



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

1	Rozdělení práce mezi členy týmu	2
2	Diagram konečného automatu	3
3	LL-gramatika	4
4	LL-tabulka	6
5	Precedenční tabulka	8
6	Členění implementačního řešení	9
7	Závěr	11

Rozdělení práce mezi členy týmu

Antonín Jarolím - syntaktický analyzátor top-down, generace kódu, LL-tabulka, gramatika

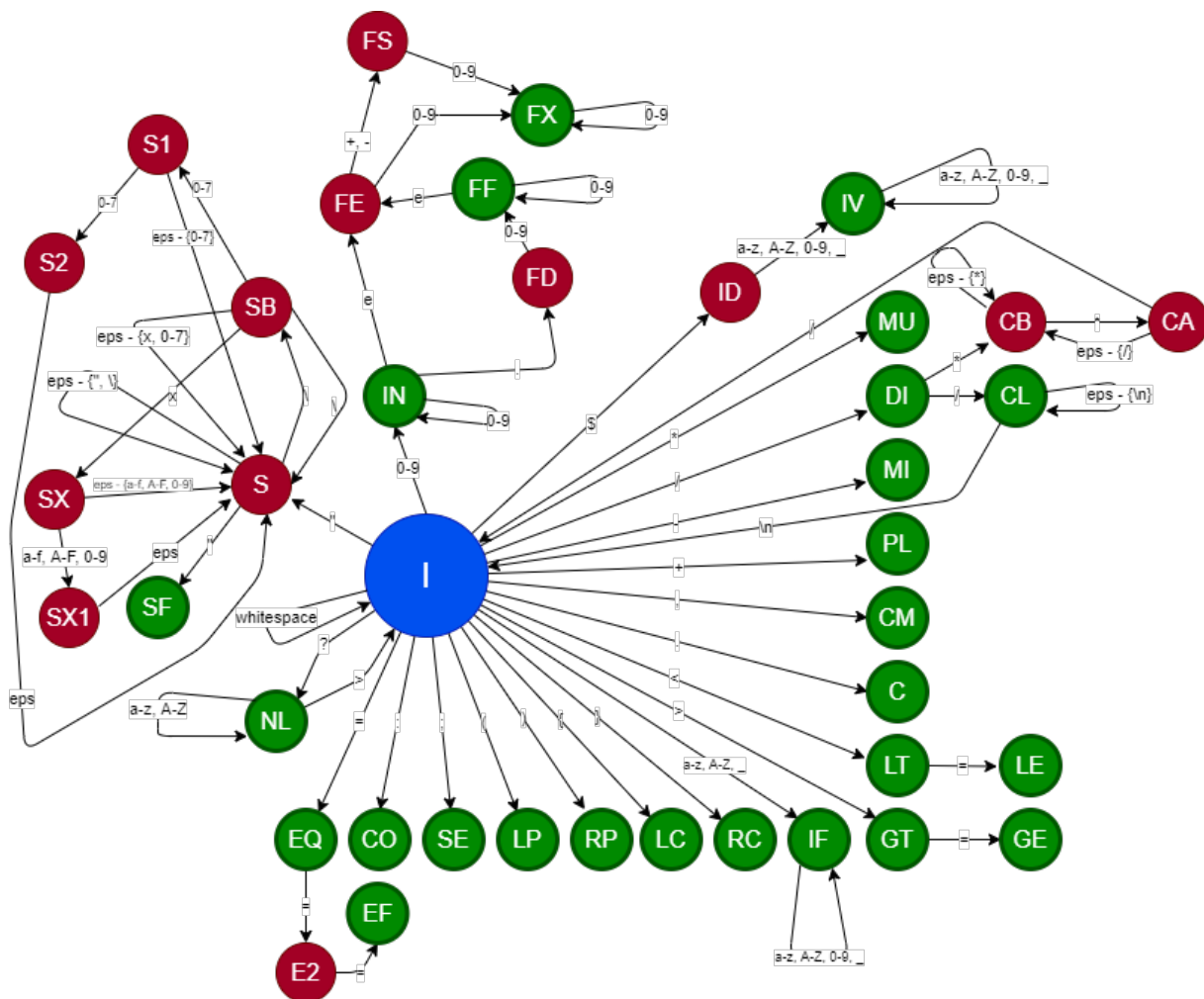
Jakub Vlk - syntaktický analyzátor bottom-up, generace kódu, precedenční tabulka, gramatika

