IFJ – protokol k projektu
Tým xjarol06, varianta TRP

Antonín Jarolím (xjarol06) - 28 bodů
Jakub Vlk (xvlkja07) - 26 bodů
Jan Brudný (xbrudn02) - 21 bodů
Jindřich Vodák (xvodak06) - 25 bodů

7. prosince 2022

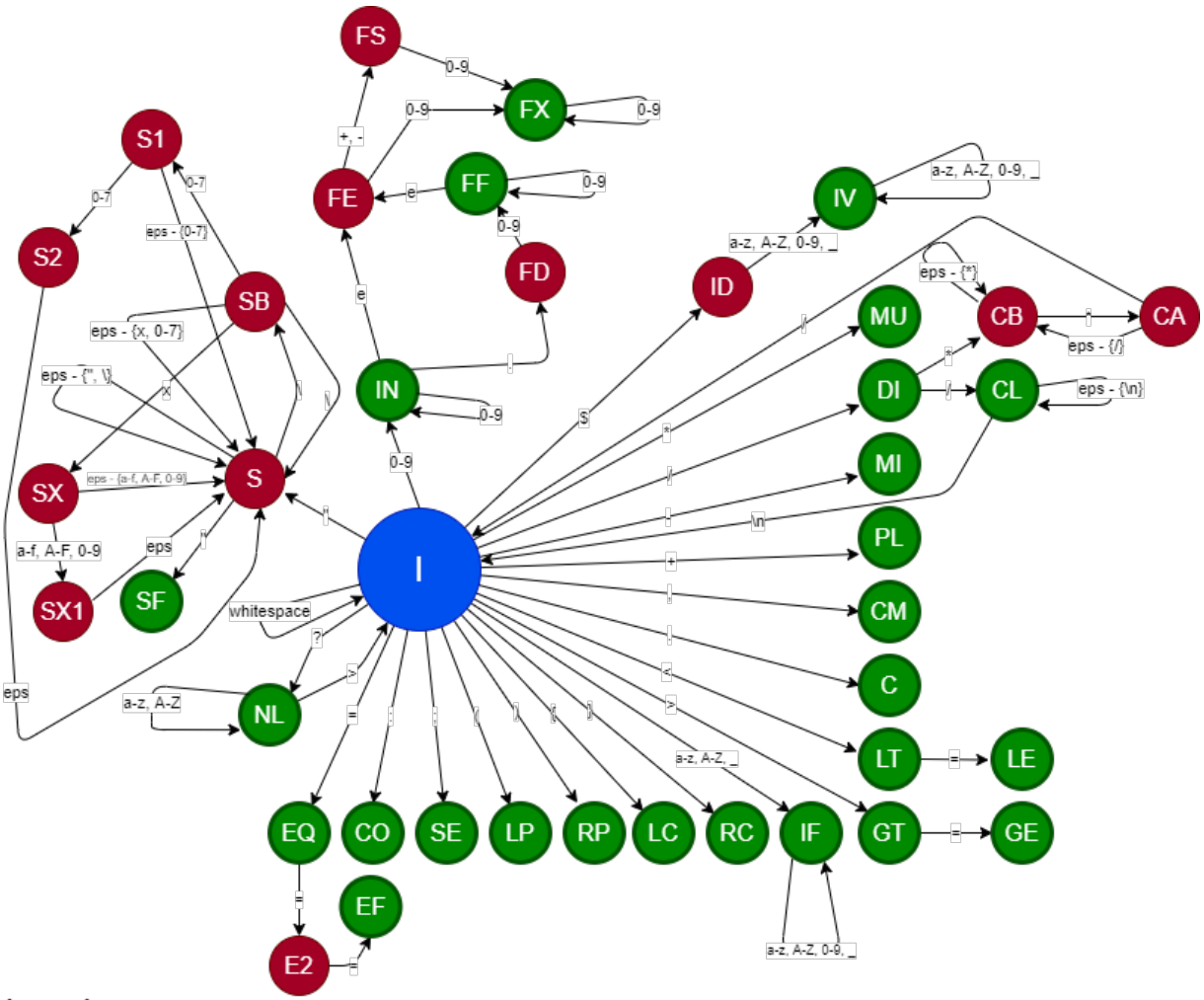
Obsah

Rozdělení práce mezi členy týmu

Antonín Jarolím - syntaktický analyzátor top-down, generace kódu, LL-tabulka, gramatika
Jakub Vlk - syntaktický analyzátor bottom-up, generace kódu, precedenční tabulka, gramatika
Jan Brudný - tabulka symbolů, generace kódu
Jindřich Vodák - lexikální analyzátor, psaní testů, dokumentace

Body byly v týmu rozděleny s přihlédnutím k množství odvedené práce, aktivnímu zapojování se do práce na projektu v průběhu semestru a vyvinuté snaze. Rozdělování probíhalo postupně v průběhu semestru - v případě, že tři členové týmu shledali, že čtvrtý odvádí velmi dobrou práci, body byly přerozděleny tak, aby čtvrtý člen získal bod navíc. V opačném případě byl jeden bod čtvrtému členovi týmu odebrán.

Diagram konečného automatu



Legenda

I = init_s
S = string_lit_s
SF = string_lit_f_s
IN = integer_lit_f_s
FD = float_lit_dot_s
FE = float_lit_e_s
FS = float_lit_sign_s
FX = float_lit_exp_f_s
FF = float_lit_f_s
IF = identifier_func_f_s
ID = identifier_var_dollar_s
IV = identifier_var_f_s
MU = multiplication_f_s
DI = division_f_s
PL = plus_f_s
MI = minus_f_s
C = concatenation_f_s
LT = lesser_than_f_s
LE = lesser_eq_f_s
GT = greater_than_f_s
GE = greater_eq_f_s
EF = eq_f_s

