přenést informaci o místě a příčině chyby do libovolné nížší úrovně vnoření funkce jazyka C. Informace je předávána přes zásobník. Mechanismus řeší problém detekce chyby ve vnořených funkcích. Prakticky se informace dostane ven z interpretu zapsaného v jazyce REBOL ve funkci *eEXRun*. Informace je předána funkci *vErr*, která vypíše požadované informace na výstup a podle příznaku v definici chyby rozhodne, zda se má program ukončit nebo ne. Výsledek pak může vypadat například takto.

```
>a
[./rbl: sourcefile 'value_word.c', function 'eVExecWord', input
'stdin']
script error, 'a' has no value
>
```

Seznam definovaných chyb a jejich krátký popis nabízí příloha D.

V kódu se stále nachází několik míst, ze kterých může být program ukončen nestandardně.

- Pokud je povoleno diagnostické testování pomocí volání *assert*, vede nesplnění testované podmínky na volání *abort*.
- V diagnostickém testu ve funkci, která nemá možnost vrátit chybu, je detekován nepovolený stav. Zde je *abort* volán záměrně, aby poukázal na místo detekce chyby.

2.3.4. Skoková tabulka

Skoková tabulka se používá jako programovací technika, kterou se nahrazuje konstrukce *switch* jazyka C. Požívá se kvůli výkonu, protože samotnému skoku předchází pouze výpočet zpravidla jednoduché mapovací funkce oproti lineárnímu průchodu řadou testů. Subjektivně je i přehlednější. Posouzení časové náročnosti pochopitelně záleží na uspořádání a počtu jednotlivých *case* v příkazu *switch*, na četnosti jejich volání, na optimalizačních technikách překladače

a na dalších faktorech. Na straně skokové tabulky pak na náročnosti výpočtu mapovací funkce. Nevýhodou skokové tabulky se může stát například skoro nezbytný jednotný prototyp volané funkce nebo zjevně větší náročnost na generovaný kód.

V mém případě by bylo nutné provést měření, zda-li má na výkon nezanedbatelný vliv. Pravým důvodem použití je však přehlednost výsledného kódu. Pro celý interpret existuje pouze jedna skoková tabulka.

```
jt_proto jtJumpTable[][JT_LAST_CALL + 1];
```

Je uspořádána jako dvourozměrné pole, jejímž prvním indexem je datový typ, nad kterým se operace provádí a druhým je identifikátor funkce, která se má provést. Prvkem tabulky je typ *jt_proto*, který je deklarován jako neotypovaný ukazatel. Řádek tabulky pak například pro datový typ REBOLu *char* může vypadat takto.

```
{F(eVParseValue), F(eVExecValue), F(iVSameChar),
F(eVConvertChar), F(eVFormatChar), F(iVMatchChar)}
```

Vídíme, že exportovaný název funkce zde figuruje jako argument makra preprocesoru `F`, které přetypovává funkci na jednotný typ – neotypovaný ukazatel.

Tabulka má v tuto chvíli tolik řádků, kolik je deklarovaných typů jazyka REBOL. Jejich očíslování (viz sekce 2.2) však neodpovídá indexům. Je tedy nutné použít mapovací funkci. Ta je reprezentována tabulkou *jtTypeMap*, jejímž indexem je číslo typu a položkou v ní je index do skokové tabulky. Dvouúrovňový překlad není nezbytný. Pokud bychom však použili pouze jednu úroveň, byla by tabulka využitá asi z jedné pětiny a velice rozsáhlá.

Funkce, jejichž odkazy jsou uložené ve skokové tabulce, jsou volány pomocí makra preprocesoru *JT_FN*.

```
#define JT_FN(prototype,value_type,call_id)\
  ( (prototype)(jtJumpTable[jtTypeMap[value_type] ][call_id]) )
```

V kódu interpretu pak volání může vypadat takto.

```
assert(JT_FN(JT_PROTO_iVMatch, vValue.vaAny.bType,
    JT_CALL_iVMatch) );

  iResult = JT_FN(JT_PROTO_iVMatch, vValue.vaAny.bType,
    JT_CALL_iVMatch)(
    eaArg->vInput, vValue, eaArg->iCase, eaArg->iAll);
```

Assertion je před voláním pouze bezpečnostním prvkem a je do jisté míry zbytečné, protože všechny prvky tabulky, které je zakázáno volat, obsahují odkaz funkci *abort()*. Interpret je pak bezpečným způsobem ukončen.

Poslední důležitým datovým polem definovaným ve stejném souboru jako skoková tabulka, je tabulka jmen datových typů jazyka REBOL. Je použit stejný dvouúrovňový překlad. Pomocí čísla typu je tak možno obdržet ukazatel na statický řetězec, představující jméno daného typu. Řádek tabulky obsahuje také číslo datového typu. Lze tedy provést i funkci inverzní, tedy z řetězce zjistit jeho číslo. Tabulka je deklarována následující konstrukcí jazyka C.

```
typedef struct{
  int   bType;
  char  *sName;
} jt_name_t;

jt_name_t jtTypeName[];
```

Přístup do ní je opět zjednodušen voláním makra, jehož použití je podobné jako u makra *JT_FN*.

```
#define JT_TYPE_NAME(type)\
  (jtTypeName[jtTypeMap[type]].sName)
```

2.3.5. Slovník

Slovník by měl umět popsat pravopis slov, které jsou do něj vloženy a jejich indexací umožnit jejich snadnou identifikaci. Pro implementaci je nutné použít mechanismus s rychlým vyhledáváním slova podle pravopisu. Ideálním řešením se zdá být hešovací funkce. Standardní knihovna operačního systému Linux nabízí správu hašovací tabulky ve dvou provedeních. Zvolil jsem reentrantní verzi implementace, konkrétně trojici volání *hcreate_r – hsearch_r – hdestroy_r*, která jsou narozdíl od starší verze, standardizované normou POSIX 1003.1-2001, reentrantní. Tato volba byla nezbytná, protože pouze tato umožňuje přístup k více tabulkám v rámci jednoho běhu programu.

