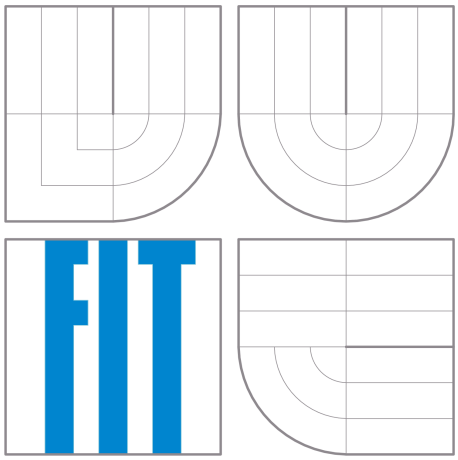
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta informačních technologií



Dokumentace k projektu z IFJ

Interpret jazyka IFJ2011

Tým 036, varianta a/2/II

Výčet rozšíření:

- LENGHT Unární operátor, který vypočítá délku řetězce.
- REPEAT Podpora cyklu repeat-until

Autoři: Riša Martin, xrisam00, 25%

Riša Michal, xrisam01, 25%

Rudolf Josef, xrudol04, 25%

Tomáš Válek, xvalek02, 25%

11.12.2011

Obsah

1. Úvodem

Tato dokumentace popisuje chování a princip interpretu jazyka IFJ2011, jeho implementaci a také problémy, které se vyskytly při řešení.

Vybrali jsme si variantu a/2/II, což znamená implementovat vyhledávání podřetězce v řetězci algoritmem Knuth-Moris-Pratt , řazení pomocí Heap sort algoritmu a tabulku symbolů pomocí abstraktní datové struktury hashovací tabulka.

Interpret jako celek se skládá ze tří hlavních částí

- Lexikální analyzátor
- Syntaktický a sémantický analyzátor
- Interpret

Každá jeho část bude dále velmi stručně popsána.

2.1. Lexikální analyzátor

Lexikální analyzátor provádí odstranění komentářů a bílých znaků. Vstupem lexikálního analyzátoru je zdrojový program a výstupem jsou tokeny.

Lexikální analyzátor pro projekt IFJ je implementován pomocí konečného automatu. Automat přechází mezi stavy a pokud neskončí v koncovém stavu, ohlásíme chybu lexikálního analyzátoru.

Popis implementace:

Lexikální analyzátor si umí správně spočítat řádky a vypsat na kterém řádku k chybě došlo.

Pokud se ve zdrojovém textu vyskytnou ASCII znaky s hodnotou menší než 32, na *stderr* se vypíše varování o nalezeném znaku a znak se ignoruje. Samozřejmě vyjma znaků: Backspace, Line Feed (LF), Form Feed (FF), Carriage Return (CR), atd.

Konečný automat pro lexikální analyzátor obsahuje celkem 41 stavů. Z prostorových důvodů je rozdělen na logické celky a je zbaven detailů.

Vysvětlivky k diagramům:

- elipsa značí stav, dvojitá elipsa značí koncový stav
- přerušovaná čára znamená, že stav pokračuje na jiném obrázku

K-A hlavní schéma:

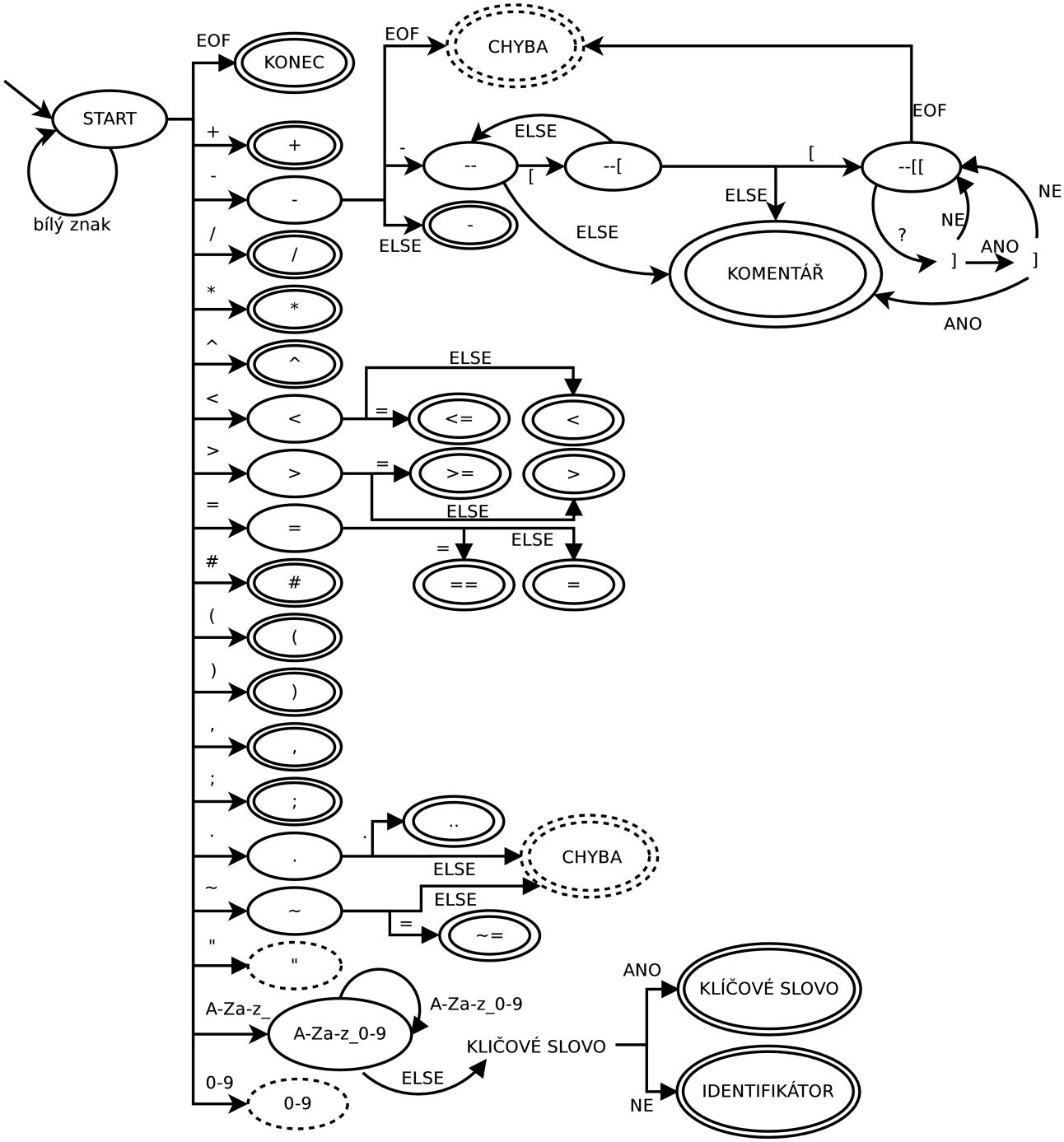


diagram 1.1

K-A pro chybu:

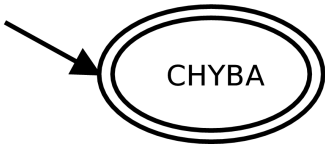
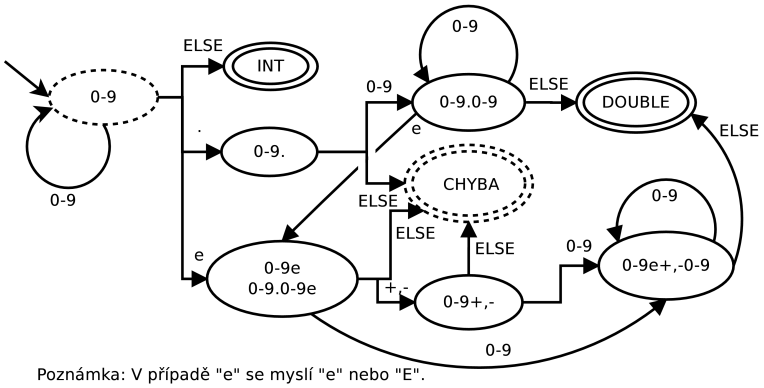


diagram 1.2

K-A pro číselný literál:



Poznámka: V případě "e" se myslí "e" nebo "E".

diagram 1.3

K-A pro řetězcový literál:

