Dokumentace k projektu pro předměty IFJ a IAL

Implementace interpretu imperativního jazyka IFJ13

Tým 099, varianta a/2/I

15. prosince 2013

Autoři:

vedoucí: Michael Vlček	`xvlcek21`	20%
Radim Sváček	`xsvace02`	20%
Roman Blanco	`xblanc01`	20%
Vojtěch Mička	`xmicka06`	20%
Richard Wolfert	`xwolfe00`	20%

Obsah

1 Úvod

Tato dokumentace pojednává o interpretu imperativního jazyka IFJ13, zadaného jako skupinový projekt pro předmety IFJ a IAL. Níže najdete popis implementace některých základních částí tohoto projektu, jako náčrt konečného automatu pro lexikální analýzu, LL gramatiku syntaktického analyzátoru, postupy některých předepsaných metod a algoritmů pro využívané prvky či samotného interpretu. Dále pak rozdělení a koordinace práce v týmu a seznam využitých zdrojů informací.

2 Lexikální analýza

Implementace lexikálního analyzátoru je řešena jako konečný automat, jež načítá znaky ze zdrojového souboru. Na základě těchto znaků vytváří lexikální analyzátor tokeny, navíc souběžně vrací číselnou hodnotu popisující daný token. Tokeny jsou reprezentovány abstraktním datovým typem, který krom samotného tokenu, jež je uložen jako řetězec, uchovává navíc i ukazatel na poslední znak a informaci o celkovém počtu alokované paměti. Pro omezení počtu alokací je paměť alokována po 8 bytech. V případě, že neexistuje z daného stavu konečného automatu přechod do libovolného dalšího stavu, vrací lexikální analyzátor informaci o lexikální chybě ve zdrojovém programu.

3 Syntaktická analýza

3.1 Parser

Syntaktická analýza tvoří základ každého překladače. Hlavní úlohou syntaktické analýzy je zjistit, zda je zdrojový kód zapsán syntakticky správně. V tomto projektu jsme na analýzu základních programovacích konstrukcí použili rekurzivní syntaktickou analýzu shora dolů. Tuto metodu jsme implementovali s použitím níže popsané LL gramatiky.

3.2 Precedenční syntaktická analýza

V této části syntaktického analyzátoru je nejprve předáno řízení vyhodnocování výrazu precedenční syntaktické analýze (*PSA*), načež je vyhodnocen samotný výraz. Pro kontrolu správnosti výrazu z hlediska syntaxe je oproti klasickému řešení pomocí zásobníku a derivačního stromu potřeba využit dvousměrně vázaný seznam, který umožňuje skloubit zásobník a derivační strom (potažmo abstraktní derivační strom) do jedné struktury. Součástí syntaktické analýzy z hlediska struktur je i tabulka symbolů a statická tabulka přechodů pro určení priority a asociativity operátorů (případně zde můžeme zahrnout i výstupní seznam instrukcí).

3.3 Dvousměrně vázaný seznam

Seznam funguje podobně jako zásobník (na začátku seznamu po inicializaci ukončovací znak `$`, vkládání prioritně/asociativních znaků `<`, ...). Veškeré prvky dvousměrně vázaného seznamu (včetně terminálů) jsou uloženy ve formě instrukcí, stejným způsobem jako instrukce ve výstupním seznamu. V případě načtení terminálu je do seznamu přidána nová „instrukce “, jejíž oba ukazatele na operandy jsou hodnoty `NULL` a ostatní atributy (typ instrukce, ukazatel na výsledek) jsou přiřazeny následovně podle typu terminálu:

1. Pokud se jedná o proměnnou, typ instrukce `INST_NOP` (prázdná instrukce, použita pouze pro reprezentaci v seznamu, nebude přidána do výstupního seznamu instrukcí) a ukazatel na výsledek ukazuje do tabulky symbolů proměnných na místo, kde je uložena příslušná proměnná. Pokud neexistuje, je vrácena chyba.

