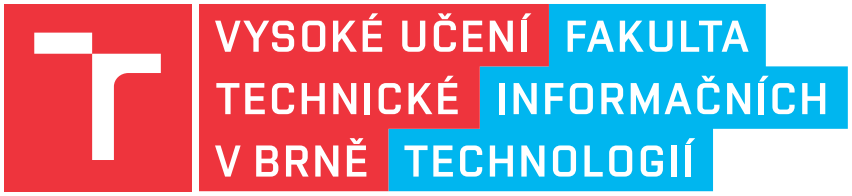


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ22

Tým xtiemn00, varianta TRP

8. prosince 2022

Vsevolod Tiemnohorov	**(xtiemn00)**	25 %
Denys Petrovskyi	(xpetro27)	25 %
Illia Baturov	(xbatur00)	25 %
Sviatoslav Shishnev	(xshish02)	25 %

Obsah

1 Úvod

Cílem tohoto projektu bylo vytvořit program v jazyce C, který funguje jako překladač zdrojového jazyka IFJ22 do cílového jazyka (mezikódu) IFJcode22. Náš program čte zdrojový kód ze standartního vstupu, zapsaný ve zdrojovém jazyce a generuje mezikód na standartní výstup (stdout). Pokud došlo k nějaké chybě, program vrací odpovídající chybový kód.

2 Návrh a implementace

Projekt se skládá z několika částí, které se vzájemně ovlivňují. Náš kompilátor se skládá z lexikální analýzy, syntatické analýzy, sémantické analýzy a generování mezikódu. Každá část a způsob jejich implementace je popsán níže.

2.1 Lexikální analýza

Lexikální analýza je úvodní částí projektu. Hlavní soubor představující tuto část je `scanner.c`. Funkce `get_next_token` postupně čte každý znak ze vstupu a převádí posloupnost znaků do struktury token, což je typ tokenu a data (textová data, hexadecimální data nebo celočíselná data), jak je uvedeno v zadání projektu. Typy tokenů jsou zapsány v hlavičkovém souboru `scanner.h`. Tato funkce vrací nulu při úspěchu, jinak vrátí odpovídající chybový kód. Tyto tokeny později použijeme v následujících částech projektu.

Lexikální analyzátor byl implementován pomocí konečného automatu, který je znázorněn na obrázku číslo 1. Každý `switch case` v `get_next_token` představuje jednotlivý stav automatu.

2.2 Syntaktická analýza

Nejsložitější a proto nejdůležitější částí projektu je syntaktická analýza.

Syntaktická analýza využívá LL-gramatiku (tabulka číslo 2) a metodu rekurzivního sestupu podle pravidel LL-tabulky (tabulka číslo 3). Každé pravidlo má svou příslušnou funkci. Syntaktická analýza vyžaduje tokeny z lexikální analýzy tím, že zavolá je v funkci `main` pomocí `get_next_token`. Současně probíhá i lexikální analýza.

2.3 Sémantická analýza

Pro sémantickou analýzu používáme tabulky symbolů. Lokální tabulky jsou potřeba pro lokální proměnné, globální tabulky jsou potřeba pro funkce. Tabulky symbolů jsou hashovací tabulkou a používají se ke kontrole, zda identifikátor existuje a zda se jeho datový typ shoduje.

2.4 Generování mezikódu

Generování kódu je poslední částí projektu. Tato část je reprezentována souborem `code_generator.c` a hlavičkovým souborem `code_generator.h`. Nejprve vypíšeme vestavěné funkce na výstup, pak vezmeme tříadresový kód z dvousměrně vázaného lineárního seznamu, který je tam po projití všech fází analýzy zdrojového kódu. Potom projdeme tento seznam a na základě tříadresového kódu zobrazíme cílový kód IFJcode22.

3 Implementované metody překladače

Při realizaci projektu jsme použili několik speciálních datových struktur. Tyto struktury jsou popsány níže.

3.1 Hašovací tabulka

K implementaci tabulek symbolů byla použita struktura hašovací tabulky. Tuto hašovací tabulku jsme převzali z druhého projektu předmětu IAL jednoho z členů týmu.

3.2 Dvousměrně vázaný seznam

Dvousměrně vázaný seznam je použitý pro uložení tříadresového kódu. Seznam pak použije generátor kódu k vygenerování příslušného kódu. Tento dvousměrně vázaný seznam jsme převzali z prvního projektu předmětu IAL jednoho z členů týmu.

3.3 Dynamický řetězec

Při realizaci projektu jsme často využívali strukturu dynamického řetězce, kterou jsme převzali z obecných informací pro tvorbu překladače ze stránek předmětu IFJ.

Tato struktura je z archivu `jednoduchy_interpret.zip`. Použili jsme soubory `str.c` a `str.h`.