Z hlediska interpretu je slovník definován typem jazyka C *dictionary_t*. Obsahuje ukazatel na hešovací tabulku a čítač položek v tabulce.

```
typedef struct{
  int iEntries;
  struct hsearch_data hTab;
} dictionary_t;
```

Funkce standardní knihovny jsou obaleny voláními interpretu. Dosáhneme tím jednotné a jednodušší volací konvence.

```
err_t eDTCreate(dictionary_t *dDict);
```

Funkce založí hešovací tabulku a připraví ji k prvnímu použití.

```
err_t eDTLookUp(dictionary_t *dDict, char *sText, unsigned *uId,
  lm_lookup_mode_t lmlmMode);
```

Funkce rozlišuje dva režimy, stejně jako její protějšek ze standardní knihovny volí se argumentem *lmlmMode*. Vstup je charakterizován parametrem *sText*, výstup pak parametrem *uId*. K porovnání klíčů v *hešovací tabulce* je použita funkce standardní knihovny *strcmp*. Tato funkce rozlišuje malá a velká písmena abecedy. Důsledkem je rozdíl v chování původní a této implementace interpretu. Tento u slov striktně rozliší *velikost* písmene[1].

[1]Anglicky tzv. *case-sensitive.*

```
err_t eDTDestroy(dictionary_t *dDict);
```

Funkce uvolní paměť přidělenou tabulce.

Interpret využívá minimálně tři takovéto slovníky. První se nazývá *globální slovník* a představuje výchozí slovník, se kterým pracuje uživatel interpretu. Samotný interpret běží v prostředí, které je skryto uživateli a používá svůj vlastní slovník slov, zvaný *interní slovník*. Poslední využívá funkce *parse* a nazývám ho *slovníkem pro funkci parse*.

2.3.6. Jmenný prostor

Jmenný prostor musí pojmout jak pohotovostní části hodnot, tak identifikátor, který bude nad tabulkou jednoznačným klíčem a bude spojovat hodnotu s položkou ve slovníku. Položkou se tedy může následující struktura.

```
typedef struct{
  unsigned uId;
  value_t vValue;
} node_t;
```

Jmenný prostor je pak tabulka těchto prvků a v praxi je definovaná takto.

```
typedef struct{
  node_t   *nArray;
  unsigned uEntries,
           uAllocated;
} namespace_t;
```

Obsahuje referenci na pole prvků, čítač položek v tabulce a hodnotu množství přidělené paměti. Tabulka s množstvím přidávaných prvků dynamicky roste. Počet volání funkce *GC_REALLOC*, se dá výrazně snížit žádostí o větší kus paměti, než je bezprostředně třeba. Poté, co je přidělená paměť spotřebována, se žádá znovu.

Způsob práce se jmenným prostorem je charakterizovaný skupinou volání, u které je dodržena jednotná volací konvence.

```
err_t eNSNew(unsigned uSize, node_t **nArray);
```

Volání připraví tabulku pro první použití. Toto volání je výhradně určeno pro použití z tohoto souboru a je vyčleněno z dále popisovaných funkcí pouze z důvodu redundance kódu.

```
err_t eNSCreate(namespace_t *nsNS);
```

Funkce založí jmenný prostor a připraví jej k použití.

```
err_t eNSStore(namespace_t *nsNS, unsigned uId, value_t vValue,
  value_t **vReference);
```

Volání slouží jako žádost o založení položky s hodnotou, reprezentovanou argumentem *vValue*. V rámci tohoto jmenného prostru ji bude jednoznačně charakterizovat předaný identifikátor *uId*. Novou polohu položky po úspěšném volání nelezneme v argumentu *vReference*.

```
err_t eNSLookUp(namespace_t *nsNS, unsigned uId,
  value_t **vValue);
```

Funkce se pokusí najít ve jmenném prostoru prvek s identifikátorem *uId*. V případě, že nalezne, lokaci jeho hodnoty nalezneme v argumentu *vValue.*

```
err_t eNSClone(namespace_t *nsNew, namespace_t *nsNs);
```

Funkce vytvoří kopii jmenného prostoru, tedy založí nový a obsah původního do nového překopíruje. Tato funkce v současné době není využita.

```
err_t eNSDestroy(namespace_t *nsNS);
```

Funkce uvolní paměť vyhrazenou pro tabulku.

Interpret obsluhuje tři oddělené jmenné prostory. Ke každému z nich existuje slovník, zmíněný v sekci 2.3.5, jehož funkce je vysvětlena tamtéž. Výčtem pak *globální jmenný prostor*, *interní jmenný prostor* a *jmenný prostor pro funkci parse.*

2.3.7. Funkce parse

Ačkoliv je funkce *parse* jazyka REBOL nativní funkcí, kterým je vyhrazena zvláštní sekce (2.3.9), její role v interpretu je zcela stěžejní a proto jí proberu samostatně.

Vlastnosti, které jsou dokumentovány v manuálu [13] a naznačeny v sekci 1.4 jsou implementovány pouze částečně. Prioritně byly naprogramované funkce, které potřebuje jednoduchý interpret pro svoji činnost. O implementaci interpretu v jazyce REBOL pojednává sekce 2.3.8.

Funkce *ePRParse*, na které vede požadavek *parse* z interpretu rozliší režim analýzy a volá funkci, která úkon provede.

```
err_t ePRParse(dictionary_t *dtDict, namespace_t *nsNs,
  value_t *vResult, value_t vInput, value_t vRules,
  int iCase, int iAll);
```

Argumenty *dtDict* a *nsNs* se používají pouze pro variantu **analýza podle pravidel** a představují slovník a jmenný prostor, v rámci kterého se budou spouštět akce. Výstup bude uložen do proměnné *vResult*, vstup je uložen ve *vInput*. *iCase* a *iAll* jsou příznaky, jejichž role je probrána v sekci 1.4.

