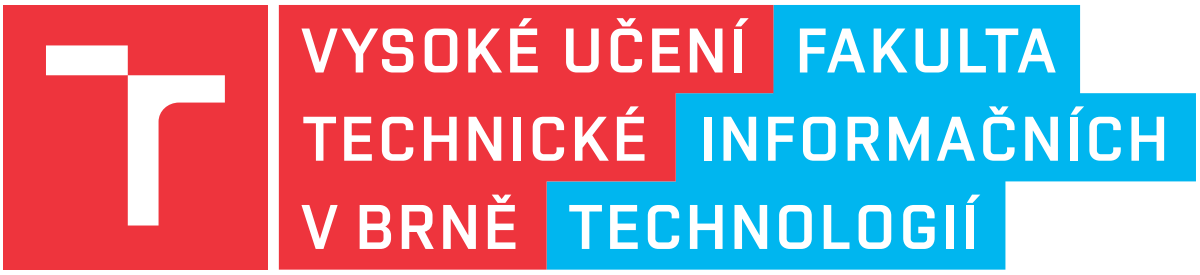


Dokumentace projektu do IFJ a IAL

Interpret jazyka IFJ16

Tým 033, varianta a/1/I

Rozšíření: SIMPLE

Seznam autorů:

Martin Ivančo (xivanc03)	20%
Filip Januš (xjanus08)	20%
Vladimír Jeřábek (xjerab21)	20%
Petr Jůda (xjudap00)	20%
Jakub Klemens (xkleme11)	20%

10. prosince 2016

Obsah

1 Úvod

Dokumentace popisuje implementaci interpretu jazyka IFJ16, zadaného v rámci projektu, do předmětů Formální jazyky a překladače a Algoritmy. Dokumentace je rozdělena na kapitoly, které se blíže zaměřují na námi zvolené řešení projektu. Součástí dokumentace jsou přílohy, obsahující strukturu konečného automatu, LL gramatiku a precedenční tabulku. Závěrem je popsáno rozdělení úkolů a celkové zhodnocení naší práce na projektu.

2 Řešení projektu

Tato kapitola popisuje jednotlivé části řešeného projektu.

2.1 Lexikální analyzátor

Lexikální analyzátor je implementován pomocí konečného automatu, který načítá zdrojový kód ze souboru a čte z něj jednotlivé znaky, které zpracovává na tokeny.

Vstupní soubor získá lexikální analyzátor z globální proměnné `file` definované v souboru `main.c`. Komunikace mezi syntaktickým analyzátorem a lexikální analýzou probíhá pomocí jedné globální struktury typu `t_token`, která obsahuje typ tokenu a v případě textového řetězce, klíčového slova nebo číslice i patřičnou hodnotu.

Syntaktický analyzátor volá funkci `get_next_token`. Ta pomocí funkce `clean_token` vyčistí globální token. Dále mu přiřadí odpovídající hodnotu podle typu načteného lexému ze zdrojového souboru. V případě, že lexikální analýza narazí na chybu, vrací hodnotu 1, jinak 0.

Schéma konečného automatu naleznete v příloze 6.1.

2.2 Syntaktická analýza

Syntaktický analyzátor je centrem celého překladu. Na vstupu dostává token z lexikálního analyzátoru, nad kterým vykonává syntaktické a sémantické kontroly a současně generuje tříadresný kód pro běh samotného interpretu.

Narazí-li syntaktický analyzátor na definici proměnné nebo definici funkce, uloží ji do tabulky symbolů na příslušnou úroveň. Jedná-li se o proměnnou (nebo funkci) na globální úrovni, uloží ji do tabulky symbolů odpovídající dané třídě, ale jedná-li se o proměnnou lokální, bude uložena do tabulky symbolů k dané funkci, kde se vyskytuje. Tím máme zajištěnou možnost, že se dvě proměnné se stejným názvem na odlišných úrovních budou překrývat. A také, že dvě proměnné se stejným názvem v různých funkcích nebudou kolidovat.

Syntaktickou analýzu dále dělíme na syntaktickou analýzu shora dolů a syntaktickou analýzu zdola nahoru.

