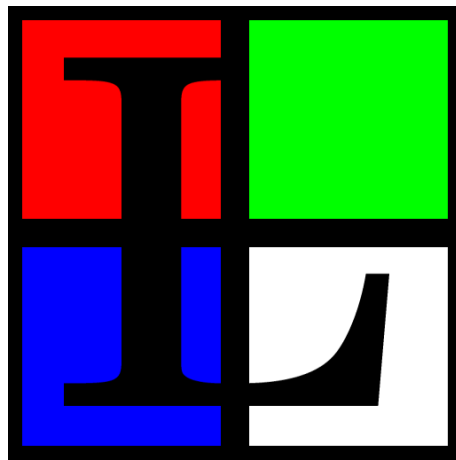


Universita Karlova v Praze
Matematicko-fyzikální fakulta
2002

SOFTWAREVÝ PROJEKT

Links

webový prohlížeč



Vývojová dokumentace

Autoři: Mikuláš Patočka
Martin Pergel
Petr Kulhavý
Karel Kulhavý

Vedoucí: Mgr. David Bednárek

Obsah

1.	Úvod	4
2.	Adresářová struktura	4
2.1	Adresáře	4
2.2	Informační soubory	4
2.3	Skripty, spustitelné soubory a konfigurační soubory	4
2.4	Zdrojové texty a hlavičkové soubory	5
2.5	Které soubory mazat	8
2.6	Pomocné programy a soubory	9
2.6.1	Arrow	9
2.6.2	Calibrate	9
2.6.3	Colour.html	9
2.6.4	Clip	9
2.6.5	Generate_font	9
2.6.6	Genps	9
2.6.7	Improcess	10
2.6.8	Makefont	11
2.6.9	Pbm2png	11
2.6.10	Pdf2html	11
2.6.11	Wb02links	11
2.7	Popis datových souborů	12
2.7.1	Tabulky znakových sad	12
2.7.2	Jazykové překlady	13
2.8	Konfigurační soubory	14
2.8.1	Záložky	14
2.8.2	Historie	15
2.8.3	Konfigurační soubory	15
3.	Obecně o programování Links	15
3.1	C jazyk	16
3.2	Přidávání zdrojů	16
3.3	Používání autoconfu	17
3.4	Velikosti dat	17
3.5	Alokace paměti	17
3.6	Chyby	17
3.7	Řetězce	18
3.8	Řetězce s prealokací	18
3.9	Seznamy	19
3.10	Jak používat grafiku (textový vs. grafický mód)	20
3.11	Psaní javascriptu	21
3.11.1	Debugování javascriptu	21
3.12	Formáty obrazových dat	21
4.	Členění do modulů	22
4.1	Select smyčka	23
4.2	Session	23

4.3	Object requester	23
4.4	Sched	23
4.5	Cache	23
4.6	Url	24
4.7	Http, https, finger, ftp, file	24
4.8	Cookies	24
4.9	Jsint	24
4.10	Javascr.l — lexikální analýza	25
4.10.1	Typy tokenů	25
4.10.2	Identifikátory	25
4.11	Javascript.y — syntaktická analýza	25
4.11.1	Syntaktická analýza	26
4.11.2	Generátor mezikódu	26
4.11.3	Fosilie	26
4.12	Builtin	26
4.13	Ipret — interpret mezikódu	27
4.13.1	Vztahy k okolí	27
4.14	View a view_gr	27
4.15	Img a img_cache	27
4.16	HTML parser: moduly html_r, html_gr, html, charsets	28
4.17	Moduly bfu a menu	28
4.18	Grafické ovladače	28
4.19	Dip, dither, font_data a font_cache	29
5.	Vnitřní struktury a jejich komunikace	29
6.	Princip jednotlivých částí	30
6.1	Fonty	30
6.2	Gamma	32
6.3	HTML parser a sazeč	34
6.3.1	HTML parser	34
6.3.2	Sazeč HTML	36
6.4	Javascript	36
6.4.1	Interpretování javascriptu	36
6.4.2	Gramatika	37
6.4.3	Typové konverze	37
6.4.4	Scheduler	37
6.4.5	Struktury javascriptu	38
6.4.6	Hlášení chyb	38
6.4.7	Ladění	39
6.5	Obecné seznamy	39
6.5.1	Úvod	39
6.5.2	Ovládání uživatelem	40
6.6	Datové struktury a funkce	41
6.6.1	Funkce	43
6.6.2	Implementace seznamu	43
6.6.3	Stromový seznam	43
6.6.4	Přidávání a editace položek — funkce edit_item	43

6.6.5	Přístup do seznamu z více oken Linksu	43
7.	Přidávání nových částí prohlížeče	44
7.1	Přidání souboru do distribuce Linksu	44
7.2	Jak přidat novou překladovou tabulku kódování	44
7.3	Přidávání nového grafického formátu	44
7.4	Přidávání nového jazyka	46
7.5	Přidání nového řetězce do jazykových překladů	46
7.6	Přidání nového fontu	47
7.7	Přidávání fontů z Ghostscriptu	48
7.7.1	Kterak postupovat jako uživatel	49
7.8	Přidání nových znaků do existujícího fontu	50
7.9	Přidávání fontů z tištěné předlohy	51
7.10	Přidání grafického driveru	51

1. Úvod

Toto je vývojová dokumentace k programu **Links**. Najdete zde popis činnosti programu, popis jednotlivých rozhraní, rozdělení na moduly, popis jednotlivých modulů, popis významných funkcí a informace, jak případně psát další části prohlížeče. Tištěná verze dokumentace neobsahuje popis interních rozhraní programu, ta jsou v plné verzi dokumentace uložené na CD.

2. Adresářová struktura

2.1 Adresáře

V adresáři `links-2.0`, kde jsou zdrojové texty **Links**, najdete tyto podadresáře:

- `.deps`: adresář obsahující dependence (informace o závislostech) jednotlivých zdrojových souborů. Dependence jsou potřeba pro program `make`, aby věděl, které soubory se mají při změně rekompilovat.
- `Unicode`: kódovací tabulky z rozličných znakových sad do unikódu, tabulky jsou ve zdrojové podobě (což je vhodné zejména pro přidávání nových tabulek a skripty (skript `gen`) pro generování zdrojových textů C).
- `doc`: adresář s veškerou dokumentací. V jeho podadresáři `tex` jsou k nalezení \TeX ové dokumentace, v podadresáři `examples` ukázkové příklady funkcí prohlížeče.
- `graphics`: veškerá pomocná data pro grafiku, zejména fonty (v podadresáři `font`), kurzorová šipka, ikona, logo. Adresář opět obsahuje skript `gen` pro generování zdrojových textů C pro kompilaci.
- `intl`: obsahuje překlady menu do různých jazyků, skript `gen-intl` pro vygenerování zdrojových textů C a skript `synclang` pro synchronisaci jazyků, když se přidá nový text.
- `parser`: zdrojové texty parseru javascriptu pro programy Lex a Bison a opět skript `gen`, který vygeneruje zdrojové texty parserů v jazyce C.

2.2 Informační soubory

Adresář obsahuje následující informační soubory, některé jsou poněkud starší a již se nepoužívají, přežívají z dřívějších dob.

- `AUTHORS`: seznam lidí, kteří do **Links** zasahovali.
- `BUGS`: známé chyby.
- `COPYING`: GNU licence.
- `Changelog`: obsahuje změny mezi jednotlivými verzemi.
- `FILES`: stručný popis souborů v adresáři.
- `INSTALL`: návod k instalaci.
- `NEWS`: novinky v **Links**.
- `README`: velmi stručná dokumentace o ovládání.
- `SITES`: seznam adres, kde se dá **Links** stáhnout.
- `TODO`: seznam, co je potřeba udělat v textové verzi.
- `colour.html`: kalibrační soubor pro zobrazování barev v textu.
- `links.1`: manuálová stránka.

2.3 Skripty, spustitelné soubory a konfigurační soubory

- `Makefile.am`: zdrojový soubor pro `automake` pro výrobu `Makefile`.
- `Makefile.in`: `automakem` vygenerovaný vstup pro `autoconf` pro výrobu `Makefile`.
- `Makefile`: základní soubor pro program `make`, nutný pro kompilaci.
- `acconfig.h`: `autoconfem` automaticky generovaný soubor. Pokud ho omylem smažete, je potřeba pustit skript `rebuild`.
- `aclocal.m4`: je soubor automaticky generovaný programem `aclocal`. Soubor nemodifikovat, vyrobí se nový když se pustí `aclocal`.
- `configure.in`: do tohoto souboru se přidávají nové testy, které se mají provádět v `configure` scriptu. Test většinou spočívá v pokusu najít nějaký soubor nebo nějakou funkci, zkompilovat s nějakou knihovnou a podle výsledku nastavit nějaké makro, které se pak později použije ve zdrojových textech **Links**.
- `configure`: známý konfigurační skript, který se pouští před kompilací.
- `config.h.in`: zdrojový soubor pro `config.h`, tento soubor je používán při pouštění `autoconfu`.
- `config.h`: je generován `configurem`, obsahuje konstanty, které definuje nebo oddefinuje `configure` skript po svém spuštění v závislosti na zjištěných komponentách systému.
- `convert_bookmarks`: pomocný skript na převedení starého formátu bookmarků (lineární formát, používaný v dřívějších textových verzích) na nový (stromový) tvar bookmarků. Nevytváří adresáře, jen čistě převede soubor do formy čitelné novými bookmarky. Vytvoření případných adresářů a zatřídění záložek do adresářů si pak uživatel musí provést sám.
- `install-sh`: skript potřebný pro `autoconf` a `automake` pro případnou instalaci souborů. Nepoužívá se, ale `autoconf` a `automake` ho vyžadují.
- `links`: spustitelný binární soubor prohlížeče, vytvoří se kompilací.
- `mailcap.pl`: perlový skript na konverzi `/etc/mailcap` do `links.cfg`. Tento skript nepsal nikdo z týmu projektu.
- `missing`: interní skript `autoconf` a `automake`.
- `mkinstalldirs`: další interní skript `autoconfu` a `automake`.
- `purge`: skript na kompletní vyčištění adresáře od všech produktů kompilace.
- `rebuild`: script na kompletní přebudování. Doporučuje se používat, když chcete **Links** od základů překonfigurovat a překompilovat, skript pusťte a on všechno zařídí. Obsahuje netriviální sekvenci příkazů, které je potřeba pro kompletní `rebuild` napsat.

2.4 Zdrojové texty a hlavičkové soubory

- `af_unix.c`: komunikace mezi několika spuštěnými instancemi **Links** přes UNIX domain sockets.
- `arrow.inc`: C zdroják obsahující grafická data kurzorové šipky. Kurzorová šipka se používá ve `svglib` a `framebufferu`.
- `atheos.cpp`: grafické rozhraní pro `Atheos`. Jako jediné je napsáno v C++, protože `Atheos` je v C++.
- `beos.c`: emulace `syscallů` `BeOSu`.
- `bfu.c`: funkce pro kreslení klikátek, dialogů, menu ..., funguje v grafice i v textovém režimu. Definice menu jsou v `menu.c`, ne tady.

- `bookmarks.c`: bookmarky, práce s nimi, ukládání, nahrávání...
- `builtin.c`: volání upcallů, implementace vestavěných metod a objektů v javascriptu.
- `builtin.h`: pomocný hlavičkový soubor s rozhraním funkcí pro práci s vestavěnými metodami a objekty javascriptu.
- `builtin_keys.h`: definice konstant pro vestavěné proměnné, metody a objekty javascriptu.
- `cache.c`: obecná LRU cache na dokumenty.
- `cfg.h`: konfigurační soubor, který se includeje ve všech modulech. Tento soubor pro zjednodušení pouze zastřešuje ostatní konfigurační soubory (`config.h`, `config2.h`) a systémově závislé konfigurace.
- `charsets.c`: funkce pro konverzi mezi znakovými sadami.
- `codepage.h`: obsahuje počet tabulek znakových sad.
- `codepage.inc`: strojově generované tabulky pro překlad kódování, soubor je v jazyce C.
- `connect.c`: funkce pro vytváření, rušení a práci se síťovým spojením.
- `context.c`: funkce pro správu javascript kontextu.
- `cookies.c`: vše pro práci s cookies.
- `default.c`: nastavení všeho možného, nahrávání a ukládání nastavení na disk, parsování příkazové řádky.
- `dip.c`: digital image processing. Zvládá tisk řetězců, písmen, počítání délky řetězců. Jsou zde funkce pro nahrávání písmenek, cacheování písmenek a jejich metrik. Dále je tu změna měřítka šedotónových bitmap a pronásobení 2 konstantních barev přes alpha masku. Gamma korekce 3*8 bitů barevné bitmapy.
- `dither.c`: vše pro ditherování bitmap. Brutálně optimalizované ditherovací enginy Floyd-Steinbergova algoritmu s maskou, funkce pro zaokrouhlování, funkce pro výrobu ditherovacích tabulek.
- `dns.c`: DNS resoluce.
- `drivers.c`: společný zdroják pro grafické drivery. Nyní obsahuje `svgalib`, `pmsHELL` (v OS/2), `X`, `Atheos`, `framebuffer`. Dále jsou zde virtuální devicy.
- `entity.inc`: C zdroják, tabulka pro překlad HTML entit do unikódu, používá se při dekódování HTML entit v HTML parseru.
- `error.c`: chybové hlášky, detekce memory leaků.
- `file.c`: kód pro nahrávání souborů z lokálního filesystému.
- `finger.c`: kód pro obsluhu protokolu finger.
- `font_include.c`: PNG fonty připravené k zakompilování do výsledného binárního spustitelného souboru. Soubor je strojově generovaný a poměrně velký.
- `ftp.c`: FTP protokol.
- `framebuffer.c`: grafický ovladač pro Linux framebuffer.
- `gif.c`: dekodér grafického formátu GIF.
- `html.c`: HTML parser.
- `html_gr.c`: sazeč HTML pro grafický mód.
- `html_r.c`: sazeč HTML pro textový mód.
- `html_tbl.c`: sazeč tabulek.

- `http.c`: HTTP protokol.
- `https.c`: HTTPS protokol.
- `img.c`: zobrazování obrázků v grafickém módu.
- `imgcache.c`: cache na dekódované obrázky.
- `ipret.c`: interpret mezikódu javascriptu.
- `ipret.h`: pomocný soubor s hlavičkami funkcí.
- `javascr.c`: lexikální analyzátor vygenerovaný flexem.
- `javascript.c`: syntaktický analyzátor vygenerovaný bisonem.
- `javascript.h`: hlavičkový soubor k syntaktickému analyzátoru.
- `jpeg.c`: dekodér grafického formátu JPEG.
- `jsint.c`: rozhraní mezi javascriptem a prohlížečem — upcally.
- `kbd.c`: funkce na čtení z klávesnice. Používá se i ve framebufferu a ve svgalibě, abychom si nemuseli dělat autorepeat a hlavně abychom mohli využít překlad národních abeced dělaný kernelem.
- `language.c`: kód na překlad menu do cizích jazyků.
- `language.h`: skriptem generovaný soubor s konstantami pro jednotlivé texty do menu.
- `language.inc`: skriptem generovaný C zdroják obsahující překladové tabulky pro překlad menu do různých jazyků.
- `links.h`: hlavní hlavičkový soubor se všemi potřebnými funkcemi. Aby u každého C souboru nemusel být hlavičkový soubor, tak je všechno v tomto souboru a odpadají starosti se vzájemnými inklusemi.
- `links_icon.c`: ikona připravená k zakompilování do výsledného binárního souboru. Ikona je vygenerována GIMPem a mírně ručně upravená.
- `listedit.c`: obecná implementace seznamů pro správu nějakých objektů, včetně hotových klikátek pro uživatele. Pak už jen stačí napsat krátký kód jako například pro bookmarky a bookmarky jsou hotovy. Dá se použít i pro správu jiných věcí, než jsou jen bookmarky (extensions, asociace, cookies...). Zavoláním jedné funkce se vytvoří okno pro správu daných objektů, uživatel může objekty mazat, stěhovat, vytvářet, prohlížet a volat na ně určité akce.
- `lru.c`: LRU cache na ukládání písmenek.
- `mailto.c`: kód pro obsluhu URL typu `mailto:user@host`.
- `main.c`: funkce `main`, inicializace a deinicializace všech subsystémů včetně grafického. Pokud nějaký `*.c` má funkci `ci` dělá inicializaci tak ji tu většinou zavolá. Pak je tu také deinicializace (proces opačný). Select smyčka zde není, je v samostatném souboru `select.c`.
- `md5.c`: funkce pro práci s MD5 v javascriptu, low-level funkce pro počítání MD5. Javascript podporuje počítání MD5. Tento soubor nevytvořil nikdo z týmu projektu, je stažený odkudsi z webu.
- `md5.h`: hlavičky funkcí k `md5.c`.
- `md5hl.c`: „obaly“ hrubých funkcí pro počítání MD5 přátelštějším kódem. Tento soubor nepsal nikdo z týmu projektu, patří k `md5.c`.
- `memory.c`: memory management, funkce na alokování a dealokování paměti, detekce memory leaků atd.
- `menu.c`: veškerá menu.
- `ns.c`: kód obsluhující prostor proměnných v javascriptu.

- `ns.h`: pomocný hlavičkový soubor pro `ns.c`.
- `objreq.c`: object requester (viz. dále).
- `os_dep.c`: věci jakkoliv závislé na OS.
- `os_dep.h`: dependence na různé OS, zahrnuje se před standardními `libc` includey.
- `os_dep_x.h`: podobně jako `os_dep.h`, ale zahrnuje se až po `libc` includech.
- `pmshell.c`: grafické rozhraní pro OS/2 `pmshell`.
- `png.c`: dekodér grafického formátu PNG.
- `pomocny.c`: pomocné funkce pro javascript.
- `sched.c`: schedulování síťových spojení, včetně `keepalive` spojení.
- `select.c`: `select` smyčka (viz. dále).
- `session.c`: řeší pohyb po dokumentu, historii, nahrávání a zobrazování dokumentů, cachuje zformátované dokumenty.
- `setup.h`: základní nastavení všeho druhu. Startovací URL, defaultní timeouty, barvy menu a dialogů a podobně.
- `struct.h`: potřebné struktury (zejména `js_context`, `namespace`) pro javascript.
- `svglib.c`: grafický ovladač pro `svglib`.
- `terminal.c`: rozhraní pro výstup na terminál.
- `tiff.c`: dekodér obrazového formátu TIFF.
- `tree.h`: opcodes mezikódu javascriptu (`javascript.y`), používá se v generátoru mezikódu a následně v interpretru (`ipret.c`).
- `types.c`: typy souborů a jejich přípony, volání externích programů.
- `typy.h`: definice typů proměnných javascriptu.
- `uni_7b.inc`: C zdroják s překladovou tabulkou mezi unikódem a 7-bit ASCII. Tabulka je strojově generovaná.
- `url.c`: funkce pro parsování URL.
- `view.c`: zobrazovač stránky v textovém módu.
- `view_gr.c`: zobrazovač stránky v grafickém módu.
- `win32.c`: port na Win32. Tento kód nepsal nikdo z týmu projektu, kód je starý a pravděpodobně nefunkční. Je zde ponechán, pro případ, že by se v budoucnu našel někdo, kdo by port pro Windows napsal.
- `x.c`: grafický driver pro XWindow.
- `xbm.c`: dekodér grafického formátu XBM.

2.5 Které soubory mazat

V adresáři s distribucí **Links** můžete smazat následující soubory. Tyto soubory je možno vygenerovat znovu kompilací nebo rebuildem. Všechny tyto soubory smaže skript `purge`.

- spustitelné programy ve formátu `elf`, `core`, `*.o`, `*.core`, ty smaže příkaz `make clean`
- adresář `.deps`
- `Makefile`, `Makefile.in`
- `aclocal.m4`
- `config.h`, `config.h.in`, `config.log`, `configure`, `config.cache`, `config.status`

- `autoh*`, ale to se při používání `rebuild` a nepouštění `autoheader` ručně nevyskytuje
- `.links_svg_modeinfo`

2.6 Pomocné programy a soubory

2.6.1 Arrow

Tento program vyrábí z `arrow.png` (který je ve formátu PNG, kde barvy číslo 0,1,2 udávají šipečku, okolí a průhlednost) do `arrow.inc`, který obsahuje řady 32-bitových čísel, pro šipečku i pro její průhlednostní masku. Šířka šipečky musí být 32 pixelů.

2.6.2 Calibrate

Tento program vyrábí černě až bíle vybarvenou obrazovku, v 256 stupních. Pracuje v truecolor režimu 640x480x16M. Používá se k měření gamma monitoru pomocí měřiče světla (hardwarového to zařízení).

2.6.3 Colour.html

Obsahuje texty o různých barvách. Vhodné pro testování **Links** v grafickém režimu, zejména ditherování, zmenšování písmenek a správné gammy.

