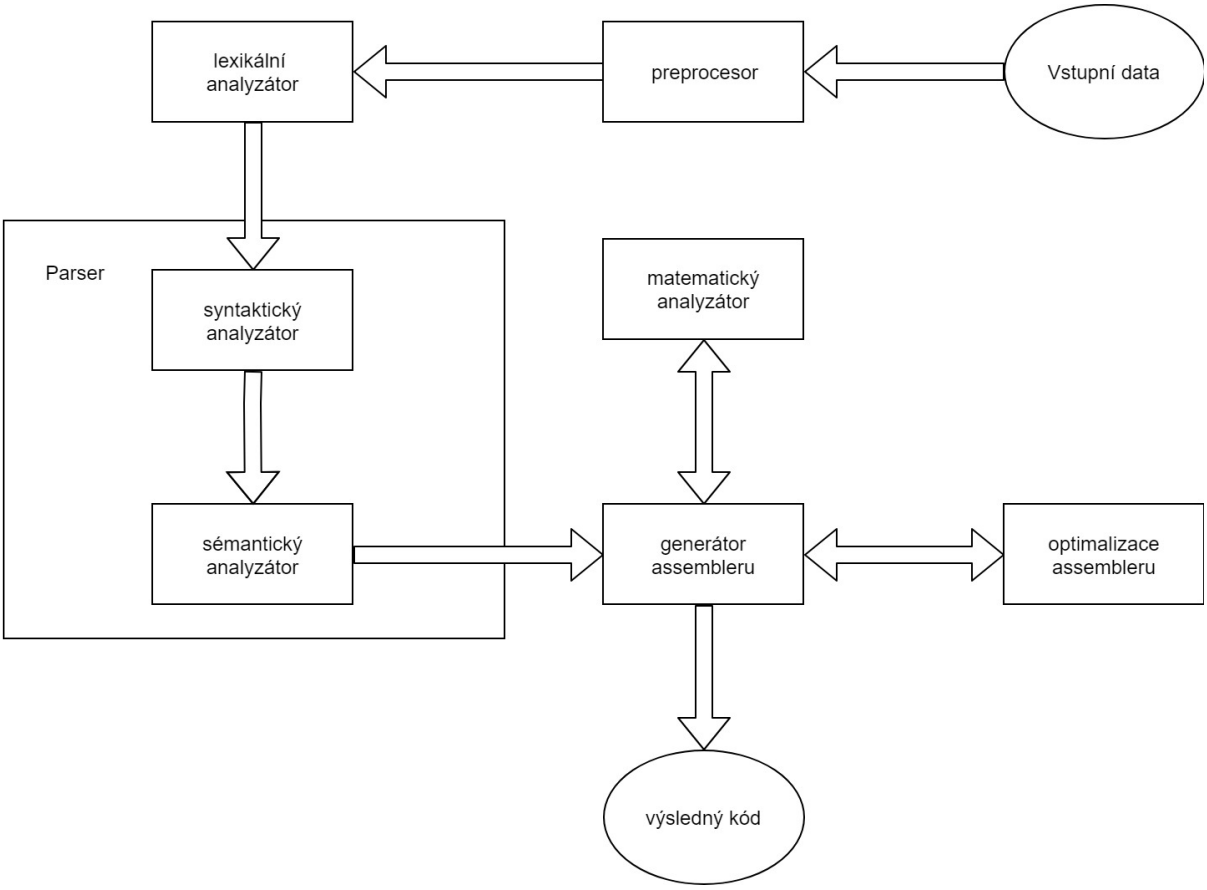
IMPLEMENTACE PŘEKLADAČE IMPERATIVNÍHO JAZYKA IFJ18

TÝM 120, VARIANTA II



Autoři:

Radek Duchoň, xduchoo7, 27%, vedoucí týmu

Josef Oškera, xoskero3, 27%

Jan Doležel, xdolez81, 27%

Jan Juda, xjudajoo, 19%

Rozšíření:

BASE

BOOLOP

FUNEXP

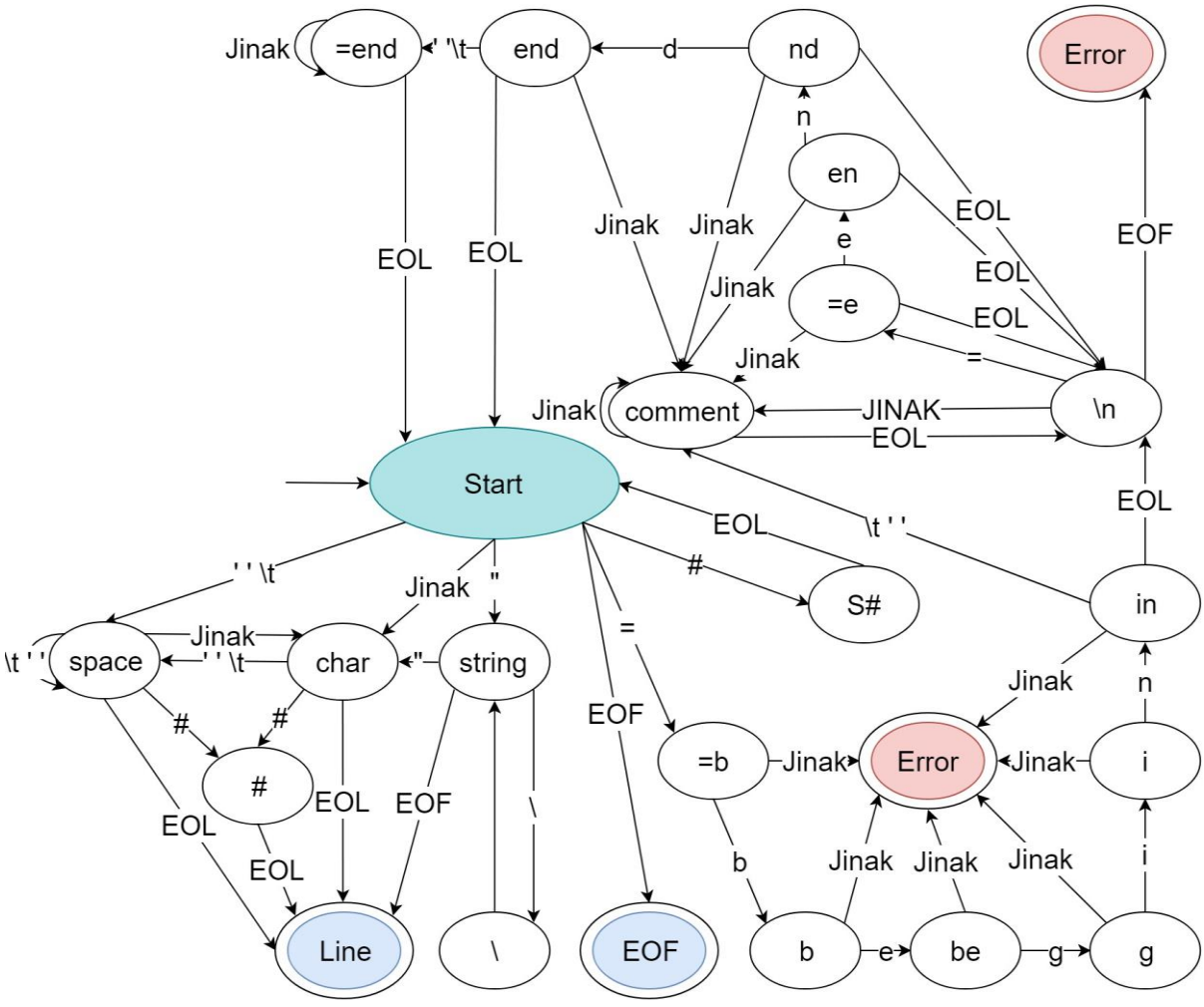
IFTHEN

OBSAH

NÁVRH KOMPILÁTORU

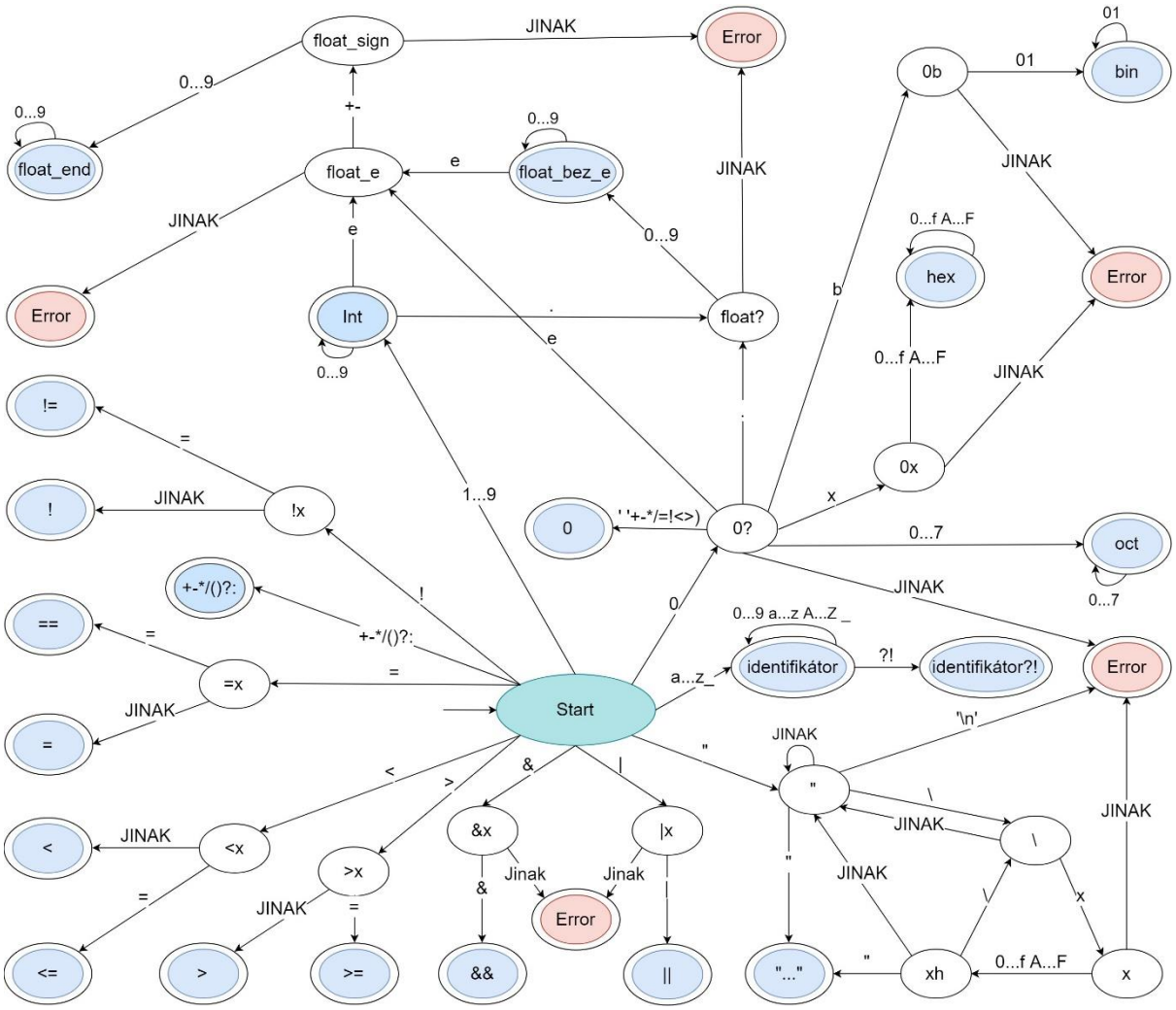
PREPROCESOR

Tento modul funguje zcela nezávisle na ostatních, dodává scanneru vždy jeden "zpracovaný" řádek znaků, ze kterého jsou odstraněny komentáře a nadbytečné bílé znaky, které nejsou potřeba pro další zpracování. V souvislosti s tím může zahlásit chybu spojenou s komentáři, kterou by jinak nešlo dohledat později. Vstup je určen proměnnou typu FILE*, čili se dá nastavit i například na čtení ze souboru namísto ze standardního vstupu.



