

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace k projektu do IFJ/IAL

Implementace překladače imperativního jazyka IFJ17

Tým 022, varianta II

Členové týmu:

Martin Omacht	– xomach00 (vedoucí)	34 %
Zdeněk Chovanec	– xchova19	33 %
Petr Hendrych	– xhendr03	33 %

Rozšíření: BASE, UNARY, FUNEXP, BOOLOP, IFTHEN, CYCLES

6. prosince 2017

1 Úvod

Cílem projektu bylo vytvoření překladače pro imperativní jazyk IFJ17. Jazyk IFJ17 je podmnožinou jazyka FreeBASIC, který je založen na jazyce BASIC. Při registracích jsme si vybrali variantu II, která spočívala v řešení projektu za pomoci tabulky s rozptýlenými položkami. Následuje výčet etap, tak jak šly za sebou.

2 Etapy projektu

Projekt jsme zahájili vytvořením prostředí pro vývoj. To zahrnovalo prvotní nastavení využitých nástrojů. Jelikož se preference operačních systémů mezi členy týmu liší, jako sestavovací program jsme zvolili nástroj **CMake**. Pro účely testování byl využit framework **Googletest**.

2.1 Lexikální analyzátor (LA)

Návrh LA byl prvním krokem k vytvoření překladače. Při návrhu deterministického konečného automatu, na kterém je LA založen jsme se snažili striktně dodržovat specifikaci jazyka IFJ17. Samotná implementace koresponduje s tímto schematickým návrhem (viz obrázek 1, strana 5) téměř jednu ku jedné. Lexikální analyzátor je samostatná jednotka, jejíž činnost je plně podřízena syntaktickému analyzátoru. V počátečních fázích vývoje byla komunikace mezi syntaktickým a lexikálním analyzátozem jednosměrná – SA v případě potřeby zavolal LA, který ze vstupu načetl další token. V průběhu vývoje jsme ale dospěli k závěru, že bude výhodnější umožnit komunikaci oběma směry. Komunikace ve směru SA → LA se projevuje tak, že SA má možnost vrátit LA právě načtený token. V případě, že SA v tuto chvíli zažádá o další token, LA nebude načítat „nový“ token ze vstupu, nýbrž mu poskytne „starý“ token, který mu byl vrácen samotným SA.

2.2 Syntaktický analyzátor (SA)

2.2.1 Analýza konstrukcí a příkazů (bez výrazů)

Pro vyhodnocení této části programů jsme se rozhodli použít metodu prediktivní syntaktické analýzy řízené LL tabulkou. Důvod, proč jsme nezvolili doporučenou metodu je ten, že prediktivní analýza se nám jevila jako efektivnější a sofistikovanější řešení. Největší výzvou této etapy byl návrh LL gramatiky. Následné sestavení LL tabulky, pomocí postupů, které nám byly prezentovány na přednáškách, již nepředstavovalo větší problém. Jak LL gramatika, tak i LL tabulka jsou součástí překladače a jsou inicializovány při spuštění programu. LL gramatika je implementována jako pole pravidel, kde prvku na indexu n odpovídá pravidlo číslo n v návrhu gramatiky (strana 6). Jelikož je velká část LL tabulky prázdná, rozhodli jsme se ji implementovat jako řádkou tabulku.

2.2.2 Analýza výrazů

Analýza výrazů je prováděna za pomoci precedenční tabulky a přidružených gramatických pravidel. Již od počátku jsme zamýšleli implementovat rozšíření FUNEXP a UNARY, která se podepsala na výsledné podobě precedenční tabulky.

2.3 Sémantická analýza

Tato část pro náš tým představovala největší výzvu. Problém jsme vyřešili následujícím způsobem. Vybraným pravidlům LL gramatiky jsme přiřadili příznak sémantické akce (zkratka: sém. akce). Každá sém. akce reprezentuje stavový automat, který při zpracování jednoho vstupu může přejít do nového stavu, nebo zůstat ve stavu, ve kterém se aktuálně nachází. V případě, že v průběhu syntaktické analýzy

je na vrcholu syntaktického zásobníku detekován neterminál s příznakem sém. akce, je příslušná akce vložena na zásobník sémantických akcí. Před načtením nové lexémy je zavolána sém. akce z vrcholu zásobníku a je jí předán aktuální token. Tato akce má za úkol daný token zpracovat (například zkontrolovat typ, existenci proměnné, vygenerovat instrukce). Jestliže akce v průběhu zpracovávání svého vstupu dosáhne koncového stavu je odebrána ze zásobníku. Jelikož na zásobníku může být sém. akcí hned několik, avšak aktivní je v daném okamžiku jen ta na vrcholu, je nutné, aby si sém. akce mezi sebou předávaly informace. K předávání informací dochází právě v okamžiku odstraňování akce ze zásobníku, kdy je právě dokončená sém. akce odebrána a její výsledek je předán akci, která se nově nachází na vrcholu zásobníku.