Analýza pomocí oddělovačů

Režim **analýza pomocí oddělovačů** je plně funkční a výsledky by v porovnání s původní verzí interpretu REBOL měli být shodné. Analýzu provádí pro tuto činnost specializovaná funkce jazyka C *eParseUsingDelimiters* deklarovaná takto.

```
err_t eParseUsingDelimiters(value_t vInput, value_t vRule,
  value_t *vResult, int iCase, int iAll);
```

Argumenty jménem i použitím odpovídají funkci *ePRParse.* Její podstatnou část práce zastane funkce *sDelimiterToken*, která je sestavená tak, aby na vstupu hledala počátek a délku *tokenu* podle pravidel uvedených v tabulce číslo 1.1 v sekci 1.4. Následují deklarace funkcí, které jsou součástí stromu volání. Jejich funkce je zřejmá z názvu.

```
char *sSkipSet(char *sSrc, unsigned *uLength, char *sSet,
  unsigned uSetLength);
int iIsOneOf(char cSrc, char *sSet, unsigned uLength, int iCase);
```

Analýza podle pravidel

Vzhledem k podobnosti algoritmů umožňujících interpretaci kódu jazyka REBOL a interpretaci pravidel funkce *parse*, využívají oba podsystémy interpretu stejný mechanismus. Celek byl vyčleněn do zvláštního souboru a je popsán v sekci 2.3.8. Vstupním bodem tohoto celku je obvykle funkce *eEXDo*. Její činnost se nastavuje argumenty funkce, obsahem skokové tabulky a funkcemi, jejichž reference skoková tabulka obsahuje. Velice podstatná informace, kterou nesou argumenty je *prostředí*[1], ve kterém jazyk poběží. Nastavení prostředí pro interpretaci pravidel pro funkci *parse* se provádí ve funkci *ePRInit*, která je volána ve vhodnou chvíli při inicializaci. Spočívá v nastavení slov s používanými jmény. Jejich hodnotami se stane datový typ *native* s odpovídající referencí na vstupní bod funkce jazyka C. Volání vypadá takto.

```
eResult = eEXDo(&dtParse, &nsParse, vResult, &vRules, &eaArg);
```

Návratový kód se testuje a podle něj se nastaví výstup z funkce. Pokud vstup pravidlům vyhověl, je do výsledku zapsán datový typ *logic* s hodnotou true. V opačném případě hodnota false. Jak již bylo uvozeno, jazyk je implementován pouze částečně a rozdíl oproti původnímu interpretu je vidět v tabulce číslo 2.1.

Změny

V původním interpretu je způsob zpracování datového typu *word* v rámci bloku odlišný. Interpret identifikuje slovo podle pravopisu a v případě, že jde o jedno z vyhrazených pro akce[1], akce se provede. V opačném případě se pokusí najít slovo v kontextu slov. Pokud najde, získá jeho hodnotu a tuto se pokusí porovnat se vstupem. Pokud nenajde, oznámí, že slovo hodnotu nemá.

V mé implementaci existují pro blok pravidel i pro provádění akcí[2] dvě odlišná *prostředí*[3]. Blok pravidel se skrytě provádí stejným způsobem jako blok kódu jazyka REBOL. Nepřišlo mi rozumné implementovat výjimku pro chování vyhodnocení datového typu *word*, která by hledala akci v *prostředí funkce parse* a v případě neúspěchu posléze hodnotu v prostředí, ze kterého byla funkce *parse* spuštěna. Rozhodl jsem se změnit chování datového typu *paren*. V původním interpretu je to způsob, jak v rámci analýzy textu spustit kód jazyka REBOL. V mém pojetí je tato vlastnost doplněna o prvek komunikace mezi jednotlivými prostředími. Hodnota výrazu, který byl spuštěn v prostředí, ze kterého byla funkce *parse* volána, není ignorována, ale je vrácena do prostředí funkce *parse*. Tam je s hodnotou nakládáno stejně, jako s obsahem datového typu *word* v původním interpretu jazyka REBOL. Je porovnána se vstupem funkce *parse*.

Pokud potřebujeme pouze provést akci a nezajímá nás hodnota, kterou *paren* vrací, můžeme využít slova *none*, které vstupu vždy vyhoví a nezmění ho. Pokud jde o porovnání vstupu s

---

[1]Označme tak dvojici instancí agregovaných typů *dictionary_t* a *namespace_t*.

[1]Levá část tabulky 2.3.7

[2]Vyhodnocení typu *paren*.

[3]Označme tak dvojici instancí agregovaných typů *dictionary_t* a *namespace_t*.

hodnotou slova, uzavřeme slovo do *paren*. Tyto dvě pravidla však nejsou univerzální a při programování je vždy nutné vycházet z funkce pravidel, která navrhujeme.

Ačkoliv funkce nezbytné k provozu interpretu s datovým typem *string* jazyka REBOL pracovat umí, pravidla sloužící k analýze textové předlohy bloku nezahrnují případ, že by se na vstupu datový typ *string* mohl objevit. Podobě je tomu s dalšímy typy, se kterými interně interpret pracuje. Seznam rozlišených datových typů na vstupu najdete v sekci 2.3.8.

Analýza vstupu typu *block* pomocí funkce *parse* není k činnosti interpretu nezbytná a také není dobře testována. Proto je volání tohoto typu ve funkci *PRParse* znemožněno. Funkce vrátí chybový kód *E_NOIMPL*.

Funkce	Implementováno
|	ano
[block]	ano
(paren)	ano[1]
none	ano
opt	ne
some	ano
any	ano
skip	ne
to	ne
thru	ne
set	ne
copy	ano
word	ne[2]
word:	ano
:word	ano
'word	ne
type!	ne
end	ano

Datový typ	Implementováno
datatype	ne
char	ano
bitset	ano
any-string	ne

Tabulka 2.1. Stav implementace jazyka pravidel pro funkci *parse*

2.3.8. Interpretace, interpret jazyka REBOL

Interpret jazyka REBOL je již zapsán v samotném jazyce REBOL. Obměnu najdeme i v článku [9].

---

[1] Útvar *paren* vrací hodnotu. Více informací v Sekci 2.3.7, odstavec *Změny*.

[2] Vyhledání hodnoty nefunguje zcela stejně jako v původním interpretu. Více informací v Sekci 2.3.7, odstavec *Změny*.

```
while [true][
  prin #">"
  result: do load make block! input
  prin #"="
  probe result
]
```

Funkce, mimo výstupu a diagnostiku, provádí všechny tři kroky, zmíněné v sekci 1.3. Volání *input* přečte ze standardního vstupu procesu řetězec, který je použit jako argument funkci *make* s cílovým typem *block*. Vytvořený blok je před interpretací nutné spojit se *jmenným prostorem*. K tomu slouží funkce *load*. Poslední fází je spuštění interpretace pomocí funkce *do*. Výsledek volání je uložen do slova *result* pro pozdější diagnostický výpis.

Proberme si jednotlivé fáze detailněji.

