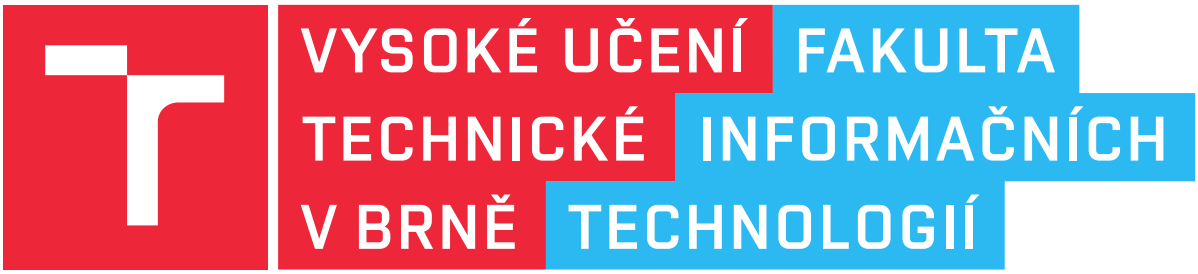


Dokumentace k projektu z IFJ a IAL

Tým xbielg00, varianta TRP

Gabriel Biel	**xbielg00**	30%
Adam Gabrys	xgabry01	15%
Jakub Mikyšek	xmikys03	25%
David Tobolík	xtobol06	30%

Obsah

1 Úvod **2**

2 Popis fungování a struktura překladače **2**

3 Lexikální analýza **2**

4 Syntaktická analýza **2**
4.1 Rekurzivní sestup 2
4.2 Precedenční analýza 3

5 Tabulka symbolů **3**

6 Abstraktní syntaktický strom **3**

7 Zásobníky **4**

8 Sémantické kontroly **4**

9 Generování kódu **4**
9.1 Kontext programu 5
9.2 Statické funkce a definice proměnných 5
9.3 Generování konkrétních částí programu 5
9.3.1 Podmínky – `if else, while` 5
9.3.2 Přiřazení 5
9.3.3 Volání funkce 5
9.3.4 Deklarace funkce 5
9.3.5 Volání return 5
9.3.6 Výrazy 6
9.3.7 Konec složeného příkazu 6
9.4 Pomocné funkce pro vygenerovaný kód 6

10 Týmová spolupráce **6**
10.1 Komunikace v týmu 6
10.2 Verzovací systém 6
10.3 Testování 6
10.4 Rozdělení práce a bodů 7
10.5 Důvod nerovnoměrného rozdělení bodů 7

11 Závěr **7**

12 Přílohy **8**

1 Úvod

Cílem našeho projektu bylo vytvoření překladače jazyka IFJ22, který je poupravenou podmnožinou jazyka PHP. Námi vytvořený překladač jsme implementovali v jazyce C a má za úkol daný zdrojový kód načíst a zpracovat několika způsoby, kdy je zapotřebí využití lexikální, syntaktické, precedenční a sémantické analýzy s následným přeložením do mezikódu IFJcode22, který již zpracovává předem poskytnutý interpret, který tento mezikód provádí.

2 Popis fungování a struktura překladače

Překladač se skládá z několika propojených spolu fungujících částí, kdy každá z nich zastává důležitou roli. Jde tedy o lexikální, syntaktickou, sémantickou a precedenční analýzu, po jejímž průchodu zastává důležitou část také generování mezikódu IFJcode22. Bylo nutné si překladač rozdělit do těchto částí, aby bylo jasné, které funkce překladače daná část zastává a lépe se nám dělila práce v týmu.

3 Lexikální analýza

Lexikální analýza (dále LA) načítá lexémy ze standardního vstupu pomocí funkce `getToken()`, ty následně přetváří na tokeny, které postupně naplňují list `TokenList`. List se po zpracování vstupu předává syntaktické analýze. LA se volá funkcí `lexAnalyser()` a je implementovaná v souborech `lex_analyzer.c` a `lex_analyzer.h`. Zpracování lexémů se řídí předem vytvořeným deterministickým konečným automatem (viz 1). Konečný automat je reprezentován v LA switchem, u kterého každý case zastupuje jeden jeho stav. Pokud automat natrefí na nesouvisející vstupní hranu a nachází se v konečném stavu, spouští se funkce `tokenCtor()` pro vytvoření tokenu a poté `appendToken()` pro vložení do seznamu tokenů. Pokud se ale automat nenachází v konečném stavu, LA hlásí lexikální chybu a překlad se ruší. O tokenu se uchovává informace o jeho typu, číslo řádku – na kterém se token nalézal a relevantní data (název proměnné, funkce, hodnota atd.).