E2 = eq_2_s
N1 = not_eq_1_s
N2 = not_eq_2_s
NF = not_eq_f_s
SB = string_lit_backslash_s
SX = string_lit_backslash_x_s
SX1 = string_lit_backslash_x_1_s
S1 = string_lit_backslash_1_s
S2 = string_lit_backslash_2_s
LP = left_par_f_s
RP = right_par_f_s
CL = com_line_f_s
CB = com_block_s
CA = com_block_ast_s
LC = left_curly_f_s
RC = right_curly_f_s
EQ = equals_f_s
CO = colon_f_s
SE = semicolon_f_s
CM = comma_f_s
NL = null_f_s

LL-gramatika

ProgramBody ::= FceDefine ProgramBody
ProgramBody ::= Command ProgramBody
ProgramBody ::=”

Command ::= DeclareVariable
Command ::= Condition
Command ::= While
Command ::= Return
Command ::= Exp semicolon
Command ::= semicolon
Command ::= FceCall

FceDefine ::= functionKey FceHeader curlyBraceLeft FunctionBody curlyBraceRight
FceHeader ::= identifierFunc leftPar FunctionDeclareParams rightPar colon FuncReturnColonType
FunctionDeclareParams ::=”
FunctionDeclareParams ::= DataType DeclareParam CommaOrEpsParams
CommaOrEpsParams ::=”
CommaOrEpsParams ::= comma DataType DeclareParam CommaOrEpsParams
DeclareParam ::= identifierVar

FuncReturnColonType ::= DataType
FuncReturnColonType ::= voidKey

FceCall ::= identifierFunc leftPar FirstFceParam rightPar
FirstFceParam ::=”
FirstFceParam ::= Statement CommaOrEpsParam
CommaOrEpsParam ::=”
CommaOrEpsParam ::= comma Statement CommaOrEpsParam

Statement ::= identifierVar
Statement ::= floatLiteral
Statement ::= stringLiteral
Statement ::= integerLiteral
Statement ::= nullKey