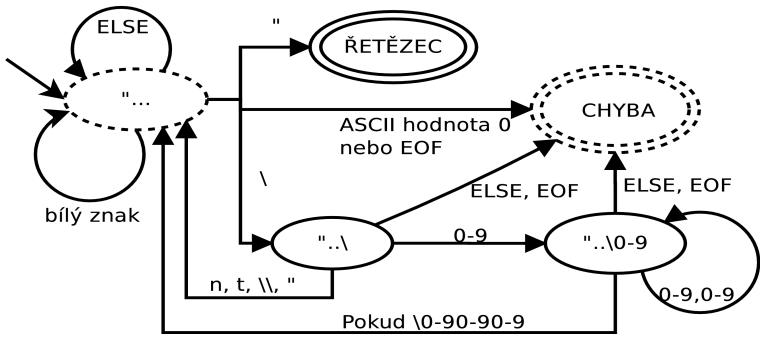


diagram 1.4

2.2. Syntaktický a sémantický analyzátor

Při zpracování deklarací funkcí a výroků funkcí používáme rekurzivní prediktitivní syntaktickou analýzu, která je implementovaná cyklem volaní funkcí *statement()* a *declaration(),* který končí parsováním funkce *main*. Jedná se v obou případech o konečné automaty, které načítávají *tokeny* po jednom. Pro zpracovaní konstrukcí jazyka jako *if-else-end*, *repeat-until* a *while-end* je použitý zásobník *statement_stack,* do kterého se ukládají adresy nekompletních skokových instrukcí. Pro zpracování konstrukce *return* se používá zásobník *return_stack*, který obsahuje adresy nekompletních skokových instrukcí. Po načtení dostatečného počtu *tokenů*, aby mohla být vygenerována instrukce, se instrukce vygeneruje.

Zpracovávaní výrazů je implementované na základě prediktivní syntaktické analýzy pomocí převodu notace (*infix_to_postfix*) a zpracování na instrukce (*postfix_parse*) jsou implementovány ve dvou nezávislých funkcích, o jejichž volání se stará obalovací funkce *parse_expression*. Struktura obou hlavních funkcí v konečné podobě připomíná hybrid mezi zásobníkovým automatem a obyčejným cyklem.

Priorita a asociativita operátorů jsou řešeny tabulkou, která je ale oproti formálně správné precedenční tabulce velmi zjednodušená a obsahuje jen prioritu operátoru a asociativitu. **Precedenční tabulka** by vypadala asi takto:

	+	-	*	/	^	<	>	<=	>=	==	~=	..	#	(	)	ID	I	D	S	F	T	N	$
+	>	>	<	<	<	>	>	>	>	>	>	>	<	<	>	<	<	<					>
-	>	>	<	<	<	>	>	>	>	>	>	>	<	<	>	<	<	<					>
*	>	>	>	>	<	>	>	>	>	>	>	>	<	<	>	<	<	<					>
/	>	>	>	>	<	>	>	>	>	>	>	>	<	<	>	<	<	<					>
^	>	>	>	>	<	>	>	>	>	>	>	>	<	<	>	<	<	<					>
<	<	<	<	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<				>
>	<	<	<	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<				>
<=	<	<	<	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<				>
>=	<	<	<	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<				>
==	<	<	<	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<	<	<	<	>
~=	<	<	<	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<	<	<	<	>
..	<	<	<	<	<	>	>	>	>	>	>	>		<	>	<			<				>
#														<	>	<			<				
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	<	<	<	
)	>	>	>	>	>	>	>	>	>	>	>	>			>								>
ID	>	>	>	>	>	>	>	>	>	>	>	>			>								>
I	>	>	>	>	>	>	>	>	>	>	>				>								>
D	>	>	>	>	>	>	>	>	>	>	>				>								>
S						>	>	>	>	>	>	>	>		>								>
F										>	>				>								>
T										>	>				>								>
N										>	>				>								>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<	<	<	<	<	<	<	<	

Obr.2.1

Poznámky k precedenční tabulce:

je operátor délky textového řetězce.

ID je jakýkoliv identifikátor proměnné.

I je celočíselný literál typu number.

D je desetinný literál typu number.

Oba typy číselných literálů by bylo možno sjednotit do jedné položky v tabulce, ale uvádím je zvlášť pro případné budoucí znovupoužití projektu.

S je řetězcový literál typu string.

F je logický literál typu boolean s hodnotou False.

T je logický literál typu boolean s hodnotou True. True a False jsou uvedeny zvlášť, protože True je považován za jiný typ tokenu než False.

