VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

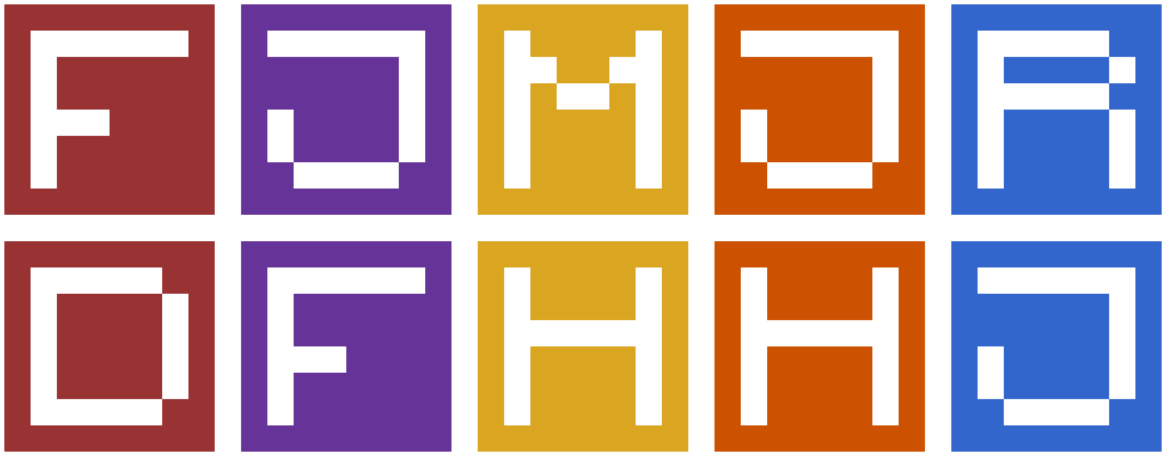
Dokumentace projektu do předmětů IFJ a IAL

Interpret jazyka IFJ16

Tým 067, varianta a/2/I

Frýz Jakub (vedoucí)	xfryzj01	20 %
Dostálík Filip	xdosta46	15 %
Hrabovský Michal	xhrabo08	25 %
Hud Jakub	xhudja00	20 %
Janík Roman	xjanik20	20 %

11. prosince 2016



Obsah

1 Úvod

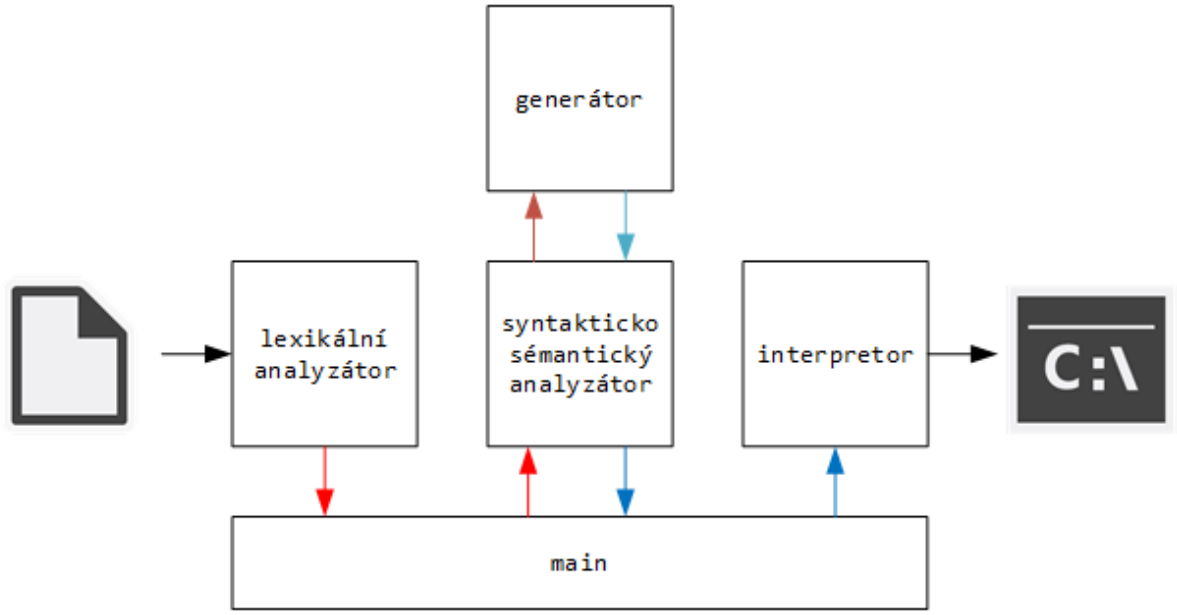
Tato dokumentace popisuje vývoj a implementaci interpretu jazyka IFJ16, jenž je podmnožinou jazyka Java. Úlohou tohoto interpretu je kontrola vstupního zdrojového kódu a jeho interpretaci v případě, že je všechno v pořádku, jinak informuje o chybách.

