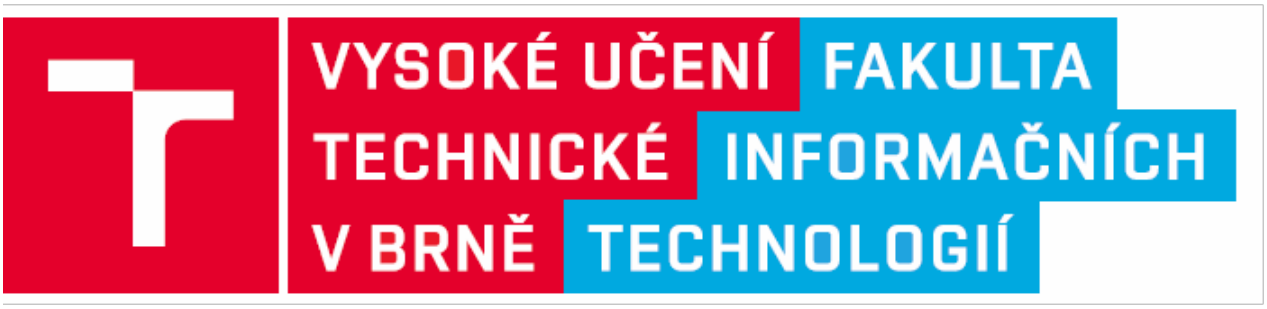
Fakulta Informačních Technologií
Vysoké Učení Technické v Brně



Dokumentace k projektu IFJ a IAL
Interpret jazyka IFJ16

Tým 004, Varianta a/2/II

11. prosince 2016

Vedoucí: **Javorka Martin (xjavor18) 20%**
Marušin Marek (xmarus08) 20%
Utkin Kirill (xutkin00) 20%
Mrva Marián (xmrvam01) 20%
Shapochkin Victor (xshapo00) 20%

Obsah

1 Úvod

Tato dokumentace popisuje návrh a implementaci interpretu jayzka IFJ16, který je zjednodušenou podmnožinou jazyka Java SE 8. Zvolili jsme si variantu zadání a/2/II, ve keteré máme naimplementovat funkci find pomocí algoritmu Knuth-Morris-Prattův, funkci sort pomocí algoritmu Heap Sort a tabulku symbolů pomocí tabulky s rozptýlenými položkami.

2 Řešení projektu

Projekt jsme se rozhodli rozdělit na několik části:

- Lexikální analyzátor
- Syntaktický analýzator
- Semantický analýzator
- Věstavené funkce
- Interpret
- Testování

Syntaktický analyzátor je "srdcem" celého interpretu, řídí ostatní části a zároven provádí i sémantickou kontrolu. Lexíkální analýzator mu zasílá tokeny, které potom analizuje. Když je zpracován celý program a je správný, tak se potom spustí interpret, který na základe instrukční pásky provede výslednou interpretaci programu.

2.1 Lexíkální analýzator

Lexikální analýzator (přip. scanner) je implementován konečným automatem (obrázek č. 1), který na vstupu dostane zdrojový soubor s posloupností znaků, odstraní všechný nedůležité části jako komentáře a bílé znaky a rozdělí je na tzv. **tokeny**.

Token (tToken) je struktura, která obsahuje informaci o lexéme do které se postupně přeřazují jí znaky za běhu scannera. Pokud se dostane lexéma do konečného stavu automatu bez zajištění chyby (vyhovuje IFJ16), tuto informaci o lexéme dostane parser, který si token požádal pomocí funkce getNextToken. Mezi informací o lexéme patří "stav", "data" potřebné pro parser a několik pomocných atributů pro správný běh v scanneru. Pokud scanner vrátí token s error-ovým stavem, parser ho vyhodnotí, vypíše požadovaný error-ový kód a zastaví program.

Velmi důležitou částí lexikálního analyzátoru je rozpoznávání a ošetřovaní řetězců, oktalových čísel, speciálních znaků a čísel, které mají největší počet stavů.

Do scanneru jsme měli implementovat buffer, který uchovává u sebe historii tokenů, používá ho parser když potřebuje dostat určitý token, který už v minulosti dostal.

