1.12.2017 v Brně

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu do předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ17

Tým 073, varianta I

Implementovaná rozšíření: BASE, FUNEXP

Seznam autorů:

Dvořák Martin, xdvora2l, 25% - vedoucí

Dvořák Jan, xdvora2m, 25%

Halva Vladislav, xhalva04, 25%

Hromádka Vojtěch, xhroma13, 25%

Obsah

1. Úvod

Tato dokumentace se zabývá vývojem a implementací překladače imperativního jazyka IFJ17, který je zjednodušenou podmnožinou jazyka FreeBASIC. Vybrali jsme si variantu zadání I, ve které je za úkol implementovat tabulku symbolů pomocí binárního vyhledávacího stromu.

Součástí dokumentace je LL-gramatika, LL-tabulka, precedenční tabulka, které jsou jádrem našeho syntaktického analyzátoru a diagram konečného automat, který specifikuje lexikální analyzátor.

2. Implementace částí

Celý projekt jsme řešili po částech:

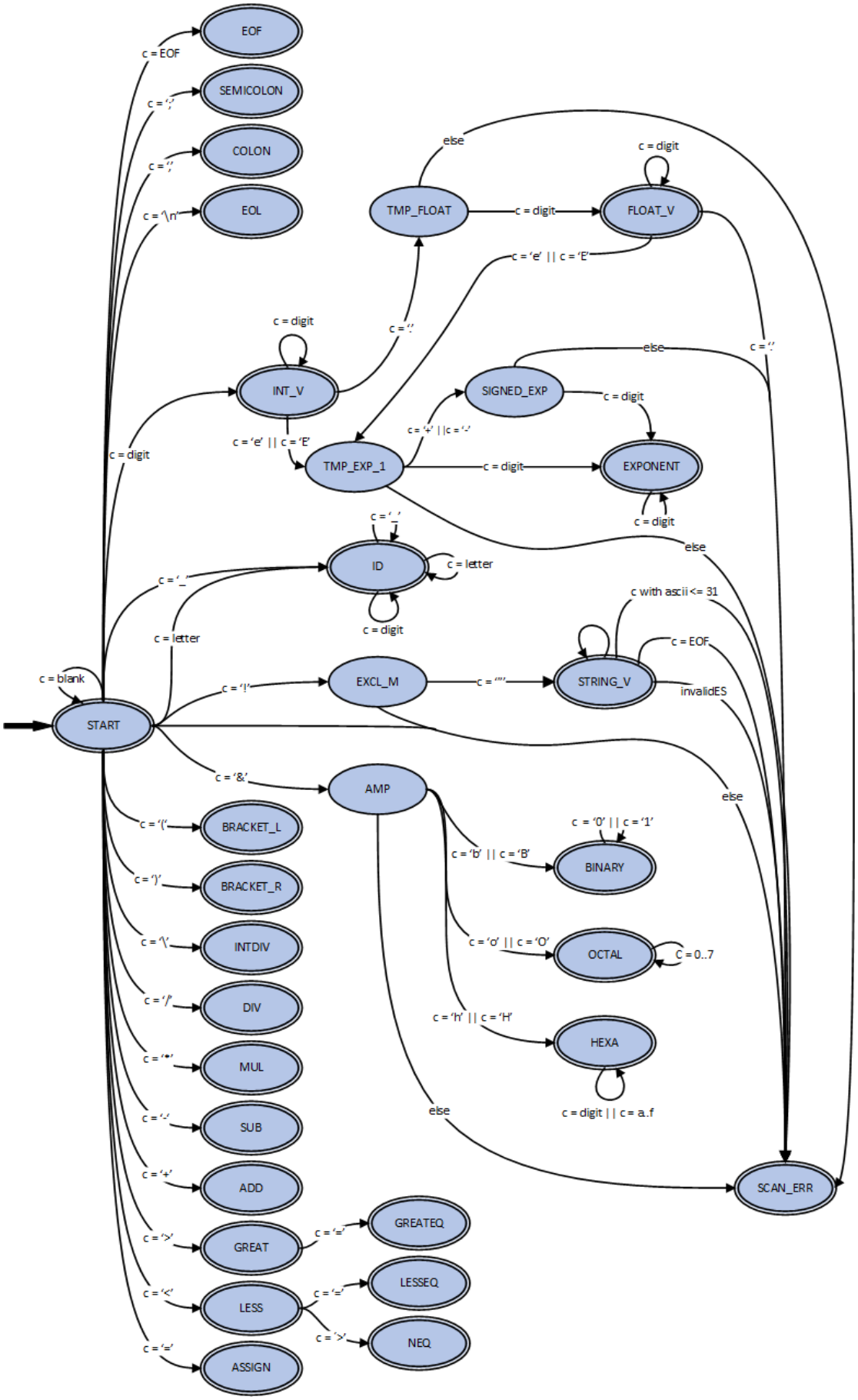
- Lexikální analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Generátor mezikódu IFJcode17

