VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu do předmětů IFJ a IAL

Interpret jazyka IFJ16

Tým 078, varianta b/3/II

11. 12. 2016

Rozšíření:
BASE
FUNEXP
SIMPLE
BOOLOP

Autoři a podíl na projektu:

Miroslava Míšová (vedoucí)	(xmisov00)	25 %
Jiří Matějka	(xmatej52)	25 %
Sava Nedeljković	(xnedel08)	25 %
Nemanja Vasiljević	(xvasil03)	25 %
Kryštof Michal	(xmicha64)	0 %

Úvod

Tato dokumentace slouží k popisu naší implementace interpretu jazyka IFJ16, který je založen na jazyce Java. Interpret se skládá ze tří hlavních částí, které budou popsány v dalších kapitolách:

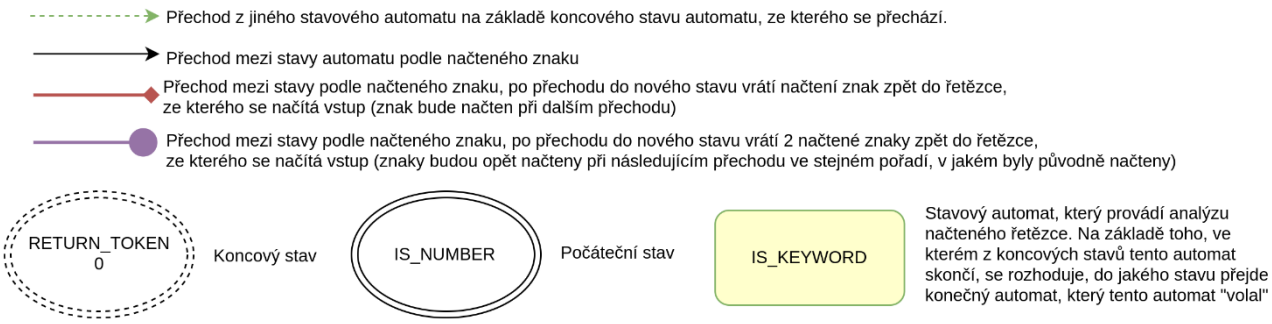
- Lexikální analyzátor
- Syntaktický, sémantický analyzátor a generátor instrukcí
- Interpret

Struktura projektu:

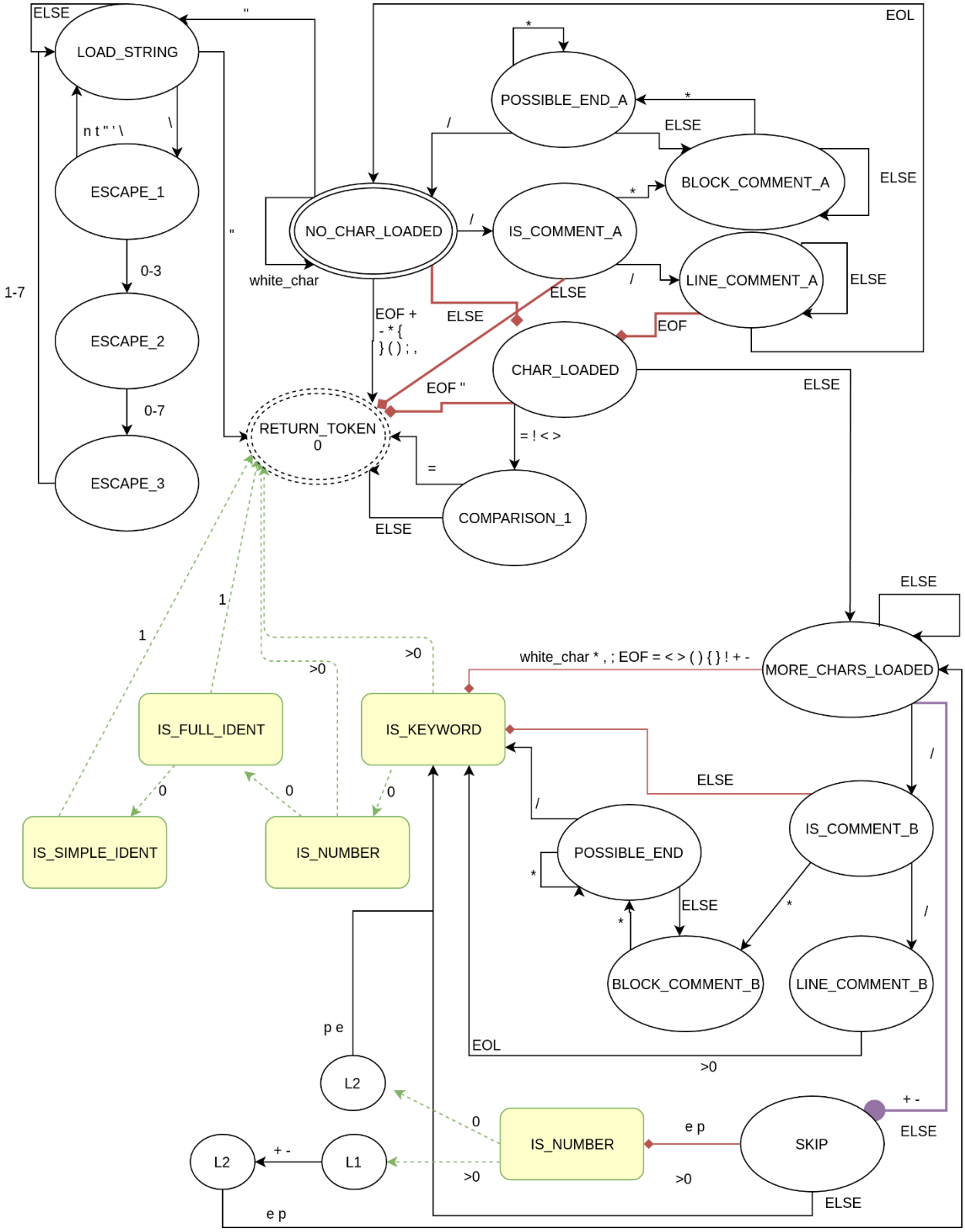
Interpret je rozdělen do několika částí: Lexikální analyzátor, syntaktický analyzátor, precedenční analyzátor výrazů a interpret. Dále jsou k projektu připojeny moduly pro práci s abstraktními datovými typy (zásobník, tabulka s rozptýlenými položkami a pole s proměnou velikostí), pamětí a modul s vestavěnými funkcemi interpretu.

