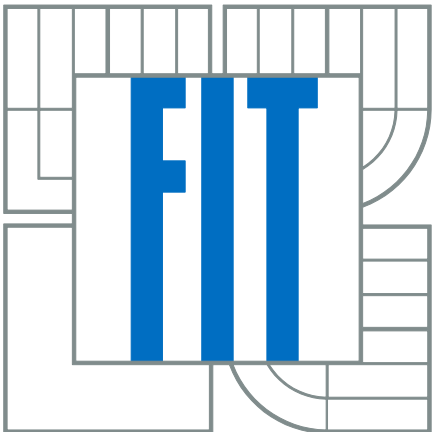
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



Dokumentace k projektu IFJ/IAL

Implementace interpretu imperativního jazyka IFJ14

Tým 105, varianta b/1/I

1.12.2014

Implementované rozšíření: ELSEIF

Maroš Janota (vedoucí)	(xjanot01)	21%
Matouš Jezerský	(xjezer01)	28%
Václav Bayer	(xbayer05)	20%
Tomáš Furch	(xfurch01)	20%
Šimon Híveš	(xhives00)	11%

Obsah

1 Úvod

Tato dokumentace popisuje vývoj interpretu jazyka IFJ14, jeho chování, princip, implementaci a také problémy, na které jsme při řešení narazili.

Vybrali jsme si zadání projektu b/1/I, které obsahuje:

1. vyhledávací Boyer-Moore algoritmus (pro vyhledávání podřetězce v řetězci),
2. řadící algoritmus Quick sort,
3. implementace tabulky symbolů formou binárního vyhledávacího stromu.

Interpret dále dělíme do tří částí:

I. lexikální analýza,
II. syntaktická a sémantická analýza,
III. interpret.

Každá z těchto částí bude následně stručně rozebrána.

2 Práce v rámci týmu

2.1 Komunikace

Skupinový projekt o takovémto rozsahu byl pro nás všechny v týmu něčím novým a zároveň velmi cenným získáním zkušeností. Jako jednu z variant komunikace jsme uznali za vhodnou pravidelné schůzky o plném počtu členů, avšak kvůli časovým indispozicím jednotlivých členů týmu v průběhu semestru se z pravidelných schůzek staly schůzky individuální s proměnným počtem členů, kteří potřebovali řešit aktuální problém. Jako primární komunikace nám sloužila domluva přes chat Fleep.io. Předávání informací a jejich synchronizaci nám zajistila webová služba GitHub podporující vývoj softwaru při používání nástroje Git.

2.2 Metodika vývoje

Metodika vývoje nebyla zpočátku stanovena, avšak v průběhu vzhledem k vývoji, nabrala směr nejblíž k metodice RUP (Rational Unified Process). Pravidelnou kontrolou stavů zadaných úkolů jednotlivých členů jsme docílili vyloučení věnování zbytečného úsilí již hotových strukturám či funkcím a dalším implementacím. Během vývoje byla snaha provázat komunikaci mezi jednotlivými moduly a zároveň průběžným testováním a zpětnou vazbou odhalit chyby nejlépe v zárodku než v koncových fázích. Testování tedy doprovázelo veškeré fáze vývoje.

2.3 Rozdělení práce

Maroš Janota:	precedenční analýza
Matouš Jezerský:	syntaktická a sémantická analýza, interpret, tabulka symbolů
Václav Bayer:	interpret, tabulka symbolů, dokumentace
Tomáš Furch:	lexikální analýza, hashovací tabulka
Šimon Híveš:	vestavěné funkce

Při tvorbě projektu byla použita vzorová kostra interpretu, která byla k dispozici pro studenty na webových stránkách předmětu IFJ.

3 Implementace

3.1 Lexikální analýza

Lexikální analyzátor (dále jen LA) je konečný automat (viz. Příloha A), kde vstupem je vstupním program, a výstupem jsou tokeny. Automat prochází mezi stavy, pokud neskončí v koncovém stavu, ohlásí chybu. Dále má za úkol odstranění bílých znaků a komentářů. Také se stará o převod řetězcových literálů - nahrazuje escape sekvence, vkládá uvozovky. Tokeny jsou předávány formou struktury. V LA se vyskytuje funkce umožňující počítání řádků, díky které lze rozpoznat, na jakém řádku se při čtení vyskytla chyba.

Činnost lexikálního analyzátoru řídí syntaktický analyzátor, který si žádá o jednotlivé tokeny funkcí *getToken.*

Klíčová slova jsou uloženy v hashovací tabulce *tHTable* pro rychlý přístup.

