



PROJEKTOVÁ DOKUMENTACE

IMPLEMENTACE PŘEKLADAČE JAZYKA IFJ19

TÝM 109, VARIANTA I

Členové týmu:

Tomáš Beránek (xberan46) 25%

Martin Haderka (xhader00) 25%

Richard Klem (xklemr00) 25%

Šimon Slobodník (xslobo06) 25%

Contents

1	Úvod	2
2	Implementace překladače	2
2.1	Lexikální analyzátor	2
2.2	Syntaktický analyzátor	2
2.3	Sémantický analyzátor	3
2.4	Generování kódu	3
3	Metodika vývoje	3
4	Komunikační kanály a verzovací systémy	4

1 Úvod

Cílem projektu bylo v jazyce C implementovat překladač jazyka IFJ19, který je zjednodušenou podmnožinou jazyka Python3, což je dynamicky typovaný imperativní (objektově - orientovaný) jazyk s funkcionálními prvky.[1] A následně jej přeložit do cílového jazyka IFJcode19 s využitím teorií automatů a precedenční syntaktické analýzy, založené na LL-gramtice.

2 Implementace překladače

V druhé kapitole se budeme podrobně věnovat návrhu a implementaci překladače. Popíšeme si i problémy, které se objevily. Funkcionalitu překladače jsme si rozdělili na několik podčástí, kterými se zabývají jednotlivé podkapitoly. Při implemetaci jsme využili zdrojových souborů z předmětu IAL (implementace BVS, zásobník, jednosměrně/dvousměrně vázaný seznam a převod na infixové na postfixovou notaci).

2.1 Lexikální analyzátor

Úkolem lexikálního analyzátoru je projít celým vstupním souborem a zkontrolovat, zda neobsahuje lexikální chyby (např.: špatné odsazení) a následně vygenerovat odpovídající tokeny[2], pro další zpracování. Jedná se o část s kterou jsme začali překladač implementovat. V první fázi jsme si navrhli automat přechodů (viz. Obrázek 1). Následně jsme pokračovali s implementací automatu, pomocí konstrukce switch - case, která se nachází v souboru IFJ_scanner.c. Největším problémem při implementaci byla kontrola escape sekvencí, kdy se v některých řetězcích objevovala kombinace například `'\n'` a `'\Z'` a bylo třeba rozlišit zda se o escape sekvenci jedná a nebo ne.

2.2 Syntaktický analyzátor

Syntaktický analyzátor za pomoci LL-gramatiky (LL-tabulky a LL-pravidel - viz. Obrázek 2) kontroluje, zda je vstupní kód syntakticky správný. Největším problémem bylo sestavení LL-tabulky, kde jsme museli definovat všechny možné stavy, které mohly nastat. LL-tabulka s využitím rekurzivního volání je implementovaná v souboru IFJ_parser.c. Funguje na principu postupného načítání tokenů a s pomocí ta-

bulky se kontrolují jednotlivá pravidla, které odpovídají syntaktické struktuře jazyka IFJ19.

2.3 Sémantický analyzátor

Část sémantického analyzátoru kontrolující definice/redefinice funkcí/proměnných funguje na principu zásobníku, který interně komunikuje s tabulkou symbolů. Tabulka symbolů je implementována v souboru *symtable.c* pomocí binárního vyhledávacího stromu, kde jako klíč slouží jméno daného identifikátoru. Tabulka symbolů je pouze jedna (nerozlišuje se lokální a globální), místo toho se používají informace ve struktuře *Record*, které určují zda je globální/lokální.

Část kontrolující kompatibilitu datových typů a dělení nulou je implementována jako funkce *do_operation* v jazyce IFJcode19 v *builtin.c*. Funkce za běhu kontroluje kompatibilitu operandů všech operací. Funkce je vždy vložena do hlavičky každého vygenerovaného souboru jazyka IFJcode19.

2.4 Generování kódu

Generování výsledného mezikódu probíhá v souborech *builtin.c* (vygenerování hlavičky, vestavěných funkcí včetně funkce na kontrolu datových typů a generování výrazů), *IFJ_stack_semantic.c* (definice proměnných) a *IFJ_parser.c* (řídící konstrukce a definice funkcí). Generování probíhá zároveň se syntaktickou analýzou (jedná se o syntaxi řízený překlad). Mezikód je generován na standardní výstup. Jelikož je generován průběžně, při chybě již vypsaná část programu zůstane na standardním výstupu, a je vypsána příslušná chybová hláška na chybový výstup.

Funkce *\$do_operation* zpracovává veškeré operace ve výrazech pomocí zásobníku dostupného v IFJcode19, ze kterého čte operandy a operace v postfixové notaci. Funkce zároveň provádí kontrolu a potřebné přetypování operandů pro aritmetické operace. V případě chybných datových typů a dělení nulou vypíše příslušný kód chyby.

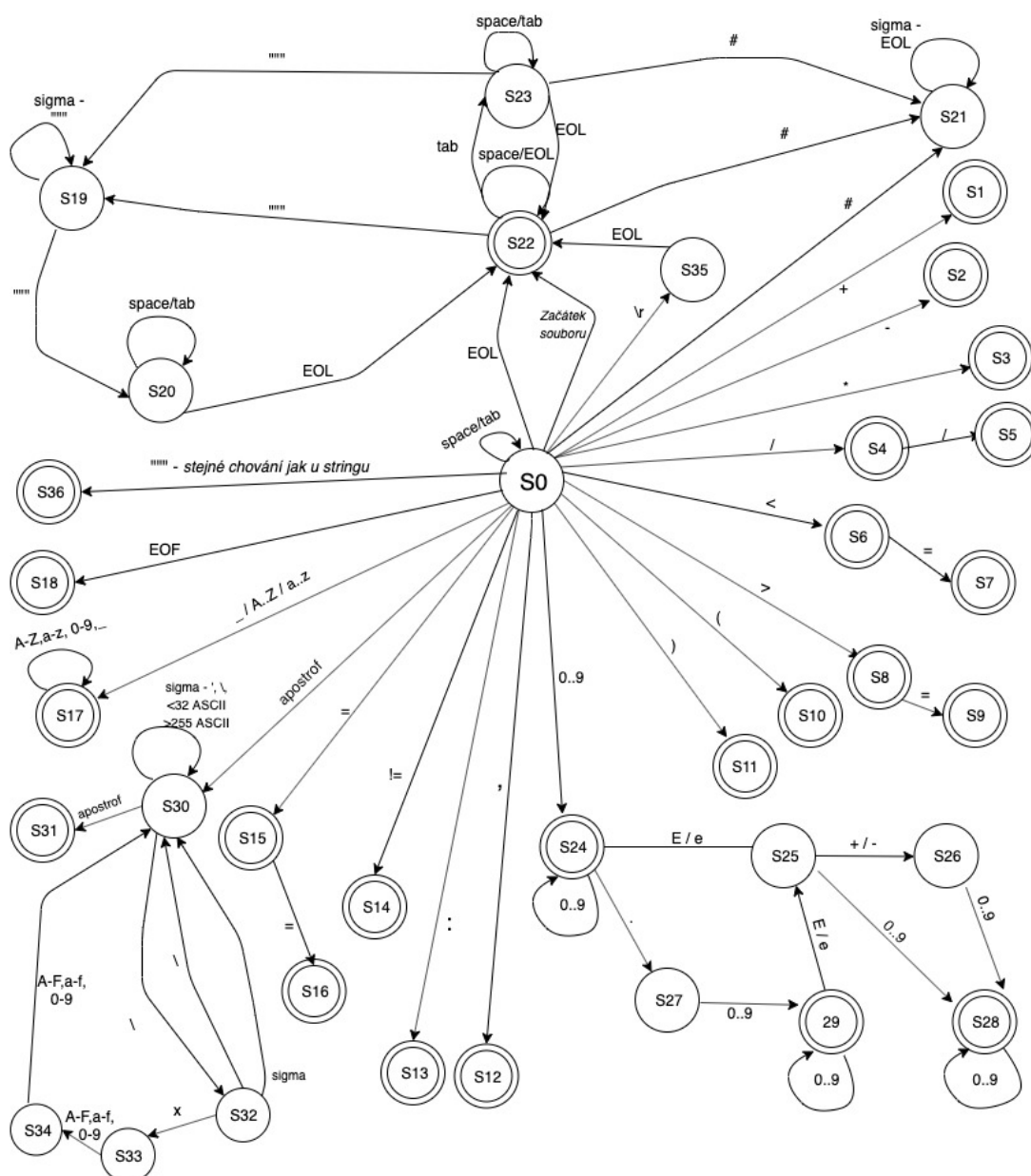
3 Metodika vývoje

Využití scram s pravidelnými sprinty o délce jednoho týdne.

4 Komunikační kanály a verzovací systémy

Ke komunikaci byl využit Slack v kombinaci s Facebook Messengerem a Skypem pro videohovory a sdílení obrazovky, přednostně se však využívalo pravidelných osobních setkání.

Jako verzovací systém byl využit git s hostingem na github.com. Propojili jsme github s aplikací Slack pro notifikaci commitů, aby bylo možné okamžitě reagovat a kontrolovat změny ostatních.



S0 STATE_START	S10 STATE_LEFT_BRACKET	S20 STATE_MULTI_LINE_COMM_AFTER	S30 STATE_STRING
S1 STATE_PLUS	S11 STATE_RIGHT_BRACKET	S21 STATE_SINGLE_LINE_COMM	S31 STATE_STRING_READ
S2 STATE_MINUS	S12 STATE_COMMA	S22 STATE_INDENT_DEDEDENT	S32 STATE_STRING_BACKSLASH
S3 STATE_MUL	S13 STATE_COLON	S23 STATE_EMPTY_LINE	S33 STATE_STRING_HEX_START
S4 STATE_DIV	S14 STATE_NOT_EQ	S24 STATE_INT	S34 STATE_STRING_HEX_END
S5 STATE_INT_DIV	S15 STATE_ASSIGNMENT	S25 STATE_DOUBLE_EXP	S35 STATE_CRLF
S6 STATE_LESS	S16 STATE_EQ	S26 STATE_DOUBLE_EXP_SIGN	S36 STATE_MULTI_LINE_LITERAL
S7 STATE_LESS_EQ	S17 STATE_ID	S27 STATE_DOUBLE_DECIMAL_POINT	
S8 STATE_GREATER	S18 STATE_EOF	S28 STATE_DOUBLE_WITH_EXP	
S9 STATE_GREATER_EQ	S19 STATE_MULTI_LINE_COMM	S29 STATE_DOUBLE_WITHOUT_EXP	

Obrázek 1: Konečný automat lexikálního analyzátoru

LL-pravidla

1. `<prog> -> <eol-opt> <st-list> EOF`
2. `<st-list> -> <stat> <eol-opt> <st-list>`
3. `<stat> -> PASS`
4. `<st-list> -> ε`
5. `<stat> -> ID <expr-or-assign> EOL`
6. `<expr-or-assign> -> = <fun-or-expr>`
7. `<return> -> RETURN <expr> EOL`
8. `<return> -> ε`
9. `<eol-opt> -> EOL <eol-opt>`
10. `<eol-opt> -> ε`
11. `<stat> -> DEF ID (<param-list>) : EOL <eol-opt> INDENT <st-list> <return> DEDENT`
12. `<param-list> -> ID <param-next>`
13. `<param-list> -> ε`
14. `<param-next> -> ε`
15. `<param-next> -> , ID <param-next>`
16. `<stat> -> IF <expr> : EOL <eol-opt> INDENT <st-list> DEDENT ELSE : EOL <eol-opt> INDENT <st-list> DEDENT`
17. `<stat> -> WHILE <expr> : EOL <eol-opt> INDENT <st-list> DEDENT`
18. `<expr-or-assign> -> (+, -, *, /, //, <, <=, >, >=, !=, ==) <expr>`
19. `<stat> -> (STR, INT, DBL, (, NONE) <expr> EOL`
20. `<fun-or-expr> -> ID <fun-or-expr2>`
21. `<fun-or-expr2> -> (<arg-list>)`
22. `<fun-or-expr> -> (STR, INT, DBL, (, NONE) <expr>`
23. `<fun-or-expr2> -> (+, -, *, /, //, <, <=, >, >=, !=, ==) <expr>`

LL-tabulka

L-tabuka	+	-	/	*	//	<	<=	>	>=	str	dbl	int	!=	==	none	id	()	eol	,	:	indent	dedent	=	def	else	while	pass	return	if	eol	
<prog>										1	1	1			1	1	1		1						1		1	1	1	1	1	
<st-list>										2	2	2			2	2	2						4			2	2	4	2	4	2	4
<stat>										19*	19*	19*			19*	5	19*								11		17	3		16		
<eol-opt>										10	10	10			10	10	10		9			10	10		10		10	10	10	10	10	
<return>																							8						7			
<param-list>																12			13													
<param-next>																			14			15										
<expr-or-assign>	18*	18*	18*	18*	18*	18*	18*	18*	18*				18*	18*			24		29	15				6								
<arg-list>										25*	25*	25*			25*	25*			26													
<arg-next>																			27	28*												
<fun-or-expr>										22*	22*	22*			22*	20	22*															
<fun-or-expr2>	23*	23*	23*	23*	23*	23*	23*	23*	23*				23*	23*			21															

Obrázek 2: LL-pravidla a LL-tabulka

Reference

- [1] ifj2019.pdf
- [2] Token = datová struktura popisující lexém.
- [3] <https://www.geeksforgeeks.org/>