Projektová dokumentácia

Implementácia prekladača imperatívneho jazyka IFJ24

Tým xbockaa00 Varianta: TRP-izp

Andrej Bočkaj	(xbockaa00)	40%
Jakub Fiľo	(xfiloja00)	40%
Pavel Glvač	(xglvacp00)	20%
Jan Kubíček	(xkubicj00)	0%

30.11.2024

Obsah

1 Úvod

Cieľom projektu je vytvoriť program v jazyku C, ktorý načíta zdrojový kód zapísaný v zdrojovom jazyku IFJ24[1], a preloží ho do cieľového jazyka IFJcode24[2]. Ak by preklad prebehol bez chýb, program vráti návratovú hodnotu 0. V prípade že by došlo k chybe, program vráti nenulovú návratovú hodnotu špecifikovanú v zadania.

Prekladač funguje ako konzolová aplikácia, ktorá načíta riadiaci program zo štandardného vstupu a generuje výsledný medzikód na štandardný výstup. Všetky chybové hlásenia/varovania či ladiace výpisy sú generované na štandardný chybový výstup.

2 Návrh a implementácia

Implementácia projektu pozostávala zo vzájomnej kooperácie parciálnych častí programu popísaných v tejto kapitole.

2.1 Lexikálna analýza

Pri implementácii prekladača je základom lexikálna analýza, ktorá je implementovaná vo funkcii `get_token`. Táto funkcia postupne číta znaky zo vstupu a na ich základe generuje tokeny. Celý proces riadi deterministický konečný automat, ktorý určuje, aký typ tokenu bude generovaný – či ide o identifikátor, kľúčové slovo, číselnú hodnotu alebo iné znaky definované v jazyku IFJ2024.

Tokeny sú reprezentované štruktúrou `Token`, ktorá obsahuje tri hlavné komponenty:

1. Typ tokenu (`Token_type`), ktorý určuje kategóriu tokenu (napr. identifikátor, ľavú zátvorku, bodkočiarku a pod.).
2. Hodnotu tokenu (`Token_Value`), ktorá predstavuje skutočnú hodnotu v reťazci, napríklad číslo.
3. Typ kľúčového slova (`Keyword`), ktorý označuje, či token je kľúčové slovo, a ak áno, jeho konkrétnu hodnotu.

Tokeny sa využívajú v ďalších fázach analýzy, kde sa spracúvajú podľa syntaktických pravidiel jazyka. Lexikálny analyzátor je založený na deterministickom konečnom automate, ktorý prechádza stavmi v závislosti od čítaných znakov. Funkcia `get_token` využíva nekonečný cyklus (switch-case), ktorý podľa sekvencie čítaných znakov prepína medzi rôznymi stavmi automatu. Ak sa počas spracovávania znaku narazí na taký, ktorý možno okamžite spracovať, vráti prislúchajúci token bez toho, aby prepínala stav automatu.

Ak dôjde k chybe (napríklad neplatnej sekvencii znakov), funkcia automaticky zavolá `handle_error(ERR_LEX)`, ktorá signalizuje chybu v lexikálnej analýze a ukončí program s návratovým kódom 1.

[1] Jazyk IFJ24 je zjednodušenou podmnožinou jazyka Zig (vo verzií 0.13.0)
[2] Medzikód

2.2 Syntaktická analýza

Riadiaca časť celého programu je syntaktická analýza, ktorá je implementovaná striedavým prístupom zhora-dole a zdola-hore.

2.2.1 SA zhora-dole

Konštrukcia syntaktickej analýzy zhora-dole je založená na LL-gramatike, ktorá využíva metódu rekurzívneho zostupu podľa pravidiel v LL-tabuľke. Všetky neterminály definované v LL-gramatike majú svoju vlastnú funkciu.

Syntaktická analýza funguje spôsobom žiadania o spracovaný token lexikálnou analýzou pomocou funkcie `get_token(file)`.

2.2.2 SA sdola-hore

Konštrukcia syntaktickej analýzy zdole-hore je založená na precedenčnej syntaktickej analýze. Jej implementácia sa nachádza v sekcií syntaktickej analýzy zhora-dole. Je volaná ako funkcia `EXPRESSION(file, token)`.