2.2.1 Syntaktická analýza shora dolů

Syntaktická analýza shora dolů analyzuje celkovou strukturu vstupního souboru, avšak pokud je očekáván výraz, předá řízení syntaktické analýze zdola nahoru. Proto, abychom obsáhli náš bezkontextový jazyk v celé jeho šíří, vytvořili jsme LL gramatiku, s jejími pravidly (viz příloha 6.2) a tu jsme pak dále naimplementovali metodou rekurzivního sestupu.

Při návrhu a později i implementaci jsme museli nedeterminizmus mezi pravidly **34, 35, 36** ošetřit pomocí sémantické kontroly. V případě, že je načten token typu identifikátor (plně kvalifikovaný nebo jednoduchý), musíme zjistit, zda se jedná o funkci (v tom případě použijeme pravidlo s voláním funkce) a nebo ne (v tomto případě se jedná o výraz). Pro toto určení využíváme tabulku symbolů, ve které vyhledáváme pomocí funkcí `find_fun`, popřípadě `find_var`.

2.2.2 Syntaktická analýza zdola nahoru

Syntaktická analýza zdola nahoru byla použita pro vyhodnocování výrazů v modulu `expressions.c`. Pracuje podle precedenční tabulky (viz příloha 6.3) za pomoci zásobníku implementovaného jako dynamický vektor.

V případě zpracovávání identifikátorů je volána funkce `find_var`, alokováno potřebné místo pro výsledek a následně je generována příslušná instrukce. V případě konstanty je vytvořena dočasná proměnná typu `symbol_variable`, až poté se alokuje místo pro výsledek a je vygenerována instrukce.

2.3 Interpret

Interpret má za úkol zpracovávat seznamy instrukcí, nagenerované syntaktickým analyzátorem pomocí funkce `generate_instruction`. Každá funkce obsahu svůj vlastní list instrukcí. Ten je reprezentován pomocí dvousměrně vázaného seznamu, obsahující typ instrukce a ukazatel na potřebná data.

Nejčastěji se zde používá ukazatel na strukturu `symbol_variable`, která reprezentuje proměnnou programu, případně odkaz na strukturu `symbol_function`, obsahující informace o funkci. V rámci interpretace jednotlivých instrukcí jsou vyhodnocovány typové a inicializační kontroly. Pokud interpret narazí na běhovou chybu, ukončí program s patřičným návratovým kódem, jinak je interpret ukončen s návratovým kódem 0.

3 Vestavěné funkce, obsah souboru `ial.c`

Vestavěné funkce zabezpečují základní funkcionalitu jazyka IFJ16. Jsou implementovány v rámci vestavěné třídy `ifj16`, ke které se může implicitně přistupovat z jakéhokoliv programu napsaného v jazyce IFJ16. Popis těchto funkcí a algoritmů k nim použitých je rozepsán v následujících podkapitolách. Zaměřují se na vstup a výstup programu a též na práci s textovými řetězci. Dále je zde popsána implementace tabulky symbolů.

3.1 Tabulka symbolů

Tabulka symbolů je implementována binárním vyhledávacím stromem. Jeho výhodou je, že prvky jsou seřazeny podle klíče. Když chceme nějaký prvek přidat, musíme ho vložit na správné místo. V naší implementaci klíč reprezentuje jméno proměnné, funkce nebo třídy. Interpret využívá jednu velkou globální tabulku, která obsahuje pouze třídy. Každý prvek této tabulky obsahuje odkaz na tabulku symbolů dané třídy. V tabulce symbolů každé třídy se nachází globální proměnné a funkce dané třídy. Každý symbol funkce pak obsahuje kromě dalších informací i odkaz na lokální tabulku symbolů dané funkce. V této tabulce se nachází všechny lokální proměnné dané funkce, včetně argumentů.

V souboru `ial.c` se nachází implementovaný binární vyhledávací strom a všechny potřebné funkce pro práci s tabulkou symbolů, jako je přidávání prvků, vyhledávání prvků, kopírování i dealokace tabulek.