Fáze *make*

Funkci *make* s požadovaným výsledkem typu *block* je možno nahradit analýzou vstupního textu pomocí funkce *parse* se specifickými pravidly. Tento krok provádí funkce *eVConvertBlock* jazyka C, definovaná v souboru *src/value_serie.c*. Jeho účel je probrán v sekci 2.3.10. Abychom mohli interpretovat první výraz, je nutné tato pravidla mít již připravena jako *block* jazyka REBOL, který prošel funkcí *load*. Proto bylo nutné maximálně zjednodušit způsob, jak programově vytvořit takový blok. K tomuto účelu slouží řada maker, definovaných v souboru *include/value_general.h*. Pokud tedy chceme vytvořit například *block* jazyka REBOL, obsahující

```
int: [some (digit)],
```

použijeme následující kód jazyka C.

```
value_t *vBlock;

RBINIT(vBlock);

  RBSETWORD("int"); RBBLOCK_BEGIN;
    RBWORD("some"); RBPAREN_BEGIN;
      RBWORD("digit");
    RBPAREN_FINISH;
  RBBLOCK_FINISH;

RBFINISH;
```

Je vidět, že vygenerovat takto *block* jazyka REBOL je nepřehledné a pracné. Proto bylo nutno pravidla zminimalizovat. Takto upravenou verzi, která je zapsaná v jazyce REBOL, vidíte v sekci C. Postupně definuje řadu slov, která jsou použita přímo v pravidle pro analýzu textové předlohy bloku *blockrule* nebo v dalších odkazovaných blocích. Pravidlo nejprve založí nový blok a uloží jej do slova *output*, kde se po zpracování bude nacházet výstup – sestavený blok. Stejným způsobem založí i zásobník úrovní vnoření *stack*. Pak testuje vstup postupně na tyto datové typy: *integer*, *char*, *get-word*, *word*, *set-word*, *datatype* a *path*. Závěr pravidla testuje začátek a konec datových typů *block* a *paren*. Neobsahuje test na několik interpretem podporovaných typů, neboť v první fázi inicializace nejsou nezbytné. Neřeší také otázku chyb ve vstupu, protože je předpoklá-

dano, že vstup, který je obsahem spustitelného souboru interpretu, bude řádně otestován. U každého z testů je definována akce, která ze vstupu vyextrahuje informaci, kterou využije pro vytvoření požadovaného datového typu a nastaví jeho hodnotu. Hodnota je pak vložena na konec výstupu *output*. U typů *block* a *paren* je použita rekurze. Pokud je detekován začátek, na konec výstupu *output* se vloží prázdný *block* nebo *paren*. Výsledek se pak uloží na zásobník. Výstup se nastaví na vloženou prázdnou hodnotu *block* nebo *paren*. Detekovaný konec datového typu *block* nebo *paren* znamená uložit do výstupu poslední hodnotu na zásobníku, která je poté odstraněna.

Fáze *load*

Fázi provádí funkce jazyka C *eEXLoadBlock*.

```
err_t eEXLoadBlock(dictionary_t *dtDict, namespace_t *nsNs,
  value_t vValue);
```

Pro každou hodnotu *any-word*, kterou obsahuje datový typ jazyka REBOL *any-block*, volá funkci jazyka C *eEXLoadWord*.

```
err_t eEXLoadWord(dictionary_t *dtDict, namespace_t *nsNs,
  value_t *vpWord);
```

V první řadě je pravopis slova uložen do slovníku voláním *eDTLookUp*, které je blíže popsáno v sekci 2.3.5. Výsledkem volání je *identifikátor slova*. Pomocí volání jazyka C *eNSLookUp*, detailněji popsaného v sekci 2.3.6, se funkce pokusí najít identifikátor ve *jmenném prostoru*. V případě, že nenajde, vytvoří novou hodnotu, kterou nastaví na typ *unset*. V obou případech pak uloží odkaz na *jmenný prostor*, *identifikátor* a odkaz na hodnotu do slova.

Nastavením hodnoty na *unset* je řešena detekce nedefinovaných proměnných. Pokud je tuto hodnotu později nucena vyhodnotit funkce *eVExecWord*, ukončí činnost nad právě zpracovávaným vstupem s chybou *E_NOVALUE*.

Fáze *do*

Datový typ *block* nebo *paren* jazyka REBOL provede volání jazyka C *eEXDo*. Stěžejní je *prostředí*[1], ve kterém má akce proběhnout. Určeno je argumentem *dtDict* a *nsNs*. Výsledek akce se uloží do předem alokovaného místa, na které ukazuje *vpResult*. Doplňující argumenty se ukládají do *eaArg*.

```
err_t eEXDo(dictionary_t *dtDict, namespace_t *nsNs,
  value_t *vpResult, value_t *vValue, exec_arg_t *eaArg);
```

Doplňující argumenty jsou reprezentovány typem *exec_arg_t*.

```
enum exec_mode_t {
  EM_PARSE = JT_CALL_eVParse,
  EM_DO = JT_CALL_eVDo
};

typedef struct{
```

[1]Označme tak dvojici instancí agregovaných typů *dictionary_t* a *namespace_t*.

```
  int           emMode;
  dictionary_t  *dtDo;
  namespace_t   *nsDo;
  value_t       *vInput;
  int           iCase;
  int           iAll;
} exec_arg_t;
```

V tuto chvíli, mimo identifikátor režimu, slouží výhradně funkci *parse*. Proměnné *dtDo* a *nsDo* charakterizují prostředí, ve kterém se budou spouštět akce. Vstup funkce je uložen do *vInput*. Chování režimu *parse* dále ovlivňují dva příznaky *iCase* a *iAll*. Více o funkci *parse* původního interpretu se dozvíte v sekci 1.4. O implementaci v této práci pak v sekci 2.3.7.

Funkce *eEXDo* se pokusí vyhodnotit hodnotu v bloku, na níž je ukázáno *offsetem* voláním *eEXEval*. Kód chyby slouží jako indikátor, zda-li vstup odpovídá pravidlům.

```
err_t eEXEval(dictionary_t *dtDict, namespace_t *nsNs,
  value_t *vpResult, value_t *vValue, exec_arg_t *eaArg);
```

Posledním článkem řetězce, odpovědného za interpretaci bloku, je volaní *eEXEval* jazyka C. Obsahuje kód, který podle zvoleného režimu a interpretovaného datového typu jazyka REBOL, zavolá pomocí skokové tabulky odpovídající funkci. Další úrovně vnoření jsou v režii obslužných funkcí, popisovaných v sekci 2.3.10.

