FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Formální jazyky a překladače / Algoritmy

Interpret jazyka IFJ16

Tým 059, varianta b/2/I

Rozšíření

Autoři

Jan Hrbotický (vedoucí)	xhrbot01	24%
Dominik Skála	xskala11	19%
Milan Hruban	xhruba08	19%
David Hél	xhelda00	19%
Martin Hons	xhonsm00	19%

Obsah

1 Úvod

Tato dokumentace popisuje implementaci imperativního jazyka IFJ16, jenž byl zadán jako týmový projekt do předmětů Formální jazyky a překladače (IFJ) a Algoritmy (IAL). V dokumentaci budete seznámeni s implementací interpretu, jeho všemi nezbytnými částmi, jako například s návrhem konečného automatu, lexikální analýzou, syntaktickou analýzou včetně precedenční tabulky a LL-gramatiky či algoritmy řazení využívané při práci s vestavěnými řetězci a vyhledávání podřetězce v řetězci. K dispozici je dále také rozdělení jednotlivých úkolů mezi členy týmu či způsob komunikace a práce v týmu.

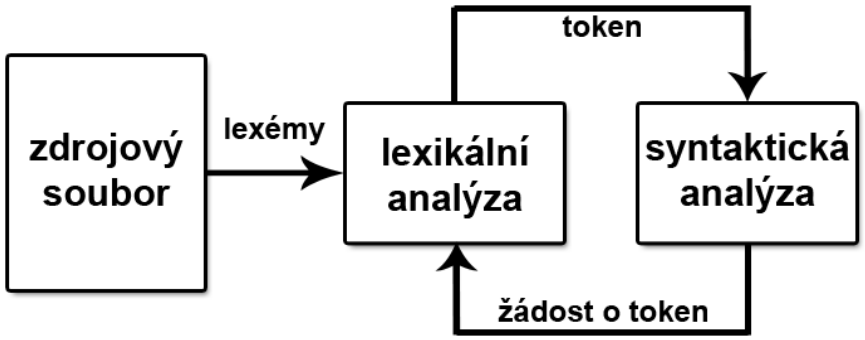
2 Řešení projektu

Zadáním projektu bylo vytvořit funkční interpret programovacího jazyka IJF16, jež patří do podmnožiny jazyka Java. V této kapitole se tak zaměříme na způsob řešení jednotlivých částí, které jsou nezbytné pro správnou funkčnost daného interpretu.

2.1 Lexikální analyzátor

První důležitou částí celého interpretu je lexikální analyzátor, jehož hlavním úkolem je číst vstupní soubor a rozdělit posloupnost znaků na takzvané **lexémy**. Tyto lexémy jsou reprezentovány **tokeny** (řetězec znaků ze zdrojového souboru), které jsou lexikálním analyzátorem posílány dále na syntaktickou analýzu. Po zpracování jednoho tokenu požádá syntaktický analyzátor funkci `getToken()` o zaslání dalšího tokenu. Velmi důležitou funkcí lexikálního analyzátoru je odstranění **bílých znaků** a **komentářů**.

Samotný lexikální analyzátor je řešen jako konečný automat (viz obrázek 6.1). Pomocí funkce `fgetc()` čteme postupně jednotlivé znaky ze vstupního souboru a podle daného znaku měníme stav konečného automatu. Počátečním stavem automatu je stav `LA_START`, z něhož začínáme načítat znaky. Konečné stavy automatu jsou označeny typickým dvojitým okrajem a určují jednotlivé typy tokenů (`tokenType`). Typem tokenu může být například klíčové slovo, číslo, závorka či středník.



2.2 Syntaktická a precedenční analýza

Syntaktický analyzátor je srdcem celého interpretu a jak již bylo zmíněno, velmi úzce spolupracuje s lexikálním analyzátorem, který žádá o tokeny, jež postupně vyhodnocuje. Syntaxí řízený překlad dělíme na dvě části, a sice `parser.c`, pro jehož implementaci byla využita metoda rekurzivního sestupu dle LL-gramatiky (viz příloha 6.2), kterou je sice snazší implementovat, avšak její nevýhoda spočívá v tom, že gramatiku již nelze bez větších zásahů do zdrojového kódu v průběhu měnit, a část `expressions.c`, kde dochází ke zpracování výrazů, což je založeno na precedenční analýze. Data, s nimiž následně pracuje také interpret, jsou ukládány do binárního stromu, v němž vytváříme uzel, který obsahuje informace o jméně či typu prvku.