2.3.1 Implementace TS

Tabulka symbolů je reprezentována tabulkou s rozptýlenými položkami s explicitně zřetězenými synonymy. Synonyma jsou svázána do jednosměrného seznamu a nejsou seřazena. V případě vkládání položky je daná položka vložena na konec seznamu. Jako mapovací funkci jsme použili funkci DJB.

2.4 Generování kódu

Původní plán byl generovat abstraktní syntaktický strom a následně z něj vygenerovat kód, avšak neshledali jsme v tomto řešení žádnou výhodu, ale spíše jen časové zdržení. Proto jsme se rozhodli generovat instrukce tří adresného kódu přímo. Instrukce jsou generovány do tří různých seznamů.

1. **globální seznam** – definice globálních proměnných a jejich inicializace
2. **seznam pro funkce** – definice funkcí
3. **instrukce hlavního těla programu**

Na konci analýzy celého vstupního programu se nejdříve vypíše instrukce z globálního seznamu, poté hlavní tělo programu a nakonec funkce.

3 Rozdělení práce

Martin Omacht

- implementace syntaktické analýzy (bez výrazů)
- implementace sémantické analýzy
- generování kódu
- testování

Zdeněk Chovanec

- návrh konečného automatu lexikálního analyzátoru
- návrh gramatik pro syntaktický analyzátor
- implementace syntaktické analýzy pro výrazy
- implementace tabulky symbolů

Petr Hendrych

- implementace lexikálního analyzátoru

- vestavěné funkce v IFJcode17
- dokumentace
- prezentace

4 Rozšíření

4.1 GLOBAL

Toto rozšíření přidává možnost definování globálních a statických proměnných. Implementace globálních proměnných nepředstavovala žádný problém – jejich definice je přidána na globální instrukční list a prostor pro ně je alokován na globálním rámci. Jelikož statické proměnné si musí uchovávat hodnotu mezi jednotlivými volání funkce, bylo to s nimi složitější. Vyřešili jsme to tak, že jsme tyto proměnné definovali jako globální. Zároveň byl k jejich názvu přidán prefix „*Sname*“, kde *name* je název funkce, ve kterých jsou definovány. Tímto nedojde ke kolizím s ostatními proměnnými.

4.2 UNARY

Implementace operace přiřazení s aritmetickou operací se jednoduše vyřešilo provedením aritmetické operace a následným přiřazením do proměnné. U unárního mínus bylo potřeba přidat jeden sloupec/řádek reprezentující unární mínus do precedenční tabulky. Problémem však je, že jeden symbol může značit 2 různé operace. Tento problém jsme vyřešili tak, že v případě, že je načten token značící operaci odčítání, je na základě předchozího načteného tokenu rozhodnuto zda-li se jedná o minus binární, či unární.

4.3 BASE

Rozšíření ovlivnilo pouze lexikální analyzátor, kde po načtení čísla v jiné soustavě se převedlo do dekadické soustavy a dále se pracovalo pouze s dekadickým číslem.

4.4 CYCLES

Zde byl nutný zásah do LL gramatiky (resp. LL tabulky). S tímto rozšířením jsme počítali již od začátku, tudíž to nepředstavovalo žádný problém. Stejně tak doplnění plné funkcionality cyklu typu **Do...Loop**. Největší záležitost se skrývala v cyklu **For...Next**, a to v definici iterační proměnné. Řešení spočívá v tom, že každý cyklus **for** je označen unikátním identifikátorem. V případě, že dojde k definici nové iterační proměnné, je tato proměnná definována na globálním rámci a je její jméno tvoří řetězec, který vznikne zkonkatenováním původní názvu iterační proměnné a identifikátoru cyklu **for**. Po skončení cyklu již k této proměnné nelze přistoupit. Pro implementaci příkazů **exit**, **continue** jsme použili dříve zmíněný zásobník sémantických akcí.

4.5 FUNEXP

Toto rozšíření se implementovalo jednoduše pomocí vyhodnocování výrazů na datovém zásobníku. Parametry funkcí totiž předáváme datovým zásobníkem, stejně tak i návratovou hodnotu funkce. Menší problém představovala implicitní konverze datového typu předávaných parametrů při volání funkce. Parametry již byly vloženy na datový zásobník a nebyl k nim přístup pro konverzi. Toto bylo vyřešeno v definici funkce, kde pokud byl parametr typu INT nebo DOUBLE bylo k nim vygenerováno dynamické přetypování. To spočívalo, v zjištění datového typu pomocí instrukce TYPE a následně přeskočení konverze, pokud datový typ odpovídal definici parametru.