Dokumentace je členěna na kapitoly a jejich podkapitoly, které popisují jednotlivé části interpretu, způsob implementace či použité algoritmy.

Na závěr je shrnutí naší práce, statistika a použité nástroje.

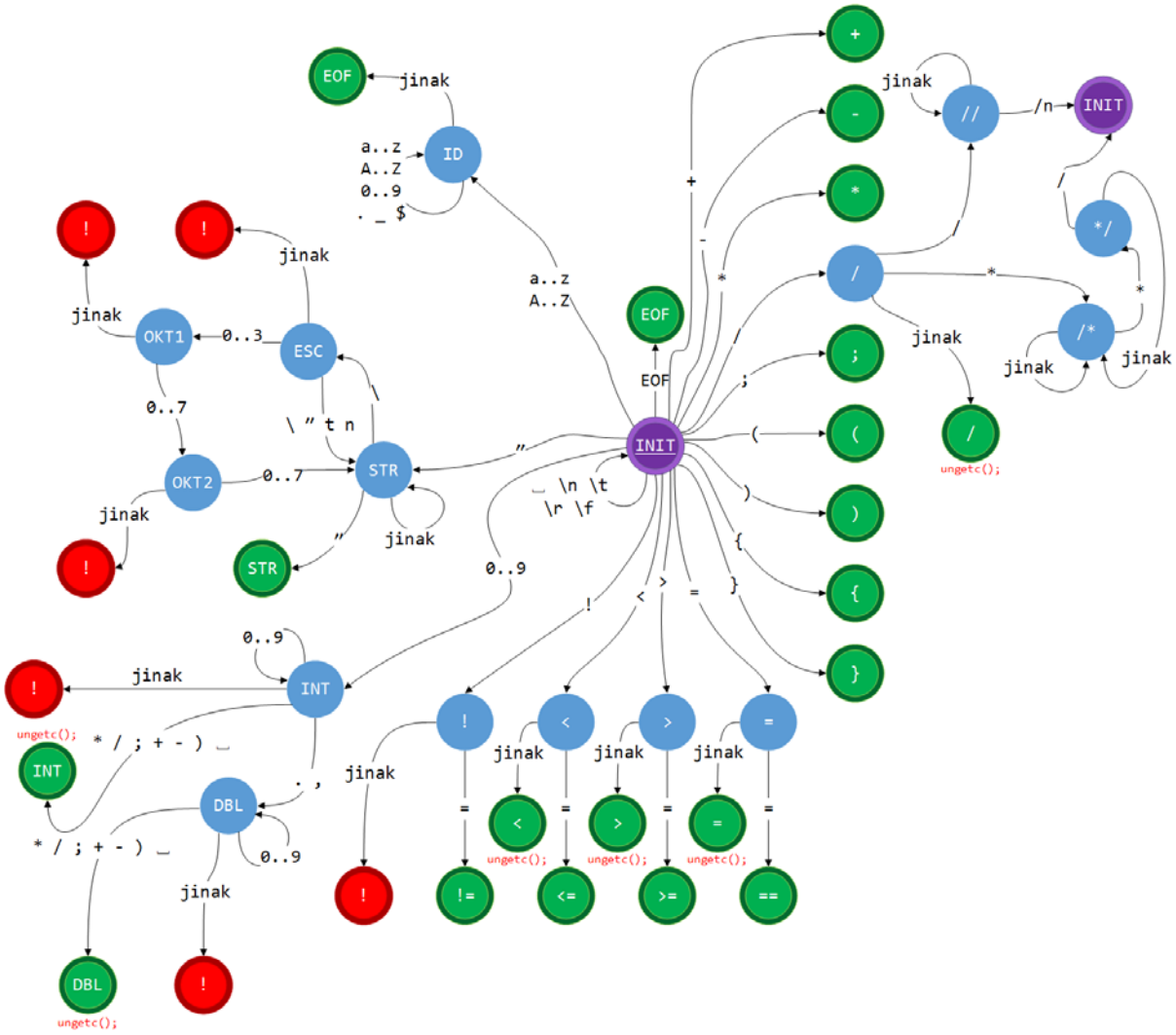
2 Struktura projektu

Projekt jsme rozdělili do 4 hlavních částí: **lexikální analyzátor**, **syntakticko-sémantický analyzátor**, **generátor 3-adresného kódu** a nakonec samotný **interpret**. Jednotlivé části budou podrobně popsány v následujících podkapitolách.



Na začátek je zde diagram zobrazující tok dat v celém projektu. Z diagramu lze vyčíst, že je zapotřebí vstupního souboru, který lexikální analyzátor hned zpracuje a vytvoří z něj frontu tokenů, nenajde-li chybu. V opačném případě o chybě informuje. Frontu tokenů následně zpracovává syntakticko-sémantický analyzátor. Ten kontroluje syntaxi a poté sémantiku. Možnou chybu opět ohlásí. Pokud je vše v pořádku, za pomoci generátoru generuje instrukce, které jsou ukládány do listu instrukcí. Výsledný list nakonec projede interpret a potřebné informace vypíše do konzole.

2.1 Lexikální analyzátor



Úlohou lexikálního analyzátoru (neboli scanneru) spočívá v načtení zdrojového souboru a jeho převedení na základě určitých pravidel na lexémy. Ty jsou prakticky reprezentovány jako tokeny a pak vkládány do fronty, která je následně zpracovávaná syntaktickým analyzátorem.