Lexikální analyzátor

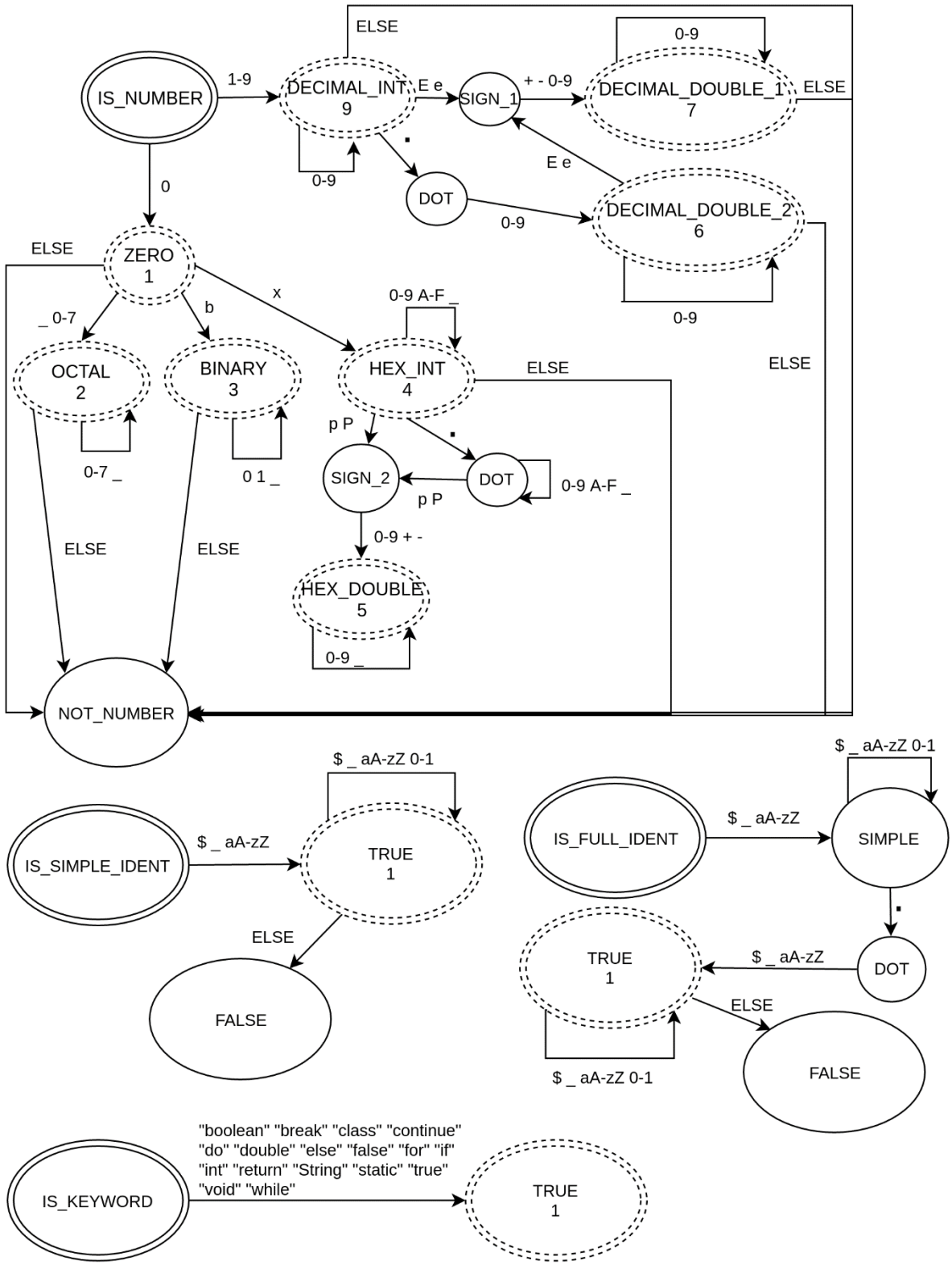
Lexikální analýza je založena na několika stavových automatech. Ve zdrojovém kódu jsou automaty reprezentovány jednotlivými funkcemi a přechody mezi stavy jsou uskutečněny pomocí switch, if a else sekvencí. Načítání vstupu je reprezentováno pomocí cyklu, ať už cyklu, který čte soubor, nebo cyklů, které procházejí pole načtených znaků. Vstup se postupně načítá do pole znaků, a každý načtený znak prochází stavovým automatem na základě lexikálních pravidel jazyka (viz konečný automat č. 1). V případě načítání čísel, klíčových slov, jednoduchých a složených identifikátorů, nebo neplatného lexému, je načtený lexém (reprezentovaný jako řetězec) poslán na vstup dalších konečných automatů, které rozhodnou, zda je lexém platný, a o jaký lexém se jedná. V případě lexikální chyby (konečný automat č. 1 se nedostal do koncového stavu) je vytisknuta chyba a vrácen token symbolizující neplatný lexém. Samotný token při úspěšném načtení vstupu je reprezentován jeho typem (načtený lexém je klíčové slovo, proměnná, operand, …) a ukazatelem buď na hodnotu načteného čísla, název proměnné či funkce, načtený řetězec, nebo nil ukazatelem.



Legenda ke konečnému automatu



Konečný automat č. 1



Konečný automat č. 2

Syntaktický a sémantický analyzátor