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	E2 = eq_2_s
S = string_lit_s	N1 = not_eq_1_s
SF = string_lit_f_s	N2 = not_eq_2_s
IN = integer_lit_f_s	NF = not_eq_f_s
FD = float_lit_dot_s	SB = string_lit_backslash_s
FE = float_lit_e_s	SX = string_lit_backslash_x_s
FS = float_lit_sign_s	SX1 = string_lit_backslash_x_1_s
FX = float_lit_exp_f_s	S1 = string_lit_backslash_1_s
FF = float_lit_f_s	S2 = string_lit_backslash_2_s
IF = identifier_func_f_s	LP = left_par_f_s
ID = identifier_var_dollar_s	RP = right_par_f_s
IV = identifier_var_f_s	CL = com_line_f_s
MU = multiplication_f_s	CB = com_block_s
DI = division_f_s	CA = com_block_ast_s
PL = plus_f_s	LC = left_curly_f_s
MI = minus_f_s	RC = right_curly_f_s
C = concatenation_f_s	EQ = equals_f_s
LT = lesser_than_f_s	CO = colon_f_s
LE = lesser_eq_f_s	SE = semicolon_f_s
GT = greater_than_f_s	CM = comma_f_s
GE = greater_eq_f_s	NL = null_f_s
EF = eq_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	curlyBraceRight	functionKey
\$	$S ::= \text{ProgramBody } \$$	$S ::= \text{ProgramBody } \$$		$S ::= \text{ProgramBody } \$$
ProgramBody	$\text{ProgramBody} ::= \epsilon$	$\text{ProgramBody} ::= \text{Command ProgramBody}$		$\text{ProgramBody} ::= \text{FceDefine ProgramBody}$
Command		$\text{Command} ::= \text{semicolon}$		
FceDefine				$\text{FceDefine} ::= \text{FceHeader curlyBraceLeft FunctionBody curlyBraceRight}$
FceHeader				$\text{FceHeader} ::= \text{functionKey identifierFunc leftPar FunctionDeclareParams rightPar colon FuncReturnColonType}$
ElseCond	$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \epsilon$
ReturnExp		$\text{ReturnExp} ::= \epsilon$		
FunctionBody		$\text{FunctionBody} ::= \text{Command FunctionBody}$	$\text{FunctionBody} ::= \epsilon$	

	identifierFunc	leftPar	rightPar	comma
\$	$S ::= \text{ProgramBody } \$$	$S ::= \text{ProgramBody } \$$		
ProgramBody	$\text{ProgramBody} ::= \text{Command ProgramBody}$	$\text{ProgramBody} ::= \text{Command ProgramBody}$		
Command	$\text{Command} ::= \text{FceCall}$	$\text{Command} ::= \text{Exp semicolon}$		
FunctionDeclareParams			$\text{FunctionDeclareParams} ::= \epsilon$	
CommaOrEpsParams			$\text{CommaOrEpsParams} ::= \epsilon$	$\text{CommaOrEpsParams} ::= \text{comma DeclareParam CommaOrEpsParams}$
A	$\text{FceCall} ::= \text{identifierFunc leftPar FirstFceParam rightPar}$			
FirstFceParam			$\text{FirstFceParam} ::= \epsilon$	
CommaOrEpsParam			$\text{CommaOrEpsParam} ::= \epsilon$	$\text{CommaOrEpsParam} ::= \text{comma Statement CommaOrEpsParam}$
DefVarAss	$\text{DefVarAss} ::= \text{FceCall semicolon}$	$\text{DefVarAss} ::= \text{Exp semicolon}$		
ElseCond	$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \epsilon$		
ReturnExp		$\text{ReturnExp} ::= \text{Exp}$		
FunctionBody	$\text{FunctionBody} ::= \text{Command FunctionBody}$	$\text{FunctionBody} ::= \text{Command FunctionBody}$		