4 Syntaktická analýza

4.1 Rekurzivní sestup

Rekurzivní sestup se volá funkcí `synAnalyser()` a je implementovaný v souborech `syn_analyzer.c` a `syn_analyzer.h`. Řídí se podle navržené gramatiky (viz 2), kde každé pravidlo představuje vlastní funkci. Funkce postupně prochází tokeny získané od lexikální analýzy, kontroluje jestli je vše gramaticky správně, naplňuje tabulku symbolů a tvoří abstraktní syntaktický strom (dále ASS). Při nalezení výrazu volá funkci `parseExpression`, která spouští precedenční analýzu. Při nalezení chyby se nastaví příslušný `errorCode` a chyba se v rekurzi propaguje dál, aby program ukončil kontrolu. Funkce vrací vytvořený ASS, naplněnou tabulku symbolů a informaci o tom, jestli rekurze proběhla syntakticky a sémanticky správně či nikoliv.

4.2 Precedenční analýza

Soubory `preced_analyzer.c`, `preced_analyzer.h`, `preced_analyzer-data.h`

Na zpracování výrazů jsme použili metodu shora-dolů - precedenční analýzu (dále jen PA). PA kontroluje sémantiku volání funkcí a zda byly použité proměnné definovány. Ke kontrole využívá tabulku symbolů. Správnou syntaxi zadaného výrazu kontroluje pomocí zásobníku PA a precedenční tabulky (viz 1). Výrazy se na zásobníku PA redukují pomocí pravidel, které jsou definovány v souboru `preced_analyzer.c`. Výstupem PA je post-fixově zpracovaný výraz na zásobníku ASS(6).

5 Tabulka symbolů

Soubory `symtable.c`, `symtable.h`

Pro implementaci tabulky symbolů jsme si ze zadání vybrali variantu TRP, tedy tabulku s rozptýlenými položkami, kdy jsme při jejím tvoření využili získané znalosti, jak z přednášek, tak z projektu do předmětu IAL. Pro práci s položkami jsme využívali hlavně funkci `ht_insert()`, kdy při vkládání položky do tabulky jsme rozdělovali, zda vkládáme funkci, či proměnnou. Obecně jsme si ukládali ID - unikátní řetězec, který slouží jako klíč. Dále počet referencí na položku a odkaz na další položku v tabulce. Pokud byla daná položka funkce, ukládali jsme si navíc její návratový typ a informace o parametrech jí přiřazené, to pomocí funkce `ht_paramAppend()`. Pro správnou funkcionalitu tabulky jsme museli samozřejmě implementovat i další funkce, jako například odstranění položky z tabulky/odstranění všech položek (funkce `ht_delete()`, `ht_delete_all()`) a ostatní, které jsou detailně popsány v souboru `symtable.h` a implementovány v souboru `symtable.c`.

6 Abstraktní syntaktický strom

Soubory `ast.c`, `ast.h`

Abstraktní syntaktický strom (dále ASS) je implementovaný jako zásobník (viz 7), který obsahuje strukturu pro ASS. Ta je tvořena typem a volitelnými daty.

- Aritmetické, řetězcový a relační operátory – každý má svůj typ, nemají data, na zásobníku předcházejí 2 operandy.
- Přiřazení – nemá data, na zásobníku následuje proměnná, do které se přiřazuje a po ní výraz nebo volání funkce, které se má přiřadit.
- Proměnná – v datech obsahuje ukazatel do tabulky symbolů.
- Konstanty – každý typ konstanty má svůj typ v ASS (int, string, float, null) a obsahuje, kromě typu null, hodnotu této konstanty.
- Deklarace funkce – v datech je uložený ukazatel do tabulky symbolů na tuto funkci.
- Volání funkce – v datech obsahuje ukazatel na strukturu volání funkce, v ní je uložen ukazatel do tabulky symbolů a seznam parametrů. Parametry mají rovněž typ (proměnná nebo konstanta – int, string, float, null) a svoje data (ukazatel do tabulky symbolů nebo hodnota konstanty).
- Volání return – typ bez návratové hodnoty nebo s návratovou hodnotou, v tom případě na zásobníku následuje výraz, který se má vrátit.