2.2 Syntaktický a sémantický analýzator

Syntaktický analyzátor (případně parser) je jeden z najrozšířenějších a nejdůležitějších modulů interpretu. Má za úlohu hned několik hlavních operací nad uživatelským vstupem. Parser kontroluje konstrukce vstupního souboru, jinými slovy má kontrolovat syntaktické správnosti a sémantický stranu programu. Vyvolá se hned po spuštění interpretu a inicializaci globálních proměn. Syntaktický a sémantický analyzátor celý čas komunikuje s modulem lexikálního analyzátoru od kterého si vyžaduje stále nový token na jeho zpracování. Jazyk IFJ16 je podmnožina jazyka JAVA SE 8, proto je naše syntaktická a sémantická analýza inspirovaná dvojfázovým přechodem - to znamená, že modul parser se volá dvakrát, přitom si pamatuje ve které fázi přechod se nachází. V prvním přechodu se kontroluje syntaktická správnost vstupního souboru a zapisuje proměny, třídy a funkce do tabulky symbolů. Druhá fáze přechodu zabezpečí sémantickou kontrolu, kde víme zjistit pomocí tabulky symbolů redefinici identifikátorů, případně jich nedeklaráci, datové typy a podobně. Obě fáze dvojfázového přechodu ŠÚ jsou založené na rekurzívním sestupu, který je popsán níže. Tabulka symbolů je důležitá část interpretu, do které parser mnohokrát zapisuje a předčítá. Z toho parser musí rozpoznat globální tabulku symbolů, a tabulky symbolů nižších úrovní, které jsou uložené právě v uvedené dřív globálně tabulce. Na základě těchto operací dokáže modul parser vyhodnotit i typovou kontrolu. Tím zjistíme jestli nám sedí jednotlivé typy proměn, které musí byt kompatibilní, počty parametrů ve funkcích a podobně.

Kromě jiného parser v druhém přechodu generuje instrukce vnitřního kódu, které v nejbližší fáze modul interpretu zpracovává a interpretuje. Z hlediska modulu parser se jedna zejména o generovaní instrukcí skoku, volání funkcí, instrukcí přiřazení a podobně. Všechny vygenerované instrukce parser ukládá na instrukční pásku, zatimco jednu instrukci reprezentuje tříadresný kód.

Rekurzívní sestup uvedený dřív je založený na LL-gramatice 1 (viz B.), podle které jsme implementovali celý modul parser. Modul je implementován jako sada funkcí, které představují jednotlivé neterminály. Naopak jednotlivé tokeny jsou vnímané jako terminální symboly LL gramatiky. V gramatice se nachází taky speciální symbol <expression>, který vyjadřuje předaní řízení precedenční analýze, která zpracovává výrazy pro podstatné zjednodušení syntaktické a sémantické analýzy. V případě že se nedá podle získaného tokenu určit pravidlo gramatiky, parser detekuje chybu a na výstup posílá příslušný návratový kód na výstup programu, pokud se v žádné z fází dvojfázového přechodu parseru nenajde syntaktická nebo sémantická chyba.

Úloha syntaktické a sémantické analýzy tímto končí a řazení se předává modulu interpret.

2.3 Precedenční analýza

Na zpracovaní výrazů je použitá syntaktická analýza “zdola nahoru”. Pokud parser narazí na vyraz, předává řízení precedenční analýze.

Precedenční analýza je implementovaná pomocí precedenční tabulky a zásobníku. Na zásobník se ukládají terminaly a neterminaly. Terminaly mohou byt identifikatory, konstanty a operátory. Neterminal je výraz, speciální neterminal $ (dollar) určující konec výrazu, a stav '<' z precedencnej tabulky.

Precedencní analýza na základě pravidla vytvoří novou položku v tabulce symbolů jako výsledek výrazu a přidá tříadresní kód na instrukční pasku. Precedenční analýza odhalí chybný výraz (syntaktickou chybu) nenajdením vhodného pravidla.

