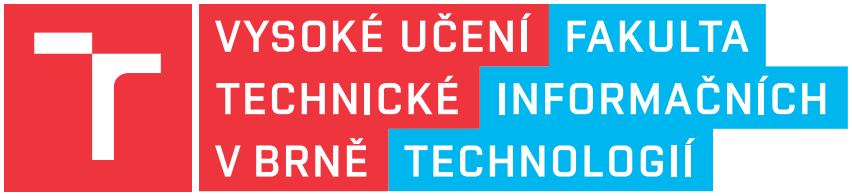


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ21

Tým 025, varianta I

Knapovský Jan	(xknapo05)	32 %
Hanus Igor	(xhanus19)	32 %
Ondroušek Adam	(xondro09)	18 %
Karásek Filip	(xkaras39)	18 %

8. prosince 2021

Obsah

1 Úvod

Základním cílem projektu bylo vytvořit kompilátor imperativního jazyka IFJ21, omezenou podmnožinu silně typované varianty jazyka LUA - TEAL. Cílovým jazykem pak byl IFJ21code, interpretovaný jazyk s tříadresnými, potažmo zásobníkovými, instrukcemi.

Výsledný kompilátor funguje jako konzolová aplikace pracující se standartním vstupem a výstupem, aneb jako filtr. Implementován je v jazyce C, za použití standartů POSIX-2008 a dřívějších.

2 Návrh a implementace

Projekt jmse rozčlenili do několika modulů. Tyto moduly, stejně jako jejich rozhraní a spolupráce budou představeny v této kapitole.

2.1 Lexikální analýza

Lexikální analyzátor, jelikož je první částí jak z pohledu průchodu dat, tak z pohledu přednášené látky, byl implementován jako první.

Jelikož můžeme považovat IFJ21 v rámci rozumných mezí za regulární jazyk, byl pro analýzu a rozlišování lexémů použit stavový automat. Hlavní část lexikálního analyzátoru tedy tvoří deterministický konečný stavový automat (viz příloha A). Tento stavový automat za pomoci výstupní a vstupní fronty, ignorujíc bílé znaky, rozliší jednotlivé lexémy, kterým jsou následně přiřazovány patřičné typy podle koncového stavu automatu.

V závislosti na typu lexému jsou následně vyextrahovány argumenty, tj. numerické literály jsou převedeny na odpovídající numerické datové typy, řetězcové literály jsou zbaveny obalujících uvozovek a pomocí jednoduchého průchodu s look-ahead jsou převedeny všechny escape sekvence na speciální znaky. Takto připravený token je pak předán k dalšímu zpracování nadřízenému modulu.

V případě, že je nalezen neplatný lexém, je vygenerováno chybové hlášení a nadřízenému modulu je předán chybový token, značící lexikální chybu.

Rozhraní lexikálního analyzátoru je tvořeno jednou jedinou funkcí `getNextToken`, která spustí lexikální analýzu a vrátí jeden nalezený lexém, neboli token. Jedná se o funkci s vnitřním stavem, která při následujícím volání bude pokračovat se čtením vstupu, kde přestala.

2.2 Syntaktická analýza

Syntaktická analýza je implementovaná pomocou rekurzívneho zostupu na základe LL gramatiky (viď příloha B) a zároveň kontroluje základné semantické pravidlá ako chybné definície funkcií alebo duplikované názvy. Vyhodnocovanie výrazov riadi precedenčná tabulka (viď příloha D) cez zásobník.

2.3 Sémantická analýza

Hlvaními dvěma úkoly sémantického analyzátoru je kontrola sémantiky, kompatibility typů a volání funkcí, a řízení generátoru kódu.

Implementačně není příliš zajímavý. Hlavním problémem není algritmus analýzy, ale práce s pamětí a přemostění mezi syntaktickým analyzátorem a generátorem kódu.

Vzhledem k omezeným možnostem zvoleného standartu volání funkcí, bylo nutné přistoupit k odchytávání volání vestavěných funkcí z rodiny `readX` a `write`. Tyto funkce jsou potom nahrazeny speciálními instrukcemi přímo v kódu, jinak také známé jako in-line funkce.

