Vysoké učení technické v Brně
Fakulta informačních technologií

Projektová dokumentace

Implementace překladače jazyka IFJ21

Tým 038, varianta II

8. prosince 2021

David Kocman	**xkocma08**	**25 %**
Martin Ohnút	xohnut01	25 %
Přemek Janda	xjanda28	25 %
Radomír Bábek	xbabek02	25 %

1 Úvod

Projekt předmětu Formální jazyky a překladače, implementující překladač imperativního jazyka IFJ21, který je podmnožinou jazyka Teal. Překladač je psán v jazyce C a podílí se na něm čtyřčlenný tým. Mezikód je psán v cílovém jazyce IFJcode21.

2 Implementace

2.1 Lexikální analyzátor

Jako první součástku jsme realizovali lexikální analyzátor, neboli „scanner“. Zdrojový kód je v souboru `lexikon.c` a skládá se z hlavní funkce `scanner`. Hlavičkový soubor `lexikon.h` obsahuje prototypy funkcí. Vstupní argument Token, který obsahuje typ, atribut a řádek, na kterém se slovo nachází, je přepisovám při každé iteraci a následně ukládán do syntaktického stromu v `parseru` a při přečtení jednoho tokenu je funkce ukončena a vrací 0. Všechny možné typy tokenů jsou popsány ve zdrojovém souboru.

Každý token je porovnáván se statickým polem, ve kterém jsou uložena všechna klíčová slova, a pokud je shoda, do typu se přiřadí „keyword“. Ostatní typy jsou přiřazeny na konci každého načtení tokenu, podle toho co v sobě obsahují.

Scanner je realizován jako deterministický konečný automat, který je realizován níže 1, a implementován v podobě jedné funkce `switch` s deseti `case` výrazy (tokeny o velikosti 1 končí hned ve startu). Automat příjmá jedině znaky, které obsahuje jazyk IFJ21. V opačném případě hlásí chybu 1.

Escape sekvence jsou podporovány. Sekvence typu `\ddd` jsou ukládány do pole a následně, pokud vše proběhlo bez problému, je sekvence převedena do číselné hodntoty a porovnávána, zda-li je v intervalu od 1 do 255. Blokové a řádkové komentáře jsou scannerem ignorovány.

2.2 Syntaktický analyzátor

Pro jazyk IFJ21 jsme zvolili prediktivní syntaktický analyzátor. Mezi jeho nesporné výhody patří především fakt, že při jakémkoliv odhalení chyby v gramatice stačilo pouze upravit samotnou gramatiku a syntaktický analyzátor mohl zůstat nedotčen. Pro jeho implementaci bylo neprve potřeba vytvořit strukturu pro zásobník (`stack.c` a `stack.h`), tabulku pro načtení LL a precedenční tabulky a tabulky s pravidly do programu (`table.c` a `table.h`), a v neposlední řadě také strukturu pro strom (`tree.c` a `tree.h`), který je výsledným výstupem syntaktické analýzy. Analyzátor je umístěn v souboru parser.c a je rozdělen na dvě části: top-down a bottom-up (nebo také precedenční analýza).

Top-down analýza se nachází ve funkci `syntax_analyzer()`. Po jejím zavolání jsou načteny všechny potřebné tabulky, vytvořen první uzel stromu a vytvořen hlavní stack, do kterého je po inicializaci přidán neterminál <`prog`>. Mezi klíčové prvky patří také ukazatel na aktuální uzel (resp. uzel reprezentující položku a vrcholu zásobníku), který postupně zpracovává jednotlivé uzly a cestuje mezi nimi (vždy od nejlevějšího k nejpravějšímu). Následně je načten první token. Analyzátor vezme aktuální token a položku na vrcholu zásobníku, a podívá se do načtené LL tabulky, v jaké buňce se tyto dvě položky setkávají. V tento moment mohou nastat 2 různé situace. Buď je daná buňka zcela prázdná, což znamená neexistující pravidlo a tudíž i syntaktickou chybu, nebo je zde umístěno číslo značící pravidlo, které je potřeba použít. Číslo tohoto pravidla zároveň značí řádek, ve kterém se toto pravidlo nachází v tabulce s pravidly. Každá buňka v této tabulce značí jednotlivé terminály a neterminály, na které se neterminál na vrcholu zásobníku rozvíjí. Tyto jednotlivé položky jsou následně přidány do aktuálního uzlu jako potomci a také přidány do zásobníku. Poté nastává fáze, kdy analyzátor načítá nové tokeny a porovnává je s vrcholem zásobníku. V případě neterminálu je potřeba použít další pravidlo a opakuje se celý proces analýzy. Pokud se zde nachází terminál a odpovídá hodnotě tokenu, položka se ze zásobníku odebere a do aktuálního uzlu jsou následně uloženy veškéré data aktuálního