Po spuštění syntaktického analyzátoru dochází k žádosti lexikálního analyzátoru prostřednictvím volání funkce `getToken()` ve funkci `pParse()` o zaslání nového tokenu. V modulu `parser.c` poté dochází k postupnému volání dalších funkcí, které kontrolují syntax vstupního programu v daném jazyku IFJ16. Pokud při analýze narazíme na výraz, voláme právě precedenční analýzu. V této části však dochází také ke kontrole, zda daný soubor obsahuje dle zadání třídu s názvem `main` a její hlavní bezparametrická statická funkce run, jež nemá žádnou návratovou hodnotu.

Pokud tedy parser při průchodu programem narazí na výraz, zavolá se precedenční analýza. Příklady, kdy je parserem volána precedenční analýza:

- a = b + c - precedence je zavolán po přečtení tokenu =
- f (a < b) precedence je zavolán po přečtení tokenu if
- foo(a+b,c) precedence je zavoláno po přečtení tokenu (

Samotná precedenční analýza však může volat také sama sebe, a sice v případě, kdy je funkce součástí výrazu pro přiřazení (např. a = a + b * c - foo(a+g,h) / 8).

2.3 Interpret

Další částí projektu je implementace interpretu, jenž vykonává instrukce takzvaného tří-adresného kódu, jenž je generován prostřednictvím syntaktického a sémantického analyzátoru. Samotný interpret pracuje se zásobníkem `instrstack`, jenž úzce spolupracuje s binárním stromem `mainTree`. Zásobník instrukcí pak obsahuje instrukce v tří-adresném kódu**,** neboli ukazatel na tři uzly binárního stromu a typu instrukce.

Hlavní a nejdůležitější částí celého interpretu je funkce `interpretMainCore()`, která se spouští právě nad daným zásobníkem instrukcí. Jádro funkce tvoří cyklus, který je podmíněn tím, zdali se na zásobníku nachází instrukce či nikoli. Interpret postupně prochází zásobníkem a na základně aktuální instrukce pak danou instrukci dále zpracuje. V případě, že se na zásobníku nenacházejí již žádné další instrukce, cyklus je ukončen.

Důležitou roli hraje v interpretu taktéž funkce `semCheck()`, jež bez ohledu na podmíněné či nepodmíněné skoky ověří sémantiku každé instrukce, čímž zajistíme to, že i část programu, která se nikdy nemůže vykonat, bude zkontrolována, zdali je napsána sémanticky správně. Jestliže funkce `semCheck()` nalezne v programu chybu, program se ukončí s danou návratovou hodnotou chyby.

2.4 Garbage collection

Pro jednodušší vývoj a ušetření času jsme přistoupili k takzvané automatické správě paměti, která se označuje také jako Garbage collection. Výhodou Garbage collectoru, který je implementován v souboru `garbage_collector.c`, je, že se již více nemusíme zabývat problémy spojenými s ověřením správné alokace či uvolněním paměti. V ostatních soborech tak již pouze voláme jednotlivé funkce Garbage collectoru, které ověří, zda je možné paměť alokovat či realokovat a následně pak ukládají ukazatel na právě alokovanou paměť. Takto alokovanou paměť si již spravuje sám Garbage collector.

3 Řešení daných algoritmů z pohledu předmětu IAL

V této části dokumentace se zaměříme na způsob implementace jednotlivých algoritmů, které slouží k práci s vestavěnými řetězci a k vyhledávání podřetězce v řetězci, a tabulky symbolů.

3.1 Boyer-Mooreův algoritmus

Pro vyhledávání podřetězce v řetězci byl využit dle zadání takzvaný Boyer-Mooreův algoritmus s modifikací „Bad mismatch table“. Pro tuto tabulku jsme využili jednosměrně vázaný seznam skládající se ze struktur `mismatchTable` a `mismatchTableItem`. Struktura mismatchTable nám ukazuje na začátek seznamu a také na aktivní prvek pomocí First a Active členů. Člen first ukazuje na další člen, atd. až na konec seznamu. Pro implementaci takového seznamu jsme využili funkcí `initMismatchTable()`, `disposeMismatchTable()`, `insertNext()`,`findChar()`,`getShiftValue()` a `updateShift()`.

