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 systemů mezi členy týmu liší, jako sestavovací program jsme zvolili nástroj **CMake**. Pro účely testování jsme využili 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 analyzátorem jednosměrná – SA v případě potřeby zavolal proceduru LA, která ze vstupu načetla další token. V průběhu vývoje jsme ale dospěli k závěru, že bude výhodnější umožnit komunikaci oběma směry. Komunikace ve směru SA $\rightarrow$ LA se projevuje tak, že SA má možnost vrátit LA právě načtený token. V prípadě, 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 (mimo výrazy)

Pro vyhodnocení této části programů jsme se rozhodli použit 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, aplikováním 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 odpovidá pravidlo číslo n v návrhu gramatiky (strana 6). Jelikož je velká část LL tabulky prázdná, rozhodli jsme se ji implementovat jako řidkou 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.

3 Sémantická analýza

Tato část pro náš tým představovala největší výzvu. Problém jsme vyřešili následujícím způsobem. Vybraným neterminálům 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 nového tokenu 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.

4 Generování kódu

Původní plán byl generovat abstraktního syntaktického stromu 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 vypíší instrukce z globálního seznamu, poté hlavní tělo programu a nakonec funkce.

5 Rozdělení práce

Martin Omacht

- vůdce
- hlavní prográmator
- implementace syntaktické a sémantické analýzy, generace kódu
- testování

Zdeněk Chovanec

- hlavní návrhář
- návrh konečného automatu lexikálního analyzátoru
- návrh gramatik pro syntaktický analyzátor
- pomocný programátor
- implementace syntaktické analýzy pro výrazy
- implementace sémantických kontrol a generace kódu
- implementace tabulky symbolů
- testování

Petr Hendrych

- implementace lexikálního analyzátoru
- vestavěné funkce v IFJcode17
- dokumentace
- prezentace

6 Rozšíření

6.1 GLOBAL

Toto rozšíření přidává možnost definování globálních a statických proměnných. Implementace globální 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 složítější. Vyřešili jsme to tak, že jsme tyto proměnné definovali jako globální. Zároveň byl k jejich názvu přidán prefix „S_*name*“, kde *name* je název funkce, ve kterých jsou definovány, tak aby nedošlo ke kolizím s ostatními proměnnými.

6.2 UNARY

Implementace operace přiřazení s aritmetickou operací se jednoduše vyřešilo provedením aritmetické operace a následným přirazení 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, čí unární.

6.3 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.

6.4 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 záludnost se skrývala v cyklu **For...Next**, a to v definici iterační proměnné. Řešení spočíva v tom, že každý cyklus **for** je označen unikátním identifikátorem. V prípadě, že dojde k definici nové iterační proměnné, je tato proměnná definována no 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.

6.5 FUNEXP

Toto rozšíření se implementovalo jednoduše pomoci vyhodnocování výrazů na datovém zásobníku. Parametry funkcí totiž předáváme datovým zásobníkem, stejně tak inávratovou 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.

6.6 IFTHEN

Podpora IF bez větve ELSE nebyla složitá. Akorát u podpory větví ELSEIF bylo složitější generování návěstí. Což bylo vyřešeno dvěma identifikátory. Jeden pro celý IF a druhý pro aktuální větev ELSEIF. Z těchto identifikátorů se následně generovaly jednotlivé návěstí s prefixy ELSE nebo ENDIF. Přičemž ELSE může být další ELSEIF.