Exp ::= Statement
Exp ::= Exp minusOp Exp
Exp ::= Exp plusOp Exp
Exp ::= Exp divisionOp Exp
Exp ::= Exp multiplicationOp Exp
Exp ::= Exp concatenationOp Exp

Exp ::= Exp lesserThanOp Exp
Exp ::= Exp lesserEqOp Exp
Exp ::= Exp greaterThanOp Exp
Exp ::= Exp greaterEqOp Exp

Exp ::= Exp eqOp Exp
Exp ::= Exp notEqOp Exp

Exp ::= leftPar Exp rightPar

DataType ::= stringNullKey
DataType ::= floatNullKey
DataType ::= intNullKey
DataType ::= stringKey
DataType ::= floatKey
DataType ::= intKey

DeclareVariable ::= identifierVar equals DefVarAss
DefVarAss ::= Exp semicolon
DefVarAss ::= FceCall semicolon

Condition ::= ifKey leftPar Exp rightPar curlyBraceLeft FunctionBody curlyBraceRight ElseCond
ElseCond ::= elseKey curlyBraceLeft FunctionBody curlyBraceRight
ElseCond ::=”

While ::= whileKey leftPar Exp rightPar curlyBraceLeft FunctionBody curlyBraceRight

Return ::= returnKey ReturnExp semicolon
ReturnExp ::= Exp
ReturnExp ::=”

FunctionBody ::= Command FunctionBody
FunctionBody ::=”

LL-tabulka

	$	semicolon	curlyBraceLeft	curlyBraceRight	functionKey
S	S ::= ProgramBody $	S ::= ProgramBody $			S ::= ProgramBody $
ProgramBody	ProgramBody ::= ε	ProgramBody ::= Command ProgramBody			ProgramBody ::= FceDefine ProgramBody
Command		Command ::= semicolon			
FceDefine					FceDefine ::= FceHeader curlyBraceLeft FunctionBody curlyBraceRight
FceHeader					FceHeader ::= functionKey identifierFunc leftPar FunctionDeclareParams rightPar colon FuncReturnColonType
ElseCond	ElseCond ::= ε	ElseCond ::= ε		ElseCond ::= ε	ElseCond ::= ε
ReturnExp		ReturnExp ::= ε			
FunctionBody		FunctionBody ::= Command FunctionBody		FunctionBody ::= ε	

	identifierFunc	leftPar	rightPar
S	S ::= ProgramBody $	S ::= ProgramBody $	
ProgramBody	ProgramBody ::= Command ProgramBody	ProgramBody ::= Command ProgramBody	
Command	Command ::= FceCall	Command ::= Exp semicolon	
FunctionDeclareParams			FunctionDeclareParams ::= ε
CommaOrEpsParams			CommaOrEpsParams ::= ε
FceCall	FceCall ::= identifierFunc leftPar FirstFceParam rightPar		
FirstFceParam			FirstFceParam ::= ε
CommaOrEpsParam			CommaOrEpsParam ::= ε
DefVarAss	DefVarAss ::= FceCall semicolon	DefVarAss ::= Exp semicolon	
ElseCond	ElseCond ::= ε	ElseCond ::= ε	
ReturnExp		ReturnExp ::= Exp	
FunctionBody	FunctionBody ::= Command FunctionBody	FunctionBody ::= Command FunctionBody	