N je literál typu nil.

$ je ukončovací znak.

Zpracování výrazů bylo implementováno tak, aby za ukončovací byl považován jakýkoliv *token*, který nemůže být součástí výrazu. Pokud výrazový *parser* takovýto *token* dostane, výraz skončí a pokud byl syntakticky správný, řízení překladu je předáno bloku, který zpracování výrazu zavolal.

Infix to postfix je schopen zpracovat i volání uživatelské nebo vestavěné funkce, pokud je celý výraz ve formátu f(parametry...). Pokud je taková funkce detekována, je zpracování předáno funkci *parse_calling*, která zpracuje parametry funkce a zajistí vytvoření instrukcí potřebných pro provedení funkce.

Během zpracování výrazu probíhají kontroly syntaktické správnosti výrazu i sémantické správnosti (deklarace proměnných a funkcí).

Výhodou tohoto této intuitivně navržené implementace je, že umí bez jakýchkoliv dalších úprav zpracovat i unární operátory jako například rozšíření LENGHT.

I když to není nijak formálně implementováno, výrazová část našeho projektu umí zpracovat následující gramatická pravidla:

E->i //i je ID nebo literál kteréhokoliv typu

E->(E)

E->E bop E //bop = binární operátor

E->uop E //uop = unární operátor

E->f(parametry)

//f(parametry) se musí rovnat celému zpracovávanému výrazu. Některé funkce povolují, aby jejich parametry byly také výrazy. V tomto případě je pro každý parametr opět voláno celé *parse_expression*.

LL-gramatika použitá v našem projektu:

<FUNC>	Function ID (<Params>)<DECLS><STS>END<S><FUNC>	<EXPR>	READ(<STR/NUM>)
<FUNC>	Epsilon	<EXPR>	SUBSTR(<STR>,<NUM>,<NUM>)
<S>	;	<EXPR>	FIND(<STR>,<STR>)
<S>	Epsilon	<EXPR>	SORT(<STR>)
<PARAMS>	Epsilon	<STR/NUM>	<STR>
<PARAMS>	ID<Params_N>	<STR/NUM>	<NUM>
<PARAMS_N	Epsilon	<NUM>	ID
<PARAMS_N	,ID<PARAMS_N>	<NUM>	NUMBER
<DECLS>	Epsilon	<STR>	ID
<DECLS>	Local ID<V_VAL>;<DECLS>	<STR>	STRING
<V_VAL>	Epsilon	<OP>	+
<V_VAL>	= <CONST>	<OP>	-
<CONST>	STRING	<OP>	*
<CONST>	NUMBER	<OP>	/
<CONST>	NIL	<OP>	^
<CONST>	BOOL	<OP>	<
<STS>	Epsilon	<OP>	>
<STS>	ID = <EXPR>;<STS>	<OP>	=
<STS>	WRITE(<EXPR><EXPR_N>);<STS>	<OP>	#
<STS>	IF<EXPR>THEN<STS>ELSE<STS>END;<STS>	<OP>	<=
<STS>	WHILE<EXPR>DO<STS>END;<STS>	<OP>	>=
<STS>	RETURN<EXPR>;<STS>	<OP>	==
<EXPR_N>	Epsilon	<OP>	~=
<EXPR_N>	,<EXPR><EXPR_N>	<OP>	..
<EXPR>	(<EXPR>)	<TYPE>	ID
<EXPR>	<TYPE>	<TYPE>	STRING
<EXPR>	#<EXPR>	<TYPE>	NIL
<EXPR>	<TYPE><OP><EXPR>	<TYPE>	NUMBER
<EXPR>	ID(<EXPR><EXPR_N>)	<TYPE>	BOOL

2.3. Interpretace příkazů

Princip vykonávání vygenerovaných instrukcí spočívá v procházení seznamem instrukcí funkce *main* a podle potřeby procházení instrukčních seznamů volaných funkcí. Na uložení kontextu volání jednotlivých funkcí se používají seznamy *ActFunc* a *I_return,* přičemž *ActFunc* je seznam názvů funkcí a *I_return slouží* na uchovávaní adresy instrukcí, kterými se pokračuje po návratu z volané funkce, v obou seznamech jsou informace uložené v pořadí v jakém byly volané funkce.