2.6.4 Clip

Ořeže boxy z pngček — písmenek pro **Links**. Volá pomocný program `improcess`.

2.6.5 Generate_font

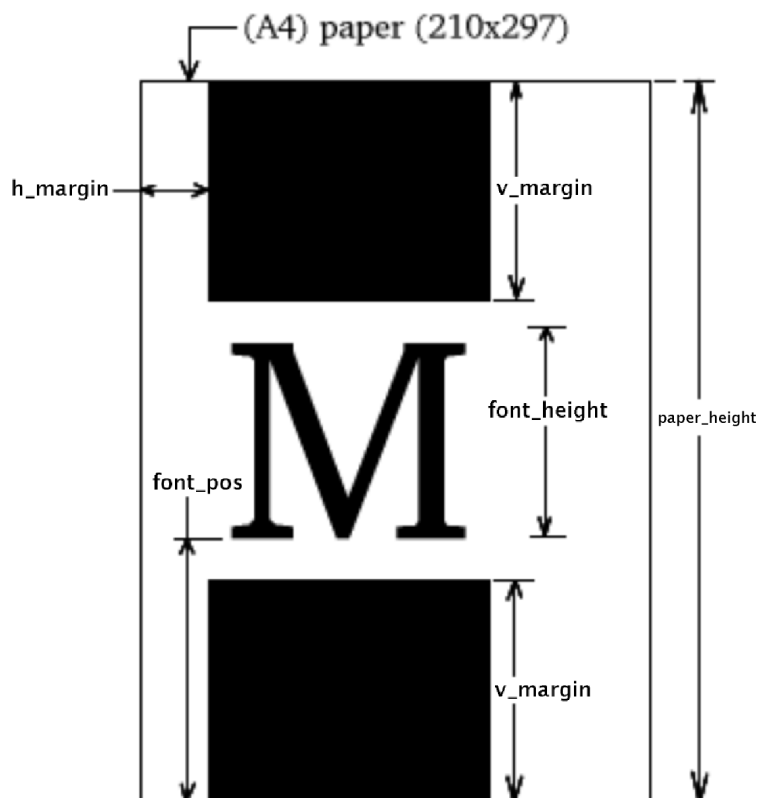
Vezme soubory v adresáři `font` a vyrobí z nich soubor `font.include.c`, který se přikompiluje do programu a obsahuje interní strukturu obsahující font ve formátu PNG. Tato data jsou tím pádem uložena ve spustitelném souboru `links` a uživatel není obtěžován žádnými dodatečnými datovými adresáři, čímž se silně zvyšuje přenositelnost celého systému (nebylo jasné, kam by se měly tyto soubory umisťovat, neboť takováto místa vhodná pro umisťování těchto souborů jsou silně závislá na konkrétním typu systému a vznikaly by s tím jen problémy).

Fonty se do systému přidávají tak, že se přidá do adresáře `font`, pak se pustí `generate_font` a pak se překompiluje `links` příkazem `make`. Detailnější popis je v kapitolách Přidávání fontů z Ghostscriptu a Přidávání fontů z tištěné předlohy.

2.6.6 Genps

Je určen k výrobě linksových fontů z fontů v Ghostscriptu. Vyrobí soubor `letters.ps`, který obsahuje všechna písmena od 0 do 255 tohoto fontu, přičemž kolem nich jsou boxy pro správné oříznutí. Tento program není určen pro ruční spouštění.

Uvnitř zdrojáku `genps.c` uživatel může nastavit proměnné `h_margin`, `v_margin`, `font_pos`, `font_height`, `paper_height`. Viz následující obrázek:



`paper_height` musí odpovídat výšce papíru, která je nastavena v `pdf2html`, tedy A4 (není důvod ji měnit), což je 297. Rozměrem těchto všech proměnných je 1 milimetr.

2.6.7 Improcess

Program `improcess` představuje nástroj pro základní manipulace s černobílými obrázky PNG. Program nahraje obrázek, provádí na něm unární operace (není tedy možno kombinovat dva obrázky do sebe) a nakonec jej uloží do souboru. Syntaxe je

```
improcess infile cmdfile outfile
```

kde `infile` je původní obrázek, `outfile` nový a `cmdfile` soubor s příkazy. `infile` může být totožné s `outfile`, k destrukci dat při tom nedojde.

`cmdfile` obsahuje řádky, kde každý řádek definuje jeden příkaz. Příkazy jsou tvaru `příkaz argument argument ...`, kde `argument` může být pouze znaménkové decimální nebo hexadecimální (s `0x` prefixem) číslo. Šedé tóny jsou reprezentovány čísly typu `int`, kde 0 je černá, `0xffffffff` bílá a při operacích je možno vyjet až do rozsahu `-0x80000000` až `0x7fffffff`, přičemž dále dojde k přetečení a nedefinovanému stupni šedé. Čísla v argumentu udávající barvu se řídí také touto konvencí.

Program umí následující příkazy:

- `clip` — ořeže do rozsahu mezi černou a bílou.
- `threshold level` — prahování. Pokud je `pixel >= level`, pak se vybělí, jinak se vyčerní.

- `flip` — transpozice. Prohodí se levá a horní hrana obrazu.
- `append lines value` — přidání řádků. Přidá `lines` řádků dospod obrazu, které budou vyplněny barvou `value`.
- `detract lines` — odstranění řádků. Odstraní `lines` řádků zespoda obrazu
- `mirror` — zrcadlení. Prohodí levou a pravou hranu obrazu.
- `blurbox pixels` — krabicové rozmazání. Provede konvoluci krabicovým konvolučním jádrem o výšce 1 pixel, šířce $2 \cdot \text{pixels} + 1$ centrováném na počátku souřadné soustavy.
- `gaussian repeat pixels` — gaussovské rozmazání. Provede `repeat`-krát `blurbox pixels`.
- `* multiplier` — násobení. Vynásobí všechny pixely hodnotou `multiplier`.
- `/ divisor` — dělení. Vydělí všechny pixely hodnotou `divisor`.
- `+ value` — přičítání. Přičte k obrazu hodnotu `value`.
- `>> shift` — posuv doprava. Vynásobí obraz 2^{shift} .
- `<< shift` — posuv doleva. Vynásobí obraz 2^{shift} .

Neplatný příkaz bude ignorován a bude se pokračovat v provádění. Před uložením do souboru musejí být všechny pixely mezi černou a bílou, budou-li mimo tento rozsah, jejich hodnota v souboru bude nedefinovaná.

Typické použití programu je pro ztluštění nebo ztenčení písmenek, které se provede gaussovským rozmazáním, vynásobením, přičtením a ořezáním.

2.6.8 Makefont

Je shellový skript, který vyrobí font do adresáře `./font/`, přičemž typ fontu je definován v souboru `Fontmap`. Musí se spouštět z adresáře, ve kterém jsou zdrojáky prohlížeče. Je určen k ručnímu spouštění. Na systému musí být nainstalován `Ghostscript`.

2.6.9 Pbm2png

Vyrábí z toku dat ve formátu více zkonkatenovaných souborů `pbm` (což je přesně formát který teče z `ghostscripta` ve skriptu `pdf2html`) sérii `png` 17-násobným vodorovně a 15-násobným svisle převzorkováním. Není určen pro ruční spouštění.

2.6.10 Pdf2html

Je skript, který vyrobí z `*.pdf` nebo `*.ps` sérii `png` souborů. Provádí převzorkování 15-násobné svisle a 17-násobné vodorovně do 8-bitové stupnice šedé. Výstupní gamma je 1.0. Výstupní obrázek obsahuje chunk `gAMA` (informace o gamma obrázku) a tento je nastaven na 1.0. Volá program `pbm2png`.

2.6.11 Wb02links

Slouží k převodu souboru `font_cache.dat` z `wb0` na soubor `png` obrázků vhodných jako font pro `Links`. Musí se nainstalovat `wb0`, pak pustit `wb0 -h100` nebo kolik chceme (s tímto nastavením bude výsup vysoký 100 pixelů) a `wb0` vytvoří ve svém pomocném adresáři soubor `font_cache.dat`.

Pak pustíme `wb02links`, přičemž `font_cache.dat` musí být v aktuálním adresáři. Vygenerují se `png` obrázky do adresáře `./font/`.

2.7 Popis datových souborů

2.7.1 Tabulky znakových sad

V adresáři `Unicode` se nacházejí tabulky znakových sad. Slouží k překladu z různých znakových sad do unikódu. Podmínkou pro znakovou sadu je, že musí být osmibitová, tedy obsahovat maximálně 256 znaků. To je důvod, proč do `Links` nejdou přidat tabulky například pro čínštinu nebo japonštinu.

Tabulky jsou uloženy v souborech s příponou `.cp`. Soubory jsou v textovém formátu, který se následně přeloží do zdrojových textů jazyka C. Formát souboru je následující:

- Komentáře se uvozují znakem „#“ (vězení) a končí koncem řádky. Komentáře se včetně uvozujícího znaku „#“ ignorovány. Od teď budu řádky pojmenovávat první, druhý, třetí, ..., tím se bude myslet po odstranění komentářů.
- Prázdné řádky se ignorují.
- Na prvním řádku je text, který má uživatel vidět v menu při výběru kódování. Tedy například:
ISO 8859-2
- Na druhém řádku je seznam řetězců, pod kterými je možno tabulku hledat v programu. Řetězce jsou uloženy v uvozovkách a odděleny čárkou a mezerou. Za posledním řetězcem na řádku není čárka. Řetězců může být libovolně mnoho, ale musí být na jedné řádce. Tedy například:
"ISO-8859-2", "8859-2", "latin2", "iso-latin2", "iso8859-2"
- Další řádky již obsahují překlady jednotlivých znaků. Na každém řádku je od začátku řádku napsáno hexadecimální číslo znaku uvozené „0x“, v rozsahu od 0 do 255. Tedy například „0x3e“. Pak následuje jeden nebo více tabelátorů či mezer a hexadecimální číslo unikódového znaku uvozené opět „0x“.
- V překladové tabulce nemusí být uveden překlad pro všechny znaky. Znaky, které nejsou v tabulce uvedeny se budou překládat 1:1, tedy na stejná čísla v unikódu. Pokud u znaku není uveden unikódový překlad (je uvedeno pouze číslo ve vstupním kódování), znak se též mapuje 1:1.
- Pořadí znaků s překlady může být libovolné (tedy znaky nemusí být nutně uvedeny ve vzestupném pořadí).

Soubor `7bit.cp` obsahuje tabulku pro 7-bitové ASCII kódování.

V souboru `index.txt` v adresáři `Unicode` je seznam jednotlivých překladových tabulek. V souboru je na každém řádku název souboru s kódováním, ale bez přípony `.cp`. Například:

```
8859_1
8859_2
mac_lat2
utf_8
cp850
```

Pořadí řádků může být libovolné.

Skripty `gen-cp` a `gen-7b` vygenerujete z tabulek soubory:

- `codepage.inc`
- `codepage.h`
- `uni_7b.inc`

V souborech s příponou `.lnx` jsou tabulky pro překlad HTML entit do unikódu. Skriptem `gen-ent` se z nich vygeneruje překladová tabulka v jazyce C s názvem `entity.inc`.

Skriptem `gen` se spustí všechny výše uvedené skripty.

Ke spuštění skriptů je potřeba unixové prostředí s programy `awk` a `sed`.

2.7.2 Jazykové překlady

Jazykové překlady fungují následovně. Každý text má číslo. K těmto číslům existují definované konstanty `T_xxx`. Tyto konstanty jsou definované v souboru `language.h`. Soubor `language.h` je automaticky generovaný, takže by se do něj nemělo nic připsávat. Taktéž se pro odkaz na text musí používat pouze makra `T_xxx` a ne čísla, protože čísla se mohou měnit. Protože spousta funkcí očekává řetězec `unsigned char *`, existuje makro `TEXT(číslo)`, které z čísla `T_xxx` udělá pointer na řetězec. S takovým pointerem se nesmí dělat žádné operace krom předání následujícímu makru.

Dále existuje makro `_(unsigned char *, struct terminal *)`, které přeloží řetězec. Prvním argumentem je řetězec, pokud je řetězec získaný makrem `TEXT()`, tak tento řetězec bude přeložen do příslušného jazyka. Pokud je řetězec získaný jinak (například „xxx“, nebo naalokovaný na haldě, nebo cokoliv jiného), tak makro `_()` vrátí tento řetězec beze změny. Druhým argumentem je terminál, na kterém se bude zobrazovat. Protože v textovém módu může být terminálů víc a každý může mít jinou znakovou sadu, bude se výsledek podle terminálu lišit.

Na místě, kde se v programu řetězec používá, se použije `TEXT(T_xxx)`, pokud se řetězec bude předávat `bfu` vrstvě (menu, dialogy, atd.) — `bfu` vrstva sama volá makro `_()`. Pokud se řetězec bude tisknout rovnou (v textovém či grafickém módu), použije se `_(TEXT(T_xxx), term)`.

V adresáři `intl` se nacházejí soubory jazykových překladů v této části popíšeme strukturu těchto souborů.

Překlady jsou uloženy v souborech s příponou `.lng`. Názvy souborů jsou odvozeny od názvu příslušného jazyka. Referenčním překladovým souborem je `english.lng`, od kterého se ostatní překlady odvíjí.

Soubory `.lng` jsou textové, řádkově orientované a mají následující strukturu:

- Na začátku každého řádku je textová konstanta začínající „T_“, pak následuje čárka (jako oddělovač) a pak řetězec v uvozovkách. Na konci řádku je opět čárka (nezapomeňte ji tam dát!!!). Každý řádek znamená přiřazení příslušného textu dané konstantě. Místo řetězce může být též `NULL`, což znamená odkaz na příslušný řetězec v `english.lng`. V souboru `english.lng` pochopitelně `NULL` nikde být nesmí.
- Na prvním řádku je konstanta `T_CHAR_SET`, které je přiřazeno kódování, v jakém je příslušný jazykový překlad proveden. Název kódování (znakové sada) je stejný jako v příslušné překódovací tabulce pro znakové sady (viz. kapitola o tabulkách znakových sad). Všechny řetězce v překladovém souboru musí být v tomto kódování.
- Na druhém řádku je konstanta `T_LANGUAGE`, které je přiřazen název daného jazyka, jak se má uživateli zobrazovat při výběru jazyka.
- Poté následuje blok řádků s konstantami `T_xxx`, kterými se odkazuje ve zdrojácích na příslušný text. Za každou konstantou je uveden překlad v daném jazyce.
- Nakonec následuje blok řádků `T_HK_xxx`, což jsou horké klávesy v menu. V příslušném řetězci je uvedeno vždy pouze jedno velké písmeno. Na horké klávesy se opět ve zdrojácích odkazuje touto konstantou (`T_HK_xxx`)

V souboru `index.txt` je seznam všech překladových souborů, na každém řádku jeden soubor, soubory jsou zde uvedeny bez přípony. Tedy například:

```
english  
brazilian_portuguese  
bulgarian  
catalan  
czech  
dutch  
estonian  
finnish  
french
```

V adresáři jsou skripty `synclang` a `gen-intl`. Skriptem `synclang` synchronisujete `english.lng` s ostatními jazyky. Po spuštění tohoto skriptu budou ve všech `.lng` souborech řádky se stejnými konstantami, tam, kde byl pro příslušný řádek nějaký překlad, bude stejný překlad i po doběhnutí skriptu, tam, kde překlad nebyl, bude `NULL`.

Skript `gen-intl` vygeneruje z tabulek C zdrojáky pro zakompilování do **Links**. Skript vygeneruje v hlavním adresáři **Links** soubory `language.h` a `language.inc`.

Pokud přidáváte nový řetězec, který se má přeložit, přidejte jej nejprve do `english.lng`, pak pusťte skript `synclang`, v ostatních jazycích na příslušné řádce `NULL` nahraďte příslušným překladem a nakonec pusťte `gen-intl`. Pak budete moci přidáný řetězec používat ve zdrojácích.

2.8 Konfigurační soubory

Prohlížeč po svém prvním spuštění vytvoří adresář `.links`, buďto v domovském adresáři uživatele (když je nastavena proměnná prostředí `$HOME`), nebo v adresáři se spustitelným souborem. Do tohoto adresáře se ukládají veškeré konfigurační informace, záložky a historie. Obsahuje tyto soubory:

- `bookmarks.html` — záložky
- `links.his` — historie URL
- `links.cfg` — hlavní konfigurační soubor
- `html.cfg` — konfigurační soubor s HTML nastavením

Soubory jsou čistě textové, tedy se znalostí jejich vnitřní struktury je lze upravovat libovolným textovým editorem.

2.8.1 Záložky

Záložky jsou uloženy ve formátu kompatibilním s prohlížečem Netscape 4.X. Tedy v HTML, kde jsou adresáře realizovány konstrukcí `<DL>`, `</DL>` a položky konstrukcí `<TD>`. Záložky se ukládají jako odkaz (tedy pomocí tagů `<A>`, ``)

Záložky se ukládají v kódování, které si uživatel zvolí. Základní nastavení je UTF-8.

Pokud si uživatel nastaví jiný soubor pro ukládání záložek, záložky se budou ukládat do daného souboru. Do souboru `.links/bookmarks.html` se záložky ukládají při základním nastavení.

Ukázka vnitřní struktury souboru:

```
<H1>Links bookmarks</H1>
```

```
<DL><P>
```



```

<DT><H3>Links</H3>
<DL>
  <DT><A HREF="http://atrey.karlin.mff.cuni.cz/~clock/
twibright/links/calibration.html">Calibration Procedure</A>
  <DT><A HREF="http://atrey.karlin.mff.cuni.cz/~clock/
twibright/links/kalibrace.html">Procedura kalibrace</A>
  <DT><A HREF="http://atrey.karlin.mff.cuni.cz/~clock/
twibright/links/">Links Homepage</A>
  <DT><A HREF="http://atrey.karlin.mff.cuni.cz/~clock/
twibright/links/index_cz.html">Links: domácí stránka</A>
  <DT><A HREF="http://links.sourceforge.net/docs/
manual-0.90-en/">Links - Manual</A>
</DL>
</DL><P>

```

2.8.2 Historie

`links.his` obsahuje historii navštívených URL. Tato historie se použije v dialogu pro zadávání URL. Struktura souboru je velmi jednoduchá. Na každém řádku je napsáno jedno URL.

2.8.3 Konfigurační soubory

Struktura konfiguračních souborů `links.cfg` a `html.cfg` je stejná. Jedná se o textové soubory, kde je na každém řádku uveden název konfigurační proměnné a za ním po mezeře následuje hodnota, případně, pokud se proměnná skládá z více hodnot najednou, mezerami oddělený seznam hodnot. Na pořadí jednotlivých řádků v souboru nezáleží. Komentáře se uvozují znakem vězení „#“ a končí s koncem řádku.

Názvy proměnných tedy nesmí obsahovat mezeru, proto se v názvu místo mezery používá podtržítka. Řetězce se ukládají do uvozovek, aby se do nich dala zapsat i mezerka. Čísla se ukládají klasicky, reálná čísla též klasicky s tečkou. Název kódové stránky se ukládá v textové podobě, stejně jak je uveden v překódovacích tabulkách, ukládá se bez uvozovek. Například `ISO-8859-2`.

Názvy jednotlivých konfiguračních proměnných nemá smysl uvádět, protože s přidáním nové volby do nastavení prohlížeče typicky přibude nová konfigurační proměnná, navíc název je intuitivní. Tedy pouze pro ilustraci:

```

enable_global_resolution 1
js_recursion_depth 1000
js_memory_limit 2k
bookmarks_codepage ISO-8859-2
bookmarks_file "/home/brain/.links/bookmarks.html"
ftp_anonymous_password "somebody@host.domain"
terminal "linux" 1 1 5 ISO-8859-2
terminal "xterm" 0 0 0 ISO-8859-2
user_gamma 0.880000
bfu_aspect 1.000000
aspect_on 1
dither_letters 1

```

Do `html.cfg` se ukládá pouze nastavení z dialogu „HTML nastavení“. Do `links.cfg` se ukládá veškerá zbylá konfigurace.

Nahrávání a ukládání konfigurace se nachází v `default.c`, kde je tabulka struktur `links_options`, ve které jsou popsány jednotlivé konfigurační proměnné, jejich názvy, mezní hodnoty a funkce pro čtení a zápis.

3. Obecně o programování Links

Tuto kapitolu by si měl přečíst každý, kdo do Links bude něco programovat. Jsou zde uvedena pravidla, která je nutno při zasahování do prohlížeče dodržovat.

