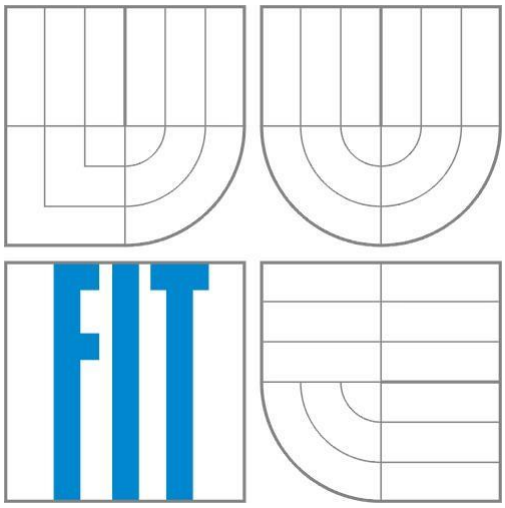


Dokumentace k projektu pro předměty IFJ a IAL

INTERPRET IMPERATIVNÍHO JAZYKA IFJ13

Tým 022, varianta a/3/II

11. 12. 2013

Vedoucí týmu:
Pavel Beran (xberan33): 20%

Členové týmu:
Tomáš Vojtěch (xvojte02): 20%
Martin Fajčík (xfajci00): 20%
Martin Kalábek (xkalab06): 20%
Ondřej Soudek (xsoude01): 20%

Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

Obsah

1 Úvod

1.1 O dokumentu

Tento dokument vznikl jako technická dokumentace k týmovému projektu do předmětů IFJ (Formální jazyky a překladače) a IAL (Algoritmy), jímž byla implementace interpretu imperativního jazyka IFJ13 v jazyce C. Dokumentace si klade za cíl seznámit čtenáře se zvoleným způsobem implementace, strukturou programu, dělením práce v týmu a s implementačními problémy, se kterými se tým v průběhu řešení projektu setkal.

1.2. Struktura dokumentu

Dokument je tematicky dělen na kapitoly a podkapitoly odpovídající jednotlivým částem výsledného programu, případně jednotlivým fázím implementace. Interpret se skládá ze tří hlavních částí, jimiž jsou lexikální analyzátor, syntaktický analyzátor a interpret. Každá z nich je popisována v samostatné části dokumentace. Prostor je věnován také popisu práce v týmu a popisu implementace podpůrných částí interpretu. Na závěr dokumentu je v samostatné kapitole zhodnocen výsledek řešení projektu a jeho přínos.

V textu se nejprve ve druhé kapitole zaobírám problematikou týmové spolupráce, zvolenou metodologii vývoje a způsoby dělení práce v týmu. Ve třetí kapitole se věnuji popisu našeho interpretu a následně postupu implementace jednotlivých částí programu. Čtvrtá kapitola popisuje průběh a metody testování. V páté kapitole poté shrnuji průběh vývoje, implementační problémy a poznatky nabyté prací na projektu. V úplném závěru práce, v šesté kapitole, uvádím metriky kódu naší implementace.

2. Rozbor požadavků a práce v týmu

2.1. Problematika práce v týmu

Zadání specifikovalo potřebu řešit projekt v týmu. Velikost týmu byla variabilní v rozmezí 4-5 osob. Náš z počátku čtyřčlenný tým tak čelil rozhodnutí, zda přibrat pátého člena. Větší tým znamená méně práce připadající na jednu osobu, nicméně více členů týmu zároveň vede k větším komplikacím v rámci komunikace a zvyšuje náročnost vedení týmu. Po delší debatě jsme se rozhodli přibrat pátého člena. Ve výsledku této volby nelitujeme.

2.2. Dělba povinností a komunikace v týmu

Ještě před započetím prvních prací na projektu bylo třeba stanovit jasná pravidla pro korektní fungování práce v týmu. Vzhledem k tomu, že základním problémem při práci v týmu bývají komunikační obtíže, rozhodli jsme se, v rámci jejich předcházení, organizovat v průběhu vývoje pravidelně týmové schůzky.

Tyto schůzky do značné míry připomínaly iterační schůzky ze Scrum metodologie. Na rozdíl od Scrum metodologie však naše vývojové cykly byly kratší a tato schůzka se tak konala vždy zhruba jednou týdně. Schůzka obvykle obsahovala diskuzi o vývoji a objevených implementačních problémech, zhodnocení provedené práce za poslední týden a následné naplánování práce na budoucí týden. Plánování práce probíhalo většinou formou diskuze mezi vedoucím a konkrétním členem týmu.

Takzvané ‚denní schůzky', používané ve zmíněné metodologii, jsme zcela vynechali, neboť by spotřebovávaly příliš mnoho času. Nahradili jsme je internetovou konferencí, která probíhala víceméně nepřetržitě formou skupinové konverzace v internetovém komunikátoru.

Pro statické informace jsme si zřídili online obdobu nástěnky, kam jsme připínali aktuální informace, které by se ve skupinové konverzaci lehce ztratily. Později sloužila také jako obdoba týmové wiki stránky projektu, kam jsme umísťovali schémata jednotlivých částí implementace, což usnadňovalo ostatním členům porozumět práci ostatních.

2.3. Sdílený projektový repozitář

Vzhledem k týmovému charakteru řešeného projektu bylo potřeba zajistit možnost práce několika lidi paralelně. Vzhledem ke značnému množství souborů, z nichž se celek skládá, jsme se rozhodli, že zřídíme pro náš projekt týmový repozitář. Ten nám umožňoval jednak pracovat simultánně a zároveň spravovat jednotlivé změny. Repozitář nám sloužil také jako záloha předchozích verzí souborů. V neposlední řadě byl skvělým nástrojem pro sledování postupu ve vývoji jednotlivých částí interpretu.