Najde-li lexikální analyzátor neznámý lexém, ohlásí to chybou.

Implementace lexikálního analyzátoru byla provedena pomocí konečného stavového automatu, jehož diagram je výše zobrazen.

2.2 Syntakticko-sémantický analyzátor

Syntakticko-sémantický analyzátor je srdcem celého interpretu. Je to taky velký celek, který jde rozdělit na tři hlavní podcelky: **syntaktický analyzátor**, **sémantický analyzátor** a **precedenční analyzátor**.

2.2.1 Syntaktický analyzátor

Kontroluje syntaktickou správnost programu. To, co je syntakticky správně určuje LL-gramatika.

```
          <body>  epsilon
          <body>  <class>
         <class>  static type class_name (<args>) { <com> }
          <args>  epsilon
          <args>  id <args_n>
        <args_n>  epsilon
        <args_n>  , id <args_n>
          <func>  ifj16.func_name( <id> );
          <func>  func_name( <ids> );
           <ids>  epsilon
           <ids>  id <ids_n>
         <ids_n>  epsilon
         <ids_n>  , id <ids_n>
           <com>  epsilon
           <com>  if ( <con> ) { <com> } else { <com> } <com>
           <com>  while ( <con> ) { <com> }
           <com>  id = <ass>;
           <com>  type id;
           <com>  type id = <ass>;
           <com>  return id;
           <ass>  id
           <con>  id <comp> id
<comp> = { >, >= , <, <=, ==, != }
```

2.2.2 Sémantický analyzátor

Kontroluje sémantickou správnost programu.

