VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentácia k IFJ a IAL projektu

Implementácia prekladača imperatívneho jazyka IFJ20.
Tím 088, varianta II

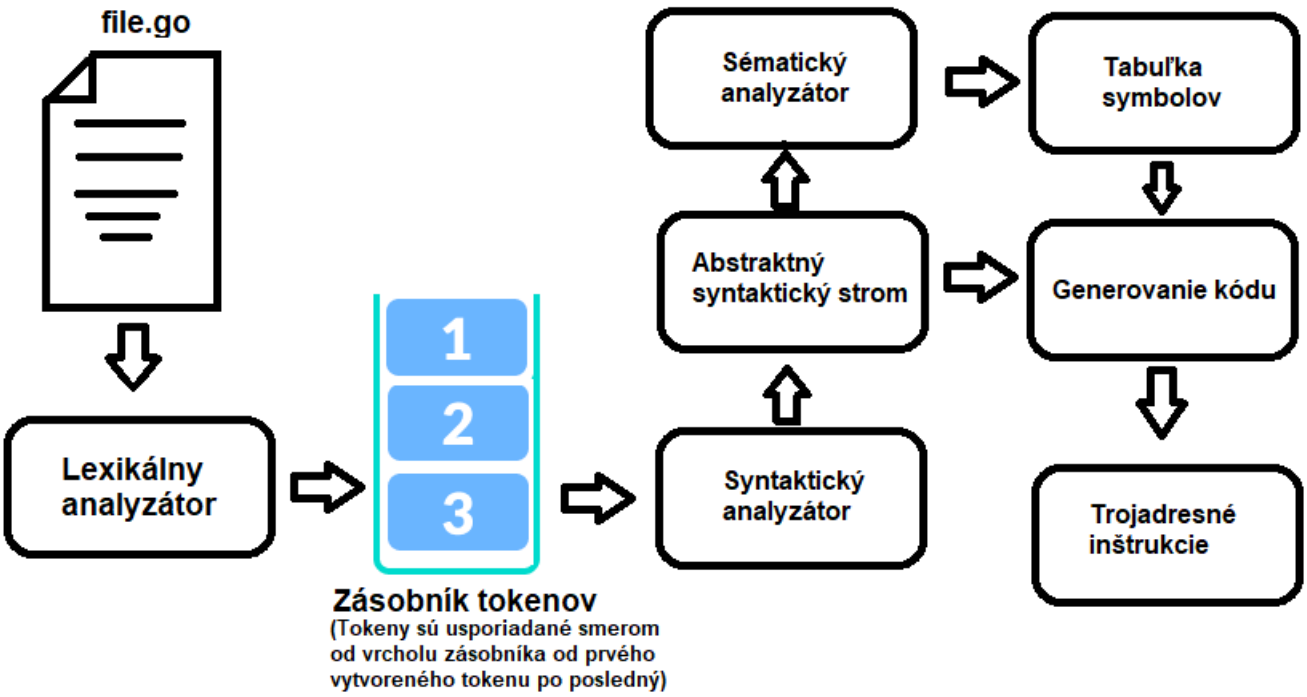
Na projekte pracovali:
Vedúci: Peter Koprda, xkoprd00: 25%
Daniel Paul, xpauld00: 25%
Pavol Babjak, xbabja03: 25%
Viliam Holík, xholik14: 25%

