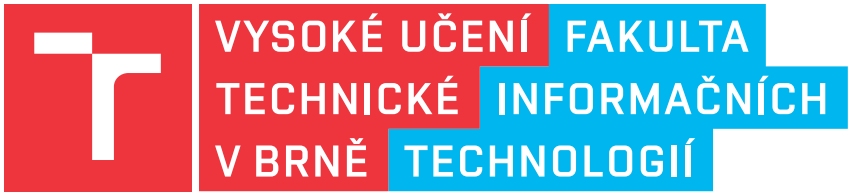

Dokumentace

Překladač imperativního jazyka IFJ18

Tým 89, varianta I

6. prosince 2018

Vojtěch Novotný	**(xnovot1f)**	25 %
Tomáš Zálešák	(xzales13)	25 %
Robin Skaličan	(xskali19)	25 %
Tomáš Smädo	(xsmado00)	25 %

Obsah

1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ18, jenž je zjednodušenou podmnožinou jazyka Ruby a přeloží jej do cílového mezikódu IFJcode18. Program funguje jako konzolová aplikace, které načítá zdrojový program ze standardního vstupu a generuje výsledný mezikód na standardní výstup nebo v případě chyby vrací odpovídající chybový kód.

2 Návrh a implementace

2.1 Lexikální analýza

Nejprve jsme implementovali lexikální analýzu. Klíčová je funkce `getToken`, jež vrací následující token ze zdrojového souboru. Funkce načítá znak po znaku ze standardního vstupu, přeskakuje komentáře a prázdné řádky, vrací podstatné informace formou struktury `token`. `Token` se skládá z typu a dat. Typ je určen výčtovým typem `enum` a data jsou ukazatel na data podle toho, o jaký typ `tokenu` se jedná. Celý proces je implementován jako deterministický konečný automat.

2.2 Syntaktická analýza

Syntaktická analýza se dělí na kontrolu rekurzivním sestupem a kontrolu správnosti výrazů precedenční analýzou. Tyto části spolu komunikují při průchodu, kde je precedenční analýza volána rekurzivním sestupem.

2.2.1 Syntaktická analýza shora-dolů

Rekurzivní sestup postupuje vstupním kódem po tokenech získaných z lexikální analýzy a porovnává jejich správnost s pravidly z LL tabulky vytvořenou na základě LL gramatiky. Pokud je podle pravidel z LL tabulky očekáván výraz, je následující token předán precedenční analýze

2.2.2 Zpracování výrazů pomocí precedenční syntaktické analýzy

Precedenční analýza zajišťuje syntaktickou kontrolu výrazů. Načítá tokeny ze vstupu. Podle tokenu a nejvyššího symbolu na zásobníku se rozhoduje, zda je vstup validní, či nikoliv. Jako zásobník je využit obousměrně propojený seznam. Samotné vyhodnocení je realizované pomocí statické tabulky, díky které poznáme prioritu a asociativitu operátorů.

2.3 Sémantická analýza

Sémantická analýza probíhá současně se syntaktickou analýzou a kontroluje:

- definice funkcí, správný počet parametrů při volání funkcí, redefinice funkcí (u vestavěných funkcí probíhá také kontrola správných typů parametrů)
- deklarace proměnných, pokus o redefinici funkce jako proměnné, použití proměnných v správných rámcích kódu
- v rámci precedenční analýzy provádí typové kontroly, při kterých může provádět implicitní typové konverze (float - integer)

Na všechny uvedené kontroly se využívají data uložené do tabulky symbolů, která je popsaná níže.

2.4 Generování cílového kódu

Generování cílového kódu ifjcode18 je provedeno za běhu syntaktické analýzy. Podle aktuálně používaného pravidla gramatiky se vypisují potřebné příkazy. Kód je generován na standardní výstup.

3 Speciální algoritmy a datové struktury

3.1 Tabulka symbolů

Pro tabulku symbolů jsme si zvolili variantu č. 1 – implementace formou binárního vyhledávacího stromu. Využívá se pro sémantické kontroly vstupného kódu. V globální tabulce symbolů (GTS) jsou uloženy informace o vestavěných a uživatelem definovaných funkcích – počet parametrů a jestli byla funkce definována. V lokální tabulce symbolů (LTS) jsou uloženy názvy definovaných proměnných a jejich datové typy.

3.2 Zásobník symbolů pro precedenční syntaktickou analýzu

Byl použit obousměrně provázaný seznam. Jako obdoba funkce Push slouží DLInsertFirst, jako Pop zase DLCopyFirst a DLDeleteFirst.