Inicializace interpretu

Inicializace interpretu probíhá ve funkci *eEXInit*. Založeno je *prostředí interní*, ve kterém běží interpret jazyka REBOL a *prostředí externí*, ve kterém pracuje uživatel. Poté je v obou spuštěn blok, který nastaví slova, odkazující na *nativní funkce interpretu.* Do *interního prostředí* se zavedou pravidla pro analýzu řetězce. Pro kompletní funkčnost funkce *parse* je nutné nastavit prostředí *parse* voláním funkce *eRBInit*, která je popsána v sekc 2.3.9. Poté je již možné zavolat funkci *eVConvertBlock*, která umí vytvořit datový typ jazyka REBOL *block* z předlohry uložené v řetězci. Poslední fáze inicializace spouští v každém z prostředí jemu odpovídající inicializační blok, který je výsledkem převodu z inicializačního řetězce, uloženého ve spustitelném souboru interpretu.

Funkce *eEXRun* vytvoří datový typ *block* jazyka REBOL z inicializačního řetězce intereprtu a spustí ho.

2.3.9. Nativní funkce jazyka REBOL

Funkce *eRBInit*, volaná při inicializaci interpretu, vytváří pomocí sady maker jazyka C pro každou z nativních funkcí interpretu specifikaci rozhraní.

V mé práci jsou rozlišovány 2 odlišné typy nativních funkcí interpretu. První typ je spouštěn z prostředí *interního* a *externího.* Druhý pak z prostředí funkce *parse*. Jejich kostra je podobná a popisuje ji následující seznam.

- Definice povinných proměnných.

- Nahrání hodnot očekávaných argumentů z prostředí nebo ze spuštěného bloku.
- Nahrání příznaků funkce z prostředí nebo ze spuštěného bloku.
- Nahrání hodnot doplňujících argumentů z prostředí nebo ze spuštěného bloku.
- Testování nepovolených stavů, test argumentů.
- Tělo funkce.
- Uložení chybového stavu, návrat.

Z důvodu podobného typu volání u většiny funkcí existuje sada maker jazyka C, která má za úkol psaní kódu zjednodušit. Část z nich je totožná s makry definovanými v části interpretace, které se věnuje sekce 2.3.8. Nově se používají tato.

```
#define EXLOADARG(name,targetptr){\
  unsigned wId;\
  R(eDTLookUp, dtDict, name, &wId, LM_FIND);\
  R(eNSLookUp, nsNs, wId, &targetptr);\
}
```

Makro se pokusí najít slovo ve *slovníku slov*. Pokud najde, použije *identifikátor slova* slova k prohledání *jmenného prostoru.* Výsledkem je ukazatel na slovo, který je uložen do cíle.

```
#define EXLOADREFINEMENT(name,flag){\
  unsigned wId;\
  value_t *vTarget;\
  if(eDTLookUp(dtDict, name, &wId, LM_FIND).ecCode != E_NOERR)\
    flag = 0;\
  else{\
    if(eNSLookUp(nsNs, wId, &vTarget).ecCode != E_NOERR)\
  flag = 0;\
  if(VT_IS_NONE(*vTarget) )\
    flag = 0;\
  else\
    flag = 1;\
  }\
}
```

Funkce makra je obdobná. Výsledkem je uložený příznak, pokud je *refinement* nelezen a nastaven na *true.*

Činnost většiny definovaných funkcí je jednoduchá a spočívá buď ve volání jinde implementované funkce nebo konstrukci *switch* jazyka C. V jednotlivých *case* se pak provádí zpravidla opět jednoduchá operace. Složitějším funkcím se budu nyní věnovat.

Funkce interního a externího prostředí

RInput

Využívá volání do standardní knihovny jazyka C *fgets.* V jednom volání funkce přečte buď

celý řetězec, připravený na vstupu nebo zaplní poskytnutý *buffer*. Jednotlivé případy se rozlišují podle návratového kódu a podle toho, je-li posledním znakem v *bufferu* `'\n'`, tedy nový řádek. Volání je uzavřeno do cyklu, ve kterém se provádí *GC_REALLOC*, v případě opakovaného čtení. Z výsledného řetězce je poté sestavena hodnota, na kterou je nastaven datový typ jazyka REBOL *string.*

RWhile

Opakovaně vyhodnocuje blok *cond* a v případě, že je výsledek hodnota *true* a nenastane jiný chybový stav, vyhodnotí také blok *body*. Výsledek vyuhodnoceného bloku *body* je zapsán do výstupu fukce.

Funkce prostředí parse

PRAny

Opakovaně vyhodnocuje kód pravidla od původního místa. Chybový kód je *E_NOERR*, pokud nenastane jiná chyba než *E_NOMATCH* nebo *E_NOERR.*

PRCopy

Funkce si z bloku pravidel přečte jméno slova, do kterého se má výstup uložit. Poté uloží pozici textového vstupu a vyhodnotí následující položku v bloku pravidel. Vytvoří datový typ *string* jazyka REBOL a jeho hodnotu nastaví na podřetězec vstupu, který leží mezi pozicí uloženou a pozicí vstupu po posledním vyhodnocení. Odkaz na hodnotu zapíše do slova.

PRSome

Funkce je podobná funkci *PRAny*. Rozdílem je posouzení chybového stavu po prvním vyhodnocení. Pokud je chybový kód po prvním spuštění jiný než *E_NOERR*, vrací se tento chybový kód a funkce již dále neopakuje.

2.3.10. Práce s datovými typy

Sekce zahrnuje velké množství funkcí, z nichž převážná většina řeší jednoduché operace. Budu se tedy věnovat těm podstatným.

Funkce nezaměřené na datový typ

Jádro formátovacích funkcí, tedy funkce *mold*, *probe* a dalších, tvoří funkce, rozhodující o podobě formátovaného znaku.

znak (dekadicky)	*formát pro string*	*formát pro char*
0	"^@"	#"^@"
1-8	"^A" - "^H"	#"^A" - #"^H"
9	"^-"	#"^-"
10	"^/"	#"^/"
11-26	"^K" - "^Z"	#"^K" - #"^Z"
27	"^["	#"^["
28	"^\"	#"^\"
29	"^]"	#"^]"
30	"^!"	#"^!"
31	"^_"	#"^_"
34	{"}	#"^""
94	"^^"	#"^^"
127	"^~"	#"^~"

Tabulka 2.2. Pravidla formátování. Neuvedené znaky zůstávají nezměněny.

iIsDoubleChar

Funkce vrací počet bytů, které znak zabere ve zformátované podobě. Její operace je znázorněna tabulkou 2.2.

vFormatChar

Provádí konverzi znaku. Argument iQuote určuje, zda-li bude výsledek uzavřen do uvozovek či složených závorek.

eVFormatChar

Ke své činnosti využívá volání *iIsDoubleChar* a *vFormatChar*.

