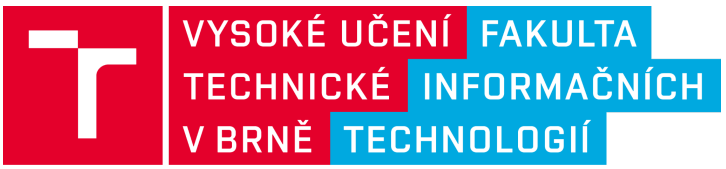


Projektová dokumentace

Implementace překladače jazyka IFJ23

Tým xluklp00, varianta TRP, rozšíření BOOLTHEN

Pavel Lukl - xluklp00 - 25 %

Jan Klanica - xklani00 - 25 %

Denis Milistenfer - xmilis00 - 25 %

Veronika Čalkovská - xcalko00 - 25 %

1. Úvod

Tato projektová dokumentace se zabývá návrhem, implementací a testováním překladače pro imperativní jazyk IFJ23. Projekt vznikl v rámci týmové spolupráce čtyř členů, kteří se podíleli na vývoji a implementaci různých částí překladače. Nad rámec povinného zadání jsme implementovali rozšíření BOOLTHEN, které přidává novou konstrukci jazyka pro explicitní zpracování boolovských výrazů.

Dokumentace komprehenzivně zachycuje klíčové aspekty projektu, systematicky rozdělené do samostatných sekcí. Mezi takové patří týmová spolupráce, samotná iplementace a testování.

2. Týmová spolupráce

Hned na začátku, kdy jsme se pomalu začali pouštět do řešení projektu, jsme se sešli v rámci hlasového hovoru na platformě Discord. Během tohoto setkání jsme vyřešili nejdůležitější otázky týkající se práce v týmu, zejména rozdělení práce. Byly navrženy dva způsoby, jak tento problém vyřešit.

Rozdělit práci rovnoměrně mezi všechny členy týmu, nebo udělat skupinky po dvou a rozdělit projekt na dvě půlky, přičemž ve skupinkách by pak proběhla samostatná dohoda o spolupráci. Vzhledem k časové náročnosti projektu byla vybrána druhá volba, neboť tento způsob umožňoval efektivně rozdělit projekt na část, která se dala implementovat hned na začátku semestru a na část, která se dala implementovat až po nastudování některých přednášek dopředu.

Zároveň se tak jádro překladače, syntaktická analýza a navazující sémantická analýza, dalo plánovat minimálně ve dvou, což bylo velkým přínosem. Jak se později ukázalo, mít vše naplánované a chápat všechny části překladače bylo stěžejní pro správnou implementaci jednotlivých části, které spolu mají spolupracovat.

Projekt byl nakonec vypracován takto:

- Pavel Lukl: LL-gramatika, sémantická analýza, testy
- Jan Klanica: LL-gramatika, rekurzivní sestup, precedenční analýza včetně precedenční tabulky, generování, testy
- Denis Milistenfer: lexikální analýza včetně konečného automatu, generování vestavěných funkcí, testy
- Veronika Čalkovská: hash tabulka, tabulka symbolů

Veškerá komunikace v rámci týmu byla řešena prostřednictvím online platformy Discord. Pro správu a sledování vývoje byl zvolen verzovací systém Git, a to s využitím vzdáleného repozitáře, který byl uložen na GitHubu.

3. Implementace

3.1. Lexikální analýza

Na začátku bylo nutné správně navrhnout strukturu lexikálního analyzátoru, aby správně a efektivně zpracoval vstupní zdrojový kód IFJ23. Proto bylo jako první krok potřeba vytvořit

diagram konečného automatu. Tento krok usnadnil následnou implementaci v souborech scanner.c/.h, scanner_func.h, scanner_types.h a token.h.