3.1 C jazyk

Links je psán celý v C, C++ se nepoužívá. **Links** je psán tak, aby byl zkompileovatelný klasickým ANSI C bez GNU rozšíření. Je třeba dát pozor na následující rozšíření, která sice projdou v gcc, ale neprojdou v ANSI C (a kterých byl původní **Links** plný a dostávali jsme na to stížnosti):

C++ komentáře

```
// tohle v cc neprojde
/* je potřeba používat tyto komentáře */
```

Label na konci bloku

```
fn()
{
    nějaký kód....
    label:
}
```

je potřeba nahradit

```
fn()
{
    nějaký kód....
    label:;
}
```

Inicialisace struktur nekonstantními výrazy

```
struct bla {
    int a, b;
};

int a1, a2;

fn()
{
    struct bla x = {a1, a2};
}
```

K otestování přenositelnosti **Links** na ANSI C je třeba ho nahrát na Solarisy nebo Irixu na Malé Straně a napsat (za současného modlení aby na Irixu nespádl slabostí kernel):

```
export CC=cc
./configure
make
```

Pak se bude kód kompilovat ANSI C místo GNU C. Vypíše to spoustu warningů, ale výsledný kód je funkční.

3.2 Přidávání zdrojů

Napišeme `foo.c`. Přidáme do `Makefile.am` do řádku `links_SOURCES foo.c` a na začátek `foo.c` přidáme:

```
#include "cfg.h"
#include "links.h"
```

Po přidání do `Makefile.am` je potřeba spustit `automake` a `autoconf`, které znovu vygenerují `Makefile`, teprve pak se po spuštění `make` bude kompilovat i přidaný `foo.c`.

3.3 Používání `autoconfu`

`configure.in` je shell script, ve kterém jsou makra. Makra jsou procesorem `m4` expandována na další příkazy shellu (jak psát portabilní shell skripty se dočtete v `info autoconf`). Řetězce se v `m4` dávají do [hranatých závorek].

Nejčastěji používaná makra:

- `AC_DEFINE(SYMBOL) AC_DEFINE(SYMBOL, HODNOTA)`
definuje `SYMBOL` v `config.h` (např. `AC_DEFINE(CHCEME_FLEXI_LIBU)`). Symbol se také musí vyskytovat v souboru `acconfig.h`.
- `AC_TRY_LINK(includey, tělo_main, úspěch-skript, neúspěch-skript)`
vyrobí program, který bude na začátku obsahovat `includey`, pak `main(){ a tělo main a pak ukončovací }`. Pokusí se program slinkovat, pokud se to povede, vykoná `úspěch-skript`, jinak `neúspěch-skript`.
- `AC_CACHE_CHECK(hláška, ac_cv_proměnná, skript)`
pokud je `ac_cv_proměnná` v `config.cache`, tak ji nahraje z té cache; jinak vykoná skript. Předpokládá se, že skript nastaví `ac_cv_proměnnou`. Ta je uložena do `config.cache` a při dalším puštění se už skript nevyvolává.

Další makra — viz `info autoconf`.

3.4 Velikosti dat

Nesmí se předpokládat, že `sizeof(int) == sizeof(void *)`. Neplatí to na Alphě. Je možno předpokládat, že `sizeof(int) <= sizeof(void *) <= sizeof(long)`.

`char` je na některých systémech `signed` a na některých (třeba IRIXy na Malé Straně) `unsigned`. Aby se zabránilo chybám (jaké v původním browseru **Links** také skutečně byly), tak se před každý `char` bude psát `unsigned char` bez specifikace se v kódu vyskytnout nesmí, `signed char` se může použít, když je znaménko potřeba.

3.5 Alokace paměti

Nepoužívat `malloc`, `realloc`, `free`. Místo nich používat `mem_alloc`, `mem_realloc`, `mem_free`. Tyto funkce mají navíc kontrolu na memory leaky. Když se vyskytne leak, tak se při ukončení napíše soubor a řádek, kde se příslušná paměť alokovala, a `dumpne core`.

- `mem_alloc` nikdy nevrátí `NULL`, vstup nesmí být 0.
- `mem_calloc` — funguje jako `mem_alloc`, ale vynuluje blok (tedy používat místo `calloc`).
- `mem_realloc` používat místo `realloc`.
- `mem_free` se nesmí volat s argumentem 0.
- `mem_alloc`, `mem_realloc`, `mem_free` není možno používat v `threadech`.

3.6 Chyby

Následující funkce je třeba volat při různých chybových stavech. `f/printf` by se pokud možno nemělo používat vůbec — třeba jednoho dne bude potřeba zpracovávání chyb změnit.

- `void error(unsigned char *str, ...)`
Syntax jako u `printf`. Vypíše chybové hlášení. Používá se k ošetřování různých vnějších chyb, jako třeba, že došla paměť (je voláno přímo z `mem_realloc`). Kód pokračuje za funkcí.
- `void debug(unsigned char *str, ...)`
Vypíše `DEBUG MESSAGE at file:line: str...` a zastaví běh na jednu sekundu, aby bylo možno zprávu přešíst. Syntax je jako u `printf`. Používá se při debugování, neměla by se vyskytnout nikde ve výsledném kódu.
- `void internal(unsigned char *str, ...)`
Pro ošetřování „can't happen“ situací. Funkce vypíše hlášení `INTERNAL ERROR at file:line: str ...`, zastaví browser a způsobí `core dump`. Syntax je jako u `printf`. Pokud si je člověk jist, že nějaká podmínka má platit, ale že by platit nemusela, pokud se někde vyskytl bug, měl by tuto podmínku otestovat, a pokud neplatí zavolat `internal`. Čím dříve se bug zachytí, tím líp se hledá, proto by se tato funkce měla hojně používat. Funkce se může (a měla by se) vyskytovat ve výsledném hotovém kódu.
- `void do_not_optimize_here(void *p)`
Nedělá vůbec nic. Slouží jen k obelhání překladače, aby neprováděl optimalizace. Ano — i `gcc 2.7.2.1` má bugy v optimalizaci.

3.7 Řetězce

Pro práci s řetězci jsou v **Links** k dispozici tyto funkce:

- `unsigned char upcase(unsigned char chr)`
Převede znak na velká písmena.
- `int casecmp(unsigned char *s1, unsigned char *s2, int len)`
Funguje jako `memcmp`, ale ignoruje velikost písmen. Je taky zaručeno (což v `memcmp` není!), že nebude sahat za první neshodující se byte, takže je možné použít k porovnávání začátků řetězců.
- `unsigned char *strncpy(unsigned char *str)`
Zkopíruje řetězec do nově alokovaného místa. Výsledek je nutno po použití uvolnit pomocí `mem_free`.
- `unsigned char *memacpy(unsigned char *str, int n)`
Zkopíruje `n` bytů ze `str` do nově alokované paměti, přidá nulu na konec a vrátí pointer.
- `void add_to_strn(unsigned char **str1, unsigned char *str2)`
Realokuje řetězec `str1` a zkopíruje na jeho konec `str2`. `str1` je pointer na pointer na řetězec - může se změnit při realokaci. Funkce není příliš rychlá, proto je lepší použít následující funkce.

3.8 Řetězce s prealokací

Následující funkce patří k sobě a mohou být používány pouze na řetězcích vytvořených pomocí `init_str()`.

Datová struktura reprezentuje posloupnost znaků, z nichž žádný není nulový (tedy to, co se myslí pod pojmem řetězec v jazyce C). Instance této struktury je definována

ukazatelem a délkou. Délka je vždy ≥ 0 , a vyjadřuje počet těch znaků, které jsou reprezentovány. Ukazatel je vždy různý od NULL. Podíváme-li se do paměti, na kterou ukazuje ukazatel, uvidíme tam obsah této datové struktury, za kterým bude nulový znak (terminátor), a na další obsah této paměti se již nesmí koukat. Ukazatel vždy vznikl z `mem_alloc` nebo `mem_realloc`. Destrukce struktury se provádí tak, že se zavolá `mem_free` na ukazatel. Konstrukce se nesmí dělat pokoutně, musí se použít `init_str()`. Datová struktura je implementována exponenciální prealokací o základu 2. Kód těchto funkcí je v souboru `links.h`.

- `unsigned char *init_str()`
Alokuje řetězec pro použití dalších funkcí. Vrácený pointer reprezentuje prázdný řetězec.
- `void add_to_str(unsigned char **str, int *len, unsigned char *str2)`
Přidá řetězec `str2` na konec řetězce `str`. Řetězec `str` musel být alokovan pomocí `init_str()`.
- `void add_bytes_to_str(unsigned char **str, int *len, unsigned char *str2, int len2)`
Jako `add_to_str`, ale přidá `len2` bytů z adresy `str2`. Byty z adresy `str2` nebudou odalokovány.
- `void add_char_to_str(unsigned char **str, int *len, unsigned char chr)`
Přidá do řetězce `str` jeden znak `chr`.
- `void add_num_to_str(unsigned char **str, int *len, int num)`
Přidá do řetězce `str` číslo `num` v dekadickém zápise.
- `void add_knum_to_str(unsigned char **str, int *len, int num)`
Přidá číslo v „lidsky čitelném zápise“, čili s použitím písmenek „k“ (kilo) a „M“ (mega). Kilo vyjadřuje 1024 a mega vyjadřuje 1048576 (což je $1024 \cdot 1024$). Jinak je funkce stejná jako `add_num_to_str()`.

Funkce se používají tak, že se inicialisuje řetězec funkcí `init_str()`, vytvoří se integerová proměnná a nainicialisuje se nulou. Dalším funkcím se předávají ukazatelé na řetězec a integerovou proměnnou. Po práci s řetězcem se na řetězec zavolá `mem_free()`.

Typické použití těchto funkcí vypadá asi takto. Je to poměrně efektivní, protože funkce dělají prealokaci (exponenciální) a tak nevolají `realloc` při každém přidání.

```
{
    int l = 0;
    unsigned char *str = init_str();
    add_to_str(&str, &l, "Text:");
    add_chr_to_str(&str, &l, ' ');
    add_num_to_str(&str, &l, 10);
    printf(str);
    mem_free(str);
}
```

Toto je špatně:

```
{
    int l = 5;
    unsigned char *str = stracpy("12345");
    add_to_str(&str, &l, "bla");
}
```

Funkce `stracpy` nedělá potřebnou prealokaci, takže výsledkem bude vystřílená paměť.

3.9 Seznamy

Pokud nějakou strukturu chceme řetězit v seznamu, musíme zajistit, aby první dva prvky té struktury byly pointery na následující a předchozí položku (*next* a *prev*). Pokud budou položky *next* a *prev* na jiném místě, bude se do struktury střílet! Příklad:

```
struct polozka_seznamu {
    struct polozka_seznamu *next;
    struct polozka_seznamu *prev;

    /* a tady jsou už další volitelné položky */
    int a, b;
    unsigned char c;
}
```

Hlavu seznamu deklarujeme takto:

```
struct list_head hlava_seznamu = { &hlava_seznamu, &hlava_seznamu };
```

Pro práci se seznamy slouží následující makra:

- `add_to_list(hlava_seznamu, položka_seznamu)`
Přidá položku do seznamu těsně za hlavu, první argument musí být pointer na `struct list_head`, druhý argument je pak položka seznamu.
- `add_at_pos(prvek_v_seznamu, položka_seznamu)`
`prvek_v_seznamu` je už zařazen v nějakém seznamu, makro přidá položku `položka_seznamu` do seznamu za příslušný prvek.
- `foreach(proměnná, hlava_seznamu)`
Expanduje se jako `for` cyklus, který pro proměnnou `proměnná` projde všechny prvky seznamu. Tělo cyklu nesmí smazat aktuální prvek.
- `foreachback(proměnná, hlava_seznamu)`
Jako `foreach`, ale projde seznam pozpátku.
- `free_list(hlava_seznamu)`
Zavolá `mem_free` na všechny prvky seznamu a seznam vyprázdní (tedy odstraní ze seznamu všechny prvky, takže ze seznamu zůstane pouze hlava).
- `int list_empty(list_head)`
Vrátí 1, pokud je seznam prázdný, jinak vrátí 0.

3.10 Jak používat grafiku (textový vs. grafický mód)

Links má běžet v textovém i grafickém módu. Zkompilovat jde buď jen pro textový mód nebo pro textový i grafický mód (pomocí `./configure --enable-graphics`). V grafickém módu se použít s parametrem `-g`.

Kód by měl být psán tak, aby uživatelé, kteří chtějí pouze textový prohlížeč, nebyli zatěžováni spoustou grafických funkcí. Textový prohlížeč by měl zůstat rozumně malý. Pokud je **Links** zkompilován pro textový i grafický mód, je definováno makro `G`. Proměnná `F` určuje, zda se právě běží v textovém(0) nebo grafickém(1) módu. Pokud byl browser zkompilován pouze pro textový mód, je `F` makro, které je definované na hodnotu 0. Typický kód, mající se rozdějit podle módu, vypadá takto:

```
if (!F) {kód pro textový mód...
#ifdef F
} else {kód pro grafický mód...
#endif
}
```

Existují další makra pro usnadnění rozhodování:

- `gf_val(x, y)`
Vrátí hodnotu `x`, pokud běží browser v textovém módu, a `y`, pokud běží v grafickém.
- `NO_GFX`
Zavolá interní chybu, pokud běží browser v textovém módu. Lepší zavolat interní chybu, než pak spadnout při sahání na neinicializované struktury...
- `NO_TXT`
Zavolá interní chybu, pokud běží browser v grafickém módu.

3.11 Psaní javascriptu

Při konfiguraci **Links** (před kompilací) je možno vypnout javascript. Proto je potřeba kód javascriptu kompilovat pouze, když je javascript zapnutý. K tomu slouží makro `JS`. Veškerý kód javascriptu tedy musí být uzavřen mezi `#ifdef JS` a `#endif`. Tedy takto:

```
#ifdef JS
    /* kód, který se provádí při zapnutém javascriptu */
#endif /* JS */
```

Pokud některé javascriptové funkce nebo kusy kódu musí být v kódu i při vypnutém javascriptu, použije se opět podmíněný překlad. Jelikož taková funkce při vypnutém javascriptu obvykle nic nebude dělat, kód bude vypadat například takto:

```
#ifdef JS
void jsint_scan_script_tags(struct f_data_c *fd)
{ /* kód, který se provádí při zapnutém javascriptu */ }
#else
void jsint_scan_script_tags(struct f_data_c *fd)
{
}
#endif
```

3.11.1 Debugování javascriptu

Podle definovanosti konstant `DEBUZIM` a `BRUTALDEBUG` se vypisují nebo nevypisují hlášky o tom, jak probíhá analýza. Závislost na ničem není. Podobně fungují i makra `debug` (původní `debug` oddefinováno v `ipret.c` a `ns.c`) a `idebug` (v `builtin.c`, `context.c`). Buďto jsou definována jako `printf`, nebo se místo nich vloží prázdné místo, normálně jsou vypnuté. Pak jsou tu konstanty `DEBUGMEMORY` — podle ní se nastavuje komentář k argumentům uloženým na bufferu, `DEBUZ_KRACHY` — při internalu a `js_erroru` vypeče do `links_debug.err` údaje o místě, kde k chybě došlo. Konstanta `DEBUZ_KRACHY` je poněkud nebezpečná a v normálním provozu by neměla být definována!

U konstanty `DEBUGMEMORY` je důležité, aby byla nastavena jenom když se alokuje pomocí `debug_mem_alloc` a freeuje pomocí `debug_mem_free`, proto je kolem nich v `struct.h` `#ifdef` na to, jestli je `DEBUGLEVEL` rovna 2.

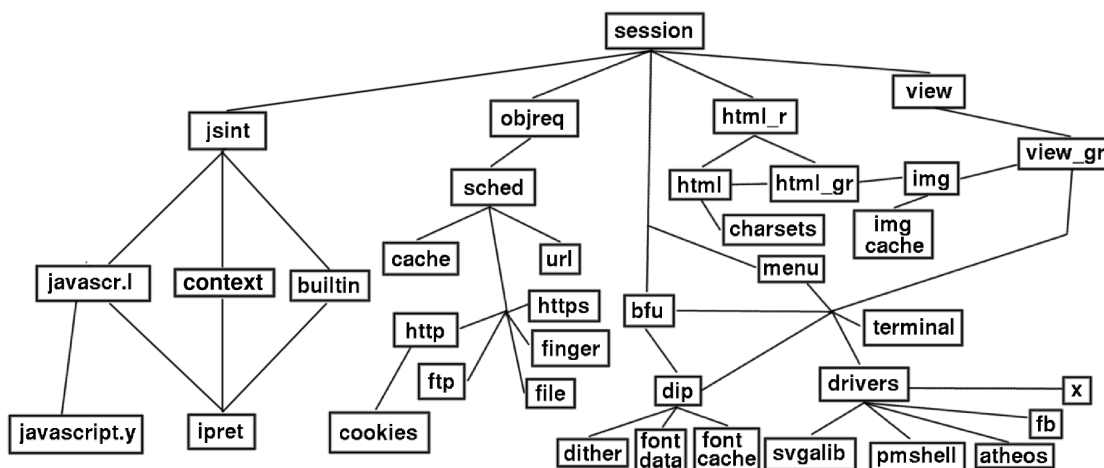
3.12 Formáty obrazových dat

Formát obrazových dat je charakterizován jednotlivými přítomnými kanály, jejich hloubkou, paměťovou organizací a gammou (kromě alphy, která nemá gammu). Fotometrickou reprezentací dále rozumějme veličinu přímo úměrnou množství fotonů, které vycházejí z daného pixelu na monitoru uživatele (nikoliv tedy množství světla na scéně

při pořizování snímku nebo množství světla, které umělec zamýšlel, aby vycházelo z monitoru uživatele). Fotometrická reprezentace budiž definována pouze tehdy, je-li prohlížeč správně okalibrován na příslušný display.

- **alpha (8 bitů na složku)** je výstupní formát generátoru písmenek ve fotometrické reprezentaci.
- **red, green, blue (16 bitů na složku)**, data úměrná osvětlení vycházejícímu z monitoru uživatele je formát vystupující z `img.c` do `dither.c` při kreslení obrázků. Je to také výsledný formát při dekódování obrázků v případě, že obrázek nemá alpu. Pokud se škáluje, výstupem škálovače je také tento formát. Hodnoty jsou ve fotometrickém prostoru.
- **red, green, blue (8 bitů na složku)**, data vycházející z dekodéru obrázků, pokud obrázek má hloubku do 8 bitů na složku. Tato data mají takovou gammu, jakou měla v obrázku. Záhy se gamma korigují na fotometrickou reprezentaci 3x16 bitů a provede se na nich případná změna velikost a nakonec ditherování.
- **red, green, blue, alpha (8 bitů na složku)**, data vycházející z dekodéru obrázků, pokud obrázek má hloubku do 8 bitů na složku. Tato data mají takovou gammu, jakou měla v obrázku. Záhy se gamma korigují a současně podloží pozadovou barvou na fotometrickou reprezentaci 3x16 bitů a provede se na nich případná změna velikost a nakonec ditherování.
- **red, green, blue (16 bitů na složku)**, data vycházející z dekodéru obrázků, pokud obrázek má hloubku 16 bitů na složku. Tato data mají takovou gammu, jakou měla v obrázku. Záhy se gamma korigují na fotometrickou reprezentaci 3x16 bitů a provede se na nich případná změna měřítka a nakonec ditherování.
- **red, green, blue, alpha (16 bitů na složku)**, data vycházející z dekodéru obrázků, pokud obrázek má hloubku 16 bitů na složku. Tato data mají takovou gammu, jakou měla v obrázku. Záhy se gamma korigují a současně podloží pozadovou barvou na fotometrickou reprezentaci 3x16 bitů a provede se na nich případná změna měřítka a nakonec ditherování.

4. Členění do modulů



Obrázek znázorňuje rozdělení programu na jednotlivé moduly (obdélníčky s názvy) a jejich vzájemnou komunikaci (čáry vedoucí mezi obdélníčky). Obrázek je kvůli pře-

hlednosti zjednodušen. Názvy modulů jsou většinou odvozeny od jmen jednotlivých C zdrojáků. Pokud tedy chcete vědět soubor, ve kterém se daný modul nachází, s největší pravděpodobností to bude soubor `jméno_modulu.c`.

4.1 Select smyčka

Základem programu je **select smyčka** nebo též **scheduler**. V této části se plánují všechny události, které mají nastat, a odtud se volají jednotlivé části prohlížeče. Select smyčka je kooperativní scheduler. Všechny části prohlížeče musí být napsány tak, aby v nich řízení nezůstávalo příliš dlouho, protože by nemohly být volány jiné části, což by se například projevovalo nereagováním na pokyny uživatele. Select smyčka se nachází v souboru `sched.c`, na obrázku zobrazena není, protože funkce scheduleru jsou volány ze všech částí prohlížeče, což by obrázek učinilo nepřehledným. Celý život linkse se odehrává ve funkci `select_loop`, která pomocí funkce `select` čeká na různé události.

4.2 Session

Modul **session** zajišťuje management formátování, natahování a zobrazování dokumentů a volání javascriptu. Modul též obsahuje cache na zformátované dokumenty. Zdrojové texty najdete v souboru `session.c`. Tento modul žádnou z výše popsaných funkcí sám nevykonává, pouze se stará o jejich správu a volá další moduly, aby práci vykonaly.