2.1. Lexikální analyzátor

Lexikální analyzátor (LA), také scanner, pracuje se zdrojovým textem tak, že odstraňuje bílé znaky a komentáře a určuje typ lexému. Lexikální analyzátor načítá zdrojový text znak po znaku a podle charakteru každého znaku mění stavy konečného automatu, kterým je specifikován. Typ lexému je určen podle stavu, ve kterém LA skončil. Pokud stav, ve kterém LA skončil, není koncový, vrátí návratovou hodnotu 1, která reprezentuje chybu v programu v rámci lexikální analýzy. Pokud LA skončí ve stavu ID (identifikátor), tak zkontroluje, zda se nejedná o klíčové nebo rezervované slovo, která jsou uložena v poli řetězců, je-li tomu tak, tak v tokenu vrací příslušné klíčové nebo rezervované slovo.

Funkce *get_next*, která je poskytována lexikálním analyzátorem, je volána syntaktickým analyzátorem a předává mu jednotlivé tokeny. LA používá zásobník pro výpočet hodnoty tokenů, který se alokuje a uvolňuje při každém volání funkce.

Oproti lexikálnímu analyzátoru popisovanému v přednáškách, kde je mezi dalšími činnostmi uvedena také komunikace s tabulkou symbolů, náš lexikální analyzátor s tabulkou symbolů nekomunikuje. Komunikuje s ní syntaktický, resp. sémantický analyzátor, jelikož zná širší kontext a tedy s ní může snáze pracovat (ukládat do ní hodnoty etc.).



Obrázek 1 - Diagram konečného automatu lexikálního analyzátoru

2.2. Syntaktický analyzátor

Syntaktický analyzátor je hlavní část celého překladače, používá a řídí všechny ostatní části překladače. Samotný syntaktický analyzátor je implementován pomocí rekurzivního sestupu shora dolů. Zpracování výrazů muselo být podle zadání pomocí precedenční tabulky (zpracování zdola nahoru). Syntaktický analyzátor za běhu provádí operace sémantické analýzy a generátoru mezikódu.

Hlavním komunikačním prvkem mezi Syntaktickým analyzátorem a Lexikálním analyzátor je datová struktura token. Do této struktury je uložen typ přečteného lexému a popřípadě další informace lexému (hodnota, jméno). Tuto datovou strukturu jsme implementovali "líně", tzn. že pokud není danou informaci třeba přepsat, zůstává v paměti stále uložena (př.: x as integer, syntaktickému analyzátoru jsou poslány postupně tři lexémy, ale i v posledním lexému je stále uloženo jméno proměnné), což nám ušetřilo mnoho dočasných proměnných.

V celkovém výsledku, především v návrhu funkcí pro syntaktický analyzátor je silně reflektována LL gramatika. Implementace obsahuje pouze pár pomocných funkcí, které z ní přímo nevychází. Nejdůležitější z těchto funkcí, je funkce zaručující bezpečné přepnutí z rekurzivního sestupu do precedenční analýzy. Je zde načten celý výraz, který má být zpracován, a je zjednodušen (zakódován na řetězec, kde jeden znak odpovídá lexému) aby nebylo obtížné ho zpracovat. Viz následující tabulka.

	*	/	M	+	-	=	N	L	G	S	R	(	)	i	$
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	>
M	<	<	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	<	>	<	<	<	<	<	<	<	<	>	<	>
-	<	<	<	>	<	<	<	<	<	<	<	<	>	<	>
=	<	<	<	<	<							<	>	<	>
N	<	<	<	<	<							<	>	<	>
L	<	<	<	<	<							<	>	<	>
G	<	<	<	<	<							<	>	<	>
S	<	<	<	<	<							<	>	<	>
R	<	<	<	<	<							<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	

Tabulka 1 - Precedenční tabulka

Legenda:

<	Uložení načtené hodnoty na zásobník.
>	Vyprázdnění nejvrchnějšího terminálu ze zásobníku.
M	Reprezentuje matematickou operaci modulo.
N,L,G,S,R	Reprezentují množinu logických operací.
i	Zastupuje identifikátory a konstanty.

