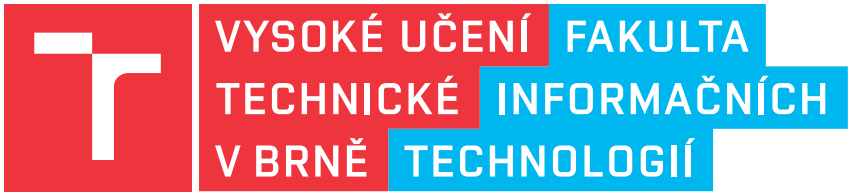


Projektová dokumentace

Implementace překladače imperativního jazyka IFJ17

Tým 104, varianta II

30. listopadu 2017

Dominik Harmim	**(xharmi00)**	25 %
Vojtěch Hertl	(xhertl04)	25 %
Timotej Halás	(xhalas10)	25 %
Matej Karas	(xkaras34)	25 %

Obsah

1 Úvod

Cílem projektu bylo vytvořit program v jazyce C, který načte zdrojový kód zapsaný ve zdrojovém jazyce IFJ17, jenž je zjednodušenou podmnožinou jazyka FreeBASIC a přeloží jej do cílového jazyka IFJcode17 (mezikód).

Program funguje jako konzolová aplikace, které načítá zdrojový program ze standardního vstupu a generuje výsledný mezikód na standardní výstup nebo v případě chyby vrací odpovídající chybový kód.

2 Návrh a implementace

Projekt jsme sestavili z několika námi implementovaných dílčích částí, které jsou představeny v této kapitole. Je zde také uvedeno, jakým způsobem spolu jednotlivé dílčí části spolupracují.

2.1 Lexikální analýza

Při tvorbě překladače jsme začali implementací lexikální analýzy. Hlavní funkce této analýzy je `get_next_token`, pomocí níž se čte znak po znaku ze zdrojového souboru a převádí na strukturu `token`, která se skládá z typu a atributu. Typy tokenu jsou `EOF`, `EOL`, prázdný token, identifikátory, klíčová slova, celé či desetinné číslo, řetězec a také porovnávací a aritmetické operátory a ostatní znaky, které mohou být použity v jazyce IFJ2017. Typ atributu je `union` a dělí se podle typu tokenu. Pokud je typ tokenu řetězec nebo identifikátor, pak bude atribut daný řetězec, když by byl typ tokenu klíčové slovo, přiřadí atributu dané klíčové slovo, pokud číslo, atribut bude ono číslo, u ostatních případů není atribut použit. S takto vytvořeným tokenem poté pracují další analýzy.

Celý lexikální analyzátor je implementován jako deterministický konečný automat podle předem vytvořeného diagramu 1. Konečný automat je v jazyce C jako jeden nekonečně opakující se `switch`, kde každý případ `case` je ekvivalentní k jednomu stavu automatu. Pokud načtený znak nesouhlasí s žádným znakem, který jazyk povoluje, program je ukončen a vrací chybu 1. Jinak se přechází do dalších stavů a načítají se další znaky, dokud nemáme hotový jeden token, který potom vracíme a ukončíme tuto funkci.

Za zmínku stojí zpracovávání identifikátorů a klíčových slov, kde načítáme znaky a průběžně ukládáme do dynamického pole a jakmile odcházíme ze stavu identifikátoru, porovnáváme, jestli načtený řetězec není shodný s nějakým z klíčových slov a podle toho se rozhodneme, zda vracíme atribut a typ tokenu klíčové slovo či identifikátor.

Pro zpracovávání escape sekvencí v řetězci máme vytvořeno pole o velikosti 4, které je zpočátku vynulováno a poté se postupně naplňuje přečtenými čísly, vždy na pozici podle toho, kolikáté číslo zrovna uvažujeme a nakonec celé třímístné čislo převedeme do ASCII podoby.

2.2 Syntaktická analýza

Nejdůležitější částí celého programu je syntaktická analýza.

Až na výrazy se syntaktická analýza řídí LL – gramatikou a metodou rekurzivního sestupu podle pravidel v LL – tabulce 2. Každé pravidlo má svou vlastní funkci, která dostává přes parametr ukazatel na `PData`, což je struktura obsahující proměnné nutné pro správné fungování analýzy. Syntaktická analýza žádá po lexikálním analyzátoru tokeny pomocí `get_next_token`. Tyto tokeny načítá lexikální analyzátor ze zdrojového souboru a zároveň provádí lexikální analýzu.