4.3 Object requester

Ke stahování objektů slouží **object requester**, v obrázku je vyznačen jako modul **objreq**, zdrojový kód je umístěn do souboru `objreq.c`. Object requester se stará o vyžadování dokumentů ze sítě, přesměrování a stahování. Samotné stahování souborů nevyřizuje, pouze zařazuje požadavky do fronty scheduleru. Object requester a všechny části pod ním pracují pouze s celými soubory. To znamená, že soubory nijak neinterpretují, nedekódují a podobně. Pokud kterákoliv jiná část prohlížeče chce stahnout nějaký soubor, zavolá právě funkci `object requesteru`.

4.4 Sched

Modul **sched**, který se nachází v souboru `sched.c`, je scheduler requestů. Úzce spolupracuje s `object requesterem`, od kterého přijímá requesty a vyřizuje je. Na druhé straně spolupracuje s `cache` na stažené soubory (modul **cache**), přidává do ní nové soubory, pokud se soubor nachází v `cache`, vrátí ho z `cache`. Scheduler requestů rozděljuje požadavky (které nejsou vyřízeny `cache`) mezi jednotlivé protokoly: `http`, `https`, `ftp`, `finger`, `file`. Scheduler také spolupracuje s modulem **url**, s jehož pomocí dekoduje URL a zjišťuje, jaký protokol má požadavek vyřídit.

Chování scheduleru requestů ovlivňují uživatelsky nastavitelné parametry v „nastavení sítě“: maximální počet spojení, maximální počet spojení k jednomu stroji, počet pokusů, atd.

4.5 Cache

Modul **cache** se nachází v souboru `cache.c`. Tento modul je souborová cache. Do ní se ukládají již stažené soubory. Chování `cache` ovlivňuje parametr „velikost cache“, který si uživatel může nastavit. Modul se stará též o vyhazování souborů z `cache`.

Souborová cache používá strategii LRU, ale poněkud modifikovanou — dříve vyřazuje velké soubory (zvýhodňuje malé soubory). Cache se po přidání nového souboru a při

určování, které soubory vyřadit, prochází na tři průchody. V prvním průchodu se prochází soubory od nejstaršího směrem k mladším a dokud velikost přesahuje určitou mez, soubory se označí k vymazání. Ve druhém průchodu se prochází cache od nejmladších souborů a pokud se nalezne označený soubor, který by šel ještě ponechat v cachi, aniž by byla překročena velikost cache, soubor se odznačí. Nakonec ve třetím průchodu se označené soubory z cache odstraní. Při stahování do souboru (funkce „download“ nebo „stahování“) se soubory do cache neukládají, pokud jejich velikost přesahuje čtvrtinu maximální velikosti cache.

4.6 Url

Modul **url**, který se nachází v souboru `url.c`, je soubor funkcí pro práci s URL. Obsahuje funkce pro parsování URL: rozdělení na části protokol, port, uživatel, heslo, server, adresář, ... Dále obsahuje funkce na spojování URL (vytváření absolutního URL z relativního) a tabulku protokolů (které funkce se mají zavolat na který protokol, parametry jednotlivých protokolů). Funkce z modulu **url** jsou volány i z jiných modulů, zejména funkce pro parsování a spojování URL, tyto závislosti v obrázku pro jednoduchost nejsou nakresleny.

4.7 Http, https, finger, ftp, file

Moduly **http**, **https**, **finger**, **ftp** a **file** najdete v stejně pojmenovaných `*.c` zdrojových souborech. Tyto moduly zajišťují rozhraní jednotlivých protokolů. Mají za úkol stáhnout konkrétní soubor a komunikace s příslušnou protistranou je již na nich. Modul **http** spolupracuje ještě s modulem **cookies**, v případě přijetí HTTP cookie.

Všechny moduly pro protokoly používají modul **connect** (v souboru `connect.c`, na obrázku není pro jednoduchost uveden). Modul obsahuje funkce pro navazování spojení a pro jednoduché I/O z/do bufferu.

4.8 Cookies

Tento modul najdete v souboru `cookies.c`. Obsahuje funkce pro práci s cookies. V modulu je seznam všech cookies a seznam všech domén a funkce, které s cookies pracují: funkce pro smazání cookie, vytvoření cookie, přijetí cookie, odmítnutí, poslání všech cookies pro daný server. Tento modul je částečně využíván i rozhraním javascriptu, protože javascript umožňuje práci s cookies, tento vztah v obrázku pro zjednodušení není zakreslen.

4.9 Jsint

Modul **jsint**, který najdete v souboru `jsint.c`, je rozhraní mezi javascriptem a prohlížečem. Rozhraní javascriptu sahá do většiny částí prohlížeče a zakreslení všech těchto vztahů by učinilo obrázek poněkud méně čitelným, proto jsou v obrázku zakresleny pouze hlavní vztahy mezi `session` a interpretem javascriptu.

Rozhraní zejména obsahuje tzv. **upcally**, neboli funkce, které volá javascript kdykoliv chce modifikovat nebo přistupovat k interním strukturám prohlížeče (dokument, rámy, okna, formuláře, obrázky, odkazy, tlačítka, vnitřní proměnné, ...). Javascript se nikdy nesmí přímo odkazovat ukazateli na konkrétní objekty v prohlížeči. Mohlo by se totiž stát, že daný objekt přestane existovat a javascript se odkáže neplatným pointerem, to by samozřejmě vedlo ke katastrofě. Proto se javascript odkazuje pomocí číselných identifikátorů jednotlivých objektů. Identifikátor je `long`, který v sobě má zakódován typ objektu a pak jednoznačnou identifikaci v rámci daného typu. Pokud se javascript odkáže na neplatný objekt, nic se nestane, `upcall` nic neprovede a vrátí se. V `upcallech` se také

vždy testuje, zda javascript má právo na ten konkrétní objekt přistupovat. Pokud ne, upcall se ukončí (jako v případě odkázání na neexistující objekt).

Některé upcally se volají přímo a přímo vrací hodnotu, jiné je potřeba volat ze select smyčky. Aby mohly upcally volané ze select smyčky vrátit nějakou hodnotu, volají funkce zvané **downcall**. Pomocí downcallů se vrátí hodnota z upcallu do javascriptu. Pokud javascript zavolá upcall ze select smyčky, zablokuje se a čeká na odpověď. Jakmile upcall zjistí požadované informace nebo provede požadovanou činnost, zavolá downcall, který javascript odblokuje, případně mu předá výsledek, a javascript pokračuje dále. Javascript se zablokuje samozřejmě pouze v tom kontextu, který zavolal upcall, ostatní kontexty běží dále.

Poslední součástí rozhraní javascriptu jsou funkce pro vytváření a rušení kontextu javascriptu a pro spouštění javascriptu.

4.10 Javascript.l — lexikální analýza

Modul **javascr.l** je lexikální analyzátor vygenerovaný programem **Flex**. Zdrojový soubor v C je možno najít v `javascr.c`, vstup pro **Flex** pak v adresáři `parser` v souboru `javascr.l`.

Úkolem lexikální analýzy je rozbít vstupní řetězec do sady tokenů, které postupují dále k syntaktické analýze. Tokeny mohou být nemálo typů. Lexikální pravidla byla opsána z normy javascript 1.1 od Netscape Corporation a poněkud upravena s vyhlídkou na snazší syntaktickou analýzu. Např. každé řídicí slovo má svůj vlastní typ tokenu (`while`, `for`, ...). K syntaktické analýze postupují tokeny číslované typem `long` (možná by stačil `int`, ale jelikož při jeho psaní nebylo jasné, jestli nebude potřeba místo čísla vracet pointer, byl vybrán typ, který je konvertovatelný na pointer.

4.10.1 Typy tokenů

Řidícím slovům stačí pouze `long` říkající vše. Jedná-li se o nějaký literál, pak pro boolské literály jsou použity dva různé tokeny (`TRUE` a `FALSE`), pro nulový literál token `NULL`. Numerický literál odesílá ještě v proměnné `yylval` buďto hodnotu celého čísla, nebo pointer na necelé. `yylval` je typu `long` kvůli přetypovatelnosti na `integer` i `pointer`. Řetězce posílají v proměnné `yylval` ukazatel na místo v paměti, kde se řetězec nachází.

4.10.2 Identifikátory

Každému identifikátoru je již při lexikální analýze přiřazen klíč, pod kterým bude v celém programu vystupovat (tento klíč je jednoznačný i pro „vnořené“ identifikátory, tedy ať už se identifikátor vyskytuje jako proměnná a stejně se jmenuje kupř. nějaká metoda nějakého objektu, budou mít tyto klíče stejné. Kde se objekt identifikátorem odkazovaný v paměti nachází se zjišťuje z „adresného prostoru“, který je až záležitostí interpretu mezikódu. Při lexikální analýze tedy vzniká jakýsi „namespace“, který přiřazuje každému identifikátoru nějaký klíč. Indexování proměnných klíčem bylo vybráno proto, že porovnat dva `longy` je podstatně rychlejší, než testovat shodu dvou řetězců (zdržení se realizuje za každý výskyt proměnné ve zdrojovém textu, nikoliv za každý dotaz na ni při interpretaci).

Lexikální analýza též odstraňuje komentáře a tomu podobné pro význam programu nepotřebné věci (konce řádku, zbytečné mezery...).

4.11 Javascript.y — syntaktická analýza

Toto je modul syntaktické analýzy javascriptu. Syntaktický analyzátor je vygenerován programem **Bison**, zdrojový kód v C je v souboru `javascript.c`.

4.11.1 Syntaktická analýza

Syntaktická analýza dostává na vstupu jednotlivé tokeny a má „říct, jak spolu souvisejí“, tedy postavit syntaktický strom podle pravidel gramatiky. Tuto gramatiku jsme téměř opsali opět z normy javascript 1.1. Lexikální i syntaktická analýza předchází vlastní interpretaci, jelikož interpretování „z čisté vody“, tedy z čistého zdrojového kódu právě interpretovaného javascriptu by bylo značně zdoluhavé, navíc bison neposkytuje dostatečně komfortní prostředí pro samotnou interpretaci — bylo by nutné udržovat pohromadě syntaktický analyzátor (který je už sám o sobě dost dlouhý) v jednom souboru s interpretačními pravidly (ta jsou ještě delší). Syntaktický analyzátor pouze podle pravidel opsaných z gramatiky z posloupnosti tokenů postaví syntaktický strom tvaru:
operátor, ukazatel na první argument, ukazatel na druhý argument, ..., ukazatel na šestý argument.

Šest argumentů není nikdy použito (maximum jsou čtyři, návrh natvrdo sestrojil sedmice je podložen těmito argumenty: Sice je v určitém smyslu poněkud marnotratný (průměrný počet potomků ve stromě je mezi dvěma a třemi), navíc není zcela pružný (změna gramatiky může někde vynutit třeba 7 synů), zato je ale poměrně jednoduchý, není potřeba udržovat počet synů a komplikovaně pro ně alokovat a odalokovávat paměť, navíc změna gramatiky by reprezentovala takový zásah do interpretu, že přidání pár slotů na syny by byla nevinná dětská hra (i když samotné přidání nějakých pravidel by rovněž problémem nebylo - postačovalo by přidat příslušná pravidla do javascript.y, eventuálně nové tokeny do javascr.l a změny interpretující funkce do ipret.c).

4.11.2 Generátor mezikódu

Generování interkódu probíhá při syntaktické analýze. Celý výpočet javascriptu je zavřen ve struktuře `struct js_context`, která obsahuje zejména:

- **timer** (interní záležitost) — používá se ke kontrole správnosti naplánování běhu javascriptu.
- **callback** (opět interní záležitost) — ukončí celé interpretování.
- **id okna**, které příslušný javascript obhospodařuje.
- **zámek**, který chrání interpretaci před časově závislou chybou (racem).
- **dva ukazatele na syntaktický strom** — jeden ukazuje do kořene a druhý do právě interpretovaného vrcholu.
- **zásobník argumentů** — pro ukládání výsledků jednotlivých kroků interpretace.
- **zásobník rodičů** - říká, kterými vrcholy se výpočet ve stromě sestupně ubíral, než došlo k našemu zavolání. Tento zásobník obsahuje pouze ty rodiče, kteří ještě nedopočetali, tedy jimž je po ukončení našeho výpočtu potřeba ještě vrátit řízení.
- **addrspace** — pojmenovaný `lnamespace` jako `localnamespace`. Udržuje informace o stavu proměnných podle jim přiřazených klíčů (klíč, typ proměnné, hodnota, ev. ukazatel na ni ...)
- **namespace** — sada dvojic klíč, řetězec identifikující proměnnou. Je ho potřeba udržovat po celou dobu výpočtu, jelikož javascript se může při běhu modifikovat a volat znovu lexikální analyzátor (a tudíž i vše za ním následující).

4.11.3 Fosilie

Funkce terminál a neterminál. V minulosti se jevilo jako dobrý nápad rozlišovat, co je terminál (nemá potomky typu pointer, ale hodnota) a co neterminál (rodič nějakých terminálů nebo neterminálu). Posléze se však ukázalo, že uzly jsou natolik heterogenní, že takovéto dělení je úplně zbytečné. Až bude trocha času na civilizování kódu, tak jednu z těchto funkcí vymažu (možná už jsem to udělal, ale nepamatuju se).

4.12 Builtin

Builtin je modul vestavěných funkcí, objektů a proměnných. Zdrojáky můžete najít v souboru `builtin.c`. Tento modul je volán z interpretu javascriptu, kdykoliv se čte nebo zapisuje vestavěná proměnná. Vestavěný objekt je například matematika, práce s řetězci, datum a další, pak samozřejmě vše co je na stránce, formuláře, odkazy, okna atd. Z pohledu javascriptu se nepozná, zda je vestavěná proměnná, funkce či objekt „uvnitř“ prohlížeče (zda se například jedná o nějaký formulář na stránce a podobně), z hlediska javascriptu je přístup na vestavěné proměnné a funkce jednotný. Pokud se jedná o „vnitřní“ součást prohlížeče, modul `builtin` zavolá `upcall` na přistoupení k danému objektu.

Jak již bylo zmíněno, mezi vestavěné funkce a objekty patří matematika. Součástí matematiky je i počítání MD5 sumy. Kód na práci s MD5 můžete nalézt v souborech `md5.c`, `md5.h`, `md5hl.c`. Tyto soubory nepsal nikdo z týmu projektu, jedná se o externí kód stažený z webu.

4.13 Ipret — interpret mezikódu

Interpret mezikódu, modul `ipret` je nejnáročnější částí celého interpretu. Jeho hlavní část lze najít v souboru `ipret.c`, v souboru `ns.c` je možno najít funkce obsluhující namespace. Interpretace probíhá tak, že funkce `ipret` postupně a organizovaně prohlíží strom. Jediné, co potřebuje vědět, je, v kterém vrcholu teď stojíme, kolikátý argument daného operátora zpracováváme a jaké argumenty už máme spočítané. Když má operátor vypočteny všechny argumenty, spočítá svou hodnotu a vrátí řízení „nadřazenému uzlu“, který jeho výpočet použije jako jeden ze svých argumentů. Důležité je, že argumenty se udržují na zásobníku argumentů a vlastnosti zásobníku zaručují, že ve chvíli, kdy se budeme shánět po svých argumentech je tam budeme mít v pořadí: poslední, předposlední, ... první (pokud nechceme jinak). Stejně tak teoretické vlastnosti zásobníku zaručují, že vždy po ukončení výpočtu v současném uzlu se vrátíme do bezprostředně předcházejícího uzlu, který má ještě zájem počítat (zájem počítat už např. nemá operátor `program`, který již zavolal svůj druhý argument. Funkce `program` vznikla z pravidla: `program->program program`, tedy pouze řetězí jednotlivé bloky programu.

4.13.1 Vztahy k okolí

Interpretace začíná zavoláním `js_create_context`, která vytvoří strukturu `js_context` (popsanou výše). V této funkci jsou do namespace a `addrspac` doplňovány vestavěné funkce.

Když je potřeba interpretování zastavit (dočasně ukončit), zavolá se `callback`. Po tomto zavolání může interpret být ještě probuzen dalším požadavkem.

Je-li potřeba s příslušným skriptem již definitivně skoncovat, zavolá se funkce `js_destroy_context`, která zruší kontext a tím i všechna data se skriptem spojená.

4.14 View a view_gr

Moduly `view` a `view_gr` můžete najít v souborech `view` a `view_gr.c`. Modul `view` má na starost zobrazování dokumentu a pohyb po dokumentu, `view_gr` zobrazuje dokument v grafickém módu. Najdete zde sazeč stránky pro textový i grafický režim, funkce pro handlování uživatelských událostí jako je například kliknutí myši, `scroll` posun stránky, vyplňování formuláře, hledání na stránce, přepínání rámců, otevírání nového okna, ukládání dokumentu. Pak také vyřizování event handlerů javascriptu: `OnClick`, `OnMouseOver`, `OnMouseOut`, ...

Z těchto modulů jsou volány „nižší“ funkce pro čtení klávesnice, čtení myši, kreslení na obrazovku, sázení textu, psaní na terminál a kreslení obrázků.

4.15 `Img` a `img_cache`

Tyto dva moduly zajišťují veškerou práci s obrázky, počínaje stahováním ze sítě a kreslením konče. Modul `img` se nachází v souboru `img.c`, obsahuje funkce pro vkládání obrázků do dokumentu, dekodéry jednotlivých obrazových formátů, které jsou uloženy v souborech `gif.c`, `jpeg.c`, `tiff.c`, `png.c`, `xbm.c` a funkce pro vykreslování obrázků do dokumentu. `img_cache` je cache na dekodéry obrázků, ukládají se do ní dekodéry ve všech stádiích běhu: ještě nespustěné (čekající na stažení obrázku ze sítě), běžící i doběhnuté (hotové dekodované a zditherované obrázky). Obrázkovou cache najdete v souboru `img_cache.c`. Obrázková cache je klasická LRU cache, jejíž velikost si uživatel může nastavit v menu.

Modul `img` je volán také z rozhraní javascriptu `upcall` na změnu zdrojového URL obrázku. Tento vztah pro jednoduchost není do diagramu zakreslen.

4.16 HTML parser: moduly `html_r`, `html_gr`, `html`, `charsets`

V modulu `html` v souboru `html.c` najdete kompletní parser HTML jazyka. V souboru `Skládá se zejména ze spousty funkcí, které parsují jednotlivé HTML elementy. html_r (v souboru html_r.c) je parser a sazeč HTML v textovém módu, html_gr (v souboru html_gr.c) parser a sazeč v grafickém módu. Tyto dva moduly postupně parsují dokument, vytvářejí jednotlivé objekty stránky a ukládají je na stránku. Grafický sazeč stránky volá z modulu img funkce pro vkládání obrázků. html.tbl.c je kód pro sázení tabulek, jak v grafickém, tak i v textovém módu. Ten je volán grafickým a textovým sazečem stránky. Modul html volá modul charset, který zajišťuje překlad kódování znakových sad. Některé další části Links modul charset též volají, ale to není do diagramu pro jednoduchost zakresleno. Chování sazeče HTML je ovlivněno „HTML nastavením“ v menu.`

4.17 Moduly `bfu` a `menu`

Veškeré uživatelsky interaktivní části `Links` jsou soustředěny do modulu `bfu`, který naleznete v souboru `bfu.c`. Jsou zde funkce pro vytváření dialogů, menu, klikátek, message boxů, okýnek s dotazem, s oznámením, okýnek pro zadávání textu. Dále pomocné funkce pro vytváření vlastních dialogů: formátování tlačítek, checkboxů, radio tlačítek, zadávacích políček, . . . V modulu `menu`, který lze najít v souboru `menu.c`, se nacházejí definice všech menu v `Links`.

Tyto moduly volají funkce grafických driverů — pro kreslení na ploch a čar na obrazovku a funkce pro sázení písmen (modul `dip`).

4.18 Grafické ovladače

Grafické ovladače zahrnují moduly `drivers`, `svgalib`, `x`, `pmsshell`, `fb`, `atheos` a `terminal`.

Jednotné rozhraní pro grafické drivery se nachází v modulu `drivers`, v souboru `drivers.c`. Toto rozhraní obsahuje jednoduchá grafická primitiva pro nakreslení čar, vybarvení plochy, zaregistrování bitmapy, nakreslení bitmapy, vrácení barvy, scroll, nastavení ořezávací plochy a další. Modul `drivers` obsahuje ještě simulaci virtuálních zařízení — přepínání virtuálních grafických konzolí v `Links` jako emulace více oken (například pro `svgalib` nebo `framebuffer`). V modulech `svgalib`, `x`, `fb`, `pmsshell`, `atheos` (v souborech `svgalib.c`, `x.c`, `fb.c`, `pmsshell.c`, `atheos.cpp`) se nacházejí implementace grafického rozhraní na jednotlivých grafických systémech.