Kvůli zvolení dvouprůchodové analýzy a následující praktičnosti spojené s prací se vstupním souborem, bylo rozhodnuto číst ze stringu, do kterého byl vstupní kód vložený, místo přímého čtení ze standardního vstupu. Další výzvou bylo zajistit, aby skener správně pracoval s bílými znaky, což bylo dosaženo pomocnými funkcemi. Tyto funkce tyto znaky buď přeskakovaly, nebo jako například v případě zarovnání ve víceřádkovém řetězci, bylo potřebné některé z těchto znaků odstranit. Pro zpracování víceznakových datových typů, jako jsou identifikátory, čísla a řetězce, byl vytvořen buffer, kde se tyto znaky dočasně ukládaly, než byly přiřazeny k hodnote daného tokenu.

Protože byl skener úzce propojen s parserem, bylo potřebné zajistit, aby generované tokeny obsahovaly všechny potřebné informace a aby byly předávány ve správném pořadí, s možností některé tokeny vracet zpět do skeneru. To bylo dosaženo pomocí zásobníku, do kterého se vrácené tokeny postupně vkládaly, a při požadavku o další token se sáhlo nejprve do tohoto zásobníku.

3.2. Tabulka symbolů

K implementaci tabulky symbolů byla zvolena hashovací tabulka s implicitně zřetězenými položkami. Hashovací tabulka je implementována v souborech hashtable.c/.h a samotná tabulka symbolů v souborech symtable.c/.h.

Tabulka symbolů je implementována jako jednosměrně vázaný seznam. Každý nový blok je reprezentován jako prvek seznamu s identifikátorem a odkazem na vlastní lokální tabulku symbolů.

Lokální tabulka symbolů je pak samotná hashovací tabulka. K hashování je použit algoritmus Polynomial rolling hash function[1]. Z čísla, které dostaneme od hashovací funkce, se vypočítá jak index, tak i krok, o který se v poli budeme posouvat při případném konfliktu. Proto je pro prvotní velikost pole použito prvočíslo 109 (číslo dostatečně velké i pro rozsáhlé bloky), aby se nejlépe využila celá jeho velikost. Po zaplnění se už velikost pouze násobí dvakrát a pro každý prvek je vygenerován nový index.

3.3. Syntaktická analýza

3.3.1. Rekurzivní sestup

Hlavní použitá metoda pro syntaktickou analýzu byl rekurzivní sestup. Jeho implementace se nachází v souborech resursive_parser.c/.h a parser.c/.h. V zásadě byla pouze podle přednášek vytvořena LL-gramatika a následně LL-tabulka, která se aplikovala podle algoritmu do kódu.

Problematickou části bylo identifikování přechodu na precedenční analýzu a kontrolování odřádkování. Oba problémy byly delegovány na sémantickou analýzu. Rozhodování o přechodu bylo provedeno prostřednictvím kontroly následujícího tokenu. Pokud nešlo o identifikátor funkce, proběhl přechod. Kontrola odřádkování byla systematicky realizována pomocí informace z lexikální analýzy, zda byl před daným tokenem nový řádek. V některých situacích nebyl nový řádek explicitně vyžadován.

[1] https://cp-algorithms.com/string/string-hashing.html

3.3.2. Precedenční analýza

Analýza výrazů byla realizována pomocí precedenční analýzy. Její implementace proběhla v souladu s teoretickými principy. Místo dvou zásobníků, jeden pro operátory a jeden pro operandy, byl využit jediný zásobník, který sloužil k ukládání obou prvků. Tento zásobník byl implementován jako lineární seznam, kde operátory, operandy a handle tvořily samostatné položky. Postupným vkládáním na a odebíráním z tohoto listu byl výraz redukován a syntakticky kontrolován v souladu s precedenční tabulkou.

Výsledkem precedenční analýzy výrazu byl abstraktní syntaktický strom, který byl použit při generování. Tento strom byl tvořen současně při redukování výrazu na zásobníku.

Náročnější situaci opět tvořil přechod zpět na syntaktickou analýzu v případě, kdy další token mohl svým typem spadat do výrazu. V takové situaci se kontrolovalo, jestli by takový token dával ve výrazu smysl, anebo vedl na chybu. Při zaručené chybě byl token z výrazu vyčleněn, výraz syntakticky vyhodnocen a token ponechán na vstupu pro rekurzivní sestup.

