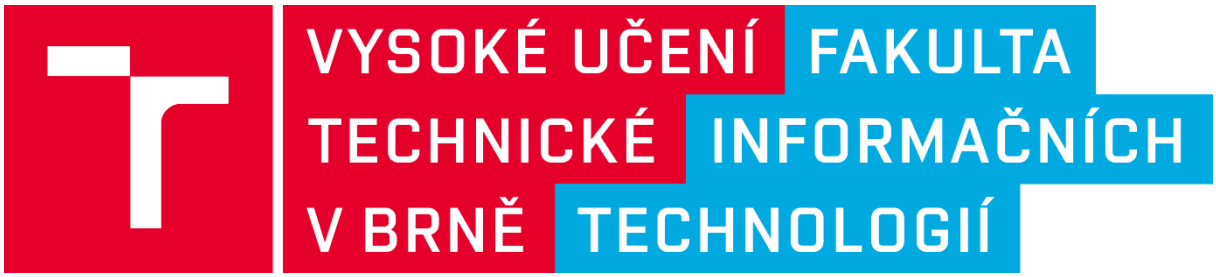


Projektová dokumentace

Implementace překladače pro imperativní jazyk IFJ23

Tým xblaze38

Varianta vv-BVS

Michal Blažek	xblaze38	50 %
Kryštof Michálek	xmicha94	30 %
Matyáš Sapík	xsapik02	20 %
Ondřej Závodský	xzavod16	0 %

5. prosince 2023

Obsah

1 Úvod

Cílem projektu bylo vytvořit překladač jazyka IFJ23 do cílového kódu v jazyce IFJcode23. Jazyk IFJ23 je podmnožinou jazyka Swift.

Překladač je pouze konzolová aplikace, která dostane na vstup kód v jazyce IFJ23 a převede ho na výstup v jazyce IFJcode23, přičemž zkontroluje vstupní kód a případně vrátí chybovou hodnotu pro odpovídající chybový stav.

2 Návrh

2.1 Lexikální analýza

Lexikální analýza je jedna z prvních částí projektu. Je vhodné ji mít hotovou co nejdříve z důvodu kompatibility v dalšími částmi projektu jako je například syntaktický analyzátor.

Lexikální analýza je uložena v souborech scanner.c a scanner.h a její jedinou funkcí, kterou využívají ostatní části projektu je getToken(). Funkce getToken() čte postupně ze standartního vstupu předem neurčený počet znaků, skládá je za sebe a tím vytváří tokeny. Tyto tokeny mají svůj formát uložený ve struktuře Token a jejich obsah je podle typu načtených znaků uložen do union tokenAttrib. Lexikální analyzátor má také za úkol určit typ tokenu, který je uložen v enum tokenType a případně i v enum tokenSecondType. To dělá pomocí koncových stavů konečného automatu. Jelikož je lexikální analyzátor schopný rozlišit stav, ve kterém se nachází, je schopný podle toho i určit typ tokenu.

Náš lexikální analyzátor je implementován pomocí konečného automatu, jehož diagram je na straně XXX. Tento konečný automat je nekonečný cyklus, který přepíná pomocí přepínače mezi stavy tohoto automatu. V případě načtení maximálního možného tokenu zkontroluje, zda se nachází v koncovém stavu. Pokud se nenachází v koncovém stavu, vrací chybovou návratovou hodnotu 1, a pokud se nachází v koncovém stavu, vrátí token pomocí parametru funkce getToken().

Další funkcí pro tento problém je funkce isKeyword(), která při načítání identifikátorů kontroluje, jestli daný identifikátor není některé z klíčových slov jazyka IFJ23. Jestliže funkce určí identifikátor jako klíčové slovo, přepíše jeho typ.

2.2 Syntaktická analýza

Syntaktická analýza se v projektu dělí na syntaktickou analýzu pomocí LL-gramatiky a na precedenční syntaktickou analýzu.

2.2.1 Syntaktická analýza založená na LL-gramatice

Syntaktická analýza založena na pravidlech z LL-gramatiky, která je na straně XXX a je obsažena v souborech parser.c a parser.h.

V LL-gramatice jsou pravidla, která jsou modelována funkcemi. Funkce se většinou jmenují podle levé strany pravidel. V těchto funkcích je rozdělení na jednotlivá pravidla pomocí if, kde každé pravidlo je poté celé zpracované a následně vrací návratovou hodnotu 0 v případě, že je vše v pořádku, nebo jinou než 0 podle typu chyby, která při překladu nastala. Při zpracovávání těchto pravidel se volají i ostatní pravidla, která vrací svoji návratovou hodnotu pravidlu, která je volalo. Jelikož my celý program rozvíjíme rekurzivně pomocí určitých pravidel, jedná se o metodu rekurzivního sestupu.

Během zpracovávání pravidel se kontrolují tokeny, o které syntaktická analýza žádá lexikální analýzu pomocí funkce getToken(). V některých případech (nejčastěji u tzv. epsilon pravidel) se načte následující token a po návratu do předchozí funkce se může stát, že by se načetl další token a předchozí by se vynechal, což by rozhodilo celou syntaktickou analýzu. Tento problém je řešen pomocí návratové

hodnoty -1, která stejně jako 0 značí správnou syntaxi, ale ještě k tomu udává, že další token je už načtený a místo čtení se má zpracovávat.

2.2.2 Precedenční syntaktická analýza

Precedenční syntaktická analýza se stará o zpracovávání výrazů v souborech expression.c a expression.h.

Precedenční syntaktická analýza je založena na precedenční tabulce (nalezneme ji na straně XXX), kde pro operátory, identifikátory a jakýsi znak označující začátek a konec výrazu (v našem případě je to $) máme pravidla. Tato pravidla využívají vždy jeden terminál z vrcholu zásobníku a jeden token ze vstupního řetězce a podle nich určují, jakým způsobem se bude dále postupovat. Jestliže precedenční tabulka určí redukci, pro terminály na zásobníku se naleznou odpovídající pravidla pomocí funkce token2Rules() a tato pravidla jsou definována ve struktuře precedenceTableRulesStruct. Dojde ke kontrole, zda pro použité pravidlo je výraz syntakticky správný (samozřejmě se během toho provádí i sémantická analýza) a pokračuje se dále ve zpracovávání tohoto výrazu.

Pro zpracovávání výrazů máme 2 hlavní funkce expression() a valueExpression(). První zmíněná funkce slouží pro zpracovávání podmínky if nebo while, kde výsledek musí být pravdivostní hodnota, tudíž očekává ve výrazu relační nebo porovnávací operátor. Výjimku tvoří výraz „let ID“, který je validní a má speciální význam. Funkce valueExpression() se stará o zpracovávání výrazů v příkazech přiřazení, v parametrech volání funkce nebo v příkazu return při návratu z funkce.

2.3 Sémantická analýza

Sémantická analýza je v založena na datech z tabulek symbolů. Najdeme ji v souborech parser.c a expression.c.

Sémantická analýza řeší v projektu převážně kontrolu, zda je funkce nebo proměnná definovaná, jestli nepřepisujeme hodnotu nemodifikovatelné proměnné, nebo zda existuje v daném případě správná typová kompatibilita. Typová kompatibilita je nejrozsáhlejší, protože se řeší u každého volání tabulky symbolů, při definování nových proměnných a funkcí s návratovou hodnotou a například i během zpracovávání výrazů.

V projektu je také možné na určitých místech provádět implicitní konverzi datových typů během překladu, což dovoluje převést datový typ Int na Double, a i s takovými případy musí sémantická analýza počítat a správně určit kompatibilitu datových typů v daném případu.

3 Implementace

3.1 Datové struktury

V projektu je využito jen pár abstraktních datových struktur. Za zmínku stojí 2 zásobníky v souborech stack.c a stack.h.

První je ve struktuře SymtabStack, který byl původně připraven pro ukládání aktivních tabulek symbolů. Tento zásobník se bohužel nepodařilo rozumně vsadit do zbytku projektu.

Druhým zásobníkem je zásobník tokenů, který je využit v souboru parser.c pro případné uložení tokenů, které bychom mohli potřebovat při zpracovávání výrazu. Dále se používá v souboru expression.c, kde si během zpracovávání výrazů ukládáme jednotlivé tokeny na zásobník.

Oba tyto zásobníky mají základní funkce Init, Top, Pop, Push a Dispose. Zásobník na tokeny má ještě funkce tStack_Second_Top() a tStack_Insert(), které slouží pro přečtení 2. prvku odshora a vložení na jiné místo v zásobníku než na vrchol.

3.2 Tabulka symbolů

Tabulku symbolů jsme podle zadání implementovali jako výškově vyvážený binární vyhledávací strom. Tabulka symbolů je uložena v souborech symtable.c a symtable.h. Node je struktura označující jednotlivé uzly stromu a aby to byl binární strom, tak ukazuje na 2 podstromy (levý a pravý). Každý uzel obsahuje jméno proměnné nebo funkce, datový nebo návratový typ, příznak func určující, zda je to funkce, příznak const_let, který udává, jestli je tento uzel proměnná definovaná pomocí let, a nakonec jsou zde uložené parametry funkcí ve struktuře Param. Tato struktura se skládá z jména, pomocí kterého ho voláme, jména ve funkci a datového typu parametru.

Tabulka symbolů obsahuje základní funkce initTree(), searchNode(), insertNode() a disposeTree(). Dále zde můžeme nalézt i funkci balanceTree(), která se snaží o výškové vyvážení stromu.

V tomto projektu je tabulek symbolů více, a to jedna globální a předem nedefinovaný počet lokálních tabulek symbolů. Globální tabulka symbolů se vytváří na začátku překladače a obsahuje definice funkcí a globálních proměnných. Lokální tabulky se vždy vytvoří, pokud vstoupíme při překladu do nějakého bloku (například if, else, while, funkce). Tyto lokální tabulky se samozřejmě vymažou po výstupu z daného bloku. Jestliže máme více zanořených bloků v sobě, definujeme pro každý z nich jednu lokální tabulku symbolů a pokud poté potřebujeme vyhledat v tabulkách symbolů například nějakou proměnnou, postupujeme od lokální tabulky představující nejvíce zanořený blok směrem ke globální tabulce symbolů.

3.3 Generátor cílového kódu

Generátor cílového kódu je sepsán v souborech code_generator.c a code_generator.h. V těchto souborech můžeme najít funkci generate_code(), která je implementována jako přepínač. Tento přepínač na základě informací, co dostane, určí, jaký blok kódu je právě překládán a podle toho vypíše na stdout seznam prováděných instrukcí v jazyce IFJcode23 pro daný blok.

Funkce generate_code() je v kódu volána v souborech parser.c a expression.c na místech, kde se zpracovává nějaký blok kódu nebo výraz.

4 Práce v týmu

4.1 Komunikace v týmu

Komunikace probíhala prostřednictvím aplikace Discord, případně některé problémy jsme řešili i osobně mezi určitými členy týmu. Dále jsme měli týmový meeting před zahájením implementace projektu, kde jsme si vybrali formu komunikace mezi členy, verzovací systém a částečně přerozdělili práci mezi členy.

4.2 Využití verzovacího systému

Pro tento projekt jsme si vybrali verzovací systém Git a pro správu souborů je využili GitHub. Pro každého z nás to byla první zkušenost s prací na jednom projektu zároveň ve více lidech. Využili jsme větví, které tento nástroj nabízí a každý jsme pracovali ve vlastní větvi. Tyto větve jsme po částech sloučili do hlavní větve main.

4.3 Rozdělení práce

Práce byla mezi členy původně rozdělena podle tabulky 1. Aby někteří členové mohli pracovat na svých částech, potřebovali vědět několik detailů z jiných částí projektu. Po domluvě byla práce přerozdělena, jak ukazuje tabulka 2.

Člen týmu	Práce, která mu byla přidělena
Michal Blažek	Syntaktická analýza pro výrazy, sémantická analýza, vedení týmu, dohlížení na pravidelné provádění práce, generátor cílového kódu, dokumentace
Kryštof Michálek	Syntaktická analýza bez výrazů, sémantická analýza, generátor cílového kódu, dokumentace
Matyáš Sapík	Lexikální analýza, vytvoření testů, dokumentace
Ondřej Závodský	Tabulka symbolů, vytvoření testů, dokumentace

Tabulka 1: původní rozdělení práce mezi členy týmu

Člen týmu	Práce, která mu byla přidělena
Michal Blažek	Lexikální analýza, syntaktická analýza, syntaktická analýza pro výrazy, vedení týmu, dohlížení na pravidelné provádění práce, tabulka symbolů, dokumentace
Kryštof Michálek	Syntaktická analýza, generátor cílového kódu, dokumentace
Matyáš Sapík	Vytvoření testů, dokumentace
Ondřej Závodský	

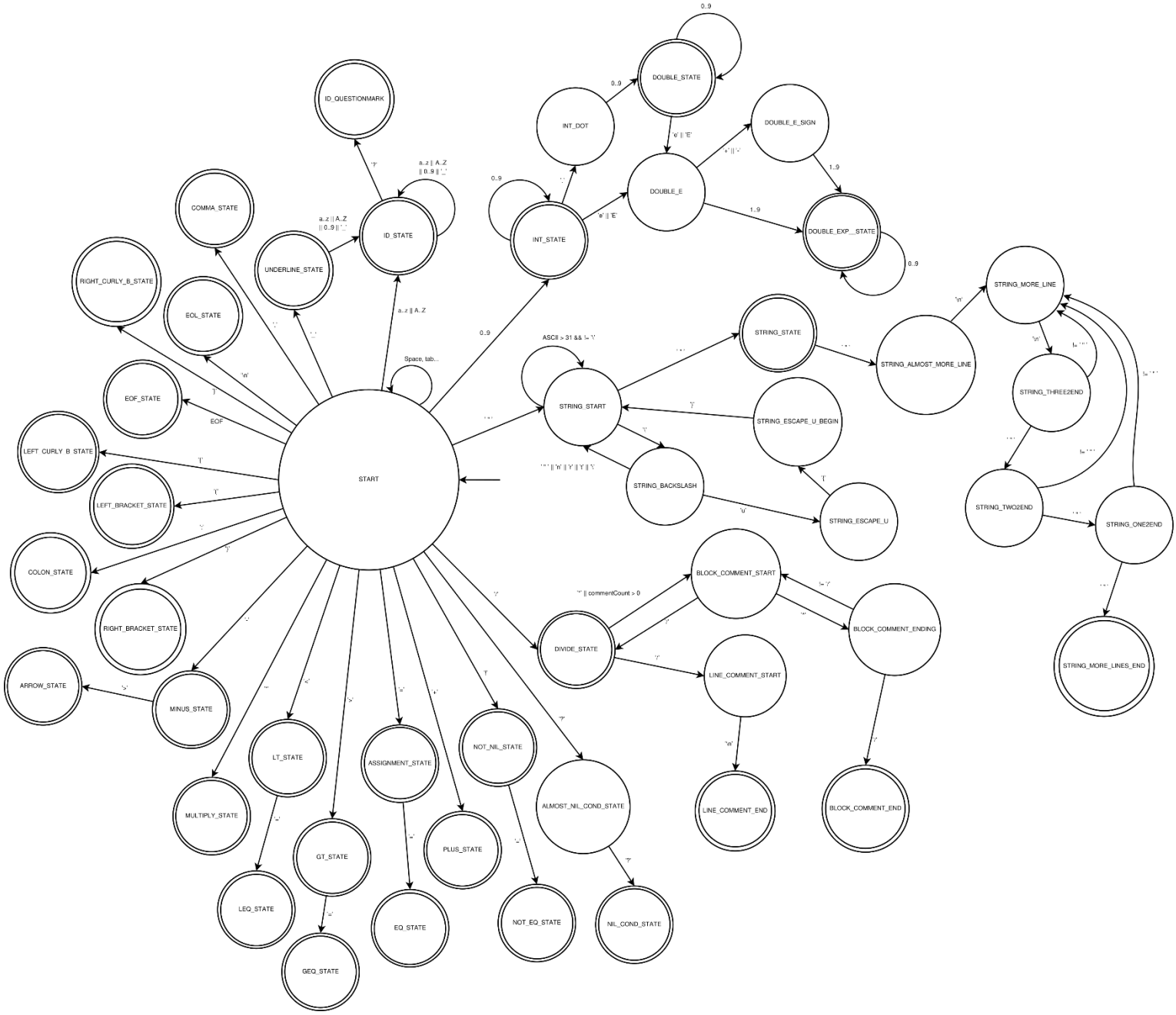
Tabulka 2: nové rozdělení práce mezi členy týmu

5 Závěr

Ze začátku jsme si moc nebyli jistí, jak máme vůbec začít s projektem. Po nějaké době, co se daná látka probírala na přednáškách, jsme ale pochopili, jakým způsobem budeme implementovat jednotlivé části projektu. Jelikož jsme nezačali úplně nejdřív, měli jsme s projektem lehké problémy, abychom vše stihli udělat včas. Komunikace v našem týmu nebyla nejlepší, takže jsme si alespoň odnesli cenné zkušenosti, jak obtížná může být práce v týmu.

Díky pokusným odevzdání jsme mohli ověřit funkčnost našeho projektu a odladit pár chyb. Tento projekt nám dokonale vysvětlil látku probíranou na přednáškách předmětů IFJ a IAL.

6 Diagram konečného automatu pro lexikální analýzu



Obrázek 1: digram konečného automatu pro lexikální analýzu

7 LL – gramatika specifikující syntaktickou analýzu

1. <PROG> -> <BLOCK> <PROG>
2. <PROG> -> EOF
3. <BLOCK> -> EOL
4. <BLOCK> -> func IDENTIFIER (<PARAM>) <RETURN> <EOL> { <ST_LIST> }
5. <BLOCK> -> let IDENTIFIER <WITH_TYPE> <ASSIGN_MARK>
6. <BLOCK> -> var IDENTIFIER <WITH_TYPE> <ASSIGN_MARK>
7. <BLOCK> -> IDENTIFIER <VOID_OR_ASSIGN>
8. <BLOCK> -> while <EXPRESSION> <EOL> { <ST_LIST> }
9. <BLOCK> -> if <EXPRESSION> <EOL> { <ST_LIST> } <EOL> <ELSE>
10. <VOID_OR_ASSIGN> -> (<CALLING_PARAM>)
11. <VOID_OR_ASSIGN> -> = <EOL> <ASSIGNMENT>
12. <ELSE> -> else <EOL> { <ST_LIST> }
13. <EOL> -> EOL
14. <EOL> -> epsilon
15. <PARAM> -> IDENTIFIER IDENTIFIER_OR_UNDERLINE : <DATA_TYPE> <NEXT_PARAM>
16. <PARAM> -> _ IDENTIFIER : <DATA_TYPE> <NEXT_PARAM>
17. <PARAM> -> epsilon
18. <NEXT_PARAM> -> , <PARAM>
19. <NEXT_PARAM> -> epsilon
20. <RETURN> -> -> <DATA_TYPE>
21. <RETURN> -> epsilon
22. <ST_LIST> -> <STATEMENT> <ST_LIST>
23. <ST_LIST> -> epsilon
24. <STATEMENT> -> return <VALUE>
25. <STATEMENT> -> if <EXPRESSION> <EOL> { <ST_LIST> } <EOL> <ELSE>
26. <STATEMENT> -> let IDENTIFIER <WITH_TYPE> <ASSIGN_MARK>
27. <STATEMENT> -> var IDENTIFIER <WITH_TYPE> <ASSIGN_MARK>
28. <STATEMENT> -> IDENTIFIER <VOID_OR_ASSIGN>