2. Pokud jde o konstantu, vytvoří se nová položka v tabulce symbolů proměnných, jejíž klíč je řetězcová reprezentace této konstanty. Dále se postupuje analogický jako u bodu 1.

3. Pokud jde o znaménko nebo levou závorku, typ instrukce je hodnota tohoto znaménka v naší enumeraci, ukazatel na výsledek je `NULL`.

Terminální symboly gramatiky jsou postupně nasouvány do seznamu podobně jako u zásobníku. Aplikace redukce podle pravidel funguje také podobně, ale při aplikaci pravidla je po kontrole jeho syntaxe rovnou možné přidat instrukci na konec výstupního instrukčního seznamu. Pro ilustraci našeho řešení uvádím příklad provádění redukce sčítání těsně před redukcí sčítání a po ní:

Stav obou seznamů před načtením posledního znaku '`;`', který značí návrat řízení analýze rekurzivního sestupu:

Dvousměrně vázaný seznam					Seznam instrukcí				
abstr.	instrukce	adr1	adr2	result	abstr.	instrukce	adr1	adr2	result
'$'	semicollon	NULL	NULL	NULL	C → i	INST_ASSIGN	GEN0	NULL	GEN0
'<'	LEFT_ARROW	NULL	NULL	NULL	C → i	INST_ASSIGN	GEN1	NULL	GEN1
C → i	INST_ASSIGN	GEN0	NULL	GEN0					
'+'	plus	NULL	NULL	NULL					
C → i	INST_ASSIGN	GEN1	NULL	GEN1					

Po načtení znaku se přidá podle statické tabulky přechodů znak '`>`':

Dvousměrně vázaný seznam					Seznam instrukcí				
abstr.	instrukce	adr1	adr2	result	abstr.	instrukce	adr1	adr2	result
'$'	semicollon	NULL	NULL	NULL	C → i	INST_ASSIGN	GEN0	NULL	GEN0
'<'	LEFT_ARROW	NULL	NULL	NULL	C → i	INST_ASSIGN	GEN1	NULL	GEN1
C → i	INST_ASSIGN	GEN0	NULL	GEN0					
'+'	plus	NULL	NULL	NULL					
C → i	INST_ASSIGN	GEN1	NULL	GEN1					
'>'	RIGHT_ARROW	NULL	NULL	NULL					

Provede se redukce $C \rightarrow C + C$:

Dvousměrně vázaný seznam					Seznam instrukcí				
abstr.	instrukce	adr1	adr2	result	abstr.	instrukce	adr1	adr2	result
'$'	semicollon	NULL	NULL	NULL	C → i	INST_ASSIGN	GEN0	NULL	GEN0
C→C+C	INST_PLUS	GEN0	GEN1	GEN2	C → i	INST_ASSIGN	GEN1	NULL	GEN1
					C→C+C	INST_PLUS	GEN0	GEN1	GEN2

4 Sémantická analýza

Sémantická analýza je v našem projektu řešena částečně v syntaktickém analyzátoru a částečně v interpretu. Při syntaktické analýze jsou detekovány statické sémantické chyby a interpret má na starost detekci běhových či sémantických chyb až při samotné interpretaci programu.

5 Algoritmy pro práci s řetězci

5.1 Implementace řadícího algoritmu

Ze zadání vyplývala povinnost implementovat vestavěnou funkci pro řazení řetězce pomocí algoritmu Heapsort. Jedná se o řadící algoritmus s linearitmetickou složitostí. Základem tohoto algoritmu je halda, která je reprezentována samotným řetězcem, který je vstupním parametrem funkce. Díky tomu nemá algoritmus žádné výraznější nároky na paměť.

Funkce nejprve ustaví haldu, jež je přímo ve vstupním řetězci. Následně se provádí odebírání vrcholu haldy a jeho záměna s posledním prvkem. Poté následuje rekonsturkce již zkrácené haldy, která opět ustaví haldu do korektní podoby. Tento cyklus se provádí, dokud jsou prvky k řazení.

