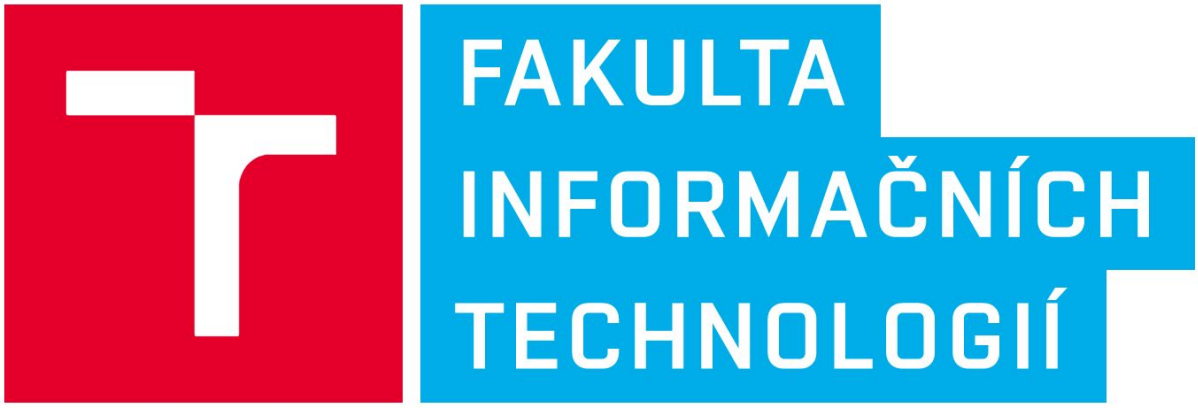


Dokumentace k projektu

Implementace překladače imperativního jazyka IFJ20

varianta: II tým: 094

V Brně dne **9. 12. 2020**

Vedoucí: Jakub Vaňo (xvanoj00)	**25%**
Ondřej Šebesta (xsebes22)	25%
Jiří Václavič (xvacla31)	25%
Zuzana Hrkľová (xhrklo00)	25%

Obsah

1. Popis projektu

Cílem projektu byla implementace překladače imperativního jazyka IFJ20 v jazyce C. Jazyk IFJ20 je podmnožinou staticky typovaného, imperativního jazyka GO. Program zkontroluje potencionální lexikální, syntaktické a sémantické chyby kódu v jazyce IFJ20 a ten následně přeloží do mezikódu IFJcode20.

Potřebnou teorii k problematice jsme získali z přednášek předmětu IAL, dále z přednášek a demonstračních cvičení předmětu IFJ a v neposlední řadě také z *knihy prof. Meduny [1]*.

2. Lexikální analýza

Lexikální analýzu jsme implementovali jako první ze všech částí projektu. Formálním modelem pro lexikální analýzu je konečný automat, který naleznete v **příloze 1**. Při navrhování lexikální analýzy jsme vycházeli primárně ze zadání. Pouze u kontroly odřádkování jsme se rozhodli pro variantu s kontrolou odřádkování v rámci syntaktické analýzy - tedy lexikální analyzátor propaguje odřádkování do syntaktické analýzy jako samostatný token a nijak jej dále nezkoumá.

Lexikální analýza je implementována v souborech `scanner.c` a `scanner.h`. Analýzu jsme nejprve implementovali jako obyčejný konečný automat. Kód se nám však nezdál příliš dobře čitelný, a proto jsme jednotlivé části implementovali do funkcí, které v sobě zahrnují i více stavů formálního modelu. Všechny funkce jsou řízeny a volány v hlavní funkci `get_next_token`, jejíž volání řídí syntaktická analýza.

Token je implementován jako struktura obsahující datový typ `token_type`, atribut `token_attribute` a pomocný příznak `returned`. Tuto strukturu předává lexikální analýza syntaktické analýze, která s ní dále pracuje.