tokenu. Poté je načten další token a proces se opakuje. Případ, kdy hodnota ze zásobníku s tokenem nesedí značí syntaktickou chybu.

Další z velmi klíčových úkolů analýzy je schopnost rozhodnout, kdy je potřeba použít pravidlo pro výraz. Vzhledem k tomu, že ne vždy, co se může zpočátku zdát jako výraz, je ve skutečnosti výraz (např. token `id` může značit začátek výrazu nebo také volání funkce), bylo nutné vytvořit `backup token`. Ve všech případech, kdy nastává tato nejednoznačná situace, je načten nový backup token. Díky němu se analyzátor může podívat, co za aktuálním tokenem následuje a poté na základě nejrůznějších podmínek vyhodnotí, zda zavolá precedenční analýzu, jejíž účel je právě vyhodnocení nejrůznějších výrazů.

Bottom-up analýza se nachází ve funkci `bottom_up()`. Při jejím zavolání jsou iniciálizovány rovnou tři zásobníky: zásobník pro ukládání všech jednotlivých tokenů, zásobník pouze pro operátory a také zásobník pro ukládání `handlů`. Dalé se také vytvoří `node buffer`, do kterého se ukládají jednotlivé uzly a následně spojují na základě toho, jaká situace v analyzátoru nastane. Proces probíhá podobně jako u top-down analýzy, nyní se však pracuje s precedenční tabulkou. Analyzátor vezme hodnotu aktuálního tokenu a hodnotu vrcholu zásobníku pouze pro operátory a najde v tabulce buňku, kde se tyto dvě položky setkávají. Výsledkem však není žádné pravidlo, ale pouze operátor $>$ nebo $<$. V případě oprátoru $<$ se vytvoří nový handle a také vytvoří nový uzel, do kterého jsou uloženy data aktuálního tokenu. Následně se tento uzel uloží do node bufferu. V případě operátoru $>$ nastává situace, kdy je potřeba všechny položky ze zásobníku až po handle spojit v jeden výraz. Analyzátor tedy vytvoří nový uzel s hodnot "expr"a spustí cyklus, během kterého se postupně odebírá položky ze zásobníků a přidává postupně uzly z node bufferu do nově vytvořeného uzlu. Celý proces se následně opakuje. V případě, kdy analyzátor narazí na neznámý token, výraz tím končí precedenční analýza také. Před jejím koncem se ještě spustí cyklus, během kterého se všechny prozatím vytvořené uzly spojí do jednoho uzlu "expr". Speciálně jsou řešeny závorky, na které když analyzátor narazí, zavolá funkci opět `bottom_up()`. Výraz uvnitř závorky se tedy zpracuje rekurzivně.

2.3 Sémantický analyzátor

Hlavní tělo sémantické analýzy (`semantic.c`) je založeno na práci se zásobníkem rámců tabulek symbolů vytvořených během analýzy kódu. Na nulté pozici je globální rámec funkcí, typy jejich parametrů, typy návratových hodnot a globální proměnné. Na dalších indexech jsou uloženy pouze lokální proměnné, přičemž každé další zanoření vytváří další rámec.