2020/2021

Obsah

1 Úvod

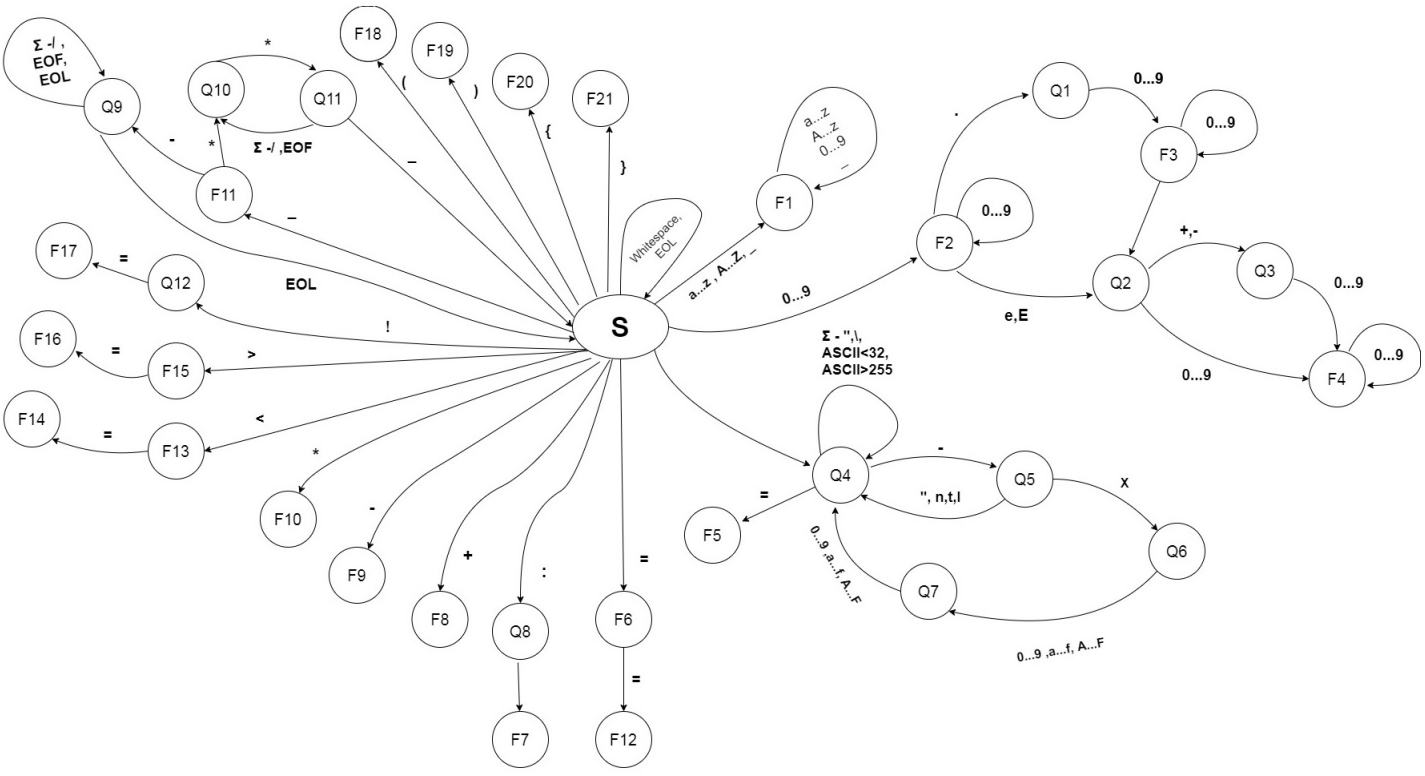
Tento dokument slúži ako dokumentácia, ktorá popisuje návrh a implementáciu prekladača imperatívneho jazyka IFJ20. Jazyk IFJ20 je zjednodušenou podmnožinou jazyka Go, čo je staticky typovaný imperatívny jazyk.
Prekladač načíta zdrojový súbor, z ktorého vytvorí tokeny, ktoré uloží na zásobník. Pomocou syntaktickej analýzy, ktorá si berie tokeny zo zásobníku sa vytvorí abstraktný syntaktický strom. Sématika využíva abstraktný syntaktický strom na ošetrenie chýb a vytvorenie tabuľky symbolov. V generovaní kódu sa využíva tabuľka symbolov a abstraktný syntaktický strom na vytvorenie trojadresných inštrukcií, ktoré sú výstupom prekladača.