Samotná funkce find pracuje následovně. Funkce obdrží dva parametry – hledaný řetězec a řetězec, ve kterém se má vyhledávat. Z hledaného řetězce si získáme jeho délku, což je hlavní hodnota pro shift, která je pro všechny znaky rovna délce tohoto slova. Dle zadání víme, že pokud obdržíme prázdný string, takový string je na začátku slova, můžeme jej tedy okamžitě vrátit s pozicí 0 - také je logické, že pokud je hledaný string delší, než string ve kterém vyhledáváme, string nenalezneme a tak vrátíme okamžitě -1. Počáteční podmínky máme tedy nastaveny a nyní můžeme provést inicializaci mismatch tabulky pomocí funkce `initMismatchTable()`.

Následně je třeba tabulku naplnit korektními znaky, což provedeme cyklem, v němž projdeme všechny znaky od začátku slova do konce a kontrolujeme, zdali již nějaký znak v tabulce není. Pokud není, přidáme jej do tabulky na začátek seznamu. Pokud tam již daný prvek je, zaktualizujeme jeho hodnotu posunu neboli shiftValue. Tato hodnota se vypočítá následovně: **shift = DELKA_SLOVA - index_PISMENE – 1**. Jakmile naplníme celou tabulku, nezbývá nám než ukončit cyklus a skočit do dalšího nového cyklu, jenž prochází string dokud string nenalezne, nebo dokud se posouváním nedostaneme za konec řetězce, ve kterém vyhledáváme. V každé iteraci tohoto cyklu jsme nuceni si nastavit zarážku, jejíž hodnota je ZARAZKA = AKT. HODNOTA ZARAZKY + shift. Po nastavení zarážky jsme nuceni spustit druhý cyklus vnořený do prvního, který nám kontroluje shodu stringu s aktuální pozici zarážky od konce. Pokud je nalezen rozdíl, získáme si hodnotu shiftValue ze struktury mismatchTable. Pokud není nalezen rozdíl, kontrolujeme písmena až na začátek stringu, pokud se všechny shodovaly, vrátíme hodnotu pozice, na které jsme řetězec nalezli. Před vrácením hodnoty jsme ještě nuceni uvolnit mismatchTable z paměti funkci `disposeMismatchTable()`.

3.2 Heap Sort

Metoda Heapsort, neboli řazení hromadou, patří mezi nejefektivnější řadící algoritmy, které jsou založeny na porovnávání prvků s asymptotickou složitostí. Základem heapsortu je speciální datový typ, jenž se nazývá hromada, nejčastěji pak binární hromada, která je založena na binárním stromu.

K implementaci samotné metody jsme přistoupili následovně. String je dle pořadí znaků ve stringu seřazen do pole charů, jež je následně zpracováváno jako kompletní binární strom s prvkem root, který má mít nejvyšší ordinální hodnotu. Strom je vyplňován zleva, což vyplývá z definice kompletního binárního stromu. Prvky s nejvyšší ordinální hodnotou, takzvané root prvky, převedeny na konec stringu a haldě (binárním stromu) je následně potřeba opravit seřazení prvku. Root prvek tedy musí mít nejvyšší ordinální hodnotu a všechny ostatní prvky musí mít hodnotu nižší. Toto pravidlo platí pro každý vrchol stromu. Zde se sice celou dobu bavíme o stromu, ale toto zobrazení do stromu, může být převedeno do zobrazení v poli charů jednoduchým pravidlem. "Děti" každého hlavního prvku nalezneme na pozicích: 2*n+1 a 2*n+2 (číslováno od 0).