Tabulky symbolů jsou implementovány jako tabulky s rozptýlenými položkami (více v sekci 3). Každá položka tabulky má jednoznačný identifikátor `char *key`, indikující jméno proměnné nebo funkce, jedná se také o klíč pomocí které ho se v tabulce vyhledává. Rozdíl mezi funkcí a proměnnou je indikován pomocí počtu návratových hodnot `int ret_values`. Pokud se jedná o proměnnou, tak je hodnota -1, v opačném případě je nezáporná. Parametry a návratové typy funkcí jsou uloženy jako lineárně vázaný seznam `fce_item_t *fce`. Pokud se jedná o proměnnou, je seznam prázdný. Dalším parametrem je pravdivostní hodnota `bool local`, která značí, jestli jde o lokální proměnnou či ne. Poslední dva řetězce souží pouze pro proměnné a značí hodnotu `char *value` a datový typ `char *type`, v případě, že jde o funkci, jsou tyto dvě položky prázdné. `struct htab_item *next_h_item` symbolizuje ukazatel na další položku řádku tabulky s rozptýlenými položkami.

Sémantický analyzátor je úzce spjat s gramatikou, na základě které jsme implementovali funkce kontrolující jednotlivé aspekty kódu. Přímo také pracuje se syntaktickým stromem, vytvořeným předešlou syntaktickou analýzou, který postupně prochází.

2.4 Generace mezikódu

Generování kódu tvoří propojení mezi jazkyky IFJ21 a IfjCode21. Je závislé na všech předchozích krocích. Pouze v minimu nutných případech implementuje sémantické kontroly a plně využívá syntak-

tického stromu vstupního programu. Výsledný kód implementuje všechny zabudované funkce jazyka IFJ21.

2.4.1 Přístup ke generování kódu

Vytváření generátoru se plně řídilo gramatickými pravidly syntaktického stromu. Před začátkem generování kódu je nejdříve zajištěno pár nutných kroků k jeho kompatibilitě se vstupním syntaktickým stromem, které vznikly důsledkem kompromisů mezi autory jednotlivých modulů. Generátor dále převede všechny escape sekvence, mezery a pár jiných znaků jazyka IFJ21 na validní escape sekvence jazyka IFJCode21.

2.4.2 Zabudované funkce

Zabudované funkce jazyka ifj21 jsou implementovány v jazyce IFJCode21 a jsou uloženy v souboru built_in.fc. V souboru je zabudovaných i pár dalších funkcí, většinou slouží k účelu implementace vnitřního zapouzdření jazyka IFJ21, jako je implicitní konverze z integeru na float, nebo ke kontrole nil hodnot. Do výsledného kódu je soubor vždy hned po začátku generování nahrán a přeskočen instrukcí skoku. Zabudovaná funkce write, která využívá nekonečný počet parametrů je implementována odlišně od ostatních zabudovaných funkcí. Ve fázi příprav před generováním kódu se v syntaktickém stromě každé volání funkce write s počtem n parametrů převede na n volání funkcí write s jedním parametrem.

2.4.3 Zastínění identifikátorů

IFJCode21 nepodporuje zastínění, proto je třeba, aby každá proměnná měla ryze unikátní identifikátor. Ve fázi příprav před začátkem generování kódu se každý identifikátor přejmenuje podle jeho typu a zanoření. Všechny identifikátory označující globální proměnné se přejmenují na id_global a podobně. Díky tomuto kroku nemůže dojít k záměně s jakýmkoliv jiným identifikátorem. Některé zabudované funkce a funkce generování kódu používají vnitřní proměnné jazyka IFJCode21 uložené v globálním rámci, pomocí přejmenování identifikátorů, ale nelze názvů těchto proměnných docílit pomocí vstupního programu.

2.4.4 Fáze generování kódu

Na nejvrchnější úrovni syntaktického stromu se nachází <main-list>, z něj lze volat a definovat funkce, popřípadě globální proměnné. Uvnitř těl funkcí se nachází <stmt-listy>, v nichž nalezneme ostatní programové instrukce. Generování kódu s tímto rozložením počítá, naviguje se v syntaktickém stromě a postupně generuje kód jednotlivých částí.