3.2 Syntaktická analýza

Syntaktická analýza (dále jen SA) je srdcem celého interpretu. Je úzce provázána se sémantickou analýzou. Při implementaci SA jsme postupovali metodou rekurzivního sestupu a využili jsme LL gramatiky (viz. Příloha B).

Vstupem SA jsou tokeny zasílany průběžně na vyžádání z lexikální analýzy. Výstupem je informace, zda je vstupní program syntakticky a sémanticky správně. SA generuje fragmenty tříadresného kódu pro funkce *if*, *while*, *readln* a *write*.

Veškeré **generování instrukcí** v parseru probíhá na instrukční pásku *tListOfInstr* (dvousměrný lineární seznam) pomocí funkce *generateInstruction*.

Při rozpoznání identifikátoru při práci s tokeny je volána **sémantická analýza**. Datová struktura pro sémantickou analýzu byla zvolena dle zadání, a to – binární vyhledávací strom, zvlášť jeden pro globální proměnné a deklarace funkcí. Pro lokální proměnné a jednotlivé funkce pak lineární jednosměrný seznam stromů *STList localTableList*. Sémantická analýza generuje tříadresný kód pro přiřazení, volání a deklarace funkcí.

Precedenční analýza (dále jen PA) je volána SA kdykoli, když narazí v pravidlech na výraz, aby jej vyhodnotila. Vstupem PA jsou tokeny zasílané ze SA ke zpracování. Tyto tokeny si nadále ukládá na hlavní zásobník a pomocí pomocného zásobníku s pravidly následně vykonává operace s tokeny dle precedenční tabulky (viz. Příloha C) a precedenčních pravidel. Mezivýsledky jsou ukládány jako unikátní proměnné, aby nedošlo ke kolizi s uživatelskými proměnnými. Následuje vygenerování tříadresného kódu pro výraz.

Přehled pravidel PA:

1. E -> E + E
2. E -> E – E
3. E -> E * E
4. E -> E / E
5. E -> E < E
6. E -> E > E
7. E -> E <= E
8. E -> E >= E
9. E -> E = E
10. E -> E <> E
11. E -> (E)
12. E -> ID

3.3 Interpret

Interpret je poslední částí aplikace a má na starost vykonávání instrukcí, které jsou zaznamenány na instrukční pásce. Pracuje s tabulkou hodnot *FStack frameStack*, která je v našem projektu zastoupena zásobníkem rámců. Rámec je lineární jednosměrný seznam a jeho prvkem je záznam obsahující klíč

proměnné (jedná-li se o literál je zakončem znakem $), data a informaci o tom je-li proměnná incializována. Pro globální hodnoty je vymezen rámec zvlášť.

Při volání hlavní funkce interpretu *inter* se nakopírují hodnoty z globální tabulky symbolů do globálního rámce *FrameList globalFrame*. Načte se první instrukce z instrukční pásky a provede se odpovídající akce. Čtení probíhá tak dlouho, dokud není vykonána poslední instrukce na instrukční pásce. Jestliže se při čtení narazí na volání funkce, dojde k nakopírování proměnných a instrukcí dané funkce z tabulky symbolů do nově vytvořeného rámce na zásobníku rámců pomocí funkce *STCopyToFrame* a *listInstrCopy*. Pokračuje se čtením instrukcí dané funkce. Jestliže dojde k rekurzi, celý tento krok se opakuje. Pokud k rekurzi nedojde, po dokončení funkce zůstane v rámci návratová hodnota, která se vloží o rámec níž a se kterou se následně pracuje dál a vrchol zásobníku se zahodí. Jestliže nezůstal na zásobníku žádný rámec a instrukční páska se dostala nakonec, nastal konec programu, který se dobral výsledku.

4 Řešení vybraných algoritmů z pohledu předmětu IAL

V této kapitole se věnujeme algoritmům, které jsou dány variantou projektu a které jsou probírány v rámci kurzu Algoritmy.

4.1 Boyer-Moore algoritmus

Tento algoritmus slouží k vyhledávání podřetězce v řetězci a v tomto projektu je uplatněn ve vestavěné funkci *find*. Při implementaci byla použita heuristika Bad Character.

