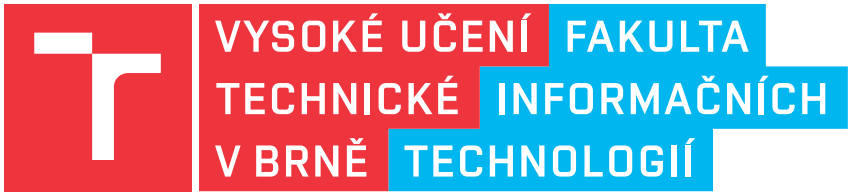


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ21

Tým 133, varianta I

Kravchuk Marina	(xkravc02)	30 %
Narush Alexey	(xnarus00)	30 %
Sartin Andrei	(xsarti00)	20 %
Tiurin Daniil	(xtiuri02)	20 %

8. prosince 2021

Obsah

1 Úvod

Výpracovaný projekt načitá zdrojový kod zapsaný ve zdrojovém jazyce IFJ21 ze standardního vstupu a generuje výsledný mezikod v jazyce IFJcode21 na standardní výstup nebo vrací odpovídající kod v případě chyby. Tato dokumentace popisuje návrh, implementace, způsob práce v týmu.

2 Návrh a implementace

Projekt jsme sestavili z několika námi implementovaných dílčích částí, které jsou představeny v této kapitole. Je zde také uvedeno, jakým způsobem spolu jednotlivé dílčí části spolupracují.

2.1 Lexikální analýza

Lexikální analýza je implementovaná ve zdrojovém souboru **`scanner.c`**. Ve **`scanner.h`** se nachází prototypy funkcí a enumerace typů tokenů. Analýza je implementována pomocí konečného automatu, struktury **`string`** pro uložení data a int hodnoty na vraceni typu tokenu (například, že dostaneme "#", vrací 'HASH'). Během lexikální analýzy pomocí funkce **`getNextToken`** se ze vstupního souboru cyklické načitá symbol po symbolů, dokud nenačte mezeru nebo logický konec tokenu.

Dokud nedosáhneme konce souboru, ověřuje se spravnost vstupních symbolů a lexikálně se analyzuje získáné tokeny. Pracujeme s tokeny pomocí konstrukce **`switch-case`** a předem nastavených předpokládaných typu tokenů. Prečítáme znaky a případně zvětšíme velikost alokované struktury **`string`**.

2.2 Syntaktická analýza

Nejdůležitější a hlavní část projektu je syntaktická anakýza. Syntaktická analýza je realizovaná iterační metodou s využitím cyklu **`while`**. Program začiná s alokaci paměti pro structuru **`Symtable`** i pro ostatní dynamické datové struktury a volá **`parser`**, který pomocí funkce **`tryGetToken`** využivá **`scanner`** a s jeho pomocí čte soubor a obdrží další token. Pak **`parser`** provádí syntaktickou analýzu podle pravidel, které jsou popsaný na poslední stránce Když se program setká s výrazem, zavolá funkci **`express`** z **`expression.c`**, která analyzuje výraz z hlediska správnosti pomocí precedenčních tabulek. Ten je kontrolován, dokud program nenarazí na konec výrazu, a poté se vrátí do **`parseru`**, kde pokračuje celková analýza programu. Pokud je nalezena chyba, překladač dokončí kontrolu, vymaže alokovanou paměť a zobrazí chybový kód na standardním chybovém výstupu. Funkce pro syntaktickou a symantickou jsou implementovány v souboru **`parser.c`**

2.2.1 Zpracování výrazů

priorita	operatory	asociativita
0	#	unární
1	/ * //	levá
2	+ -	levá
3	..	pravá
4	<<= >>= == ~=	levá

Když **`parser`** načte výraz, předá token funkci **`express`**, která následně vytvoří zásobník a přidá do něj token. Poté je zavolána funkce parseru **`tryGetToken`** a nový token je zkontrolován pomocí precedenční tabulky a poté je přidán do zásobníku, v případě chyby je zavolána funkce **`changeError`**, která ukončí program a vrátí kód chyby.

Při kontrole výrazu se volá funkce **EXPRESSION_FUNC**, která odešle zpracovávanou část výrazu a přeloží ji. do IFJ CODE21. V případě, že po zavolání této funkce není výraz ukončen, je zavolán nový token a pokračuje se v kontrole pomocí precedenční tabulky. Když program dosáhne konce výrazu, paměť alokovaná pro zásobník se vymaže a program se vrátí k překladači, aby pokračoval v syntaktické a sémantické kontrole.