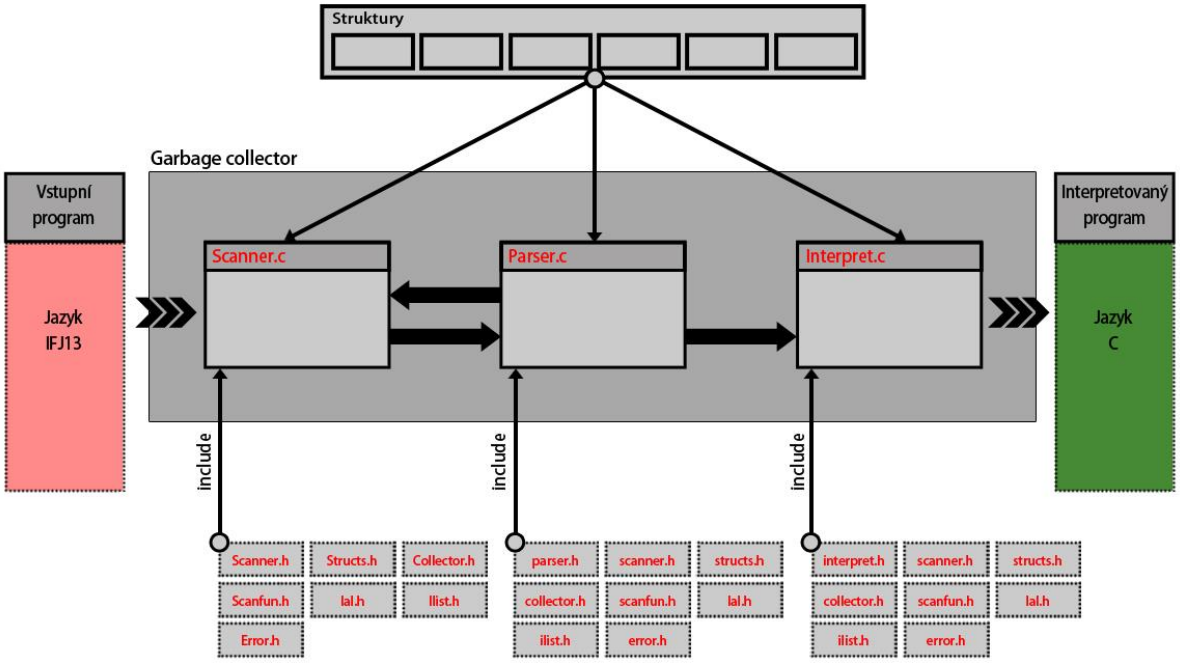
2.4. Online dokument pro plánování a dělbu práce

V pokročilejším stádiu vývoje rostl počet prováděných změn. Změny se odehrávaly jak v kódu, tak často také v návrhu jednotlivých částí. Často se tak kompletně změnil celý princip fungování jednotlivých částí nebo rozhraní mezi nimi. S rostoucím počtem těchto změn již nebylo možné spoléhat na pouhou domluvu a rozhodli jsme se tak pro tvorbu online dokumentu, ve kterém bude zapsáno aktuální rozhraní částí, stav jejich implementace a komu je implementace přidělena.

3. Proces implementace

3.1. Struktura interpretu

Interpret programovacího jazyka je poměrně rozsáhlý a značně složitý program, jehož kompletní a detailní popis by vydal na mnoho stran, a tudíž není pro potřeby dokumentace vhodný. Proto se zde budu věnovat spíše popisu hlavních částí interpretu z hlediska funkčnosti a implementace. Zjednodušené schéma naší implementace zachycuje obrázek [3.1.1] níže.[1]



Obrázek 3.1.1: Schéma interpretu jazyka IFJ13

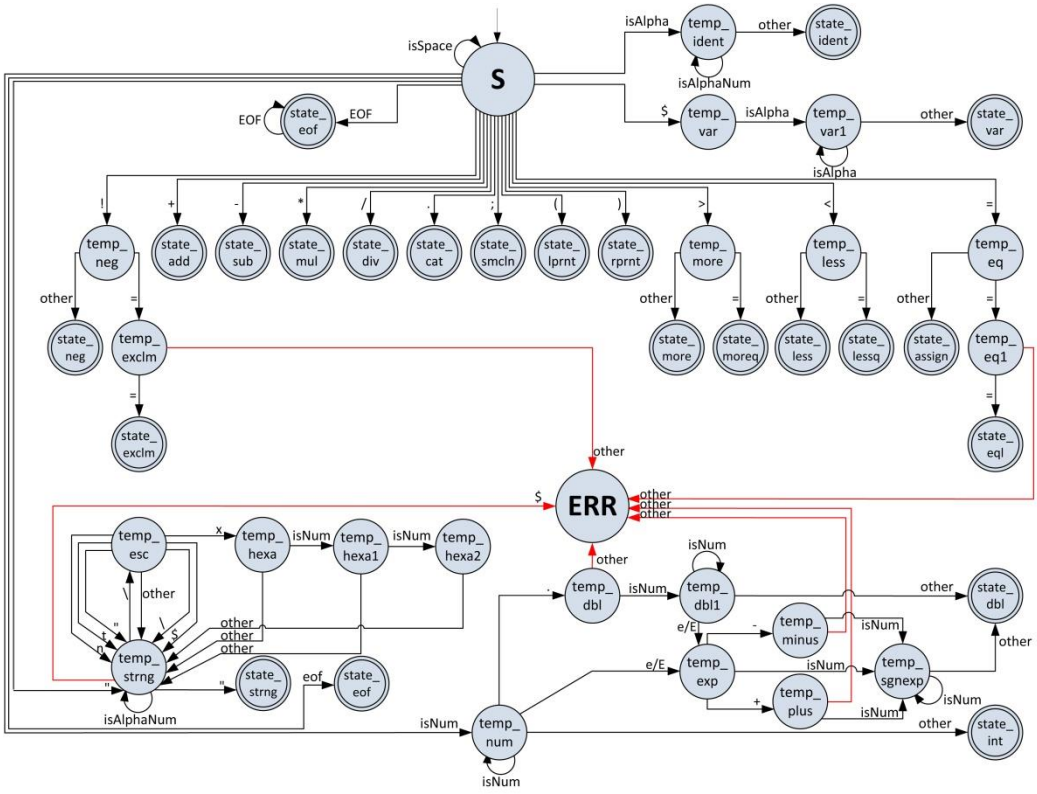
Námi implementovaný interpret jsme rozdělili do tří hlavních oddílů. První implementovanou částí byl scanner[2], který provádí lexikální analýzu. Rozpoznává jednotlivé lexémy vstupního programu a předává je dále ve formě tokenů. Druhou a zároveň nejsložitější součástí interpretu je parser[3], jenž provádí syntaktickou analýzu a veškeré potřebné kroky pro tvorbu posloupnosti tříadresného vnitřního kódu. Součástí parseru je také část sémantické analýzy. Posledním implementovaným podprogramem je interpret tříadresného kódu[4], který se stará o provádění posloupnosti tříadresného vnitřního kódu. Interpret obsahuje zbývající část sémantické analýzy, která nemůže být provedena již v parseru z důvodu dynamického typování jazyka. Lexikální analyzátor byl implementován v předstihu. Části parser a interpret tříadresného kódu byly vyvíjeny v pozdější době a to souběžně, každý jednou vývojovou skupinou. Vývojové skupiny vznikly rozdělením týmu z důvodu úspory času a hlubšímu porozumění implementovanému oddílu. Implementovali jsme také garbage collector, který slouží k ukládání ukazatelů na všechny alokované prvky, aby mohly být později uvolněny bez úniků.