	identifierVar	voidKey	floatLiteral	stringLiteral	integerLiteral
\$	$S ::= \text{ProgramBody } \$$		$S ::= \text{ProgramBody } \$$	$S ::= \text{ProgramBody } \$$	$S ::= \text{ProgramBody } \$$
ProgramBody	$\text{ProgramBody} ::= \text{Command ProgramBody}$		$\text{ProgramBody} ::= \text{Command ProgramBody}$	$\text{ProgramBody} ::= \text{Command ProgramBody}$	$\text{ProgramBody} ::= \text{Command ProgramBody}$
Command	$\text{Command} ::= \text{DeclareVariable}$ $\text{Command} ::= \text{Exp semicolon}$		$\text{Command} ::= \text{Exp semicolon}$	$\text{Command} ::= \text{Exp semicolon}$	$\text{Command} ::= \text{Exp semicolon}$
FuncReturnColonType		$\text{FuncReturnColonType} ::= \text{voidKey}$			
FirstFceParam	$\text{FirstFceParam} ::= \text{Statement CommaOrEpsParam}$		$\text{FirstFceParam} ::= \text{Statement CommaOrEpsParam}$	$\text{FirstFceParam} ::= \text{Statement CommaOrEpsParam}$	$\text{FirstFceParam} ::= \text{Statement CommaOrEpsParam}$
Statement	$\text{Statement} ::= \text{identifierVar}$		$\text{Statement} ::= \text{floatLiteral}$	$\text{Statement} ::= \text{stringLiteral}$	$\text{Statement} ::= \text{integerLiteral}$
DeclareVariable	$\text{DeclareVariable} ::= \text{identifierVar equals DefVarAss}$				
DefVarAss	$\text{DefVarAss} ::= \text{Exp semicolon}$		$\text{DefVarAss} ::= \text{Exp semicolon}$	$\text{DefVarAss} ::= \text{Exp semicolon}$	$\text{DefVarAss} ::= \text{Exp semicolon}$
ElseCond	$\text{ElseCond} ::= \epsilon$		$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \epsilon$
ReturnExp	$\text{ReturnExp} ::= \text{Exp}$		$\text{ReturnExp} ::= \text{Exp}$	$\text{ReturnExp} ::= \text{Exp}$	$\text{ReturnExp} ::= \text{Exp}$
FunctionBody	$\text{FunctionBody} ::= \text{Command FunctionBody}$		$\text{FunctionBody} ::= \text{Command FunctionBody}$	$\text{FunctionBody} ::= \text{Command FunctionBody}$	$\text{FunctionBody} ::= \text{Command FunctionBody}$

	nullKey	stringNullKey	floatNullKey	intNullKey
\$	$S ::= \text{ProgramBody } \$$			
ProgramBody	$\text{ProgramBody} ::= \text{Command ProgramBody}$			
Command	$\text{Command} ::= \text{Exp semicolon}$			
FunctionDeclareParams		$\text{FunctionDeclareParams} ::= \text{DeclareParam CommaOrEpsParams}$	$\text{FunctionDeclareParams} ::= \text{DeclareParam CommaOrEpsParams}$	$\text{FunctionDeclareParams} ::= \text{DeclareParam CommaOrEpsParams}$
DeclareParam		$\text{DeclareParam} ::= \text{DataType identifierVar}$	$\text{DeclareParam} ::= \text{DataType identifierVar}$	$\text{DeclareParam} ::= \text{DataType identifierVar}$
FuncReturnColonType		$\text{FuncReturnColonType} ::= \text{DataType}$	$\text{FuncReturnColonType} ::= \text{DataType}$	$\text{FuncReturnColonType} ::= \text{DataType}$
DataType		$\text{DataType} ::= \text{stringNullKey}$	$\text{DataType} ::= \text{floatNullKey}$	$\text{DataType} ::= \text{intNullKey}$
DefVarAss	$\text{DefVarAss} ::= \text{Exp semicolon}$			
ElseCond	$\text{ElseCond} ::= \epsilon$			
ReturnExp	$\text{ReturnExp} ::= \text{Exp}$			
FunctionBody	$\text{FunctionBody} ::= \text{Command FunctionBody}$			

	stringKey	floatKey	intKey
FunctionDeclareParams	$\text{FunctionDeclareParams} ::= \text{DeclareParam CommaOrEpsParams}$	$\text{FunctionDeclareParams} ::= \text{DeclareParam CommaOrEpsParams}$	$\text{FunctionDeclareParams} ::= \text{DeclareParam CommaOrEpsParams}$
DeclareParam	$\text{DeclareParam} ::= \text{DataType identifierVar}$	$\text{DeclareParam} ::= \text{DataType identifierVar}$	$\text{DeclareParam} ::= \text{DataType identifierVar}$
FuncReturnColonType	$\text{FuncReturnColonType} ::= \text{DataType}$	$\text{FuncReturnColonType} ::= \text{DataType}$	$\text{FuncReturnColonType} ::= \text{DataType}$
DataType	$\text{DataType} ::= \text{stringKey}$	$\text{DataType} ::= \text{floatKey}$	$\text{DataType} ::= \text{intKey}$

	ifKey	elseKey
\$	$S ::= \text{ProgramBody } \$$	
ProgramBody	$\text{ProgramBody} ::= \text{Command ProgramBody}$	
Command	$\text{Command} ::= \text{Condition}$	
Condition	$\text{Condition} ::= \text{ifKey leftPar Exp rightPar curlyBraceLeft FunctionBody curlyBraceRight ElseCond}$	
ElseCond	$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \text{elseKey curlyBraceLeft FunctionBody curlyBraceRight}$
FunctionBody	$\text{FunctionBody} ::= \text{Command FunctionBody}$	