Aktivita Syntaktického a sémantického analyzátoru (dále jen SSA) se dělí na dva průběhy. V prvním průběhu se kontroluje primárně správná konstrukce pomocí LL gramatiky (syntaxe jazyka) a ukládají se statické proměnné a funkce do globální tabulky symbolů. Na základě postupně ukládaných dat se již rozhoduje o prvních sémantických chybách programu. V druhém průběhu se kontrolují sémantická pravidla, ukládají se data do lokálních tabulek symbolů, volá se kontrola výrazů a generuje se vnitřní kód.

Vstupem SSA jsou jednotlivé tokeny, které jsou získávány z lexikálního analyzátoru pomocí *get_token()*. V případě načtení neplatného tokenu je ukončena práce SSA s návratovým kódem lexikální chyby a návrat do modulu *main.c*. V jiném případě je vstupní token zpracován a simuluje se tvorba derivačního stromu. Za předpokladu, že se derivační strom podařilo vytvořit, je program syntakticky správně. Derivační strom se vytváří rekurzivně směrem shora dolů, na základě LL gramatiky.

SSA ukládá data o proměnných a funkcích do tabulek symbolů, které jsou reprezentovány tabulkami s rozptýlenými položkami. Globální tabulka symbolů je uložena na zásobník tabulek symbolů, který dále bude používán interpretem. Každá z lokálních tabulek symbolů náleží vždy jedné funkci a k té je také následně připojena.

