Dokumentace k projektu IFJ/IAL

Implementace překladače imperativního jazyka IFJ19

Tým 048 varianta II

11. prosince 2019

Autoři:

Veverka Jiří	xvever12	23% -vedoucí
Halabica Michal	xhalab00	30%
Salih Adam	xsalih01	30%
Rusín Tomáš	xrusin04	17%

Úvod

Tento projekt se zabývá návrhem a implementací jazyka IFJ19, který je zjednodušenou podmnožinou jazyka Python 3. Implementace je realizována v jazyce C. Program je rozdělen na 3 části:

- lexikální analýzu (popsán v souboru scanner.c)
- syntaktickou analýzu se sémantickou analýzou (popsány v souboru parser.c)
- generování cílového kódu (popsán v souboru scanner.c).

Zvolili jsme si variantu II, které mělo za úkol vytvořit tabulku symbolů pomocí tabulky s rozptýlenými položkami.

Obsah

1 Lexikální analýza

Lexikální analýza, dále jen (LA) je implementována pomocí deterministického konečného automatu popsaného v souboru `scanner.h` a implementována v souboru `scanner.c`. Lexikální analýzu popisuje konečný automat jehož graf je v příloze 9.1.

LA je vstupní částí překladače. Zpracovává vstupní zdrojový kód v jazyce IFJ19, který je přiveden na standardní vstup. Načítá vstup po jednom znaku a postupně generuje jednotlivé tokeny a ukládá je do datové struktury ‘List’ s přístupem FIFO.

2 Syntaktická analýza

2.1 Syntaktická analýza shora dolů

Syntaktická analýza (SA) je implementována v souboru `parser.c`. Parser přijímá frontu tokenů vytvořené v Lexikální analýze, které postupně redukuje a kontroluje, zda je program syntakticky správně napsaný. Všechny pravidla, podle kterých se kontroluje, jsou rozdělena do jednotlivých funkcí popsaných v LL tabulce. Následně jsou jednotlivé funkce rekurzivně volány.

Sémantická analýza se provádí současně se syntaktickou analýzou. K provedení sémantické analýzy se používá tabulka symbolů, kde existuje tabulka symbolů pro globální proměnné a tabulka symbolů pro definice funkcí.

Tabulka symbolů (TS) je implementována za pomocí tabulky s rozptýlenými položkami (HashTable). Při vyhledávání v hashtable se skočí na položku o zadaném klíči a vyhledává se v seznamu, dokud se nedojde k nalezení požadované položky. Pokud není funkce, nebo proměnná nalezena, dochází k chybě 3 (nedefinovaná proměnná/funkce).

2.1.1 LL- gramatika

{} jsou pro pravidla gramatiky

<> jsou pro tokeny

{body} → {statement}<EOL>{body}
{body} → ε
{function} → <def> <id> <(>{arg-list}<)><:><EOL><ident>{body}<dedent>
{arg-list} → <id>
{arg-list} → <id><,>{arg-list}
{if} → <if> {expression}<:><EOL><indent>{body}<dedent> {if_else}
{if_else} → <else> <:><EOL><indent>{body}<dedent>
{while} → <while> {expression}<:><EOL><indent>{body}<dedent>
{statement} → <id> <=> {expression} <EOL>
{statement} → <id> <+=> {expression} <EOL>
{statement} → <id> <-=> {expression} <EOL>
{statement} → <id> <*=> {expression} <EOL>
{statement} → <id> </=> {expression} <EOL>
{statement} → {expression}
{statement} → {while}
{statement} → {if}
{statement} → <break>
{statement} → <pass>
{statement} → <return> {expression}
{statement} → <return>
{statement} → ε
{expression} → ε
{expression} → <id> <(>{expression-list}<)>
{expression} → <(>{expression}<)>
{expression} → {expression} <<operator>> {expression}
{expression} → <<raw_val>>
{expression} → <not> {expression}
{expression-list} → {expression}
{expression-list} → {expression}<,> {expression_list}
<<value>> → <id> || <int> || <float> || <string> || <none>
<<operator>> → <+>|| <->|| <*>|| </> || <==>|| <!=> || < > >|| < >= > || < < >|| < <= >|| <and> || <or> || <not>