2.2.3 Precedenční analyzátor

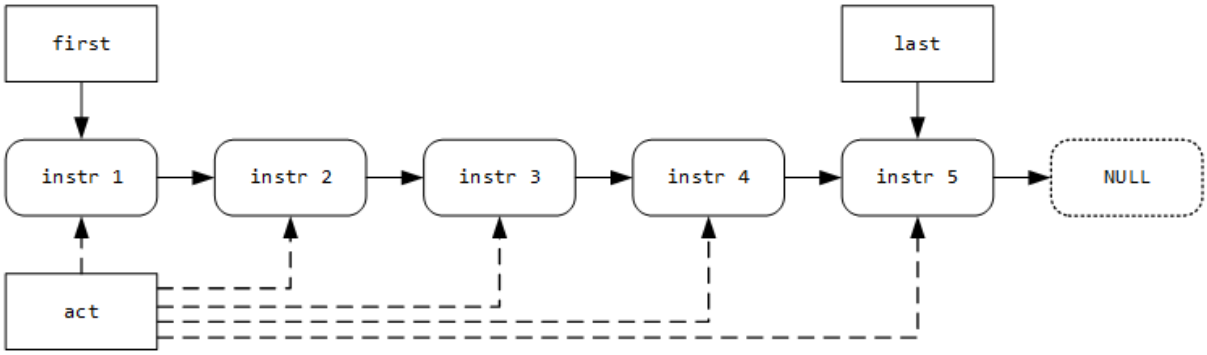
	==	!=	>	<	<=	>=	+	-	/	*	(	)	i	IG	DB	ST	$
==	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
!=	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
>	<	<	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
<	<	<	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
<=	<	<	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
>=	<	<	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>
+	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<	>
-	<	<	>	>	>	>	>	>	<	<	<	>	<	<	<	<	>
/	<	<	>	>	>	>	<	<	>	>	<	>	<	<	<	<	>
*	<	<	>	>	>	>	<	<	>	>	<	>	<	<	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	==	<	<	<	<	>
)	>	>	>	>	>	>	>	>	>	>	-1	>	-1	-1	-1	-1	>
i	>	>	>	>	>	>	>	>	>	>	-1	>	-1	-1	-1	-1	>
ID	>	>	>	>	>	>	>	>	>	>	-1	>	-1	-1	-1	-1	>
DB	>	>	>	>	>	>	>	>	>	>	-1	>	-1	-1	-1	-1	>
ST	>	>	>	>	>	>	>	>	>	>	-1	>	-1	-1	-1	-1	>
$	<	<	<	<	<	<	<	<	<	<	<	-1	<	<	<	<	-1

(IG = integer, DB = double, ST = string)

Tabulka precedenční analýzy

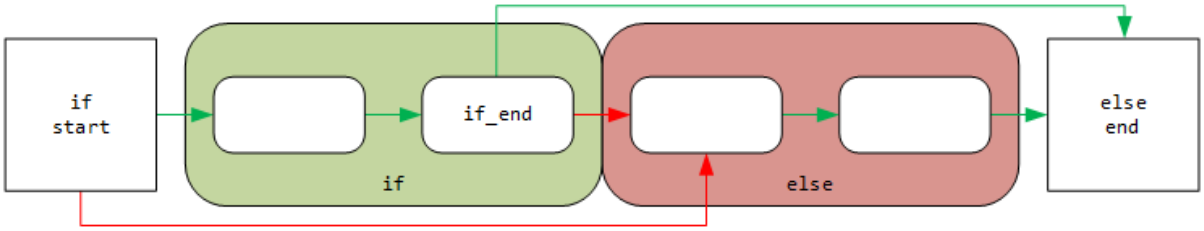
2.3 Generátor 3-adresného kódu

Generátor se skládá ze tří prvků: **generační funkce, listu instrukcí** a **zásobníku adres na instrukce.**