Rekontrukce haldy probíhá v cyklu, kdy je třeba ověřit, zda je otcovský uzel větší než oba synovské uzly. V opačném případě dojde k jejich záměně, přičemž se musí pokračovat i s haldou, jejíž otcovský uzel je nyní aktuálně vyměněný prvek. Tento jev se opakuje, dokud není dosaženo pravidla haldy.

5.2 Implementace vyhledávacího algoritmu

Stejně jako v případě řadícího algoritmu, byl i algoritmus pro vyhledávání v poli určen zadáním. Pro implementaci bylo tedy nutné využít Knuth–Morris–Prattův algoritmus řazení pole. Ten je založený na myšlence, kdy je zbytečné porovnávat znaky, jež již porovnány byly. K tomu, aby bylo možno omezit opětovné porovnávání znaků je nutné implementovat pole hodnot pro každou pozici řetězce, které označují index řetězce, na který se funkce při neúspěšném porovnání vrátí a bude porovnávat další znaky s dalšími znaky zdrojového řetězce. Vstupem funkce je kromě hledaného řetěze i zdrojový řetězec. Funkce vrací index zdrojového řetězce, na kterém byla nalezena shoda.

Nejprve je nutné vytvořit pole hodnot, které je vytvořeno na základě hledaného řetězce. V něm se můžou vyskytovat opakující se sekvence znaků, které je možno přeskočit. Například při řetězci $ABABC$ není nutné opakovaně porovnávaly znaky AB, ale při neshodě na pátém znaku je možné se vrátit na třetí znak a pokračovat. Výsledné pole tak vypadá následovně – [-1,0,0,1,2].

Poté již přichází na řadu samotné vyhledávání. V ideálním případě se při shodě s prvním znakem hledaného řetězce začnou porovnávat další znaky až do dosažení konce hledaného řetězce, načež se vrátí index shody s prvním znakem. Pokud ovšem kdykoliv během porovnávání narazí na neshodu znaků, obnoví se index v hledaném řetězci tak, že se k hodnotě počátku shody přičte hodnota z pomocného pole na indexu neshody. Díky tomu se omezí počet porovnávání již porovnaných znaků a výsledný proces je rychlejší.

6 Způsob a řešení realizace tabulky symbolů

6.1 Způsob realizace tabulky symbolů

Ze zadání byl určen způsob realizace tabulky symbolů, v našem případě binárním stromem. Nejprve bylo potřeba z opory a přednášek k předmětu IAL zjistit informace o binárním stromu, tedy že se jedná o strom prvků, kde každý prvek, krom posledních, má dva ukazatele na levý a pravý podstrom. V levém podstromu se pak uchovávají hodnoty nižší než jejich kořenový prvek a v pravém podstromu hodnoty vyšší.

6.2 Binární strom

Prvním důležitým bodem po získání informací o binárním stromu bylo ujasnění si, k čemu tabulka symbolů slouží a jaké operace nad ní budou požadovány. Jelikož v každém prvku bude potřeba uchovávat více informací, zvolili jsme jako každou položku stromu strukturu. Dále jsme se rozhodli realizovat dvě odlišené tabulku symbolů. Jednu pro uchovávání proměnných a druhou pro funkce. To zejména kvůli přehlednosti při jejich používání.

6.3 Tabulka symbolů proměnných

Do tabulky symbolů proměnných je jako unikátní vyhledávací klíč uvedeno jméno proměnné a dva ukazatele na každý podstrom (v případě neexistujícího podstromu nastavené na hodnotu `NULL`). Dále bylo zjevné, že je potřeba uchovat datový typ proměnné a její data. Tyto dva prvky jsme se pro přehlednost rozhodli uložit do další struktury a uchovávaná data pak realizovat unionem, pro jeho výhodné uložení v paměti, „union zabírá v paměti jen tolik místa, kolik největší jeho položka“. Union obsahuje položky pro uchování datového typu `integer`, `double`, `string` a `boolean`. Pro datový typ `NULL` není třeba mít vyhrazenou položku, neboť neuchovává žádná data.

