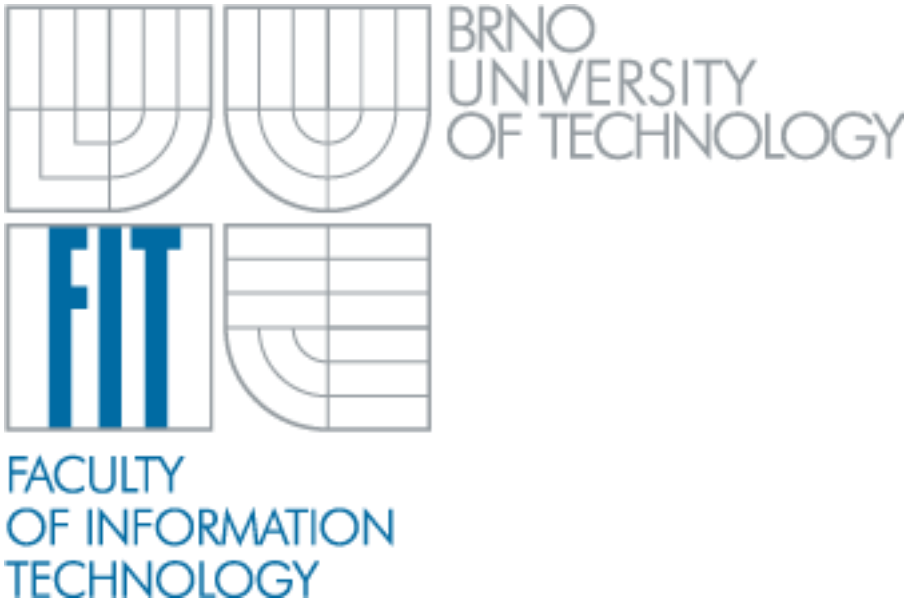


Dokumentace k projektu IFJ/IAL

Implementace interpretu imperativního jazyka IFJ14

Tým číslo 57

Varianta: B/1/I

Matúš Bútora (xbutor01) - 25 %

Daniel Dušek (xdusek21) – 25 %

Roman Jaška (xjaska00) – 25 %

Filip Kalous (xkalou03) – 25 %

Jiří Sochor (xsocho12) – 0 %

Obsah

1. Úvod

Interpret je speciální počítačový program, který slouží k přímému vykonání zápisu jiného programu z jeho zdrojového kódu, v daném jazyce. Program tedy není nutno převádět do strojového kódu.

Tento dokument se zabývá popisem implementace interpretu imperativního programovacího jazyka IFJ14.

Interpret načte zdrojový soubor zadaný jediným parametrem příkazové řádky, vyhodnotí správnost kódu z hlediska lexikálního, syntaktického a sémantického a následně kód interpretuje a vrátí 0, proběhne-li kontrola bez problémů. V opačném případě interpret skončí s návratovou hodnotou určující typ chyby.

2. Zadání

Jazyk IFJ14 je až na některé výjimky podmnožinou jazyka Pascal, který je staticky typovaný, hlavně procedurálně založený jazyk. V současnosti už využívaný hlavně k výuce programování a algoritmů.

Program se skládá z 3 hlavních stanic. Lexikálního analyzátoru, který je zodpovědný za načítání jednotlivých lexémů a vstupní kontrolu souboru na nepovolené znaky. Tento jest volán syntaktickým analyzátorem, který řídí celý překlad a kontroluje *syntaxní*, ale i *sémantickou* správnost programu. Poslední stanicí je interpret, který přijme od předchozí stanice instrukce, které vykoná.