SCANNER

Scanner je první část kompilátoru, která se spouští přímo z funkce main. Po počáteční inicializaci scanner opakovaně volá preprocesor, který mu předává řádky vstupu. Ukončení tohoto opakovaného volání nastane, když preprocesor zahlásí, že načetl konec vstupu. Jednotlivé řádky jsou poté znak po znaku zpracovávány a dle konečného automatu



jsou generovány jednotlivé tokeny. Zpracovávání jednoho řádku končí, když scanner narazí na znak konce řádku a vygeneruje jemu odpovídající token. Každý token odpovídá právě jednomu lexému. Pro reprezentaci tokenů ve scanneru jsme zvolili strukturu „struct token", která obsahuje ID tokenu a 3 datové položky: ukazatel na string, integer a double. Tyto datové položky slouží pro uchování literálů či jmen identifikátorů. Pro ID tokenu jsme vytvořili datový typ „enum TOKTYPE", který specifikuje výčet všech stavů KA scanneru.

Výstupem scanneru je dvourozměrné pole tokenů, se kterým pracují ostatní části projektu.

SYNTAKTICKÁ ANALÝZA

Tento modul je zavolán po zdárném ukončení scanneru, přijímá pole polí tokenů a stav, ve kterém se má spustit (prvotní stav je "PROG"). Následně využívá LL gramatiky při metodě rekurzivního sestupu shora dolů. Využívá pomocnou funkci "zavyraz", která přeskakuje výrazy (tedy úspěšně ignoruje jakýkoliv token, který může náležet výrazu). Syntaktický analyzátor nijak neupravuje tokeny, proto pouze vrací integer značící, zda skončil správně či s chybou.

1)	S	->	def ID (A) EOL SV end EOL S
2)	S	->	if V then EOL SIF Q S
3)	S	->	while V do EOL SV end EOL S
4)	S	->	EOF
5)	S	->	P S
6)	SV	->	if V then EOL SIF Q SV
7)	SV	->	while V do EOL SV end EOL SV
8)	SV	->	€
9)	SV	->	P SV
10)	SIF	->	if V then EOL SIF Q SIF
11)	SIF	->	while V do EOL SV end EOL SIF
12)	SIF	->	€
13)	SIF	->	P SIF
14)	P	->	IDN1
15)	P	->	EOL
16)	P	->	V EOL
17)	IDN1	->	= V EOL
18)	IDN1	->	EOL
19)	IDN1	->	V EOL
20)	A	->	IDN An
21)	A	->	€
22)	An	->	, IDN An
23)	An	->	€
24)	Q	->	elsif V then EOL SIF Q
25)	Q	->	else EOL SV end EOL
26)	Q	->	end EOL

S = Hlavní tělo programu
P = Příkaz
V = Výraz
SV = Tělo funkce v cyklu/funkci
SIF = Tělo funkce v podmínce
Q = konstrukce podmínky
IDN1 = Identifikátor na první pozici řádku, který může být proměnná
A = První argument funkce
An = další argument funkce

N\T	DEF	ID	IDN	(	)	EOL	IF	ELSIF	ELSE	THEN	WHILE	DO	END	PŘIŘAĎ	VÝRAZ	ČÁRKA	EOF
S	1	5	5	5	5	5	2	5	5	5	3	5	5	5	5	5	4
SV	9	9	9	9	9	9	6	9	9	9	7	9	8	9	9	9	9
SIF	13	13	13	13	13	13	10	12	12	13	11	13	12	13	13	13	13
P	16	16	14	16	16	15	16	16	16	16	16	16	16	16	16	16	16
IDN1	19	19	19	19	19	18	19	19	19	19	19	19	19	17	19	19	19
A			20		21												
An					23												
Q								24	25				26				