4.6 IFTHEN

Podpora IF bez větve ELSE nebyla složitá. U větvi ELSEIF bylo potřeba zajistit podporu pro jejich potenciálně nekonečné množství. Toho bylo docíleno tím, že se pro každou tuto větev vygeneruje unikátní identifikátor.

4.7 BOOLOP

V zásadě se jednalo pouze o rozšíření precedenční tabulky o logické operátory.

5 Komunikace v týmu

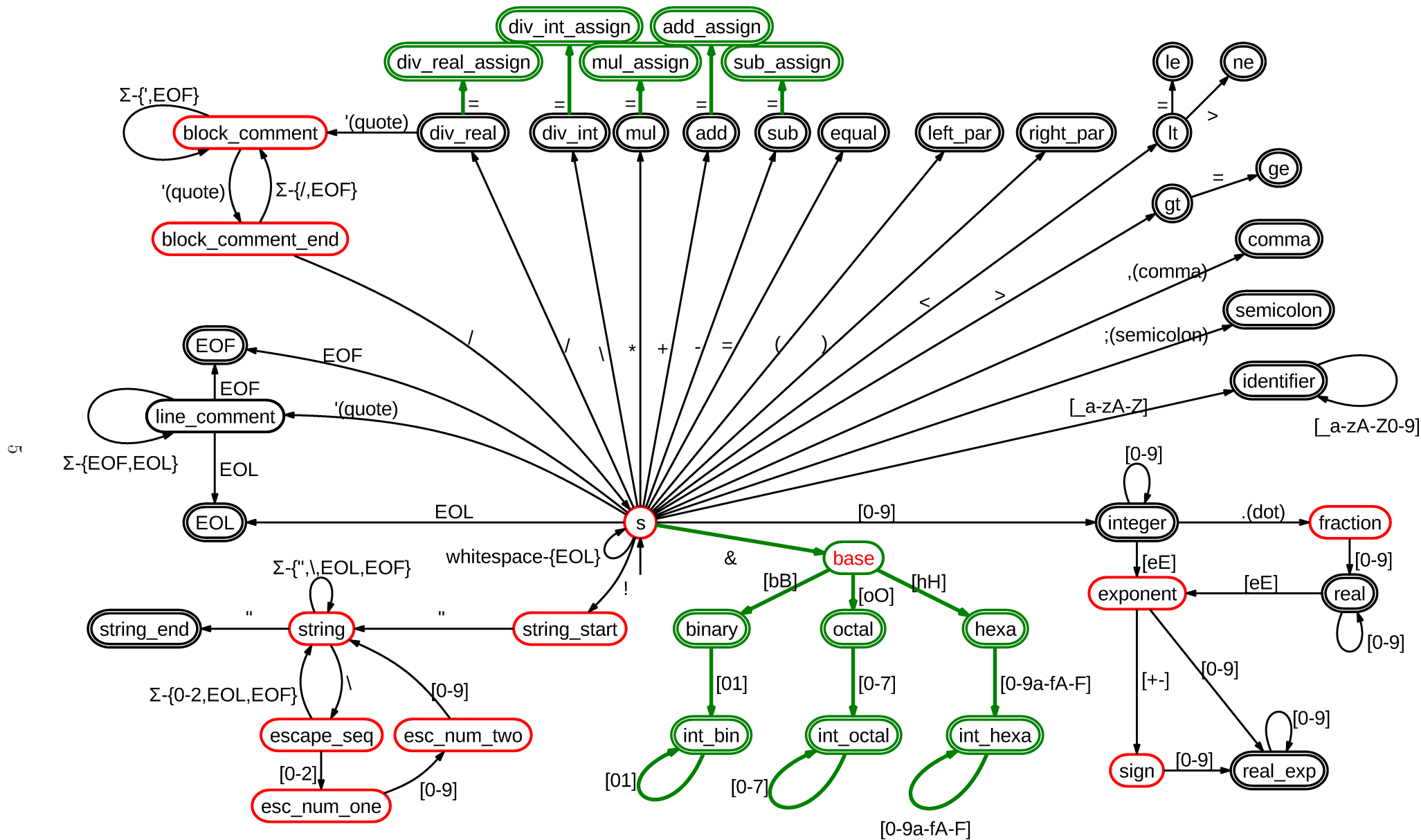
Pro online komunikaci jsme si vybrali Messenger, kde jsme si založili skupinový chat. Pokud se však objevil větší problém, tak jsme se sešli všichni osobně. Pro sdílení souborů projektu a verzování jsme zvolili verzovací systém Git hostovaný na službě Github. Github nám umožňoval například code review nově nahraných verzí nebo přidávání issues pokud něco nefungovalo.