Boyer-Moore algoritmus je považován za neúčinnější vyhledávací algoritmus v řetězcích. Algoritmus porovnává dva vstupní řetězce, v projektu proměnné string a pattern (dále jen vzor), přičemž vrací na výstupu celé číslo reprezentující první prvek indexu pole stringu, na kterém byl nalezen podřetězec vzor. Jako první krok je příprava tabulky skoků, pomocí které se potom ve vyhledávacím algoritmu posouvá vzor. Tabulka obsahuje 256 prvků (podle počtu znaků v ascii), přičemž index tabulky reprezentuje ascii hodnotu charu a samotná hodnota určuje číslo, o které je možné při vyhledávání vzor posunout. Veškeré prvky tabulky obsahují hodnotu délky vzoru, kromě indexu reprezentující znaky, které se nacházejí ve vzoru. Tabulka na těchto indexech obsahuje hodnotu délky vzoru, od které je odčítána poloha daného znaku ve vzoru zmenšeného o jedničku. Ve skutečnosti se vzor nikam neposouvá (mění se jen indexy, na kterých se pole porovnávají), slouží nám to pro lepší představu. Algoritmus prohledává znaky vzoru zprava doleva. V případě neshody při porovnávání znaků se použije dopředu připravená tabulka skoků pro určení posunu doprava. V případě shody algoritmus porovnává po směru doleva všechny znaky vzoru do té doby, dokud se nedostane nakonec vzoru nebo se neshodne při porovnávání. V případě shody při porovnávání se provede skok a porovnávají se znaky dál, přičemž se opakuje výš popsaná metoda. V případě, že se dostane na konec řetězce a vzor v něm nebyl vyhledán, algoritmus se ukončí a vrátí nulu.

Vyznačuje se tím, že porovnávaný vzor se vždy posouvá jen doprava a to maximálně až do délky vzoru. V nehorším případě má tento algoritmus složitost O(m+n). Algoritmus je možné ještě více

optimalizovat pomocí kombinované heuristiky Bad Character a Good Suffix kde se při porovnani vybírá větší skok.

4.2 Quick sort

Quick sort je nejrychlejší řadící algoritmus a v našem projektu je uplatněn pro vestavěnou funkci *sort*. Z nabídky dvou implementací (rekurzivní a iterativní), jsme si vybrali právě rekurzivní, protože je přehlednější a není v tomto případě potřeba používat zásobník.

Algoritmus využívá mechanismu rozdělení „partition“, jehož autorem je C.A.R. Hoare. Mechanismus určuje hranici mezi dvěma poli, kde na levé straně jsou hodnoty menší než tato daná hranice a na pravé straně zas naopak větší. Tato hranice se běžně označuje jako *pivot* (v našem projektu i stejnojmenná proměnná). Vhodným určením této hranice můžete ovlivnit i velmi pozitivně rychlost řazení. Pivot lze určit matematickým výpočtem, pevně nebo náhodně. My jsme si zvolili matematický výpočet a to aritmetický průměr součtu první a poslední hodnoty. Tím jsme zajistili největší efektivitu a zvýšení rychlosti tohoto řadícího algoritmu, optimalizovali jsme jej. Po zvolení pivotu následuje hledání znaků v jednotlivých částech. Jakmile se narazí na větší číslo než je pivot v levé části a menší číslo než je pivot v pravé části, dochází k záměně. Tímto způsobem pokračujeme dál, dokud se oba indexy vyhledávající v částech nezkříží. [1]

4.3 Tabulka symbolů

Jak už bylo dříve naznačeno, tabulka symbolu dle specifik zadání pro naši variantu je implementována formou binárního vyhledávacího stromu. Vyhledávání je implementováno opět rekurzivně kvůli přehlednosti a vyloučení nutnosti využití zásobníku.

Popis využití a implementace byl už z velké části použit při objasnění fungování sémantického analyzátoru. Shrňme akorát ve stručnosti. Pro globální proměnné a deklarace funkcí je samostatně implementovaný binární strom a pro jednotlivé funkce a jejich proměnné je zaveden lineární jednosměrný seznam jednotlivých stromů.

5 Závěr

Velké plus při naši tvorbě byl aspekt na tvorbu univerzálního kódu, tzn. kód byl tvořen tak aby byl připraven na pozdější možné změny a úpravy a přitom bylo zamezeno co největšímu vzniku chyb.

Využili jsme druhého pokusného odevzdání, což bylo v plánu hned od začátku. Na přednáškách se probírala potřebná látka k dokončení projektu s mírným zpožděním, takže v průběhu tvorby projektu byla nutnost samostudia. Pokusné odevzdání proběhlo bez interpretu, avšak s výsledky hodnocení jsme byli spokojeni a interpret byl následně hned zhotoven a nám již zbývalo akorát testovat.