3. Syntaktická analýza

Syntaktická analýza je dle zadání rozdělena na dvě části - na analýzu rekurzivním sestupem a precedenční analýzu. Formálním modelem pro analýzu rekurzivním sestupem je bezkontextová gramatika viz **Příloha 2**, potažmo LL tabulka viz **Příloha 3**. Správnost LL gramatiky jsme ověřili pomocí nástroje *DIDEFOM [2]*. Pro precedenční analýzu je formálním modelem precedenční tabulka viz **Příloha 4.**

3. 1 Rekurzivní sestup

Na základě LL gramatiky a LL tabulky jsme implementovali analýzu rekurzivním sestupem. Pro každý neterminál jsme vytvořili odpovídající funkci. V každé takové funkci jsou pak kontrolována odpovídající pravidla. Tato analýza je implementována v souborech `parser.c` a `parser.h`. Pokud vyhodnotíme token jako výraz, je volána precedenční analýza.

3. 2 Precedenční analýza

Precedenční analýza je implementována v souborech `expressions.c` a `expressions.h`. V těchto souborech se také nachází implementace zásobníku používaného pro zpracování výrazů. Analýza postupně redukuje výrazy a kontroluje správnou posloupnost symbolů k čemuž používá precedenční tabulku, dále pak provádí sémantické akce a generuje odpovídající kód v jazyce IFJcode20.

4. Sémantická analýza

Sémantickou analýzu řešíme formou vkládání sémantických akcí do syntaktické analýzy. Na počátku běhu syntaktické analýzy vkládáme do globální tabulky symbolů vestavěné funkce, počet jejich parametrů, počet návratových hodnot a poté i datové typy jednotlivých parametrů / návratových hodnot. Jakmile je funkce v kódu IFJ20 volána, provedeme kontrolu a případně hlásíme sémantickou chybu. Na tomto principu fungují také uživatelské funkce, ty ale vkládáme do globální tabulky symbolů až při jejich definici.

Během precedenční analýzy kontrolujeme typovou kompatibilitu operandů.

Po provedení syntaktické analýzy dodatečně kontrolujeme, zda byla funkce `main` definována a zda její definice neobsahuje žádný parametr ani návratovou hodnotu.

5. Tabulka symbolů

Tabulku symbolů jsme implementovali pomocí tabulky s rozptýlenými položkami. Tuto variantu tabulky symbolů jsme si ze seznamu zadání vybrali cíleně, protože ji považujeme za nejvhodnější. Maximální velikost mapovacího pole jsme zvolili jako prvočíslo **27 487** a předpokládáme, že by se kapacita pole neměla naplnit.

Synonyma jsme vyřešili pomocí lineárně vázaného seznamu. Index položky v tabulce symbolů vypočítáváme pomocí **DJB hashovací funkce**. V této funkci používáme konstantu **5381**, která je v tomto typu hashovací funkce ověřená jako číslo způsobující nejméně kolizí. *[3]*.

Každá položka tabulky je uložena ve struktuře `table_item` a obsahuje klíč reprezentovaný jejím názvem, ukazatel na následující položku v lineárním seznamu a strukturu `item_data`, ve které ukládáme data potřebná pro sémantické akce. Pro práci s tabulkou symbolů jsme implementovali pomocné funkce, konkrétně pro inicializaci TS, vkládání do TS, mazání z TS, hledání v TS a uvolnění TS.

Tabulka je implementována v souborech `symtable.c` a `symtable.h`.

6. Generování kódu

Na generování cílového kódu, kterým je v našem případě mezikód IFJcode20 používáme pomocné funkce. Tyto funkce jsou implementovány v souborech `generator.c` a `generator.h`. Funkce jsou volány během syntaktické analýzy a vypisují mezikód na standardní výstup.

Nejprve jsme vyřešili generování vestavěných funkcí, kde každá vestavěná funkce má odpovídající funkci, která vytiskne její reprezentaci v IFJcode20. Dále jsme implementovali funkce, které generují určité části kódu - například `main_start`, `func_start`, `func_call`, `for_start`, `for_end` a podobně.