- Podmínky – if, else, while – nemají data, na zásobníku následuje výraz, který je v podmínce.
- Konec výrazu – značí konec výrazu nebo volání funkce, slouží k rozpoznání konce postfixového zápisu.
- Konec složeného výrazu – značí konec složeného příkazu ve složených závorkách (if, else, deklarace funkce).

Řídící struktury jsou uloženy prefixově, výrazy postfixově, toho se s výhodou využívá v generování kódu (viz 9).

7 Zásobníky

Soubory `stack.c`, `stack.h`

Zásobníky byly potřeba na více místech (ASS, precedenční analýza, generování kódu, . . .). Implementace zásobníků je tedy obecná pomocí makra, které je možné použít pro libovolný ukazatel. Pro operaci pop je potřeba v makru specifikovat destruktor, který správně uvolní prvek, což je vhodné především pro komplexnější struktury. Při tvorbě ASS je potřeba, aby byl začátek programu na vrcholu zásobníku, proto je zásobník rozšířen o funkci, která umožňuje vložení prvku na konec zásobníku, aby se ASS do zásobníku mohl vkládat sekvenčně.

8 Sémantické kontroly

Všechny sémantické kontroly probíhají s pomocí tabulky symbolů. Před spuštěním syntaktické analýzy, proběhne tzv. „první průchod“ za pomocí funkce `fncDeclarationTable`, která kontroluje jen definice funkcí a nahrává je společně s jejich parametry a návratovým typem do tabulky symbolů. V návaznosti na tento první průchod se v precedenční analýze při volání funkcí provádí kontrola typu a počtu parametrů nebo zda byla funkce definována. Sémantické kontroly pro proměnné a jejich (ne)definování se provádějí v syntaktické a precedenční analýze.

9 Generování kódu

Soubory `code_gen.c`, `code_gen.h`, `code_gen_static.c`, `code_gen_static.h`

Generování kódu zajišťuje funkce `codeGenerator`, která slouží jako kontroler celého generování. Předává si řízení s funkcemi, které slouží pro generování konkrétních částí, stará se o správu kontextu programu, uvolnění dat apod.

Byly domluveny konvence předávání hodnot volání funkcí, pomocných funkcí (viz 9.4) a vyhodnocených výrazů. Parametry vestavěných funkcí a funkcí definovaných uživatelem (kromě funkce write, která se generuje přímo) se předávají v předdefinovaných proměnných ve vytvořeném lokálním rámci. Lokální rámec připravuje a uklízí vždy volající. Pomocné funkce očekávávají parametry vždy na zásobníku, toto řešení jsme zvolili pro omezení vytváření rámců, s ohledem na malý počet parametrů těchto funkcí, které berou 1 nebo 2 parametry. Návratová hodnota se u všech funkcí a u výrazů předává na zásobníku.

9.1 Kontext programu

Funkce `codeGenerator` vytváří a aktualizuje kontext programu a předává ho některým funkcím. Kontext obsahuje čítače návěští, zásobník zanoření a ukazatel na aktuálně deklarovanou funkci. Zásobník zanoření obsahuje prvky, které identifikují typ a pořadí složených příkazů.

9.2 Statické funkce a definice proměnných

Před spuštěním veškerého generování kódu jsou vygenerovány statické části programu a definice proměnných. Statická část obsahuje definici globálních pomocných proměnných, které slouží místo registrů a definici vestavěných a pomocných funkcí. Tato část je přeskočena instrukcí `JUMP` až na začátek programu. Na začátku se vytvoří nový lokální rámec a nadefinují se všechny proměnné, které jsou použity v hlavním těle programu (i v podmínkách a cyklech, tudíž všude kromě deklarace funkcí).