2.4 Generování cílového kódu

Cílovým kódem je, jak již bylo zmíněno, IFJ21code. Bylo rozhodnuto, že z možných přístupů, tříadresného a zásobníkového, bude použit ten prvý.

2.4.1 Správa paměťového prostoru

Vzhledem k použitému tříadresnému přístupu, který operuje s pomocnými proměnnými, a k nutnosti ony porměnné v cílovém jazyce IFJ21code předem deklarovat, byla správa paměti ve výsledném programu řešena metodou inspirovanou prací s registry v jazyce assembly x86.

Paměťová místa (proměnné) byly rozděleny do dvou typů, na dočasné (registry) a trvalé (proměnné). Zatímco proměnné zůstávají plně v režii nadřazeného modulu, pomocí funkcí a řídících struktur poskytnutých rozhraním, k registrům je přistupováno zvláštním způsobem.

Při provádění jakékoliv instrukce generující jakýkoliv typ výsledku, je tento výsledek uložen do volného registru (pokud neexistuje žádný volný registr, je vytvořen registr nový). Tento registr je potom zpřístupněn nadřízenému modulu pro použití mezivýsledku v navazujících výpočtech. Jakmile nadřízený modul tento mezivýsledek již nepotřebuje, vrátí tento registr zpět generátoru kódu, který jej zařadí do zásobníku volných registrů. Takto uvolněný registr je při první příležitosti znovu použit.

Pokud je zásobník volných registů plný, je tento registr *zapomenut* a dále nevyužíván. Platnost registru končí s koncem rámce.

2.4.2 Generování výrazů

Generování výrazů je řízeno přímo nadřízeným modulem. Tzn. nadřízený modul přímo nařizuje jaké instrukce, potažmo skupiny instrukcí, budou použity na jaké proměnné/registry. Základním předpokladem pro nejmenší paměťovou stopu při generování výrazů je, že nadřízený modul *nedrží*, registry použité k ukládání mezivýpočtů a okamžitě po použití je uvolňuje, hlásíc tuto skutečnost generátoru kódu. Ten následně takto uložený registr znovupoužije v následujících krocích výpočtu. Z testování provedeným autorem generátoru kódu vyplynulo, že většina výrazů je možna vyřešit za použití ne více než 5-ti registrů.[1]

2.4.3 Generování struktur řízení toku programu

Struktury řízení toku programu, ve zdrojovém jazyce struktury while a if, jsou generovány pomocí podmíněných a nepodmíněných skoků na návěští. Jména návěští jsou automaticky procedurálně generovaná, za účelem globální unikátnosti návěští. Generátor kódu však tato jména skrývá před uživatelem, pro záznam rozpracovaných podmínek a cyklů používá zásobník, takže uživatel není zatěžován podrobnějším řízením této části.

2.4.4 Standard volání funkcí

Volání funkcí probíhá standardizovaným způsobem, a to tak, že před samotným voláním musí být generátoru kódu předány hodnoty argumentů. Tyto hodnoty generátor kódu uloží do nového rámce. Tento rámec se po provedení volání stává lokálním rámcem a uvnitř funkce se tudíž s argumenty pracuje stejně jako s lokální proměnnou. Stejně tak před návratem je potřeba připravit návratové hodnoty do aktuálního rámce a po návratu tyto hodnoty přečíst z teď již dočasného rámce do proměnné dle zadání.

Pro zabránění náhodného spuštění funkce při procházení programu shora dolů, je funkce obalena skokem, který se stará, aby program deklaraci funkce přeskočil, mělo-li by se stát že spouštění instrukcí dojde k začátku funkce.

Vestavěné funkce z rodiny `readX` a `write` se však této konvenci vymykají, z důvodů nekompatibility způsobu volání s konvencí volání použitou v našem generátoru kódu, nebo, jak je tomu v případě `read` pro zbytečnou složitost.