4 Práce v týmu

4.1 Způsob práce v týmu

Práce na implementaci nezačala hned po prvním osobním setkání, nýbrž bylo rozhodnuto o tom, aby se každý člen plně seznámil se zadáním a měl alespoň částečnou vizi pro implementaci jednotlivých částí projektu do příštího setkání, kde jsme si určili týmovou vizi implementace a rozdělení jejich částí. Po dokončení všech částí a jejich propojení jsme začali společně projekt testovat jako celek.

4.1.1 Verzovací systém

Pro verzování byl použit repositář na webu GitHub. Každý pracoval ve své větvi a po funkční implementaci daného problému se změny aplikovali do hlavní větve. Spojováním těchto funkčních částí vznikla finální aplikace, jež byla podrobena finálnímu testování a debuggování.

4.1.2 Komunikace v rámci týmu

Komunikace probíhala přes internet pomocí aplikace Messenger a na pravidelných týdenních setkáních na akademické půdě FITu.

4.2 Rozdělení práce

Práci na projektu jsme si rozdělili rovnoměrně. Tabulka 1 shrnuje rozdělení práce v týmu mezi jednotlivými členy.

Člen týmu	Přidělená práce
Vojtěch Novotný	vedení týmu, generování ifjcode18, lexikální analýza
Tomáš Zálešák	precedenční analýza, testování, dokumentace
Tomáš Smädo	sémantická analýza, implementace tabulky symbolů, dokumentace
Robin Skaličan	syntaktická analýza, dokumentace

Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

5 Závěr

Z počátku bylo těžké pochopit komunikaci jednotlivých částí a jejich implementaci. Pravidelné schůzky, „domácí úkoly“ v podobně nastudování si témat v nichž jsme neměli úplně jasno a dobrá atmosféra ve společném chatu velmi pomohla. Díky tomu, že jsme si určili již před implementací, co která část bude přesně dělat, dostávat za parametry, kdy bude volaná a co bude vracet bylo propojování dokončených částí jen s minimálními problémy. Implementace neprobíhala v časové tísni, vše jsme si časové rozvrhli pomocí „deadlinů“, které byly víceméně dodržovány. Průběžná implementace byla kontrolována na schůzkách, včetně společného řešení nejasností. Díky funkční týmové práci jsme získali cenné zkušenosti pro práci ve více lidech na společném projektu.

6 Přílohy

	+	-	*	/	<	<=	>	>=	==	!=	(	)	T	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<							<	>	<	>
<=	<	<	<	<							<	>	<	>
>	<	<	<	<							<	>	<	>
>=	<	<	<	<							<	>	<	>
==	<	<	<	<							<	>	<	>
!=	<	<	<	<							<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
T	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	

Tabulka 2: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů

1. <prog> -> <main>
2. <main> -> <stat><main>
3. <main> -> <func><main>
4. <main> -> EOL
5. <main>
6. <main> -> EOF
7. <stat> -> id = <expr> EOL
8. <stat> -> id EOL
9. <stat> -> <expr> EOL
10. <stat> -> if <expr> then EOL
11. <st-list>
12. else EOL
13. <st-list>
14. end EOL
15. <stat> -> while <expr> do EOL
16. <st-list>
17. end EOL
18. <stat> -> id = <func-call>
19. <stat> -> <func-call>
20. <stat> -> inputs()
21. <stat> -> inputi()
22. <stat> -> inputf()
23. <stat> -> print(<arg-list-print>)
24. <stat> -> length(<arg-list>)
25. <stat> -> substr()
26. <stat> -> ord()
27. <stat> -> chr()
28. <st-list> -> <stat><st-list>
29. <func> -> def id (<param-l> EOL
30. <st-list> END
31. <param-l> -> <param> <param-l2>
32. <param-l2> -> , <param> <param-l2>
33. <param-l2> ->)
34. <param> -> id
35. <func-call> -> id <arg-list>
36. <func-call> -> id (<arg-listb> EOL
37. <arg-list> -> <arg> <arg-list2>
38. <arg-list> -> EOL
39. <arg-list2> -> , <arg> <arg-list2>
40. <arg-list2> -> EOL
41. <arg-listb> -> <arg> <arg-list2b>
42. <arg-listb> ->)
43. <arg-list2b> -> , <arg> <arg-list2b>
44. <arg-list2b> ->)
45. <arg-list-print> -> <arg> <arg-list2>
46. <arg> -> id
47. <arg> -> int
48. <arg> -> float
49. <arg> -> string

Tabulka 3: LL – gramatika řídící syntaktickou analýzu