[1] V plné velikosti vložen na konci dokumentu jako příloha [A]. Zde vložená miniatura slouží pouze jako ilustrace.
[2] V textu je na některých místech ve smyslu scanneru uveden „lexikální analyzátor“, pro potřeby dokumentace jsou tyto dva názvy považovány za ekvivalentní.
[3] V textu je na některých místech ve smyslu parseru uveden „syntaktický analyzátor“, pro potřeby dokumentace jsou tyto dva názvy považovány za ekvivalentní.
[4] Je třeba rozlišovat v jakém kontextu je název „interpret“ použit. Zde je uvažováno použití slova ve smyslu jednotky realizující tříadresný kód, nikoliv ve smyslu celku interpreta jazyka.

3.2. Lexikální analyzátor - scanner

Pokud bychom si interpret jazyka představili coby posloupnost dílčích podprogramů, nimiž je zpracovávaný kód vstupního programu postupován, jako první bychom narazili na lexikální analyzátor. Implementace lexikálního analyzátoru se nachází v souborech `scanner.c` a `scanner.h`, k činnosti je povoláván voláním `NextToken()`, případne `UngetToken()` v souboru `parser.c`. Lexikální analyzátor pro svůj běh využívá také funkce implementované v souborech `scanfun.c` a `scanfun.h`. Vstupem lexikálního analyzátoru je zdrojový program ve zdrojovém jazyce. V našem případě byl zdrojovým jazykem imperativní jazyk IFJ13.

Lexikální analyzátor byl v rámci projektu implementován jako konečný automat. Tento konečný automat načítá jeden po druhém znaky ze vstupní posloupnosti znaků zdrojového programu. Na základě přijatých znaků se postupně pomocí adekvátních přechodů přechází mezi jednotlivými stavy konečného automatu a vyhodnocuje tak o jaký lexém se jedná, respektive do jakého koncového stavu se při načtení celého lexému dojde. V případě, že daný lexém není validní, a nelze pomocí něj tedy dojít do koncového stavu jiného, nežli do stavu značícího chybu, je program ukončen s chybovým návratovým kódem.[1]



Obrázek 3.3.1: Diagram konečného automatu lexikálního analyzátoru

Z obrázku [3.3.1][2] diagramu konečného automatu je možné vyčíst, že automat disponuje 35 stavy, včetně počátečního a chybového stavu. Celkem 23 stavů z těchto je koncových, to jest implikací skutečnosti, že existuje právě 23 rozlišitelných typů tokenů. Podle výsledného koncového stavu se načtený lexém zpracuje na odpovídající token.

[1] Před samotným ukončením programu je obstaráno konkrétní uklizení po dosavadním běhu interpretu.
[2] V plné velikosti vložen na konci dokumentu jako příloha [B]. Zde vložená miniatura je zamýšlena pouze jako ilustrace.

```
typedef struct
{
    tState state;
    void* data;
}tToken;
```

Kód 3.2.1: Struktura tToken

Token, jak je z výše uvedeného útržku kódu [3.2.1] zřetelné, je realizován jakožto struktura tToken nesoucí typ tokenu určený dle koncového stavu konečného automatu a jeho data. Ta obsahují dle typu rozpoznaného lexému buďto název, nebo hodnotu.

Lexikální analyzátor nezpracovává celý zdrojový program najednou. Zpracovává lexémy vždy po jednom a ve chvíli, kdy si parser zažádá o další token mu jej předá. V případě, že parser potřebuje další token, zavolá funkci `NextToken()`. Při syntaktické analýze může nastat situace, kdy parser obdrží neočekávaný token. V takovém případě může volat funkci `UngetToken()`, která načtený token uloží na zásobník vytvořený pro tento účel. Funkce `NextToken()` nejprve kontroluje tento zásobník a v případě nálezu tokenu na vrcholu zásobníku pošle parseru právě tento token.

Při návrhu konečného automatu bylo třeba věnovat zvýšenou pozornost stavům obstarávajícím načítání čísel typu double a stavům zpracovávajícím escape sekvence v textových literálech. Tyto dvě části konečného automatu byly nejsložitější a tvořily tak největší překážky v implementaci scanneru, se kterými se musel tým potýkat. Vzhledem k poměrně vysokému počtu stavů, z nichž konečný automat sestává, nebylo snadné zvážit všechny vzniknutelné situace již v počátcích vývoje. Návrh automatu proto musel být v průběhu vývoje několikrát přepracován kvůli odhaleným nedostatkům tak, aby bylo zajištěno korektní a skutečně deterministické chování.

3.3. Syntaktický analyzátor - Parser

Pokud zachováme pohled na interpret coby posloupnost podprogramů, tak se jako druhý v řadě nachází syntaktický analyzátor. Implementován je v souborech `parser.c` a `parser.h` a je spouštěn voláním `parse()` z funkce `main()` implementované v souboru `main.c`. Vstupem parseru jsou tokeny, jenž jsou na vyžádání předávány ze scanneru. Výstupem parseru jakožto celku byla již posloupnost instrukcí trojadresného vnitřního kódu ve formě jednosměrně vázaného seznamu.

Vzhledem ke skutečnosti, že v projektu, jenž byl předmětem implementace, se jedná o tzv. syntaxí řízený překlad, je syntaktický analyzátor hlavní řídící jednotkou celého našeho programu. Lze jej označit za samotné srdce interpretu. Z uvedeného plyne, že syntaktický analyzátor je tedy jednoznačně nejsložitější implementovanou částí celého interpretu. Proto mu bude věnován adekvátní prostor, značně větší nežli prostor vyhrazený pro popis ostatních části interpretu.