9.3 Generování konkrétních částí programu

9.3.1 Podmínky – `if else, while`

Podmínka `if` se na začátku vyhodnotí a v případě nepravdivosti výrazu v podmínce se skočí na návěští `else` doplněné pořadovým číslem z kontextu. Před návěští `else` se vygeneruje skok za složený příkaz. U cyklu `while` se vygeneruje návěští začátku cyklu (opět doplněné o pořadové číslo z kontextu), při neplatnosti podmínky skočí až za konec cyklu. Návěští konce složených příkazů `else` a `while` jsou generovány na konci složeného příkazu (viz 9.3.7).

9.3.2 Přiřazení

Nejdříve se vyhodnotí výraz, který se má přiřadit a potom se přiřadí výsledek ze zásobníku pomocí instrukce `POPS`.

9.3.3 Volání funkce

Před voláním funkce se vytvoří dočasný rámec, ve kterém se nadefinují a inicializují argumenty funkce. Před každým přiřazením proměnné do parametru se zkontroluje její inicializace. Zavolá se funkce, případně se přiřadí hodnota ze zásobníku a uklidí se rámec.

9.3.4 Deklarace funkce

Vygeneruje se příslušné návěští a definice lokálních proměnných použitých ve funkci, kromě jejich parametrů (ty jsou definovány při volání funkce). Celá deklarace se přeskočí instrukcí `JUMP`, aby její obsah nezasahoval do hlavní části programu.

9.3.5 Volání return

Podle kontextu se vygeneruje `EXIT` nebo návrat z funkce. Před navrácením hodnoty se zkontroluje její typ, poté se vloží na zásobník a vygeneruje se `RETURN`.

9.3.6 Výrazy

Výrazy využívají postfixového zápisu v ASS. Jednotlivé operandy se vkládají na zásobník a redukují operacemi na mezivýsledky případných dalších operací. Před každým použitím proměnné, se kontroluje její inicializace.

9.3.7 Konec složeného příkazu

Když je na vrcholu zásobníku ASS konec složeného výrazu, vyhodnotí se následující akce podle typu složeného výrazu v kontextu.

- `if` – vygeneruje skok na konec složeného příkazu `else`.
- `else` – generuje návěští značící konec příkazu `else`.
- `while` – generuje skok na začátek cyklu a návěští konce cyklu.
- deklarace funkce – vygeneruje kontrola návratu z funkce, v případě `void` funkce se vygeneruje navrácení z funkce. Aktualizuje se kontext programu.

9.4 Pomocné funkce pro vygenerovaný kód

Ve statické části se generují pomocné funkce, které se potom volají na příslušných místech v programu. Jedná se o implicitní typové konverze, vyhodnocování podmínek, kontrola typů apod.

10 Týmová spolupráce

10.1 Komunikace v týmu

Vzájemná domluva a komunikace probíhala online i offline. A to prostřednictvím Discord serveru – pro převážně obecnější problematiky, pro ty složitější a již více konkrétní nám sloužila funkce GitHub Issues, kde jsme mohli jednoduše diskutovat o již konkrétní části kódu. K tomu jsme se každý čtvrtek setkávali na schůzích uskutečněných v budově školy, kde jsme řešili ty nejzásadnější problémy a rozhodnutí z hlediska směřování.

10.2 Verzovací systém

Při výběru verzovacího systému pro nás bylo jasnou volbou zvolit populární a osvědčený Git s nahráváním na server GitHub, kde probíhala i již zmíněná vzájemná komunikace.

10.3 Testování

Pro ověření správnosti jsme používali framework GoogleTest v jazyce C++, který nám sloužil k unit testování. Co se týče integračního testování, zde jsme zvolili cestu manuálních testů.

10.4 Rozdělení práce a bodů

- Gabriel Biel (xbielg00)
  - architektura překladače, lexikální analýza, precedenční tabulka, precedenční analýza, testy pro precedenční analýzu, rozhraní pro tabulku symbolů, funkce v generátoru kódu
  - **30%**
- Adam Gabrys (xgabry01)
  - Konečný automat, sémantické kontroly pro deklarace funkcí, generování built-in funkcí, dokumentace
  - **15%**
- Jakub Mikyšek (xmikys03)
  - gramatika, LL tabulka, rekurzivní syntaktická analýza, sémantické kontroly pro proměnné, integrační testování pro parser, dokumentace
  - **25%**
- David Tobolík (xtobol06)
  - sémantické kontroly pro volání funkce, generování kódu, datové struktury pro ASS, zásobníky, pomocná makra a funkce, struktura projektu, cmake a struktura testů, testy (lexikální analýza, ASS, zásobníky), code review, makefile, dokumentace
  - **30%**