Aby se příkaz návratu z funkce nevyskytoval v hlavním těle programu, vyřešili jsme to pomocí bool proměnné `in_function`, která symbolizuje, že se právě nacházíme ve funkci.

2.2.1 Zpracování výrazů pomocí precedenční syntaktické analýzy

Precedenční analýza je v syntaktické analýze definována a volána jako ostatní pravidla v LL – gramatice, ale je zvlášť implementována v souboru `expression.c`, a její rozhraní je v souboru `expression.h`.

Při zpracovávání výrazů je použita precedenční tabulka 4. Jelikož operátory + a – mají stejnou asociativitu a prioritu, mohli jsme je zjednodušit do jednoho sloupce a řádku tabulky + −. Totéž jsme mohli udělat s operátory * a / (v tabulce sloupec a řádek * /) a také všemi relačními operátory (v tabulce sloupec a řádek r). Řádek a sloupec i symbolizuje identifikátor, číselnou hodnotu nebo řetězec. Pro získání indexu ze symbolu do tabulky jsme si vytvořili funkci get_prec_table_index, které se jako parametr zadá symbol, a vrátí index do tabulky. Mezi povolené symboly ve výrazech patří všechny operátory, literály, které jsou v tabulce zastoupeny již dříve popsaným i, a závorky. Tyto symboly jsou terminály. Všechny ostatní symboly, které výraz obsahovat nemůže, jsou zastoupeny symbolem $. Řádky tabulky označují vrchní terminál v zásobníku symbolů a sloupce symbol v aktuálním tokenu.

Po spuštění analýzy se podle vrchního terminálu v zásobníku symbolů a symbolu aktuálního tokenu provádějí různé operace a v některých případech se volá funkce get_next_token.

Pokud znak z tabulky je <, vložíme zarážku za vrchní terminál, aktuální symbol vložíme na vrchol zásobníku a zavoláme funkci get_next_token. Pokud znak z tabulky je > a existuje pravidlo, podle kterého se dají zredukovat položky zásobníku symbolů od nejvrchnější položky až po zarážku, tak je zredukujeme a odstraníme zarážku. Během provádění redukce podle stanovených pravidel se otestuje sémantika levého a pravého operandu a pokud je to potřebné a možné, provede se implicitní přetypování jednoho nebo obou operandů. Pokud znak je =, aktuální symbol vložíme na vrchol zásobníku a zavoláme funkci get_next_token. To opakujeme až do chvíle, kdy na místě v tabulce určeném vrchním terminálem v zásobníku symbolů a symbolem aktuálního tokenu žádný znak není. Pokud vrchní terminál i aktuální symbol je $ a na vrcholu zásobníku zůstal jeden výsledný neterminál, syntaktická analýza je úspěšná, v jiných případech končí neúspěšně.

2.3 Sémantická analýza

Ve struktuře PData jsou uloženy tabulky symbolů, lokální pro lokální proměnné, a globální pro funkce. Tabulky symbolů jsou implementovány jako tabulky s náhodným rozptýlením a slouží ke kontrole, zda daný identifikátor existuje a zda souhlasí jeho datový typ, případně návratová hodnota.

2.4 Generování cílového kódu

Generování cílového kódu pro nás znamená generování mezikódu IFJcode17. Kód je generován na standardní výstup po dokončení všech analýz, přičemž jednotlivé dílčí části jsou generovány v průběhu analýz do interní paměti programu. Při generování mezikódu jsou generovány i komentáře pro jeho zpřehlednění.

2.4.1 Rozhraní generátoru kódu

Generování kódu je implementováno jako samostatný modul v souboru code_generator.c, jehož rozhraní se nachází v souboru code_generator.h Rozhraní nabízí 3 základní funkce. Jedná se o funkci, která připraví generování kódu, tato funkce se volá na začátku syntaktické analýzy. Funkce, která uvolní z paměti všechny rezervované prostředky generátorem kódu se volá po dokončení syntaktické analýzy. Poslední důležitá funkce generuje vytvořený kód na patřičný výstup. Rozhraní nabízí spoustu dalších funkcí pro generování jednotlivých částí programu, např. generování začátku funkce, volání funkce, deklarace proměnnné, příkazu input, funkcí pro zpracování výrazů, podmínek, cyklů, aj. Všechny tyto funkce se volají ve správnou chvíli a se správnými parametry v průběhu syntaktické analýzy a syntaktické analýzy výrazů.