Syntaktickou analýzu lze provádět dvěma možnými způsoby. Prvním z nich je takzvaný rekurzivní sestup, neboli jinými slovy metoda „shora dolů“, při níž je derivační strom generován od kořene k listům. Druhým způsobem je metoda zvaná precedenční analýza, nebo také „analýza zdola nahoru“, při které se postupuje při tvorbě derivačního stromu naopak a tedy od listů ke kořenu. Pro naši implementaci interpretu jsme se rozhodli využívat obou těchto metod zároveň. Implementace tak kombinuje využití rekurzivního sestupu pro obecnou syntaktickou analýzu a metodu „zdola nahoru“ pro vyhodnocování výrazů.

3.4. Syntaktický analyzátor - Rekurzivní sestup a LL gramatika

Hlavní část syntaktické analýzy je v našem interpretu realizována pomocí rekurzivního sestupu, který využívá prostředku zvaného LL gramatika. Tou jest množina přepisových pravidel, skládající se z terminálů[1] a non-terminálů[2]. Námi implementovanou LL gramatiku je možné nalézt v příloze [D].

Jak již z názvu vyplývá, syntaktická analýza zde probíhá rekurzivním voláním patřičných funkcí zvolených právě na základě pravidel LL gramatiky, dokud nedojde k úplnému zpracování vstupu, nebo v případě vstupu obsahujícího chybu k syntaktické, případně sémantické (viz. podkapitola 3.7. dále) chybě.

3.5. Syntaktický analyzátor - Precedenční analýza

Zatímco běžný kód je v rámci syntaktické analýzy zpracováván výše popsaným rekurzivním sestupem s využitím LL gramatiky, pro zpracování výrazů nebyl tento postup vhodný, a tak jsme využili precedenční analýzy. Použili jsme algoritmus probíraný na přednášce. Námi implementovaná precedenční analýza je založená na precedenční tabulce uvedené v tabulce [3.5.1] níže.

	+	-	*	/	.	===	!==	>	<	>=	<=	(	)	ID	$
+	G	G	L	L	G	G	G	G	G	G	G	L	G	L	G
-	G	G	L	L	G	G	G	G	G	G	G	L	G	L	G
*	G	G	G	G	G	G	G	G	G	G	G	L	G	L	G
/	G	G	G	G	G	G	G	G	G	G	G	L	G	L	G
.	G	G	L	L	G	G	G	G	G	G	G	L	G	L	G
===	L	L	L	L	L	G	G	L	L	L	L	L	G	L	G
!==	L	L	L	L	L	G	G	L	L	L	L	L	G	L	G
>	L	L	L	L	L	G	G	G	G	G	G	L	G	L	G
<	L	L	L	L	L	G	G	G	G	G	G	L	G	L	G
>=	L	L	L	L	L	G	G	G	G	G	G	L	G	L	G
<=	L	L	L	L	L	G	G	G	G	G	G	L	G	L	G
(	L	L	L	L	L	L	L	L	L	L	L	L	E	L	U
)	G	G	G	G	G	G	G	G	G	G	G	U	G	U	G
ID	G	G	G	G	G	G	G	G	G	G	G	U	G	U	G
$	L	L	L	L	L	L	L	L	L	L	L	L	U	L	U

Tabulka 3.5.1: Precedenční tabulka

3.6. Syntaktický analyzátor - Generátor vnitřního kódu

Zatímco v odborné literatuře je často generátor vnitřního kódu prezentován v podobě samostatné komponenty interpretu, v naší implementaci tomu tak není. Rozhodli jsme se jej implementovat jakožto součást parseru.

Instrukce pro interpret[3] jsou tedy generovány průběžně za běhu syntaktického analyzátoru. Děje se tak jak v rámci rekurzivního sestupu, tak rovněž v případě precedenční analýzy a to voláním funkce `Generate_instruction()`.

Klíčové bylo navrhnout vhodnou instrukční sadu, neboť na ní závisí celková rychlost interpretu. Nevhodně zvolená instrukční sada dokáže také způsobit mnoho komplikací, a proto jsme

[1] Terminálem rozumíme vstupní symbol, který není dále přepisován dle pravidel dané gramatiky.
[2] Non-terminálem rozumíme vstupní symbol, který je přepsán dle některého z pravidel dané gramatiky.
[3] Zde uvažováno ve smyslu interpretu vnitřního kódu, nikoliv interpretu jakožto celku.

jejímu návrhu věnovali značné úsilí. Instrukční sada je implementována společně s funkcemi a strukturami potřebnými pro práci se seznamem instrukcí v souborech `ilist.c` a `ilist.h`. Námi navržená instrukční sada obsahuje celkem 28 typů instrukcí, včetně instrukcí pro vestavěné funkce. O jednotlivých instrukcích více pojednává podkapitola [3.8].

```
typedef struct
{
    int instType; // instruction type
    void *addr1;  // adress 1
    void *addr2;  // adress 2
    void *addr3;  // adress 3
} tInstr;
```

Kód 3.6.1: Struktura tInstr

Jak je z útržku kódu [3.6.1] zřetelné, instrukce obsahuje informaci o tom, o jakou z instrukcí se jedná, tedy jakého je typu a následně tři adresy typu ukazatel na void. Co se bude na které adrese nacházet se odvíjí právě od typu instrukce, neboť rozhraní bylo u jednotlivých instrukcí vymýšleno tak, aby byla práce s instrukcí co nejpohodlnější. Bylo tedy zavrženo univerzální rozhraní, jenž by bylo shodné pro všechny instrukce.

3.7. Sémantický analyzátor

Obdobně jako generátor vnitřního kódu bývá také sémantický analyzátor často popisován jako oddělená komponenta. Při návrhu jsme se však přiklonili k poněkud decentralizovanější variantě implementace, a to zejména z důvodu zjednodušení interpretu coby celku. Implementovali jsme tedy menší část sémantické analýzy již v rámci parseru a zbývající část jako součást interpretu trojadresného vnitřního kódu.

V rámci parseru se provádí některé sémantické kontroly typové kompatibility a pokusy o redefinice funkcí. Zbývající sémantické kontroly, jimiž jsou například kontrola volání nedefinované funkce, kontrola chybějících parametrů při volání funkce, kontrola nedeklarovaných proměnných, kontrola dělení nulou, nebo kontrola chyb při přetypování jsou prováděny až v interpretační části.[1]

3.8. Interpret tříadresného vnitřního kódu

Poslední částí na cestě od zdrojového programu k interpretovanému programu je interpret vnitřního kódu. Je implementován v souborech `interpret.c` a `interpret.h` a je z funkce `main()` v souboru `main.c` zavolán až po dokončení všech aktivit syntaktického analyzátoru a sice pomocí volání `inter()`.