Co se samotných funkcí týče, funkce sort pracuje s polem charů, tedy s řetězcem. String předaný parametrem funkce je přiřazen do pomocného stringu, následně je zjištěna délka tohoto stringu - poté se provede oprava haldy rekurzivní funkcí `repairHeap`, která má jako parametry pomocný string a délku stringu - tato délka stringu se nám bude do budoucna hodit pro určování, které znaky se nám již podařilo korektně seřadit - rekurzivní funkce `repairHeap` si nejprve určí největší znak v celém poli znaků, následně si cyklem provede kontrolu, zdali není největší znak na první pozici ve stringu, pokud ano, skončíme funkci - pokud na první pozici není, provádíme swap znaků od spodu stromu k vrcholu. Po swapu kontrolujeme, jestli jsou všechny znaky korektně umístěny. Pokud ne, provedeme rekurzivní volání funkce `repairHeap`. Pro swap znaků jsme použili funkci `swap`, která prohodí dva znaky za pomocí pointerů. V případě, že jsou všechny znaky korektně umístěny, rekurze není uskutečněna a funkce `repairHeap` končí. Dochází k návratu do funkce sort, která následně prohodí největší znak s posledním znakem a opakuje cyklus, dokud není seřazeno celé pole znaků (string).

3.3 Tabulka symbolů

Tabulku symbolů jsme implementovali dle vybraného zadání, a sice pomocí binárního vyhledávacího stromu, tedy speciálního druhu binárního stromu. Uzlem stromu (položkou tabulky) je struktura obsahující informace o jméně či typu prvku. Každý uzel může mít maximálně dva další syny, kteří se označují jako takzvaný levý a pravý podstrom. Hodnota ve vrcholu je větší než hodnota v levém podstromu, ale menší než hodnota v pravém podstromu.

Binární vyhledávací strom obsahuje uzel třídy, kde každý uzel má podstromy, a to funkce třídy, proměnné třídy, podstrom lptr a rptr, na něž jsou v případě, že se v programu nacházejí, navázány další. Kromě uzlu třídy však existuje také uzel funkce, jenž má jako podstromy proměnné funkce. Vyhledávání v binárním stromu pak probíhá následovně. Ze všeho nejdříve si zvolíme start vyhledávání a následně porovnáváme název uzlu (klíč) s názvy třídy (klíč). Rekurzivně procházíme celý binární stromu, dokud danou položku nenajdeme. Pokud se nám nepodaří prvek nalézt, vracíme hodnotu null.

4 Práce v týmu

Jelikož se jedná o týmový projekt, je tento úkol založen zejména na správné komunikaci mezi jednotlivými členy a schopnosti pracovat ve skupině, což je pro studenty užitečné zejména v budoucí praxi.

4.1 Rozdělení práce

- **Jan Hrbotický** – organizace práce, rozdělení jednotlivých úkolů dalším členům týmu a implementace binárního stromu, jádra interpretu, práce na zásobníku a garbage collectoru
- **Dominik Skála** – tvorba makefilu, implementace vestavěných funkcí pro práci s řetězci, algoritmu heap sort, lexikálního analyzátoru
- **Milan Hruban** – implementace syntaktického analyzátoru, testování funkčnosti interpretu
- **David Hél** – tvorba dokumentace, prezentace, implementace garbage collectoru, zásobníků a práce na precedenční analýze, tvorba precedenční tabulky
- **Martin Hons** – práce na precedenční analýze, zásobníku a lexikálním analyzátoru, testování funkčnosti, tvorba precedenční tabulky

4.2 Plánování a vedení

Implementace interpretu byla časově velmi náročná, tudíž bylo nezbytnou součástí projektu také správné plánování. Jelikož neměla většina členů našeho týmu s úkolem podobného rozsahu příliš mnoho zkušeností, tuto část jsme zřejmě trochu podcenili a museli jsme o to více pracovat v závěrečných týdnech. Náš tým si zprvu neurčoval žádné pevné termíny a spoléhali jsme tak, že každý z členů dodrží své slovo a svou část stihne v určitém časovém intervalu splnit. Často jsme se však díky tomu dostávali do časové tísně. Týmový projekt byl pro většinu členů týmů zcela něčím nový a i podobné komplikace pro nás byly skvělou zkušeností.