2.2 Syntaktická analýza (výrazů) zdola nahoru

SA zdola nahoru je založena na základě precedenční analýzy. Precedenční tabulka tvořena operátory je sestavena na základě priority operátorů. Porovnávají se hodnoty na vstupu a na zásobníku, následně je vybrán operátor s vyšší prioritou. Ze zásobníku se postupně načítají tokeny vytvořené v Lexikální analýze a následně jsou redukovány a předávány na postfixový zásobník.

2.2.1 Precedenční tabulka

	* , /, //	+, -	<, <=, >, >=, ==, !=	and, or, not	(	)	$	id
*, /, //	>	>	>	>	<	>	>	<
+, -	<	>	>	>	<	>	>	<
<, <=, >, >=, ==, !=	<	<	>	>	<	>	>	<
and, or, not	<	<	<	>	<	>	>	<
(	<	<	<	<	<	>	x	<
)	>	>	>	>	>	>	>	x
$	<	<	<	<	<	x	x	<
id	>	>	>	>	x	>	>	x

3 Generátor cílového kódu

Cílem generátoru je vytvořit z derivačního stromu, který se vytváří v parseru při syntaktické analýze, výpočetně ekvivalentní kód v jazyce IFJCode19.

3.1 Struktura výsledného programu

Jako první bylo potřeba si zvolit strukturu, od kterého budeme výsledný kód koncipovat. Efektivně jsme si rozdělili kód do funkcí, přičemž vestavěné funkce budou na nejvrchnější části programu, uživatelské funkce se vygenerují pod vestavěné funkce, a poté se do nejspodnější části vygeneruje uživatelský kód na nejspodnější úrovni (main).

3.2 Předávání argumentů a návratových hodnot

Dalším krokem bylo definovat si komunikaci mezi jednotlivými bloky kódu. Napadly nás dvě možnosti:

1. Buď si vytvářet vždy dva framy, kdy by např. dočasný frame by byl uživatelský s uživatelsky definovanými proměnnými a druhý, lokální frame by byl pro potřeby mezivýpočtů.
2. Ukládat si mezivýpočty na datový stack, přičemž by se před zavoláním funkce pushly argumenty funkce na stack a po provedení funkce na stacku zůstane pouze jedna hodnota s výsledkem.

Po delším rozhodování jsme se rozhodli pro druhou možnost, protože nám přišla efektivnější, bližší reálnému assembleru a také jednodušší na implementaci.

3.3 Vyhodnocování výrazů

Pravidla pro vyhodnocování výrazů jsme si definovali stejně jako u funkcí, tedy po vyhodnocení výrazů musí být výsledek vždy uložen na stacku.

3.4 Vestavěné funkce

Pro účely implicitní konverze jsme si vytvořili vestavěné typesafe funkce na provádění aritmeticko-logických a relačních operací, které si kontrolují typovou kompatibilitu.

Typovou kompatibilitu jsme si definovali v tabulkách popsané v příloze 9.2

4 Optimalizace

Při práci na překladači jsme narazili na několik případů, které by se daly optimalizovat, aby výsledný kód byl méně komplexní a zároveň funkčně ekvivalentní. Tyto optimalizace jsme implementovali, ale z důvodu zachycení testovacích případů jsem je v odevzdaném zdrojovém kódu vypnuli.

4.1 Reference counting

Jednou z implementovaných optimalizací je počítání referencí funkcí a proměnných, která probíhala současně při sémantické analýze. Na základě této informace se pak generátor rozhodoval, zda funkci vygenerovat, nebo ji vynechat.