6.4 Tabulka symbolů funkcí

Tabulka symbolů pro funkce obsahovala nejprve pouze jméno funkce jako unikátní vyhledávací klíč, dvojici ukazatelů na podstromy a položku uchovávající počet parametrů funkce. S vývojem parseru však bylo zjištěno, že je potřeba přidat ukazatel na seznam instrukcí a také ukazatel na tabulku symbolů proměnných. Pro práci s oběma typy tabulek bylo potřeba udělat funkce pro inicializaci, vyhledávání, vkládání a zrušení celé tabulky. Funkce pro práci s tabulkou symbolů pro funkce a proměnné jsou v principu stejné, ale místy se liší ve funkčnosti. Krom rozdílných parametrů (kvůli rozdílným prvkům ve strukturách) například funkce pro vyhledávání v tabulce proměnných při shodě klíčů provede změnu zadaných prvků, kdežto vyhledávací funkce pro tabulku symbolů funkcí vrátí chybu (jelikož v jazyce IFJ13 nelze přetěžovat funkce). Nejzajímavější z funkcí jsou funkce pro vkládání a rušení tabulky. Funkce pro vkládání nejprve v cyklu projde podle příslušného vyhledávacího klíče celý binární strom, za použití funkce `strcmp(char *s1, char *s2)`. „Ta porovnává řetězce `s1` a `s2`. Pokud je `s1` < `s2` pak vrací hodnotu menší než 0, pokud jsou si rovny, pak vrací 0, pokud je `s1` > `s2` pak vrací hodnotu větší než 0.“Pokud nebyla nalezena žádná odpovídající položka, alokuje místo pro novou položku, pro řetězec reprezentující vyhledávací klíč a pokud se jedná o tabulku proměnných také datový typ `string`, nealokuje paměť i pro tento řetězec. Následně podle datového typu vloží příslušná data z předaných parametrů a vrátí v případě úspěchu makro pro úspěch, jinak makro pro chybu.

6.5 Zásobník ukazatelů na položky binárního stromu

Při realizaci funkce pro rušení binárního stromu jsme zjistili potřebu zásobníku ukazatelů na položky binárního stromu. Zásobník jsme pojali jako jednosměrně vázaný seznam struktur a několik funkcí pro práci s nimi. Jedna položka zásobníku vždy obsahuje užitečná data, tedy ukazatel na položku tabulky symbolů a ukazatel na předešlý prvek seznamu (zásobníku). Jako funkce pro práci s ním pak stačilo implementovat inicializaci, vkládání (`Push`) a odstranění (`Pop`) z vrcholu zásobníku. Funkce pro přístup k vrcholu zásobníku není potřeba, kvůli snadnému přístupu k datům ve struktuře. Funkce pro rušení binárního stromu pak za použití zásobníku prochází binární strom a postupně ruší jednotlivé prvky voláním funkce `free()`. Rovněž je třeba uvolnit paměť nealokovanou pro vyhledávací klíč a v případě, že se jedná o tabulku proměnných a uchovávaný datový typ byl `string`, pak i pro něj nealokovanou paměť. Při inicializaci tabulky funkcí se rovnou vkládají záznamy o vestavěných funkcích jazyka IFJ13.

6.6 Další funkce

Během implementace dalších částí interpretu bylo zjištěno, že je potřeba dalších funkcí. Mezi funkce pracující s tabulkou proměnných přibyla funkce, která projde a okopíruje existující tabulku do nové. Pro tuto bylo třeba napsat pomocnou funkci, která postupně procházela binární strom až k nejlevějšímu prvku, `letftMost()` a plnila pomocný zásobník ukazateli na každý prvek. Pak se skrze zásobník procházel strom odspodu nahoru a kopírovala jednotlivé položky. V každém prvku se také kontrolovala existence jeho ukazatele na prvý podstrom,

případně se nad ním zavolala pomocná funkce `letftMost()`. Mezi funkce pracující s tabulkou funkcí pak také přibyla funkce která opět za pomoci funkce `letftMost()` procházela strom a kontrolovala existenci ukazatele na seznam instrukcí. Pak už jen několik testovacích funkcí kde stačily jednoduché průchody binárním stromem a tisknutí položek.