	comma	identifierVar	voidKey	floatLiteral
S		S ::= ProgramBody $		S ::= ProgramBody $
ProgramBody		ProgramBody ::= Command ProgramBody		ProgramBody ::= Command ProgramBody
Command		Command ::= DeclareVariable Command ::= Exp semicolon		Command ::= Exp semicolon
CommaOrEpsParams	CommaOrEpsParams ::= comma DeclareParam CommaOrEpsParams			
FuncReturnColonType			FuncReturnColonType ::= voidKey	
FirstFceParam		FirstFceParam ::= Statement CommaOrEpsParam		FirstFceParam ::= Statement CommaOrEpsParam
CommaOrEpsParam	CommaOrEpsParam ::= comma Statement CommaOrEpsParam			
Statement		Statement ::= identifierVar		Statement ::= floatLiteral
DeclareVariable		DeclareVariable ::= identifierVar equals DefVarAss		
DefVarAss		DefVarAss ::= Exp semicolon		DefVarAss ::= Exp semicolon
ElseCond		ElseCond ::= ε		ElseCond ::= ε
ReturnExp		ReturnExp ::= Exp		ReturnExp ::= Exp
FunctionBody		FunctionBody ::= Command FunctionBody		FunctionBody ::= Command FunctionBody

	stringLiteral	integerLiteral	nullKey
S	S ::= ProgramBody $	S ::= ProgramBody $	S ::= ProgramBody $
ProgramBody	ProgramBody ::= Command ProgramBody	ProgramBody ::= Command ProgramBody	ProgramBody ::= Command ProgramBody
Command	Command ::= Exp semicolon	Command ::= Exp semicolon	Command ::= Exp semicolon
FirstFceParam	FirstFceParam ::= Statement CommaOrEpsParam	FirstFceParam ::= Statement CommaOrEpsParam	
Statement	Statement ::= stringLiteral	Statement ::= integerLiteral	
DefVarAss	DefVarAss ::= Exp semicolon	DefVarAss ::= Exp semicolon	DefVarAss ::= Exp semicolon
ElseCond	ElseCond ::= ε	ElseCond ::= ε	ElseCond ::= ε
ReturnExp	ReturnExp ::= Exp	ReturnExp ::= Exp	ReturnExp ::= Exp
FunctionBody	FunctionBody ::= Command FunctionBody	FunctionBody ::= Command FunctionBody	FunctionBody ::= Command FunctionBody

	stringNullKey	floatNullKey	intNullKey	stringKey
FunctionDeclareParams	FunctionDeclareParams ::= DeclareParam CommaOrEpsParam	FunctionDeclareParams ::= DeclareParam CommaOrEpsParam	FunctionDeclareParams ::= DeclareParam CommaOrEpsParams	FunctionDeclareParams ::= DeclareParam CommaOrEpsParam
DeclareParam	DeclareParam ::= DataType identifierVar	DeclareParam ::= DataType identifierVar	DeclareParam ::= DataType identifierVar	DeclareParam ::= DataType identifierVar
FuncReturnColonType	FuncReturnColonType ::= DataType	FuncReturnColonType ::= DataType	FuncReturnColonType ::= DataType	FuncReturnColonType ::= DataType
DataType	DataType ::= stringNullKey	DataType ::= floatNullKey	DataType ::= intNullKey	DataType ::= stringKey

	floatKey	intKey	ifKey
S			S ::= ProgramBody $
ProgramBody			ProgramBody ::= Command ProgramBody
Command			Command ::= Condition
FunctionDeclareParams	FunctionDeclareParams ::= DeclareParam CommaOrEpsParams	FunctionDeclareParams ::= DeclareParam CommaOrEpsParams	
DeclareParam	DeclareParam ::= DataType identifierVar	DeclareParam ::= DataType identifierVar	
FuncReturnColonType	FuncReturnColonType ::= DataType	FuncReturnColonType ::= DataType	
DataType	DataType ::= floatKey	DataType ::= intKey	
Condition			Condition ::= ifKey leftPar Exp rightPar curlyBraceLeft FunctionBody curlyBraceRight ElseCond
ElseCond			ElseCond ::= ε
FunctionBody			FunctionBody ::= Command FunctionBody

	elseKey	whileKey	returnKey
S		S ::= ProgramBody $	S ::= ProgramBody $
ProgramBody		ProgramBody ::= Command ProgramBody	ProgramBody ::= Command ProgramBody
Command		Command ::= While	Command ::= Return
ElseCond	ElseCond ::= elseKey curlyBraceLeft FunctionBody curlyBraceRight	ElseCond ::= ε	ElseCond ::= ε
While		While ::= whileKey leftPar Exp rightPar curlyBraceLeft FunctionBody curlyBraceRight	
Return			Return ::= returnKey ReturnExp semicolon
FunctionBody		FunctionBody ::= Command FunctionBody	FunctionBody ::= Command FunctionBody