2.4.5 Generování definice funkcí

Při nalezení uzlu, který značí definici funkce vygeneruje program kód jejího těla, které je odděleno z vrchu instrukcí JUMP na její konec, čímž se zajistí, že interpret funkci vykoná pouze při zavolání. Ihned za oddělovací instrukcí JUMP se nachází návěští funkce a instrukce PUSHFRAME, která vloží dočasný rámec na vrchol zásobníku lokálních rámců. Při definici funkcí je třeba mít v dočasném rámci předpřipravené i nadefinované návratové hodnoty (v naší implementaci jsou značené procentem, pořadníkovým číslem od 1 do nekonečna a písmenem r), v některých případech je ale nutné návratové hodnoty nadefinovat až v samotném těle funkce. Dále je očekáváno, že hodnoty všech příslušných parametrů dané funkce budou uloženy v proměnných nyní v lokálním rámci pod identifikátorem ve formátu procento a pořadníkové číslo od 1 do nekonečna. Dále následuje kód jednotlivých statementů, funkce je zakončena instrukcemi POPFRAME a RETURN.

2.4.6 Volání funkcí

Při volání funkcí se dodrží všechna pravidla již zmíněná při generování definic funkcí. Po nalezení uzlu obsahující volání funkce se zavolá instrukce CREATEFRAME, která vytvoří dočasný rámec k uschování parametrů a získání návratových proměnných. Po načtení parametrů do příslušných proměnných se zavolá instrukce CALL společně s názvem požadované funkce. Pokud bylo volání funkce součástí uzlu přiřazení, využíjí následně návratové hodnoty uložené na dočasném rámci.

2.4.7 Vyčíslování výrazů

Následující sekce se věnuje logickým a výpočetních výrazům. Pokud funkce narazí na aritmetickořetězcový výraz expr, spustí funkci eval_expr, která vygeneruje kód, který pomocí zásobníkových operací výraz vyčíslí a zanechá výsledek na vrcholu zásobníku. Výraz pak lze jednoduše ze zásobníku využít na libovolný účel. Vyčíslení logických výrazů je obdobné, avšak nepotřebuje zásobníkové operace. Místo toho využije klasické tříadresné porovnávací instrukce a zanechá výsledek v proměnné COMP_RES v globálním rámci. Vyčíslování výrazů se všeobecně potýká s nevalidními typy a hodnotami a implicitní konverzí typu integer na float.

2.4.8 Podmíněné výrazy a cykly

Poslední sekcí, která stojí za zmínku je generovnání podmíněných struktur. V případě generování konstrukce if-else je práce jednoduchá. Program nejprve vygeneruje kód pro vyčíslení logického výrazu a podle jeho výsledku připraví dvě návěští: návěští else a návěští endif. Obě návěští jsou doplněna názvem funkce, ze které jsou volána a unikátním číslem značící dosavadní počet použitých návěští. Generování while cyklu je o něco složitější. Před začátkem generování kódu cyklu a kódu vyčíslení podmínky upraví jazyk C syntaktický strom tak, aby veškeré lokální definice proměnných byly předsunuty před začátek cyklu a na jejich místo vloží statementy přiřazení. Následně pomocí návěští while a endwhile spolu s unikátním číslem a podmíněným skokem na bázi podmínky režíruje množinu požadovaných příkazů.

3 Tabulka s rozptýlenými položkami

Implementace tabulky s rozptýlenými položkami se nachází v souborech `symtable.c` a `symtable.h`. Jako mapovací funkce je použita sdbm funkce viz s požitou magickou konstantou 65599, její výsledek modulo počet řádků funkce určuje index prvku v tabulce. V tabulce se vyhledává podle řetězce `char *key` značící název funkce nebo proměnné. V případě dlouhého seznamu prvků na řádku je tabulka automaticky realokována a rozšířena. Pro případné hledání, mazání položek, kopírování tabulky, přidávání parametrů funkce, typů do položky seznamu apod. jsou implemntovány pomocné funkce pro usnadnění práce.