4.2 Nepoužité výrazy

Druhou optimalizací, kterou jsme vypnuli bylo generování výrazů, který se nepřiřazoval žádné proměnné (s výjímkou volání funkce)

5 Datové typy

5.1 Fronta

Frontu s přístupem FIFO jsme implementovali jako lineárně spojitý seznam s odkazy na začátek a konec. Tento datový typ je popsán v souboru ‘List.h’ a její implementace v ‘List.c’.

Strukturu List jsme použili při lexikální analýze na ukládání tokenů, které se později z této datové struktury odebírají při lexikální analýze.

5.2 Zásobník

Strukturu zásobníku s přístupem LIFO jsme simulovali ve struktuře ‘Stack’ popsaného v souboru ‘Stack.h’ a implementovaného v souboru ‘Stack.c’. Tato struktura k simulaci využívá pole prvků `StackItem` a ukazatel na vrchol zásobníku.

Zásobník jsme v projektu využili na dvou místech - na ukládání počtu indentů v lexikální analýze a na ukládání tokenů s operátory v syntaktické analýze výrazů. Z tohoto důvodu jsme se rozhodli implementovat zásobník genericky a implementovali jsme ‘StackItem’ jako union, který může nabývat vícera datových typů.

5.3 Hashovací tabulka

Hashovací tabulku jsme implementovali po vzoru druhého domácího úkolu z předmětu IAL a využili jsme ji v sémantické analýze symbolů proměnných a funkcí.

Vyplatilo se nám tyto dva případy rozdělit a implementovat je jako dvě tabulky s jiným typem obsahu - ‘SymbolMeta’ a ‘FuntionMeta’, které obsahují informace jako počet referencí, nebo počet argumentů.

Tuto genericitu jsme implementovali, stejně jako u zásobníku, pomocí unionů.

Dále jsme si pro účely našeho překladače doimplementovali mj. funkce jako ‘containsKey’, ‘copyHashTable’ nebo ‘mergeTables’.

5.4 Řetězec

Jedna z prvních struktur, které jsme implementovali byla struktura ‘String’, která zapouzdřuje pole znaků do jedné entity - řetězce. Takto definovaný řetězec jsme používali v každé fázi různě. V lexikální analýze se nám osvědčila operace `appendCharacter`, popř `stringEquals`. V syntaktické a sémantické analýze se nám vyplatilo předávání řetězců jako jednu entitu a v generátoru cílového kódu to bylo vytvoření formátovaného stringu s proměnným počtem argumentů.

6 Práce v týmu

Na začátku semestru jsme si domluvili pravidelné schůzky každé úterý. Vzhledem k práci, která postupem semestru přibývala i z jiných předmětu bylo obtížné pravidelné dodržení těchto schůzek. Proto jsme po každé části překladače, která se probrala na přednášce, udělali časově delší schůzku, kde se vytvořila převážná část kódu. Následně jsme si rozebrali zbylé části a vytvářely finální podobu. Ke komunikaci jsme využili Discord a jako verzovací program Git na GitHub.

7 Rozdělení práce

Práce nebyla rozdělena podle jednotlivých částí překladače, ale snažili jsme se plánovat týdenní sprinty, aby každý mohl pracovat na všech částech překladače a tím měl požadované znalosti překladače jako celku. Toto rozdělení mělo další výhodu a to, že nedocházelo k blokaci práce členů jinými členy.

Salih Adam :

Embedded, Generator, List, Node, Parser, Resource, Scanner, Stack, String, Token, main, symtable.