2.1. Varianta zadání

Náš team řešil variantu zadání B/1/l, které zadává tyto podmínky:

- Pro implementaci vyhledávání podřetězce v řetězci musí být použit Boyer-Mooreův algoritmus (libovolný typ heuristiky)
- Pro implementaci řazení musí být použit algoritmus Quick sort
- Pro implementaci tabulky symbolů musí být použit binární vyhledávací strom

3. Implementace

3.1. Lexikální analyzátor

Lexikální analyzátor, zvaný scanner se stará o načítání lexémů ze zdrojového souboru a o jejich zpracování konečným automatem. Každý lexém je po zpracování odeslán (uložen) jako token. K němu pak přistupuje syntaktický analyzátor, neboli parser.

V konkrétním případě je zdrojový soubor zpracováván znak po znaku a podle aktuálního stavu v konečném automatu jsou žádané znaky propuštěny a akumulovány, nežádoucí znaky jsou pak vyhodnoceny jako terminátory výrazu a podnět k tomu, aby byl token uložen a odeslán.

Jsme-li v počátečním stavu, zpracováváme, avšak ignorujeme všechny bílé znaky, neb z hlediska interpretu nejsou podstatné.

V Příloze 1 je uveden model konečného automatu ilustrující funkci lexikálního analyzátoru.

3.2. Syntaktický analyzátor

Vzhledem k tomu, že popis „Syntaxí řízený překladač“ by na náš interpret seděl, je tedy syntaktický analyzátor jádrem celého projektu. Jeho úkolem je překládat zdrojový kód na instrukce pro interpret a po čas této operace zároveň kontrolovat syntaktickou a sémantickou správnost kódu.

Je-li v kódu sémantická či syntaktická chyba, syntaktický analyzátor terminuje překlad s návratovým kódem odpovídajícím kódu chyby specifikovaným zadáním.

Syntaktický analyzátor si pro zajištění zdrojového kódu volá ve smyčce funkci lexikálního analyzátoru getNextToken().

V této fázi se provádí i kontrola datových typů a některá přetypování.

Kontrola syntaktické správnosti je prováděna podle pravidel LL gramatiku, jejíž pravidla jsou uvedeny v příloze č. 2, precedenční tabulka pak v příloze č. 3.

3.3. Interpret

Interpret je finální komponentou celého projektu, zodpovědnou za interpretaci instrukcí, které přijme od syntaktického analyzátoru.

Interpret přijímá instrukce a data, se kterými pracují v podobě jednosměrně vázaného seznamu. Při interpretaci funkcí, ale i některých instrukcí, je třeba interpret volat rekurzivně tak, aby byly vykonány všechny operace.

Při operacích s datovým typem realu a integeru dochází v této fázi k přetypování, vyžaduje-li to situace.

3.4. Binární vyhledávací strom

K implementaci tabulky symbolů jsme dle zadání použili binární vyhledávací strom (dále jen BVS). Jde o abstraktní datovou strukturu založenou na binárním stromu, kde o uložení uzlu v rámci celého stromu (levý či pravý podstrom) rozhoduje hodnota klíče.

Vytváříme klíče třech typů. Pro identifikátory používáme název identifikátoru s prefixem “V”, pro funkce používáme prefix “F” a pro samostatné hodnoty generujeme pseudonáhodný klíč. Generace pseudonáhodného klíče je založena na čase a vestavěné funkci jazyka C, srand().

Pro jednotlivé funkce vytváříme vždy lokální tabulku symbolů, tedy její vlastní BVS.

Nejčastější využití tabulky symbolů v tomto projektu je pro vyhledávání uzlu podle klíče.

3.5. Quick sort

Jde o jeden z nejrychlejších a běžných algoritmů, tedy samozřejmě jeden z nejpoužívanějších, hlavně z důvodů jeho jednoduchosti. Časová složitost tohoto algoritmu je v nejhorším případě $O(N^2)$, v nejlepším případě je to pak O(N log N), tzv. logaritmická složitost.

Pro co nejefektivnější práci quicksort algoritmu je třeba dobře zvolit pivot, v našem projektu je využita metoda mediánu z tří prvků. Metoda spočívá v (pseudo)náhodném výběru 3 prvků, ze kterých vybereme medián a ten použijeme jako pivot. V konkrétním případě jde o první, prostřední a poslední prvek.