Testování probíhalo pomocí vlastní testovací sady zhotovené k tomuto účelu. Použita byla i veřejně dostupná sada.

S projektem o podobném rozsahu jsme se nikdy v minulosti nesetkali. Byla to pro nás nová zkušenost a zároveň výzva. Nezdary při implementaci, nešvary při domluvě, nutnost komunikace s ostatními a

občasná časová tíseň nám ukázaly, že ne vždy je vše optimální a jednoduché, a to co jsme si odnesli, jsou cenné a těžce vydobyté zkušenosti. Naštěstí tyto potíže se vyskytly jen výjimečně a po většinu času společná práce na projektu probíhala bez problémů.

5.1 Metriky kódu

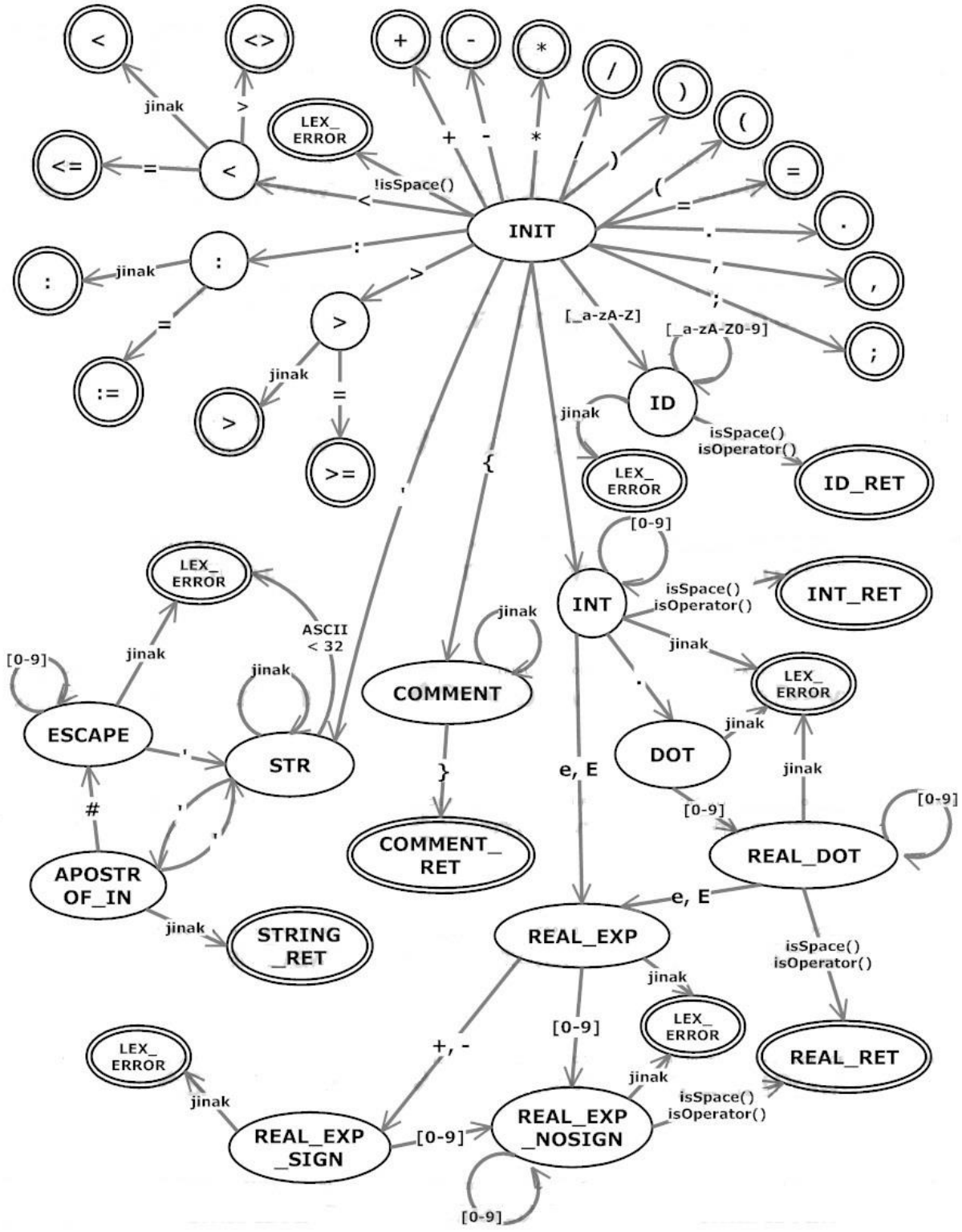
- Počet souborů: 18
- Celkový počet řádků zdrojového kódu: 4714
- Počet Git commitů: 274

5.2 Použitá literatura

[1] HONZÍK, Jan M. Algoritmy: Studijní opora. 2014. verze 14-Q.