4 Další datové struktury

4.1 Deftable

V souborech `deftable.c` a `deftable.h` je implementace tabulky funkcí. Ta obsahuje informaci o tom, zda byla funkce pouze deklarována a nebo definována. Také je zde zaznamenáno, jestli byla funkce volána nebo ne. Tímto se jednoduše na konci sémantické analýzy vyhodnotí, zda nedošlo k volání volání nedefinované funkce, nebo funkce byla pouze deklarována, ale už nebyla definována. Tabulka funkcí je tedy používána výhradně během sémantické analýzy.

4.2 Dynamické string buffery

V souborech `mystring.h` a `mystring.c` jsou implementovány funkce, které pracují s dynamickými string buffery, které se používají při generování kódu. ADT buffer_ automaticky kontroluje, zda byla překročena přiřazená kapacita paměti, dále dokáže pracovat s formátovaným vstupem podobně jako jiné formátované funkce jazyka C.

5 Práce v týmu

5.1 Dorozumívací kanály a správa verzí

Všichni v týmu jsme se již znali a navíc jsme spolubydlící na koleji, takže valná většina naší komunikace byla face-to-face. Pokud jsme se ale nezastihli nebo odjeli na víkend, používali jsme messenger. Pro správu verzí jsme zvolili Git a jeho vzdálený repozitář GitHub.

5.2 Rozdělení práce

xkocma08	Vedoucí; Lexikální analyzátor; LL-gramatika; dokumentace
xbabek02	Generace kódu; LL-gramatika
xjanda28	Sémantický analyzátor; Tabulka symbolů; Testy
xohnut01	Syntaktický analyzátor; Syntaktický strom