Místo toho jsou tyto funkce kompilovány jako in-line, vkládáním instrukcí pro provedení sémantiky funkce přímo na dané místo v kódu, bez skoků.

[1] Nejedná se o nijak statisticky ověřené tvrzení, pouze experimentální ověření na omezeném počtu vstupů.

2.5 Kompilace projektu pro spuštění

Projekt byl testován a projektován pro překlad na kompilátoru `gcc v7.5.0` se standardními knihovnami jazyka C, vyhovující standardům `c99` a `POSIX-2008`.

K projektu je přiložen soubor `makefile` pro automatizovaný překlad pomocí programu `GNU make`.

3 Speciální algoritmy a datové struktury

Při tvorbě překladače jsme implementovali několik speciálních datových struktur, které jsou v této kapitole ve zkratce představeny představeny.

Vzhledem k tomu, ze projekt byl psán za běhu, podle našich aktuálních znalostí a informací týkajících se formálních jazyků a našich aktuálních časových možností, nedošlo k implementaci jednotných knihoven pro veškeré datové struktury. Datové struktury byly tedy požívány a programovány ad-hoc, s funkcemi které zrovna byly třeba a nad daty které bylo třeba zpracovat.

3.1 Binární strom

Jediná struktura programovaná primárně jako knihovna za jediným účelem, a to použití jako tabulka symbolů. K tomuto účelu je používán v četných instancích, jako tabulka jak globálních, tak dílčí tabulky lokálních souborů.

3.2 Zásobník

Struktura používaná v mnoha instancích a implementacích napříč celým projektem.

Nejpodstatnější ze všech implementací zásobníků v projektu, je implementace rozšířeného zásobníku pro precedenční analýzu výrazů. Další použití našly zásobníky v generátoru kódu, kde byly použity za účelem sledování právě otevřených vetví struktur řízení toku programu.

3.3 Fronta

Posledním abstraktním datovým typem, který má v projektu nenahraditelné místo, je fronta, použitá v lexikálním analyzátoru ve dvou instancích, jako vstupní a výstupní. Tato implementace není nutná, ani nijak nepřidává na efektivitě lexikálního analyzátoru, pouze zlepšuje čitelnost a pochopitelnost kódu za cenu složitější podpůrné struktury.

4 Práce v týmu

4.1 Spolupráce v týmu

Projekt byl psán postupně, práce nebyla nijak předem rozdělena ani plánována.

4.1.1 Verzovací systém

Pro správu zdrojového kódu byl použit verzovací systém `Git`, jako vzdálený hostitel pak `GitHub`

4.1.2 Komunikace v rámci týmu

Všechna komunikace ohledně projektu probíhala přes službu Discord nebo osobně. Nejdůležitější témata byla probírána buď osobně, nebo přes hlasové služby Discord.

4.2 Rozdělení práce mezi členy týmu

Jak již bylo uvedeno, práce na projektu nebyla předem nijak rozdělena. Tabulka 1 tedy shrnuje rozdělení, jaké vyvstalo v průběhu řešení projektu.

Člen týmu	Přidělená práce
Jan Knapovský	teoretická podpora týmu, dokumentace, generování cílového kódu, lexikální analyzátor
Igor Hanus	syntaktická analýza, precedenční analýza, technická podpora týmu
Adam Ondroušek	tabulka symbolů, sémantický analyzátor, příprava testů
Filip Karásek	tabulka symbolů, sémantický analyzátor, příprava testů