Funkce odvozené od typu function

Ačkoliv jsou implementovány i uživatelské funkce, které se snaží zajistit *lokální kontext* pro jejich proměnné, nebyla zvolena optimální metoda a proto může docházet k chybám. Interpret globální proměnné, které by byly přepsány lokálnímy, ukládá do tak zvaných *korekčních bloků*, ze kterých je po ukončení funkce obnovena jejich hodnota. Jednotlivé úrovně se však ovlivňují. Bylo by vhodné iplmenetovat postup, který naznačuje článek [4]. Úprava by však byla poměrně náročná, proto sem se jí nezabýval.

eLoadArgumentsBasic

Funkce nahraje do prostředí argumenty volané funkce.

eLoadArgumentsOptional

Funkce nahraje do prostředí slovo, jehož hodnotou je příznak, zda byla funkce volána

se stejnojmenným typem *refinement*. Zavolá funkci *eLoadArgumentsBasic*, která nahraje možné volitelné argumenty.

Funkce zaměřené na typ series

eVStubSeriesCreate

Funkce žádá o místo pro data série. Příznak napovídá, zda-li se má držet stanovené délky.

eVConvertBlock

Pokud je zdrojový typ řetězec, pro konverzi využívá funkci *parse* a pravidla analýzy do textu zapsaného bloku.

eVParseParen

Typ *paren* se v režimu *parse* interpretuje obdobně jako block v režimu *do*. Výsledek je předán k porovnání se vstupem.

Funkce zaměřené na typ word

eVExecWord

K rozhodnutí, jakým další způsobem se bude zpracovávat hodnota do slova uložená, slouží konstrukce jazyka C *switch*. Pro každý z implementovaných režimů je chování odlišné.

Kapitola 3. Závěr

Cílem práce bylo částečně implementovat interpret vyššího programovacího jazyka REBOL, který ukáže možný způsob, jak vytvořit některé funkční celky interepretu. Prostředkem k dosažení cíle měl být programovací jazyk C.

Výsledkem je interpret, ve kterém je implementováno na dvě desítky datových typů, a podobný počet nativních funkcí. Implementace některých částí funkcí chybí. Tyto části nejsou nezbytné pro funkčnost zbylého kódu nebo na ně nebyl kladen důraz v zadání.

Stěžejní částí interpretu je analýza a konverze předlohy datového typu *block*. Tato část umožňuje běh interpretu zapsaného v jazyce REBOL. Pravidla pro rozpoznávání vstupu nejsou vzhledem k naimplementovanému počtu datových typů kompletní. Z důvodu minimalizace sady pravidel nebyly části, odpovídající datovým typům, které nejsou stěžejní pro běh interpretu, začleněny. Kompletní seznam datových typů, které jsou na vstupu rozpoznány naleznete v sekci 2.3.8.

Podstatné bylo rozhodnutí, jak umožnit stanovení priority operací. Všechny funkce interpretu využívají notace *prefix*. Vzhledem ke skutečnosti, že neexistuje žádná funkce využívající jinou notaci, není nutné k zajištění jednoznačnosti priorit vyvíjet žádné další prostředky. Přesto je v interpretu dostupná metoda uzavření do závorek realizovaná pomocí datového typu *paren.*

Implementovány jsou jak nativní, tak uživatelské funkce jazyka. Pro řešení problému *lokálních kontextů* však nebyla zvolena optimální metoda, proto ne ve všech případech pracuje zcela správně. Systém zpracování chyb náležitě informuje o místě detekce a příčinách vzniku chyby. Umožňuje stanovit, zda-li je chyba pro interpret fatální či nikoliv. Interpret dokáže svoji činnost po zotavení z chyby obnovit. Více informací o funkčnosti této části interpretu najdete v sekci 2.3.3.

Jednou ze zásad vypracování byl i úkol vyřešit detekci nedefinovaných proměnných. Problém z větší části řeší funkce jazyka C *eEXLoadWord*, která přiřazuje slovu místo, kde bude uložena jeho hodnota. Pokud funkce shledá, že slovo již existuje, připojí k němu odpovídajcící hodnotu. Pokud je nové, nastaví jeho hodnotu na datový typ *unset*. Interpret tuto hodnotu při zpracování detekuje a aktuálně zpracovávaný příkaz ukončí. Více informací naleznete v sekci 2.3.8.

Je možné, že existuje rozpor mezi chováním původní a mé implementace interpretu, aniž je to v textu zmíněno. Tento problém je způsoben možnostmi studia původního interpretu. Mimo dokumentaci [13] a konzultace, byla jediným zdrojem informací *metoda zpětného inženýrství* [1]. Pokud jsem tedy na problém nebyl upozorněn nebo mi příčinu k pochybnostem nedala nekonzistence výsledného kódu, neměl jsem důvod problém hledat.

Příloha A. Instalace

K instalaci interpretu je nezbytná knihovna *garbage collectoru.* Zdrojový kód v aktuální verzi[1] je součástí přílohy cd-rom. Instalace probíhá standardním způsobem, obvyklým pro prostředí *GNU*.

Prvním krokem bude dekomprimace archivu.

```
alone@edmund:~/dl$ tar xvzf tar xvzf gc6.3.tar.gz
gc6.3/reclaim.c
gc6.3/allchblk.c
gc6.3/misc.c
...
gc6.3/digimars.mak
gc6.3/Makefile.direct
gc6.3/NT_STATIC_THREADS_MAKEFILE
alone@edmund:~/dl$
```

Nyní je nutné balíček nastavit pomocí skriptu automatické konfigurace *configure.*

```
alone@edmund:~/dl$ cd gc6.3
alone@edmund:~/dl/gc6.3$ ./configure
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
...
config.status: creating include/Makefile
config.status: executing depfiles commands
config.status: executing default commands
alone@edmund:~/dl/gc6.3$
```