Funkcia algoritmu spočíva v plnení dvoch zásobníkov. Prvý zásobník `precStack` je určený na uchovávanie relačných operátorov, terminálov a aj neterminálov ktoré sa do zásobníku vkladajú/vyberajú podľa precedenčnej tabuľky. Vyberáme stĺpec a riadok podľa najvrchnejšieho terminálu v zásobníku `precStack` a terminalu, ktorý je na vstupe algoritmu. Druhú zásobník `ruleStack` sa napĺňa terminálmi len v prípade redukcie zo zásobníka `precStack` podľa určeného pravidla. V tomto zásobníku dostávame usporiadanie terminálov, z ktorých dokážeme jednoducho vytvoriť abstraktný syntaktický pod strom pre každý výraz.

2.3 Sémantická analýza

Sémantická analýza kontroluje správnosť programového kódu, najmä z hľadiska dátových typov, deklarácií premenných, výrazov a rozsahov. Tento proces nasleduje po syntaktickej analýze, ktorá naplní abstraktný syntaktický strom (AST). Po vytvorení stromu sémantická analýza vykonáva rôzne kontroly, pričom využíva tabuľku symbolov, ktorá uchováva informácie o deklarovaných premenných (typ, hodnotu, konštantu), funkciách a ďalších symboloch.

Typová kontrola: Overuje, či sú výrazy kompatibilné podľa svojich typov (napr. pri priradení hodnôt, aritmetických operáciách, alebo pri volaní funkcií s nesprávnymi parametrami). Zabezpečuje, že operátory používajú kompatibilné typy.

Overenie deklarácií: Kontroluje, či sú všetky premenné správne deklarované a či ich typy zodpovedajú použitiu v programe.

Kontrola rozsahov: Pre každú premennú zisťuje, či sa používa v správnom rozsahu, či už lokálne alebo globálne. Tabuľka symbolov uchováva informácie o rozsahu premenných, čo umožňuje správne overenie prístupu k nim.

Spracovanie funkcií: Kontroluje definície funkcií (názov, typ parametrov, návratový typ) a pri ich volaní overuje kompatibilitu typov parametrov a návratového typu s typmi používanými v programe.

Okrem toho kontroluje aj cykly, podmienky a ďalšie konštrukcie kódu, pričom tabuľka symbolov je neustále využívaná na overenie správnosti týchto operácií.

Ak sa počas analýzy objaví chyba (napr. nedefinovaná premenná), program vyvolá chybu pomocou funkcie handle_error(), ktorá zastaví vykonávanie a vráti prednastavený chybový kód (napr. ERR_UNDEFINED_ID). Tento proces sa vykonáva v hlavnej funkcii ProcessTree(), ktorá volá príslušné funkcie podľa spracovávaného kódu (napr. podmienky, výrazy), ktoré sa potom volajú navzájom. Týmto spôsobom sa sémantická analýza dostane až na koniec programu.

2.4 Generátor kódu

Generátor kódu prekladá zdrojový kód v jazyku IFJ24 do medzi kódu IFJcode24. Pracuje s abstraktným syntaktickým stromom (AST), ktorý je vytvorený počas syntaktickej a sémantickej analýzy, a generuje príslušné inštrukcie pre vykonanie programu.

Implementácia

Implementácia generátora zahŕňa niekoľko hlavných častí:

- **Deklarácie premenných**:
  Funkcie generateGlobalVarDecl a generateLocalVarDecl generujú inštrukcie pre deklarácie globálnych a lokálnych premenných, prípadne aj ich inicializáciu a priradenie hodnôt.
- **Deklarácie konštánt**:
  Funkcie generateGlobalConstDecl a generateLocalConstDecl zabezpečujú správne generovanie konštánt, ktoré sú nemodifikovateľné počas vykonávania programu.
- **Kontrolné štruktúry**:
  Pre podmienené príkazy a cykly sa používajú funkcie ako generateIfStatement a generateWhileStatement, ktoré generujú štítky (LABEL) a skoky (JUMP) na riadenie toku programu.
- **Vstavané funkcie**:
  Funkcia generateBuildInFuncions generuje kód pre vestavné funkcie jazyka IFJ24, ako napríklad write, readi32 atď. Tieto funkcie sú často využívané pre prácu s dátami.
- **Generovanie funkcií**:
  Funkcie generateFunctionParams a generateFunctionEnd zabezpečujú generovanie štruktúry funkcií vrátane ich parametrov, ako aj operácie súvisiace s rámcami. Volanie funkcií sa realizuje pomocou funkcie generateFunctionCall.