SYMTABLE

Tento modul slouží k uchování tabulky symbolů a je implementován pomocí tabulky s rozptýlenými položkami. Vybrali jsme si variantu s hashovací tabulkou, jelikož jsme ji již někteří implementovali v rámci jiného předmětu, a tudíž jsme mohli využít své zkušenosti a znalosti s její implementací. Pro rozptýlení využíváme sdbm hash funkci, která je převzata z http://www.cse.yorku.ca/~oz/hash.html.

Položka tabulky vždy obsahuje dvojici klíč (ukazatel na "string") a data (ukazatel na jakékoliv data).

Samotná tabulka je tvořena strukturou obsahující pole lineárně vázaných seznamů obsahujících položky.

SÉMANTICKÝ ANALYZÁTOR

Sémantický analyzátor (tzv. První průchod, zkráceně PP) lineárním průchodem prochází pole tokenů, při kterém přidává do příslušné tabulky symbolů funkce a proměnné u kterých si zaznamená kde jsou definovány. Dále kontroluje, zda proměnná není volaná jako funkce. Také zde probíhá kontrola, zda je v místě volání funkce definována. Při průchodu funkcemi si přidává informace o voláních do dat funkce v symtable, která tato funkce provádí. Poté co PP narazí na volání funkce z těla programu probíhá rekurzivní ověření, zda je volaná funkce v tomto místě definovaná, a to i pro všechna její volání. Při této činnosti vytváří lineární seznam volání pro zastavení kontroly případné rekurze jakéhokoliv druhu.

GENERÁTOR ASSEMBLERU

Generátor assembleru (zkratka GA) je volán přímo z funkce main jako třetí část kompilátoru. GA částečným rekurzivním sestupem prochází pole tokenů, přičemž se řídí jednoduchými obecnými pravidly a s pomocí dopředného prohledávání určuje konkrétní řídící struktury. Podle typu řídící struktury generuje řídící generuje řídící struktury pro IFJcode18 do ramfile. V průběhu generování těchto struktur se spouští modul matematika, které je předávána informace kolik tokenů je pro ni.

MATEMATIKA

Matematický analyzátor (dále jen MA) je volán přímo z generátoru assembleru nad konkrétním úsekem tokenů, který byl identifikován jako výraz, a má na starost kompletní zpracování výrazů. Nejdříve MA provádí kontrolu platnosti rozsahu, nad kterým byl zavolán a poté kontroluje, zda všechny tokeny, které má zpracovávat patří do jeho množiny terminálů. Dále MA spouští convert, který provádí syntaktickou kontrolu výrazů a převod do postfixu. Po všech kontrolách se spustí hlavní cyklus, který jede token po tokenu a sestavuje z nich výsledný kód. MA používá datovou strukturu zásobník, který je implementovaný podle materiálů z předmětu IAL. Konstanty jsou ukládány právě na tento zásobník za účelem optimalizování výsledného kódu – všechny operace, které se provádí čistě nad konstantami též jako výsledek vrací konstantu, a tudíž negenerují asemblerový kód. Po ukončení hlavního cyklu se zkontroluje, zda na zásobníku něco nezůstalo (výsledek konstantových operací) a pokud ano, tak se toto pošle na asemblerový zásobník a prohlásí se za výsledek tohoto výrazu. Výsledek výrazu se tedy vždy nachází na assemblerovském zásobníku.

MA vrací řídící strukturu sa_pos, kde je mimo jiné i položka s chybovým kódem.