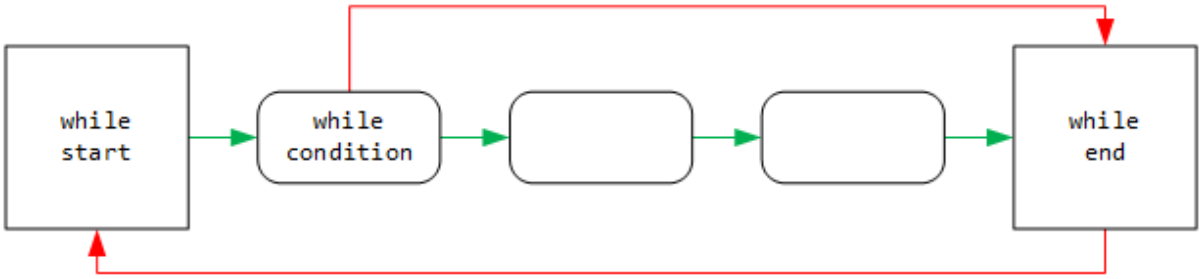
Generační funkce je vyvolávána sémantickým analyzátorem vždy, když zpracuje kousek kódu. Ta získá jaký typ instrukce má vygenerovat, adresy dvou operandů adresu s výsledkem. Vygenerované instrukce vkládá do listu. Interpret se poté posouvá v listu pomocí ukazatele *next*. V případě složitějších struktur, jako jsou if-else a while cyklus, generační funkce navíc tvoří speciální ukazatele, které usnadňuje pohyb interpretu v listu.

Ukazatele na začátky funkcí jsou ukládány i do binárního stromu.



(zelené šipky – ukazatel next, červené šipky – speciální ukazatel *nope*)

Takhle je generována struktura if-else do listu. V prvku if_start je uložena adresa na podmínku. Interpret, v případě, že vyhodnotí podmínku negativně, přesune se pomocí ukazatele *nope*, a pak pokračuje zas přes ukazatel *next*. Díky tomu není třeba pomocného zásobníku.



A zde je generace while cyklu. V prvku *while_condition* je uložena podmínka, pokud interpret vyhodnotí podmínku negativně, přeskočí na *while_end* přes *nope* a z něj na další prvek, a pokud pozitivně, posouvá se přes prvky až na *while_end*, odkud přeskočí po *nope* na *while_start* a může znovu vyhodnotit podmínku.

2.4 Interpret

Interpret má na starost vykonávání programu. Provádí také kontrolu typů-

Z naší speciální struktury získá ukazatel na funkci *run()* a provede všechny jednoduché operace na globální úrovni. Pak se provádí interpretace tří-adresného kódu uloženého v listu vytvořeného generátorem. Pokud se narazí na volání vestavěná funkce, tak ji zavolá a výsledek uloží na místo určené v listu. V případě volání uživatelem definované funkce, uloží se do zásobníku adresa, kam se má interpret vrátit po provedení funkce, zkopírují se lokální proměnné, které jsou v binárním stromu. Poté se provede ona funkce. Při návratu se přepíšou lokální proměnné a odstraní se adresa ze zásobníku.

3 Řešení vybraných algoritmů

V rámci implementace interpretu měl každý tým řešit konkrétní problémy řešit za pomoci různých algoritmů:

1. **Knuth-Morris-Prattův algoritmus** pro vyhledávání podřetězce v řetězci
2. **heapsort** pro řazení znaků v řetězci
3. **binární vyhledávací strom** pro řešení tabulky symbolů

3.1 Knuth-Morris-Prattův algoritmus

Knuth-Morris-Prattův algoritmus je algoritmus pro vyhledávání podřetězce v řetězci zleva doprava. Má lineární časovou složitost $O(m+n)$.