6.7 BOOLOP

ZDENDA NEVI CO SEM NAPSAT :D

7 Komunikace v týmu

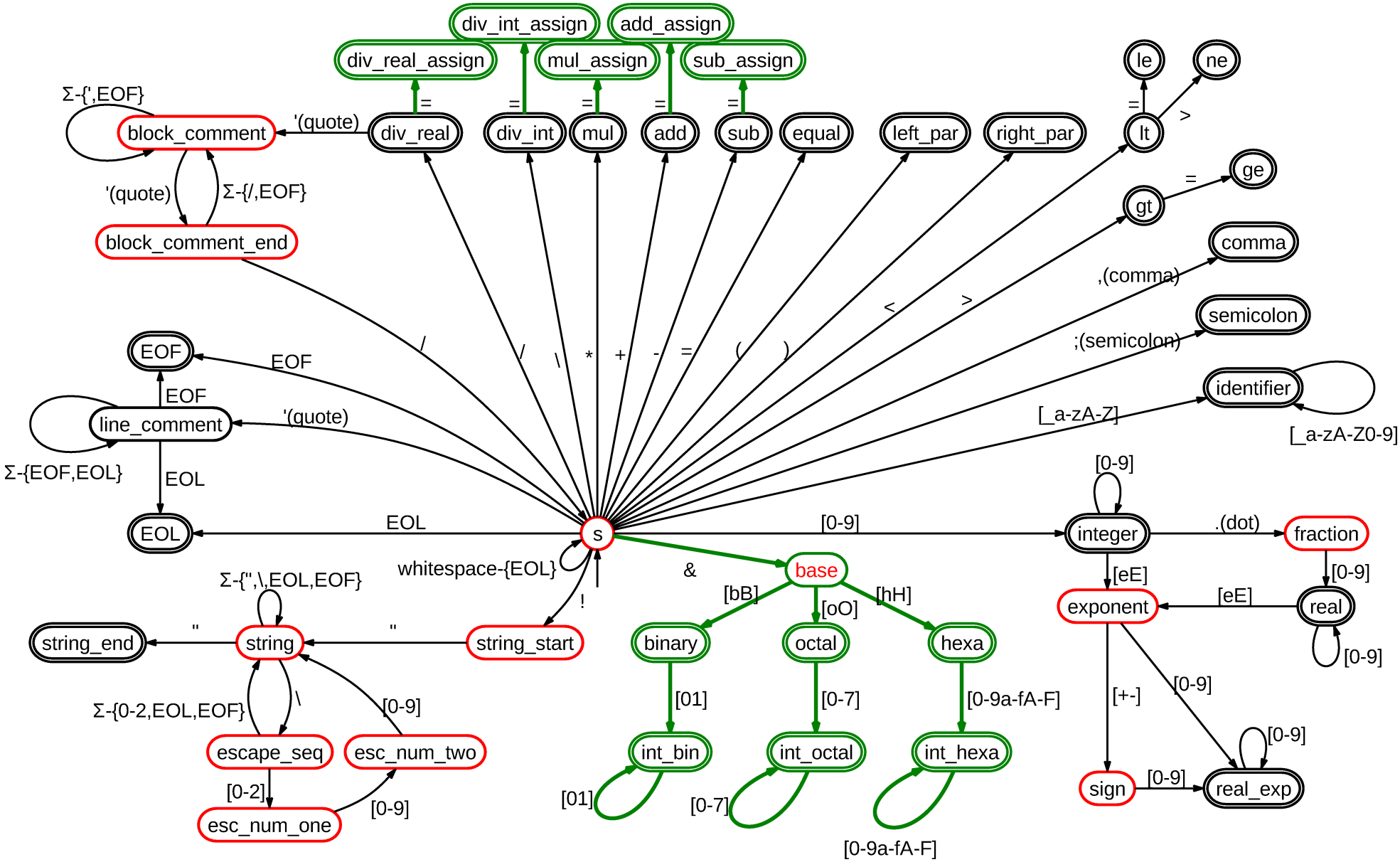
Jelikož dva ze tří členů bydlí na kolejích a ve stejném pokoji, tak komunikace spočívala převážně s posledním členem týmu. Jako online komunikaci jsme si vybrali Messegner, kde jsme si založili skupinový chat pro snadnou komunikaci. Pokud se však objevil větší problém, tak jsme se sešli všichni osobně Pro sdílení souborů projektu a verzování jsme zvolili verzovací systém Git hostovaný na službě Github. Github nám také umožňoval code review nově nahraných verzí nebo například přidávání issues pokud něco nefungovalo.

8 Projekt v číslech

- počet řádků:
- počet commitů:
- počet souborů:

9 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 třešníčkou na dortu. Pro nás to znamenalo především to, že jsme mohli začít dolaďovat menší nedostatky a implementovat zbývající rozšíření, na která jsme ještě měli zálusk.