CONVERT

Pro potřebu našeho projektu se všechny výrazy konvertují do námi preferované podoby. Nejdřív je přijat řádek tokenů s výrazem, který se zde upravuje. Konkrétně se doplňují závorky k funkcím a v rámci funkcí se zde následně převrací pořadí argumentů pro další práci, zejména funkce print, které bychom jinak dávali argumenty v opačném pořadí.

Následně se zde převádí výrazy na postfix, o kterém jsme se dozvěděli v předmětu IAL a zaujal nás, jelikož nás ihned napadlo, že by se s jeho pomocí dalo implementovat rozšíření FUNEXP. Dle normy byly jednotlivým operátorům přisouzeny priority a algoritmicky se následně převádí zápis do postfixové notace (za pomoci zásobníků, kde jsme se též inspirovali prezentacemi z předmětu IAL), ovšem oproti normě se chováme k operátorům negace jako k funkcím (kterým jsme též doplnili závorky), jelikož jsou unární a pracuje se s nimi v námi navrženém řešení lépe tak, že zůstanou na jedné pozici. Další rozdíl oproti normě je, že v našem postfixu jsou stále závorky, a to právě pro vyhodnocování funkcí.

Při zmíněném převodu do postfixu probíhá také syntaktická kontrola výrazu.

POSTUP PRÁCE

Naše skupina se poprvé sešla dne 9.11.2018, kdy jsme si společně prošli zadání, navrhli základní strukturu tokenů scanneru a výčet typů těchto tokenů. V dalších dnech jsme se opakovaně scházeli a navrhovali jsme konečný automat scanneru, zjišťovali, jaké všechny stavy budeme potřebovat, a jak se mezi nimi bude přecházet. Už v této části řešení jsme se rozhodli implementovat rozšíření BASE. Konečný automat byl později ještě několikrát předěláván, ale vesměs se jednalo o drobné přidávání či odebírání nebo slučování stavů. V rané části návrhu jeden z nás přišel s nápadem vytvořit preprocesor pro odstranění komentářů a nadbytečných bílých znaků.

Po dokončení prvotní implementace scanneru jsme se všichni společně vrhli na parser, čímž začalo velmi intenzivní třítýdenní období naší práce, během kterého jsme se neúnavně scházeli ve škole v CVT téměř každý pátek, sobotu, neděli a kterýkoliv jiný školní den, kdy na to byl po přednáškách čas. Vývoj parseru v rámci našeho projektu probíhal značně zajímavě, jelikož jsme vyšli z myšlenky vytvořit syntaxí řízený překlad s využitím rekurzivního sestupu. Během tvorby tohoto parseru jsme postupně začlenili rozšíření BOOLOP, IFTHEN a FUNEXP. Spolu s tímto parserem jsme též začali implementovat matematický analyzátor, který analyzoval matematické výrazy a volání funkcí a generoval pro ně kód. Výsledkem našeho snažení byl sice funkční parser (syntaktický analyzátor s kontrolou sémantiky) a dokonce uměl přímo generovat kód řídících struktur, bohužel jsme až asi o týden později zjistili, že náš syntaktický analyzátor (součást parseru) vůbec nesplňuje formální požadavky, a to přesně v tom, že neimplementuje LL1 gramatiku a že se vlastně ani moc nejedná o rekurzivní sestup ale spíše o námi nazvanou metodu „prediktivního pádu" (či též také „padající rekurze"), tudíž jsme se rozhodli navrhnout zcela novou gramatiku a k ní i nový syntaktický analyzátor. Ze starého parseru jsme odstranili syntaktické kontroly a vyextrahovali jsme z něj sémantický analyzátor. Zbytek starého parseru posloužil jako generátor kódu.

Jelikož v zadání není povoleno vytvářet dočasné soubory a výstupní kód má být posílán na standardní výstup, tak jsme si pro uschování vygenerovaného kódu před jeho finálním vytisknutím vytvořili knihovnu ramfile, pomocí které zapisujeme do pole řetězců, všechny data jsou tedy uchována v operační paměti (RAM – od toho název ramfile).