Před hledáním je potřeba spočítat tzv. **prefix funkci** (u nás *kmpgraph*) z hledaného vzoru, která udává, kolik předchozích porovnání můžeme použít znovu. Což jinými slovy znamená, o kolik prvků můžeme posunout hledaný vzor vůči řetězci. Díky tomu se nemusíme při prohledávání řetězce vracet zpět. Časová složitost prefix funkce je $O(m)$, kde *m* je délka vzoru.

Teď už můžeme postupně porovnávat prvky vzoru i řetězce. Pokud se znaky shodují, inkrementujeme index a dále porovnáváme, pokud se neshodují, nahradíme index hodnotou, která je v **prefix funkci**, a pokračujeme v porovnávání, dokud vzor nenajdeme nebo se nedostaneme na konec řetězce, v tom případě vrátíme -1.

3.2 Heap sort

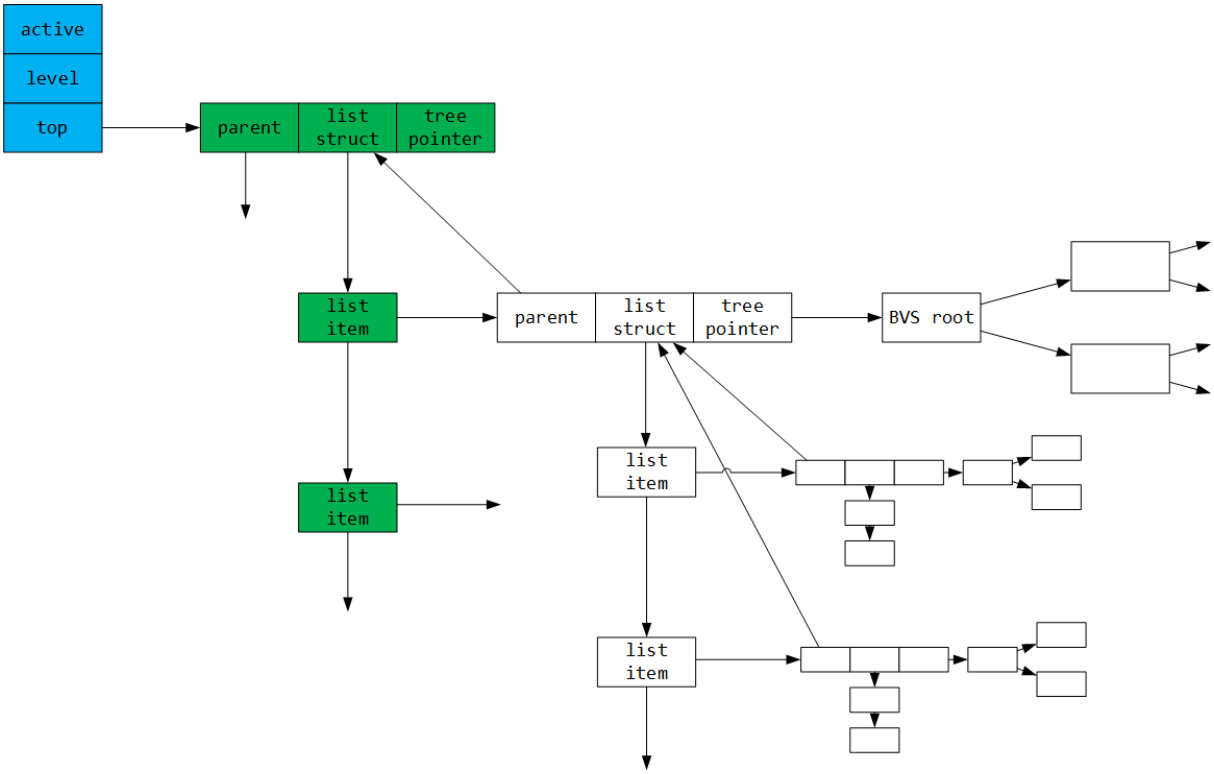
Heap sort, neboli řazení haldou, patří mezi chytré řadící algoritmy. Má zaručenou složitost $O(n * \log n)$ a konstantní nároky na paměť.