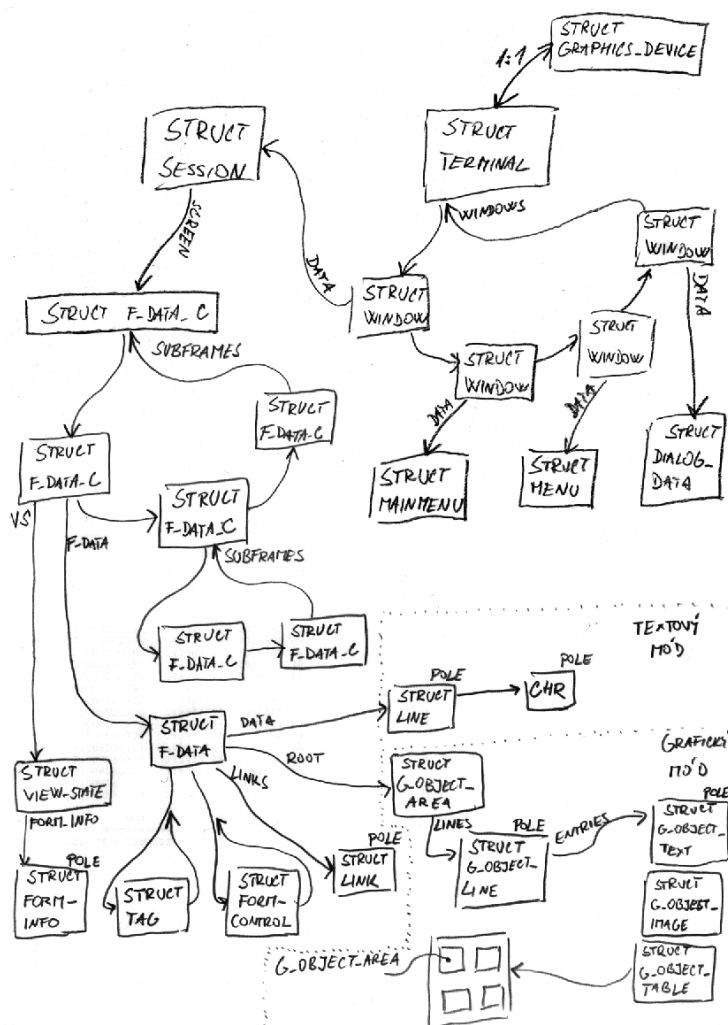
Modul `terminal` (soubor `terminal.c`) obsahuje rozhraní pro výstup na textový terminál: psaní písmen, čtení událostí myši (z `gpm`), čtení z klávesnice, inicialisace, ukončení,

mazání obrazovky, ... Grafický driver **fb** a modul pro výstup na terminál používají ještě funkce pro čtení klávesnice ze souboru `kbd.c`. Tento modul pro jednoduchost není zakreslen v diagramu, modul obsahuje inicialisaci klávesnice a funkce pro čtení kláves.

4.19 Dip, dither, font_data a font_cache

Modul **dip** provádí rendering písmen a bitmap v grafickém módu, tisk textu, aplikaci gamma korekce, modul **dither** provádí ditherování pomocí Floyd-Steinbergova algoritmu. V souboru `dither.c` jsou nízkoúrovňové ditherovací rutiny, v souboru `dip.c` rutiny na gamma korekci, alfa kanál, funkce pro nahrávání a výběr fontů, tisk řetězců. Součástí souboru `dip.c` je i **font_cache** — cache na nascalovaná, zditherovaná písmenka s již aplikovanou gamma korekcí. Cache používá metodu LRU pro uvolňování místa. Bitmapy fontů, neboli modul **font_data**, jsou uloženy v souboru `font.include.c` ve formě PNG souborů uložených v C zdrojáku.

5. Vnitřní struktury a jejich komunikace



Na obrázku nahoře je vidět důležité struktury prohlížeče a jejich vzájemnou provázanost.

Obdélníčky na obrázku znázorňují jednotlivé struktury, šipky naznačují vazby mezi nimi — tedy ukazatele. Pokud je u struktury napsáno „pole“, znamená to, že ukazatel míří ne na strukturu, ale na pole struktur. Názvy u šipek znamenají název příslušného

pointeru. Šipky do kruhu znázorňují kruhový seznam (jako například kruhový seznam struktur `window`, na který ukazuje ukazatel `windows` ze struktury `terminal`)

Struktura `terminal` odpovídá jednomu oknu prohlížeče. Tedy po zavolání funkce „Otevři nové okno“ se vytvoří nová struktura `terminal`. Struktura tedy odpovídá jednomu oknu v okenním systému nebo jedné virtuální konzoli.

`graphics_device` je přítomna pouze v grafickém režimu a je ve vazbě 1:1 se strukturou `terminal`. Ze struktury `terminal` ukazuje pointer `windows` na kruhový seznam struktur `window`.

Struktura `window` vyjadřuje plochu, na kterou lze kreslit a která dostává události od klávesnice a myši. Takovouto plochou může být menu (`struct menu`), hlavní menu (`struct main_menu`), dialog (`struct dialog_data`) nebo plocha, kde se zobrazuje HTML stránka (`struct session`). Ukazatel `data` míří na strukturu, která dále popisuje tuto plochu. Plochy jsou navrstveny nad sebou, události propadávají od nejvyšší k nejnižší vrstvě, dokud ji nějaká vrstva nezachytí a nezpracuje.

`struct session`, jak již bylo řečeno, popisuje oblast, kde se zobrazuje HTML stránka. Teoreticky koncepce **Links** umožňuje více `session` na jednom terminálu, v praxi je na jednom terminálu vždy jedna `session`.

`struct f_data_c` reprezentuje jeden rám na HTML stránce. Jelikož rámy v HTML mohou mít podrámy, podrámy mohou obsahovat opět další podrámy atd., struktura obsahuje kruhový seznam podrámů `subframes`, jak znázorňují šipky do kruhu.

Ze struktury `f_data_c` míří ukazatel `vs` na strukturu `view_state`, která obsahuje dynamické informace z příslušného `f_data_c`. To je například pozice na stránce, pozice ve vyhledávání a podobně. Mezi dynamické informace také patří stav formulářů (vyplněné hodnoty, stav tlačítek atd.), které jsou reprezentovány polem struktur `form_info`, na které z `struct view_state` míří stejnojmenný ukazatel. Každé `form_info` odpovídá stavu jednoho elementu formuláře.

Statické informace o stránce jsou umístěny ve `struct f_data`, která se ukládá v dokumentové cachi. Struktura `f_data` obsahuje pole odkazů (struktur `link`), kruhový seznam struktur `tag`, což jsou tzv. anchors (místa v dokumentu, na která se dá odkazovat, vzniknou konstrukcí `...`). Kruhový seznam struktur `form_control` zachycuje statické informace o formulářích. Každá struktura `form_control` odpovídá jednomu elementu formuláře.

Statický popis stránky se liší pro textový a pro grafický mód. V textovém módu je stránka jednoduše popsána polem `line` řádek na stránce, kde se každá řádka skládá z pole znaků `chr`.

V grafickém režimu je stránka popsána soustavou grafických objektů `g_object_area`, `g_object_line`, `g_object_image`, `g_object_text`, `g_object_table`. `g_object_area` popisuje obdélníkovou plochu v dokumentu. Obsahuje pole řádků `g_object_line`, kde každá řádka může být složena z textů (`g_object_text`), obrázků (`g_object_image`) a tabulek (`g_object_table`). Tabulka je opět soustava objektů `g_object_area`, které se opět skládají z řádek atd.

6. Princip jednotlivých částí

6.1 Fonty

Fonty jsou obdélníky s barvou pozadí, na kterých je nakreslené písmeno barvou popředí. Obdélníky se nemohou překrývat. V následujících řádkách textu se obdélníky na výšku dotýkají. Při jejich tisknutí se nepoužívají žádné fontovací funkce X, `svgalib`, `pm-`

shell ani ničeho podobného. Používají se pouze funkce pro tisk bitmap. Bitmapy jsou renderovány programem (čímž je zaručeno, že bude moci být 100% kontrolováno, jak budou vypadat) přímo ve formátu vhodném pro dané výstupní zařízení (záleží na barevné hloubce) a pak jsou kresleny na obrazovku.

Základem pro fonty jsou obrázky písmenek, tak jak mají vypadat. Písmo je bílé a papír černý. Mezi tím jsou stupně šedé, protože písmenko je převzorkované z velkého rozlišení. Tyto šedotónové obrázky jsou zakompilovány ve spustitelném souboru links a mají výšku typicky 112 pixelů, nicméně mohou ji mít libovolnou. Souvislý text se skládá z řádek, což jsou stejně vysoké pruhy textu. V obrázku písmenka (toho v adresáři font) je horní okraj (horní hrana nejspodnější pixelové řady) ztotožněn s hraniční přímkou těchto pruhů, a dolní hrana (dolní okraj nejspodnější pixelové řady) je stotožněna s hraniční přímkou o jednu řadu níže. Hraniční přímka je útvar o nulové tloušťce.

Překrývání ani ligatury se nepodporují z důvodu, že přinášejí málo vizuálního zlepšení za cenu zavlečení obtížných problémů do sázení, tiskání, scrollování, rozsvěcování části textu a podobně. Italické písmo se nepodporuje z důvodu, že by mezera mezi písmeny z důvodu překryvu byla někdy zaměnitelná s mezerou mezi slovy.

Partie, které jsou v obrázcích bílé, budou kresleny „inkoustem“, partie černé budou kresleny „papírem“. Části s barvou mezi tím budou kresleny lineárně (ve fotonovém prostoru) mezi tím (tedy např. je-li png 8-bitové a barva z rozsahu 0 (černá) – 255 (bílá) je 12, pak bude smícháno 12/255 inkoustu a (255–12)/255 papíru). Když jsou písmenka barevná (což se nedoporučuje, protože to má za následek zbytečné zvětšení PNG), tak se zkonvertují na černobílé podle aproximační formulky vyjadřující přibližný jas vnímaný člověkem.

Předlohy písmenek jsou hodně vysoké obrázky, například 112 pixelů. Pro kreslení řekněme 16 pixelů vysokého písmenka je třeba bitmapu zmenšit. Na to se použije algoritmus, který namapuje výstupní pixely na vstupní (oboje pixely bere jako obdélníky) a v každém výstupním pixelu vypočítá průměrnou barvu na základě informací o pixelech vstupních. Dělá se to převzorkováním v jednom směru a následným převzorkováním v druhém směru. Pořadí se volí tak, aby mezivýsledek měl tu menší plochu z obou možných variant.

Písmenka, která jsou moc titěrná, takto vyjdou správně - tedy tak, jak by se jevila, kdyby byla natištěna na papíře a snímána idealizovaným scannerem s dokonalým objektivem a CCD prvkem se čtvercovými pixely, mezi nimiž nejsou mezery. Ovšem, jak je možno vidět v televizi při záběru na dokument s malým písmem (a jak také vyplývá z naměřených funkcí vnímání kontrastu v závislosti na prostorové frekvenci u lidského oka), když jsou písmenka malá, vypadají šedivě a nevýrazně. Je to způsobeno faktem, že malé černé a bílé detaily vytvoří šedivou, která způsobuje subjektivní vjem sníženého kontrastu. Tomuto se předchází následnou korekcí, která slučuje filtr pro zvýšení kontrastu (s oříznutím na černou a bílou, samozřejmě) a filtr pro zvýšení ostrosti. Parametry filtru byly voleny empiricky, aby to dobře vypadalo. Byly dělány také pokusy s převzorkováním ostrou dolní propustí namísto mapování obdélníků, ale výsledek byl špatný (ačkoliv metoda byla ta jediná matematicky správná pro převzorkování velkého obrazu na malý tak, aby se reprezentovatelné frekvence zachovaly beze změny amplitudy i fáze).

Pro tento kombinovaný filtr je použita matice 3x3 bodů natvrdo zakódovaná do algoritmu (a optimalizovaná na základě symetrie). Filtr se používá do výšky písmenek 32 pixelů, dále se již nepoužívá.

Zdrojové PNG soubory jsou uloženy ve spustitelném souboru. Program `generate_font` slouží pro výrobu `font_include.c`, jenž obsahuje pole bytů zapsané v syntaxi jazyka C, ve kterém jsou uloženy po sobě v pořadí podle čísla znaků všechny potřebné PNG, a pomocnou tabulku umožňující nalézt písmeno a zjistit, jak je jeho PNG dlouhé. PNG se v paměti rozkóduje pomocí `libpng` a putuje dále do zpracování.

Výhody celého tohoto přístupu jsou následující: pro kompletní funkci prohlížeče je potřeba jeden soubor a příslušné nainstalované knihovny (v případě, že je zkompilevan staticky, je třeba jen jeden spustitelný soubor, který může být teoreticky puštěn i místo initu), nevznikají tedy potíže s tím, kam umístit data na různých operačních systémech. Vzhled a podporované fonty nejsou platformně a konfiguračně závislé. Písmenka se dají snadno přidávat (stačí získat vzor písmenka ve tvaru obrázku bez omezení na výšku a šířku, ten je možno získat i z tištěného materiálu nascanováním, úpravou obrazu GIMPem a uložením ve formátu PNG) pouhým překompilováním prohlížeče (což je standardní instalační procedura). Písmenka jsou dobře čitelná i při malé pixelové výšce (na rozdíl od neantialiasovaných fontů, které jsou při nízkých výškách téměř nebo zcela nečitelné).

Písmenka z PNG jsou standardně uložena 8-bitová šedá s gammou 1.0 což odpovídá tomu, že jsou přímo úměrná světlu vycházejícímu z obrazovky. Použijeme-li je jako alpha masku na 16-bitové barvy, vzniknou nám čísla 0 až $255*255*257$. K tomuto číslu přičteme 127 a vydělíme to 255, čímž dostaneme číslo v rozsahu 0– $255*257$ (65535).

Různé druhy písma jsou dělány tak, že každý druh písma má extra sadu obrázků. Podtržené písmo se dělá přikreslováním podtrhávací čáry přes font: při kreslení znaků se nastaví clip nad podtrhovací čáru, pak se nakreslí podtrhovací čára a clip se nastaví pod podtrhovací čáru a nakreslí se písmenka znovu.

Jinak má každý font jméno ve formátu `family-weight-slant-adstyl-spacing`. `Family` může být libovolné jméno (fonty ho mají malými písmeny, nehledí se na velká a malá písmena) složené jen z písmen a podtržítok. `weight` je „bold“ nebo „medium“, `slant` je „italic“ nebo „roman“, `adstyl` je „sans“ nebo „serif“, `spacing` je „mono“ nebo „vari“.

Když se hledá vhodný font, tak se napřed chce, aby seděly všechny položky co označují font. Tím se najde v souboru `font/catalogue` jediný font, který se dá na první místo a vyškrtně se. Pak se zruší postupně položky `family`, `adstyl`, `weight`, `spacing`, `slant`. Po zrušení každé položky se projde katalog a vyhovující fonty se nasází na další místa. Nakonec je průchod se zrušenými všemi položkami, a tam musí projít všechny fonty, takže se všechny fonty tímto seřadí do žebříčku oblíbenosti pro zadaný požadavek na font. To se nacpe do struktury `struct font`, a když se hledá písmenko, tak se po hitparádě jde zezhora dolů a dokavad se nenajde, tak se jde. Když se nenajde ani na posledním místě, tak se vrátí znak reprezentující chybějící písmenko (lebka, kaňka a podobně).

6.2 Gamma

Do monitoru vstupuje elektrický signál a vystupuje z něj optický signál. Elektrický signál je přímo úměrný počtu elektronů a optický signál počtu fotonů. Co je foton a elektron snad každý ví. Všechny elektrony jsou stejně velké. Všechny fotony jedné vlnové délky jsou stejně velké.

Počet fotonů není přímo úměrný počtu elektronů na vstupu. Platí například, že $fotony = elektrony^{2.2}$. To 2.2 je gamma toho monitoru. Obdobné kalkulace platí i pro jiná zařízení, pokud jsme schopni se dohodnout, v jakých jednotkách se měří vstupní a výstupní veličiny zařízení. Ne každý monitor má gammu stejnou. A gamma jednoho exempláře monitoru se může lišit i pro jednotlivé kanály red, green, blue. Nejlepší je proto gammu změřit pomocí testovacího obrazce a nastavit ji do prohlížeče, protože pak dostaneme nejdokonalejší obraz. Při zadání nesprávných hodnot gamma mohou vzniknout závady zobrazování jako barevné závoje při ditherování, příliš kontrastní jedny partie obrazu a nedostatečně kontrastní ostatní partie obrazu, případně nesprávné podání barevného tónu v určitých partiích. V případě, že není možno určit gammu, lze použít aproximaci, kdy se všechny tři gammy nastaví stejně, a to na hodnotu 2.2.

Základní podmínka pro to, abychom očekávali vůbec věrný obraz, je nastavit správně jas monitoru. S kontrastem si pak budeme moct kroutit jak chceme, kvalitu obrazu neo-

vlivní, jen jeho intenzitu. Nastavovací procedura následuje: nastavte kontrast na minimum a jas na maximum. Zmenšete obraz abyste jasně viděli rozdíl mezi rámečkem obrazu ještě zasaženým elektronovým paprskem a krajem skla paprskem nezasaženým. Pak snižte jas dokud nepřestane být vidět toto rozhraní. Pro zlepšení viditelnosti je vhodné zhasnout, zatáhnout záclony a podobně. Jakmile rozhraní přestane být vidět, jas se již nesmí dále ubírat. V tomto okamžiku velmi opatrně nalepte kus lepicí pásky na knoflík jasu aby se již nedal nikdy více otáčet nebo se zapřísahajte na Bibli, že do elektronického menu již nikdy na jas nesáhnete. Pak si nastavte kontrast dle libosti a zvětšete obraz zpět do původní velikosti. V případě, že nejste proceduru schopni provést přesně, je lepší nechat jas trochu větší než trochu menší. Přesný popis této procedury je http://www.inforamp.net/poynton/notes/brightness_and_contrast/index.html

Display gamma je gamma exponent mezi hodnotou vstupující do grafického ovladače (například do paměti grafické karty, do X protokolu, atd.) a počtem fotonů vystupujících z luminoforu obrazovky následkem ozáření elektronovým paprskem.

Jaký má display gamma exponent, to říkají proměnné `display_red_gamma`, `display_green_gamma` a `display_blue_gamma`. Protože gammy se mohou lišit pro různé barevné kanály, tak jsou tam tři. Cílem zobrazení **Links** je, aby počet fotonů dopadajícího do oka uživatele při osvětlení 64 luxů byl stejný jako počet fotonů dopadajícího do oka uživatele, sedícího před monitorem s gammou monitoru 2.2 a gammou obrázku 1/2.2 (obrázek podle sRGB standardu) při osvětlení 64 luxů.

V případě, že máme obrázek sRGB a monitor s gammou 2.2 a osvětlení 64 luxů, postupuje **Links** následovně: vezme raw data z obrázku a pošle je do obrazovky. Pokud je osvětlení jiné, hodí se před soupnutím obrázek umocnit na následující čísla:

Osvětlení	Umocnit na
0 lx	1.333333333333
15 lx	1.111111111111
64 lx	1.000000000000
≥ 200 lx	0.888888888888

Tato magická čísla označme jako `user_gamma` a nechme je nastavit uživatele.

V případě, že gammy monitoru jsou jiné, a gamma obrázku je jiná, dělají se tyto akce:

- Dekóduj obrázek jako raw data.
- Umocni raw data na `user_gamma/obrazek_gamma`.
- Pošli to do ditherovače.
- Zditheruj to, teď je to úměrné osvětlení co opravdu poleze z monitoru.
- Umocni to na `1/display_gamma`.
- Pošli to do monitoru.

Proto je ve zdrojovém kódu `dip.c` `wanted_red_gamma`, `wanted_green_gamma` a `wanted_blue_gamma`. Jsou to gammy, které jsou zařazeny do obrazového řetězce jako exponenciální funkce $x^{-wanted_red_gamma}$, $x^{-wanted_green_gamma}$, $x^{-wanted_blue_gamma}$.

V přenosovém řetězci uvnitř linksu se dělají následující procedury:

- Nahraje se PNG obrázek písmenka do paměti a to tak, aby hodnoty uložené v paměti vygenerované přímo knihovnou libpng byly přímo úměrné osvětlení předmětu (tedy počtu fotonů) onoho virtuálního písmenka.
- Vypočte se barva pozadí z HTML podle specifikace HTML 4.0 (sRGB barevný

prostor), aby tato barva byla vyjádřena ve fotonech. Tedy se vezme číslo z HTML a umocní se na $0.45454545\dots$. Barvy z HTML jsou interpretovány podle standardu sRGB.