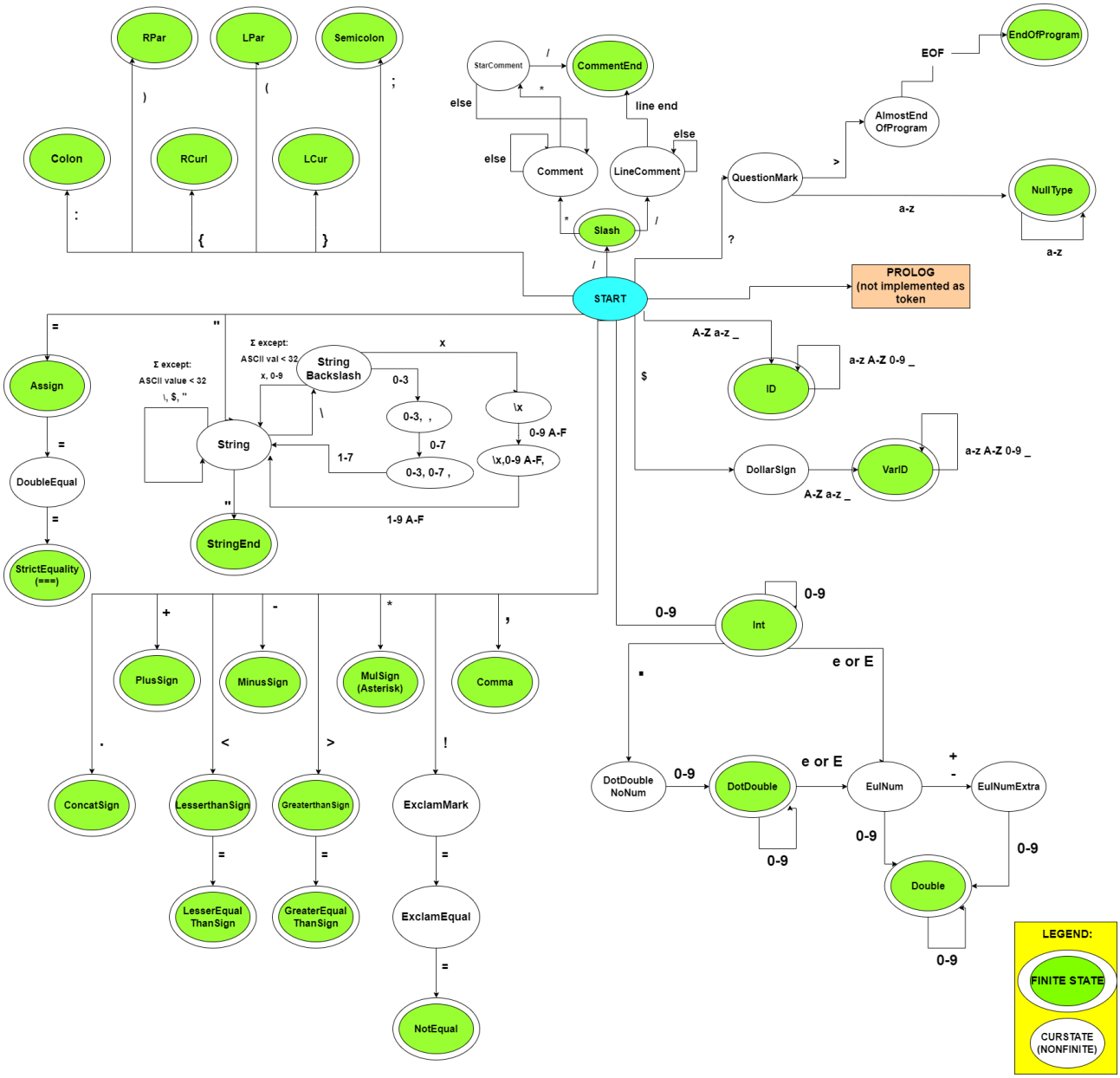
10.5 Důvod nerovnoměrného rozdělení bodů

Pro nerovnoměrné rozdělení bodů jsme se rozhodli jelikož někteří členové týmu odvedli značně větší část práce. Všichni členové týmu se na tomto rozdělení shodli.