2.5 Makefile

Podstatou súčasťou projektu bolo implementovanie súboru Makefile, kde sú nastavené pravidlá pre preklad ako napríklad prekladač gcc a všetky potrebné parametre. Hneď ako prvé bolo potrebné doplniť linkovanie, pri ktorom sa skompilované objekty spájajú do jedného spustiteľného súboru.

Súbor makefile sme okrem prekladu využili aj na automatické zabalenie projektu pre odovzdanie a čistenie dočasných či skompilovaných objektov.

3 Práca v týme

Pre efektívnu prácu v tíme sme si museli na prvých schôdzkach dohodnúť pravidlá komunikácie, verzovací systém a hlavne rozdelenie práce.

3.1 Verzovací systém

Pre správu súborov projektu sme zvolili verzovací systém Git súčasne s využitím vzdialeného repozitára na platforme GitHub.

3.2 Komunikácia

Prevažná väčšina komunikácie medzi členmi tímu bola zabezpečená aplikáciou Discord. Prezenčné schôdzky v priestoroch školy boli uskutočňované v intervale dvoch týždňov.

3.3 Rozdelenie práce

Zo začiatku sme sa snažili rozložiť prácu na projekte rovnomerne, no postupom času sa začala prejavovať menšia aktivita od určitých členov tímu. Na zaznamenávanie pokroku a definovanie individuálnych úloh sme zvolili používanie aplikácie Linear, ktorá nám pomohla presnejšie určiť výsledné hodnotenie podľa odrobenej práce.

Meno člena tímu	Rozdelenie práce
Andrej Bočkaj	Vedenie tímu, SA zhora-dole, SA zdola-hore, Dokumentácia
Jakub Fiľo	Lexikálna analýza, Sémantická analýza, Tabuľka symbolov, Dynamic string
Pavel Glvač	Generátor kódu, Doxygen, Abstraktný strom
Jan Kubíček	Unity tests

3.3.1 Odchýlky od rovnomerného rozdelenia bodov

Jan Kubíček

- Ohlásil v deň 26.10.2024 že nebude môcť venovať projektu dostatočnú aktivitu na absolvovanie zápočtu.

Pavel Glvač:

- Zanedbal detailne vypracovanie abstraktného stromu a tabuľky symbolov.
- Venoval čas a úsilie na implementovanie generátora cieľového kódu.

4 Záver

Výsledkom našej práce bolo vytvorenie komplexnej aplikácie.

Počas semestra sme získavali potrebné informácie ohľadom tvorby prekladačov ktoré sme vedeli využiť pri našom projekte.

Postupom času sme si viac uvedomovali rozsiahlosť a nepredvídateľnú časovú náročnosť častí programu, ktorú bolo potrebné vyladiť. Testovanie a ladenie bolo komplikovanejšie než sme predpokladali z čoho vznikali časové straty.

Zistili sme ako zlepšiť počiatočný návrh programu spolu s kontrolou priebežnej funkcionality.

Na projekte sme začali pracovať včas ale ignorovanie termínov on niektorých členov tímu nás dostalo pod časový tlak.

Počas vývoja sme sa stretli s problémami kvôli nejasnostiam zo zadania. Stalo sa že sme vytvorili návrh algoritmu, ktorý bolo potrebné prepísať kvôli horšej funkčnosti a nekompatibilite s inými časťami programu.

Tento projekt nám priniesol veľa znalostí ohľadom fungovania prekladačov ako aj znalosti a skúsenosti s projektmi väčšieho rozsahu. Prakticky nám objasnil preberanú látku na predmetoch IFJ a IAL.