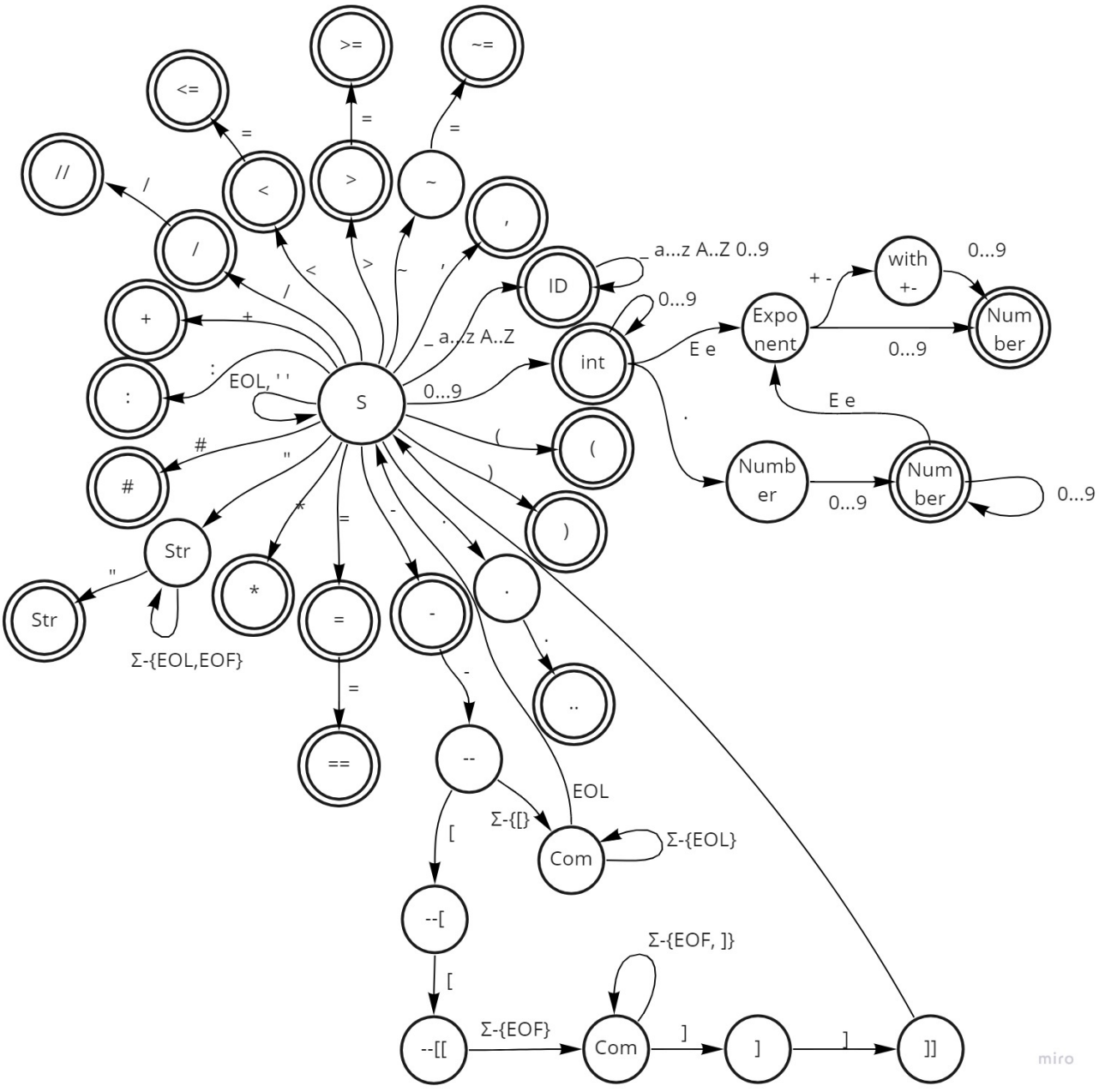
6 Závěr

Na projektu jsme začali pracovat relativně brzo, ale kvůli projektům jsme se do full-time práce pustili až v první čtvrtině listopadu. Potýkali jsme se s nejasnostmi v zadání a občasnými problémy s Gitem, ale projekt jsme nakonec stihli, i přesto, že většina týmu měla ještě projekt do ITU, na kterém museli také pracovat.

Celkově nám projekt přinesl hodně zkušeností ohledně překladačů a algoritmů, ale také bezesných nocí a nervů.

7 Diagramy a tabulky

Následující část obsahuje Konečný automat lexikálního analyzátoru, LL-gramatiku použitou při syntaktické analýze a její pravidla, LL-tabulku a precedenční tabulku použitou při bottom-up analýze.



Obrázek 1: Konečný automat lexikálního analyzátoru

LL-Gramatika:

(1) <prog> → require "ifj21"<main-list>
(2) <main-list> → <def-decl-fcall> <main-list>
(3) <main-list> → ϵ
(4) <stmt-list> → <stmt> <stmt-list>
(5) <stmt-list> → ϵ
(6) <stmt> → id <assign-or-fcall>
(7) <stmt> → <decl-local>
(8) <stmt> → <while>
(9) <stmt> → <if>
(10) <stmt> → <return>
(11) <assign-or-fcall> → (<param-list>)
(12) <assign-or-fcall> → <id-list> = <f-or-item-list>
(13) <f-or-item-list> → expr <item-another>
(14) <f-or-item-list> → id <fcall-or-item-list>
(15) <fcall-or-item-list> → (<param-list>)
(16) <fcall-or-item-list> → <item-another>
(17) <id-list> → , id <id-list>
(18) <id-list> → ϵ
(19) <item> → id
(20) <item> → expr
(21) <return> → return <return-list>
(22) <return-list> → ϵ
(23) <return-list> → <return-f-or-items>
(24) <return-f-or-items> → expr <return-f-or-items'>
(25) <return-f-or-items> → id <f-or-return-list>
(26) <f-or-return-list> → (<param-list>) <return-f-or-items'>
(27) <f-or-return-list> → <return-f-or-items'>
(28) <return-f-or-items'> → , <return-f-or-items>
(29) <return-f-or-items'> → ϵ
(30) <param-list> → ϵ
(31) <param-list> → <item-list>
(32) <item-list> → <item> <item-another>
(33) <item-another> → , <item> <item-another>
(34) <item-another> → ϵ
(35) <def-decl-fcall> → global id : <f-or-type>
(36) <def-decl-fcall> → function id (f-arg-list) <return-types> <stmt-list> end
(37) <def-decl-fcall> → id (<param-list>)
(38) <decl-local> → local id : <type> <decl-assign>
(39) <f-or-type> → function (<types>) <return-types>
(40) <f-or-type> → <type> <decl-assign>
(41) <decl-assign> → = <f-or-item>
(42) <decl-assign> → ϵ
(43) <f-or-item> → expr
(44) <f-or-item> → id <id-or-fcall>
(45) <id-or-fcall> → ϵ
(46) <id-or-fcall> → (<param-list>)
(47) <f-arg-list> → <f-arg> <f-arg-another>
(48) <f-arg-list> → ϵ
(49) <f-arg-another> → , <f-arg> <f-arg-another>