Důležitou součástí každého týmu je bezesporu vedoucí, jehož hlavním úkolem je rozdělit práci mezi ostatní členy a dohlédnout, zda bylo všechno potřebné uděláno. V našem týmu se této práce zhosti Jan Hrbotický, který pravidelně informoval všechny členy o aktuálním plánu a poctivě vedl každou schůzku. V týmu však nebyl žádný větší problém a všichni členové si svou práci plnili poměrně spolehlivě.

4.3 Komunikace v týmu

Správná komunikace byla pro úspěšné vyřešení projektu nezbytnou. Každý týden jsme tak pořádali porady, během nichž byla rozdělena práce, členové byli seznámeni s danými problémy a následně jsme si vytyčili cíle pro nadcházející týden. V případně potřeby jsme také komunikovali prostřednictvím programu TeamSpeak. K rychlé komunikaci byl využit také chat na Facebook, který však není zrovna ideálním řešení, když často svádí ke konverzaci mimo téma a důležité informace o projektu tak někteří členové nemusejí zaregistrovat. Na Facebooku byla vedoucím týmu vytvořena také skupina, kde byl každý týden vkládán aktualizovaný plán včetně důležitých dokumentů s rozdělením práce či štábní kulturou.

4.4 Správa verzí

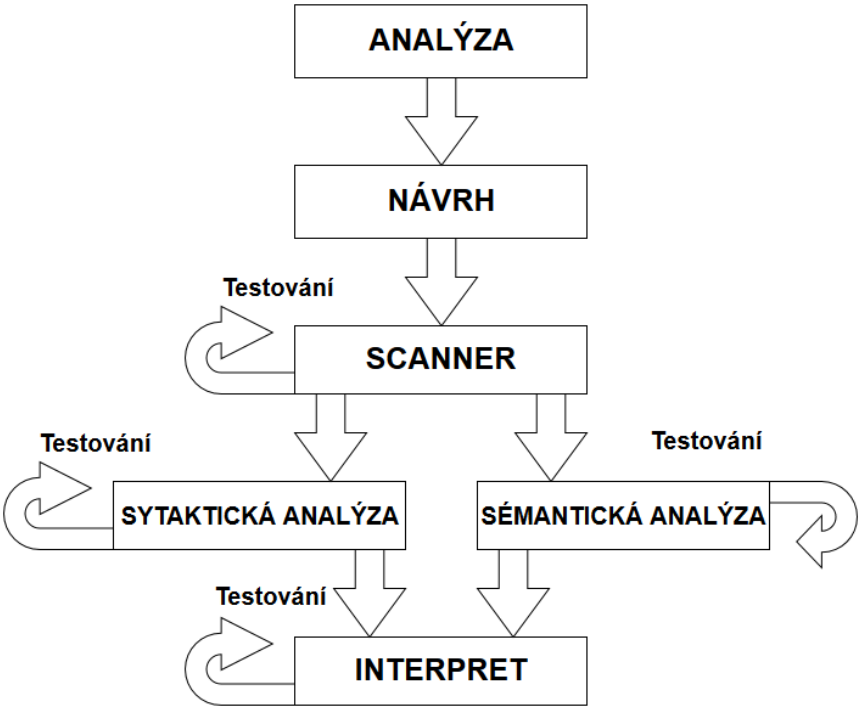
Pro správu verzí jsme využili systém správy verzí Git na privátním serveru jednoho z členů týmu a webovou službu GitHub, která umožňuje vývoj software za pomoci právě verzovacího nástroje Git. Jedná se o bezplatnou službu, která je pro týmový projekt velmi vhodná. Každý z členů si může vytvořit svou vlastní větev, v níž pracuje nezávisle na ostatních členech týmu. Jednotlivé větve se pak spojí do jedné hlavní větve (tzv. `master`). Díky tomu jsme tak neměli problém se synchronizací veškerých informací, především pak zdrojových kódů. Podle dostupných grafů lze navíc například zjistit, jakým způsobem se každý z členů zapojil do projektu.

4.5 Rozdělení bodů

Po společné dohodě všech členů týmu jsme se rozhodli pro drobnou úpravu procentuálního rozdělení bodů. Jelikož měl vedoucí týmu s projektem největší starosti a všechny své povinnosti si poctivě plnil, nastudoval si potřebný materiál s předstihem, vedl každou schůzi a vždy rozděloval a kontroloval práci, rozhodli jsme se na úkor každého z členů přidělit vedoucímu čtyři procenta. Vedoucí týmu tedy obdržel 24%, zbytek týmu si rozdělil rovnoměrně zbylých 76%.