Precedenční tabulka

	+ -	* /	ID lit.	.	(	)	boolOP	$
+ -	>	<	<	>	<	>	>	>
* /	>	>	<	>	<	>	>	>
ID lit.	>	>		>		>	>	>
.			<		<	>	>	>
(	<	<	<	<	<	=	<	<
)	>	>		>		>	>	>
boolOP	<	<	<	<	<	>		>
$	<	<	<	<	<		<	

Členění implementačního řešení

Lexikální analyzátor sestává ze dvou souborů - hlavičkového souboru *lex.h*, ve kterém jsou definovány všechny důležité struktury a hlavičky funkcí používaných dále v programu, a zdrojového souboru *lex.c*, jehož kód vykonává samotnou lexikální analýzu. Lexikální analyzátor generuje tokeny ve formě struktury definované v hlavičkovém souboru, která obsahuje typ lexému (využit výčtový typ `lexType` definovaný rovněž v hlavičkovém souboru), informace o pozici v textu (pro ladění a chybové výpisy) a vnořenou datovou strukturu `data_t`. Tato vnořená struktura je typu union a jejím účelem je uchovávat datový obsah tokenu (pokud jej tedy token má) - například v případě celočíselného literálu je do proměnné `valueInteger` datového typu `int` uvnitř struktury uložena hodnota daného literálu. V případě řetězcového literálu využívá struktura `data_t` pomocné knihovny *dynstring.c*, která obsahuje speciální datový typ `dynStr_t` a funkce výrazně usnadňující práci s dynamickými řetězci. To s sebou přináší značné výhody nejen skrze jednodušší ladění, ale také například při práci s escape sekvencemi.

Pro vygenerování tokenu ze zdrojových dat je nutno zavolat hlavní a největší funkci celého lexikálního analyzátoru `getToken()`. Tato funkce načítá vstupní data znak po znaku pomocí jednoduché funkce `getNextChar()` a skrze vnitřní konečný automat nalezne pro lexém odpovídající stav. Pokud je detekován identifikátor či literál, je při čtení dat zároveň aktivován také buffer, který všechny přečtené znaky ukládá, a pokud jde o řetězcový literál obsahující jednu nebo více escape sekvencí, je zároveň aktivován druhý buffer uchovávající danou escape sekvenci. Tato escape sekvence je okamžitě zpracována dle svého typu, konvertovaný znak je uložen do datového bufferu a sekvenční buffer je vyčištěn. Při dosažení separátoru (dle lexému může jít o bílý znak, speciální znak či jen začátek dalšího lexému) je konečný stav a buffer předán druhé části analyzátoru, která na základě stavu rozhodne o typu tokenu a uloží do něj jeho data, a pokud je to nutné (v případě přerušení začátkem nového lexému), na začátek vstupního proudu je pomocí funkce `ungetNextChar()` vrácen poslední načtený znak.

Lexikální analyzátor podporuje několik zajímavých funkcí, které slouží buď ke zjednodušení procesu ladění nebo k usnadnění práce při dalších fázích překladu. Jako první příklad uveďme funkci `printTokenData()`, jejímž vstupem je token a výstupem informace o typu tokenu, jeho obsahu a jeho pozici ve zdrojovém textu. Jde o velmi jednoduchou funkci, která se ale ukázala jako zcela nepostradatelná i v posledních fázích práce na projektu. Pro účely testování vznikl jednoduchý program obsahující tuto funkci, který byl schopen načíst data ze zdrojového souboru a cyklicky vypsat informace o všech zpracovaných tokenech. Tento program sloužil jako důležitá pomoc při práci na lexikálním analyzátoru a pomohl vyřešit spoustu obtížně zachytitelných chyb.