2.4 Interpret

Interpret byl poslední částí projektu, má za úkol zpracovávat výstup programu a vytvářet cílový kód v případě úspěšného ukončení sémantické a syntaktické analýzy. Interpret postupně vykonává instrukcí (tříadresní kód) z instrukční pásky. Je volán z mainu po skončení syntaktické analýzy ve dvou krocích. V prvním kroku vyhodnotí všechny instrukce pro statické proměny, které jsou na samostatné instrukční pásce. Ve druhém kroku interpret vyhodnotí instrukce, které se nachází ve funkcích. Nejdřív si přemapujeme identifikátory (proměny a konstanty) z tabulky symbolů na lokální rámec. Pro intrepretovaní jednotlivých se nejdřív ověří, jestli se identifikátory nachází v rámci. V případě že se identifikátor nenajde, hledá se v tabulce symbol pro třídu, ve které se funkce nachází.

Interpret prochází jednotlivé instrukci za sebou v cyklu. Výjimkou jsou instrukce skoku, které přímo odkazují na následující instrukci. Volaní funkce je vyřešené rekurzivním voláním interpretu, kdy se mu předá instrukční páska pro volanou funkci a tabulka symbolů pro volanou funkci.

3 Algoritmy a datové struktury

V této kapitole budou popsáný algoritmy, které byli implementovaný v našem projektu.

3.1 Heap sort

Tento algoritmus patří do skupiny třídicích algoritmů a je jedním z nejefektivnějších algoritmu. Základní myšlenkou je využití datové struktury označované jako „halada“. Halada je binární strom o N úrovních, který má všechny uzly na všech (n-1) úrovních a na nejvzdálenější n-té úrovní má všechny uzly zleva, lze snadno implementovat polem. Pokud otec má index i, tak jsou jeho potomci na indexech 2i (levý syn) a 2i+1 (pravý syn). Potom máme funkci zvanou Sift Down, která znovu staví hromadou porušenou v kořeni – proseje prvek v koření, který jako jediný porušuje pravidlo hromady. Algoritmus je velmi rychlý, ale není stabilní. Jeho náročná složitost je O (n log n).

3.2 Knuth-Morris-Pratt

Algoritmus se používá při vyhledávání podřetezce v jínem řetezci. Základní výhledávácí algoritmus se porovná podřetetezec s řetezcem postupně po znaku. Má lineární časovou složitost O ($m + n$) a je význačný tím, že zachovává minimální možný počet celkových porovnání, tzn. neporovnává žádnou odpovídající dvojici (vzoru a řetězce) dvakrát. Před vyhledáváním vytvoříme si t.z vyhledávací tabulku, která bude mít pro každou pozici ve vzorku napsané číslo, které nám bude určovat, jaký prvek vzoru musíme porovnat s aktuálním znakem v řetezci. V případě že prvky neshodují, nemusí procházet už porovnané prvky a číslo z tabulky mu dá informace o tom, s kterým znakem vzoru má tento aktuální prvek řetězce porovnat před dalším posunutím.

3.3 Tabulka symbolů

Tuto tabulku symbolů jsme měli implementovat pomocí tabulky s rozptýlenými položkami. Pro hash tabulku používáme funkci djb2 (autor Bernstein). Tato tabulka je datovou strukturou pro

potřeby rychlého vyhledávání. Základem je obyčejné pole. Položkami pole jsou ukazatele na lineární seznamy, do nichž je možné uložit položky, které mají unikátní vyhledávací klíč. Důležitá část tabulky je tzv. hashovací funkce. Na vstupu dostává vyhledávací klíč (textový řetězec) a výstup je hash, který použijeme jako index do pole.

3.4 Garbage collector

Pro lepší správu pamětí jsme se rozhodli využít garbage collector. Funguje tak, že vyhledává a uvolňuje úseky pámětí, které již program(přip. proces) nepoužívá. Uvolnění pamětí provádí funkce freeGC() na konci programu.

3.5 Rozšíření

Rozhodli jsme se v zadání implementovat jedno rozšíření.

3.5.1 Boolean

Scanner pro rozšíření boolean v konečném automatu obsahuje navíc 2 klíčových slova (true a false) a jeden stav (state).