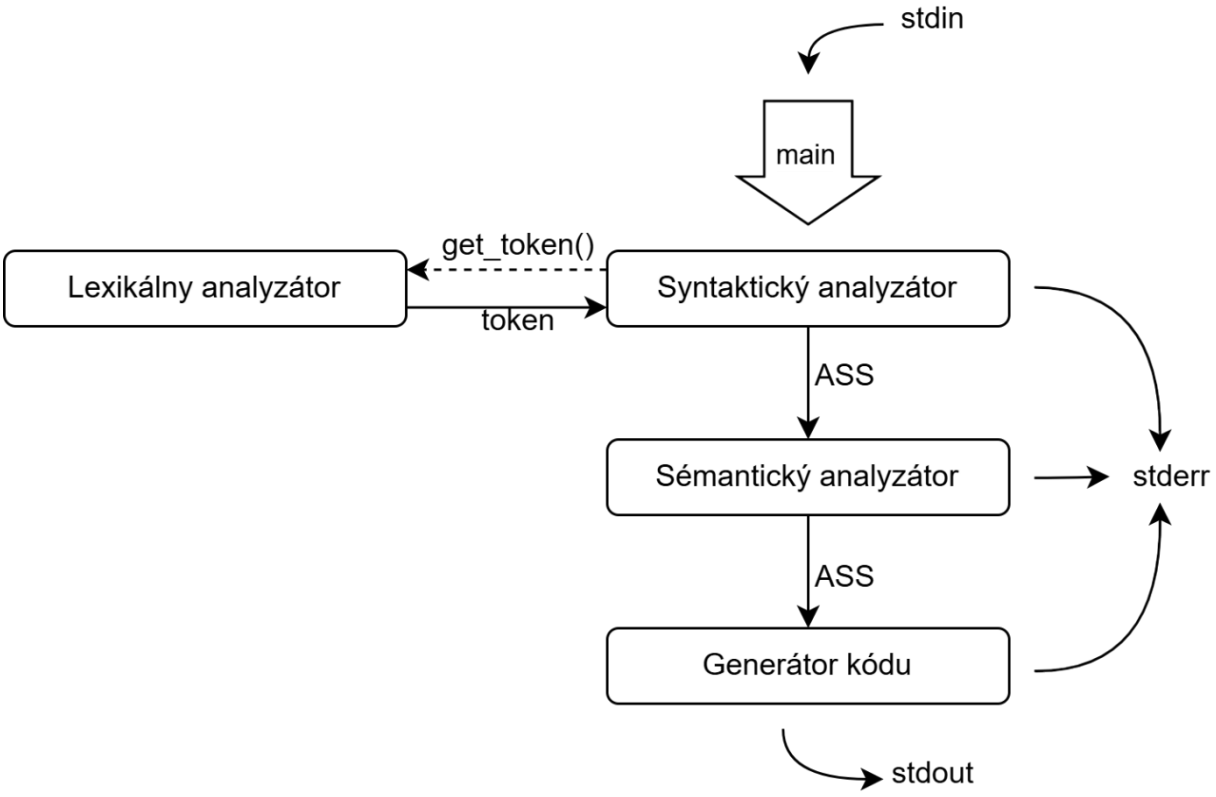
5 Literatúra

M. Stanis, "DJB2 hash function," *The Art of Code*, Mar. 12, 2020. [Online]. Available: https://theartincode.stanis.me/008-djb2/