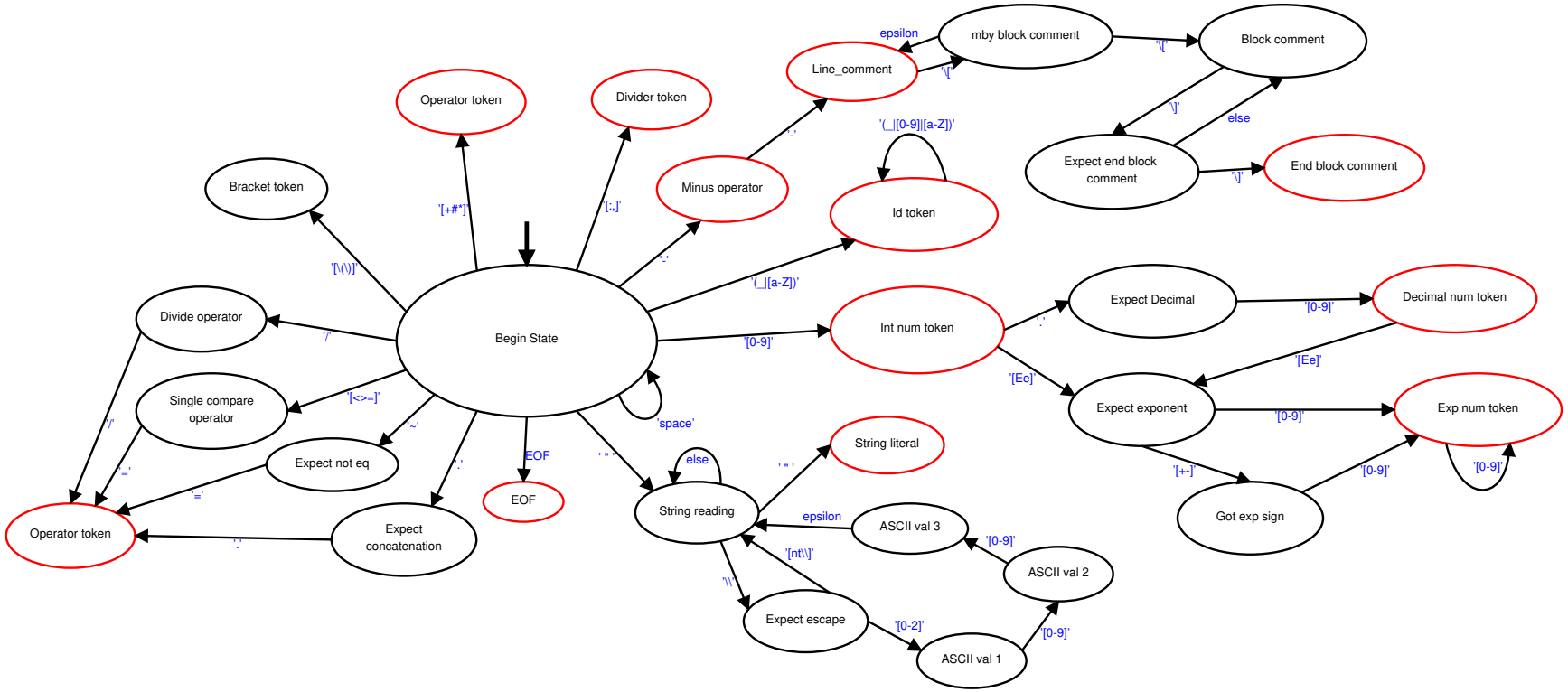
Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

5 Závěr

Projekt se ze začátkuzdál velice složitý a vzhledem k faktu, že znalosti, potřebné k vypracování projektu, jsme získávali až později v semestru, až nezvládnutelný. Kvůli tomu bylo velice těžke předem rozdělit úkoly a práci, takže ve výsledku každý člen týmu pracoval na čem uznal za vhodné, potažmo na tom co zbylo.

Bohužel i přesto, že náš tým byl sestaven dostatečně brzy, se ukázalo, že rozdíly ve znalostech jsou značné a dohánění bude velice náročné. Shodou menších chyb