Interpret vnitřního kódu během vývoje podstoupil asi nejradikálnější přepracování ze všech, které jsme v průběhu řešení projektu uskutečnili. To bylo způsobeno tím, že jsme v počátečním návrhu pracovali s myšlenkou implementovat interpret nerekurzivní formou za využití několika speciálních instrukcí v naší instrukční sadě, přímo dedikovaných k realizaci skoků v posloupnosti instrukcí. K rozhodnutí o kompletním přepracování návrhu došlo až v pozdní fázi vývoje, a to poměrně spontánně během jedné z týmových schůzek. Jeden ze členů týmu přednesl, coby řešení jedné z komplikací, návrh na předělání interpretu nerekurzivního na interpret rekurzivní. Předložené

[1] Většina sémantických kontrol je prováděna až v interpretační části, neboť v době, kdy probíhala syntaktická analýza, nebyly ještě k dispozici veškeré informace potřebné pro provedení sémantické analýzy.

řešení se zdálo být natolik elegantní, že po delší diskuzi padlo rozhodnutí interpret skutečně celý přepracovat i přes skutečnost, že již je velká část původního návrhu hotova. Rekurzivní implementace interpretu nám umožňovala snazší předávání parametrů do funkcí a následné předání návratové hodnoty dané funkce zpět. Každá instance interpretu si totiž vytváří svou vlastní pomocnou tabulku symbolů, která je využívána právě pro parametry a návratovou hodnotu funkce. Pohlédneme-li nyní retrospektivně na toto rozhodnutí, hodnotíme jej za korektní i přes nutné zavržení tou dobou již hotové části interpretu.

```
typedef struct
{
  struct listItem *first;
  struct listItem *last;
  struct listItem *active;
} tListOfInstr;
```

Kód 3.8.1: Struktura tListOfInstr

Interpret zpracovává posloupnost instrukcí v podobě jednosměrně vázaného seznamu. Jak je z útržku kódu [3.8.1] patrné, seznam instrukcí obsahuje ukazatel na první, aktivní a poslední prvek seznamu, jenž je typu `listItem`. Pro každou instrukci z naší instrukční sady je implementována jedna větev v řídící struktuře switch.

3.9. Vestavěné funkce – Shell sort

Jednou z vestavěných funkcí, kterými musí dle zadání náš interpret disponovat, je funkce `Sort_string()`. V naší variantě zadání projektu bylo specifikováno využití řadícího algoritmu Shell sort. Jedná se o nejrychlejší řadící algoritmus ze třídy algoritmů kvadratické složitosti.

Shell sort vzdáleně připomíná řadící algoritmus Insertion sort, ovšem s tím rozdílem, že Shell sort disponuje tzv. snižujícím se přírůstkem. Jinými slovy tedy jde o to, že algoritmus neřadí prvky sousední, nýbrž prvky vzdálené právě o tento přírůstek. Postupným snižováním této mezery postupně dojde k seřazení všech prvků řazeného pole. Při dosažení mezery o velikosti jedna se již v podstatě jedná o výše zmíněný Insertion sort algoritmus.

Výhodou tohoto přístupu je rychlé přesunutí nejvyšších a nejnižších hodnot na adekvátní stranu řazeného pole. V posledním kroku, kdy již probíhá obdoba Insertion sortu, se již přesunuje pouze minimální počet prvků.

Problematická je především volba nejvhodnější dekrementace přírůstku. Vzhledem k tomu, že nebyla axiomaticky prokázána žádná „nejlepší“ posloupnost tohoto snižování, je tento výběr zejména dílem experimentování a porovnávání podávaných výsledků. Vhodnou posloupnost lze zvolit zejména pokud jsou na vstupu očekávány nějaké specificky neseřazené, například částečně seřazené posloupnosti. Pokud jsou vstupy zcela náhodné, a nelze tak predikovat jejich typické rozložení neseřazených prvků, bývá často použit Shell sort v základní podobě, tak jak jej roku 1959 definoval jeho autor Donald Shell. V této formě je počáteční mezera definována vztahem *n/2*, kde *n* je počet prvků řazeného pole. V každém kroku poté dojde k celočíselnému dělením dělení dle vztahu *m/2,* kde *m* je aktuální mezera mezi řazenými prvky. Právě tento přístup jsme zvolili v naší implementaci, neboť nelze nijak predikovat, jaké řetězce bude koncový uživatel posílat na vstup.

3.10. Vestavěné funkce – Knuth-Morris-Prattův algoritmus

Algoritmus vychází z myšlenky, že je zbytečné vracet se při neúspěšném porovnání v řetězci zpět a kontrolovat znovu znaky, které již byly porovnány. Za tímto účelem jsme vytvořili tabulku, ve které je zapsáno, se kterým znakem vzoru se má kontrolovat aktuální znak řetězce při neúspěšném porovnání. Tato informace se může lišit pro každý znak vzoru. Znamená to, že při neshodě nemusí procházet již porovnané prvky. Pokračuje ve vyhledávání dle informace z výše zmíněné tabulky. Pro inspiraci nám posloužila ukázková implementace Knuth-Moris-Pratt algoritmu probíraná v přednáškách předmětu IAL.

3.11. Tabulka symbolů

V rámci syntaktické analýzy bylo nezbytné ukládat zpracovávaná data, aby mohla být dále použita při interpretaci programu. Za tímto účelem bylo třeba vytvořit tabulku symbolů. V naší variantě zadání byla tabulka symbolů specifikována jakožto hashovací tabulka. Implementace hashovací tabulky se nachází, včetně všech funkcí pro operace nad ní vykonávané, v souborech `ial.c` a `ial.h`. Ve stejných souborech se nachází také implementace tabulky funkcí.

```
typedef struct tableItem
{
    tKey key;
    tData* data;
    struct tableItem* nextItem;
} tTableItem;
```

Kód 3.11.1: Struktura tTableItem

Z výše uvedeného útržku kódu [3.11.1] je možné vyčíst, z jakých komponent se skládá prvek každý hashovací tabulky. Obsahuje vždy hashovací klíč, jímž je řetězec, dále obsahuje data typu `tData`, jenž jsou popsaná v útržku kódu [3.11.2] uvedeném níže a ukazatel na následující položku tabulky symbolů.

```
typedef struct
{
    int varType;
    void* varValue;
} tData;
```

Kód 3.11.2: Struktura tData

Výše uvedený útržek kódu ukazuje, že data nesená prvkem hashovací tabulky obsahují informaci o typu nesené hodnoty a poté hodnotu samotnou coby void ukazatel. Tímto je umožněno ukládání hodnot libovolného datového typu. Ten je následně používán tak, že se na základě informace o typu ve `varType` přetypuje na adekvátní datový typ.