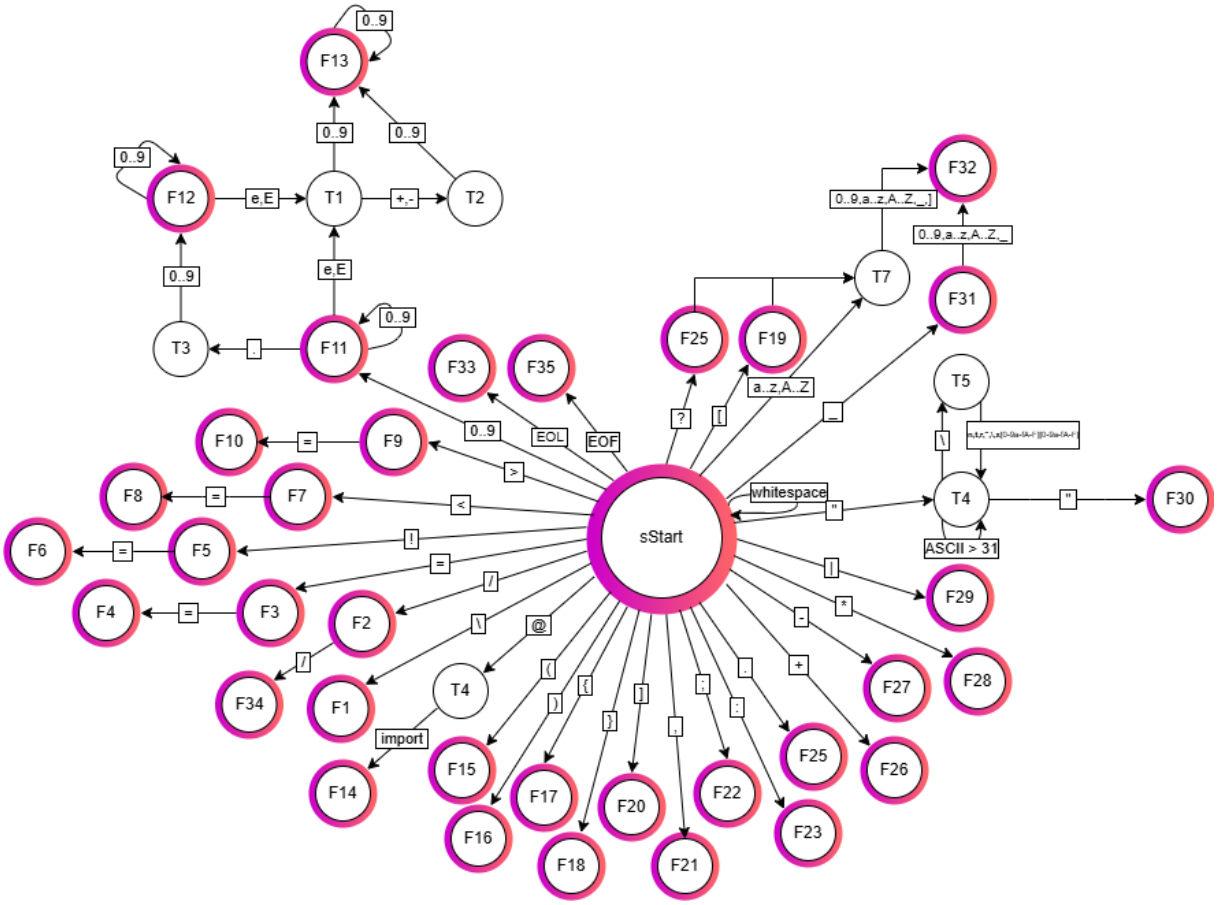
6 Členenie implementačného riešenia

IFJ-PROJECT01

- doc - Programová dokumentácie
- tests - Unity testy, Interpret, Testovacie vstupy
- src
  - ast - implementácia abstraktného syntaktického stromu stromu
  - Code_generator - implementácia generátora cieľového kódu
  - error - definovanie chybového hlásenia
  - lexical_analyser - implementácia lexikálnej analýzy pre spracovávanie tokenov
  - Makefile - definovanie pravidiel pre preklad spolu s linkovaním programu
  - newstring - pomáha ukladať jednotlivé charaktery do pola
  - semantic - implementácia sémantickej analýzy
  - stack - implementácia zásobníka pre ukladanie terminalov a neterminalov
  - symtable - implementácia tabuľky symbolov
  - syntactic_analysis - implementácia syntaktickej analýzy



7 FSM diagram pre lexikálnu analýzu



Legenda:

F1 sBackSlash	F13 sExponent	F21 Comma	F29 Pipe
F2 sDivision	T1 sExponentStart	F22 Semicolon	T4 sStringLiteral
F3 sAssign	T2 sExponentSign	F23 Colon	T5 sEscapeSequence
F4 sEqual	T3 sDot	F24 Dot	F30 sStringLiteral
F5 sExclamation	T4 sImport“@”	F25 sQuestionMark	F31 Dispose
F6 sNotEqual	F14 sImport	F26 Addition	F32 sKeywordOrIdentifier
F7 sLessThan	F15 LeftParenthesis	F27 Subtraction	T7 sKeywordOrIdentifier
F8 sLessEqual	F16 RightParenthesis	F28 Multiplication	F33 EOL
F9 sGreaterThan	F17 LeftCurlyBracket	F29 Pipe	F34 sComment
F10 sGreaterEqual	F18 RightCurlyBracket	T4 sStringLiteral	F35 sEnd
F11 sIntLiteral	F19 sLeftSquareBracket	T5 sEscapeSequence	
F12 sFloatLiteral	F20 RightSquareBracket	F30 sStringLiteral	