- Namíchají se barvy podle alfa-masky, kterou písmenko představuje. Hodnota 255 je úplná opacita inkoustu písmenka a hodnota 0 je úplná transparence inkoustu.
- Vzniklé písmenko (které je přímo úměrné fotonům na scéně) je gamma-korigováno $1/\text{wanted_gamma}/\text{display_gamma}$. Hodnoty jednotlivých pixelů jsou tedy umocněny na $1/\text{display_gamma}/\text{wanted_gamma}$.
- Vzniklý obrázek je ditherován v ditherovací engine. Ditherovací engine očekává obrázek úměrný tomu, jak vstupuje do displaye, ale ditherovací engine zná gammu displaye a počítá s ní. Ditherovací engine tedy gammu dat nemění, ale ke své správné funkci gammu displaye znát potřebuje, protože předpokládá, že barvy se aditivně sčítají sčítáním fotonů dopadlých na jednotlivé receptory v oku.
- Z ditherovacího engine vypadnou data, která již gammu definovanou nemají, a představují interní reprezentaci grafického driveru.
- Driver data umístí na obrazovku.
- Výsledný obraz leze ze skla obrazovky do oka pozorovatele. Počet fotonů tohoto obrazu je počet fotonů na scéně umocněný na $1/\text{wanted_gamma}$. `Wanted_gamma` jsou doporučeny jako 1.0 (pro jasně osvětlené pracoviště), 1.125 (normální pracoviště), 1.25 (temné pracoviště), 1.5 (temná komora, jako například při promítání diapozitivů).

V případě zobrazování obrázku z png, jpg, a podobně, se dělají následující procedury:

- Požádá se grafická knihovna (libpng, libjpeg) o to, aby vydala obrázek ve formátu 3x8 bitů s gammou $1/\text{display_gamma}/\text{wanted_gamma}$.
- Tato data se nechají ditherovat ditherovací engine. Ten vydá nespecifická data pro display.
- Data se zobrazí na obrazovce. Výsledné světlo je světlo na reálné scéně umocněné na $1/\text{wanted_gamma}$.

Vzhledem k tomu, že nejrozumnější (a nejběžnější) je PNG a JPG obrázky ukládat s gammou rovnou 0.45454545 (gamma sRGB standardu), nevzniká v řetězci degradace, která by vznikla, přenášely-li by se někde 8-bitová data s gammou 1 (tedy úměrná osvětlení scény).

6.3 HTML parser a sazeč

Zobrazovací systém se skládá ze dvou částí — HTML parseru a sazeče HTML.

6.3.1 HTML parser

HTML parser se nachází v souboru `html.c`. Vstupem do něj je funkce `void parse_html(unsigned char *html, unsigned char *eof, void (*put_chars)(void*, unsigned char*, int), void (*line_break)(void*), void *(*special)(void*, int, ...), void*data, unsigned char *head)`

Tato funkce dostane argumenty `html` a `eof`, které znamenají odkud a kam se má parsovat ve vstupním textu. Znak, na který ukazuje `eof` a další znaky **již nebudou zparsovány**.

Dále dostane pointery na funkce `put_chars`, `line_break` a `special`. A konečně poslední dva argumenty jsou `data`, která se předávají těmto funkcím a HTTP hlavička `head`.

V HTTP hlavičce je například uložen „Refresh:“, pokud se na daném URL vyskytuje. Slouží parseru k tomu, aby se například podle refreshu mohl správně zařídit.

Funkce `put_chars` se volá, když chce parser vysázet řetězec. První argument jsou `data`, která HTML parser dostane (`data`), druhý argument je řetězec, který se má vysázet, a třetí argument je délka, kolik znaků se má vysázet.

Funkce `line_break` je volána, když se má přejít na novou řádku. Dostává pouze jeden argument, a to jsou opět `data`, která HTML parser dostane pro tyto funkce.

Poslední z funkcí, funkce `special`, se volá na různé speciální efekty (například obrázek, rám, tabulka, ...). Prvním argumentem jsou opět `data`, druhý argument je identifikace akce — jedno z maker `SP_xxx`. Další argumenty závisí na konkrétní akci.

Zde je přehled jednotlivých maker speciálů `SP_xxx` a jejich význam:

Makro	Význam	Argumenty	Vrací
<code>SP_TAG</code>	<code></code>	<code>unsigned char *</code> název tagu	nic
<code>SP_CONTROL</code>	jakákoliv položka formuláře	<code>struct form_control *</code> ona položka	nic
<code>SP_TABLE</code>	získání tabulky pro překódování znakových sad	nic	<code>struct conv_table *</code>
<code>SP_USED</code>	dotaz, zda se skutečně sází; nesází, pouze zjišťuje velikost textu kvůli tabulkám	nic	0=nesází se 1=sází se
<code>SP_FRAMESET</code>	frameset	<code>struct frameset_param *</code>	<code>struct frameset_desc *</code>
<code>SP_SCRIPT</code>	javascript ve <code><script src=„...“ ></code>	NULL nebo řetězec v <code>src</code> atributu	nic
<code>SP_IMAGE</code>	obrázek	<code>struct image_description *</code>	nic
<code>SP_NOWRAP</code>	dočasně se má přestat zalamovat	integer, 0=zalamovat, 1=nezalamovat	nic

HTML parser provádí porcování HTML, vyřizování HTML tagů a správu zásobníku. Když se má vypsát nějaký text, zavolá funkci `put_chars`. Když se má zlomit řádek, zavolá se funkce `line_break`. Když se má vložit nějaký „special“, parser zavolá funkce `special`.

Jak je spravován zásobník: Zásobník je uložen v `html_stack`. Je to seznam položek `html_element`. Každá položka má u sebe atributy textu (`attr`) a odstavce (`parattr`). Když parser nalezne v HTML souboru nový tag, tak vytvoří nový `html_element`, pushne ho na zásobník a nastaví mu styl písma, jak to definuje ten tag. Když parser nalezne uzavírací tag, tak danou strukturu `html_element` ze zásobníku odstraní.

HTML sazeč (volaný funkcemi `put_chars`, `line_break` a `special`) bere jako platný styl to, co je na vrcholu zásobníku. Vrchol zásobníku je definován makrem `html_top`. Atributy elementu na vrcholu zásobníku jsou přístupné přes `html_top->attr` a atributy odstavce přes `html_top->parattr`.

HTML parser používá globální proměnnou `d_opt`, která obsahuje uživatelsky nastavitelné volby, podle kterých se má parsovat. Parsování stránky by nemělo záviset na jiných proměnných, aby se nemíchaly dokumenty v cachi. Součástí identifikace zparsovaných stránek v cachi je mimo jiné i `d_opt`.

Upozornění!

`d_opt` není mimo HTML parser definované. Proto tuto proměnnou v jiných částech **Links** nepoužívejte!

6.3.2 Sazeč HTML

Browser má dva sazeče HTML. Jeden pro textový a druhý pro grafický mód.

V textovém módu sazeč vytváří obyčejné dvojrozměrné pole, které obsahuje znaky na jednotlivých řádkách. Jedna řádka je popsána strukturou `struct line`. Textový sazeč se nachází v souboru `html_r.c`.

Grafický sazeč je v souboru `html_gr.c`. V grafickém módu sazeč vytváří strom grafických objektů. Každý grafický objekt je popsán strukturou `g_object`. Objekt má metody `draw`, `destruct`, `mouse_event`. Objekt může obsahovat další objekty. Kořenový objekt je `g_object_area`. Ten obsahuje řádky — `g_object_line`. Každý řádek obsahuje další objekty — `g_object_text`, `g_object_image`, případně `g_object_table`. Tabulka obsahuje opět několik `g_object_area` pro každé políčko — a tak můžou objekty vytvářet strom libovolné struktury.

6.4 Javascript

6.4.1 Interpretování javascriptu

Javascript se začíná interpretovat po natažení **celého** obsahu, tj. až když je pod střechou

```
<script language="javascript">.*</script>
```

Interpretace začíná výrobou kontextu (`js_create_context`). Při výrobě kontextu se naalokuje struktura `js_context` a její položky se inicialisují.

Pokračuje se neparsováním kódu, tedy zavoláním funkce `js_execute_code`, které se jako argument předá pointer na kontext a pointer na kód. Parser zdrojový kód zkompiluje do podoby stromového mezikódu, nad nímž by šlo uvažovat o optimalizacích. Toto řešení bylo zvoleno proto, že k parsování javascriptu lze snadno použít utilit `flex` (ev. `lex`) a `bison` (ev. `yacc`). Tyto nástroje jsou reentrantní způsobem, který pro naše způsoby připomíná použití parního válce k louskání ořechu, tj. ve chvíli, kdy by se měl vyrobit samomodifikující kód (což je u javascriptu obvyklé), by se interpretace značně zdržovala, vůbec stranou necháváme fakt, že by bylo potřeba ošetřovat i pravidelná přerušení interpretace, ke kterým dochází dosti často. Proto po zavedení stránky proběhne přestavba zdrojového textu do pohodlnější (a rychleji zpracovatelné) stromové podoby. V této podobě se kód interpretuje k tomuto účelu speciálně vyvinutým interpretem, který by měl mít možnost postihnout všechny odstíny práce javascriptu. Bohužel některé vlastnosti javascriptu jsou dle našeho názoru značně odtážené od programátorského pohledu, nemluvě o tom, že autoři javascriptu tyto partie používají zřídka správně, proto bylo od některých položek upuštěno s vyhlídkou na to, že buďto nejsou důležité, nebo se v některé z pozdějších verzí dodělají. Jedná se zejména o funkci `eval(string a)`, která má za běhu javascriptu spustit vyhodnocování řetězce zadaného jako argument. Takovouto funkci používají někteří programátoři například k tomu, aby vyvolali funkce různých jmen namísto toho, aby toto rozlišili parametrem (například `funkce1()`; místo `funkce(1)`);

Takovéto jednání je vyloženým diletantismem a není důvodu nabízet mu oporu, zvláště je-li na práci dostatek závažnějších problémů. Druhou vynechanou položkou je funkce `sort` u pole, která by sice mohla být užitečná, ale jednak si ji může člověk napsat v javascriptu jinak, navíc by měla mnoho podobného funkci `eval`.

Je-li skript zinterpretován, zůstávají jeho údaje v paměti, jelikož v rámci stejného kontextu lze očekávat, že budou spouštěny „eventové handlers“, tedy ovladače událostí. Tyto se začasť odkazují na funkce definované v těle stránky (v souladu se specifikací Javascript 1.1 od Netscape Corporation).

Datové struktury spojené se skriptem definitivně opouštějí paměť až ve chvíli, kdy je stránka uživatelem opouštěna. Při této příležitosti se zavolá funkce `js_destroy_context`, která zruší celý kontext skriptu, tedy jeho nosnou entitu.

6.4.2 Gramatika

Gramatiku javascriptu se podařilo dostat z normy Javascript 1.1. Poté, co jsme ji implementovali a zkusili napsat několik stránek, jsme zjistili, že máloco se tak liší, jako dnes provozované skripty a gramatika jim určená. Autoři stránek náhodně vynechávají středníky na koncích statementů, divoce míchají operátory vzetí hodnoty v poli a dotazu na člen objektu. Bylo tudíž nutné poněkud gramatiku upravit. Jelikož jsme se však nechtěli od specifikací odchylovat (v souladu s cíli projektu, mezi nimiž figuruje např. postavení browseru, který bude zobrazovat korektní kód, o nekorektním se jasně řeklo, že browser nesmí zhroutit, zato jej browser smí zinterpretovat přibližně libovolně). Zvolili jsme toto řešení: Gramatiku jsme upravili tak, aby popsání gramatické nesmysly bylo možné provádět bez omezení, leč každý přestupek proti normě je „oceněn“ warningem. Hlášení warningů stejně jako errorů lze vypnout. Pro účely vývoje stránky je vhodné si tyto zapnout, na libovůli uživatele, který si stránky prohlíží, pak je, aby se nechal informovat o zlozvycích autora, nebo ne.

6.4.3 Typové konverze

JavaScript disponuje nemalým množstvím typů proměnných, konstant a dalších, interpret písícím jedincům nepřijemných položek. Připouštíme rovnou, že ne všechny konverze jsou implementovány (kupř. jsme kategoricky odmítli dělat dekompilaci, tj. konverzi funkce do řetězce, která je v normě specifikována jako zobrazení kanonického zdrojáku. Jednalo by se vlastně o další kompilátor, jehož stavba by sice nebyla zásadně náročná, ale vzhledem k tomu, že jedinou výhodou pro pisatele stránek, ev. jejich čtenáře, by byla, že si mohou takovouto funkci v řetězci modifikovat, nebo nakonec dát vyhodnotit (chyby v kódu takto odhalovat nelze), jediná věc, která vypadá rozumně, je udělat „fake dekompilaci“, tedy ať vypadá konvertovaná funkce jak chce, vždycky napíšeme:

```
function funkce(){alert("Funkce je interní!");}
```

Z důvodů ladění interpretu je pravděpodobně občas možné udělat normou zakázané vzetí nedefinované hodnoty. K tomuto došlo při vývoji. Aby interpret nepadal tak často při dotazech na dosud neimplementované položky, bylo možné dostat hodnotu `undefined`. V současné době by mělo být již opraveno, ale vzhledem k počtu možných konverzí, je pravděpodobné, že takováto chyba ještě někde přežívá.

6.4.4 Scheduler

V **Links** je základní vlastností konečnost prováděných operací, jelikož jedním ze základních cílů projektu byla stabilita. Platí invariant, že žádný kus kódu nesmí být vykonáván neomezenou dobu (výjimkou je snad jenom `gethostbyname`). Jak je tomu v modulu javascriptu? Parsování zařizují `flex` a `bison`, jejichž kód považujeme za konečný, tyto „lamače textu“ nejsou při práci vyrušovány interrupty. Po napsání kódu je interpretace „objednána“ nastavením timeru (v této chvíli se provádění

javascriptu přeruší). Interpretace probíhá po pevných počtech kroků (100), tedy nejpozději po sto iteracích interpretu (interně označovaných jako žvýknutí) je interpretace přerušena a tím je zabráněno tomu, aby zlomyslný autor javascriptu browser paralyzoval. Některé browsery mají problémy přežít např. tento kód:

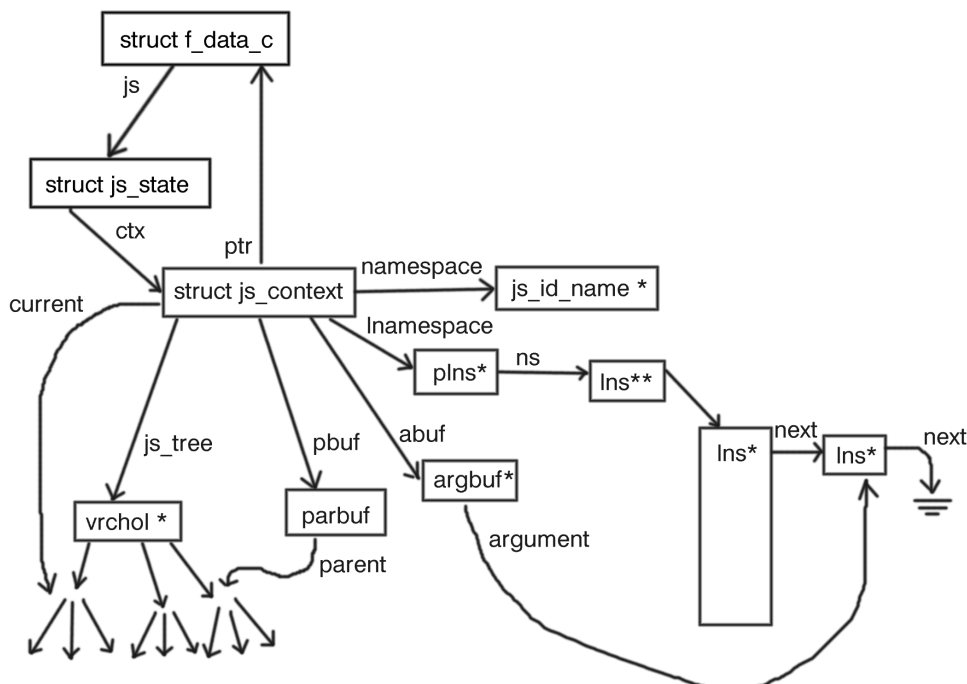
```
<script language="javascript">while(1);</script>
```

Zobrazování dialogových nebo jen hrozivých okének (`alert`, `confirm`, `prompt`) interpretaci přeruší do chvíle, kdy jsou tato okénka uživatelem zrušena (bráníme se tak možnosti diskreditovat browser například tímto kódem:

```
<script language=javascript>
while(1)alert("Cha, cha, cha!");
</script>
```

který rovněž některé browsery těžko snášejí. Toto však, s ohledem na možnost nastavit si timery není dostatečné, proto je každé okno vyprodukované javascriptem opatřeno tlačítkem „Kill script“, které vyvolá uživatelskou chybu (zvolání funkce `js_error` spolehlivě interpretaci zastaví), na ploše pak zůstane viset několik okének, která bude muset uživatel „doklepat“.

6.4.5 Struktury javascriptu



Z každého rámu v HTML dokumentu (`f_data_c`) vede ukazatel `js` na strukturu `js_state`, která obsahuje frontu skriptů, které se mají spustit, a ukazatel `ctx` na kontext právě běžícího skriptu.

`js_context` je hlavní struktura popisující kontext javascriptu. Z ní vede ukazatel `js_tree` na interpretovaný strom, jehož uzly jsou ještě provázány spojovým seznamem. Do stromu míří i pointer `current`, který ukazuje na momentálně interpretovaný uzel. Struktura `parbuf` je zásobník otců, tedy opět ukazatelů do stromu. Pointer `namespace` ukazuje na globální namespace, neboli hashovací tabulku všech identifikátorů ve stromě. Lokální namespace (`lnamespace`) je representován strukturou `plns`, ze které vede ukazatel na další lokální namespace (na obrázku není nakreslen) a pointer `ns` na hashovací tabulku jednotlivých identifikátorů. Přímo na identifikátory se může odkazovat zásobník argumentů `argbuf`.

6.4.6 Hlášení chyb

Chyby jsou v javascriptu rozděleny do několika skupin:

1. **Chyby lexikální** — o nich je konstatováno, že nastala chyba
2. **Chyby syntaktické** — o nich je také pouze konstatováno, že nastala chyba.
3. **Chyby sémantické** — používání nedefinovaných vlastností, metod apod. je odchyťováno až za běhu. Narozdíl od předchozích dvou je konstatováno, co je ve skriptu špatně. Tato informace u předchozích dvou chybí, protože s ohledem na použité kompilační nástroje lze sice zjistit, jak pokračuje zdrojový text, ale není jasné, mezi kterými stavy k chybě došlo.
4. **Warningy** — při kompilaci dochází k nesouladu mezi tím, co se ještě někteří autoři stránek odvažují prohlásit za javascript, a mezi tím, co jako javascript specifikuje norma. Jelikož téměř všechny browsery chyby různých typů tolerují, bylo nutné udělat to samé i u **Links** (má-li být použitelný). Každá chyba, kterou jsme se rozhodli tolerovat, je rovněž oceněna oknem hlásajícím, že nastal warning, ale interpretace pokračuje.

Ke každému hlášení o chybě je připojena citace řádku, na kterém byla zjištěna, proto může být citován pozdější řádek, kupř. jedná-li se například o zapomenutý středník na konci řádku, je uživatel uvedoměn až s koncem následného komentáře.

Příklad:

```
a=5
/* Tohle je ten
 * roztahaný
 * komentář, který zavíní,
 * že je zobrazena až
 * TATO ŘÁDKA */
```

6.4.7 Ladění

Celý modul je doslova prošpikován debugovacími nástroji. Lexikální analyzátor umí vypisovat jednotlivé tokeny, které se čtou, syntaktický analyzátor zase dovede hlásit, jaká redukce proběhla. Dojde-li k lexikální nebo syntaktické chybě, lze těmito nástroji snadno zjistit, co není v pořádku. Je pravda, že pro obvyčejného uživatele se jedná o parní válec na ořechy, ale ve chvílích, kdy gramatika nebyla úplně v pořádku, byly tyto nástroje nepostradatelné. Stejně tak interpret mezikódu je osazen výkonnými hlásiči, které říkají, v jakém stavu se interpret nachází (pokolikáté se prochází kterým uzlem stromu, občas i co se vrací). Obsluha interních funkcí a proměnných je neméně upovídána. Hlásí vstup a výstup do ovladače od každé funkce. Přestože debugovacích výstupů je poměrně mnoho, chybí tam občas konkrétní údaje (čísla, která se napočítala apod.). Toto je chyba, ale ve chvíli, kdy člověk loví chybu v interpretu, bývá dostatečné, když zjistí, mezi kterými dvěma stavy se stalo něco divného, na interpret je stejně potřeba vzít debugger a vypisovat si každou podezřelou veličinu.