3.6. Boyer-Mooreův algorimus

Algoritmus funguje na principu porovnávání dvou řetězců, odzadu. Náš řetězec položíme „proti“ řetězci, se kterým budeme porovnávat (dále jako porovnávaný řetězec) a porovnáme poslední znak našeho řetězce se znakem odpovídajícím indexu posledního řetězce našeho znaku v porovnávaném řetězci. Nedojde-li ke shodě, posuneme náš řetězec směrem doprava oproti porovnávanému řetězci o tolik indexů, kolik je délka našeho řetězce na znak, který se shoduje s právě kontrolovaným znakem. A proces opakujeme.

Dojde-li ke shodě posledního znaku našeho řetězce, posuneme se s kontrolou našeho řetězce o znak doleva a opět porovnáváme.

4. Teamová práce

Na projektu jsme na začátku plánovali pracovat v pěti lidech, avšak střet s realitou nás poučil, že žádný plán neobstojí. Již během prvních schůzek se ukázalo, že jeden z našich spolupracovníků bude mít problém spolupracovat, takže jsme na projektu byli nuceni dělat pouze ve čtyřech lidech.

4.1. Simultánní práce na více částech zároveň

Abychom minimalizovali čas, který stráví každý jednotlivec našeho teamu čekáním na dokončení jemu předcházející části, stanovili jsme si vnitřní „jazyk“, kterým budou naše části projektu mezi sebou komunikovat.

Každý z pracovníků si takto tedy vytvořil speciální soubor, ve kterém si nasimuloval očekávaný vstup od předchozí komponenty a pracoval s datovým tokem z tohoto souboru. Díky této strategii jsme byli schopní pracovat zároveň, což se s blížící se deadline ukázalo jako velice výhodné, neb bychom bez ní neměli šanci odevzdat včas.

V moment kdy byla jedna z komponent úspěšně dokončena, jsme ji spojili s komponentou po ní následující a postupně jsme projekt skládali dohromady.

Jednou z prvních jednotek, která byla hotová, byl lexikální analyzátor, po němž hned následoval syntaktický analyzátor a precedence.

Nejpozději zhotovenou částí byl podle našeho původního předpokladu interpret.

4.2. Synchronizační prostředky a komunikace

Při vývoji jsme použili služby známé webové stránky github.com, která díky našim školním emailům umožňuje používání bezplatných privátních repositářů. V užším výběru byl i bitbucket, který funguje na stejném principu jako github, ale je kompletně zdarma. Díky studentské slevě na privátní repositáře však nakonec od tohoto bylo upuštěno.

Ke komunikaci v teamu jsme využili všech dostupných prostředků, počínajíc GSM zařízeními, pokračujíc přes jabber, skype a dnes velice rozšířený Facebook, konče špionážními satelity.

4.3. Netradiční techniky

V průběhu projektu jsme dali šanci brainstormingu, mind-mappingu i skupinové lázni ve vířivce. Také nás dohromady spojila nenávist vůči pátému členovi našeho teamu, který s námi odmítal komunikovat a neposkytl žádné vysvětlení ke svým akcím. Všechny tyto techniky se dle našich závěrů taktéž příznivě podepsaly na výsledku naší práce.

4.4. Rozdělení práce

Práce na lexikální analýze a dokumentaci byla přidělena Danielu Duškovi (xdusek21), neb jsme teamově shledali, že k těmto dvěma úkonům má všechny předpoklady.

Syntaktická analýza byla přidělena Romanu Jaškovi (xjaska00), neb jsme teamově rozhodli, že jeho programátorské zkušenosti byli v době začátku prací na projektu nejvyšší. Také se k práci samostatně přihlásil.

Precedenční část a implementace algoritmů byla přidělena Filipu Kalousovi (xkalou03), dobrovolně se přihlásil.