7 Rozdělení práce v týmu a komunikace

7.1 Schůze a komunikace v týmu

Díky blízkosti ubytování členů našeho týmu bylo možné konat poradní schůze prakticky kdykoliv bylo potřeba, nebyla tedy potřeba obtěžovat se s organizací a zřizováním komunikačních nástrojů.

7.2 Správa verzí zdrojového kódu

Pro přehlednost vývoje interpretu jsme použili verzovací system Git z důvodu předchozí zkušenosti s jeho používáním jednoho člena týmu. Zdrojové kódy jsme měli uložené na soukromém repozitáři webové služby Bitbucket.

8 Literatura, reference, zdroje informací

[1] Skripta k předmětu IAL

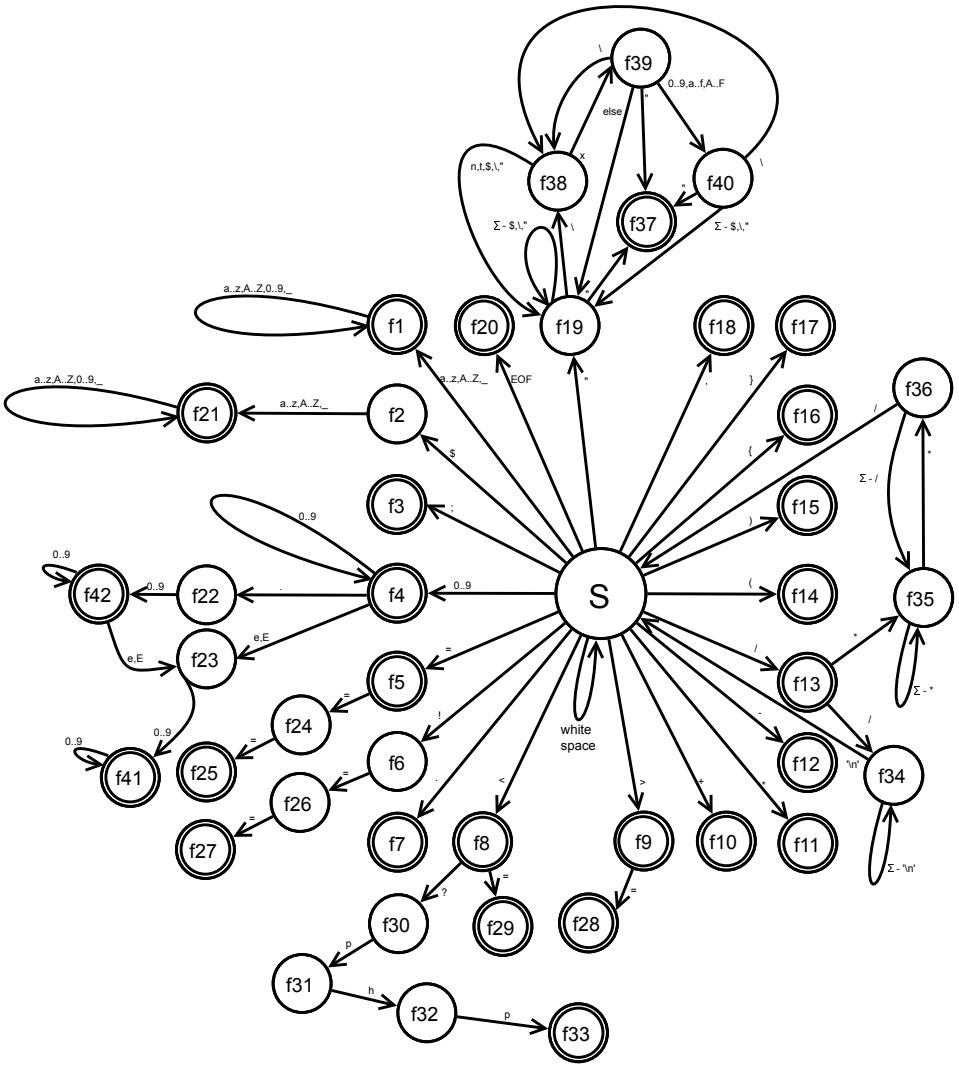
[2] Skripta k předmětu IFJ

[3] Popis typu `union`: http://www.sallyx.org/sally/c/c16.php

[4] Popis `strcmp`: http://www.sallyx.org/sally/c/c19.php

A Přílohy

A.1 Konečný automat



f1	`identificator`	f16	`left_braces`	f31	`php_entry1`
f2	`variable1`	f17	`right_braces`	f32	`php_entry2`
f3	`semicollon`	f18	`comma`	f33	`php_entry3`
f4	`int_num`	f19	`string_literal`	f34	`comment_line`
f5	`equal`	f20	`EOF`	f35	`comment`
f6	`not_equal`	f21	`variable2`	f36	`comment1`
f7	`concatenate`	f22	`double_num`	f37	`string_literal`
f8	`less`	f23	`double_num_exp`	f38	`escape`
f9	`greater`	f24	`equal2`	f39	`escape_x`
f10	`plus`	f25	`equal3`	f40	`escape_x1`
f11	`multiply`	f26	`not_equal2`	f41	`double_num_exp_end`