6.5 Obecné seznamy

6.5.1 Úvod

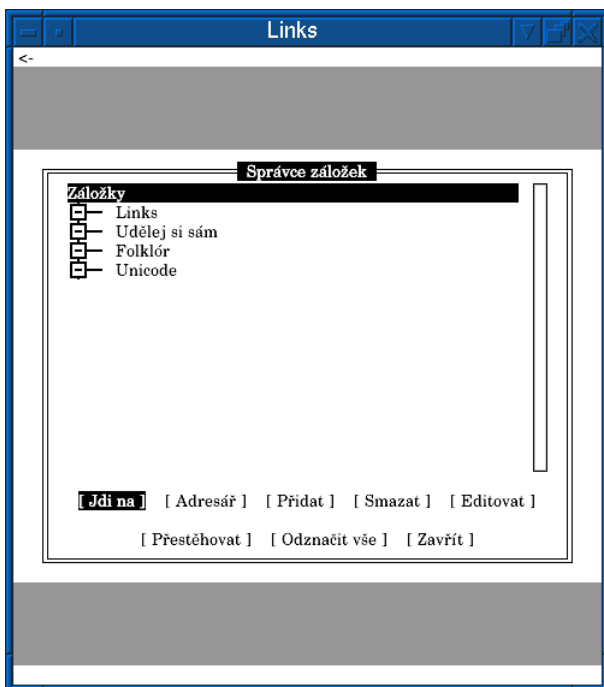
V souboru `listedit.c` jsou funkce a datové struktury umožňující jednoduše definovat nějaký seznam, se kterým bude uživatel **Links** pracovat. Z využitím funkcí z `listedit.c` se dají vytvořit velice jednoduchým přídatným kódem například bookmarky, asociace, přípony,

Seznam může být buď plochý, nebo stromový. Plochý seznam je lineární seznam, kde jedna položka následuje druhou. Ve stromovém seznamu jsou kromě položek ještě adresáře. V adresářích mohou být položky nebo další adresáře.

Obecné seznamy neřeší vyrábění seznamů, jejich ukládání na disk, čtení z disku. To si musí implementátor obstarat sám.

6.5.2 Ovládání uživatelem

Při zavolání funkce `create_list_window` se uživateli **Links** zobrazí okno se seznamem. Okno vypadá takto:



Ovládání okna je následující:

- klávesy NAHORU, DOLŮ — posun kurzorem po seznamu o jednu položku.
- klávesy HOME, END — přesun na začátek respektive konec seznamu.
- klávesy PAGE-UP a PAGE-DOWN — posun na předchozí resp. následující stránku seznamu.
- *, INSERT — označení nebo odznačení položky.
- klávesy DOLEVA, DOPRAVA — výběr tlačítka ve spodní části okna.
- ENTER — aktivace právě vybraného tlačítka na aktuální položce.
- mezerník — otevření nebo zavření adresáře.
- + a - — otevření respektive zavření adresáře.
- kliknutí myši na tlačítko — výběr a zároveň aktivace tlačítka.
- kliknutí levým tlačítkem na položku seznamu — výběr položky.
- kliknutí pravým tlačítkem na položku seznamu — označení nebo odznačení položky.
- potáhnutí prostředním tlačítkem myši, otočení kolečkem myši — scroll.
- potáhnutí scrollovací lištou v okně — scroll.

V hlavním okně jsou tato tlačítka (věci týkající se adresářů se pochopitelně vyskytují pouze stromových seznamů):

- **Adresář** — vytvoří nový adresář za vybranou položkou, pokud je vybraná položka otevřený adresář, vloží do něj.
- **Přidat** — přidá novou položku za vybranou položku, pokud je položka otevřený adresář, vloží do něj.
- **Smazat** — smaže vybranou položku, pokud se jedná o adresář, smaže i jeho obsah.
- **Editovat** — edituje aktuální položku.
- **Odznačit vše** — odznačí všechny položky.
- **Přestěhovat** — přestěhuje označené položky za aktuálně vybranou položku. Pokud je aktuálně vybraná položka otevřený adresář, přesune do něj.
- **Zavři** — zavře okno se seznamem.

6.6 Datové struktury a funkce

Seznamy používají dvě významné datové struktury. První z nich je `struct list`. Tato struktura obsahuje vlastní seznam. Skládá se z těchto částí:

- `void *next` — ukazatel na následující položku.
- `void *prev` — ukazatel na předchozí položku.
- `unsigned char type` — 0.bit říká, zda se jedná o adresář (1), nebo o položku (0); 1.bit říká, zda je adresář otevřený (1), nebo uzavřený (0). 1.bit není pro položku využit, je využit pouze pro adresář; 2.bit říká, zda je položka označena (1 znamená označeno).
- `int depth` — hloubka položky, hlava má hloubku -1.
- `void *fotr` — pokud se jedná o stromový seznam, ukazuje na otce (adresář, ve kterém je bezprostředně položka), používá se pro práci se zavřenými adresáři.

Druhá struktura je `struct list_description`, obsahuje popis seznamu. Pro každou implementaci seznamu je potřeba vytvořit jednu takovouto strukturu, ta se pak předává všem funkcím pracujícím se seznamy. Struktura obsahuje tyto položky:

- `unsigned char type` — říká, zda se jedná o plochý (hodnota 0) nebo stromový (hodnota 1) seznam.
- `struct list* list` — ukazatel na vlastní seznam.
- `void *(*new_item)(void * initial_data)` — funkce na vytvoření nové položky v seznamu, naalokuje novou položku a vrátí na ni ukazatel. Argument jsou buď `data` na inicialisaci položky (získaná od funkce `default_value`, nebo `NULL`, když má být daná položka prázdná. Jestliže `initial_data` není `NULL`, funkce musí tato `data` **odalokovat**.
- `void (*edit_item)(struct dialog_data *dlg, void *item, void(* ok_fn)(struct dialog_data*, void*, void *, struct list_description*), void *ok_arg, unsigned char dlg_title)` — funkce na editaci položky. Vytvoří dialog, ve kterém může uživatel editovat příslušnou položku. `item` je editovaná položka. Funkce `ok_fn` se zavolá při úspěšném ukončení editace (typicky zmáčknutí OK v editačním dialogu), jako první argument se jí předá `dlg`, druhý argument je `ok_arg`, třetím argumentem se jí předá pointer na editovanou položku, jako poslední argument se jí předá popis seznamu. Argument `dlg_title` může nabývat hodnot `TITLE_EDIT` nebo `TITLE_ADD`. Tento argument se v editační funkci využije pro rozlišení textu v dialogu pro uživatele (zda se jedná o přidávání nové položky, nebo o editaci již existující položky).

- `void *(*default_value)(struct session*ses, unsigned char type)` — používá se, když uživatel klikne na tlačítko „vyrob novou položku“. Tato funkce vytvoří (naalokuje) data pro inicialisaci nové položky. Data jsou ve formátu, který si implementátor zvolí, předávají se funkci `new_item`, která data následovně odalokuje (viz výše). Argument `type` se používá pro rozlišení mezi adresářem (hodnota 0) a položkou (hodnota 1).
- `void (*delete_item)(void *)` — smaže položku seznamu. Pokud předchůdce respektive následovník není NULL, upraví ukazatele předchůdce a následovníka.
- `void (*copy_item)(void * old, void * new)` — dostane dvě naalokované položky, zkopíruje obsah `old` do `new` s výjimkou pointerů provazujících seznam. Data v `old` položce (nikoliv položku `old`!) odalokuje.
- `unsigned char *type_item(void *item, int what)` — naalokuje řetězec, do kterého vytiskne položku `item`. Argument `what` rozlišuje, zda se má vytisknout celá položka (hodnota 0) — například při mazání položky, nebo pouze titulek položky (hodnota 1). Funkce by měla být schopna u stromového seznamu vytisknout i hlavu seznamu, typicky by měla tisknout nějaký fixní text (například „Bookmarky“).
- `int codepage` — číslo kódové stránky, ve které se předávají všechny řetězce.
- `int window_width` — šířka hlavního okna na zobrazování seznamu.
- `int n_items` — počet položek, které se mají zobrazovat v hlavním okně.
- `int item_description` — kód řetězce popisující položku (např. kód řetězce „záložka“).
- `int already_in_use` — kód řetězce hlásajícího, že je list již otevřen v nějakém jiném okně. Např. kód řetězce „Záložky jsou již otevřeny v jiném okně“
- `int window_title` — kód řetězce s titulkem hlavního okna.
- `int delete_dialog_title` — kód řetězce s titulkem dialogu na mazání položky.
- `int button` — kód řetězce s popisem uživatelského tlačítka.
- `void (*button_fn)(struct session*ses, void*item)` — funkce uživatelského tlačítka (např. „Jdi na záložku“. Pokud uživatelské tlačítko není definováno, `button_fn` je NULL. Tlačítko nefunguje na adresáře ani na hlavu seznamu (u stromového seznamu). Pokud bude třeba funkci předávat další argumenty, definice se přepíše.

Struktura dále obsahuje tyto vnitřní proměnné, na které by implementace seznamu neměla sahat, všechny by měly být při inicialisaci struktury nastaveny na hodnotu 0.

- `struct list *current_pos` — ukazatel na položku v hlavním okně, na které je kurzor.
- `struct list *win_offset` — ukazatel na 1. položku v hlavním okně.
- `int win_pos` — vertikální pozice kurzoru v hlavním okně
- `int open` — 0=hlavní okno zavřeno, 1=hlavní okno otevřeno
- `int modified` — data byla modifikována, bylo by dobré je potřeba uložit na disk (pokud se ukládají). Pokud je `modified` 0, není seznam potřeba ukládat, protože se nezměnil.
- `struct dialog_data *dlg` — ukazatel na dialog s hlavním oknem, je platný jen v případě, že `open` je rovno 1.

6.6.1 Funkce

Funkce viditelné z modulu `listedit.c` jsou tyto:

- `int create_list_window(struct list_description*, struct list*, struct terminal *, struct session *)`
Vytvoří okno pro práci se seznamem. Jako argumenty dostává popis seznamu, pointer na seznam, terminál, na kterém se má okno vytvořit, a `session`.
- `void redraw_list_window(struct list_description *ld)`
Překreslí obsah okna se seznamem.
- `void reinit_list_window(struct list_description *ld)`
Znovu inicialisuje okno do stavu jako po spuštění `create_list_window`. To znamená přesune kurzor a výhled na seznam na začátek a překreslí.

6.6.2 Implementace seznamu

Pokud chcete přidat nový seznam, nejprve vytvořte novou instanci struktury `struct list_description` a nainicialisujte ji příslušnými hodnotami a funkcemi, které napíšete. Pokud budete chtít vytvořit okno, zavolejte `create_list_window` s Vaším seznamem.

6.6.3 Stromový seznam

Každý člen seznamu má u sebe proměnné: hloubka, typ (adresář/položka), flag otevřeno/zavřeno, ukazatel na otce.

- **Hloubka** je číslo od 0 výše, hlava má jako jediná hloubku -1. Stromový seznam je interně uložen jako lineární seznam. Adresář je uložen tak, že je nejprve adresář a za ním následuje jeho obsah. Obsah adresáře je tedy po adresáři následující souvislý blok položek, které mají hloubku větší než hloubka dotyčného adresáře.
- **Typ** může být buď 1 (adresář), nebo 0 (položka).
- **Flag** otevřeno/zavřeno se používá jen u adresáře. U položky se ignoruje.
- **Ukazatel na otce** slouží k urychlení vykreslování seznamu. Když jsou některé adresáře zavřené, tak je potřeba jejich obsah přeskočit.

6.6.4 Přidávání a editace položek — funkce `edit_item`

Když uživatel zmáčkne tlačítko „Přidat položku“, vyrobí se nová položka (zavolá se funkce `new_item`), která se nepřidá do seznamu. Poté se zavolá funkce `edit_item` na editaci položky. Po úspěšné editaci položky (zmáčknutí „OK“) se teprve položka přidá do seznamu. Při zmáčknutí „Zrušit“ se položka smaže.

Při editování položky (uživatel zmáčkne tlačítko „Edit“) se vytvoří nová položka, zkopíruje se do ní obsah té původní (zavolá se `copy_item`) a opět se zavolá `edit_item`. Jestliže uživatel zruší editaci tlačítkem „Cancel“, položka se smaže. V opačném případě se obsah položky zkopíruje zpět do původní (opět pomocí `copy_item`) a položka se smaže.

Při zmáčknutí „Cancel“ tedy funkce `edit_item` musí položku smazat. Při zmáčknutí „OK“ je zavolána funkce, kterou `edit_item` dostane jako argument (viz výše).

6.6.5 Přístup do seznamu z více oken Linksu

Struktury nejsou odolné proti více paralelním přístupům (například ve se ve struktuře ukládá pozice kurzoru v okně atd.), proto je přístup z více oken zakázán. Funkce `create_list_window` nejprve nastaví ve `struct list_description` proměnnou `open` na 1, při zavření hlavního okna se proměnná `open` nastaví na 0. Pokud se při zavolání funkce `create_list_window` zjistí, že proměnná `open` je již nastavena na 1, uživateli se objeví

upozornění, že okno je již jednou otevřeno a že ho nejprve musí zavíít. Pochopitelně toto platí pro stejný seznam. Není problém mít otevřeno více oken, každé od jiného seznamu.

Použitý princip umožňuje závodění. V „ideálním případě“ se totiž může stát, že výše popsaný test selže a tedy bude okno seznamu otevřeno vícekrát. Jelikož je vysoce nepravděpodobné, že se toto uživateli podaří (kliknout ve více oknech současně na otevření okna se seznamem, a ještě mít stěstí, že operační systém nascheduluje oba procesy ve „vhodném“ pořadí), tak by vynaložené úsilí na implementaci precizního zamykání bylo neadekvátní k výsledku. Proto byla zvolena jednodušší varianta.

7. Přidávání nových částí prohlížeče

7.1 Přidání souboru do distribuce Linksu

Pokud chcete přidat nový `.c` soubor do distribuce, editujte soubor `Makefile.am` a na řádku začínající `links_SOURCES=` připište název souboru (včetně přípony `.c`). Pak spusťte `rebuild reconf` a nechte ho doběhnout.

Jestliže přidávaný soubor není zdrojový kód (ani jeden ze souborů `.c`, `.h`, `.inc`), tak ho přidejte do `Makefile.am`, ale na řádku začínající `EXTRA_DIST=`. Tyto soubory budou přidány do distribuce při spuštění `make dist`. Po přidání je opět potřeba spustit `rebuild reconf`.

7.2 Jak přidat novou překladovou tabulku kódování

K přidání nové kódovací tabulky vlezte do adresáře `Unicode` a proveďte následující. Postup bude demonstrován na přidání kódování CP 852:

- 1) Do souboru `cp852.cp` vytvořte (nebo odněkud zkopírujte) překladovou tabulku pro kódování CP852.
- 2) Na první dva řádky souboru napište
`CP 852`
`"cp852", "852"`
- 3) Do souboru `index.txt` přidejte řádek:
`cp852`
- 4) Spusťte skript `gen`.

Po provedení této procedury je potřeba recompileovat prohlížeč, aby se tabulka dostala i do binárního souboru prohlížeče.

7.3 Přidávání nového grafického formátu

Zde bude popsán modelový postup, jak přidat nový grafický formát `xbm`. Jiný grafický formát se přidává analogicky.

- 1) Vytvořte soubor `xbm.c`, na začátku napište:
`#include "cfg.h"`
`#ifdef G`
`#include "links.h"`
Úplně na konec souboru napište:
`#endif /* G */`

- 2) Do souboru `xbm.c` je potřeba napsat funkce `xbm_start` a `xbm_restart`. Funkce budou mít toto rozhraní:

```
void xbm_start(struct cached_image *cimg);
void xbm_restart(struct cached_image *cimg, unsigned char *data,
int length);
```

 kde `data` jsou vstupní data obrázku, `length` je délka dat a `cimg` popisuje dekodér obrázku.
- 3) Do `links.h` přidejte k definicím typů obrázků `IM_JPG`, `IM_GIF` nový typ `IM_XBM` s číslem, které následuje v posloupnosti číslování typů obrázků.
- 4) Do `links.h` přidejte:

```
/* xbm.c */
#ifdef G

void xbm_start(struct cached_image *cimg);
void xbm_restart(struct cached_image *cimg, unsigned char *data,
int length);
#endif /* G */
```
- 5) Do `img.c` a `types.c` na místech, kde je „image/jpg“ přidejte analogicky „image/x-bitmap“.
- 6) V `xbm.c` definujte strukturu `struct xbm_decoder` analogicky ke struktuře `struct jpg_decoder`. Pokud potřebujete používat dekodér i v `img.c`, strukturu dejte do `links.h`
- 7) Na místa, kde se v `img.c` volá `jpg_restart` přidejte analogicky `xbm_restart`.
- 8) Na místa, kde se v `img.c` volá `jpg_start` přidejte analogicky `xbm_start`.
- 9) Do funkce `destroy_decoder` v `img.c` přidejte analogicky destrukci `xbm` dekodéru.

Pokud přidávaný dekodér používá nějaké externí knihovny, je potřeba do skriptu `configure` přidat na tuto knihovnu a možnost vypnutí tohoto dekodéru. Do souboru `configure.in` přidejte na začátek (k ostatním podobným blokům) blok (uveden příklad pro knihovnu jpeg):

```
AC_ARG_WITH(libjpeg, [ --without-libjpeg compile without JPEG support],
[if test "$withval" = no; then disable_jpeg=yes; else disable_jpeg=no; fi])
  cf_have_jpeg=no
if test "$disable_jpeg" != yes ; then
  AC_CHECK_HEADERS(jpeglib.h)
  AC_CHECK_LIB(jpeg, jpeg_destroy_decompress)
  if test "$ac_cv_header_jpeglib_h" = yes && test
"$ac_cv_lib_jpeg_jpeg_destroy_decompress" = yes; then
    AC_DEFINE(HAVE_JPEG)
    cf_have_jpeg=yes
    image_formats="$image_formats JPEG"
  fi
fi
```

Všechn kód pro nový grafický formát dejte podmíněně kompilovat, pokud je definováno příslušné makro, například `HAVE_JPEG`

Pokud je knihovna nalezena a prohlížeč se bude kompilovat s podporou přidávaného grafického formátu, musíte do proměnné `image_formats` přidat název Vašeho formátu, který se má vypisovat po doběhnutí `configure` skriptu při vypisování výsledků.

V předchozím příkladu testu knihovny se již do proměnné přiřazuje. Do proměnné přiřadíte takto:

```
image_formats="$image_formats JPEG"
```

Při psaní funkcí `xbm_start` a `xbm_restart` musí platit následující invarianty:

Když se zná hlavička obrázku, tak se do `cimg` vyplní položky `width`, `height`, `buffer_bytes_per_pixel`, `red_gamma`, `green_gamma`, `blue_gamma`, `strip_optimized` a zavolá se `header_dimensions_known(cimg)`. To vytvoří buffer vyplněný pozadím (pokud je `alfa`, tak průhlednou) a `restart()` do toho bufferu začne vyplňovat byty, jak je dekoduje. V případě, že narazí na chybu, zavolá `end(cimg)` a `return`. Když dekodování skončí (další byty se můžou ignorovat a obrázek se již nebude měnit), zavolá se také `end(cimg)`; `return`;

Pokud se v `xbm_start` něco nepodaří, zavolá se `end(cimg); return`; . Strukturu `xbm_decoder` si musí dekodér sám naalokovat, vyplnit si do něj, co potřebuje, a na konci ji musí zase uvolnit.

Pokud se zavolá `end()`, tak už nebude zavolán `restart()`. `start()` se zavolá jenom jednou úplně na začátku, aby se vyrobil dekodér. Do dekodéru mezi `restarty` nikdo cizí nesahá. `end()` může volat `destroy()`, jinak se `destroy` nemůže zavolat z funkcí dekodéru. `restart()` i `destroy()` se můžou mimo funkce dekodéru volat kdykoliv. `restart()` dostane blok dat, která se mají dekodovat.

7.4 Přidávání nového jazyka

Zde uvedeme modelový postup, jak přidat do **Links** francouzštinu. Přidání jiného jazyka je analogické. Všechny popisované operace provádějte v adresáři `intl`.

- 1) Vytvořte soubor `french.lng`.
- 2) Do souboru `index.txt` přidejte řádek
`french`
- 3) Spusťte skript `synclang`.
- 4) Na prvním řádku (s konstantou `T__CHAR_SET`) napište místo `NULL` název znakové sady, ve které budete překlad psát. Řádek bude například vypadat takto:
`T__CHAR_SET, "ISO-8859-1",`
- 5) Na druhý řádek namísto `NULL` napište název jazyka. v tomto případě tedy druhý řádek bude vypadat takto:
`T__LANGUAGE, "French",`
- 6) V jednom okně si zobrazte soubor `english.lng` a v druhém editujte soubor `french.lng`, kam ke každé konstantě `T_xxx` napište místo `NULL` do uvozovek příslušný překlad. Překlad pište v tom kódování, jaké jste uvedli na začátku překladového souboru.
- 7) Spusťte skript `gen-intl`.
- 8) Překompilujte **Links**.
- 9) Nyní v **Linksu** bude nový jazyk — francouzština.