Obr. 1: Schéma programu

2 Lexikálna analýza

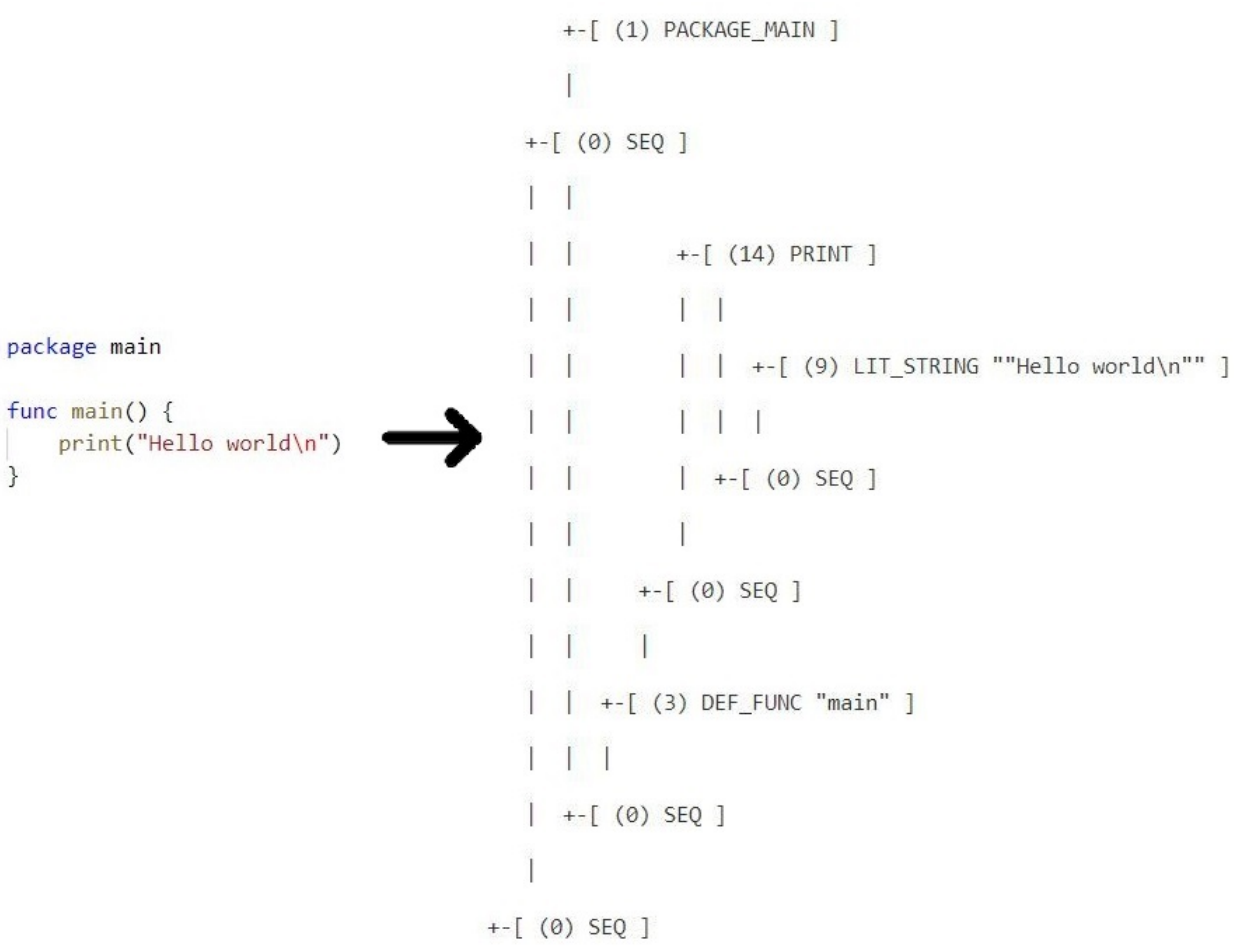
Pri tvorbe prekladača sme začali implementáciou lexikálnej analýzy. Táto časť ako jediná pracuje so zdrojovým textom. Lexikálny analyzátor zbaví zdrojový text nedôležitých častí (bielych znakov a komentárov) a prevádza lexémy na tokeny. Token je štruktúra zložená z typu a hodnoty. Typy tokenu sú napr. EOL, EOF, identifikátor, kľúčové slovo, celé alebo desatinné číslo, reťazec, porovnávacie a aritmetické operátory. Lexikálny analyzátor je implementovaný ako deterministický konečný automat vytvorený pomocou diagramu 2.
Tento konečný automat je implementovaný v súbore scanner.c v ktorom je nekonečne opakujúci switch, kde každý case predstavuje jeden stav v našom automate. Ak program načíta znak, ktorý nie je v jazyku IFJ20 definovaný, tak je program ukončený s návratovou hodnotou 1. Inak sa prechádza do ďalších stavov a načítavajú sa ďalšie znaky až pokiaľ sme nezistili typ tokenu. Vygenerované tokeny sme následne ukladali na zásobník, ktorý sme použili v syntaktickej analýze na generovanie abstraktného syntaktického stromu.



Obr. 2: Diagram konečného automatu

3 Syntaktická analýza

Riadi sa LL-gramatikou (6) a je implementovaná metódou rekurzívneho zostupu podla pravidiel v LL tabuľke. Neterminál je reprezentovaný funkciou, ktorá aplikuje pravidlá gramatiky. Vstupom pre našu syntaktickú analýzu je zásobník tokenov, ktorý vytvoril scanner. Pomocou funkcie *getToken* si syntaktický analyzátor zoberie token z vrcholu zásobníku a na základe neho aplikuje dané pravidlo gramatiky. Ak aktuálny token nesedí do žiadneho pravidla gramatiky, program končí návratovou hodnotou 2. V prípade, že je zásobník tokenov prázdny a gramatika povoluje skončiť ε vráti syntaktická analýza návratovú hodnotu 0. Vytváranie výrazov je riešené precedenčnou tabuľkou, ktorá je použitá v syntaktickom analyzátore vo funkcií *expr*. Výsledkom syntaktického analyzátoru je abstraktný syntaktický strom, ktorý sa používa pri sématickej analýze a generovaní kódu. Tento abstraktný syntaktický strom sa vytvára už spomenutou metódou rekurzívneho zostupu.
(Implementácia v súbore parser.c a štruktúry v libmine.h)



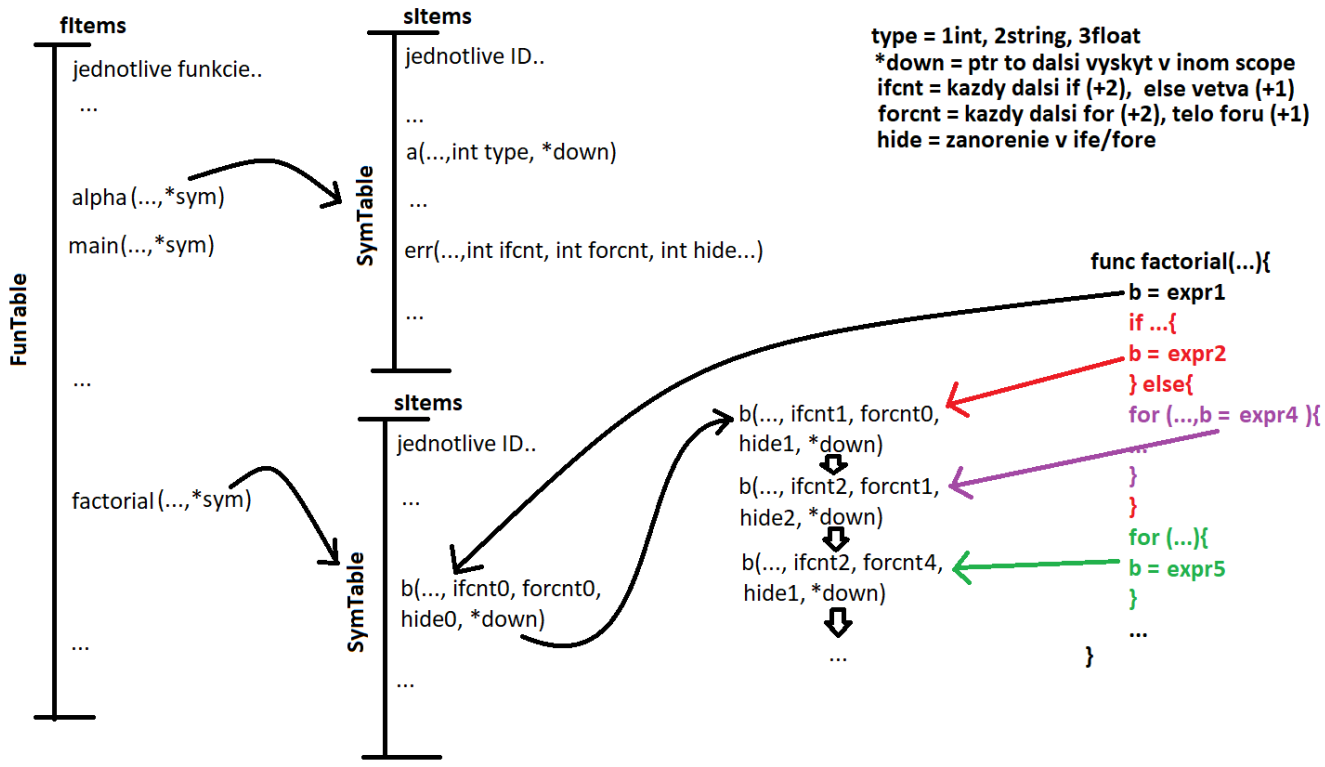
Obr. 3: Konverzia zdrojového kódu na abstraktný syntaktický strom

4 Sématická analýza

Ako vstupné dáta pre vykonanie sémantickej analýzy prijímame abstraktnú dátovú štruktúru Abstract Syntaxt Tree (ďalej AST), ktorá je vytvorená počas syntaktickej analýzy parserom. Základným prvkom tejto štruktúry a zdrojom dát pre sémantickú analýzu sú takzvané Nodes, máme vopred definované pravidlá pre ich vytváranie syntaktickým analyzátorom, vrátane aplikácie precedenčnej tabuľky jednotlivých operácii už počas vytvárania AST. Následná sémantická analýza je tzv. dvojpriechodová, pri prvom priechode si najskôr nájdeme všetky funkcie, jednotlivé dáta o nich (najmä názov funkcie, počet parametrov a návratových hodnôt príslušných dátových typov) si uložíme do tabuľky s rozptýlenými položkami *FunTable*. Akonáhle prejdeme všetkými funkciami povrchne, vchádzame do ich jednotlivých tiel, jednotlivé symboly si vkladáme do tabuľky symbolov *SymTable* pričom kontrolujeme základné sémantické pravidlá zdrojového kódu jazyka Ifj20. Je to najmä kontrola dátových typov pri jednotlivých operáciách vrátane správnosti počtu a typov návratových hodnôt, počtu parametrov pri volaní funkcie, prípadné delenie nulovou konštantou. Pokiaľ pri analýze narazíme na chybu, je táto chyba vypísaná na štandardný výstup a prekladač sa zastaví s daným kódom chyby.

5 Tabuľka s rozptýlenými položkami

Implementácia tejto tabuľky vychádzala z našich znalostí z predmetu IAL, pričom jej implementácia je aj súčasťou zadania spoločného projektu pre daný predmet. Jednotlivé prototypy funkcií a dátové štruktúry sme zadeklarovali v hlavičkovom súbore *symtable.h*, tento hlavičkový súbor následne využíva samotný *symtable.c* kde sme deklarovali jednotlivé funkcie podľa zadania. Samotná inicializácia tabuľky prebieha pred začiatkom sémantickej analýzy pomocou funkcií *ftInit* pre funkcie a *stInit* pre symboly.
Pre lepšie pochopenie implementácie tejto našej tabuľky prikladáme nasledovný obrázok.



Obr. 4: Tabuľka s rozptýlenými položkami

Jednotlivé funkcie sa nachádzajú v štruktúre *FunTable*, pričom každá funkcia obsahuje svoju tabuľku symbolov. Každá tabuľka symbolov má medzi sebou zreťazené vo forme lineárne viazaného zoznamu štruktúru **down*, ktoré slúžia pre prácu so symbolmi v jednotlivých scopoch, nakoľko tieto symboly nemajú globálnu platnosť a v zdrojovom kóde programu platia pre ne špecifické pravidlá. Pri práci s touto tabuľkou využívame hashovaciu funkciu *hashCode*. Pri následnom hľadaní v tabuľke *FunTable* a *SymTable* využívame implementovaných funkcií *ftSearch* a *stSearch* pričom ako parameter tejto funkcie je okrem samotnej tabuľky aj daný kľuč podľa ktorého sa daná funkcia snaží nájsť danú položku.

6 Generovanie kódu

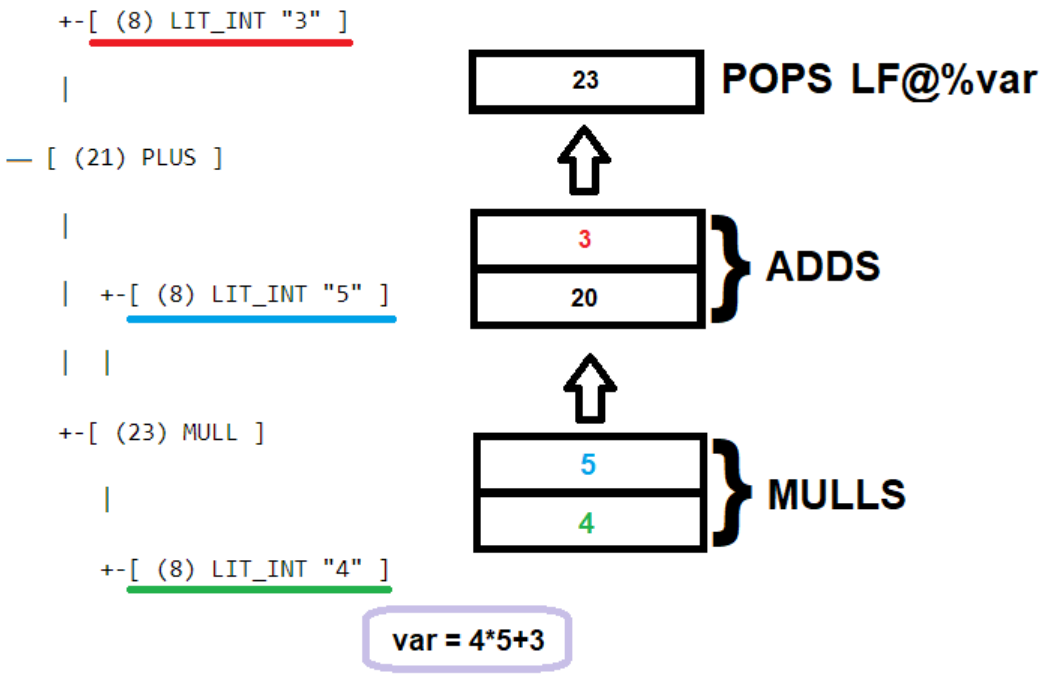
Generovanie cieľového kódu prebieha až po ukončení všetkých predchádzajúcich analýz, t.j. po ukončení lexikálnej, syntaktickej a sémantickej analýzy. Cieľový kód sa vygeneruje iba vtedy, ak všetky predchádzajúce analýzy skončia s návratovou hodnotou 0. Generovanie kódu je implementované ako samostatný modul v súbore code_gen.c, pričom jeho rozhranie sa nachádza v súbore code_gen.h. Na začiatku je vygenerovaná hlavička kódu .IFJcode20, definície globálnych premenných v globálnom rámci a skok do hlavného tela programu.

6.1 Generovanie funkcií

Funkcie sú tvorené návestím v tvare $meno_funkcie a svojím lokálnym rámcom. Pre každú funkciu sa vždy generuje lokálny rámec, ktorý je po odchode z funkcie dostupný ako dočasný rámec. Pred volaním funkcie sú hodnoty argumentov uložené do dočasného rámca a po vstupe do funkcie sú všetky parametre uložené na lokálnom rámci.

6.2 Generovanie výrazov

Výrazy sú spočítané na zásobníku pomocou metódy, ktorá sa vyučuje v predmete ISU. Pomocou prechodu *postorder* sa na zasobník vložia literály a potom sa podľa rootu vykoná na zásobníku operácia. Po vykonaní výpočtu sa výsledok pomocou inštrukcie POPS vloží do premennej.



Obr. 5: Vizualizácia počítania výrazov pomocou postorder prechodu

7 Rozdelenie práce

Akonáhle sme si ujasnili štruktúru projektu, tak sme si prácu rozdelili následovne:

Peter Koprda: Lexikálna analýza, generovanie kódu, testovanie
Daniel Paul: Syntaktická analýza, generovanie kódu, dokumentácia
Pavol Babjak: Sématická analýza, testovanie, tabuľka symbolov, dokumentácia
Viliam Holík: Sématická analýza, generovanie kódu, tabuľka symbolov

8 Záver

Projekt bol veľmi praktický a prínosný. Osvojili sme si v ňom prácu v týme, prácu s githubom a taktiež uplatnili teoretické znalosti získané na predmetoch IFJ a IAL. Komunikácia prebiehala priebežne cez platformu Discord. Na projekte sme pracovali v priebehu semestra, ale napriek tomu by sme potrebovali viac času projekt doladiť. V priebehu vývoja sme sa stretli s nejasnosťami v zadaní, ktoré boli objasnené na fórach. Tento projekt nám priniesol veľa znalostí ohľadne prekladačov, algoritmov a abstraktných dátových typov.

9 LL - gramatika

```
(1) prolog → PACKAGE MAIN EOL program
(2) prolog → EOL prolog
(3) program → FUNC IDENTIF L_RBRACKET params R_RBRACKET returntype L_CBRACKET EOL stmt R_CBRACKET eol program
(4) program → ε
(5) params → IDENTIF type params2
(6) params → ε
(7) params2 → COMMA params
(8) params2 → ε
(9) type → INT
(10) type → FLOAT
(11) type → STRING
(12) returntype → L_RBRACKET type returntype2 R_RBRACKET
(13) returntype → ε
(14) returntype2 → COMMA type returntype2
(15) returntype2 → ε
(16) stmt → IDENTIF stmt2 EOL stmt
(17) stmt → IF expr L_CBRACKET EOL stmt R_CBRACKET ELSE L_CBRACKET EOL stmt R_CBRACKET EOL stmt
(18) stmt → RETURN stmt5 EOL stmt
(19) stmt → FOR id_def SEMICOLON expr SEMICOLON id_init L_CBRACKET EOL stmt R_CBRACKET stmt
(20) stmt → PRINT L_RBRACKET terms R_RBRACKET EOL stmt
(21) stmt → ε
(22) stmt2 → DEFVAR expr
(23) stmt2 → INITVAR expr
(24) stmt2 → L_RBRACKET expr arg_comma R_RBRACKET
(25) stmt2 → COMMA stmt3 INITVAR stmt5
(26) arg_comma → COMMA expr arg_comma
(27) arg_comma → ε
(28) stmt3 → IDENTIF stmt4
(29) stmt4 → COMMA stmt3
(30) stmt4 → ε
(31) stmt5 → expr arg_comma
(32) id_def → IDENTIF DEFVAR expr
(33) id_def → ε
(34) id_init → IDENTIF INITVAR expr
(35) id_init → ε
(36) str_arg → LIT_STRING
(37) str_arg → IDENTIF
(38) int_arg → LIT_INT
(39) int_arg → IDENTIF
(40) terms → LIT_INT terms2
(41) terms → LIT_FLOAT terms2
(42) terms → LIT_STRING terms2
(43) terms → IDENTIF terms2
(44) terms → ε
(45) terms2 → COMMA terms
(46) terms2 → ε
(47) eol → EOL
(48) eol → ε
(49) expr → IDENTIF expr3
(50) expr → LIT_INT expr2
(51) expr → LIT_FLOAT expr2
(52) expr → LIT_STRING expr2
(53) expr → L_RBRACKET expr R_RBRACKET
(54) expr → INPUTS L_RBRACKET R_RBRACKET
(55) expr → INPUTI L_RBRACKET R_RBRACKET
(56) expr → INPUTF L_RBRACKET R_RBRACKET
(57) expr → LEN L_RBRACKET str_arg R_RBRACKET
(58) expr → CHR L_RBRACKET int_arg R_RBRACKET
(59) expr → ORD L_RBRACKET str_arg COMMA int_arg R_RBRACKET
(60) expr → SUBSTR L_RBRACKET str_arg COMMA int_arg COMMA int_arg R_RBRACKET
(61) expr2 → PLUS expr
(62) expr2 → MINUS expr
(63) expr2 → MUL expr
(64) expr2 → DIV expr
(65) expr2 → GT expr
(66) expr2 → GT_EQ expr
(67) expr2 → LESS expr
(68) expr2 → LESS_EQ expr
(69) expr2 → EQUAL expr
(70) expr2 → NOT_EQUAL expr
(71) expr2 → ε
(72) expr3 → expr2
(73) expr3 → L_RBRACKET expr arg_comma R_RBRACKET
```

Obr. 6: LL-gramatika

10 LL - tabulka

Odkaz na .xls súbor Tabulka LL gramatiky

11 Precedenčná tabuľka

	+ -	* /	(	)	<	>	<=	>=	==	!=	Identifier, int, float, string	$
+ -	>	<	<	>	>	>	>	>	>	>	<	>
* /	>	>	<	>	>	>	>	>	>	>	<	>
(	<	<	<	=	<	<	<	<	<	<	<	
)	>	>		>	>	>	>	>	>	>		>
<	<	<	<	>							<	>
>	<	<	<	>							<	>
<=	<	<	<	>							<	>
>=	<	<	<	>							<	>
==	<	<	<	>							<	>
!=	<	<	<	>							<	>
Identifier, int, float, string	>	>		>	>	>	>	>	>	>		>
$	<	<	<		<	<	<	<	<	<	<	

Obr. 7: Precedenčná tabuľka