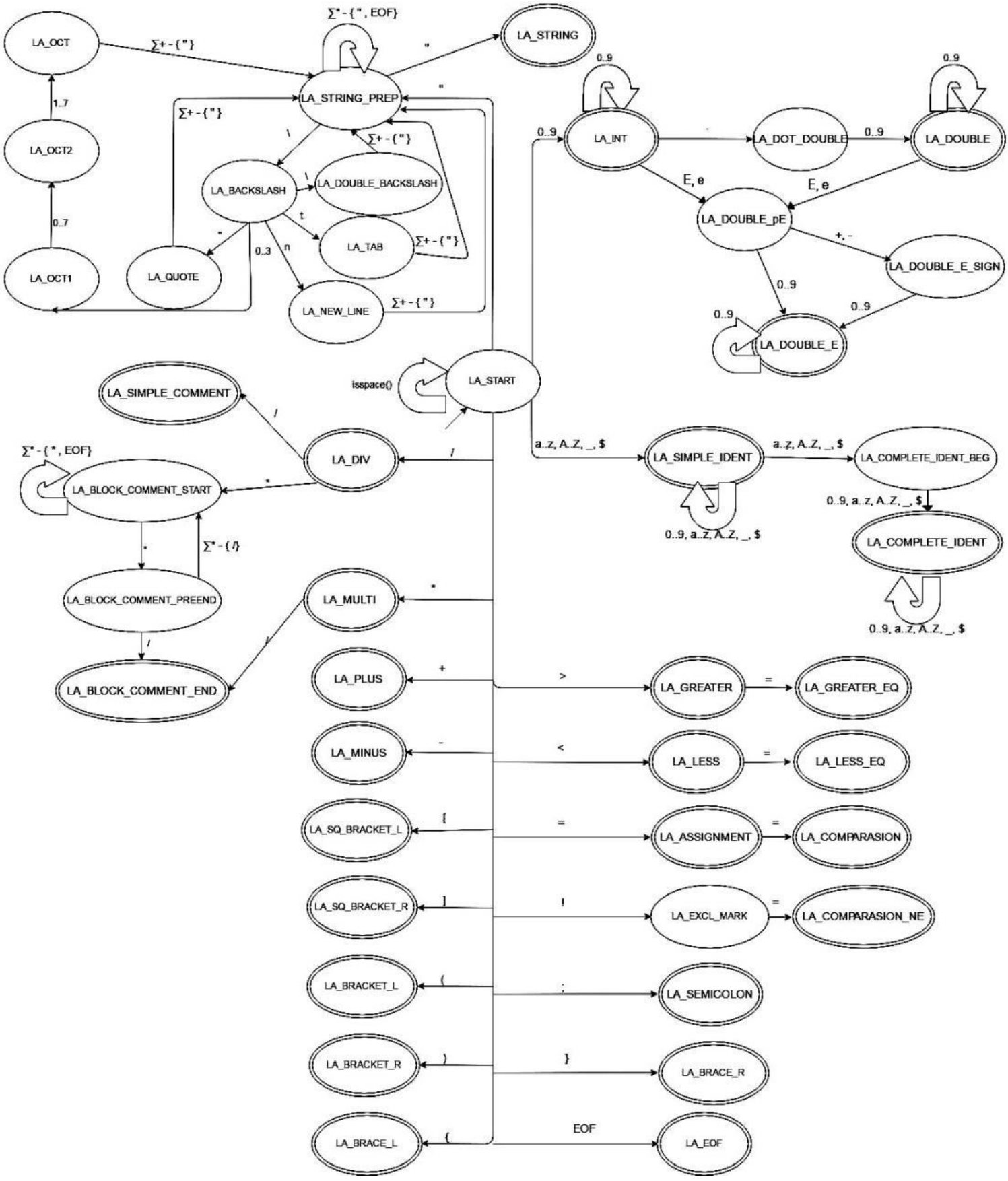
4.6 Vývoj

Přestože jsme si na začátku projektu neurčili, jaký vývojový cyklus či metodu pro implementaci našeho interpretu zvolíme, nevíce jsme se zřejmě přiblížili metodě RUP čili **Rational Unified Process**, která se vyznačuje takzvaným iterativním vývojem. Zaměřili jsme se především na postupný vývoj jednotlivých komponent interpretu, jež bylo možné postupně upravovat dle potřeb. Po dokončení každé části následovalo důkladné testování funkčnosti a spuštění kódu. Sice jsme byli nuceni často opravovat části kódu, jelikož na sebe musely jednotlivé části přesně navazovat, avšak mnoho chyb se nám podařilo odladit hned v začátcích.



5 Přílohy

5.1 Návrh konečného automatu



5.2 LL-gramatika

```
<FILE> -> <classes> EOF
<classes> -> <class> <classes>
<classes> -> ε
<class> -> class_kw class_ID {<class_body>}
<class_body> -> static_kw <data_type> var_ID <var_assign>
<class_body>
<class_body> -> static_kw <return_type> func_ID(<params>)
{<commands>} <class_body>
<class_body> -> ε

<var_assign> -> = <expression> ;
<var_assign> -> ;

<data_type> -> String_kw
<data_type> -> double_kw
<data_type> -> int_kw

<return_type> -> String_kw
<return_type> -> double_kw
<return_type> -> int_kw
<return_type> -> void_kw


<params> -> ε
<params> -> <data_type> param_ID <params_next>

<params_next> -> ε
<params_next> -> , <data_type> param_ID <params_next>

<commands> -> ε
<commands> -> if_kw (<expression>) {<commands>} else_kw
{<commands>} <commands>