7. Pomocné struktury a funkce

7. 1 Dynamický řetězec

Předem nevíme, jak dlouhý načteme identifikátor nebo řetězec ze standardního vstupu. Potřebovali jsme proto implementovat pomocné funkce, které nám umožní vytvářet potencionálně nekonečný řetězec. Tyto funkce jsou k nalezení v souborech `string.c` a `string.h`.

Pro práci s dynamickým řetězcem je potřeba jej nejprve *inicializovat*. Tuto úlohu obsluhuje funkce `initialize_string`, která *alokuje* v řetězci paměť pro 2 znaky. Pro vkládání znaků do řetězce je potřeba využít funkci `append_string`, která *realokuje* paměť pro délku řetězce + 1 a vloží požadovaný znak na konec řetězce. Po každé práci s dynamickým řetězcem je vhodné uvolnit paměť - funkcí `free_string`.

7. 2 Návratové hodnoty

Návratové hodnoty jsou uloženy v hlavičkovém souboru `retvals.h`. Tento soubor byl vytvořen pro lepší čitelnost návratových hodnot v kódu. Každá návratová hodnota má definovanou svou konstantu a v komentářích má popsáno základní použití.

8. Práce na projektu

8. 1 Rozdělení úkolů

- **Jakub Vaňo, xvanoj00** - Tabulka symbolů, generování instrukcí, testy
- **Ondřej Šebesta, xsebes22** - Lexikální analýza, SA rekurzivním sestupem, dynamický řetězec, dokumentace
- **Jiří Václavič, xvacla31** - Testovací program, SA zdola-nahoru, sémantická analýza, generování instrukcí
- **Zuzana Hrkľová, xhrklo00** - Tabulka symbolů, generování instrukcí, testy

8. 2 Komunikace

Komunikace probíhala bohužel, jen a pouze, online. Komunikačním kanálem jsme zvolili Discord. V průběhu řešení projektu proběhlo několik hovorů, ale majoritním prostředkem komunikace byl *chat*.

8. 3 Verzování

Pro ukládání a revizi pokroku práce na projektu jsme potřebovali verzovací nástroj. Nejlépe takový, se kterým umí každý člen týmu pracovat. Zvolili jsme proto Git a GitHub.

9. Závěr

Projekt byl náročný. Situace okolo koronaviru nám znemožnila osobní schůzky a museli jsme zvolit online komunikaci. Snažili jsme se projekt dělat již od začátku semestru, což nám zajistilo 1 bonusový bod navíc. S blížícím se odevzdáním IFJ projektu se blížilo také odevzdání spousty dalších projektů, a proto se nám nepodařilo projekt zcela dotáhnout k dokonalosti.

Vyzkoušeli jsme si, jak probíhá vzdálená týmová spolupráce, zažili jsme spousty sporů v GitHub repozitářích a naučili se, jak probíhá překlad z jednoho programovacího jazyka do druhého. Také jsme museli překonat drobnou jazykovou bariéru mezi českým a slovenským jazykem.

10. Zdroje

[1] MEDUNA, Alexander. Elements of compiler design. Boca Raton: Auerbach Publications, 2008. ISBN 1-4200-6323-5.

[2] Didaktické demonstrace modelů pro popis formálních jazyků [online], dostupné z: http://www.fit.vutbr.cz/~meduna/work/doku.php?id=lectures:didefom:start