V druhém běhu SSA se průběžně generuje tří adresný kód a ten se ukládá na hlavní instrukční pásku v případě inicializace statických proměnných, nebo na instrukční pásku jednotlivých funkcí. Tu si v případě volání dané funkce kopíruje interpret na hlavní instrukční pásku.

START	CLASS
CLASS	class simple_ident { CLASS_BODY } CLASS
CLASS	{e}
CLASS_BODY	static TYPE simple_ident DEF CLASS_BODY
CLASS_BODY	{e}
TYPE	void
TYPE	int
TYPE	double
TYPE	string
TYPE	bool
DEF	(DEF_ARGUMENTS) {FUNC }
DEF	;
DEF	= EXPR;
DEF_ARGUMENTS	TYPE simple_ident DEF_ARGUMENTS_2
DEF_ARGUMENTS	{e}
DEF_ARGUMENTS_2	, TYPE simple_ident DEF_ARGUMENTS_2
DEF_ARGUMENTS_2	{e}
FUNC	TYPE simple_ident VAR_EXPR FUNC
FUNC	FUNC_BODY FUNC
FUNC	{e}
FUNC_BODY	simple_ident GUIDANCE
FUNC_BODY	full_ident GUIDANCE
FUNC_BODY	return RETURN_EXPR
FUNC_BODY	if (EXPR) IF_ELSE_SECTION ELSE_EXISTANCE
FUNC_BODY	while (EXPR) IF_ELSE_SECTION
VAR_EXPR	;
VAR_EXPR	= EXPR;
GUIDANCE	;
GUIDANCE	= EXPR;
GUIDANCE	(EXPR ;
RETURN_EXPR	;
RETURN_EXPR	EXPR ;
ELSE_EXISTANCE	else IF_ELSE_SECTION
ELSE_EXISTANCE	{e}
IF_ELSE_SECTION	{ FUNC_BODY_pom_1 }
IF_ELSE_SECTION	FUNC_BODY
FUNC_BODY_pom_1	FUNC_BODY FUNC_BODY_pom_1
FUNC_BODY_pom_1	{e}

Precedenční analýza výrazu

Precedenční analýza je použita ke zpracování výrazu. Jakmile syntaktický analyzátor narazí na výraz, zavolá precedenční analyzátor. Precedenční analyzátor potom daný výraz zpracuje podle tabulky pravidel, a v případě, že nikde nenastala chyba, vrátí zpracovaný výraz. Výsledný výraz je posloupností tokenů ve formátu postfix.

Precedenční analyzátor ke své činnosti potřebuje i lexikální analyzátor, od kterého zažádá o další token pomocí funkce *get_token()*. Každý zpracovaný token precedenční analyzátor musí najít v tabulce symbolů nebo v lokální nebo globální tabulce symbolů, kde jsou uloženy informace o deklarovaných proměnných a funkcích.

Všechna pravidla ke zpracování výrazu se nacházejí v precedenční tabulce, která je nezbytnou součástí precedenční analýzy. Algoritmus zpracování výrazu je realizován cyklem, popř. rekurzivním voláním. Ke zpracování a redukování pravidel je použita struktura zásobník.

	+	-	*	/	(	)	i	f	,	$	==	>	<	>=	<=	!=	!	&&	||
+	>	>	<	<	<	>	<	<	>	>	>	>	>	>	>	>	>	>	>
-	>	>	<	<	<	>	<	<	>	>	>	>	>	>	>	>	>	>	>
*	>	>	>	>	<	>	<	<	>	>	>	>	>	>	>	>	>	>	>
/	>	>	>	>	<	>	<	<	>	>	>	>	>	>	>	>	>	>	>
(	<	<	<	<	<	=	<	<	=		<	<	<	<	<	<	<	<	<
)	>	>	>	>		>			>	>	>	>	>	>	>	>	>	>	>
i	>	>	>	>		>			>	>	>	>	>	>	>	>	>	>	>
f					=						>	>	>	>	>	>	>	>	>
,	<	<	<	<	<	=	<	<	=		>	>	>	>	>	>	>	>	>
$	<	<	<	<	<		<	<			<	<	<	<	<	<	<	<	<
==	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
>	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
<	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
>=	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
<=	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
!=	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
!	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
&&	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>
||	<	<	<	<	>	>	<	<	<	<	>	>	>	>	>	>	>	>	>