Halabica Michal :

Error, Generator, Parser, Scanner, String, Token

Veverka Jiří :

Generator, Scanner, String

Rusín Tomáš :

Scanner, Instructions, Dokumentace

8 Závěr

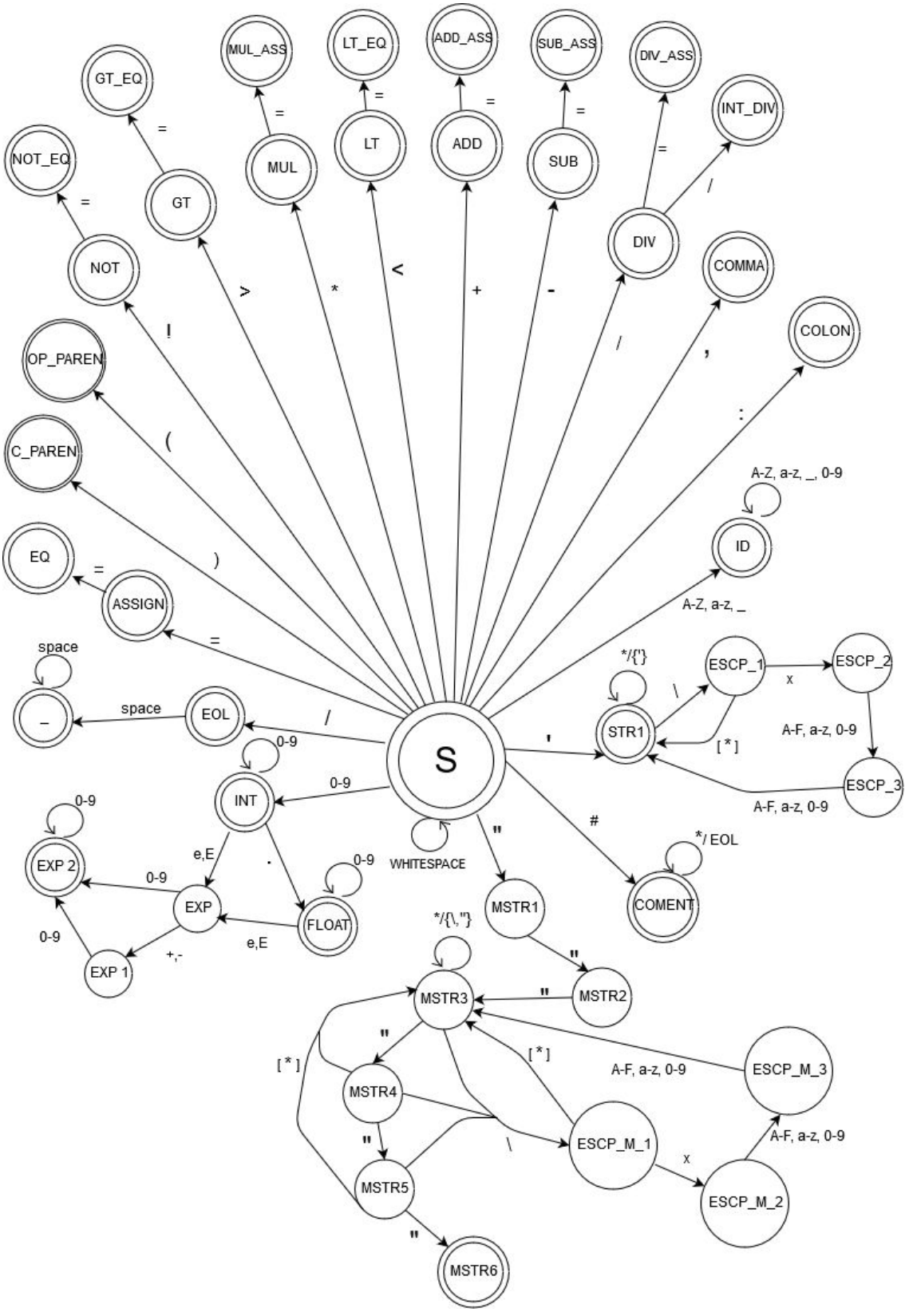
Projekt byl velmi časově náročný, ale pro první pokusné odevzdání byl program z větší části hotov. Pro druhé technické odevzdání jsme vytvořili více testů a povedlo se nám zlepšit lexikální analýzu, spolu s tím celkové hodnocení. Díky projektu jsme si mohli ověřit znalosti získané na přednášce a smysluplně je aplikovat.