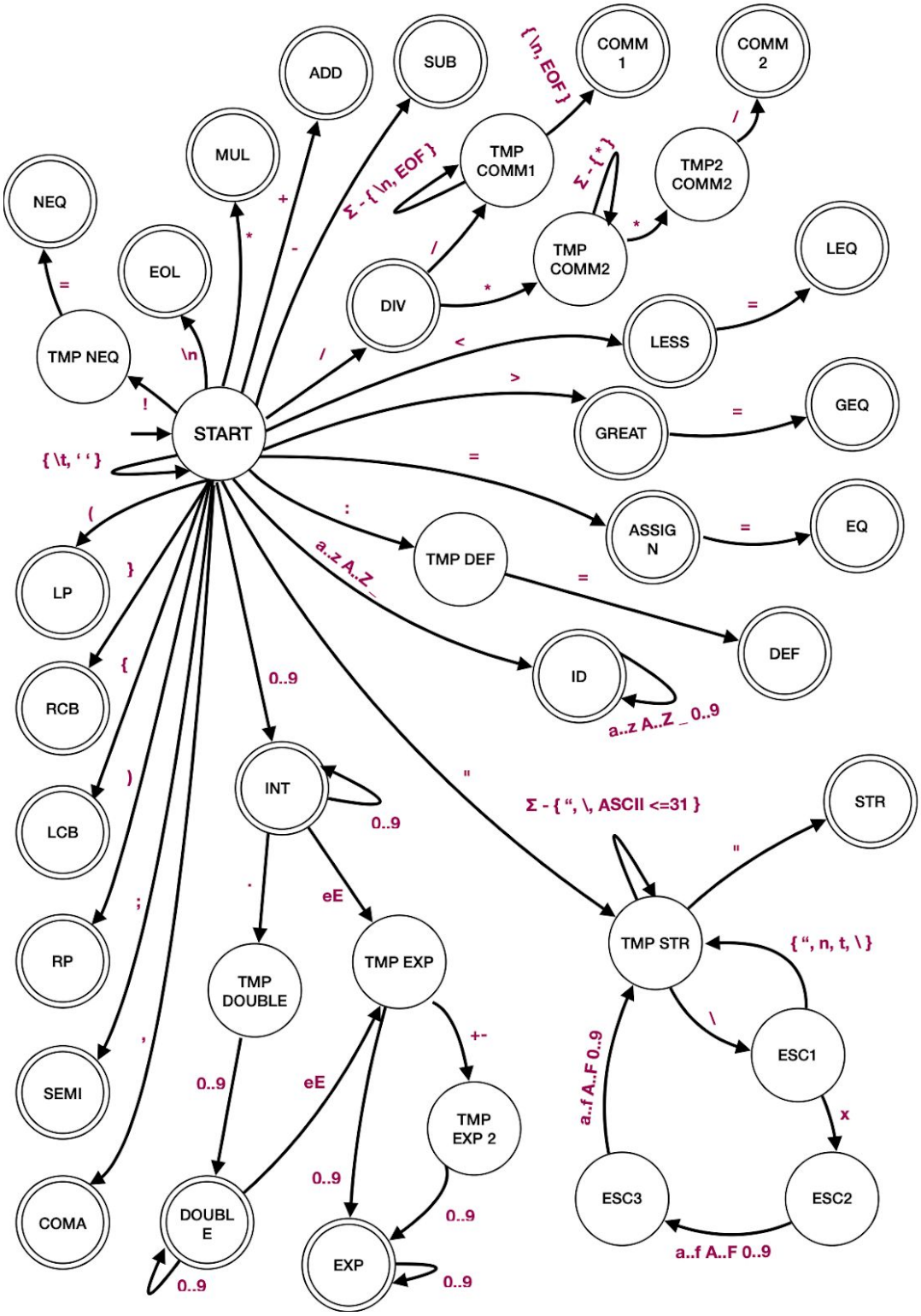
[3] Breaking Daniel J. Bernstein's Algorithm [online], dostupné z: https://medium.com/@khorvath3327/breaking-daniel-j-bernsteins-algorithm-2536e545d9c6

11. Přílohy

Seznam příloh

- Příloha 1 - Konečný automat
- Příloha 2 - Bezkontextová gramatika
- Příloha 3 - LL tabulka
- Příloha 4 - Precedenční tabulka