	+ -	// * /	(	)	i	$	= ∼=	<<= >>=	str	#	..	nil
+ -	>	>	<	>	<	>	>	>	#	<	#	#
// * /	>	>	<	>	<	>	>	>	#	<	#	#
(	<	<	<	=	<	#	<	<	<	<	<	#
)	>	>	#	>	#	>	>	>	#	#	#	#
i	>	>	#	>	#	>	>	>	#	#	#	#
$	<	<	<	#	<	#	<	<	<	<	<	#
= ∼=	<	<	<	>	<	>	#	#	<	<	#	<
<<= >>=	<	<	<	>	<	>	#	#	<	<	#	#
str	>	#	#	>	#	>	>	>	#	#	>	#
#	<	<	<	>	#	>	>	>	<	#	#	#
..	#	#	#	>	#	>	>	>	<	#	>	#
nil	#	#	#	>	#	>	>	#	#	#	#	#

2.3 Sémantická analýza

Při běhu program kontroluje jména funkcí, jejích parametry a dodává informaci o nich do **tabulky symbolů**. Když kompilator uvidí identifikátor, pro sěmantickou analýzu bude znamenat že se jedna o proměnnou nebo název funkce, a v záležitosti od místa v programu ta proměnná bude nebo přidaná do tabulky symbolů nebo bude zkontrolováno, jestli ta proměnná už existuje. Největši problém, se kterém jsme se setkali v sémantické analýze, bylo použití proměnných uvnitř různých bloků bud' na různých úrovních nebo na stejném úrovni. Rešením bylo využiti proměnnou pro hloubku **deep** a seznam binárních stromu pro vyhledávaní proměnných. S její pomoci každá deklarace proměnné v nověm bloku vytvářela nový binární strom na nové hloubce (jeden strom pro každou hloubku). A každý konec bloku znamenal, že proměnné na tomto úrovni už se nevyužijí a celý ten strom je odstraněn. Proměnné na 0 úrovni budou odstraněny když tělo funkce skončí. Vstupní parametry byly přeneseny do tabulky symbolů jako proměnné 0 úrovně. Funkci budou odstraněny z tabulky symbolů už před ukončením programu. Ještě jedna kontrola probíhá když musíme srovnávat jednu nebo několik proměnné s návratovými hodnotami funkce anebo z nějakým počtem výrazů. V těch případech využíváme strukturu **seznam** a tabulku symbolů pro shodu typů. Taky vždycky se kontrolují typy výstupních parametrů funkci a co ta funkce vrací.

3 Speciální algoritmy a datové struktury

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

3.1 Binární vyhledávací strom.

Vybrali jsme variantu projektu s abstraktní dátovou strukturou **binární vyhledávací strom**. Na základě toho implementováce tabulky symbolů byla udělana jako binární vyhledávací strom. Operace nad binárním stromem jsme prováděli **rekurzivně**. Ve struktuře **Symtable** máme 2 struktury pro různé druhy tabulek: tabulka symbolů pro funkce **func_tree** a tabulka symbolů pro proměnné **var_tree**.

Při vytváření tabulky symbolů pro proměnné jsme uvídělí jeden problém. Proměnné může mít stejné názvy, ale se nacházet na různých urovních. Abychom jsme to mohli rozlišovat, vytvařili jsme nový binarný strom na

každé urovní. Tím pádem jsme získalilineární seznam binárních stromů. Každý uzel stromu obsahuje identifikátor, ukazatele na jeho dva podstromy a data. V binárních stromech hledáme pomocí jmena proměnne. Implementovali jsme několik funkcí pro práci s tabulkou. To jsou následující funkce: inicializace, přidání nové položky, přidání vstupních a výstupních argumentů (pro funkci), vyhledání položky, uvolnění tabulky z paměti.