A Diagram konečného automatu specifikující lexikální analyzátor



Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

B LL – gramatika

1. <program> -> function ID(<foo_arg>) : <types> <st-list> end <program>
2. <program> -> global ID : function(<types>) : <types> <program>
3. <program> -> ε
4. <st_list> -> local ID : <type> <local_define> <st-list>
5. <st_list> -> if <expression> then <st_list> else <st_list> end <st_list>
6. <st_list> -> while <expression> do <st_list> end <st_list>
7. <st_list> -> write(<foo_call_arg>) <st-list>
8. <st_list> -> ID <id_operations> <st-list>
9. <st_list> -> return <ret_value> <st-list>
10. <st_list> -> ε
11. <next_id> -> ,ID <next-id>
12. <next_id> -> = <value> <second_value>
13. <value> -> reads(<foo_call_arg>)
14. <value> -> readi(<foo_call_arg>)
15. <value> -> readn(<foo_call_arg>)
16. <value> -> tointeger(<foo_call_arg>)
17. <value> -> substr (<foo_call_arg>)
18. <value> -> ord(<foo_call_arg>)
19. <value> -> chr(<foo_call_arg>)
20. <value> -> ID<value_operations>
21. <value> -> <expression>
22. <foo_arg> -> ID : <type> <foo_arg_next>
23. <foo_arg> -> ε
24. <foo_arg_next> -> ,ID : <type> <foo_arg_next>
25. <foo_arg_next> -> ε
26. <foo_call_arg> -> <var_types> <foo_call_arg_next>
27. <foo_call_arg> -> ε
28. <foo_call_arg_next> -> ,<var_types> <foo_call_arg_next>
29. <foo_call_arg_next> ->ε
30. <ret_value> -> <expression>
31. <ret_value> -> ε
32. <types> -> <type> <next_types>
33. <types> -> ε
34. <next_types> ->, <type> <next_types>
35. <next_types> -> ε
36. <type> -> integer
37. <type> -> number
38. <type> -> string
39. <local_define> -> = <value>
40. <local_define> -> ε
41. <id_operations> -> <next_id>
42. <id_operations> -> (<foo_call_arg>)
43. <var_types> -> <expression>
44. <var_types> -> ID
45. <var_types> -> ε
46. <next_value> ->, <var_types> <next_value>
47. <value_operations> -> (<foo_call_arg>)
48. <value_operations> -> ε

Tabulka 2: LL – gramatika řídící syntaktickou analýzu

C LL – tabulka

	function	global	local	if	while	write	ID	return	,	=	(	reads	readi	readn	tointeger	substr	ord	chr	integer	number	string	expr	ε
<program>	1	2	3																				
<st_list>			4	5	6	7	8	9															10
<next_id>									11	12													
<value>							20					13	14	15	16	17	18	19				21	
<foo_arg>							22																23
<foo_arg_next>									24														25
<foo_call_arg>							26															26	26
<foo_call_arg_next>									28														29
<ret_value>																						30	31
<types>																			32	32	32		33
<next_types>									34														35
<type>																			36	37	38		
<local_define>										39													40
<id_operations>									41	41	42												
<var_types>							44															43	45
<next_value>									46														
<value_operations>											47												48

Tabulka 3: LL – tabulka použitá při syntaktické analýze

D Precedenční tabulka

	+-	/*//	<	<=	>	>=	==	~=	#	..	(	)	i	nil	$
+-	>	<	>	>	>	>	>	>	<	>	<	>	<	X	>
/*//	>	>	>	>	>	>	>	>	<	>	<	>	<	X	>
<	<	<	X	X	X	X	X	X	<	<	<	>	<	<	>
<=	<	<	X	X	X	X	X	X	<	<	<	>	<	<	>
>	<	<	X	X	X	X	X	X	<	<	<	>	<	<	>
>=	<	<	X	X	X	X	X	X	<	<	<	>	<	<	>
==	<	<	X	X	X	X	X	X	<	<	<	>	<	<	>
~=	<	<	X	X	X	X	X	X	<	<	<	>	<	<	>
#	>	>	>	>	>	>	>	>	X	>	<	X	<	X	>
..	<	<	>	>	>	>	>	>	<	<	<	>	<	X	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	X	X
)	>	>	>	>	>	>	>	>	>	>	X	>	X	X	>
i	>	>	>	>	>	>	>	>	>	>	X	>	X	X	>
nil	X	X	>	>	>	>	>	>	X	X	X	X	X	X	X
$	<	<	<	<	<	<	<	<	<	<	<	X	<	<	X

Tabulka 4: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů