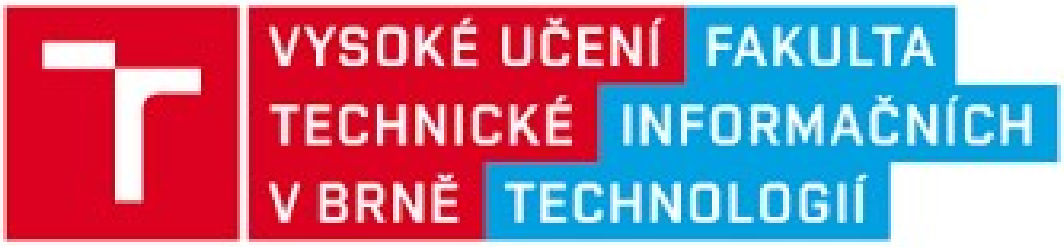


Dokumentace k projektu z předmětu IFJ

Implementace překladače imperativního jazyka IFJ20

Tým 093, Varianta I	**Jakub Hlava**	(xhlava52)	25%
	Lukáš Kraus	(xkraus13)	25%
	Jan Pryč	(xprycj00)	25%
9. prosince 2020	**Thi Bao Ngoc Vu**	(xvuthi00)	25%

Návrh překladače

Lexikální analýza

Lexikální analýza se skládá z hlavní funkce *getToken* a pomocných funkcí *isKW* a *processString*.

Funkce *getToken* načítá znak po znaku ze standardního vstupu a pomocí deterministického konečného automatu, jehož diagram je v příloze 1, implementovaného pomocí konstrukce **switch-case** jazyka **C** provádí samotnou analýzu a určuje, zda načítané znaky odpovídají nějakému typu tokenu a případně jakému nebo zda načítané znaky nemohou utvořit platný token jazyka **IFJ20**.

Znaky načítáme do dynamické struktury *expStr*, kterou jsme vytvořili za účelem snadného ukládání řetězců o neznámé délce, tedy atributů zpracovávaných tokenů.

Jazyk **IFJ20** jsme rozdělili na určitě typy tokenů. Mezi ně ve stručnosti patří identifikátory, klíčová slova, literály (celočíselné, řetězcové a desetinné), aritmetické, logické a relační operátory, závorky, oddělovací znaky a speciální tokeny typu EOL, EOF a NULL, které značí konec řádku, souboru a prázdný token, který je tvořen v případě lexikální chyby.

Tokeny jsou ukládány do struktury token, která si uchovává typ tokenu a jeho nepovinný atribut (např. v případě operátorů je atribut prázdný), který udává hodnotu literálu, jméno proměnné nebo funkce nebo konkrétní klíčové slovo.

Funkce *isKW* je pomocná funkce na podporu konečného automatu, abychom rozpoznali, zda načtený identifikátory patří mezi klíčová slova nebo se k němu má překladač chovat jako k běžnému identifikátoru.

Funkce *processString* již během lexikální analýzy převádí formát řetězcového literálu z formátu, který akceptuje jazyk **IFJ20** na formát, který je vyžadován výsledným jazykem **IFJcode20** tím, že nahrazuje bílé a speciální znaky a escape sekvence na kódy znaků.

Syntaktická analýza

Při tvorbě překladače jsme se rozhodli pro implementaci dvouprůchodové syntaktické analýzy.

V prvním průchodu se sbírají informace o funkcích z jejich deklarací a ukládájí se do tabulky symbolů. Druhý průchod již rozkládá kód podle LL-gramatiky viz příloha 2.
Pravidla jsou reprezentována funkcemi.

Načítání tokenů pro dva průchody je řešeno pomocnou pamětí, kterou tvoří struktura *tokenCache*, ta se před prvním průchodem naplní tokeny a poté je možné strukturou volně procházet a získávat z ní tokeny pomocí funkce *readToken*.

Precedenční analýza pro výrazy

Precedenční analýza je založena na precedenční tabulce (příloha 4), podle které se výrazy vyhodnocují. Výraz spolu s prioritami z tabulky symbolů se postupně pro účely vyhodnocení ukládá do obousměrně vázaného seznamu, odkud se jeho části vybírají a vyhodnocují.

Sémantická analýza

Za pomoci struktur tabulky symbolů a pomocných zásobníků, které jsou předávány mezi funkcemi syntaktické analýzy ve formě struktur *symTable* a gen, jsou kontrolovány datové typy proměnných, počty a datové typy vstupních a výstupních parametrů funkcí a existence identifikátorů v daném rámci.