4 Generování cílového kodu

Funkce pro generování kódu se nachází v souboru **`interpreter.c`**, a je volána v souborech **`parser.c`** a **`expression.c`** během syntaktické a sémantické analýzy. Pro práci s generátorem kódu používáme tabulku symbolů. Nejprve se vygeneruje hlavička programu ".IFJCODE21"a zavolá se funkce "main". Vestavěné funkce jsou pak vypsány. Poté se spustí vlastní generování programu. Když kompilátor narazí na deklaraci funkce, zavolá funkci **`GEN_START_OF_FUNCTION`** ze souboru **`interpeter.c`**, která vytvoří hlavičku funkce. Při výskytu definice proměnné se zavolá funkce **`GEN_DEFVAR_VAR`**. Tyto funkce jsou příkladem implementace překladu kódu z IFJ21 do IFJCODE21. Při psaní těchto funkcí se vyskytl problém s předefinováním. proměnné pro výrazy. Řešením tohoto problému bylo přesunout tyto proměnné do GF. Kromě toho se vyskytla komplikace se zápisem hodnoty **`nil`**. Aby se tomuto problému opět předešlo, byla změněna vestavěná funkce **`write`**. Pokud je v tomto případě hodnota proměnné **`nil`**, změní se typ proměnné z **`nil@nil`** na **`string@nil`**. Tím je zajištěno, že se hodnota **`nil`** zapíše správně a že proměnné s touto hodnotou neztratí své vlastnosti. Funkce **`EXPRESSION_FUNC`** slouží k převodu výrazů na cílový kód. Argumenty jsou této funkci předávány po jejich stažení ze zásobníku v souboru **`expression.c`**. Poté, co tato funkce obdrží 2 operandy a 1 operátor, je pomocí **`switch-case`** určen typ operandu a příslušný kód je vypsán ve formátu IFJCODE21.

5 Práce v týmu

5.1 Způsob práce v týmu

Na projektu jsme začali pracovat koncem října. Práci jsme si dělili postupně, tj. neměli jsme od začátku stanovený kompletní plán rozdělení práce. Na dílčích částech projektu pracovali většinou jednotlivci nebo dvojice členů týmu. Nejprve jsme si stanovili strukturu projektu a vytvořili překladový systém.

5.1.1 Verzovací systém

Pro správu souborů projektu jsme používali verzovací systém **`Git`**. Jako vzdálený repositář jsme používali **`GitHub`**.

Git nám umožnil pracovat na více úkolech na projektu současně v tzv. větvích. Většinu úkolů jsme nejdříve připravili do větve a až po otestování a schválení úprav ostatními členy týmu jsme tyto úpravy začlenili do hlavní vývojové větve.

5.2 Rozdělení práce mezi členy týmu

Práci na projektu jsme si rozdělili rovnoměrně s ohledem na její složitost a časovou náročnost.

Tabulka 1 shrnuje rozdělení práce v týmu mezi jednotlivými členy.

Člen týmu	Přidělená práce
Kravchuk Marina	vedení týmu, organizace práce, dohlížení na provádění práce, lexikální analýza, syntaktická analýza, semantická analýza, generování cílového kodu, testování, dokumentace
Narush Alexey	lexikální analýza, syntaktická analýza, sémantická analýza, generování cílového kodu, testování, dokumentace
Sartin Andrei	lexikální analýza, generování cílového kodu, dokumentace
Tiurin Danil	generování cílového kodu, dokumentace

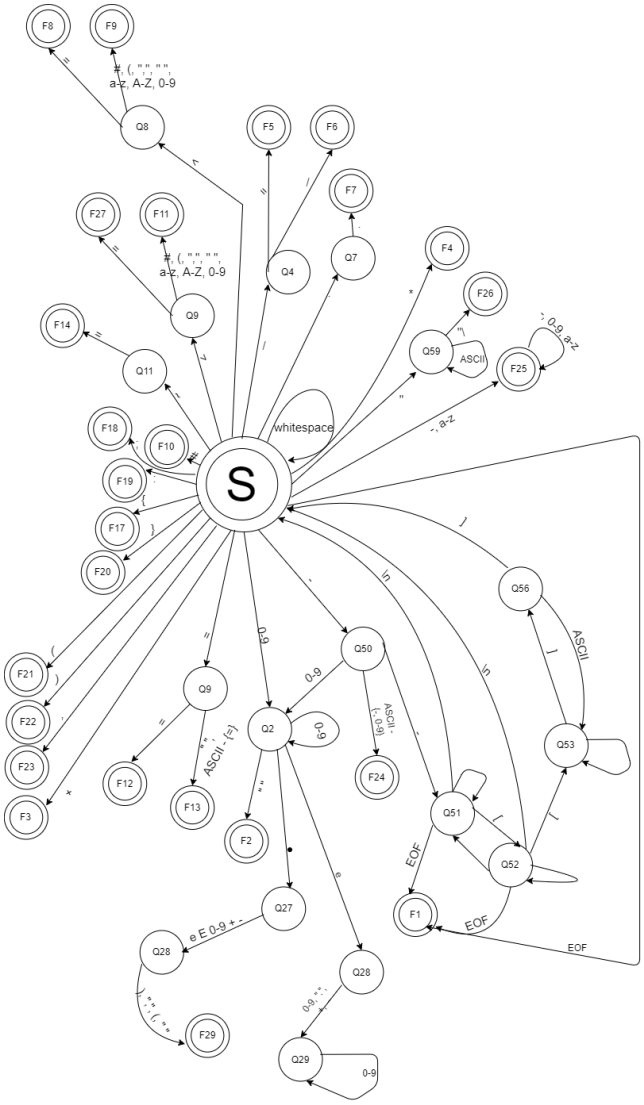
Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