4. Testování

4.1. Sada testovacích vstupů

Po dokončení implementace všech jednotlivých částí bylo vše třeba důkladně otestovat. Pro tyto potřeby jsme měli nachystánu již v předstihu sadu testovacích vstupů ve formě zdrojových programů v jazyce IFJ13. Ty připravil včetně očekávaných výstupů jeden ze členů týmu již v době, kdy ještě neprobíhala implementace. Těchto testovacích programů bylo připraveno celkem 80, přičemž byly řazeny od nejjednodušších až po některé velice komplikované. To nám umožňovalo ladit chyby postupně.

4.2. Automatický testovací skript

Vzhledem ke skutečnosti, že v době pokusného odevzdávání nebyla ještě dokončena implementace generování vnitřního kódu, posloužilo nám testovací odevzdání pouze jako kontrola lexikálního a sémantického analyzátoru. Neměli jsme tak ještě zhruba týden a půl před odevzdáním procentuální představu o funkčnosti naší implementace a bylo tedy třeba testovat všechny komponenty velice důkladně. Parser byl přitom testován a laděn již v průběhu vývoje a tak s ním v této fázi nebylo mnoho práce. Mohli jsme se tak plně koncentrovat na ladění ostatních částí celku a spoléhat se přitom na správnou činnost parseru.

Za tímto účelem byl dodatečně implementován propracovaný testovací skript, implementovaný v jazyce bash, který umožňoval mimo jiné zapnout automatickou kontrolu programem valgrind. To bylo vhodné pro kontrolu úniků paměti při testovacím běhu interpretu. Skript umožňoval zřetězené spouštění programů z testovací sady. Bylo možné pro tyto programy automaticky dosazovat náhodné vstupy, díky čemuž jsme byli schopni odhalit chyby, které by jinak zůstaly bez povšimnutí. Testovací program zapisoval za běhu své výstupy do souboru, čímž nám poskytoval detailní informace o vzniklých chybách.

4.3. Ruční testování a debugování

Kromě výše uvedeného automatizovaného testování bylo nutné testovat také ručně. Zejména bylo nutné, v případě chyb, u nichž nebylo možné odhadnout, čím jsou způsobeny, program krokovat pomocí debuggeru.

Testování bylo prováděno jak na platformě Microsoft Windows (ve verzích Windows 8 PRO 64bit, Windows 7 64bit a Windows 7 32bit), tak na platformě Linux, kde bylo využito hned několik různých distribucí, konkrétně Fedora 19 64bit, Mint 15 64bit , Ubuntu 13.04 64bit a virtualizovaně také Ubuntu 13.10 32bit.

5. Závěr

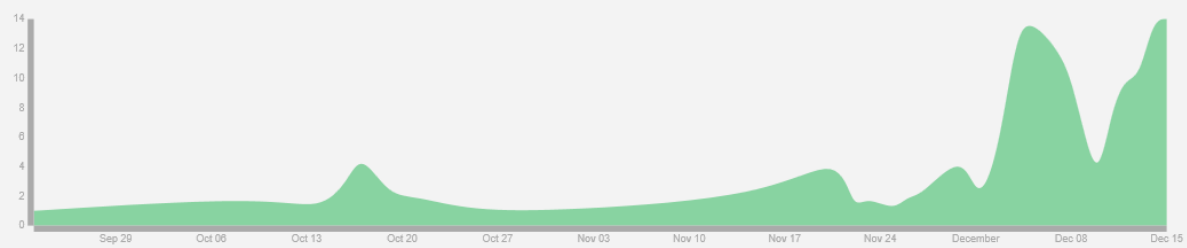
5.1. Shrnutí zdaření implementace

Během zhruba dvou měsíců, které jsme měli na implementaci interpretu vyhrazeny, se nám podařilo interpret jazyka IFJ13 úspěšně implementovat. Výsledný program vyhovuje specifikacím popsaným v zadání. Podává korektní výstupy při vložení ukázkových programů v jazyce IFJ13 uvedených v zadání.

5.2. Shrnutí průběhu vývoje

Na implementaci interpretu jsme měli přibližně dva měsíce, což je adekvátní doba pro vývoj programu takovéhoto rozsahu a složitosti. Komplikací však byla neznalost dané problematiky v počátcích vývoje. Oblast formálních jazyků a překladačů je skutečně zajímavá, avšak pro většinu řešitelů zcela neznámá. S problematikou překladače se všichni členové týmu setkali poprvé až při tvorbě tohoto projektu. Velkou pomocí nám tak byl poskytnutý jednoduchý interpret.

Znamenalo to pro nás zejména velmi pozvolný začátek vývoje, neboť bylo potřeba nejprve vstřebat látku probíranou v předmětech IFJ a IAL. Proto byla velká část (zejména v počátcích vývoje) doby vyhrazené na implementaci projektu nevyužita, respektive nevyužita pro samotnou implementaci. Zhruba polovina z této doby byla využita na samostudium dané problematiky, neboť nebylo vzhledem k uzávěrce projektu možné čekat na kompletní výklad ve výše zmíněných předmětech. Pokud bychom čekali na dokončení výkladu látky o překladačích, zcela jistě bychom nestihli včas implementovat kompletní program.



Obrázek 5.2.1: Graf commitů do repozitáře

Implementace tedy probíhala zejména v druhé polovině vyhrazené doby, což názorně ukazuje výše uvedený obrázek [5.2.1] četnosti commitů do sdíleného repozitáře. Z obrázku je také patrný pozvolný počátek implementace, plynoucí z postupně rostoucích znalostí dané problematiky. Vrchol vývoje je patrný až v posledních třech týdnech vývoje, kde poslední týden tvořilo zejména testování a ladění již hotového interpretu.