8 LL - gramatika

1. FIRST ::= VAR_DEF FIRST
2. FIRST ::= CONST_DEF FIRST
3. FIRST ::= FN_DEF FIRST
4. FIRST ::= t_EOF

5. STATEMENT ::= VAR_DEF STATEMENT
6. STATEMENT ::= CONST_DEF STATEMENT
7. STATEMENT ::= CALL_DEF STATEMENT
8. STATEMENT ::= IF_DEF STATEMENT
9. STATEMENT ::= ELSE_DEF STATEMENT
10. STATEMENT ::= WHILE_DEF STATEMENT
11. STATEMENT ::= RETURN_DEF STATEMENT
12. STATEMENT ::= t_}

13. VAR_DEF ::= t_var t_id ASSIGN_VAR
14. CONST_DEF ::= t_const t_id CONST_EXT
15. FN_DEF ::= t_pub t_fn t_id t_(PARAM t_) SCOPE
16. CALL_DEF ::= t_id CALL_EXT
17. IF_DEF ::= t_if t_(EXPRESSION t_) IF_EXT
18. ELSE_DEF ::= t_else SCOPE
19. WHILE_DEF ::= t_while t_(EXPRESSION t_) SCOPE
20. RETURN_DEF ::= t_ret EXPRESSION

21. CONST_EXT ::= t_: VAL_TYPE t_= EXPRESSION
22. CONST_EXT ::= t_= ASSIGN_CONST
23. IF_EXT ::= SCOPE
24. IF_EXT ::= t_| t_id t_| SCOPE
25. CALL_EXT ::= t_(ARG t_)
26. CALL_EXT ::= t_= EXPRESSION
27. CALL_EXT ::= t_: VAL_TYPE t_= EXPRESSION
28. CALL_EXT ::= t_. CALL_OBJ
29. CALL_OBJ ::= t_id t_(ARG t_) t_;

30. SCOPE ::= t_{ STATEMENT
31. ASSIGN_VAR ::= t_;
32. ASSIGN_VAR ::= t_: VAL_TYPE t_= EXPRESSION t_;
33. ASSIGN_VAR ::= t_= EXPRESSION t_;
34. ASSIGN_CONST ::= EXPRESSION
35. ASSIGN_CONST ::= t_import t_(string t_) t_;

36. EXPRESSION ::= t_;
37. EXPRESSION ::= t_)

38. ARG ::= t_id ARGS
39. ARG ::= string ARGS
40. ARG ::= EXPRESSION ARGS
41. ARGS ::= t_)
42. ARGS ::= t_, ARG
43. PARAM ::= t_)
44. PARAM ::= t_id t_: VAL_TYPE PARAM
45. PARAM ::= t_, t_id t_: VAL_TYPE PARAM

46. VAR_TYPE ::= t_var
47. VAR_TYPE ::= t_const
48. VAL_TYPE ::= DataTypes
49. FN_TYPE ::= VAL_TYPE
50. FN_TYPE ::= t_void

9 LL - tabuľka

	eof	id	var	const	pub	if	else	while	return	:	=	|	.	(	)	;	}	{	import	string	,	DataTypes	void
FIRST	4		1	2	3																		
STATEMENT		7	5	6		8	9	10	11								12						
VAR_DEF			13																				
CONST_DEF				14																			
FN_DEF					15																		
CALL_DEF		16																					
IF_DEF						17																	
ELSE_DEF							18																
WHILE_DEF								19															
RETURN_DEF									20														
CONST_EXT										21	22												
IF_EXT												24							23				
CALL_EXT										27	26		28	25									
CALL_OBJ		29																					
SCOPE																		30					
ASSIGN_VAR										32	33					31							
ASSIGN_CONST															34	34			35				
EXPRESSION															37	36							
ARG		38													40	40				39			
ARGS															41						42		
PARAM		44													43						45		
VAR_TYPE			46	47																			
VAL_TYPE																						48	
FN_TYPE																						49	50

10 Precedenčná tabuľka

	relačné	+	-	*	/	premen	(	)	čísla	null	$
relačné		<	<	<	<	<	<	>	<	<	>
+	>	>	>	<	<	<	<	>	<		>
-	>	>	>	<	>	<	<	>	<		>
*	>	>	>	>	>	<	<	>	<		>
/	>	>	>	>	>	<	<	>	<		>
premen	>	>	>	>	>			>			>
(	<	<	<	<	<	<	<	=	<	<	
)	>	>	>	>	>			>			>
čísla	>	>	>	>	>			>			>
null	>							>			>
$	<	<	<	<	<	<	<		<	<	