	whileKey	returnKey
S	$S ::= \text{ProgramBody } \$$	$S ::= \text{ProgramBody } \$$
ProgramBody	$\text{ProgramBody} ::= \text{Command ProgramBody}$	$\text{ProgramBody} ::= \text{Command ProgramBody}$
Command	$\text{Command} ::= \text{While}$	$\text{Command} ::= \text{Return}$
ElseCond	$\text{ElseCond} ::= \epsilon$	$\text{ElseCond} ::= \epsilon$
While	$\text{While} ::= \text{whileKey leftPar Exp rightPar curlyBraceLeft FunctionBody curlyBraceRight}$	
Return		$\text{Return} ::= \text{returnKey ReturnExp semicolon}$
FunctionBody	$\text{FunctionBody} ::= \text{Command FunctionBody}$	$\text{FunctionBody} ::= \text{Command FunctionBody}$

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 řídicí 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 relativně snadné pochopitelnosti principu jeho fungování.

Analýzátor využívá dynamicky ukládanou tabulku vytvořenou voláním funkce `createLLTable()` v souboru *LLTable.c*. Ta volá staticky zapsanou sadu funkcí, které vkládají pravidla do tabulky. Nabízí poté rozhraní, které voláním funkce `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 je na vrcholu zásobníku výraz, předává řízení precedenčnímu analyzátoru. Pro správné zpracování se zde používá funkce `ungetToken()`.

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. Zásobník, který je použit pro implementaci, obsahuje postupně se zpracovávající se výraz. Výraz je posloupnost tokenů, které jsou postupně měněny na neterminály podle precedenční tabulky. Pokud není pravidlo v precedenční tabulce nalezeno, je vrácena chyba a překlad se ukončí. Takto se pokračuje, dokud není zásobník prázdný. Poté je řízení předáno zpět syntaktickému analyzátoru.

LL-tabulka je v našem případě implementována jako dvojrozměrné pole. Data jsou do tabulky vkládána za běhu pomocí funkce `insertMember()`, která vloží pravidlo na příslušné místo v tabulce. Kostrou LL-tabulky je zdrojový soubor `LLtable.c` a hlavičkový soubor `LLtable.h`.

Gramatika gramatika je formalismus, o který se opírá celý překlad. Celá gramatika je umístěna v souboru `LLgrammar.txt`. V průběhu práce na projektu jsme gramatiku průběžně vylepšovali. Tabulka je poměrně obsáhlá a vyskytuje se v ní mnoho pravidel, aby mohla zachytit maximální množství chyb.

Generace kódu je zahájena sémantickou akcí, která je zavolána při provedení derivace. Každá derivace může mít svou sémantickou akci. Data ze vstupního programu a další informace o literálech a proměnných jsou ukládány na spolu s instrukcí do seznamu uchovávacího vygenerovaný mezikód. Po převedení celého vstupního souboru je spuštěna analýza kódu. V této fázi jsou kontrolovány typy funkcí a výrazů v programu. Analýza může do mezikódu přidat další instrukce, které zefektivňují, kontrolují nebo přetypovávají výrazy. Další věc, o kterou se stará předzpracování, je přesouvání proměnných definovaných v cyklech před daný cyklus, aby zmíněné proměnné nebyly redefinovány. Po nalezení konce souboru je zahájena sémantická akce o nalezení konce souboru, což pro překladač znamená pokyn k zahájení předzpracování tak, jak je popsáno výše, a následnému převodu mezikódu do IFJcode22. Implementace této funkcionality se nachází v souborech s předponou "gen", jako například `gen_gen.c` apod.

Pomocné soubory: Projekt překladače obsahuje také zdrojový soubor `common.c` a jeho hlavičkový soubor `common.h`. Jde o pomocný soubor obsahující funkce a makra, která napomáhají příjemnějšímu programování, jednoduššímu výpisu ladicích výpisů a následnému jednoduchému přepnutí překladače do produkčního módu. Konkrétně tyto soubory obsahují např. pojmenování konstant či funkce, které převádějí výčtové typy na odpovídající řetězcovou reprezentaci. Taktéž se zde nachází funkce pro ukončení programu v případě chyby, která pro účely ladění vypíše, ze kterého řádku byl překlad ukončen a následně ukončí překlad s očekávaným návratovým kódem. Pro účely testování se zde nacházejí redefinice různých funkcí, tak aby bylo možné testovat jednotlivé části samostatně bez ohledu na implementaci ostatních částí.

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.