7.5 Přidání nového řetězce do jazykových překladů

Tato kapitola popisuje přidání nového řetězce do jazyků. Všechny popisované operace se provádějí v adresáři `intl`. Na modelovém postupu ukážeme, jak se přidá text „Welcome to links!“, v češtině „Vítej v programu links!“. Přidání jiných řetězců je analogické.

- 1) Vymyslete konstantu, kterou se budete odkazovat na přidávaný řetězec. Konstanta musí začínat `T_` a ještě nesmí vyskytovat v souboru `english.lng`. V našem případě zvolíme `T_WELCOME_TO_LINKS`.
- 2) Do souboru `english.lng` přidejte řádek, který na začátku bude obsahovat konstantu, pak čárku a za ní anglickou verzi řetězce v uvozovkách, na konci řádky pak opět čárku:
`T_WELCOME_TO_LINKS, "Welcome to links!",`
Nezapomeňte na čárku na konci řádky!
- 3) Pusťte skript `syncclang`.
- 4) Editujte ostatní jazyky a na řádku obsahující `T_WELCOME_TO_LINKS` napište místo `NULL` překlad do příslušného jazyka v kódování příslušného jazyka (uvedeném na začátku překladového souboru). Pro češtinu tedy v souboru `czech.lng` bude řádek vypadat takto:
`T_WELCOME_TO_LINKS, "Vítej v programu links!",`
 Řetězec bude v kódování ISO 8859-2.
- 5) Jazyky, které needitujete budou obsahovat anglickou verzi textu.
- 6) Pusťte skript `gen-intl`.
- 7) Nyní můžete v programu používat přidávaný řetězec. Na místě, kde se v programu řetězec používá, se použije `TEXT(T_xxx)`, pokud se řetězec bude předávat `bfu` vrstvě (menu, dialogy, atd.) — `bfu` vrstva sama volá makro `_()`. Pokud se řetězec bude tisknout rovnou (v textovém či grafickém módu), použije se `_(TEXT(T_xxx), term)`.

Podobný postup se provádí při přidávání horkých kláves. Rozdíl je pouze v tom, že konstanta začíná `T_HK_`, jinak je postup stejný.

7.6 Přidání nového fontu

Pokud chcete vytvořit nový font (toto dělejte jen v případě, že by se znaky kryly, zbytečně mnoho adresářů s fonty by mohlo prohlížeč zpomalit), řiďte se následujícím postupem.

- 1) Do adresáře `graphics/font` přidejte nový adresář s názvem fontu. Název fontu musí mít tento tvar:
`family-weight-slant-adstyl-spacing`
 - **family** — může být libovolné jméno, nehledí se na velká a malá písmena (pro přehlednost je dobré ho napsat malými písmeny), jméno může být složeno pouze z písmen a podtržíték.
 - **weight** — je buď `bold` pro silnou tloušťku, nebo `medium` pro normální tloušťku.
 - **slant** — je buď `italic` pro skloněné písmo, nebo `roman` pro obyčejné písmo.
 - **adstyl** — může být `sans` pro bezpatkové písmo, nebo `serif` pro písmo s patkami.
 - **spacing** — je `mono` pro font s pevnou šířkou písmen (se stejnou šířkou všech znaků), nebo `vari` pro font s různě širokými znaky.
- 2) Do tohoto nově vytvořeného adresáře nahrajte `PNG` soubory pro jednotlivé znaky (pro každý znak jeden soubor). Soubory musí být pojmenovány čtyřmístným

hexadecimálním unikódovým číslem znaku, přípona musí být `.png`. V názvu jsou povolena jen malá písmena, název souboru se smí skládat pouze z číslic a písmen a-f. Zde jsou příklady jmen:

```
23aa.png
0020.png
30bf.png
```

Soubory musí být šedotónové PNG obrázky, pokud budou barevné, budou prohlížečem zkonvertovány na černobílé a pouze zabírat více místa. V obrázku je bílá interpretována jako „inkoust“, černá jako „papír“, přechod mezi nimi jako příslušné namíchání „inkoustu“ a „papíru“. Aspect ratio obrázku se ignoruje, alpha kanál se kombinuje s pozadím. Na velikosti obrázků nezáleží, všechny obrázky budou zkonvertovány na stejnou výšku. Výška se doporučuje kolem 112 pixelů. U monospaced fontů (s pevnou šířkou znaku) musí mít všechny obrázky stejný poměr výšky k šířce.

Doporučujeme, aby obrázky byly šedotónové, bez alphy, s gamma 1.0, čtvercovými pixely, bez extra informací jako jsou například komentáře, čas, datum, barevné profily, pozadová barva, offset, To je vše je z důvodu úspory místa a také proto, že tyto informace jsou ve fontu naprosto zbytečné a nevyužijí se.

- 3) Pokud chceme přidat aliasy na první položku jména, vytvoříme v adresáři soubor `aliases` a do něj napíšeme jednotlivé varianty, každou na začátek samostatného řádku. Například takto:

```
new_century_school
century_school_book
new_century_school_book
century
```

- 4) Po přidání všech znaků spusťte v adresáři `graphics` skript `gen`, který vygeneruje ze všech fontových souborů (ze všech PNG) jeden velký C zdroják `font_include.c` s fontem.
- 5) Překompilujte **Links**.
- 6) Nyní máte v prohlížeči přidán Váš font.

7.7 Přidávání fontů z Ghostscriptu

Tento odstavec popisuje, jak font ve formátu `*.pfb`, `*.afm`, `*.pfa` (nebo jeho části) přidat do prohlížeče. Vstupem procedury je font ve formátu, který je schopen přečíst Ghostscript a výstupem sada obrázků ve formátu PNG, které se přidají do adresáře `graphics/font/<jméno fontu>`.

Přidávání fontů je poloautomatické s použitím několika scriptů a pomocných programů, které jsou přibaleny v CVS vydání **Links**. Tyto programy nejsou v běžné distribuci, protože pro instalaci a běh prohlížeče nejsou potřeba. Programy se kompilují také bez použití `configure` skriptu, a vyžadují ke svému běhu striktnější verzi `libpng` než samotný prohlížeč (nejsou tam workarouny na schopnosti `libpng`, které chybějí ve starších verzích). Procedura také vyžaduje, aby na systému byl nainstalován `Imagemagick` (program `convert`). Během přidání je možno vygenerovat agregované symboly, kde už stačí jen jednoduchá editace GIMPem, a můžeme tak dodělat chybějící akcenty z fontů, které nemají příslušné symboly.

Celý proces probíhá následovně: Uživatel zadá do souboru `Fontmap`, který chce font, a pak pustí skript `makefont`. Výstupem `makefontu` bude hromada souborů ve formátu PNG v adresáři `font/new`. Program `makefont` provádí následující akce:

- 1) Spustí program `genps`, který vygeneruje soubor `letters.ps`.

- 2) Spustí skript `pdfhtml`, který vygeneruje samotná písmenka z `letters.ps`, která nad a pod sebou budou mít pomocné bílé obdélníky. Skript `pdf2html` spouští `ghostscript`, jehož výstup jde přes rouru ve formátu `pbm` (pouze bílé a černé pixely) do programu `pbm2png`, který provede decimaci dat 17x vodorovně a 15x svisle.
- 3) Odstraní odpadky po skriptu `pdf2html` — tento skript byl původně určen pro převod PDF do HTML se sadou PNG obrázků, a z důvodů zjednodušení vývoje je zde použit beze změny.
- 4) Spustí program `clip`. Tento odřeže (za pomoci volání programu `improcess` a `convert`) režijní obdélníky z obrázků písmenek a podle tabulek, které v `clip.c` jsou zaprogramovány, provede přechíslování fontu (z rozsahu 0–255) na Unicode (0000–ffff) a podle další tabulky, která je také zakompilovaná v `clip.c`, spojení některých dvojic znaků fontu do unicodových znaků. Mezi znaky dvojice je vložen kostkovaný blok (`spacer.png`), který slouží k jednoznačnému optickému oddělení znaků pro práci v GIMPu. Je vysoký 112 pixelů, což je také výška výstupních znaků skriptu `makefont`. Pokud uživatel nastaví jinou výšku, musí `spacer.png` změnit, aby procedura vůbec fungovala. Jinak totiž `convert` odmítne znaky spojit.

7.7.1 Kterak postupovat jako uživatel

- Vstoupíme do adresáře `graphics/`
- V `genps.c` můžeme provést nastavení, ale to jen v extrémním případě, a defaultní hodnoty v komentářích necháme, aby se vědělo, co tam patří. Celý systém je totiž seřízený na následující podmínky:
 - Výška obrázku je 112 pixelů.
 - Písmenko X (`0058.png`) na fontu `/CenturySchL` a `/CenturySchL-Bold` je přesně 67 pixelů vysoké, nahoře a dole má ostré přechody mezi 0 (černá) a 255 (bílá), čili tam nejsou šedé pixely, dole je 23 pixelů mezera.

Proto tam nechte defaultní hodnoty:

```
float font_pos=300;
float font_height=392.9619;
float h_margin=100;
float v_margin=120;
float paper_height=842;
```

- Pustíme `make`.
- Pustíme `genps`.
- Upravíme soubor `Fontmap`. Ten obsahuje 1 z následujících typů řádek:

```
/Links-generated (c0590161.pfb) ;
/Links-generated (/usr/local/share/ghostscript/fonts/c059061.pfb);
/Links-generated /CenturySchL-Bold ;
```

Komentáře se dělají pomocí znaku `%` na začátku řádku. Buď se tam může napsat rovnou jméno souboru s fontem (pokud je v adresáři kde ho Ghostscript najde, 1. řádek), nebo cesta (2. řádek) a nebo jméno fontu který už je v Ghostscriptu nadefinovaný (3. řádek). Vybereme si 1 z nich, modifikujeme podle našeho fontu a ostatní zakomentujeme. Když máme font pro ghostscript (většinou `*.pfb` a `*.afm`, to `*.pfb` jsou písmenka a `*.afm` je metrika), tak se dá pomocí pomocných programů a skriptů přidat.

- Zkontrolujeme, jestli v `clip.c` máme napsanou překladovou tabulku pro daný font. Máme-li například japonské písmo Hiragana, podíváme se kde je


```
#ifdef HIRAGANA
```

a tam je tabulka `copy[]` a `merge[]`. `Copy` říká která písmenka se z fontu mají okopírovat rovnou a `merge` říká ty akcentované co se pak budou lepit GIMPem. V `copy[]` je vždy po sobě číslo znaku ve fontu ghostscriptu (0–255) a číslo znaku v unicode (0–65535). V `merge` jsou trojice: číslo znaku v ghostscriptu, číslo příslušného akcentu v ghostscriptu, a číslo v unicode na který se to má zmergovat. Pokud v `clip.c` pro příslušné písmo tyto tabulky nejsou, tak si přidáme `#define` a přepíšeme si je. Můžeme stávající tabulky také opravit, když zjistíme že je v nich chyba. Nejlépe se tabulky píšou tak, že si dáme v Xech do 1 okna `gv letters.ps` a do vedlejšího prohlížeč s unikódovými tabulkami z <http://www.unicode.org/>. Do třetího okna dáme `vi clip.c` a vizuálně hledáme ke každému unikódovému znaku odpovídající znak v `letters.ps` a rovnou to píšeme do zdrojáku `clip.c`.

- V `makefontu` je možno nastavit ořezávání zhora a zdola, a rozlišení. Ale nenastavujte to, jen v případě, že nebude zbytlí. Nechte defaultní hodnoty:

```
export hundred_dpi=1703
export top_promile=198
export bottom_promile=238
```

- Pustíme `makefont` a on nám font vygeneruje do `font/new/`. Pak GIMPem zpracujeme mergeované akcenty. V GIMPu vždy nahrajeme obrázek se sloučeným písmenkem a akcentem (mezi nimi je čtverečkový pruh), přesuneme akcent nad písmenko, odřízneme čtverečkový pruh a část, kde byl akcent, a uložíme.

Pomocí `makefont` se vygenerují do adresáře `font/new/` písmenka která jsou ve fontu přímo. Písmenka s akcentem která tam nejsou, ale je tam to písmenko a akcent se vygenerují že v tom výsledném obrázku je vlevo písmenko, mezi tím takový kostkovaný pruh a vpravo akcent. Pak se vezme GIMP a akcent se přendá nad písmenko do esteticky hodnotné polohy a odřízne se kostkovaný pruh a to, co je vpravo od kostkovaného pruhu. Pak se to uloží do obrázku zpět. Při troše zručnosti takovéto akcentování jde celkem rychle.

- Po zpracování všech sloučených znaků GIMPem je nutno na výsledné soubory pustit korekci pomocí programu `improcess`, protože GIMP nesprávně ukládá gammu a písmenka by byla při velkém zvětšení zbytečně zubatá. Na každý obrázek.png se pustí `improcess -f obrázek.png ' ' obrázek.png`, čímž se násilím nastaví gamma na hodnotu 1.0 a obrázek bude zase v pořádku. Vzniklá PNG přemístíme do nějakého vhodného adresáře ve `font/` a je hotovo.
- Pokud je nahoře nebo dole moc velký nebo moc malý okraj nebo účaří písmen není v 5/24 výšky (odspoda), tak změním nastavení `top_margin` a `bottom_margin` v `makefont`. `top_margin` udává promile výšky stránky A4 které budou zahozeny jako horní okraj, analogicky `bottom_margin`. Chceme-li, aby se měnilo rozlišení výstupního písmene (aby bylo jemnější nebo hrubší), tak se změní `ten_dpi` v `makefont`.
- Pustíme v adresáři `graphics` skript `gen`.
- `cd ..`
- `make` — tím se fonty zakompilují do prohlížeče.

7.8 Přidání nových znaků do existujícího fontu

Přidání nového znaku do již existujícího fontu je jednoduché. Řiďte se následujícím postupem. Pokud možno přidávejte znaky do již existujících adresářů, protože zbytečně mnoho adresářů s fonty by mohlo prohlížeč zpomalit.

- 1) Do adresáře `graphics/font` do příslušného adresáře fontu přidejte PNG soubory pro nové znaky (pro každý znak jeden soubor). Soubory musí být pojmenovány čtyřmístným hexadecimálním unikódovým číslem znaku, přípona musí být `.png`.

V názvu jsou povolena jen malá písmena, název souboru se smí skládat pouze z číslic a písmen a-f. Zde jsou příklady jmen:

23aa.png
0020.png
30bf.png

Soubory musí být šedotónové PNG obrázky, pokud budou barevné, budou prohlížečem zkonvertovány na černobílé a pouze zabírat více místa. V obrázku je bílá interpretována jako „inkoust“, černá jako „papír“, přechod mezi nimi jako příslušné namíchání „inkoustu“ a „papíru“. Na velikosti obrázků nezáleží, všechny obrázky budou zkonvertovány na stejnou výšku. Aspect ratio obrázku se ignoruje, alpha kanál se kombinuje s pozadím. Výška se doporučuje kolem 100 pixelů.

Doporučujeme, aby obrázky byly šedotónové, bez alphy, s gamma 1.0, čtvercovými pixely, bez extra informací jako jsou například komentáře, čas, datum, barevné profily, požadová barva, offset, ... To je vše je z důvodu úspory místa a také proto, že tyto informace jsou ve fontu naprosto zbytečné a nevyužijí se.

Pokud přidáváte nové znaky do monospaced fontů (s pevnou šířkou znaku) musí mít nové obrázky stejný poměr výšky k šířce jako původní obrázky.

- 2) Po přidání všech znaků spusťte v adresáři `graphics` skript `gen`, který vygeneruje ze všech fontových souborů (ze všech PNG) jeden velký C zdroják `font_include.c` s fontem.
- 3) Překompilujte **Links**.
- 4) Nyní máte v prohlížeči nové znaky.

7.9 Přidávání fontů z tištěné předlohy

Pokud chceme přidat do prohlížeče unicode znaky, které jsou pro nás důležité, ale nemáme od nich soubor čitelný Ghostscriptem, ale pouze papírovou předlohu a scanner, stačí písmo nascannovat, nahrát do GIMPu, invertovat, oříznout tak, aby účaří bylo na 5/24 výšky (počítáno odspoda), vhodně stranově oříznout, pomocí `image -> colors -> levels` oříznout zašuměné bílé a černé partie (bez změny gammy, předpokládáme, že scanner jako většina scannerů generuje výstup s gammou 1.0) a uložit do souboru `xxxx.png`, kde `xxxx` je hexadecimální kód Unicode znaku.

7.10 Přidání grafického driveru

Zde najdete ukázkový postup, jak přidat nový grafický driver do **Links**. Postup bude demonstrován na driveru pro framebuffer, přidání jiného driveru je zcela analogické.

- 1) Do skriptu `configure.in` přidejte option na vypínání framebufferu a test na detekci framebufferu (jak vypadá test silně závisí na konkrétním driveru). V případě, že test uspěje (driver se bude kompilovat), definujte v `configure` skriptu makro `GRDRV_FB`, do proměnné `drivers` přidejte název grafického driveru a pokud driver bude potřebovat nějaké knihovny, přidejte je do proměnné `LIBS`. Pro framebuffer bude kód na vypínání framebufferu vypadat takto:

```
AC_ARG_WITH(fb, [ --without-fb compile without Linux Framebuffer
graphics driver], [if test "$withval" = no; then disable_fb=yes;
else disable_fb=no;fi])
```

A kód na detekci takto:

```
if test "$disable_fb" != yes ; then
    AC_CHECK_HEADERS(linux/fb.h)
    AC_CHECK_HEADERS(linux/kd.h)
    AC_CHECK_HEADERS(linux/vt.h)
```

```

AC_CHECK_HEADERS(sys/mman.h)
if test "$ac_cv_header_linux_fb_h" = yes
&& test "$ac_cv_header_linux_kd_h" = yes
&& test "$ac_cv_header_linux_vt_h" = yes
&& test "$ac_cv_header_sys_mman_h" = yes
&& test "$ac_cv_header_sys_ioctl_h" = yes
&& test "$cf_have_gpm" = yes; then
    AC_DEFINE(GRDRV_FB)
    drivers="$drivers FB"
fi
fi

```

- 2) Do souboru `acconfig.h` přidejte řádek

```
#undef GRDRV_FB
```

- 3) Vytvořte soubor `framebuffer.c` a přidejte ho do mezi zdrojáky **Links**.

- 4) Na začátek `framebuffer.c` napište:

```
#include "cfg.h"
```

```
#ifdef GRDRV_FB
#include "links.h"

```

a na konec napište

```
#endif /* GRDRV_FB */
```

- 5) V souboru `framebuffer.c` implementujte všechny potřebné funkce rozhraní grafických driverů a nadefinujte proměnnou `struct graphics_driver fb_driver`, kterou inicialisujte příslušnými funkcemi. Část inicialisace může být přesunuta do funkce `init_driver`. Kód může vypadat například takto:

```

struct graphics_driver fb_driver={
    "fb",
    fb_init_driver,
    init_virtual_device,
    shutdown_virtual_device,
    fb_shutdown_driver,
    fb_get_driver_param,
    fb_get_empty_bitmap,
    fb_get_filled_bitmap,
    fb_register_bitmap,
    fb_prepare_strip,
    fb_commit_strip,
    fb_unregister_bitmap,
    fb_draw_bitmap,
    fb_draw_bitmaps,
    NULL, /* fb_get_color --- filled in fb_init_driver */
    fb_fill_area,
    fb_draw_hline,
    fb_draw_vline,
    fb_hscroll,
    fb_vscroll,
    fb_set_clip_area,
    fb_block,
    fb_unblock,
    NULL, /* set_title */
    0, /* depth (filled in fb_init_driver function) */
    0, 0, /* size (is empty) */

```

```
        GD_DONT_USE_SCROLL,    /* flags */
};
```

- 6) Do souboru `drivers.c` přidejte na začátek (k ostatním driverům):

```
#ifdef GRDRV_FB
extern struct graphics_driver fb_driver;
#endif
```

a do inicialisace pole `graphics_drivers` přidejte na konec před `NULL` řádek s grafickým driverem pro framebuffer. Inicialisace tedy bude vypadat asi takto:

```
struct graphics_driver *graphics_drivers[] = {
#ifdef GRDRV_PMSHELL
    &pmsHELL_driver,
#endif
#ifdef GRDRV_ATHEOS
    &atheos_driver,
#endif
#ifdef GRDRV_X
    &x_driver,
#endif
#ifdef GRDRV_SVGALIB
    &svga_driver,
#endif
#ifdef GRDRV_FB
    &fb_driver,
#endif
    NULL
};
```

- 7) Pusťte `rebuild reconf` a nechte ho doběhnout.
8) Nyní máte přidáný nový driver pro framebuffer.