9 Zdroje

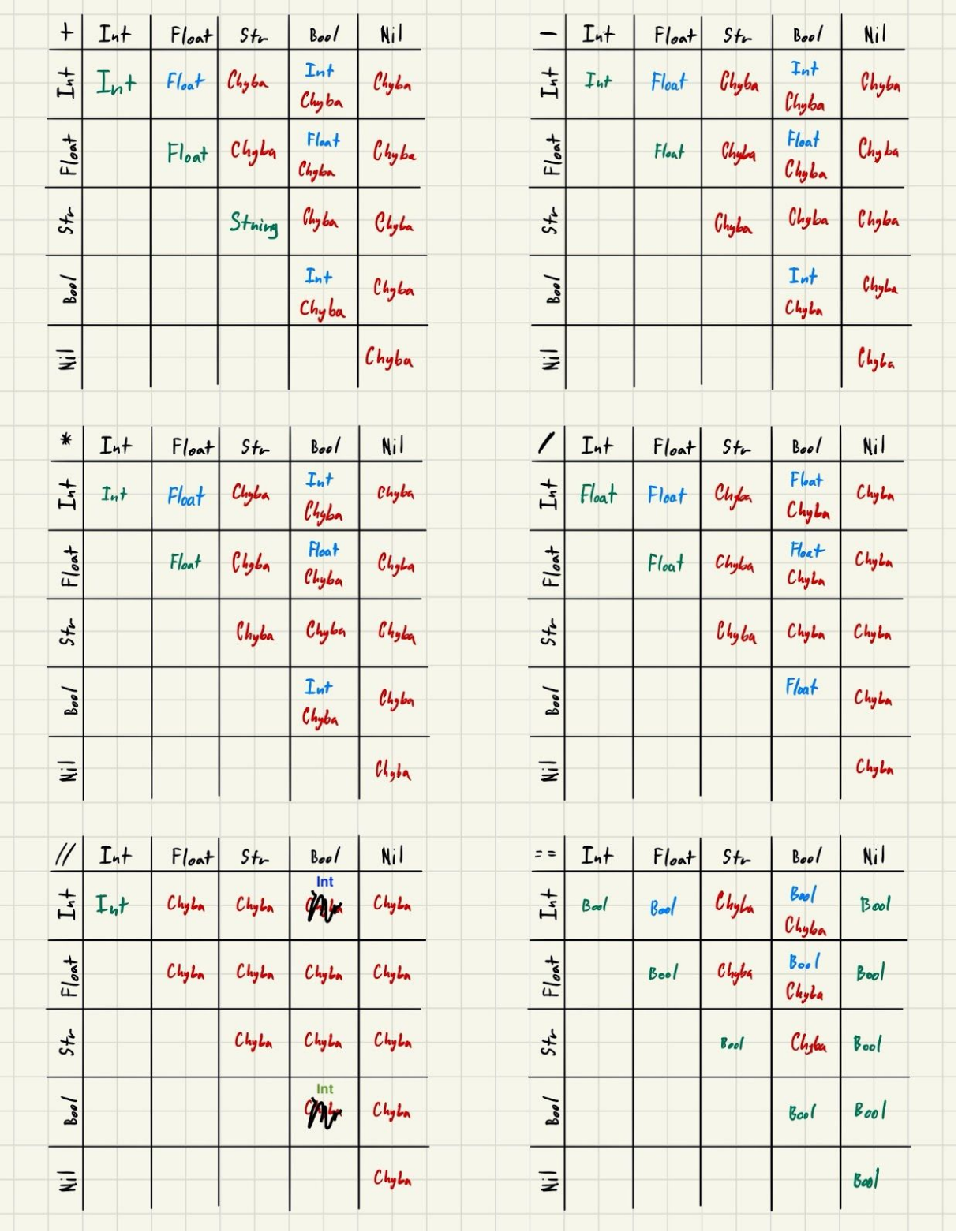
[1] Alexander Meduna a Roman Lukáš. Přednášky Formální jazyky a překladače [online]. [cit. 2019-12-11]. Dostupné z: http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/