Obrázek 1: Konečný automat specifikující lexikální analyzátor

LL Gramatika IFJ17

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

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

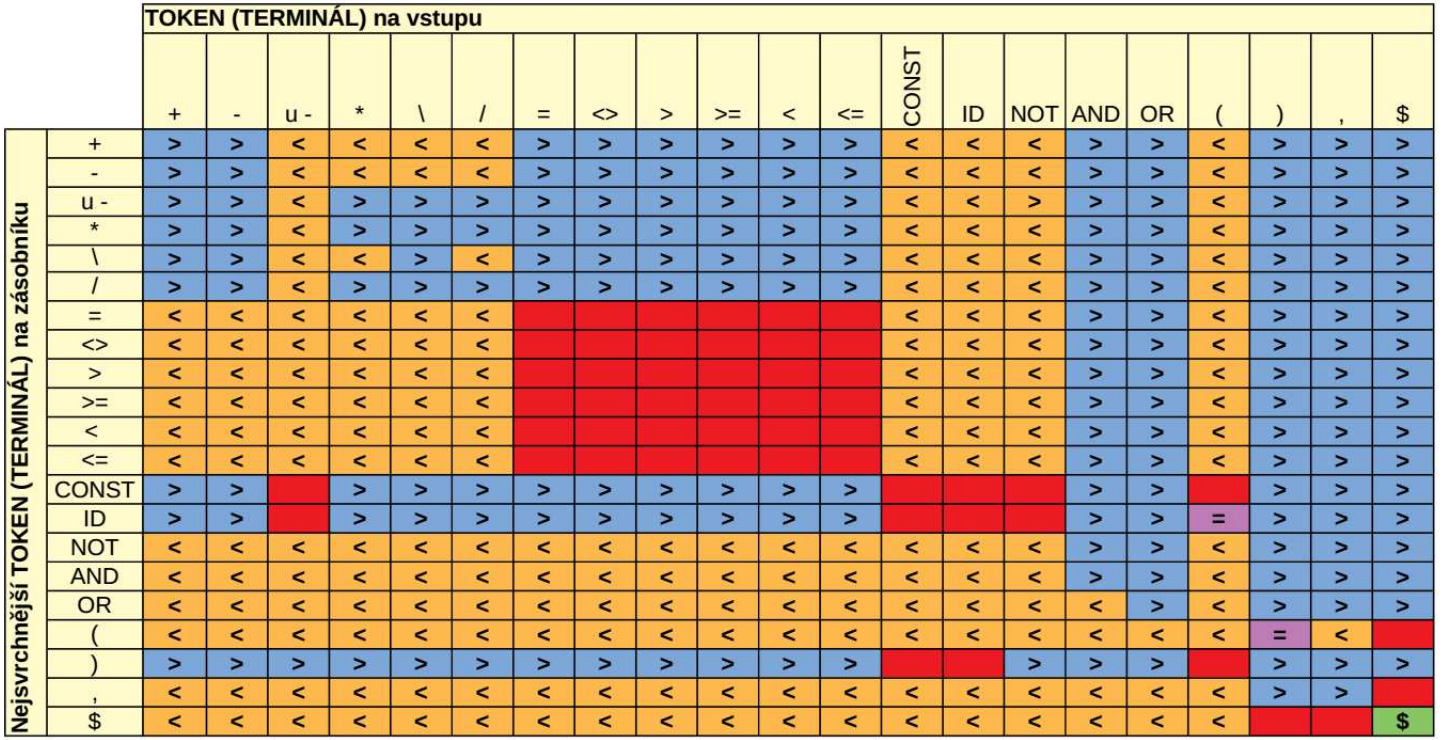
(63) | **FOR**
(64) *LoopTypeEnd* → **,** *LoopType* *LoopTypeEnd*
(65) | ε
(66) *ForStmt* → **FOR** **ID** *TypeOpt* ‘ = ‘ *Expression* **TO** *Expression* *StepOpt* **EOL** *StmtSeq* **NEXT** *IdOpt*
(67) *TypeOpt* → **AS** *Type*
(68) | ε
(69) *StepOpt* → **STEP** *Expression*
(70) | ε
(71) *IdOpt*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

Značení

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

	Klíčová slova																													Literály			Ostatní														
Pravidla	AS	DECLARE	DIM	DO	DOUBLE	ELSE	END	FUNCTION	IF	INPUT	INTEGER	LOOP	PRINT	RETURN	SCOPE	STRING	THEN	WHILE	UNTIL	BOOLEAN	CONTINUE	ELSEIF	EXIT	FALSE	FOR	NEXT	SHARED	STATIC	TRUE	TO	STEP	STR	INT	REAL	ID	COMMA	SEMICOL	LPAR	RPAR	EOL	SUB	ASSIGN	SUB =	ADD =	MUL =	DIVI =	DIVR =	EOF = $
Line		1	1					1							1																								1								
LineEnd																																							2								3
GlobalStmt		4	6					5							8																								7								
InnerStmt			9	13					11	16			15	17	12						19		18		14			9							10				20								
StmtSeq			21	21		22	22		21	21		22	21	21	21						21	22	21		21	22		21							21				21								
VarDecl			23																									24																			
SharedVar			25																																												
VarDef																																			26												
InitOpt																																							28		27						
FuncDecl		29																																												
Type					31						30					32				33																											
FuncDef								34																																						
ParamDecl																																			35												
Params																																			36			37								
ParamsNext																																				38		39								
ReturnStmt														40																																
Assignment																																			41												
InputStmt										42																																				
PrintStmt													43																																		
ExpressionList																								44					44			44	44	44	44				45	44							
ScopeStmt															46																																
IfStmt									47																																					
IfStmtElseif						49	49															48																									
IfStmtElse						50	51																																							
DoStmt				52																																										
DoStmtEnd																		53	53																					54								
TestTypeStart																		55	56																													
TestTypeEnd																		57	58																					59								
ExitStmt																							60																									
ContinueStmt																					61																											
LoopType				62																					63																						
LoopTypeEnd																																				64				65						
ForStmt																									66																						
TypeOpt	67																																									68						
StepOpt																														69									70								
IdOpt																																			71				72								
AssignOperator																																									73	74	75	76	77	78	

Obrázek 2: LL Tabulka



Obrázek 3: Precedenční tabulka