Za zmínku stojí tyto instrukce:

- **NOP** – generuje se v případě potřeby kompletování nekompletních skokových instrukcí, zjednodušuje samotnou implementaci a plní místo v prázdných sekcích konstrukcí jazyka IFJ2011
- **CALL_START** – založí novou tabulku symbolů pro proměnné podle vzoru (generuje se v případě volání uživatelem definovaných funkcí)
- **CALL** – vloží novou tabulku symbolů do popisu funkcí a nastaví seznam zpracovávaných instrukcí příslušící k této funkcí
- **COPY, COPY1** – instrukce na přenos dat z tabulky do tabulky
- **CALL_STOP** – vymaže dočasnou tabulku symbolů a zrekonstruuje stav před voláním

2.4. Knuth-Moris-Pratt

Knuth–Morris–Pratt byl zformulován v roce 1977. Publikovali ho Donald Knuth, Vaughan Pratt a James H. Morris. Algoritmus používá pomocné celočíselné pole o délce rovnající se počtu symbolů hledaného slova. Tato tabulka se nazývá *partial match table* nebo *failure function* a podle hodnot v ní obsažených se algoritmus rozhodne od kterého symbolu hledaného slova bude pokračovat porovnávání, pokud dojde k neúspěšnému porovnání.

Algoritmus je možno implementovat jako konečný automat se třemi stavy: *Start*, *Read* a *Stop*. Naše implementace se skládá ze dvou funkcí. Hlavní funkce (v projektu nazvaná *kmp_find*, odpovídající funkci, označované v předmětu IAL jako *KMPMatch*) a pomocná funkce pro sestavení pomocného pole (v projektu má funkce název *kmp_table*, v předmětu IAL má název *KMPGraf*). Pomocná funkce je volaná hlavní funkcí pouze jednou. Protože má pomocná funkce lineární složitost O(m) a hlavní funkce lineární složitost O(n), celková složitost algoritmu je O(n+m), kdy v nejhorším případě m = n, tedy provede se 2*n porovnání.

Implementovat KMP nebyl problém, protože je dostatečně popsán v opoře předmětu IAL a na Wikipedii i s implementací v pseudokódu.

2.5. Tabulka symbolů

Tabulka symbolů je implementována [1] beztypovou tabulkou s rozptýlenými položkami(dále hashovací). Kolize synonym jsou řešeny jednoduchým jednosměrným seznamem bez hlavičky.

V tabulce je použita *Brensteinova hashovací* funkce, protože je jednoduchá, má dobrý rozptyl a je rychlá. Používá bitové posuny a součty.

Velikost tabulky symbolů je variabilní, ale v praxi se používá 128 položek. Za předpokladu, že funkce má rovnoměrný rozptyl, je možné do tabulky o této velikosti efektivně uložit 128*10 = 1280 položek co považujeme za [2] dostačující.

1 Hashovací tabulka je navržená pro práci s jakýmkoliv datovými typy. Každá položka může mít rozdílný(libovolný) datový typ.

Implementace tabulky nabízí široký výběr operací navržených pro maximální rychlost vkládání a vyhledávání položek. Tabulku je možné vytvářet staticky nebo dynamicky. Staticky je vytvořená tabulka funkcí, dynamicky jsou vytvářené a uvolňované *stack-y* volaných funkcí.

Do tabulky funkcí jsou vkládané záznamy o funkcích a každá funkce mám vzorový *stack* obsahující záznamy identické s *tokeny* - typ dat a jejich hodnota.

Během řešení projektu se ukázalo, že nejvýhodnější je uvolňovat tabulky symbolů(funkcí nebo *stacku*) globálně (např. ve funkci *main*). Z tohoto důvodu byly do operací nad tabulkou doplněny iterátory.

Z důvodu šetření místem v paměti se nevytvářejí kopie vyhledávacích klíčů vkládaných do tabulky. Lexikální analyzátor je vrací jako identifikátory a ty už jsou dynamicky alokované. Proto se při *run-time* uvolňování *stack-ů* volaných funkci neuvolňují vyhledávací klíče(jsou uvolňované globálně), což značně šetří paměť(při rekurzi) a počet systémových volání.

2.6. Heap sort

Heapsort je řadící metoda, která ke *stringu* (poli znaků) přistupuje jako k ADT hromada. Hromada je binární strom, ve kterém hodnota v kořeni má vyšší hodnotu než hodnoty potomků. Časová složitost je v nejhorším případě O(n.log(n)), zatímco paměťová náročnost je konstantní, protože řazení probíhá *in situ* (na místě).