3.4 Zásobník symbolů

Inspirovali jsme se obsahem osmé sady přednášek. Ukazuje, jak správně používat zásobník symbolů v kombinaci s prediktivní analýzou pomocí precedenční tabulky (tabulka číslo 4). Zásobník jako datovou strukturu jsme procvičovali v předmětu IAL. K vytvoření precedenční tabulky jsme také použili materiály přednášek.

4 Práce v týmu

Náš způsob týmové práce je popsán níže.

4.1 Verzovací systém a komunikace

Na projektu jsme začali pracovat začátkem listopadu.

Použili jsme GitHub jako vzdálený repositář. Díky tomu jsme byli schopni sdílet kód mezi sebou.

Každý sám navrhoval, na které části bude pracovat, a tak jsme se dohodli na plánu práce. Každý z nás si pomáhal, když měl potíže, často jsme kvůli tomu pořádali společná setkání pro více lidí nebo celý tým. Jednalo se o online setkání v Discordu nebo osobní setkání ve studovnách.

4.2 Rozdělení práce

Práci na projektu jsme si rozdělili rovnoměrně, podle náročnosti a času práce na vlastní části. Tabulka 1 shrnuje rozdělení práce v týmu mezi jednotlivými členy.

Člen týmu	Přidělená práce
Vsevolod Tiemnohorov	vedení týmu, syntaktická analýza, sémantická analýza, testování
Denys Petrovskyi	lexikální analýza, generování cílového kódu, testování
Illia Baturov	generování cílového kódu, dokumentace, testování
Sviatoslav Shishnev	konzultace, syntaktická analýza, sémantická analýza, testování

Tabulka 1: Rozdělení práce v týmu

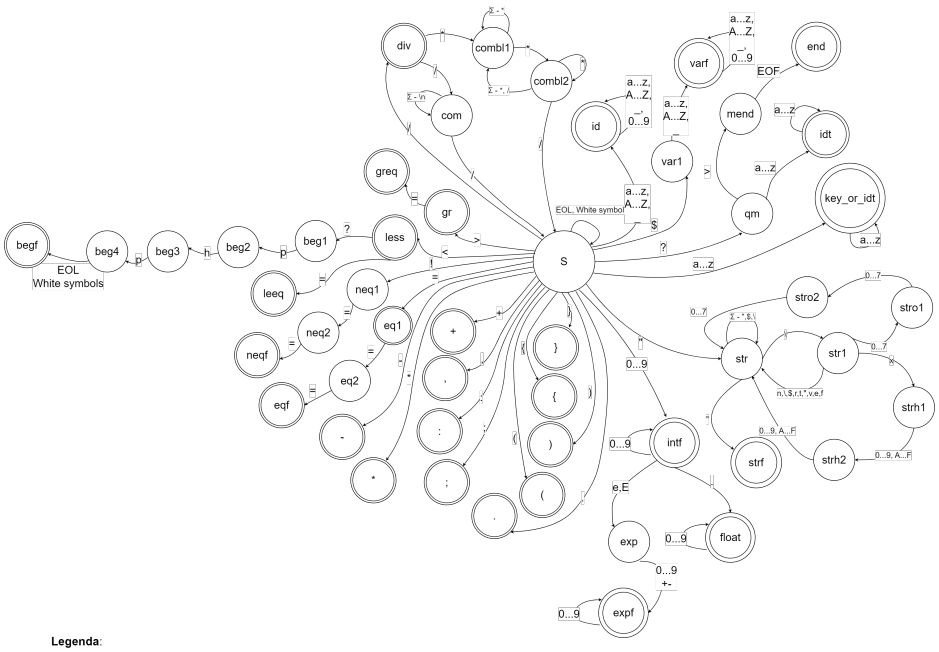
5 Závěr

Před sestavením týmu jsme se už znali, takže jsme neměli problémy s komunikací. O projektu jsme diskutovali online a během skupinových schůzek, velmi rychle jsme se dohodli na použití konkrétního verzovacího systému. To znamená, že práce v týmu dopadla docela dobře.

Když jsme začali dělat projekt, měli jsme problémy s organizací a pochopením zadaní projektu kvůli jeho velikosti. Při vývoji projektu a studiu školicích materiálů, včetně přednášek, jsme začali lépe chápat mechanismy překladače. Největší problém nám dělala organizace práce na projektu, kvůli které jsme to často odkládali na později, ale nakonec jsme na to přišli.

Tento projekt nám vysvětlil principy překladače a získali jsme zkušenosti s týmovou prací na velkém projektu.