5.3. Zhodnocení přínosnosti práce na projektu

Projekt byl jak rozsahem, tak složitostí několikanásobně náročnější, než ostatní námi doposud implementované projekty. Umožnil nám tak vyzkoušet si práci na projektu obdobného rozsahu a náročnosti, s jakými se lze setkávat v praxi, a to od úplného počátku vývoje, tedy od návrhu až po kompletní odladění výsledného programu.

Další a pravděpodobně nejpřínosnější vlastností projektu byla nutnost pracovat v týmu. Práce v týmu je u rozsáhlejších projektů a takřka při jakémkoliv vývoji v praxi naprosto nevyhnutelná. Proto tyto nabyté zkušenosti skutečně oceňujeme.

6. Rozdělení práce a metriky kódu

6.1. Rozdělení práce

- **Pavel Beran:**
  - Vedení týmu
  - Dělení práce
  - Syntaktický analyzátor
- **Tomáš Vojtěch:**
  - Návrh lexikálního analyzátoru
  - Interpret
  - Tvorba dokumentace
- **Martin Fajčík:**
  - Návrh lexikálního analyzátoru
  - Lexikální analyzátor
  - Syntaktický analyzátor
- **Martin Kalábek:**
  - Vedení vývoje interpretu
  - Interpret
  - Testování
- **Ondřej Soudek:**
  - Interpret
  - Testovací program
  - Testování

6.2. Metriky kódu

Počet souboru:
20 (bez testovacích programů)

Počet řádků zdrojového kódu:
4863 (bez testovacích programů)

Počet commitů do repozitáře:
174

Velikost statických dat:
169KB (bez testovacích programů)

Velikost spustitelného programu:
63KB (systém Windows 7, 64bitová architektura, překlad bez ladících informací)

7. Použité zdroje informací

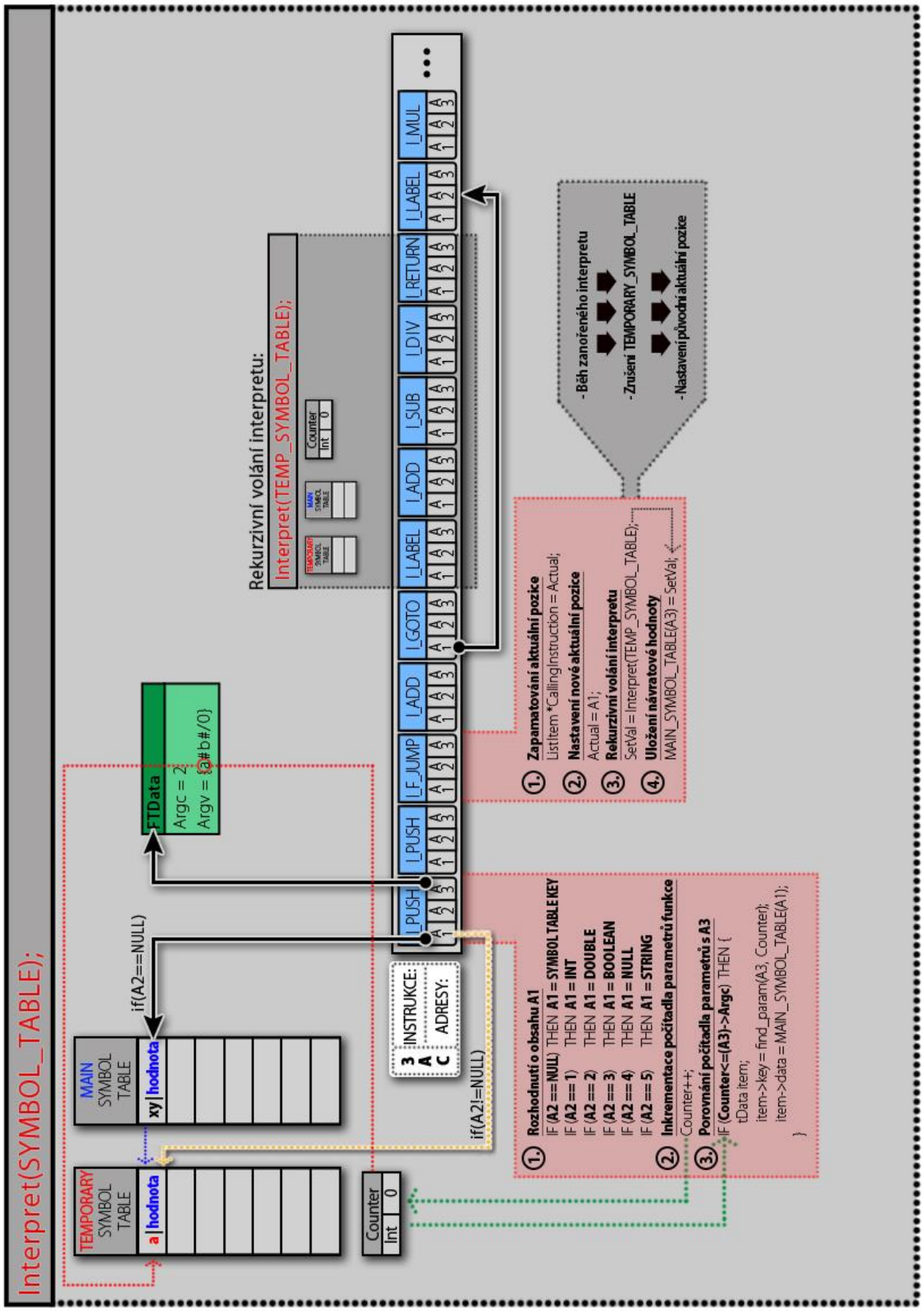
[1] C. Prof. Ing.Jan M Honzík, Studijní opora k předmětu ALGORITMY, Brno, 2012.

[2] A. Meduna a R. Lukáš, Studijní opora k předmetu Formální jazyky a překladače, Brno, 2012.