29. <STATEMENT> -> while <EXPRESSION> <EOL> { <ST_LIST> }
30. <STATEMENT> -> EOL
31. <DATA_TYPE> -> INT
32. <DATA_TYPE> -> DOUBLE
33. <DATA_TYPE> -> STRING
34. <ASSIGN_MARK> -> = <EOL> <ASSIGNMENT>
35. <ASSIGN_MARK> -> epsilon
36. <ASSIGNMENT> -> IDENTIFIER (<CALLING_PARAM>)
37. <ASSIGNMENT> -> <VALUE>
38. <CALLING_PARAM> -> <VALUE>
39. <CALLING_PARAM> -> IDENTIFIER <WITH_VALUE> <NEXT_CALLING>
40. <CALLING_PARAM> -> epsilon
41. <WITH_VALUE> -> : <VALUE>
42. <WITH_VALUE> -> epsilon
43. <NEXT_CALLING> -> , <CALLING_PARAM>
44. <NEXT_CALLING> -> epsilon
45. <VALUE> -> INT_VALUE
46. <VALUE> -> DOUBLE_VALUE
47. <VALUE> -> STRING_VALUE
48. <WITH_TYPE> -> : <DATA_TYPE>
49. <WITH_TYPE> -> epsilon

8 Použitá LL – tabulka

	EOF	EOL	func	IDENTIFIER	_	(	)	return	let	var	while	if	else	:	,	->	return	INT	DOUBLE	STRING	=	epsilon	INT_VALUE	DOUBLE_VALUE	STRING_VALUE
<PROG>	2	1	1	1					1	1	1	1													
<BLOCK>		3	4	7					5	6	8	9													
<VOID_OR_ASSIGN>						10															11				
<ELSE>													12												
<EOL>		13																				14			
<PARAM>				15	16																	17			
<NEXT_PARAM>															18							19			
<RETURN>																20						21			
<ST_LIST>		22		22				22	22	22	22	22										23			
<STATEMENT>		30		28				24	26	27	29	25													
<DATA_TYPE>																		31	32	33					
<ASSIGN_MARK>																					34	35			
<ASSIGNMENT>				36																			37	37	37
<CALLING_PARAM>				39																		40	38	38	38
<WITH_VALUE>														41								42			
<NEXT_CALLING>															43							44			
<VALUE>																							45	46	47
<WITH_TYPE>														48								49			

Tabulka 3: LL – tabulka pro syntaktickou analýzu

9 Precedenční tabulka pro zpracování výrazů

	!	??	==, !=, <, >, <=, >=	+,-	*,/	(	)	E	$
!		>	>	>	>		>		>
??	<	<	<	<	<	<	>	<	>
==, !=, <, >, <=, >=	<	>		<	<	<	>	<	>
+,-	<	>	>	>	<	<	>	<	>
*,/	<	>	>	>	>	<	>	<	>
(	<	>	<	<	<	<	=	<	
)	>	>	>	>	>		>		>
E	>	>	>	>	>		>		>
$	<	<	<	<	<	<		<	

Tabulka 4: Precedenční tabulka pro výrazy