Interpretační část vypracoval Matúš Bútora. Práce na něho dobrovolně zbyla.

Sémantická část byla původně uvažována pro našeho neaktivního pracovníka, takže ji nakonec museli vypracovat Filip Kalous společně s Romanem Jaškou.

4.5. Odůvodnění 0 bodů pro Jiřího Sochora

Pan Sochor se objevil na dvou schůzkách, během kterých si dobrovolně zvolil sémantickou část.

Následně jsme od pana Sochora slyšeli jen výmluvy typu: sestře se narodilo dítě, přehnal jsem to o víkendu a skončil v nemocnici, mám zápal plic a ležím v nemocnici..., a na jeho facebookovém účtu přidával fotky z akcí, kterých se v době, kdy měl ležet v nemocnici účastnil.

Když jsme se ho následně pokusili vícekrát kontaktovat, pouze nás ignoroval, vedoucího našeho teamu si dokonce zablokoval na facebooku a neodpovídal na telefonní volání.

Toto chování považujeme za nepřípustné, a neboť pan Sochor vůbec nepřidal ruku k dílu, ani v nejmenším, považujeme za správné nepřidělit mu body.

5. Závěr

V projektu jsme úspěšně implementovali lexikální a syntaktický analyzátor a interpret imperativního jazyka IFJ14.

Projekt nás naučil, jak pracovat v teamu a posunul hranice našeho programátorského poznání.