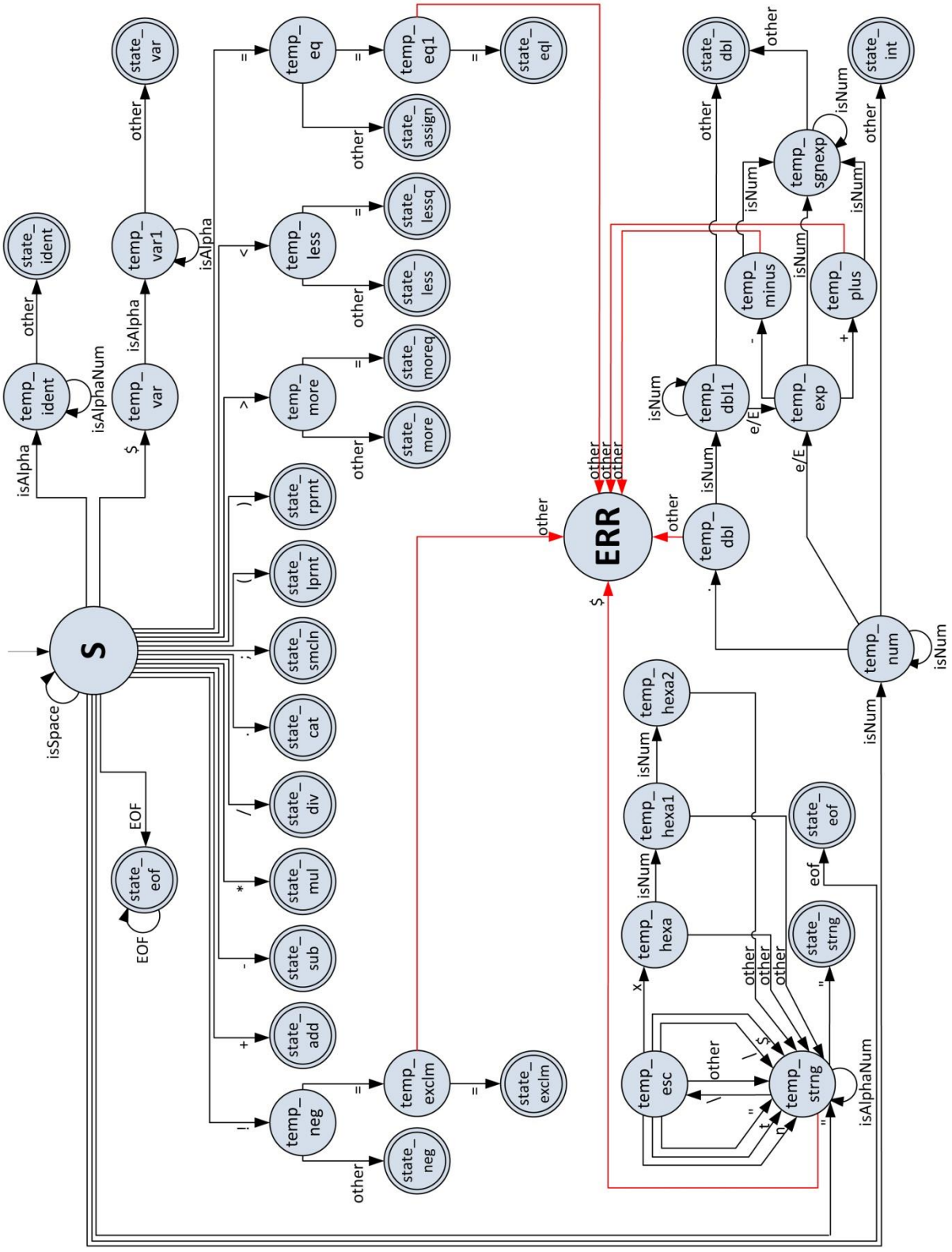
[3] P. Mička, „Algoritmy.net - Příručka vývojáře,“ 2010. [Online]. Available: http://www.algoritmy.net/article/154/Shell-sort. [Přístup získán 1 12 2013].

8. Přílohy

A Schéma rekurzivního interpretu trojadresného kódu



B Diagram konečného automatu lexikálního analyzátoru



C LL gramatika

[1:]	<PROG>		<?php	<BODY>	eof						
[2:]	<BODY>		eps								
[3:]	<BODY>		<STAT>	<DEKFUN>	<STAT>	<BODY>					
[4:]	<DEKFUN>		eps								
[5:]	<DEKFUN>		function	fid	(	<IDLIST>	)	{	<STAT>	}	
[6:]	<IDLIST>		eps								
[7:]	<IDLIST>		id	<IDN>							
[8:]	<IDN>		eps								
[9:]	<IDN>		,	id	<IDN>						
[10:]	<STAT>		id	=	<EXPRF>	;	<STAT>				
[11:]	<STAT>		eps								
[12:]	<STAT>		if	(	<EXPR>	)	{	<STAT>	}	<ELSE>	<STAT>
[13:]	<ELSE>		eps								
[14:]	<ELSE>		else	{	<STAT>	}					
[15:]	<STAT>		return	<EXPR>	;	<STAT>					
[16:]	<STAT>		while	(	EXPR	)	{	<STAT>	}	<STAT>	
[17:]	<FUN>		function	fid	(	<TERMLIST>	)				
[18:]	<TERMLIST>		eps								
[19:]	<TERMLIST>		<TERM>	<TERMN>							
[20:]	<TERMN>		,	<TERM>	<TERMN>						
[21:]	<TERMN>		eps								
[22:]	<EXPRF>		<FUN>								
[23:]	<EXPRF>		<INLINEFUN>								
[24:]	<EXPRF>		<EXPR>								
[25:]	<EXPR>		expression								
[26:]	<EXPR>		<TERM>								
[27:]	<LITEXP>		int								
[28:]	<LITEXP>		double								
[29:]	<LITEXP>		bool								
[30:]	<LITEXP>		<STRFORM>								
[31:]	<TERM>		null								
[32:]	<TERM>		<LITEXP>								
[33:]	<STRFORM>		str								
[34:]	<STRFORM>		id								
[35:]	<STRFLIST>		)								
[36:]	<STRFLIST>	,	<TERM>	<STRFLIST>							
[37:]	<INLINEFUN>		boolval	(	<TERM>	)					
[38:]	<INLINEFUN>		doubleval	(	<TERM>	)					
[39:]	<INLINEFUN>		intval	(	<TERM>	)					
[40:]	<INLINEFUN>		strval	(	<TERM>	)					
[41:]	<INLINEFUN>		get_string	(	)						
[42:]	<INLINEFUN>		strlen	(	<TERM>	)					
[43:]	<INLINEFUN>		get_substring	(	<TERM>	,	<TERM>	,	<TERM>	)	
[44:]	<INLINEFUN>		find_string	(	<TERM>	<STRFLIST>					
[45:]	<INLINEFUN>		sort_string	(	<TERM>	)					
[46:]	<INLINEFUN>		put_string	(	<TERM>	<STRFLIST>					