Ostatní funkce zaměřující se na jednotlivé symboly v tabulce se nachází v souboru `symbol.c`

3.2 Vestavěné funkce

3.2.1 Funkce `print`

Funkce `print` je odlišná od ostatních funkcí jazyka IFJ16, protože dokáže pojmout argument různých datových typů, které nazýváme term. Je schopna obsloužit i konkatenaci těchto termů. K této funkci proto přistupujeme odlišně i v samotném interpretu, který této funkci nepředává klasickou proměnnou, ale symbol z tabulky symbolů. Tento symbol se pak ve funkci `print` identifikuje a podle svého typu a obsah se vytiskne na standardní výstup.

3.2.2 Funkce závislé na `ial.c`

Podle námi vybraného zadání jsme implementovali funkci `find` pomocí Knuth–Morris–Prattova algoritmu. Funkce `sort` je implementována pomocí algoritmu Quick sort.

Funkce `find` hledá první instanci podřetězce v řetězci. K tomu využívá Knuth–Morris–Prattova algoritmu implementovaného v souboru `ial.c`. Tento algoritmus se skládá ze dvou částí. V první se stanoví vektor `f`, který určuje znak v podřetězci, na který se má algoritmus vrátit v případě neúspěchu. Ve druhé pak hledá shodu v řetězci. V případě neúspěchu se řídí podle dříve vytvořeného vektoru `f`. V případě úspěchu vrací pozici podřetězce v řetězci, při neúspěchu vrací hodnotu -1.

Funkce `sort` seřadí znaky v zadaném řetězci. Využívá k tomu algoritmus Quick sort implementovaný v souboru `ial.c`. Tento algoritmus rozdělí řetězec na dvě části, pak postupně porovnává znaky těchto částí se znakem ve středu řetězce. Pokud najde dvojici znaků, které jsou obráceně, vymění je a takto pokračuje až do chvíle, kdy se indexy překříží. Pak se funkce rekurzivně volá pro obě části samostatně až do momentu, kdy je řetězec seřazen.

3.2.3 Ostatní vestavěné funkce

Funkce `readInt`, `readDouble` a `readString` čtou ze standardního vstupu řetězec ukončený znakem konce vstupu nebo koncem řádku. Funkce `readString` pak tento načtený řetězec vrátí. Funkce `readInt` a `readDouble` využívají funkci `readString` pro načtení vstupního řetězce a následně jej převedou na celé nebo desetinné číslo. Pokud vstup není validní, interpret je ukončen s návratovým kódem 7.

Funkce `length`, `substr` a `compare` zabezpečují základní práci s řetězci. Naše implementace využívá funkcí z C-knihovny `string.h` na všechny potřebné operace.

4 Práce v týmu

Prioritou našeho týmu bylo projekt nepodcenit a mít ho hotový již na první pokusné odevzdání. Proto jsme se rozhodli, začít samostudiem materiálu z dřívějších let potřebných k vypracování projektu. Velkou výhodou bylo, že se téměř všichni členové týmu znali již z předchozí části studia. Nejvíce se to projevilo na bezproblémové komunikaci a pozitivním přístupu všech členů týmu.

Na konci září byly stanoveny pravidelné týdenní schůzky, na kterých jsme rozdělovali práci a diskutovali dosavadní postup projektu. Pro potřeby interaktivní komunikace byl založen soukromý skupinový chat na sociální síti Facebook. Pro potřeby videokonferencí jsme využívali službu Hangouts. Ke správě a verzování kódu byl použit systém GIT. Díky aktivnímu přístupu členů se nám povedlo projekt dokončit v námi stanoveném čase.

4.1 Rozdělení práce

Martin Ivančo	Tabulka symbolů, vestavěné funkce, testování
Filip Januš	Vyhodnocování výrazů, syntaktická analýza zdola nahoru, testování
Vladimír Jeřábek	Syntaktická analýza shora dolů, integrace, testování
Petr Jůda	Vedoucí týmu, lexikální analýza, interpret, dokumentace
Jakub Klemens	Lexikální analýza, interpret, testování