Dále jsme se museli potýkat s problémem u rozšíření FUNEXP s možností volání funkcí ve výrazech. Nastával zde problém se závorkováním výrazů, nebylo jednoduše rozeznatelné, zda se jedná o závorku ukončující výčet parametrů funkce či závorku patřící k výrazu. Tuto problematiku řeší počítání levých a pravých závorek.

Dalším problémem bylo rozeznání, zda se jedná o identifikátor proměnné, nebo volání funkce. Musela být implementována další funkce, která zjednodušeně nahlédne na následující lexém, a podle toho rozeznává, zda se mají zpracovávat parametry funkce, či nikoliv.

Zpracování výrazu tedy probíhá za pomoci precedenční tabulky a jednoduchého zásobníku.

Pokud nastane chyba v syntaxi, nebo sémantice zpracovávaného kódu, je nastavena globální proměnná na správný chybový stav (implicitně je natavena syntaktická chyba) a je “probublána” chybová návratová hodnota z nejvíce zanořené části rekurze.

2.2.1. LL-gramatika

Implementace se striktně drží LL-gramatiky. Oproti základní verzi (základnímu zadání) zde dochází k nejednoznačnosti pouze u předzpracování výrazů, a řešení této problematiky je zmíněno výše. Každý neterminál, který je zmíněný v gramatice je reprezentován samostatnou funkcí.

```
1.<S> -> <F><M><F>
2.<F>-> epsilon
3.<M>-> epsilon

4.<M> -> SCOPE EOL <BODY> END SCOPE EOL
5.<F> -> DECLARE <FUNC_LINE> EOL <F>
6.<F> -> <FUNC_LINE> EOL <BODY> END FUNCTION EOL <F>

7.<FUNC_LINE> -> FUNCTION ID (<PARAM>) AS <TYPE> EOL

8.<PARAM> -> epsilon
9.<PARAM> -> ID AS <TYPE> <PARAMS_N>
10.<PARAMS_N> ->, ID AS <TYPE> <PARAMS_N>
11.<PARAMS_N> -> epsilon

12.<BODY> -> epsilon
13.<BODY> -> DIM ID AS <TYPE> <=> EOL <BODY>
14.<=> -> epsilon
15.<=> -> <EXP>
16.<EXP> -> ID(<PARAM_F>) <EXP>
17.<EXP> -> ( <EXP>
18.<EXP> -> CONSTANT <EXP>

19.<PARAM_F> -> epsilon
20.<PARAM_F> -> ID <PARAM_FN>
21.<PARAM_FN> -> ,ID <PARAM_FN>
22.<PARAM_FN> -> epsilon

23.<BODY> -> ID = <EXP> EOL <BODY>
24.<BODY> -> INPUT ID  EOL <BODY>
25.<BODY> -> PRINT <EXP>; <EXP_N> EOL <BODY>
26.<EXP_N> -> epsilon
27.<EXP_N> -> <EXP>; <EXP_N>

28.<BODY> -> IF <EXP> THEN EOL <BODY> ELSE EOL <BODY> END IF EOL <BODY>
29.<BODY> -> DO WHILE <EXP> EOL <BODY> LOOP EOL <BODY>
30.<BODY> -> RETURN <EXP> EOL <BODY>

31.<TYPE> -> INTEGER
32.<TYPE> -> FLOAT
33.<TYPE> -> STRING
```

LL-tabulka

LL	S	M	F	Body	Func_line	Param	Type	Param_N	=	Exp	Exp_N	Param_f	Param_fn
Scope	1	4	2										
End				12									
Declare	1	3	5										
Eol									14		26		
Function	1	3	6		7								
Id				23		9			15	16	27	20	
(									15	17	27		
)						8		11				19	22
As													
,								10					21
Dim				13									
=													
Input				24									
Print				25									
;													
If				28									
Then													
Else				12									
Do				29									
While													
Return				30									
Integer							31						
Dlouble							32						
String							33						
Constant									15	18	27	20	
$	1	3	2										

Legenda:

První sloupec reprezentuje neterminály a první řádek reprezentuje terminály.

Je zde zavedena mírná míra abstrakce, u čtení výrazů, jelikož se analyzátor přepíná do precedenčního zpracování a výrazy se před zpracovávají (viz výše). Je zde pomyslně předáno řízení a tabulkou se již nezpracovávají.

2.3. Sémantická analýza

Sémantický analyzátor poskytuje funkce volané syntaktickým analyzátorem, které vyhodnocují sémantickou správnost kódu. Mezi hlavní sémantické kontroly patří kontrola kompatibility datových typů, deklarací,definic proměnných a funkcí, následně také kontrola datové kompatibily mezi levou a pravou stranou při přiřazení. Při své činnosti používá tabulku symbolů.