Generátor kódu

Generátor pro svoji funkci používá strukturu *gen* popsanou níže.

Kvůli alokaci pomocných struktur má generátor krom funkcí pro samotné generování kódu i funkce pro inicializaci a destrukci, které je potřeba zavolat před, resp. po dokončení syntaktické analýzy.

Samotné funkce, které generují kód v jazyce **IFJcode20** jsou spouštěny syntaktickou analýzou po naplnění pomocných struktur daty.

Nepodařilo se nám v termínu implementovat veškerou funkcionalitu generátoru a proto jsou některé části při generování vynechány.

Tabulka symbolů

Naše varianta zadání nás zavázala k vytvoření tabulky pomocí binárního vyhledávacího stromu.

Kvůli potřebě rozlišovat zastínění proměnných a oddělovat proměnné mezi funkcemi jsme se rozhodli lokální tabulku symbolů implementovat jako zásobník zásobníků tabulek symbolů, kdy nejvyšší zásobník reprezentuje funkci, zásobníky v něm reprezentují sekce uvnitř funkce a proměnné v samotné sekci jsou již uloženy pomocí binárního vyhledávacího stromu.

Globální tabulka symbolů nedává v **IFJ20** smysl, protože v tomto jazyce není možnost definovat globální proměnné, proto je globální tabulka symbolů nahrazena stromem funkcí, taktéž implementovaným pomocí binárního vyhledávacího stromu. Ten obsahuje informace o funkcích – jejich jména a počty a datové typy vstupních i výstupních parametrů.

Tyto zásobníky a strom funkcí jsou zastřešeny strukturou *symTable*.

Speciální techniky, algoritmy a struktury

Struktura *expStr* – dynamické řetězce

Jak již bylo zmíněno v části o lexikální analýze, struktura *expStr* slouží pro ukládání a zpracování načítaných řetězců o neznámé délce. Toho je dosaženo postupnou realokací pole při naplnění jeho kapacity o poměrnou velikost.

Struktura *expStr* obsahuje pole znaků, údaj o množství zaplněného a celkem alokovaného prostoru a má obslužné funkce pro inicializaci a dealokaci, pro připisování načítaných znaků a pro „export“ výsledného řetězce jako atributu s možností přímého převodu na celé nebo desetinné číslo.

Pomocná paměť na tokeny *tokenCache*

Tato struktura má využití v syntaktické a sémantické analýze a umožňuje opakovaně procházet načtené tokeny s možností ponechat token pro přečtení další volané funkci, která poté rozhodne o dalším postupu, případně se vracet na uložené místo v paměti pro účely vyčítání dalších dat.

Na základě těchto možností a principů postupem času krom základních obslužných funkcí na inicializaci, deinicializaci, zápis a čtení vznikly ještě funkce na práci se „záložkou“ v paměti a pro posun hlavy na uložené místo, které se používají jako náhrada pomocných zásobníků na různých místech.

Pomocné struktury *aStack*, *vStack*, *printQueue*, …

Obvykle jednoúčelové struktury, zásobníky a fronty, které se používají jako pomocné pro uchování tokenů, informací o proměnných a dalších pomocných dat obvykle při generování kódu nebo sémantické analýze.

Obecně jsou implementovány obvykle formou jednosměrně nebo obousměrně vázaných lineárních seznamů s pomocnými ukazateli na začátek, konec nebo aktivní prvek. Tuto formu

implementace jsme zvolili i přes nepřesné názvosloví především kvůli jejímu charakteru, kdy nejsme omezování velikostí předem alokovaných prostor atp.

Struktura *gen*

Pomocná struktura generátoru, která si udržuje globální počítadlo zajišťující unikátnost pomocných i ostatních proměnných a dále obsahuje ukazatele na tabulku symbolů a instance *vStack* a *aStack* pro komunikaci mezi syntaktickou analýzou a generátorem kódu.

Práce v týmu

Plánování

Při tvorbě projektu jsme se nedrželi pevného plánu rozdělení prací. Práce jsme si dělili postupně podle náročnosti jednotlivých částí projektu a podle priorit závislostí součástí na sobě.

Spolupráce v týmu

Pro správu kódu jsme použili verzovací systém Git a hosting GitHub, kde jsme využili i jeho funkce pro správu projektu - Issues, Pull Requests, Milestones, Project.