Druhým příkladem je funkce `ungetToken()`, která funguje analogicky s `ungetNextChar()` - na výstup lexikálního analyzátoru vrátí poslední zpracovaný token. Samotný lexikální analyzátor obsahuje buffer, v němž je vždy uložen poslední zpracovaný token, a přepínač uchovávající informaci o tom, zda se má při volání funkce `getToken()` zpracovat nový lexém či právě vrátit tento poslední token. Jde o funkci důležitou pro syntaktický analyzátor, který se tak nemusí starat o uchovávání tokenů a stačí mu funkci jednoduše zavolat.

Syntaktický analyzátor je hlavní řídící jednotkou překladače a podobně jako lexikální analyzátor je implementován ve dvou souborech - hlavičkovém *parser.h* a zdrojovém *parser.c*. Využívá prediktivní analýzu řízenou LL-tabulkou. Tento postup byl zvolen z důvodu své relativní jednoduchosti.

Analyzátor využívá dynamicky uloženou tabulku vytvořenou voláním funkce `createLLTable()` v souboru *LLTable.c*. Ta volá staticky zapsanou sadu funcí, které vkládají pravidla do tabulky. Nabízí poté rozhraní, které voláním funce `getLLMember()` zavolá. Buňka tabulky je uložena

jako struktura `tableMember`, která ukládá neterminál (výčtový typ `nonTerminalType`), terminál (výčtový typ `lexType`) a pravidla (vnořená struktura).

Při překladu syntaktický analyzátor zpracovává tokeny, které mu předá lexikální analyzátor, a používá strukturu `stack` definovanou v hlavičkovém souboru *stack.h*, kam poté ukládá struktury `stackMem_t`. `stackMem_t` může být terminál, neterminál nebo precedent. Pokud analyzátor detekuje, že na vrcholu zásobníku je neterminál *Expression* nebo *BoolExp*, předává řízení precedenčnímu analyzátoru. Pro správné zpracování se zde se používá funce `ungetToken()`.

TODO

Precedenční analyzátor používá precedenční tabulku, která je součástí tohoto dokumentu. Je tvořen opět dvěma soubory - zdrojovým *expParse.c* a hlavičkovým *expParse.h*. Precedenční analyzátor načítá tokeny ze zdroje skrze funkci `getToken()` z lexikálního analyzátoru a poté vyhledává v precedenční tabulce asociativitu (tabulka je implementována jako dvojrozměrné pole). Stack může obsahovat asociativní pravidla, token, případně neterminály (výraz nebo pravdivostní výraz). Pokud je detekována levá asociativita, jsou prohledána všechna pravidla pro výraz. Pokud není pravidlo nalezeno, je vrácena chyba. Takto se pokračuje, dokud se na stacku nenachází pouze znak `$`. Poté je řízení předáno zpět syntaktickému analyzátoru.

LL-tabulka je v našem případě implementována jako dvojrozměrné pole. Bohužel se nám nepodařilo vymyslet způsob, kterým by bylo možno tabulku vytvořit staticky takovým způsobem, aby byla dostatečně přehledná. Proto je tabulka vytvářena za běhu pomocí funkce `insertMember()`, která vkládá pravidla na příslušné místo v tabulce. Kostrou LL-tabulky je zdrojový soubor *LLtable.c* a hlavičkový soubor *LLtable.h*.

Gramatika a její návrh byl jednou z prvních věcí, které se náš tým při práci na projektu věnoval. Návrh probíhal jako týmová aktivita, kdy jsme procházeli jednotlivé části syntaxe jazyka IFJ22 a vzájemně si zpochybňovali své navrhy. Postupně jsme tato pravidla sepsali do souboru *LLgrammar.txt* a v průběhu práce na projektu gramatiku průběžně vylepšovali. Tabulka je poměrně obsáhlá a vyskytuje se v ní mnoho pravidel, domníváme se však, že značnou část syntaktických chyb se našemu překladači daří odchytávat právě díky tomu.