Precedenční analýza má přidané nové pravidla pro operací nad typem boolean. Pro každé pravidlo je aj nový typ instrukci v tříadresním kódu. Zároven byla rozšířená precedenční tabulka o tři nové operátory (not, or a and). Pro tyto operátory a pravidla je rozšířená typová kontrola, kde na dvou stranách výrazu očekává hodnotu typu boolean. Jsou upravené stavy určující přednost pro porovnávací operátory == (equal) a != not equal. Tyto operátory se ted můžou vyskytnout ve výrazu spolu s ostatnými porovnávacími operátory (< , > , <=, >=). Parser rozšířil kontrolu tokenu o keyword boolean a rozšířil typovou kontrolu. Interpret přidal vykonávaní instrukcí not, and a or.

4 Práce v týmu

Během práce na projektu, jsme měli různé způsoby komunikaci mezi členy týmu. Jako hlavní komunikační prostředek byla zvolená aplikace **Slack**, ve které jsme vytvořili skupinu pro náš projekt, a kde řešili problémy vznikající během naše práci. Taky jsme používali aplikace **MeisterTask** pro lepší přehled úkolů, které máme v plánu a které už máme vypracované.

Pro uchovávání našého projektu jsme využívali repository **BitBucket**.

Během celého semestru jsme měli několik schůzek. Na první schůzce jsme si rozdělili jednotlivé častí projektu. Potom schůzky probíhali pouze několikrát pro posouzení jednotlivých častí zadaní.

4.1 Rozdělení práce

Javorka Martin – parser, interpret, dokumentace
Marušin Marek – syntaktická a sémantická analýza (parser), generovaní vnitřních tříadresných instrukcí, návrh LL gramatiky, pomocné struktury, předávaní zpětného tokenu do scanneru, dokumentace
Utkin Kirill – práce s řetězci ,věstavené funkce, garbage collector + testování, dokumentace

Mrva Marián – scanner, interpret, dokumentace
Shapochkin Victor – ial, tabulka symbolů, garbage collector + testování, dokumentace

5 Závěr

Práce nad projektem byla moc zajímavá a přinesla nám spoustu zkušeností do budoucna, především v jazyku C a vývoje aplikace ve velkém týmu. Projekt byl časově náročný, ale moc užitečný. Jsme velmi rádi, že byl vytvořen termín 2. pokusného odevzdání, diky kterému jsme stihli doladit funkčnost projektu před koncovým odevzdáním.

Výsledný program funguje dle zadaní a má jedné rozšíření **BOOLEAN**.