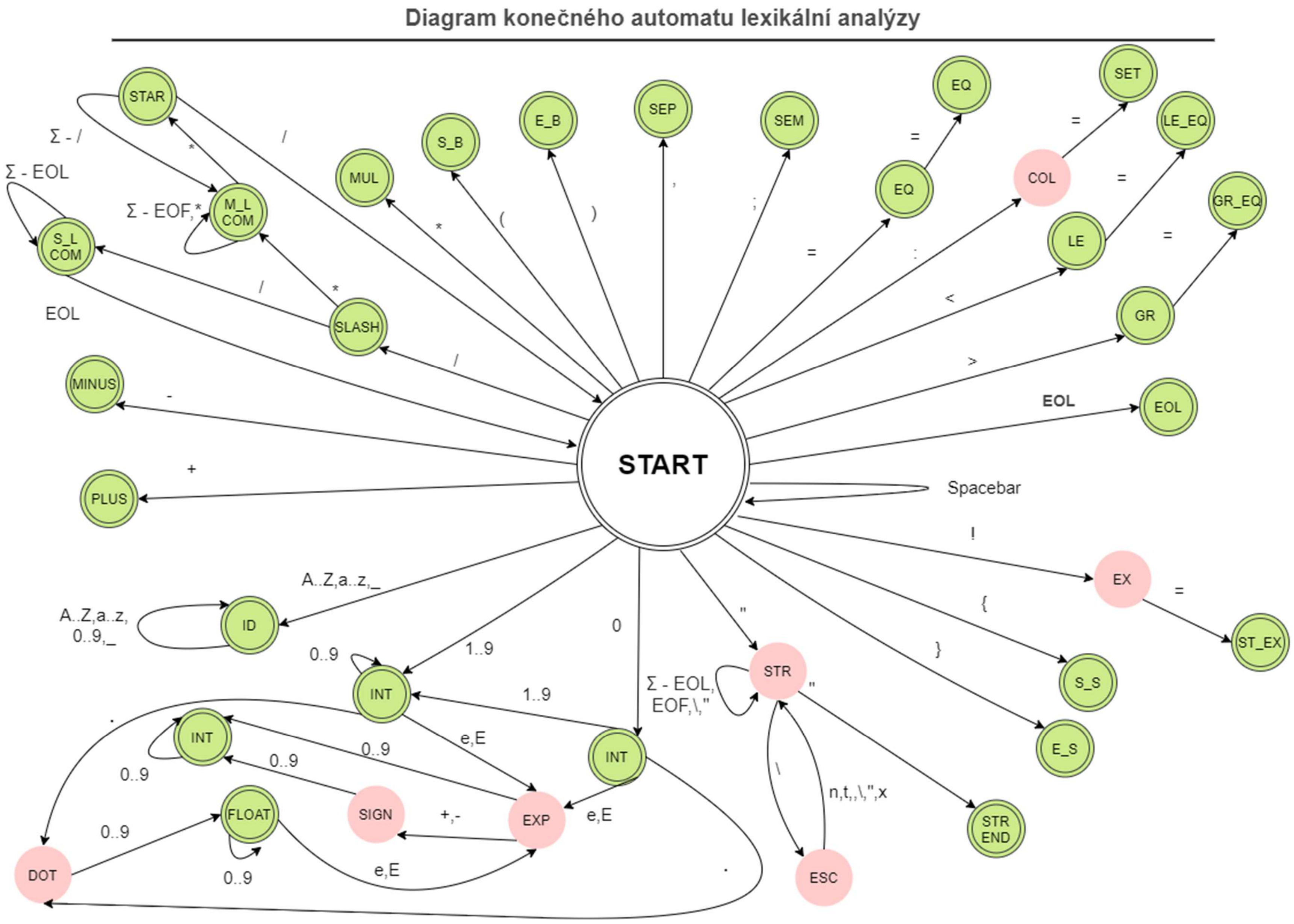
Pro komunikaci jsme mimo nástrojů GitHubu využívali hojně Discord a TeamSpeak.

Komunikace probíhala jak průběžně na Discordu, tak minimálně jednou týdně setkáním celého týmu na TeamSpeaku a probráním splněných cílů, plánů do budoucna a případných problémů.