Příloha 1 - Konečný automat



Příloha 2 - Bezkontextová gramatika

PROGRAM -> OPT_EOL package main eol FUNC_DEFS OPT_EOL eof
FUNC_DEFS -> FUNC_DEF NEXT_FUNC_DEFS
FUNC_DEF -> func id (OPT_EOL FUNC_PARAMS) RET_TYPES { eol COMMANDS } eol
NEXT_FUNC_DEFS -> FUNC_DEF NEXT_FUNC_DEFS
NEXT_FUNC_DEFS -> ε
FUNC_PARAMS -> PARAM NEXT_PARAM
FUNC_PARAMS -> ε
NEXT_PARAM -> , OPT_EOL PARAM NEXT_PARAM
NEXT_PARAM -> ε
PARAM -> id TYPE
RET_TYPES -> (TYPE NEXT_RET_TYPE)
RET_TYPES -> ε
NEXT_RET_TYPE -> , TYPE NEXT_RET_TYPE
NEXT_RET_TYPE -> ε
TYPE -> int
TYPE -> float64
TYPE -> string
COMMANDS -> COMMAND COMMANDS
COMMANDS -> ε
COMMAND -> if exp { eol COMMANDS } else { eol COMMANDS } eol
COMMAND -> for OPT_ID_DEF ; exp ; ID_ASSIGN { eol COMMANDS } eol
COMMAND -> return RET_VALS eol
COMMAND -> id COMMAND_CONTINUE eol
COMMAND_CONTINUE -> := exp
COMMAND_CONTINUE -> ID_LIST = OPT_EOL R_ASSIGN
COMMAND_CONTINUE -> (FUNC_ARGS)
OPT_ID_DEF -> ID_DEF
OPT_ID_DEF -> ε
ID_DEF -> id := exp
ID_ASSIGN -> id ID_LIST = OPT_EOL R_ASSIGN
R_ASSIGN -> FUNC_CALL
R_ASSIGN -> EXPRESSION_LIST
FUNC_CALL -> id (FUNC_ARGS)
FUNC_ARGS -> TERM NEXT_ARG
FUNC_ARGS -> ε
NEXT_ARG -> , TERM NEXT_ARG

NEXT_ARG -> ε
ID_LIST -> , id ID_LIST
ID_LIST -> ε
OPT_EOL -> eol
OPT_EOL -> ε
TERM -> string_val
TERM -> int_val
TERM -> float64_val
TERM -> id
RET_VALS -> EXPRESSION_LIST
RET_VALS -> TERM
RET_VALS -> ε
EXPRESSION_LIST -> exp NEXT_EXPRESSION
NEXT_EXPRESSION -> , exp NEXT_EXPRESSION
NEXT_EXPRESSION -> ε

Příloha 3 - LL tabulka

	package	main	eol	eof	func	id	(	)	{	}	,	int	float64	string	if	exp	else	for	;	return	:=	=	string_val	int_val	float64_val	$
PROGRAM	1		1																							
OPT_EOL	41		40	41		41		41								41										
FUNC_DEFS					2																					
FUNC_DEF					3																					
NEXT_FUNC_DEFS			5	5	4																					
FUNC_PARAMS						6		7																		
RET_TYPES							11		12																	
COMMANDS						18				19					18			18		18						
PARAM						10																				
NEXT_PARAM								9			8															
TYPE												15	16	17												
NEXT_RET_TYPE								14			13															
COMMAND						23									20			21		22						
OPT_ID_DEF						27													28							
ID_ASSIGN						30																				
RET_VALS			48			47										46							47	47	47	
COMMAND_CONTINUE							26				25										24	25				
ID_LIST											38											39				
R_ASSIGN						31										32										
FUNC_ARGS						34		35															34	34	34	
ID_DEF						29																				
FUNC_CALL						33																				
EXPRESSION_LIST																49										
TERM						45																	42	43	44	
NEXT_ARG								37			36															
NEXT_EXPRESSION			51						51		50															

Příloha 4 - Precedenční tabulka

pozn.: REL = Relační operátory

	+ -	/ *	REL	(	)	VAR	$
+ -	>	<	>	<	>	<	>
/ *	>	>	>	<	>	<	>
REL	<	<	X	<	>	<	>
(	<	<	<	<	=	<	X
)	>	>	>	X	>	X	>
VAR	>	>	>	X	>	X	>
$	<	<	<	<	X	<	K