<commands> -> while_kw (<expression>) {<commands>}
<commands>
<commands> -> return_kw <expression> <commands>
<commands> -> var_ID = <expression> ; <commands>
<commands> -> func_ID(<call_params>) ; <commands>
<commands> -> <data_type> var_ID <var_assign> <commands>

<call_params> -> ε
<call_params> -> <expression> <call_params_next>

<call_params_next> -> ε
<call_params_next> -> , <expresison> <call_params_next>
```

5.3 Precedenční tabulka

	(	)	/	*	+	-	==	!=	<	>	<=	>=	!	,	;	ID
(	<	=	<	<	<	<	<	<	<	<	<	<	<	=	F	<
)	F	>	>	>	>	>	>	>	>	>	>	>	>	>	>	F
/	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<
*****	<	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<
+	<	>	<	<	>	>	>	>	>	>	>	>	>	>	>	<
-	<	>	<	<	>	>	>	>	>	>	>	>	>	>	>	<
==	<	>	<	<	<	<	>	>	>	>	>	>	>	>	>	<
!=	<	>	<	<	<	<	>	>	>	>	>	>	>	>	>	<
<	<	>	<	<	<	<	>	>	>	>	>	>	>	>	>	<
>	<	>	<	<	<	<	>	>	>	>	>	>	>	>	>	<
<=	<	>	<	<	<	<	>	>	>	>	>	>	>	>	>	<
>=	<	>	<	<	<	<	>	>	>	>	>	>	>	>	>	<
!	=	>	>	>	>	>	>	>	>	>	>	>	>	>	F	<
,	<	=	<	<	<	<	F	F	F	F	F	F	F	=	F	<
;	<	F	<	<	<	<	<	<	<	<	<	<	F	=	F	<
ID	F	>	>	>	>	>	>	>	>	>	>	>	F	>	>	F

6 Metriky kódu

- **Počet souborů:** 21
- **Počet řádků kódu:** 5461

7 Zdroje

[1] HORDĚJČUK, Vojtěch. *Binární strom: Binární vyhledávací strom* [online]. [cit. 2016-12-09]. Dostupné z: http://voho.cz/wiki/datova-struktura-binarni-strom/

[2] HONZÍK, Jan. *Algoritmy – studijní opora*. Brno: Vysoké učení technické, 2016.

[3] MEDUNA, Alexander a Roman LUKÁŠ. *Formální jazyky a překladače – studijní opora*. Brno: Vysoké učení technické, 2016