A Diagram konečného automatu



Legenda:

S	START	eq1	EQUAL1	begf	BEGINNING_FINALE	key_or_idt	KEY_OR_IDENTIFIER
+	ADD	eq2	EQUAL2	com	COMMENT	str	STRING
-	SUB	eqf	EQUAL_FINALE	div	DIVISION	str1	STRING1
*	MUL	neq1	NOT_EQUAL1	combl1	COMMENT_BLOCK1	stro1	STRING_OCTAL1
,	COMMA	neq2	NOT_EQUAL2	combl2	COMMENT_BLOCK2	stro2	STRING_OCTAL2
:	COLON	neqf	NOT_EQUAL_FINALE	id	IDENTIFIER	strh1	STRING_HEXADECIMAL1
;	SEMICOLON	less	LESS	var1	VARIABLE1	strh2	STRING_HEXADECIMAL2
.	DOT	leeq	LESS_OR_EQUAL	varf	VARIABLE_FINALE	strf	STRING_FINALE
(	PAR_LEFT	beg1	BEGINNING1	qm	QUESTION_MARK	intf	INT_FLOAT
)	PAR_RIGHT	beg2	BEGINNING2	mend	MAYBE_END	float	FLOAT
{	BRACE_LEFT	beg3	BEGINNING3	end	END	exp	EXPONENTIAL
}	BRACE_RIGHT	beg4	BEGINNING4	idt	IDENTIFIER_TYPE	expf	EXPONENTIAL_FINALE

Obrázek 1: Diagram konečného automatu

B LL – gramatika

#	Levá strana	Pravá strana	Predict
1	<prog>	begin <st-list>	{ begin }
2	<st-list>	<state> ; <st-list>	{ return, var_id, id, lit }
2.5	<st-list>	<f-list> } <st-list>	{ function, if, else, while }
3	<st-list>	end	{ end }
3.5	<st-list>	$	{ $ }
4	<state>	function id <define> : type_id { <f_list>	{ function }
5	<state>	return <expr>	{ return }
6	<state>	var_id <exoras>	{ var_id }
7	<state>	<expr>	{ var_id lit }
7.5	<state>	if (<expr>) { <st-list>	{ if }
8	<state>	else { <st-list>	{ else }
9	<state>	while (<expr>) { <st-list>	{ while }
10	<state>	id <declare>	{ id }
11	<define>	(<f_p-list>	{ (}
12	<declare>	(<f_plist_declare>	{ (}
13	<f_p-list>	, <f_param> <f_p-list>	{ , }
14	<f_p-list>	<f_param> <f_p-list>	{ type_id }
15	<f_p-list>	)	{) }
16	<f_param>	type_id var_id	{ type_id }
17	<f_plist_declare>	, <f_param_declare> <f_plist_declare>	{ , }
18	<f_plist_declare>	<f_param_declare> <f_plist_declare>	{ var_id, literlas, null}
19	<f_plist_declare>	)	{) }
20	<f_param_declare>	literals	{ literals }
21	<f_param_declare>	var_id	{ var_id }
22	<f_param_declare>	null	{ null }
23	<f-list>	<f_state> ; <f-list>	{ return, var_id, id }
24	<f-list>	<f_state> } <f-list>	{ if, else, while }
25	<f-list>	ε	{ } }
26	<f_state>	return <expr>	{ return }
27	<f_state>	var_id <exoras>	{ var_id }
28	<f_state>	var_id <exoras>	{ var_id lit }
29	<f_state>	if (<expr>) { <f_list>	{ if }
30	<f_state>	else { <f_list>	{ else }
31	<f_state>	while (<expr>) { <f_list>	{ while }
32	<f_state>	id <declare>	{ id }
33	<exoras>	eq <expr>	{ eq }
34	<expr>	id <declare>	{ id }
35	<expr>	<bottomtotop>	{ lit, var_id (}

Tabulka 2: LL – gramatika

C LL – tabulka

	begin	return	else	id	if	var_id	function	while	end	$	(	,	type_id	)	}	eq	lit	operator	null
<prog>	1																		
<st-list>		2	2.5	2	2.5	2	2.5	2.5	3	3.5							2		
<state>		5	8	10	7.5	6	4	9									7		
<declare>											12								
<define>											11								
<f_param>													16						
<f_p-list>												13	14	15					
<f_plist_declare>						18						17		19			18		18
<f-list>		23	24	23	24	23		24	24	3.5					25				
<f-state>		26	30	32	29	27		31									28		
<exoras>																33			
<expr>				34		35					35						35		
<f_param_declare>						21											20		22

Tabulka 3: LL – tabulka

D Precedenční tabulka

	*	/	+	-	.	<	>	<=	>=	===	!==	(	)	id	$
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
.	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
===	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
!==	<	<	<	<	<	<	<	<	<	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	E
)	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>
id	>	>	>	>	>	>	>	>	>	>	>	E	>	E	>
$	<	<	<	<	<	<	<	<	<	<	<	<	E	<	E

Tabulka 4: Precedenční tabulka