(50) <f-arg-another> → ϵ
(51) <f-arg> → id : <type>
(52) <return-types> → ϵ
(53) <return-types> → : <type> <type-list>
(54) <type-list> → , <type> <type-list>
(55) <type-list> → ϵ
(56) <types> → <type> <type-list>
(57) <types> → ϵ
(58) <if> → if <cond> then <stmt-list> else <stmt-list> end
(59) <while> → while <cond> do <stmt-list> end
(60) <cond> → <item> <cond-oper> <item>
(61) <type> → integer
(62) <type> → string
(63) <type> → number
(64) <type> → nil
(65) <cond-oper> → <
(66) <cond-oper> → >
(67) <cond-oper> → ==
(68) <cond-oper> → =
(69) <cond-oper> → <=
(70) <cond-oper> → >=

	require	id	(	)	=	expr	,	return	global	:	function	end	local	if	else	while	integer	string	number	nil	<	>	==	~=	<=	>=	ε
<prog>	1																										
<main-list>		2							2		2																3
<def-decl-fcall>		37							35		36																
<f-or-type>											39						40	40	40	40							
<f-arg-list>		47		48																							
<return-types>		52						52	52	53	52	52	52	52		52											52
<stmt-list>		4						4				5	4	4	5	4											
<param-list>		31		30		31																					
<types>				57													56	56	56	56							
<type>																	61	62	63	64							
<f-arg>		51																									
<f-arg-another>				50			49																				
<type-list>		55		55			54	55	55		55	55	55	55		55											55
<stmt>		6						10					7	9		8											
<item-list>		32				32																					
<f-or-item>		44				43																					
<assign-or-fcall>			11		12		12																				
<decl-local>													38														
<while>																59											
<if>														58													
<return>								21																			
<item>		19				20																					
<id-list>					18		17																				
<f-or-item-list>		14				13																					
<decl-assign>		42			41			42	42		42	42	42	42	42	42											42
<cond>		60				60																					
<return-list>		23				23		22				22	22	22	22	22											
<item-another>		34		34			33	34				34	34	34	34	34											
<fcall-or-item-list>		16	15				16	16				16	16	16	16	16											
<cond-oper>																					65	66	67	68	69	70	
<return-f-or-items>		25				24																					
<id-or-fcall>		45	46					45	45		45	45	45	45	45	45											45
<return-f-or-items'>		29					28	29				29	29	29	29	29											
<f-or-return-list>		27	26				27	27				27	27	27	27	27											

Obrázek 2: LL-Tabulka

	+	-	*	/	//	#	id	..	(	)	$
+	>	>	<	<	<	<	<		<	>	>
-	>	>	<	<	<	<	<		<	>	>
*	>	>	>	>	>	<	<		<	>	>
/	>	>	>	>	>	<	<		<	>	>
//	>	>	>	>	>	<	<		<	>	>
#	>	>	>	>	>		<		<		>
..							<	<	<	>	>
id	>	>	>	>	>	>		>		>	>
(	<	<	<	<	<	<	<	<	<	=	
)	>	>	>	>	>	>		>		>	>
$	<	<	<	<	<	<	<	<	<		

Obrázek 3: Precedenční tabulka