Sémantický analyzátor disponuje funkcemi na vkládání a čtení z tabulky symbolů. Překladač využívá dvou tabulek symbolů. Označeny jsou jako “globální” a “lokální”. “Globální” tabulka se používá pro ukládání funkcí (definovaných, deklarovaných a vestavěných). Je zde velice důležitý příznak, zda je funkce pouze deklarována a volána. Tyto příznaky se kontrolují na úplném konci překladu. Tato “globální” tabulka symbolů (tj. binární strom) se maže pouze při ukočení práce celého překladače. Druhá tabulka symbolů je “lokální”. Funguje stejně jako “globální” tabulka, rozdíl mezi těmito tabulkami je, že do “lokální” jsou uloženy pouze proměnné dané funkce, a při ukončení funkce je tabulka symbolů vyprázdněna.

Sémantický analyzátor implementuje mimo jiné funkci, která kontroluje kompatibilitu přiřazení do proměnné. Tato funkce je volána při každém výskytu přiřazení a dalo by se říct, že patří do syntaktického analyzátoru navzdory tomu, že nereprezentuje žádný lexém. Tato funkce si ukládá datový typ levé části výrazu a následně čeká na zpracování pravé části, nakonec provede jejich vyhodnocení.

Pro sémantickou kontrolu výrazů jsme zvolili přístup, založený na čtení reprezentace výrazu v upravené polské (postfixové) notaci, která je generována, jako vedlejší produkt precedenčního zpracování výrazu. Dalším vedlejším produktem předzpracování výrazu je datová struktura “expr_operand”, která je dále popsána v části generování mezikódu. Sémantika z této struktury používá jednu položku, která říká, jaký datový typ má identifikátor, který se má zpracovávat. Kontrola je provedena pomocí zásobníku, kdy je po přečtení identifikátoru uložen jeho datový typ na zásobník. Po přečtení operátoru jsou na zásobníku dvě položky (datové typy obou operandů), kompatibilita jejich datový typů je zkontrolována (případné implicitní konverze jsou poznamenány do speciální datové struktury, která slouží pro generátor kódu), a následně je nahrán na zásobník výsledný datový typ. Výsledný datový typ výrazu, včetně implicitních konverzí je po zpracování celého výrazu na vrcholu zásobníku.

2.4. Generátor mezikódu

Součástí zadání byl také mezikód IFJcode17, který je cílovým jazykem našeho překladu. Překlad probíhá přímo z abstraktního syntaktického stromu, který je výstupem syntaktického, resp. sémantického analyzátoru.

Vzhledem k implementovanému rozšíření FUNEXP, kdy mohou být funkce volány ve výrazech a zároveň se mohou objevovat výrazy v parametrech funkcí jsou parametry funkcí předávány přes datový zásobník (čteny jsou z něho od posledního).

Všechny funkce jsou volány ještě před zpracováním výrazu, ve kterém se vyskytují. Vytvoří si lokální rámec, jsou jim předány parametry (z datového zásobníku), vykoná se tělo, a jejich návratové hodnoty (popř. i implicitní) jsou ponechány na datovém zásobníku. Po vykonání všech volání v rámci výrazu jsou návratové hodnoty uloženy do proměnných, deklarovaných pro návratové hodnoty funkcí na dočasném rámci, jejich identifikátory jsou vloženy do pole operandů (ve kterém jsou uloženy potřebné informace o všech operandech výrazu, jako typ, identifikátor, nebo hodnota, v pořadí, v jakém se ve výrazu vyskytují).

Následně se generuje kód výrazu pracující již pouze s polem operandů, polem operátorů (obsahujícím typ operátoru a příznaky nutnosti implicitního přetypování) a řetězcem reprezentujícím výraz v postfixové formě (používaný pro rozeznání, zda se má zpracovat operand, nebo operátor). Výsledek výrazu (hodnota) je ponechána na vrcholu datového zásobníku (díky tomu je možné, aby se výraz objevil v parametru funkce a dalo se k němu chovat stejně jako k jednoduché hodnotě).

V případě, že se ve výrazu objeví řetězcový literál, je volána funkce, která ho automaticky převede do formátu potřebného pro IFJcode17, tj. všechny znaky s ASCII hodnotou 0–32, nebo 92 převede na Escape sekvence.

Při volání jedné z vestavěných funkcí (Length, SubStr, Asc, Chr), se vloží část mezikódu reprezentující tuto funkci, chování je totožné s voláním uživatelských funkcí.