Matematický analyzátor si během naší práce na projektu též prošel značnými změnami. V prvotním návrhu se všechny výsledky i mezivýsledky předávaly přes proměnné, to nás ale dovedlo k problému, že pro každý mezivýsledek by bylo nutné vytvořit další proměnnou (v rámci analýzy jednoho výrazu), což by bylo velmi neefektivní. Tak jsme se rozhodli předělat analyzátor výrazů do zásobníkové formy, kde se vše – jak mezivýsledky, tak výsledky samotných výrazů – bude předávat přes zásobník. Tento návrh se nám po nějakou dobu zdál jako osvědčený, ale když po dokončení parseru přišla vlna testování, tak jsme zjistili, že v určitých případech, hlavně v cyklech, se nám na zásobníku hromadí výsledky nevyužitých výrazů, což nám v některých situacích způsobuje značné problémy. Tudíž přišel třetí a v této chvíli snad poslední návrh, ve kterém zůstává předávání mezivýsledků a

parametrů funkcí přes zásobník, ale výsledky výrazů se předávají přes jednu globální proměnnou, což zajistí, že se nám na zásobníku nebude hromadit nic, co by nemělo.

Pro vyhodnocování matematických výrazů jsme zvolili použití postfixové notace. Jeden z nás přišel se skvělým nápadem, že převod do postfixu by mohl probíhat už v preprocesoru v čistém vstupním textu. Jelikož se implementace tohoto řešení sám ujal, tak jsme se s ním moc nehádali. O pár dní později po vyhnutí se několika problémům (například otázce, zda je identifikátor proměnná nebo funkce) jsme ovšem narazili na ultimátní překážku v tomto řešení, kterou byl unární operátor not, a zjistili jsme tak, že převod do postfixu spolu se všemi čtyřmi námi implementovanými rozšířeními prostě není proveditelný v preprocesoru (tudíž před lexikální, syntaktickou a hlavně sémantickou analýzou) a že celý tento převod tedy bude nutné přepsat do varianty s tokeny, která se nyní volá přímo z matematického analyzátoru. Přesto však naše snažení v tomto způsobu implementace nebylo úplně zbytečné, protože jsme během něj došli ke skvělému nápadu, a to k tomu, že doplněním neuvedených závorek při volání funkcí na správná místa (podle počtu parametrů...) se hodně zjednoduší vyhodnocování výrazů.

Jedním z posledních problémů, na které jsme v postupu práce narazili, se objevil při tvorbě vestavěných funkcí, konkrétně funkce print. Vzhledem k tomu, že matematický analyzátor zpracovává výrazy z leva a jejich mezivýsledky ukládá na zásobník, ze kterého si poté funkce při volání načítají svoje parametry, se stalo, že parametry funkce print se tiskly v opačném pořadí – od konce. To jsme ovšem vyřešili obrácením parametrů všech funkcí ještě před zahájením samotného vyhodnocování výrazu.

Po vyřešení všech hlavních problémů a po dosažení přijatelné funkčnosti jsme začali s další velkou vlnou testování, při níž jsme využili automaticky se vyhodnocujících testů, které se spustí s každou novou aktualizací našeho git repositáře na serveru jednoho našeho kolegy. Výsledkem testů je vygenerovaná html stránka obsahující informace o úspěšnosti všech testů. Díky této testovací vlně (a díky druhému pokusnému odevzdání) jsme dokázali zachytit a opravit spousty drobných chybiček.

Spolu s testováním jsme se uchýlili k optimalizování námi generovaného kódu. V poslední etapě naší práce jsme začali psát tuto dokumentaci a ucelovat komentáře kódu do konečné podoby.

„Poslední překážkou", se kterou jsme se setkali, bylo rozdělení bodů. Jeden z nás sám uznal, že si nemyslí, že by odvedl tolik práce jako ostatní, a tak sám požádal o částečné odebrání jeho části bodů, tak se tedy dospělo k výslednému hodnocení.