Rozdělení práce

Člen týmu	Přidělená práce
Jakub Hlava (xhlava52)	vedení týmu, organizace práce, lexikální analýza, generování kódu, tvorba a implementace tabulky symbolů, testování
Lukáš Kraus (xkraus13)	implementace syntaktické analýzy, precedenční analýzy, tvorba LL-gramatiky, LL-tabulka, precedenční tabulka, testování
Jan Pryč (xprycj00)	dokumentace, prezentace, implementace syntaktické analýzy, tvorba LL-gramatiky, testování
Thi Bao Ngoc Vu (xvuthi00)	implementace sémantické analýzy, tvorba a implementace tabulky symbolů, testování

Přílohy



Obrázek 1 Diagram automatu lexikální analýzy

Legenda k diagramu

START	START
SEP	SEPARATOR
SEM	SEMICOLON
EQ	EQUALS
COL	COLON
SET	SET
LE	LESS
LE_EQ	LESS_EQUAL
GR	GREATER
GR_EQ	GREATER_EQUAL
EOL	END_OF_LINE
SPACEBAR	SPACEBAR (WHITESPACE)
EX	EXCLAMATION_MARK
ST_EX	STATE_EXLAMATION
S_S	START_BRACKET
E_S	END_PRACKET
STR	STRING
ESC	ESCAPE

STREND	STRING_END
INT	INTEGER
EXP	EXPONENT
SIGN	SIGN
DOT	DOT
FLOAT	FLOAT64
ID	IDENTIFIER
PLUS	PLUS
MINUS	MINUS
SLASH	COMMENTARY, DIVISION
S_L COM	SINGLE_LINE COMMENTARY
M_L COM	MULTI_LINE COMMENTARY
STAR	STAR
MUL	MULTIPLY
S_B	START_BRACKET
E_B	END_BRACKET
Σ	ALPHABET
EOF	END_OF_FILE

Příloha 2: LL - Gramatika

<typstart> -> (type <typ>)
<typstart> -> ε
<typ> -> ,type <typ>
<typ> -> ε
<příkaz> -> ID <operation>
<operation> -> := <single_right> EOL <příkaz>
<operation> -> <IDNext> = <right_value> EOL <příkaz>
<příkaz> -> if <expression> { EOL <příkaz> } else { EOL <příkaz> } EOL <příkaz>
<příkaz> -> ε
<příkaz> -> for <f_def>;<expression>;<f_set> { EOL <příkaz> } EOL <příkaz>
<příkaz> -> return <ret_vals> EOL <příkaz>
<příkaz> -> call (<arg>) EOL <příkaz>
<ret_vals> -> ε
<ret_vals> -> <expression> <ret_next>
<ret_vals> -> <term> <ret_next>
<ret_next> -> ,<expression> <ret_next>
<ret_next> -> ε
<f_def> -> ε
<f_def> -> ID := <single_right>
<f_set> -> ε
<f_set> -> ID = <single_right>
<IDNext> -> ,ID <IDNext>
<IDNext> -> ε
<single_right> -> call (<arg>)
<single_right> -> <expression>
<single_right> -> <term>
<right_value> -> <single_right> <right_next>
<right_next> -> ,<single_right> <right_next>
<right_next> -> ε
<arg> -> <term> <argNext>
<arg> -> ε
<argNext> -> ε
<argNext> -> ,<term> <argNext>
<expression> -> exp
<term> -> int_value
<term> -> str_value
<term> -> float64_value
<term> -> ID
<func_def> -> EOF

Příloha 3: LL - Tabulka

	EOL	func	(	ID	$	,	if	for	return	EOF	int_value	str_value	float64_value	expression	:	call
<func_def>		3								47						
<parametry>			4													
<vstup>				5	6											
<vstupNext>					8	7										
<typstart>			9		10											
<typ>					12	11										
<příkaz>				13	17		16	18	19							20
<ret_vals>				23	21						23	23	23	22		
<f_def>				27	26											
<ID>																
<IDNext>					31	30										
<single_right>				34							34	34	34	33		32
<right_value>				35							35	35	35	35		35
<right_next>					37	36										
<arg>				38	39						38	38	38			
<argNext>					40	41										
<term>				46						43	44	45				
<ret_next>					25	24										
<f_set>				29	28											
<operation>					15	15									14	

Příloha 4: Precedenční tabulka

	+	-	*	/	(	)	i	$	LO
+	>	>	<	<	<	>	<	>	>
-	>	>	<	<	<	>	<	>	>
*	>	>	>	>	<	>	<	>	>
/	>	>	>	>	<	>	<	>	>
(	<	<	<	<	<	=	<		<
)	>	>	>	>		>		>	>
i	>	>	>	>		>		>	>
$	<	<	<	<	<		<		<
LO	<	<	<	<	<	>	<	>	

první	druhý

LO = logické operátory