11 Závěr

Projekt nás svým objemem ze začátku opravdu překvapil a chvíli nám trvalo, než jsme se zorientovali v zadání. Naštěstí nám opravdu pomohlo scházet se každý týden a prodiskutovat jednotlivé problémy, či náš pokrok v práci. Velkou část sehrálo také to, že jsme se všichni znali a komunikovali vcelku bez prodlev a problémů a to jak na GitHubu, tak na Discord serveru, který jsme si pro projekt vytvořili(viz 10). Práce nám s odstupem času šla stále lépe, jelikož jsme rozuměli dané problematice i specifikacím zadání stále více. Projekt nám přinesl nejen znalosti do předmětů IFJ a IAL, ale velmi nás posunul v práci s verzovacím systémem a v celkové práci v týmu na větších projektech, jako je tento.

12 Přílohy



Obrázek 1: Deterministický konečný automat stavů.

1. <prog> -> <prolog> <seq-stats> <epilog>
2. <prolog> -> <?php <wspace> declare(strict_types=1);
3. <seq-stats> -> <stat> <seq-stats>
4. <seq-stats> -> <fnc-decl> <seq-stats>
5. <seq-stats> -> ε
6. <fnc-decl> -> function functionId (<param>) : <fnc-type> { <st-list> }
7. <st-list> -> <stat> <st-list>
8. <st-list> -> ε
9. <stat> -> if (<expr>) { <st-list> } else { <st-list> }
10. <stat> -> while (<expr>) { <st-list> }
11. <stat> -> <assign> ;
12. <stat> -> return <expr> ;
13. <stat> -> return ;
14. <fnc-type> -> void
15. <fnc-type> -> <type>
16. <param> -> <type> <var> <params>
17. <param> -> ε
18. <params> -> , <type> <var> <params>
19. <params> -> ε
20. <type> -> int
21. <type> -> string
22. <type> -> float
23. <type> -> ?int
24. <type> -> ?string
25. <type> -> ?float
26. <assign> -> <expr>
27. <assign> -> <var> <r-side>
28. <r-side> -> = <expr>
29. <r-side> -> ε
30. <var> -> $varId
31. <epilog> -> ?>EOF
32. <epilog> -> EOF

Obrázek 2: LL gramatika.

	<?php	function	if	while	$varId	return	return;	void	int	string	float	?int	?string	?float	,	=	?>	EOF	$
<prog>	1																		
<prolog>	2																		
<seq-stats>		4	3	3	3	3	3												5
<fnc-decl>		6																	
<st-list>			7	7	7	7	7												8
<stat>			9	10	11	12	13												
<fnc-type>								14	15	15	15	15	15	15					
<param>									16	16	16	16	16	16					17
<params>															18				19
<type>									20	21	22	23	24	25					
<assign>					27														
<r-side>																29			30
<var>					31														
<epilog>																	32	33	

Obrázek 3: LL tabulka.

	$\times$	$/$	$+$	$-$	$\cdot$	$<$	$>$	$\geq$	$\leq$	$\neq$	$=$	$($	$)$	i	**$**
$\times$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$/$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$+$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$-$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$\cdot$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$
$<$	$<$	$<$	$<$	$<$	$<$					$>$	$>$	$<$	$>$	$<$	$>$
$>$	$<$	$<$	$<$	$<$	$<$					$>$	$>$	$<$	$>$	$<$	$>$
$\geq$	$<$	$<$	$<$	$<$	$<$					$>$	$>$	$<$	$>$	$<$	$>$
$\leq$	$<$	$<$	$<$	$<$	$<$					$>$	$>$	$<$	$>$	$<$	$>$
$\neq$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$			$<$	$>$	$<$	$>$
$=$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$			$<$	$>$	$<$	$>$
$($	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$=$	$<$	
$)$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$		$>$		$>$
i	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$		$>$		$>$
$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$		$<$	

Tabulka 1: Precedenční tabulka.