Řazení znaků ve *stringu* bylo nutné implementovat nerekurzivně, protože *string* může obsahovat mnoho znaků. Nakonec jsme zvolili implementaci, která provádí sift (prohození hodnot uzlů tak, aby ze stromu vznikla hromada někdy nazýváno heapify) zdola nahoru, protože bylo možno algoritmus rozložit do čtyř jednoduchých krátkých funkcí.

Funkce ***hs_swap*** prohodí ve *stringu* dva znaky.

Funkce ***hs_heapify*** s použitím funkce hs_swap prohodí znaky v hromadě (jeden kořen a dva potomci) tak, aby znak s nejvyšší ordinální hodnotou byl v kořenovém uzlu. Využívá při tom toho, že při implementaci hromady nad polem platí, že index levého potomka se rovná dvojnásobku indexu kořene a index pravého potomka je roven dvojnásobku indexu kořene plus jedna. Provedením této funkce dojde k siftu jednoho uzlu celého *stringu*.

Funkce ***hs_heapify_all*** provede hs_heapify nad první polovinou *stringu* (len/2) postupně od nejvyššího indexu k nejmenšímu a tím zajistí sift celé hromady zdola nahoru.

Funkce *heapsort* je funkce, která je jako jediná z těchto čtyř dostupná ostatním modulům projektu a v cyklu provádí *hs_heapify_all*, prohození prvního a posledního znaku *stringu* a snížení délky úseku *stringu*, který je ještě potřeba seřadit. Tím zajišťuje, že od konce *stringu* se postupně hromadí znaky s nejvyšší ordinální hodnotou. Cyklus končí, když úsek určený k seřazení má nulovou délku. Po skončení algoritmu je *string* seřazený vzestupně.

Při implementaci bylo nejsložitější pochopit podstatu *heap sortu* a abstrakce binárního stromu nad polem. Nejlepším způsobem pro pochopení se ukázala animace ukazující průběh řazení.

2 Každá běžící funkce má vlastní kopii hashovací tabulky – stack – z čeho vyplývá menší přeplnění tabulek. Viz. interpret.

3. Způsob práce v týmu

3.1. Vývojový cyklus

Při vývoji interpretu byl použit vodopádový model. Jeho nedostatkem jsou měnící se požadavky, což nám však nevadilo nebo zadání projektu bylo jasně stanovené a neměnilo se.

Interpret byl implementovaný kombinací extrémního a defenzivního programování, což mělo za následek hodně nestandardních situací např. selhávání invariant v části implementované defenzivním programováním. Časem jsme však chyby odladili.

3.2. Návrh

Použili jsme kombinaci přístupu zdola-nahoru a shora-dolů, což mělo za následek jednoduchou implementaci. Hned na začátku jsme stanovili všechny rozhraní, takže moduly bylo možné vyvíjet nezávisle a bez problémů.

3.3. Rozdělení úloh

- Martin Riša: Vedení týmu, Syntaktická a sémantická analýza řídících konstrukcí a vestavěných funkcí, Interpret
- Michal Riša: Hashovací tabulka, Syntaktická a sémantická analýza volaní funkcí a deklarace funkcí, Interpret, testování
- Josef Rudolf: Zpracování výrazů, řadící algoritmus HEAP SORT, vyhledávací algoritmus KMP FIND, testovaní
- Tomáš Válek: Lexikální analýza, dokumentace, testovaní

4. Metriky kódu

- počet zdrojových souborů: 13
- počet řádků: 6333
- velikost spustitelného souboru: 73341 (B) Linux 64bit

5. Závěr

Interpret splňuje zadání a je doplněný o rozšíření. Dodržuje formát vstupních a výstupních dat. Jelikož to byl první týmový projekt, tak při řešení jsme narazili na problémy týkající se zejména týmové spolupráce, jako jsou schůzky a komunikace mezi členy v týmu. Byl zřízen přístup na SVN, což umožnilo snadnější výměnu souborů mezi členy týmu.

Nejvíce času nám zabralo ladění a opravováních chyb. Interpret byl testován na datech ze zadání projektu a také na našich navržených datech.

Projekt nám dodal spoustu zkušeností a nových poznatků, které v budoucnu uplatníme.