10 Přílohy

10.1 Konečný automat lexikálního analyzátoru



10.2 Tabulka typové kompatibility



+	Int	Float	Str	Bool	Nil
Int	Int	Float	Chyba	Int Chyba	Chyba
Float		Float	Chyba	Float Chyba	Chyba
Str			String	Chyba	Chyba
Bool				Int Chyba	Chyba
Nil					Chyba

−	Int	Float	Str	Bool	Nil
Int	Int	Float	Chyba	Int Chyba	Chyba
Float		Float	Chyba	Float Chyba	Chyba
Str			Chyba	Chyba	Chyba
Bool				Int Chyba	Chyba
Nil					Chyba

*	Int	Float	Str	Bool	Nil
Int	Int	Float	Chyba	Int Chyba	Chyba
Float		Float	Chyba	Float Chyba	Chyba
Str			Chyba	Chyba	Chyba
Bool				Int Chyba	Chyba
Nil					Chyba

/	Int	Float	Str	Bool	Nil
Int	Float	Float	Chyba	Float Chyba	Chyba
Float		Float	Chyba	Float Chyba	Chyba
Str			Chyba	Chyba	Chyba
Bool				Float	Chyba
Nil					Chyba

//	Int	Float	Str	Bool	Nil
Int	Int	Chyba	Chyba	Int [illegible]	Chyba
Float		Chyba	Chyba	Chyba	Chyba
Str			Chyba	Chyba	Chyba
Bool				Int [illegible]	Chyba
Nil					Chyba

==	Int	Float	Str	Bool	Nil
Int	Bool	Bool	Chyba	Bool Chyba	Bool
Float		Bool	Chyba	Bool Chyba	Bool
Str			Bool	Chyba	Bool
Bool				Bool	Bool
Nil					Bool

!=	Int	Float	Str	Bool	Nil
Int	Bool	Bool	Chyba	Bool	Bool
Float		Bool	Chyba	Bool	Bool
Str			Bool	Chyba	Bool
Bool				Bool	Bool
Nil					Bool

<	Int	Float	Str	Bool	Nil
Int	Bool	Bool	Chyba	Bool Chyba	Chyba
Float		Bool	Chyba	Bool Chyba	Chyba
Str			Bool	Chyba	Chyba
Bool				Bool	Chyba
Nil					Chyba

>	Int	Float	Str	Bool	Nil
Int	Bool	Bool	Chyba	Bool Chyba	Chyba
Float		Bool	Chyba	Bool Chyba	Chyba
Str			Bool	Chyba	Chyba
Bool				Bool	Chyba
Nil					Chyba

and	Int	Float	Str	Bool	Nil
Int	Chyba	Chyba	Chyba	Chyba	Chyba
Float		Chyba	Chyba	Chyba	Chyba
Str			Chyba	Chyba	Chyba
Bool				Bool	Chyba
Nil					Chyba

and	Int	Float	Str	Bool	Nil
Int	Chyba	Chyba	Chyba	Chyba	Chyba
Float		Chyba	Chyba	Chyba	Chyba
Str			Chyba	Chyba	Chyba
Bool				Bool	Chyba
Nil					Chyba

not	Int	Float	Str	Bool	Nil
Int	Chyba	Chyba	Chyba	Chyba	Chyba
Float		Chyba	Chyba	Chyba	Chyba
Str			Chyba	Chyba	Chyba
Bool				Bool	Chyba
Nil					Chyba