Generace kódu je implementována tak, že kdekoliv, kde je provedena derivace, se rozhodne (na základě provedené derivace), která funkce pro generování kódu bude použita. Hodnoty a další informace o literálech a proměnných jsou ukládány na stack, odkud jsou při generaci načteny a následně párovány s konkrétními instrukcemi z IFJcode22. Kvůli nutnosti definice proměnných před cykly (aby nedošlo k redefinici proměnných při interpretaci, což by vyústilo v chybu) byl použit buffer, do kterého se prvně generuje kód. V bufferu je možné měnit pořadí jednotlivých příkazů. Tento buffer je implementován jako lineární seznam. Po dokončení překladu jedné komponenty je celý buffer vypsán na standardní výstup.

Pomocné soubory: Tato implementace překladače obsahuje mimo jiné také zdrojový soubor *common.c* a jeho hlavičkový soubor *common.h*. Jde o pomocnou knihovnu obsahující různé pomocné funkce a makra, která napomáhají běhu programu ve všech jeho součástech. Mimo jiné tyto soubory obsahují např. definice všech možných chyb, a také funkce, které vypisují příslušnou chybu v případě, že některá část programu selže - například se zde vyskytuje funkce `PrintErrorExit()`, která je zodpovědná za výpis chyby s odpovídajícím chybovým kódem a následné korektní ukončení programu. Mimo to se zde objevují kontrolní prvky pro některé z testovacích souborů, které jsme vytvořili a posléze využili k testování jednotlivých částí pro-

gramu. Pomocné soubory rovněž zahrnují například definice všech terminálů a neterminálů.

Pomocné soubory jsou v programu využívány např. pro ladicí výpisy. To se ukázalo jako nesmírně efektivní cesta k debugování programu - kdykoliv se vyskytla chyba, z chybového výpisu se dalo vyčíst mnoho údajů, které pomohly chybu eliminovat. V souvislosti s tím jsme se zároveň naučili používat variadická makra; práce s proměnným počtem argumentů nám velmi pomohla právě při psaní ladicích funkcí.

Závěr

Pro náš tým šlo o poměrně velmi náročný projekt, který zabral spoustu času a práce. Všichni členové týmu se však shodli na tom, že je projekt velmi přínosný a člověk, který jej dokončí, získá spoustu důležitých znalostí a zkušeností. Velkou výhodou projektu je, že zadání nabízí relativně velkou svobodu a spoustu věcí může člověk implementovat libovolným způsobem či metodou. Samotný rozsah projektu nutí členy týmu, aby skutečně spolupracovali, společně rozvrhovali čas a úkoly, což náš tým hodnotí pozitivně.

Naše práce na projektu probíhala formou četných schůzek a plánování, společného programování ať už online či offline a vzájemnou kontrolou. Jednotlivé dílčí části projektu byly rozděleny mezi členy týmu dle zájmu a schopností. Pro správu projektu byl využit populární verzovací systém Git, respektive cloudové úložiště GitHub. Lepší schopnost práce s Gitem a celkově větší porozumění tomuto systému jsou dalším ovocem práce na tomto projektu, což je dobře, jelikož je Git skutečně mocný nástroj a v týmu se bez něj či jeho alternativ prakticky nelze obejít.

Během práce na projektu se náš tým setkal s mnohými problémy, které zahrnovaly jak špatnou práci s Gitem či problémy vzniklé nesprávným zacházením s dynamickou pamětí, tak i špatné rozvržení času. Skrze týmovou práci se nám však podařilo tyto problémy překonat a z chyb se také poučit (například právě v souvislosti s prací s Gitem). Díky tomu považujeme projekt za velmi přínosnou zkušenost a jsme rádi, že jsme mohli tuto práci absolvovat.