6 Přílohy

A. Konečný automat



B. Pravidla LL gramatiky

1	<prog>	->	<var> <sekvence_funkci> BEGIN <sekvence_prikazu> . EOF
2	<slozeny_prikaz>	->	BEGIN <sekvence_prikazu>
3	<sekvence_prikazu>	->	<prikaz> <strednik_nebo_end>
4	<sekvence_prikazu>	->	END
5	<strednik_nebo_end>	->	END
6	<strednik_nebo_end>	->	; <prikaz> <strednik_nebo_end>
7	<sekvence_funkci>	->	<funkce> <sekvence_funkci>
8	<sekvence_funkci>	->	eps
9	<funkce>	->	FUNCTION <id> (<deklarace_parametry>) : <typ_funkce> ; <telo_funkce>
10	<telo_funkce>	->	<var> <slozeny_prikaz>
11	<telo_funkce>	->	FORWARD ;
12	<id>	->	ID
13	<prikaz>	->	<id> := <vyraz_nebo_volani>
14	<vyraz_nebo_volani>	->	LITERAL <op>
15	<vyraz_nebo_volani>	->	<id> <operator_nebo_volani>
16	<operator_nebo_volani>	->	(<parametry>)
17	<operator_nebo_volani>	->	<op>
18	<prikaz>	->	IF <vyraz> THEN <slozeny_prikaz> <else>
19	<else>	->	ELSE <slozeny_prikaz>
20	<else>	->	eps
21	<prikaz>	->	BEGIN <sekvence_prikazu>
22	<prikaz>	->	WHILE <vyraz> DO <slozeny_prikaz>
23	<prikaz>	->	READLN (<id>)
24	<prikaz>	->	WRITE (<parametry>)
25	<prikaz>	->	READLN (<id>)
26	<vyraz>	->	<id>
27	<vyraz>	->	<vyraz> <operator_nebo_eps>
28	<vyraz>	->	(<vyraz>)
29	<operator_nebo_eps>	->	eps
30	<operator_nebo_eps>	->	<op> <vyraz>
31	<op>	->	*
32	<op>	->	/
33	<op>	->	+
34	<op>	->	-
35	<op>	->	<
36	<op>	->	>
37	<op>	->	<=
38	<op>	->	>=
39	<op>	->	=
40	<var>	->	VAR <deklarace> <sekvence_deklaraci>
41	<var>	->	eps

42	<sekvence_deklaraci>	->	<deklarace> <sekvence_deklaraci>
43	<sekvence_deklaraci>	->	eps
44	<deklarace>	->	<id> : <typ> ;
45	<typ>	->	INTEGER
46	<typ>	->	REAL
47	<typ>	->	STRING
48	<typ>	->	BOOLEAN
49	<parametry>	->	<id_nebo_literal> <param_nebo_eps>
50	<parametry>	->	eps
51	<param_nebo_eps>	->	, <id_nebo_literal> <param_nebo_eps>
52	<param_nebo_eps>	->	eps
53	<deklarace_parametry>	->	<id> : <typ_funkce> <dekl_param_nebo_eps>
54	<deklarace_parametry>	->	eps
55	<dekl_param_nebo_eps>	->	; <id> : <typ_funkce> <dekl_param_nebo_eps>
56	<dekl_param_nebo_eps>	->	eps
57	<id_nebo_literal>	->	LITERAL
58	<id_nebo_literal>	->	<id>
59	<typ_funkce>	->	INTEGER
60	<typ_funkce>	->	REAL
61	<typ_funkce>	->	STRING
62	<typ_funkce>	->	BOOLEAN
63	<vyraz>	->	<id> <fcall>
64	<fcall>	->	(<parametry>)
65	<fcall>	->	eps

C. Precedenční tabulka

	+	-	*	/	<	>	<>	<=	>=	=	(	)	id	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*****	>	>	>	<	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
id	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	