6 Projekt v číslech

- počet řádků: 8 883
- počet commitů: 342

7 Shrnutí

Projekt byl něco zcela nového pro všechny z nás a zabral opravdu velkou část semestru. Museli jsme se naučit mnohému jako například: práce v týmu a komunikace v něm, práce s verzovacím systémem Git, nemluvě o nabytých znalostech z oblasti teorie formálních jazyků. Využili jsme pokusného odevzdání a myslíme, že splnilo svůj účel. V těch pár dnech, které mu předcházely, nás totiž přinutilo pracovat usilovněji, než bychom pracovali za normálních okolností. Samotně vyhodnocení pokusného odevzdání a relativně dobrý výsledek už byl jen třešničkou na dortu. Pro nás to znamenalo především to, že jsme mohli začít doladovat menší nedostatky a implementovat zbývající rozšíření, na která jsme ještě měli záhlusk.



Obrázek 1: Konečný automat specifikující lexikální analyzátor

LL Gramatika IFJ17

- (1) $Line \rightarrow GlobalStmt \ ScopeStmt \ LineEnd$
- (2) $LineEnd \rightarrow \mathbf{EOL} \ LineEnd$
- (3) $\quad \quad \quad | \ \varepsilon$
- (4) $GlobalStmt \rightarrow FuncDecl \ \mathbf{EOL} \ GlobalStmt$
- (5) $\quad \quad \quad | \ FuncDef \ \mathbf{EOL} \ GlobalStmt$
- (6) $\quad \quad \quad | \ SharedVar \ \mathbf{EOL} \ GlobalStmt$
- (7) $\quad \quad \quad | \ \mathbf{EOL} \ GlobalStmt$
- (8) $\quad \quad \quad | \ \varepsilon$
- (9) $InnerStmt \rightarrow VarDecl$
- (10) $\quad \quad \quad | \ Assignment$
- (11) $\quad \quad \quad | \ IfStmt$
- (12) $\quad \quad \quad | \ ScopeStmt$
- (13) $\quad \quad \quad | \ DoStmt$
- (14) $\quad \quad \quad | \ ForStmt$
- (15) $\quad \quad \quad | \ PrintStmt$
- (16) $\quad \quad \quad | \ InputStmt$
- (17) $\quad \quad \quad | \ ReturnStmt$
- (18) $\quad \quad \quad | \ ExitStmt$
- (19) $\quad \quad \quad | \ ContinueStmt$
- (20) $\quad \quad \quad | \ \varepsilon$
- (21) $StmtSeq \rightarrow InnerStmt \ \mathbf{EOL} \ StmtSeq$
- (22) $\quad \quad \quad | \ \varepsilon$
- (23) $VarDecl \rightarrow \mathbf{DIM} \ VarDef$
- (24) $\quad \quad \quad | \ \mathbf{STATIC} \ VarDef$
- (25) $SharedVar \rightarrow \mathbf{DIM} \ \mathbf{SHARED} \ VarDef$
- (26) $VarDef \rightarrow \mathbf{ID} \ \mathbf{AS} \ Type \ InitOpt$
- (27) $InitOpt \rightarrow '=' \ Expression$
- (28) $\quad \quad \quad | \ \varepsilon$
- (29) $FuncDecl \rightarrow \mathbf{DECLARE} \ \mathbf{FUNCTION} \ \mathbf{ID} \ '(' \ Params \ ')' \ \mathbf{AS} \ Type$
- (30) $Type \rightarrow \mathbf{INTEGER}$

(31)			DOUBLE
(32)			STRING
(33)			BOOLEAN
(34)	<i>FuncDef</i>	→	FUNCTION ID <i>'(' Params ')</i> AS Type EOL StmtSeq END FUNCTION
(35)	<i>ParamDecl</i>	→	ID AS Type
(36)	<i>Params</i>	→	<i>ParamDecl ParamsNext</i>
(37)			ϵ
(38)	<i>ParamsNext</i>	→	' , ' ParamDecl ParamsNext
(39)			ϵ
(40)	<i>ReturnStmt</i>	→	RETURN Expression
(41)	<i>Assignment</i>	→	ID AssignOperator Expression
(42)	<i>InputStmt</i>	→	INPUT ID
(43)	<i>PrintStmt</i>	→	PRINT Expression ;' ExpressionList
(44)	<i>ExpressionList</i>	→	<i>Expression ;' ExpressionList</i>
(45)			ϵ
(46)	<i>ScopeStmt</i>	→	SCOPE EOL StmtSeq END SCOPE
(47)	<i>IfStmt</i>	→	IF Expression THEN EOL StmtSeq <i>IfStmtElseif</i> <i>IfStmtElse</i> END IF
(48)	<i>IfStmtElseif</i>	→	ELSEIF Expression THEN EOL StmtSeq <i>IfStmtElseif</i>
(49)			ϵ
(50)	<i>IfStmtElse</i>	→	ELSE EOL StmtSeq
(51)			ϵ
(52)	<i>DoStmt</i>	→	DO DoStmtEnd
(53)	<i>DoStmtEnd</i>	→	<i>TestTypeStart Expression EOL StmtSeq LOOP</i>
(54)			EOL StmtSeq LOOP <i>TestTypeEnd</i>
(55)	<i>TestTypeStart</i>	→	WHILE
(56)			UNTIL
(57)	<i>TestTypeEnd</i>	→	WHILE Expression
(58)			UNTIL Expression
(59)			ϵ
(60)	<i>ExitStmt</i>	→	EXIT LoopType LoopTypeEnd
(61)	<i>ContinueStmt</i>	→	CONTINUE LoopType LoopTypeEnd
(62)	<i>LoopType</i>	→	DO

(63)			FOR
(64)	<i>LoopTypeEnd</i>	→	, <i>LoopType</i> <i>LoopTypeEnd</i>
(65)			ε
(66)	<i>ForStmt</i>	→	FOR ID <i>TypeOpt</i> '=' <i>Expression</i> TO <i>Expression</i> <i>StepOpt</i> EOL <i>StmtSeq</i> NEXT <i>IdOpt</i>
(67)	<i>TypeOpt</i>	→	AS <i>Type</i>
(68)			ε
(69)	<i>StepOpt</i>	→	STEP <i>Expression</i>
(70)			ε
(71)	<i>IdOpt</i>	→	ID
(72)			ε
(73)	<i>AssignOperator</i>	→	'='
(74)			'- ='
(75)			'+' ='
(76)			'* ='
(77)			'\ ='
(78)			'/' ='
(79)			

Značení

- Neterminály: Vyznačeny kurzívou (*GlobalStmt*, *ScopeStmt*, ...).
- Terminály(tokeny): Zapsány velkými písmeny a vyznačeny tučně (**IF**, **LOOP**, ...), případně ohraňeny v apostrofech ('=', '(', ')', ...).
- Pravidla spjatá s rozšířeními vyznačena zeleně.

[illegible]

Obrázek 2: LL Tabulka

		TOKEN (TERMINÁL) na vstupu																				
		+	-	u -	*	\	/	=	<>	>	>=	<	<=	CONST	ID	NOT	AND	OR	()	,	\$
Nejsvrchnější TOKEN (TERMINÁL) na zásobníku	+	>	>	<	<	<	<	>	>	>	>	>	>	<	<	<	>	>	<	>	>	>
	-	>	>	<	<	<	<	>	>	>	>	>	>	<	<	<	>	>	<	>	>	>
	u -	>	>	<	>	>	>	>	>	>	>	>	>	<	<	>	>	>	<	>	>	>
	*	>	>	<	>	>	>	>	>	>	>	>	>	<	<	<	>	>	<	>	>	>
	\	>	>	<	>	>	>	>	>	>	>	>	>	<	<	<	>	>	<	>	>	>
	/	>	>	<	>	>	>	>	>	>	>	>	>	<	<	<	>	>	<	>	>	>
	=	<	<	<	<	<	<							<	<	<	>	>	<	>	>	>
	<>	<	<	<	<	<	<							<	<	<	>	>	<	>	>	>
	>	<	<	<	<	<	<							<	<	<	>	>	<	>	>	>
	>=	<	<	<	<	<	<							<	<	<	>	>	<	>	>	>
	<	<	<	<	<	<	<							<	<	<	>	>	<	>	>	>
	<=	<	<	<	<	<	<							<	<	<	>	>	<	>	>	>
	CONST	>	>		>	>	>	>	>	>	>	>	>				>	>		>	>	>
	ID	>	>		>	>	>	>	>	>	>	>	>				>	>	=	>	>	>
	NOT	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>	>	>
	AND	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>	>	>
	OR	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	<	>	>	>
	(<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>	>	>			>	>	>		>	>	>
	,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	
	\$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<			\$

Obrázek 3: Precedenční tabulka