f12	`minus`	f27	`not_equal3`	f42	`double_num_end`
f13	`comment`	f28	`greater_equal`	s	Počáteční stav
f14	`left_parent`	f29	`less_equal`		
f15	`right_parent`	f30	`php_entry`		

A.2 Pravidla gramatiky

C	→	i
C	→	C OPERATION C
plus	→	+
minus	→	-
divide	→	/
concat	→	.
less	→	<
lessequal	→	<=
greater	→	>
greaterequal	→	>=
equal	→	===
nonequal	→	!==
C	→	(C)

Legenda:

C	–	nonterminál
i	–	terminál – identifikátor
plus, minus, ...	–	nontermální reprezentace znaménka
OPERATION	–	{plus, minus, ...}

A.3 LL gramatika

PROGRAM	→	<?php MAIN
MAIN	→	MAIN MAIN
MAIN	→	eof
MAIN	→	STATEMENTS
MAIN	→	function id ARGDEF { STATEMENTS }
ARGDEF	→	variable NEXTARG
ARGDEF	→	ϵ
NEXTARG	→	, variable
NEXTARG	→	ϵ
STATEMENTS	→	return expr ;
STATEMENTS	→	variable = ASSIGNMENT
STATEMENTS	→	if (EXPR) { STATEMENTS } else { STATEMENTS }
STATEMENTS	→	while (EXPR) { STATEMENTS }
STATEMENTS	→	ϵ
STATEMENTS	→	STATEMENTS STATEMENTS
ASSIGNMENT	→	expr ;
ASSIGNMENT	→	id CALL ;
CALL	→	(PARAMETERS)
PARAMETERS	→	PAR NEXTPAR
PARAMETERS	→	ϵ
NEXTPAR	→	, PAR NEXTPAR
NEXTPAR	→	ϵ
PAR	→	int_num
PAR	→	double_num
PAR	→	string_literal
PAR	→	null
PAR	→	false
PAR	→	true
PAR	→	variable

B Metriky kódu

Počet souborů: 15 souborů

Počet řádků zdrojového textu: 6217 řádků

Velikost statických dat: 948B

Velikost spustitelného souboru: 71556B (systém Linux, 64 bitová architektura, při překladu bez ladicích informací)