5 Závěr

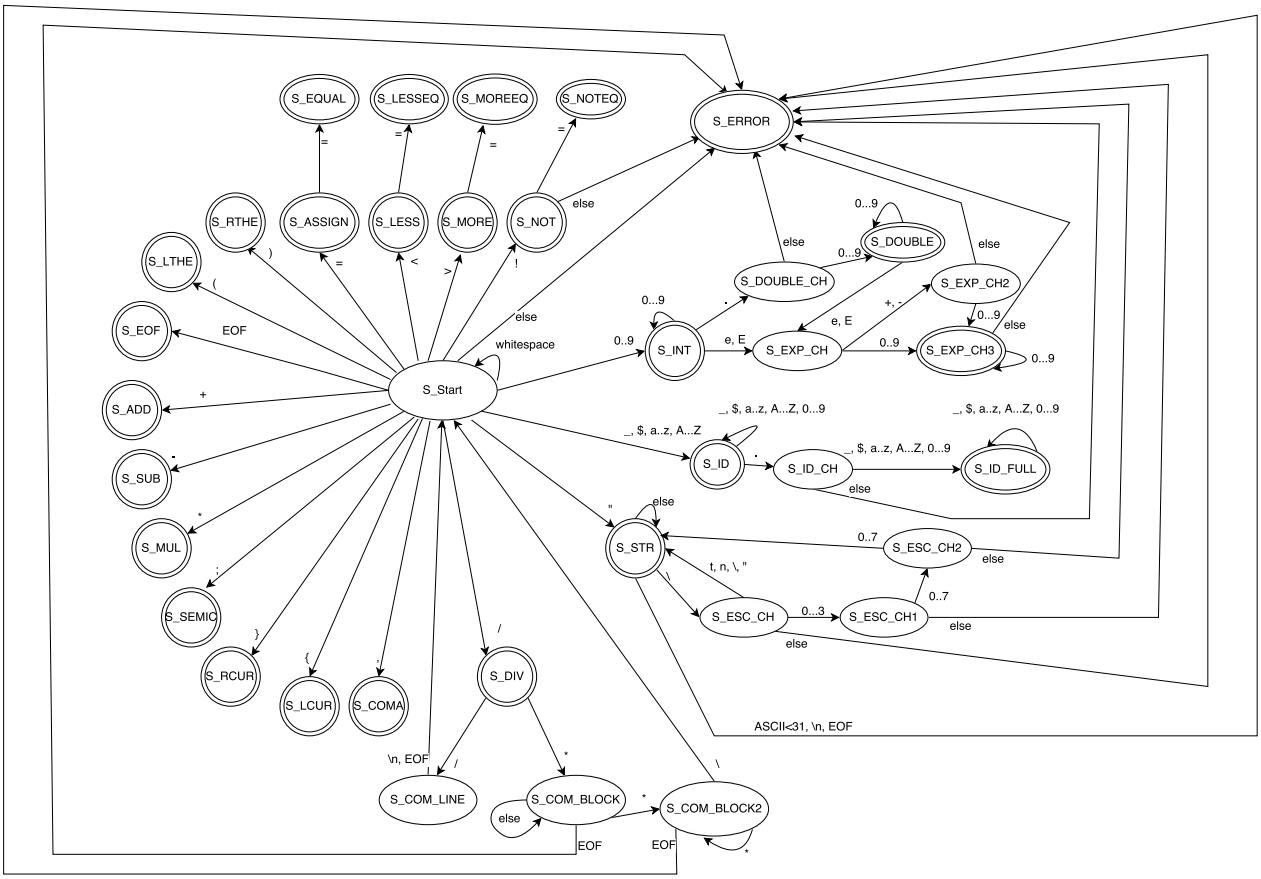
Projekt IFJ16 se nám povedlo dokončit podle našich představ. Díky dostatečné časové rezervě nám zbyl čas na implementaci rozšíření SIMPLE, které původně v plánu nebylo. Velkým přínosem do budoucna byla zkušenost s prací v týmu, kde jsme si ověřili, jak důležitá je dobrá komunikace mezi jednotlivými účastníky vývojového procesu.

Reference

[1] Studijní materiály do předmětů IFJ a IAL.