2.4.2 Začátek generování

Na začátku generování jsou inicializovány potřebné datové struktury (které jsou na závěr opět uvolněny), vygenerována hlavička mezikódu, která zahrnuje potřebné náležitosti pro korektní interpretaci mezikódu, definici globálních proměnných na globálním rámci a skok do hlavního těla programu. Poté jsou vygenerovány vestavěné funkce, které jsou zapsány přímo v jazyce IFJcode17.

2.4.3 Generování funkcí

Každá funkce je tvořena návěštím ve tvaru `$identifikator_funkce` a svým lokálním rámcem. Pro funkce se vždy generuje výchozí návratová hodnota podle jejího návratového datového typu, která se ukládá na lokální rámec, který je po odchodu z funkce dostupný jako dočasný rámec. Před zavoláním funkce jsou hodnoty parametrů uloženy do dočasného rámce s číslem, které odpovídá pořadí daného parametru a po vstupu do funkce jsou všechny předané parametry uloženy na lokální rámce se svým názvev. Při generování návratu z funkce je předán výsledek zpracovaného výrazu do proměnné s návratovou hodnotou funkce a proveden skok na konec funkce.

2.4.4 Generování výrazů

Všechny výrazy jsou během syntaktické analýzy výrazů ukládány na datový zásobník a v pravou chvíli jsou provedeny patřičné operace s hodnotami na vrcholu zásobníku. Hodnota zpracovaného výrazu se ukládá do globální proměnné na globálním rámci.

2.4.5 Generování návěští

Všechna návěští, pokud neuvažujeme návěští pro funkce a jiná speciální návěští, jsou generovány ve tvaru `$identifikator_funkce%label_deep%label_index`, kde `identifikator_funkce` zajistí unikátnost návěští mezi jednotlivými funkcemi, `label_deep` je hloubka zanoření návěští (například u podmínek a cyklů) a `label_index` je číselná hodnota, která se inkrementuje s každým dalším návěštím uvnitř dané funkce.

2.5 Překladový systém

Projekt je možné přeložit dvěma způsoby, buď nástrojem CMake nebo nástrojem GNU Make. Oba způsoby vytvoří spustitelný soubor `ifj17_compiler`.

2.5.1 CMake

Při vývoji, ladění a testování projketu někteří z nás používali CMake, protože překlad tímto nástrojem je integrován ve vývojovém prostředí CLion, které většina členů týmu používala a protože nastavení pravidel pro překlad tímto nástrojem je jednoduché.

V souboru `CMakeLists.txt` jsou nastavena pravidla pro překlad. Je zde nastaven překladač `gcc` a všechny potřebné parametry pro překlad. Dále je zde uvedeno, že se pro překlad mají používat všechny soubory s příponou `.c` a `.h` a taky je zde uveden název spustitelného souboru.

Soubory pro překlad je možné vygenerovat příkazem `cmake .` a překlad provést příkazem `cmake --build .` nebo pro toto využít nástroje vývojového prostředí.

2.5.2 GNU Make

Jedním z požadavků pro odevzdání projektu bylo přiložit k odevzdanému projektu i soubor `Makefile` a přeložení projektu příkazem `make`. Z tohoto důvodu jsme vytvořili překladový systém i pomocí nástroje GNU Make.

V souboru `Makefile` jsou nastavena pravidla pro překlad. Je zde nastaven překladač `gcc` a všechny potřebné parametry pro překlad. Vytvořili jsme pravidla, které nejdříve vytvoří objektové soubory ze všech souborů s příponou `.c` v daném adresáři za pomoci vytvořených automatických pravidel a překladače `gcc`, který dokáže automaticky generovat tyto pravidla, více viz *Dependecy Generation* v kapitole *C and C++* knihy [2]. Následně je ze všech objektových souborů vytvořen jeden spustitelný soubor.

Nástroj GNU Make jsme mimo překlad využili i pro automatické zabalení projektu pro odezvdání, generování dokumentace, spouštění automatických testů a čištění dočasných souborů.

3 Speciální algoritmy a datové struktury

Při tvorbě překladače jsme implementovali několik speciálních datových struktur, které jsou v této kapitole představeny.

3.1 Tabulka s rozptýlenými položkami

Implementovali jsme tabulku s rozptýlenými položkami, která slouží jako tabulka symbolů, což byl mimo jiné i požadavek vyplývající ze zadání, který souvisí s předmětem IAL. Tabulku jsme implementovali jako tabulku s explicitním zřetězením synonym, protože jsome potřebovali, aby počet položek, který do ní můžeme uložit byl dynamický a teoreticky neomezený. Pro zřetězení synonym jsme použili lineárně vázané seznamy. Velikost mapovacího pole jsme dimenzovali tak, aby byla rovna prvočíslu a její naplnění nepřesáhlo 75 %. Jako optimální velikost jsme tedy zvolili číslo 27 457.

Jako mapovací funkci jsme použili GNU ELF Hash použitou v algoritmu, který používá UNIX ELF. Tato funkce je variantou PJW Hash. Použití této mapovací funkce jsme vyhodnotili jako nejvhodnější, více viz [3]. Funkci jsme upravili tak, aby vracela datový typ a hodnotu, která je přípustná pro naše mapovací pole.

Každá položka tabulky obsahuje unikátní klíč v podobě řetězce, kterým je identifikátor funkce či proměnné. Dále obsahuje ukazatel na další prvek (synonymum) a datovou část. Datová část obsahuje informaci o tom, jakého datového typu je daný symbol (v případě funkce se jedná o datový typ návratové hodnoty), pravdivostní hodnoty, které indikují, jestli byl daný symbol již definován a zda je symbol globální. Posledním prvkem datové části, který se používá pouze v případě, že se jedná o funkci, je počet a datové typy formálních parametrů funkce, který je ve formátu řetezce (Pokud např. daná funkce má dva parametry, první typu `Double` a druhý typu `Integer`, uloží se následující řetězec `di`.).

Implementovali jsme i několik potřebných funkcí, které tvoří rozhraní pro práci s tabulkou. Jedná se o následující funkce: inicializace, přidání nové položky, přidání určitého parametru pro konkrétní položku tabulky (v případě, že se jedná o funkci), vyhledání položky, odstranění položky a uvolnění tabulky z paměti. Při implementaci jsme se inspirovali studijní literaturou předmětu IAL [1].

3.2 Dynamický řetězec

Vytvořili jsme strukturu `Dynamic_string` pro práci s řetězci dynamické délky, kterou používáme např. pro ukládání řezězce nebo identifikátoru u atributu tokenu v lexikální analýze, protože dopředu nevíme, jak bude tento řetězec dlouhý.

Daná struktura a rozhraní operací nad ní je popsáno v souboru `dynamic_string.h` a operace jsou implementovány v souboru `dynamic_string.c`.

Struktura v sobě uchovává ukazatel na řezězec, velikost řetězce a informaci o tom, kolik paměti je pro řetězec vyhrazeno. Implementovali jsme operace pro inicializaci struktury, uvolnění vyhrazených dat, přidání znaku na konec řezězce, porovnávání řetězců, kopie celých řezězců a další. Při inicializaci struktury se vyhradí paměťový prostor pouze pro určitý počet znaků (určeno konstantou), až při nedostatku vyhrazené paměti se velikost paměti zvýší.

3.3 Zásobník symbolů pro precedenční syntaktickou analýzu

Implementovali jsme zásobník symbolů v souboru `symstack.c`, který používáme při precedenční syntaktické analýze. Jeho rozhraní je v souboru `symstack.h`. Má implementovány základní zásobníkové operace jako `symbol_stack_push`, `symbol_stack_pop` a `symbol_stack_top`. Pro účely precedenční syntaktické analýzy jsme implementovali další funkce jako `symbol_stack_top_terminal`, která vrátí nejvrchnější terminál, a funkci `symbol_stack_insert_after_top_terminal`, která vloží symbol za nejvrchnější terminál.

Struktura položky zásobníku obsahuje symbol, který jsme získali během precedenční analýzy z tokenu pomocí funkce `get_symbol_from_token`, jejímž parametrem je ukazatel na token, a vrátí symbol ve formě

výčtového typu. Dále obsahuje datový typ tohoto symbolu, který je inicializován pouze tehdy, pokud symbol může mít datový typ, a používá se při sémantické kontrole. Struktura také obsahuje ukazatel na další položku.

4 Práce v týmu

4.1 Způsob práce v týmu

Na projektu jsme začali pracovat koncem října. Práci jsme si dělili postupně, tj. neměli jsme od začátku stanovený kompletní plán rozdělení práce. Na dílčích částech projektu pracovali většinou jednotlivci nebo dvojice členů týmu. Nejprve jsme si stanovili strukturu projektu a vytvořili překladový systém.

4.1.1 Verzovací systém

Pro správu souborů projektu jsme používali verzovací systém Git. Jako vzdálený repositář jsme používali GitHub.

Git nám umožnil pracovat na více úkolech na projektu současně v tzv. větvích. Většinu úkolů jsme nejdříve připravili do větve a až po otestování a schválení úprav ostatními členy týmu jsme tyto úpravy začlenili do hlavní vývojové větve.

4.1.2 Komunikace

Komunikace mezi členy týmů probíhala převážně osobně nebo prostřednictvím aplikace Slack, kde jsme si buď psali přímo mezi sebou nebo si vytvářeli různé skupinové konverzace podle tématu.

V průběhu řešení projektu jsme měli i několik osobních setkání, kde jsem probírali a řešili problémy týkající se různých částí projektu.

4.2 Rozdělení práce mezi členy týmu

Práci na projektu jsme si rozdělili rovnoměrně s ohledem na její složitost a časovou náročnost. Každý tedy dostal procentuální hodnocení 25 %. Tabulka 1 shrnuje rozdělení práce v týmu mezi jednotlivými členy.

Člen týmu	Přidělená práce
Dominik Harmim	vedení týmu, organizace práce, dohlížení na provádění práce, konzultace, kontrola, testování, dokumentace, struktura projektu, generování cílového kódu
Vojtěch Hertl	lexikální analýza, testování, dokumentace, prezentace
Timotej Halás	syntaktická analýza, syntaktická analýza výrazů, sémantická analýza, testování, dokumentace
Matej Karas	syntaktická analýza, sémantická analýza, testování, dokumentace

Tabulka 1: Rozdělení práce v týmu mezi jednotlivými členy

5 Závěr

Projekt nás zprvu trochu zaskočil svým rozsahem a složitostí. Postupem času, až jsme získali dostatek znalostí o tvorbě překladačů na přednáškách IFJ, jsme projekt začali řešit.

Náš tým jsme měli sestaven velmi brzy, byli jsme již předem domluveni na komunikačních kanálech, osobních schůzkách a na používání verzovacího systému, tudíž jsme s týmovou prací neměli žádný problém a pracovalo se nám společně velmi dobře.

Na projektu jsme začali pracovat trochu později, takže jsme neměli časovou rezervu, ale nakonec jsme všechno bez problému stihli. Jednotlivé části projektu jsme řešili většinou individuálně za použití znalostí z přednášek nebo materiálů do předmětů IFJ a IAL.

V průběhu vývoje jsme se potýkali s menšími problémy týkajícími se nejasností v zadání, ale tyto jsme vyřešili díky fóru k projektu. Správnost řešení jsme si ověřili automatickými testy a pokusným odevzdáním, díky čemuž jsme byli schopni projekt ještě více odladit.

Tento projekt nám celkově přinesl spoustu znalostí ohledně fungování překladačů, prakticky nám objasnil probíranou látku v předmětech IFJ a IAL a přinesl nám zkušennosti s projekty tohoto rozsahu.

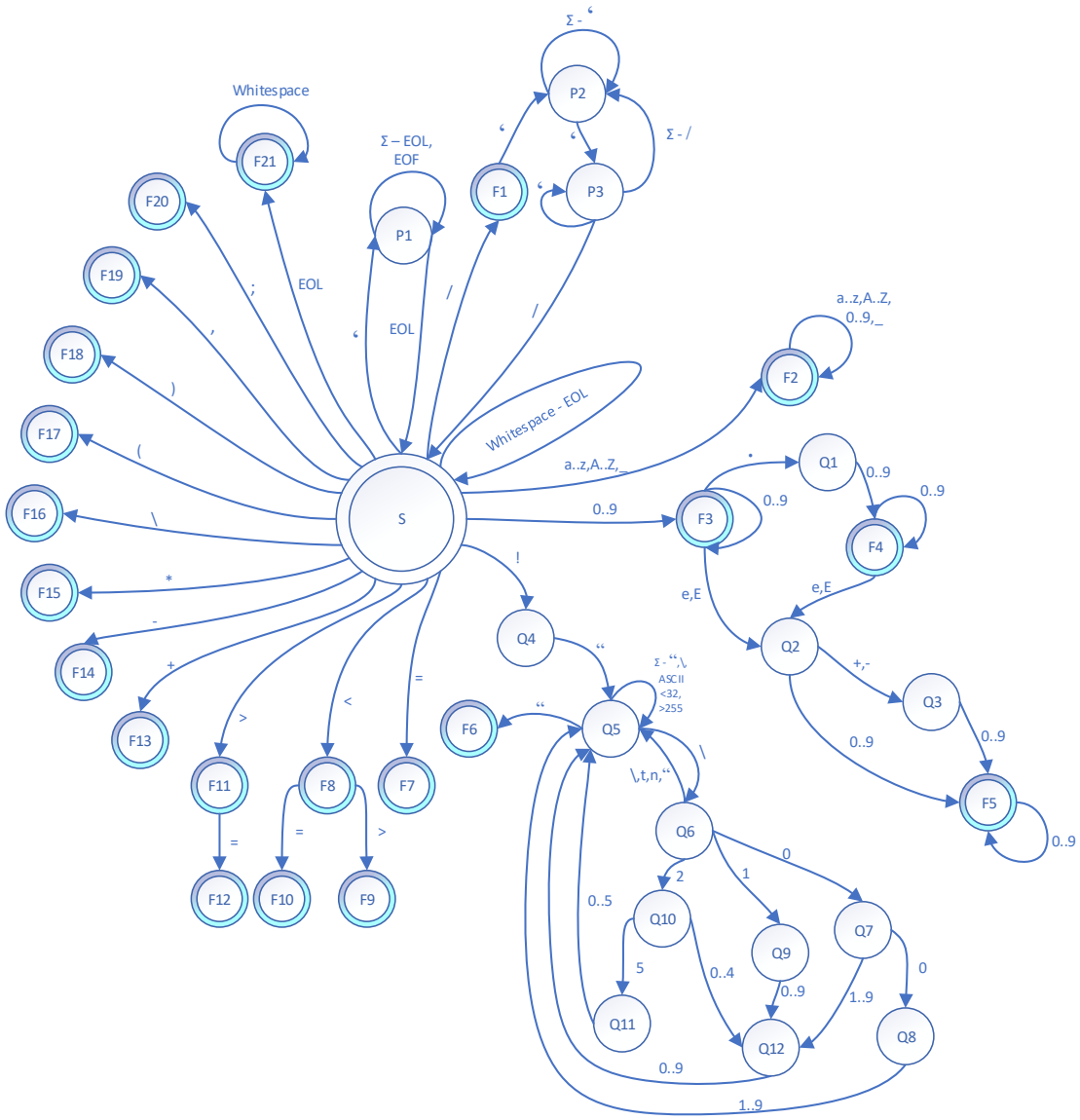
Literatura

[1] Honzík, J. M.; Hruška, T.; Máčel, M.: *Vybrané kapitoly z programovacích technik*. Brno: VUT v Brně, třetí vydání, 1991, ISBN 80-214-0345-4.

[2] Mecklenburg, R. W.; Oram, A.: *Managing projects with GNU make*. Cambridge [Mass.]: O'Reilly, třetí vydání, 2005, ISBN 05-9600-610-1.

[3] Wikipedia: PJW hash function. [online], rev. 5. srpen 2017, [vid. 2017-11-29]. Dostupné z: https://en.wikipedia.org/wiki/PJW_hash_function

A Diagram konečného automatu specifikující lexikální analyzátor



Legenda:

S	START
P1	COMMENTARY
F1	SLASH
P2	BLOCK_COMMENTARY
P3	BLOCK_COMMENTARY_LEAVE
F2	IDENTIFIER_OR_KEYWORD
F3	NUMBER
Q1	NUMBER_POINT
F4	NUMBER_DOUBLE
Q2	NUMBER_EXPONENT
Q3	NUMBER_EXPONENT_SIGN
F5	NUMBER_EXPONENT_FINAL
Q4	STRING_START
Q5	STRING
Q6	STRING_ESCAPE
Q7	STRING_ESCAPE_ZERO
Q8	STRING_ESCAPE_ZERO_ZERO
Q9	STRING_ESCAPE_ONE
Q10	STRING_ESCAPE_TWO
Q11	STRING_ESCAPE_TWO_FIVE
Q12	STRING_ESCAPE_REST
F7	EQUALS
F8	LESS_THAN
F9	NOT_EQUAL
F10	LESS_OR_EQUAL
F11	MORE_THAN
F12	MORE_OR_EQUAL
F13	PLUS
F14	MINUS
F15	TIMES
F16	DIV
F17	LEFT_BRACKET
F18	RIGHT_BRACKET
F19	COMMA
F20	SEMICOLON
F21	EOL

Obrázek 1: Diagram konečného automatu specifikující lexikální analyzátor

B LL – gramatika

1. <prog> -> DECLARE FUNCTION ID (<params>) AS <type> <prog>
2. <prog> -> FUNCTION ID (<params>) AS <type> EOL <statement> END FUNCTION <prog>
3. <prog> -> EOL <prog>
4. <prog> -> <scope>
5. <scope> -> SCOPE EOL <statement> END SCOPE <end>
6. <end> -> EOL <end>
7. <end> -> EOF
8. <params> -> ID AS <type> <param_n>
9. <params> -> ε
10. <param_n> -> , ID AS <type> <param_n>
11. <param_n> -> ε
12. <statement> -> DIM ID AS <data_type> <def_var> EOL <statement>
13. <statement> -> IF <expression> THEN EOL <statement> ELSE EOL <statement> END IF EOL <statement>
14. <statement> -> DO WHILE <expression> EOL <statement> LOOP EOL <statement>
15. <statement> -> ID = <def_value> EOL <statement>
16. <statement> -> INPUT ID EOL <statement>
17. <statement> -> PRINT <expression> ; <print> EOL <statement>
18. <statement> -> RETURN <expression> EOL <statement>
19. <statement> -> ε
20. <def_var> -> = <expression>
21. <def_var> -> ε
22. <def_value> -> ID (<arg>)
23. <def_value> -> ASC (<arg>)
24. <def_value> -> CHR (<arg>)
25. <def_value> -> LENGTH (<arg>)
26. <def_value> -> SUBSTR (<arg>)
27. <def_value> -> <expression>
28. <arg> -> <value> <arg_n>
29. <arg> -> ε
30. <arg_n> -> , <value> <arg_n>
31. <arg_n> -> ε
32. <value> -> INT_VALUE
33. <value> -> DOUBLE_VALUE
34. <value> -> STRING_VALUE
35. <value> -> ID
36. <print> -> <expression> ; <print>
37. <print> -> ε
38. <type> -> INTEGER
39. <type> -> DOUBLE
40. <type> -> STRING

Tabulka 2: LL – gramatika řídící syntaktickou analýzu

C LL – tabulka

	DECLARE	FUNCTION	EOL	SCOPE	EOF	ID	,	DIM	IF	DO	INPUT	PRINT	RETURN	=	ASC	CHR	LENGTH	SUBSTR	INTEGER	DOUBLE	STRING	INT_VALUE	DOUBLE_VALUE	STRING_VALUE	$
<prog>	1	2	3	4																					
<scope>				5																					
<end>			6		7																				
<params>						8																			9
<param_n>							10																		11
<statement>						15		12	13	14	16	17	18												19
<def_var>														20											21
<def_value>						22									23	24	25	26							27
<arg>																						28	28	28	29
<arg_n>							30																		31
<value>						35																32	33	34	
<print>																									36, 37
<type>																			38	39	40				

Tabulka 3: LL – tabulka použitá při syntaktické analýze

D Precedenční tabulka

	+ -	* /	\	r	(	)	i	$
+ -	>	<	<	>	<	>	<	>
* /	>	>	>	>	<	>	<	>
\	>	<	>	>	<	>	<	>
r	<	<	<		<	>	<	>
(	<	<	<	<	<	=	<	
)	>	>	>	>		>		>
i	>	>	>	>		>		>
$	<	<	<	<	<		<	

Tabulka 4: Precedenční tabulka použitá při precedenční syntaktické analýze výrazů