Interpret

Interpret provádí interpretaci tří adresného kódu, který je generován syntaktickým analyzátorem. Tří adresný kód je uložen v instrukční pásce, která je implementována pomocí abstraktního datového typu jednosměrně vázaného lineárního seznamu. Pro řešení cyklu while, se používá zásobník, na který ukládá ukazatel na danou instrukci. Když interpret narazí na instrukci, která ukončuje cyklus while, z vrcholu zásobníku vezme instrukci a nastaví ji jako následující zpracovávanou. Volání funkce v interpretu je řešeno pomoci rekurzivního volání interpretu. U každého volání funkce vytváří kopii lokální tabulky pro danou funkci, kterou posune na vrchol zásobníku tabulek. Interpret pracuje pouze s tabulkou, která se nachází na vrcholu (Lokalní tabulka) a s tabulkou, která je uložená jako první (Globální tabulka). Při interpretaci se ošetřuje, zda je proměnná neinicializovaná, dělení nulou a neplatný vstup.

Vybrané algoritmy z pohledu předmětu IAL

Boyer-Mooreův algoritmus

Vestavenou funkci *find()* jsme implementovali na základě Boyer-Mooreova algoritmu. Rozhodli jsme se použít *bad-character* heuristiku. Algoritmus porovnává vyhledávaný podřetězec v řetězci zprava doleva. Pokud tedy narazíme v textu na znak, který daný podřetězec vůbec neobsahuje, můžeme se posunout o celou délku vyhledávaného podřetězce dále.

I	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
text-->		F	I	N	D	I	**N**	A	H	A	Y	**S**	T	A	C	K	**N**	**E**	**E**	**D**	**L**	**E**	I	N
0	5	N	E	E	D	L	**E**	<----- *pattern*																
5	5						N	E	E	D	L	**E**												
11	4												N	E	E	D	**L**	**E**						
15	0																**N**	**E**	**E**	**D**	**L**	**E**		

return i

Shell sort

Shell sort je jednoduchý algoritmus řazení typu in situ. Prvky, které jsou dál od sebe, se seřadí dříve než prvky, které jsou blíže. Shell sort je druh řazení vkládáním, které povoluje výměnu prvků, které nejsou vedle sebe (vzdálených prvků). Je to ale nestabilní řadicí metoda tj. nezachovává původní pořadí dvou prvků se stejným klíčem.

Hlavní myšlenkou je seřazení částí posloupnosti prvků. V každém dalším průchodu se velikost zpracovávané části posloupnosti zmenšuje a zároveň jednotlivé zpracovávané části jsou už částečně seřazené.

Efektivita shell sort algoritmu je závislá na mezeře zpracovávaných části, která se nazývá gap sekvence, a v naší implementace je zvolená jako velikost pole prvků, která se každým průchodem zmenšuje na polovinu.

Tabulka s rozptýlenými položkami

Tabulka s rozptýlenými položkami je datová struktura, která je založena na ukládání klíče, který má v tabulce jednoznačnou hodnotu a slouží k identifikaci položky tabulky. V případě, že se dva a více klíčů namapují do stejného místa, dochází ke kolizi. Těmto klíčům říkáme synonyma. V naší implementaci jsme užili explicitně zřetězeného seznamu synonym. Má podobu pole, v němž jsou uloženy ukazatele na začátky seznamů synonym reprezentovaných lineárním seznamem.

Vývoj

Rozdělení práce

Jiří Matějka – Lexikální analyzátor, správa Git repozitáře, testování, abstraktní datové typy, funkce pro správu paměti a pomocné funkce
Miroslava Míšová – Tabulka s rozptýlenými položkami, SSA s generováním vnitřního kódu, dokumentace
Sava Nedeljković – Vestavěné funkce, precedenční analýza, zásobník pro SSA
Nemanja Vasiljević – Interpret, vestavěné funkce
Kryštof Michal (nesplněno) – Postfix, testování, dokumentace, pomocné funkce

Práci jsme se již od začátku snažili rozdělit rovnoměrně, avšak kvůli pár problémům jsme museli některé úkoly přerozdělit.

Závěr

Projekt bereme jako velký přínos našeho studia. Naučil nás jak v praxi efektivně využívat abstraktní datové typy a týmové spolupráci.

Metriky

celkem:

- 7744 řádků kódu
- 153 testovacích souborů
- 777 commitů na Gitu

Literatura

- Manuálové stránky Linuxu
- Forum: www.stackoverflow.com
- Studijní opora předmětu IAL
- Projekty do předmětu IJC, IAL
- Přednášky předmětu IFJ
- Sara Baase, Allen Van Gelder, Computer algorithms: introduction to design and analysis. 1978