Staví na Selection sortu, který vyhledával prvky tím, že v každém cyklu projel nesetříděnou část pole, našel minimum (nebo maximu) a vložil jej na začátek (konec). Heap sort se celý proces snaží zrychlit pomocí *haldy*. Halda je binární strom se speciálními vlastnostmi, kde v kořenu bude vždy uloženo minimum (maximum). Tento strom však není vůbec třeba tvořit, neboť se na naše tříděné pole můžeme koukat jako na haldu, tím pádem lze celý proces provést na místě.

3.3 Binární vyhledávací strom

Binární strom se používá pro realizaci tabulky symbolů. Využíváme nerekurzivní verzi stromu.

Do binárního stromu se ukládají: třídy, funkce, globální proměnné a lokální proměnné. Jednotlivé stromy se následně uloží do speciální struktury, která je zobrazena pod tímto textem.



Takto vypadá struktura, do které se ukládají jednotlivé binární stromy.

Modrá struktura: active = aktivní prvek, level = úroveň zanoření

Část pro binární stromy: na první úrovni (body) mrtvý, na druhé (class) uchovává globální proměnné, na třetí (funkce) lokální proměnné (na čtvrté (bloky) by to byly proměnné platné v bloku příkazů)

4 Vývoj projektu

Zde bude popsáno, jak probíhal vývoj projektu, rozdělení části mezi členy a jakých bylo použito prostředků k dosažení cíle.

4.1 Rozdělení práce

Filip Dostálík

- lexikální analyzátor

Jakub Frýz

- fronta pro lexikální analýzu
- generátor + list a zásobník potřebný pro chod generátoru
- dokumentace, prezentace

Jakub Hud

- binární vyhledávací strom
- interpret

Michal Hrabovský

- syntakticko-sémantický analyzátor + precedenční analýza
- interpret
- provázání všech částí projektu

Roman Janík

- vestavěné funkce
- Knuth-Morris-Prattův algoritmus
- řadící algoritmus Heapsort

4.2 Použité nástroje

Komunikace

- Slack
- Facebook

Správa repositáře

- GitHub for Windows
- Git for Windows

Programování

- Code::Blocks (mingw)
- Linux subsystem for Windows (gcc)
- Notepad++
- Visual Studio Code

+ Microsoft Word, PowerPoint a Visio

5 Závěr

Díky tomuto projektu jsme se naučili mnohému, ať už šlo o týmovou spolupráci a komunikaci, správu Git repositáře či organizaci našeho volného času a v neposledním případě rozšíření našich dovedností v jazyce C.

Snažili jsme se projekt dodělat do pokusného odevzdání, to se nám bohužel nepodařilo ani napodruhé. Nepodařilo se nám ani implementovat žádná rozšíření.

I přes nezdar jsme však byli schopni projekt dodělat do vcelku funkční podoby.

5.1 Statistika

Statistika repositáře:

Název projektu:	2016-IFJ-IAL-Projekt
Věk repositáře:	77 dní, 44 aktivních (57,14 %)
Počet souborů:	58
Celkový počet řádků kódu:	5861
	10089 přidaných
	4228 smazaných
Celkový počet commitů:	560
	12,7 commitů za aktivní den
	7,3 commitů za všechny dny
Celkový počet pull requestů:	100

(platí ke dni 11. 12. 2016 16:08)

Statistika souborů (mimo souborů nastavení):

typ souboru	soubory	prázdné	komentáře	Kód
C	9	815	306	3267
C header	10	116	188	363
Java	4	1	3	107
Markdown	1	4	0	15
make	1	7	9	12
SUM	**25**	**943**	**506**	**3764**

(platí ke dni 11. 12. 2016 16:08)

5.2 Zdroje

- přednášky a podklady k předmětu IFJ a IAL
- algoritmy.net
- ITnetwork.cz
- Wikipedia.org (převážně anglická)