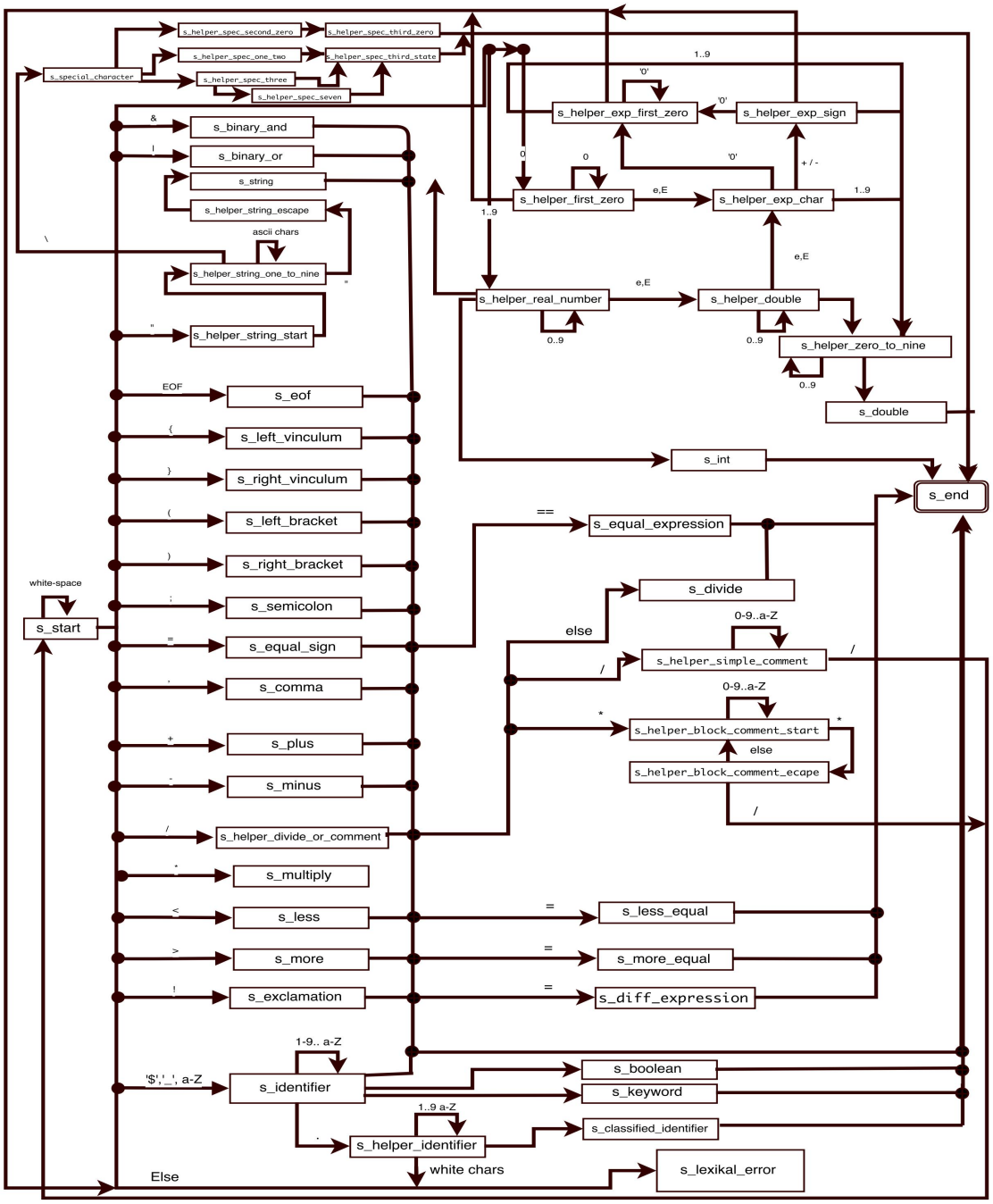
6 Metriky kódu

- Počet souborů: 34
- Řádků kódů: 11746

6.1 Literatura

- Přednášky, skripta a podklady z předmětu IFJ a IAL
- MEDUNA, Alexander. *Formal languages and computation: models and their applications*. New York: CRC Press, c2014. ISBN 978-1-4665-1345-7.

A Konečný automat pro lexikální analýzu



Obrázek 1: Graf konečného automatu

B LL – gramatika

1. <prog> → <class_ list><eof>
2. <class_ list> → <class><class_ list>
3. <class_ list> → <e>
4. <class> → class id class_ body
5. <class> → <e>
6. <class_ body> → <class_ body><class_ body>
7. <class_ body> → <e>
8. <class_ body> → static <type> id <class_ id_ list>
9. <class_ id_ list> → ;
10. <class_ id_ list> → =<term>;
11. <class_ id_ list> → <function_ definition>
12. <function_ definition> → (<func_ arguments>)<body_ list>
13. <func_ arguments> → <type> id <func_ arguments2>
14. <func_ arguments> → <e>
15. <func_ arguments2> → ,<type> id <func_ arguments2>
16. <func_ arguments2> → <e>
17. <body_ list> → <body><body_ list>
18. <body_ list> → <e>
19. <body> → <e>
20. <body> → ;
21. <body> → return<term_ list>;
22. <body> → if(<term>)<body_ list>else<body_ list>
23. <body> → while(<term>)<body_ list>
24. <body> → <type> id <id_ list>;
25. <body> → id<id_ statement>;
26. <id_ statement> → (<parlist>)
27. <id_ statement> → =id(<id_ statement>)

28. <id_ statement> → .id<id_ statement>
29. <term_ list> → <term>
30. <term_ list> → <e>
31. <parlist> → <term><parlist2>
32. <parlist> → <e>
33. <parlist2> → ,<term><parlist2>
34. <parlist2> → <e>
35. <id_ list> → <e>
36. <id_ list> → =id(parlist)
37. <term> → id
38. <term> → <string_ constant>
39. <term> → <int_ constant>
40. <term> → <float_ constant>
41. <term> → <expression>
42. <type> → null
43. <type> → integer
44. <type> → double
45. <type> → string
46. <type> → boolean
47. <type> → void

C Precedenční tabulka

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

Obrázek 2: Precedenční tabulka