6 Závěr

Reference

[1] Slajdy z přednášek předmětu Formální jazyky a překladače
[2] Slajdy z přednášek předmětu Algotitmy

A Diagram konečného automatu specifikující lexikální analyzátor



Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

LEGEND: S - START_STATE
- F1 - END_OF_FILE
- F2 - INT
- F3 - INC
- F4 - MULTIPLY
- F5 - DIV
- F6 - MOD
- F7 - DOTDOT
- F8 - LESSOREQUAL
- F9 - LESS
- F10 - HASH
- F11 - MORE
- F12 - EQUAL
- F13 - ASSIGNED
- F14 - NOTEQUAL

F15 - ID
F16 - FLOAT
F17 - LEFT_VINCULUM
F18 - SEMICOLON
F19 - COLUMN
F20 - RIGHT_VINCULUM
F21 - LEFT_BRACKET
F22 - RIGHT_BRACKET
F23 - COMMA
F24 - DEC
F25 - IDENTIFICATOR
F26 - RETEZEC
F27 - MOREOREQUAL

B LL – gramatika

	EXPR	require	"IFJ21"	local	id	EOF	return	function	(	)	ε	,	:	if	else	then	while	do	end	"="	integer	num	str	nil
<PROGRAM_START>		1																						
<FUNC_DEF>								2																
<FUNC_DEF_NEXT>								3																
<PARAM>					4																			
<PARAM_NEXT>											5	6												
<TYPE>																					7	8	9	10
<TYPE_NEXT>											12										11	11	11	11
<OUT_PARAM>											14		13											
<DEF_ID>					15																			
<DEF_ID_NEXT>											16	17												
<DEF_VAR>				18																				
<ASSIGNED>											19									20				
<VARS>	22										21													
<STATEMENT>				28	23		26				27			24			25							

1. <PROGRAM_START> -> Require ”ifj21” <FUNC_DEF> <FUNC_DEF_NEXT> EOF
2. <FUNC_DEF> -><PARAM ><OUT_PARAM> <STATEMENT> end
3. <FUNC_DEF_NEXT> -> <FUNC_DEF> <FUNC_DEF_NEXT>
4. <PARAM> -> id : <TYPE> <PARAM_NEXT>
5. <PARAM_NEXT> -> ε
6. <PARAM_NEXT> -> , <PARAM>
7. <TYPE> -> integer
8. <TYPE> -> number
9. <TYPE> -> string
10. <TYPE> -> nil
11. <TYPE_NEXT> -> <TYPE> , <TYPE_NEXT>
12. <TYPE_NEXT> -> ε
13. <OUT_PARAM> -> : <TYPE> <TYPE_NEXT>
14. <OUT_PARAM> -> ε
15. <DEF_ID> -> id <ID_NEXT> = EXPR EXPRE_NEXT
16. <ID_NEXT> -> ε
17. <ID_NEXT> -> , id <ID_NEXT>
18. <DEFVAR> -> local id <ID_NEXT> : <TYPE> <ASSIGNED>
19. <ASSIGNED> -> ε
20. <ASSIGNED> -> = EXPR EXPR_NEXT
21. <VARS> -> ε
22. <VARS> -> EXPR EXPR_NEXT

23. < STATEMENT > -> < DEFID > < STATEMENT >
24.< STATEMENT > -> if EXPR then < STATEMENT > else < STATEMENT > end
25. <STATEMENT> -> while EXPR do <STATEMENT> end
26. <STATEMENT> -> return <VARS> <STATEMENT>
27. <STATEMENT> -> ε
28. <STATEMENT> -> <DEFVAR>