Překlad provedeme příkazem *make.*

```
alone@edmund:~/dl/gc6.3$ make
Making all in doc
make[1]: Entering directory ‘/home/alone/dl/gc6.3/doc’
make[1]: Nothing to be done for ‘all’.
make[1]: Leaving directory ‘/home/alone/dl/gc6.3/doc’
...
(cd .libs && rm -f libgc.la && ln -s ../libgc.la libgc.la)
make[1]: Leaving directory ‘/home/alone/dl/gc6.3’
alone@edmund:~/dl/gc6.3$
```

[1]Soubor *gc6.3.tar.gz.*

Instalace binarních souborů je specifická pro distribuci operačního systému *GNU/Linux*. V mém případě je to distribuce *Slackware.*

Následuje přepnutí do režimu superuživatele.

```
alone@edmund:~/dl/gc6.3$ su
Password authentication bypassed.
root@edmund:/home/alone/dl/gc6.3#
```

Spustíme pomůcku *checkinstall*, která vytvoří instalační balíček.

```
root@edmund:/home/alone/dl/gc6.3# checkinstall

checkinstall 1.6.0, Copyright 2002 Felipe Eduardo Sanchez Diaz Duran
        This software is released under the GNU GPL.
...
**********************************************************************

 Done. The new package has been saved to

 /home/alone/dl/gc6.3/gc6.3-6.3-i386-1.tgz
 You can install it in your system anytime using:

     installpkg gc6.3-6.3-i386-1.tgz

**********************************************************************

root@edmund:/home/alone/dl/gc6.3#
```

Balíček pomocí nástroje *installpkg* nainstalujeme.

```
root@edmund:/home/alone/dl/gc6.3# installpkg gc6.3-6.3-i386-1.tgz
Installing package gc6.3-6.3-i386-1...
PACKAGE DESCRIPTION:
gc6.3: Package created with checkinstall 1.6.0
Executing install script for gc6.3-6.3-i386-1...

root@edmund:/home/alone/dl/gc6.3# exit
alone@edmund:~/dl/gc6.3$ exit
```

Tím je knihovna připravená pro použití.

Zdrojový kód interpretu se překládá stejným způsobem. Instalace pro účely testování není nutná.

Balíček nejprve rozbalíme.

```
alone@edmund:~/src/wrk$ tar xvzf rbl.tar.gz
rbl/
rbl/r/
```

```
rbl/r/Makefile
...
rbl/include/exec_h.h
rbl/include/error_h.h
rbl/COPYING
alone@edmund:~/src/wrk$
```

Následuje nastavení. V tomto kroku hledá konfigurační skript *configure* knihovnu *garbage collectoru.* Pokud nenajde, automatické nastavení se ukončí a oznámí chybu.

```
alone@edmund:~/src/wrk$ cd rbl
alone@edmund:~/src/wrk/rbl$ ./configure
checking for a BSD-compatible install... /usr/bin/ginstall -c
checking whether build environment is sane... yes
checking for gawk... gawk
...
checking for dlopen in -ldl... yes
checking for GC_malloc in -lgc... yes
checking how to run the C preprocessor... gcc -E
...
config.status: creating include/config.h
config.status: include/config.h is unchanged
config.status: executing depfiles commands
```

Nyní zbývá interpret přeložit.

```
alone@edmund:~/src/wrk/rbl$ make
Making all in src
make[1]: Entering directory ‘/home/alone/src/wrk/rbl/src’
if gcc -DHAVE_CONFIG_H -I. -I. -I../include    -g -Wall \
-DREDIRECT_MALLOC=GC_malloc -DIGNORE_FREE -I/usr/local/include \
-MT dt.o -MD -MP -MF ".deps/dt.Tpo" -c -o dt.o dt.c; \
then mv -f ".deps/dt.Tpo" ".deps/dt.Po"; else rm -f ".deps/dt.Tpo"; exit 1; fi
...
make[1]: Entering directory ‘/home/alone/src/wrk/rbl’
make[1]: Nothing to be done for ‘all-am’.
make[1]: Leaving directory ‘/home/alone/src/wrk/rbl’
alone@edmund:~/src/wrk/rbl$
```

Spustitelný soubor interpretu je připraven k použití. Na závěr provedeme test.

```
alone@edmund:~/src/wrk/rbl$ src/rbl
>quit
alone@edmund:~/src/wrk/rbl$
```

Příloha B. Funkce, příklady použití

V současné verzi interpretu jsou podporovány následující funkce. V hranaté závorce za názvem najdeme specifikaci rozhraní funkce.

+, -[left [number! char!] right [number! char!]]

```
>+ 1 2
=3
>- 3 5
=-2
```

tail[series [any-block! any-string! bitset!]];back[series [any-string! any-block!]]

```
>a: tail [1 2]
=[]
>back a
```

```
    =[2]
```

insert[series [any-block! any-string! bitset!] value [any-type!]];head[series [any-block! any-string!]]

```
    >a: insert [1 2] 0
    =[1 2]
    >head a
    =[0 1 2]
```

complement[value [logic! number! char! binary! string! bitset!]]

```
    >complement make bitset! []
    =make bitset!
    #{FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF}
```

do[value [any-type!]]

```
    >do [+ 1 1]
    =2
```

input[]

```
    >input
    asd
    ="asd"
```

make[type [datatype!]]

```
    >a: make function! [a] [+ a 1] none
    =none
    >a 1
    =2
```

last[series [any-string! any-block! bitset!]]

```
    >last [0 1 2]
    =2
```

load[source [any-block! any-string!]]

```
    >do load make block! input
    prin 1
    1>
```

parse[input [any-block! any-string!] rules[block! string! none]]

```
    >parse input none
    a s d
    =["a" "s" "d"]
```

prin[value [any-type!]]

```
    >prin 1
    1=1
```

print[value [any-type!]]

```
    >print 1
    1
    =1
```

probe[value [any-type!]]

```
>probe make char! 1
#"^A"
```

quit[]

```
>quit
```

remove[series [any-block! any-string! bitset!]]

```
>remove [1 2]
=[2]
```

union[set1 [any-block! any-string! bitset!] set2 [any-block! any-string! bitset!]]

```
>digit: make bitset! [- #"0" #"9"]
=make bitset!
#{000000000000FF0300000000000000000000000000000000000000000000000000}
>alpha: make bitset! [- #"0" #"9" - #"a" #"z"]
=make bitset!
#{00000000000000000000000FEFFFF07000000000000000000000000000000000}
>alphanum: union alpha digit
=make bitset!
#{000000000000FF0300000000FEFFFF07000000000000000000000000000000000}
```