6. Metriky kódu

Počet řádků kódu: 5093

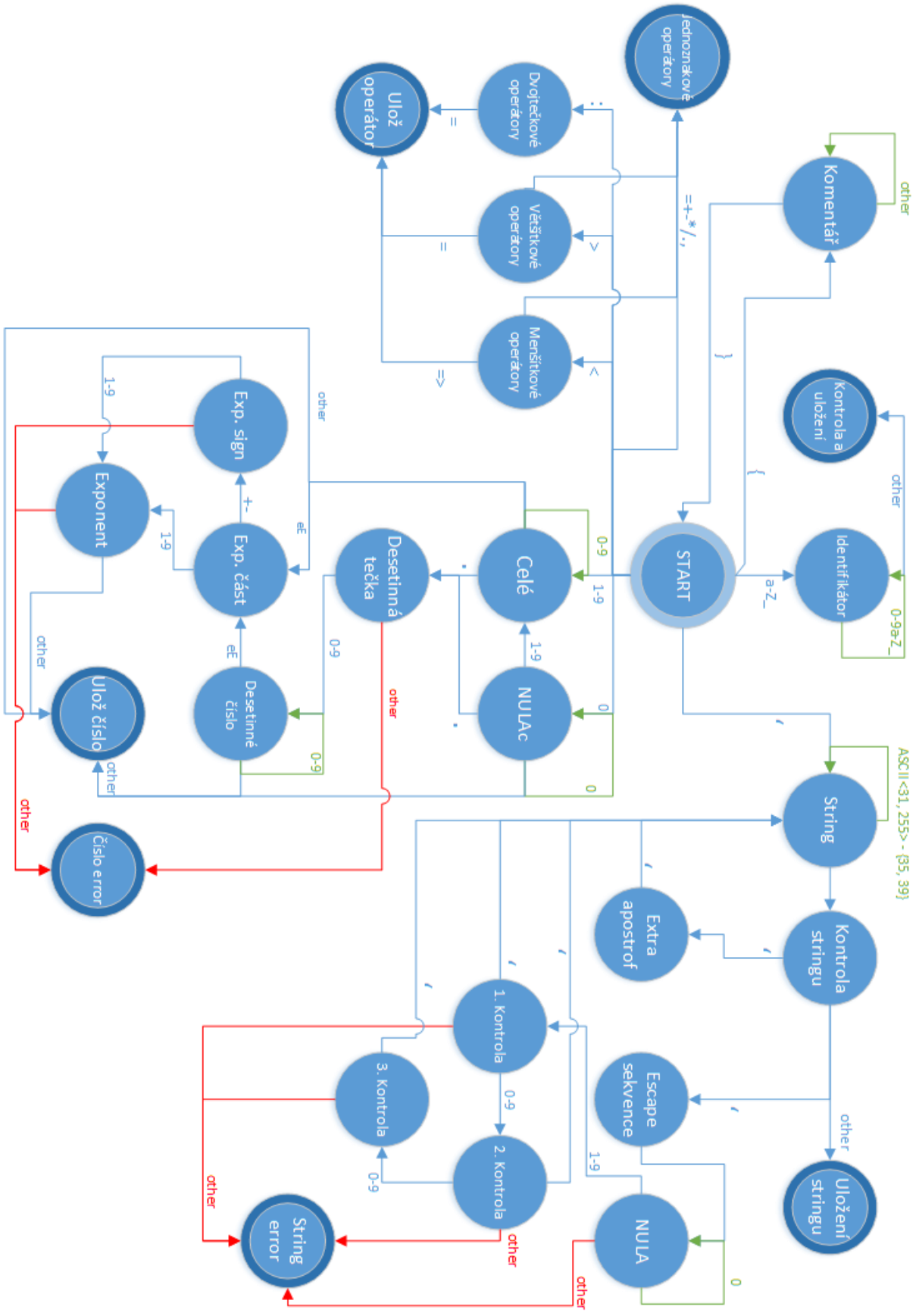
Počet zdrojových souborů: 17 + Makefile

Velikost spustitelného souboru: 88473 bytes

7. Zdroje

[1] Prof. Ing. Jan M. Honzík, CSc., ALGORITMY – Studijní opora. Verze: 14-Q, 2014.

Příloha č. 1 – Ilustrační návrh konečného automatu



Příloha č. 2 – LL gramatika

1. *[program] → [var_def_block] [fun_def_list] [main]*
2. *[var_def_block] →* ***var*** *[var_def] [var_def_list]*
3. *[var_def_block] → ε*
4. *[var_def_list] → [var_def] [var_def_list]*
5. *[var_def_list] → ε*
6. *[var_def] → [var_id] : [type] ;*
7. *[fun_def_list] →* ***function*** *[fun_id] ([param_list]) : [type] ; [fun_body] ; [fun_def_list]*
8. *[fun_def_list] → ε*
9. *[fun_body] →* ***forward***
10. *[fun_body] → [var_def_block] [body]*
11. *[body] →* ***begin*** *[stmt_list]* ***end***
12. *[main] → [body]*
13. *[stmt_list] → [stmt] [stmt_more]*
14. *[stmt_list] → ε*
15. *[stmt_more] → ; [stmt_list]*
16. *[stmt_more] → ε*
17. *[stmt] → [body]*
18. *[stmt] → [var_id]*
19. *[stmt] →* ***if*** *[expr]* ***then*** *[body]* ***else*** *[body]*
20. *[stmt] →* ***while*** *[expr]* ***do*** *[body]*
21. *[stmt] →* ***readln*** *[id]*
22. *[stmt] →* ***write*** *([term_list])*
23. *[assign] → [expr]*
24. *[assign] → [fun_id] ([term_list])*
25. *[term_list] → ε*
26. *[term_list] → [term] [term_more]*
27. *[term_more] → , [term_list]*
28. *[term_more] → ε*
29. *[param_list] → ε*
30. *[param_list] → [param] [param_more]*
31. *[param_more] → ε*
32. *[type] →* ***integer***
33. *[type] →* ***real***
34. *[type] →* ***string***
35. *[type] →* ***boolean***
36. *[param] → [var_id] : [type]*
37. *[term] → [var_id]*
38. *[term] → [t_expr_int]*
39. *[term] → [t_expr_dou]*
40. *[term] → [t_expr_str]*
41. *[term] → [t_expr_boo]*

Příloha č. 3 - Precedenční tabulka

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