Kód podmíněného výrazu a cyklu je založen na negaci podmínky. Tedy u podmíněného výrazu se v případě splnění podmínky vykoná příslušná část kódu, a v případě, že podmínka splněna není, je generován podmíněný skok na alternativní (else) větev kódu. U cyklu je tomu podobně, při nesplnění podmínky je generován skok z cyklu.

2.5. Tabulka symbolů

Tabulka symbolů je řešená pomocí binárního vyhledávacího stromu. Všechny operace pro práci s tabulkou symbolů jsou implementovány rekurzivně (preferovali jsme čitelnost kódu, před časovou efektivitou, která je zde zanedbatelná). Časová složitost algoritmů je logaritmická.

Využívá se zde dvou datových struktur. První z nich reprezentuje uzel binárního stromu. Tudíž obsahuje dva ukazatele na další uzly ve stromu a vyhledávací klíč (jméno funkce, nebo proměnné - jediná užitečná informace). Druhá datová struktura obsahuje užitečný obsah uzlu. Zde je uchovávána informace o datovém typu proměnné. V případě vkládání funkcí jsou naplněny záznamy o tom, zda funkce byla definována a její seznam parametrů (kódovaný řetězec, kde každý znak znamená jeden parametr) a datovém typu funkce, je uložena návratová hodnota.

3. Práce v týmu

3.1. Rozdělení práce

- Dvořák Martin: vedoucí týmu, syntaktická analýza, sémantická analýza, tabulka symbolů
- Dvořák Jan: lexikální analyzátor
- Halva Vladislav: generátor mezikódu
- Hromádka Vojtěch: automatizované testy

3.2. Metodika vývoje softwaru

Při vývoji a implementaci našeho řešení jsme postupovali vývojovou metodikou iterace. První iterace trvala přibližně 4 týdny, za úkol bylo zvládnout od každé části alespoň částečnou funkčnost, kromě generátoru mezikódu. Druhá iterace spočívala v doplnění generátoru mezikódu a zároveň ladění chyb z první iterace. V následujících iteracích docházelo k odstraňování chyb, ladění programu, popř. implementaci rozšíření, obecně tyto iterace trvaly mnohem kratší dobu než první, to se opakovalo až do doby odevzdání projektu.

3.3. Automatizované testy

Testy jsme vytvářeli souběžně s vývojem a implementací projektu. Při první iteraci byly části testovány jednotlivě (lexikální analýza, syntaktická analýza, sémantická analýza). Během prvních týdnů jsme neměli mnoho testů, avšak i relativně malé množství pomohlo odhalit značné množství implementačních chyb.

V pozdějších iteracích, kdy byla implementace již částečně hotová a bylo třeba ji kontrolovat jako celek, byly vytvořeny automatizované testy ve skriptovacím jazyce BASH. Testy fungují na principu porovnávání návratových hodnot z překladače, interpretu a standardního výstupu s očekávanými hodnotami a vypíší, zda se program choval správně, či nikoliv.

3.4. Schůzky a komunikace v týmu

Schůzky probíhaly pravidelně každý týden, ne vždy se však účastnili všichni členové týmu. Jednalo se, jak o schůzky informativní (každý člen obeznamoval zbytek týmu o jeho činnosti daný týden na projektu), tak o schůzky produktivní, kdy jsme aktivně vymýšleli další postup vývoje překladače. Nejčastěji jsme diskutovali o komunikaci mezi jednotlivými částmi překladače, nebo nad návrhem použitých datových struktur.

Převážná část komunikace ale probíhala přes sociální sítě. Tuto formu komunikace jsme zvolili kvůli její největší výhodě, a to možnosti rychlých odpovědí, bez nutnosti být na jednom místě. V komunikaci mezi členy týmu nebyl žádný problém. Jediný problém občas spočíval ve špatně zformulované otázce či odpovědi.

3.5. Systém správy zdrojového kódu

Potřebovali jsme nějaký efektivní a jednoduchý způsob sdílení verzí zdrojových kódů a dalších informací týkajících se projektu. Jednotně jsme se shodli na použití webové služby GitHub, která podporuje vývoj softwaru za pomoci verzovacího nástroje Git.

Závěr

S projektem tohoto rozsahu se nikdo z nás doposud nesetkal. Řešení tohoto problému nám dalo spoustu nových a užitečných zkušeností ohledně práce v týmu a znalostí týkajících se problematiky překladačů. Očekávali jsme, že řešení tohoto projektu bude časově náročné, proto jsme s ním začali poměrně brzy v průběhu semestru. Díky včasnému zahájení prací jsme bez velkých komplikací stihli první pokusné odevzdání, které dopadlo relativně dobře. Do druhého pokusného odevzdání jsme stihli opravit některé chyby a dosáhli tak ještě lepšího výsledku.