while[cond [block!] body [block!]]

```
>while[parse input [#"a"]][1]
a
b
=1
```

Příloha C. Pravidla pro analýzu textu

```
[caret: make char! 94 dblquote: make char! 34
onechar: complement make bitset! []
digit: make bitset! [- #"0" #"9"]
ws: union make bitset! #" " make bitset! make char! 10
alpha: make bitset! [- #"0" #"9" - #"a" #"z"]
alphanum: union alpha digit
int: [some (digit)]
word: [[(alpha) any [(alphanum) | #"-"]] | #"-" | #"|"]
boundary: [end | some (ws) |
  position: #"[" :position | position: #"]" :position |
  position: #"(" :position | position: #")" :position]
blockrule: [(
  output: make block! 0
  stack: make block! 0 none) (block) any (ws)]
block: [any (ws) any [
  copy element (int) (boundary)
    (insert tail output make integer! element none) |
  #"#" (dblquote) copy element [(caret) (onechar) | (onechar)]
    (insert tail output make char! element none) (dblquote) (boundary)
|
  #":" copy element (word) (boundary)
    (insert tail output make get-word! element none) |
  copy element (word) [
    (boundary)
      (insert tail output make word! element none) |
    #":" (boundary)
      (insert tail output make set-word! element none) |
    #"!" (boundary)
      (insert tail output make datatype! element none) |
    (path: make path! 0 none) some [#"/" copy elementn (word)
      (insert tail path make word! elementn none)] (boundary)
      (insert/only tail output head insert head path make word!
element none)
  ] | [#"[" (element: make block! 0 none) | #"(" (element: make paren!
0 none)]
    (insert/only tail output element insert/only tail stack output
      output: element none) (block) |
  [#"]" | #")"]
    (output: last stack remove back tail stack element: none none)
(boundary)
]]]
```

Příloha D. Seznam Chybových stavů

E_NOERR
Operace proběhla bez chyby.

E_NOERREXIT
Operace proběhla bez chyby. Je požadováno ukončení interpretu.

E_NOIMPL
Funkce nebo její část není implementována.

E_GETOPT
Chyba volání, provádějící analýzu argumentů programu.

E_NOMEM
Nedostatek paměti nebo jiná chyba související se správou paměti.

E_IO
Chyba vstupu/výstupu.

E_INTERNAL
Vnitřní chyba interpretu.

E_WORDMISS
Slovo nebylo ve slovníku nalezeno.

E_IDMISS
Hodnota s tímto identifikátorem nebyla ve jmenném prostoru nalezena.

E_INVARG
Nesprávný typ argumentu funkce.

E_INVREF
Chybný formát datového typu refinement.

E_INVNS
Slovo postrádá záznam o jmenném prostoru.

E_INVFNSPEC
Chyba v interface specification funkce.

E_ARGMISS
Chybí argument funkce.

E_NOMATCH
Výsledek porovnání je negativní.

E_ALTERNATE
Nalezeno alternující pravidlo.

E_RANGE
Hodnota mimo povolený rozsah.

E_NOVALUE
Slovu doposud nebyla přiřazena hodnota.

E_EVIL
Interní chyba. Nepovolený stav.

Příloha E. Výpis obsahu cd-rom, statistiky

dp
Kořenový adresář diplomové práce. Zdrojové kódy pro sázecí systém Lout.

rbl
Kořenový adresář interpretu.

rbl/r
Doplňky interprety zapsané v jazyce REBOL.

rbl/src
Zdrojové kódy interpretu.

rbl/configure
Konfigurační nástroj balíčku.

rbl/include
Hlavičkové soubory interpretu.

dp.tar.gz
Archiv zdrojových kódů diplomové práce.

rbl.tar.gz
Archiv interpretu zdrojových kódů interpretu.

gc6.3.tar.gz
Zdrojové kódy knihovny garbage collectoru.

lout-3.30.tar.gz
Zdrojové kódy sázecího systému Lout.

velikost zdrojového kódu	178kB
počet řádků	7156
počet souborů se zdrojovým kódem	39
velikost spustitelného souboru (bez ladících informací, gcc 3.4.6)	160kB

Odkazy na literaturu

[1] Reverse engineering. URL *http://en.wikipedia.org/wiki/Reverse_engineering.*

[2] Kerninghan, B.W. - Ritchie, D.M.. *The C Programming Language*. Prentice-Hall Englewood Cliffs, 1978. Bratislava: Alfa, 1986

[3] Hans-Juergen Boehm. *Space efficient conservative garbage collection.* Conference on Programming Language Design and Implementation. SIGPLAN : ACM Special Interest Group on Programming Languages, ACM Press New York, NY, USA. URL *http://portal.acm.org/citation.cfm?doid=155090.155109.*

[4] Rebol Words and Contexts, alias Bindology. URL *http://www.fm.vslib.cz/~ladislav/rebol/contexts.html.*

[5] Hans Juergen-Boehm. The Boehm-Demers-Weiser conservative garbage collector. URL *http://www.hpl.hp.com/personal/Hans_Boehm/gc/*.

[6] Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd, 2005. URL *http://www.network-theory.co.uk/gcc/intro/*. ISBN: 0-9541617-9-3

[7] GCC online documentation. URL *http://gcc.gnu.org/onlinedocs/*.

[8] The GNU project. URL *http://www.gnu.org/*.

[9] Ladislav Mečíř. REBOL Programming/Advanced/Interpret. URL *http://en.wikibooks.org/wiki/REBOL_Programming/Advanced/Interpreter*.

[10] MinGW – Minimalistic GNU for Windows. URL *http://www.mingw.org/*.

[11] Hans Juergen-Boehm, Alan Demers, Scott Schenker. Mostly Parallel Garbage Collection. URL *http://www.hpl.hp.com/personal/Hans_Boehm/gc/papers/pldi91.ps.Z*.

[12] Perl. URL *htt://www.perl.com/*.

[13] Team of authors. REBOL/Core Users Guide. URL *http://www.rebol.com/docs/core23/rebolcore.html.*

[14] REBOL Programming/Language Features/Parse. URL *http://en.wikibooks.org/wiki/REBOL_Programming/Language_Features/Parse.*

[15] Regular expression. URL *http://en.wikipedia.org/wiki/Regular_expression.*

[16] WIKIPEDIA. URL *http://wikipedia.org/*.

Rejstřík