6 Přílohy

6.1 Konečný automat



6.2 LL gramatika

1:	$< prog >$	$\rightarrow$	$class\ id.simple\ \{ < inclass > \}\ < prog >$
2:	$< prog >$	$\rightarrow$	ε
3:	$< inclass >$	$\rightarrow$	$static\ < decl > < inclass >$
4:	$< inclass >$	$\rightarrow$	ε
5:	$< decl >$	$\rightarrow$	$< type >\ id.simple\ < varfunc >$
6:	$< decl >$	$\rightarrow$	$void\ id.simple\ < func >$
7:	$< varfunc >$	$\rightarrow$	$= < expr > ;$
8:	$< varfunc >$	$\rightarrow$	$;$
9:	$< varfunc >$	$\rightarrow$	$< func >$
10:	$< func >$	$\rightarrow$	$(< parlist >) \{ < infunc > \}$
11:	$< parlist >$	$\rightarrow$	$< type >\ id.simple\ < paritem >$
12:	$< parlist >$	$\rightarrow$	ε
13:	$< paritem >$	$\rightarrow$	$, < type >\ id.simple\ < paritem >$
14:	$< paritem >$	$\rightarrow$	ε
15:	$< infunc >$	$\rightarrow$	$< type >\ id.simple\ < locvar > < infunc >$
16:	$< infunc >$	$\rightarrow$	$< stat > < infunc >$
17:	$< infunc >$	$\rightarrow$	ε
18:	$< locvar >$	$\rightarrow$	$;$
19:	$< locvar >$	$\rightarrow$	$= < expr > ;$
20:	$< stat >$	$\rightarrow$	$id.simple\ < assfunc > ;$
21:	$< stat >$	$\rightarrow$	$id.full < assfunc > ;$
22:	$< stat >$	$\rightarrow$	$if\ (< expr >)\ < stat > < is\_else >$
23:	$< is\_else >$	$\rightarrow$	$else\ < stat >$
24:	$< is\_else >$	$\rightarrow$	ε
25:	$< stat >$	$\rightarrow$	$while\ (< expr >)\ < compstat >$
26:	$< stat >$	$\rightarrow$	$< compstat >$
27:	$< stat >$	$\rightarrow$	$return\ < expr\_non > ;$
28:	$< assfunc >$	$\rightarrow$	$= < assign >$
29:	$< assfunc >$	$\rightarrow$	$(< params >)$
30:	$< params >$	$\rightarrow$	$< value > < more\_param >$
31:	$< params >$	$\rightarrow$	ε
32:	$< more\_param >$	$\rightarrow$	$, < value > < more\_param >$
33:	$< more\_param >$	$\rightarrow$	ε
34:	$< assign >$	$\rightarrow$	$id.simple\ (< params >)$
35:	$< assign >$	$\rightarrow$	$id.full\ (< params >)$
36:	$< assign >$	$\rightarrow$	$< expr >$
37:	$< compstat >$	$\rightarrow$	$\{ < incompstat > \}$
38:	$< incompstat >$	$\rightarrow$	$< stat > < incompstat >$
39:	$< incompstat >$	$\rightarrow$	ε
40:	$< type >$	$\rightarrow$	int
41:	$< type >$	$\rightarrow$	$double$
42:	$< type >$	$\rightarrow$	$String$
43:	$< value >$	$\rightarrow$	$integer$
44:	$< value >$	$\rightarrow$	$decimal$
45:	$< value >$	$\rightarrow$	$text$
46:	$< value >$	$\rightarrow$	$id.simple$
47:	$< value >$	$\rightarrow$	$id.full$
48:	$< expr\_non >$	$\rightarrow$	$< expr >$
49:	$< expr\_non >$	$\rightarrow$	ε

6.3 Precedenční tabulka

	+	-	*	/	<	>	<=	>=	==	!=	ID	$	(	)
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<	<	<	<	<	>	>	<	>	<	>
!=	<	<	<	<	<	<	<	<	>	>	<	>	<	>
ID	>	>	>	>	>	>	>	>	>	>	#	>	#	>
$	<	<	<	<	<	<	<	<	<	<	<	#	<	?
(	<	<	<	<	<	<	<	<	<	<	<	#	<	=
)	>	>	>	>	>	>	>	>	>	>	#	>	#	>

Pravidla:
E → E+E
E → E-E
E → E*E
E → E/E
E → E<E
E → E>E
E → E<=E
E → E>=E
E → E==E
E → E!=E
E → (E)
E → i