Implementace precedenční analýzy proběhla v souborech precedence_parser.c/.h, precedence_table.h, ListPP.c/.h, parser.c/.h.

3.4. Sémantická analýza

Analýza všech ostatních konstrukcí jazyka se provádí v sémantické analýze. Implementace byla provedena v souborech semantic_analyser.c/.h a v samotném souboru recursive_parser.c.

Část sémantické analýzy se nachází v recursive_parser.c, což poukazuje na první problém při implementaci. V naší implementaci nešla sémantická analýza oddělit od rekurzivního sestupu, kde je pak v důsledku výsledný kód obtížné pochopit. Většina sémantické analýzy probíhá tak, že se sbírají data o jazykových konstrukcích v rekurzivním sestupu a následovně se tyto data kontrolují. Tyto kontroly jsou implementované jako funkce v souborech semantic_analyser.c/.h, ale často se objevily případy, u kterých tento způsob kvůli charakteru rekurzivního sestupu není možné provést, a tak se zvýšila komplexita kódu.

Dalším problémem, který se vyskytl bylo předávání sbíraných dat mezi funkcemi rekurzivního sestupu. Metodou pokus-omyl se implementace nakonec dostala k předávání dat parametry, přičemž každá jazyková konstrukce a její části jsou zpracovávány na co nejvyšší úrovni sestupu.

Poslední větší překážku tvořili případy, kdy předávání dat parametry nebylo triviální a značně zvyšovalo komplexitu kódu. Jelikož všechny funkce rekurzivního sestupu si už předávali strukturu typu ParserOptions, kde je uložen aktuální token a další informace potřeba pro funkčnost kompileru, bylo pro tyto případy jednodušší do struktury ParserOptions přidat strukturu SemanticContext (definovaná v parser.h), která obsahuje data potřebná pro vyřešení těchto problémů. Tyto případy byly například kontrola nových řádku nebo přítomnost příkazu return ve větvech funkce.

3.5. Generování cílového kódu

Díky pečlivě naplánovanému postupu byla realizace generování kódu prakticky bezproblémová. Namísto okamžitého výpisu výsledného kódu na výstup byly informace uchovávány v bufferech. Tato strategie umožnila odložit některé výpisy na později a předejít tak potenciálním komplikacím, například generování deklarací proměnných před začátkem cyklu while. Celý proces

vyžadoval tři buffery: jeden pro hlavní tělo programu (main), druhý pro funkce a třetí pro oblasti (scope). Do bufferu pro main byl vypisován hlavní tok programu, do bufferu pro funkce obsah funkcí a do bufferu pro scope části kódu, které vyžadovaly dodatečné generování před samotnou oblastí. Tím pádem to, co bylo potřeba dogenerovat dodatečně, bylo posíláno do mainu nebo funkcí, podle dané situace, a obsah zbytku do bufferu scopu. Na konci scopu se buffer scopu "vysypal" do jednoho z již zmíněných bufferů a mohlo se začít nanovo dalším scopem. Konkrétní příklady takových oblastí jsou funkce a if nebo while v mainu.

Předávání parametrů do funkcí bylo řešeno pomocí zásobníku, což eliminuje potřebu zvláštních dočasných rámců (byly použity pouze pro pomocné proměnné ve výrazech). Návratová hodnota z funkcí nebo výrazů byla ukládána na zásobník.

Generování výrazů bylo díky abstraktnímu syntaktickému stromu poměrně přímočaré. Jediné, co bylo potřeba zajistit, byly implicitní konverze literálů. Každý uzel stromu obsahoval navíc informaci o datovém typu, kterým dabývá výsledek dané operace s danými operandy, který byl vyhodnocen při tvorbě stromu v precedenční analýze. Nejvyšší uzel aritmetického výrazu tak obsahoval informaci o datovém typu, na který byly literály na nejnižší úrovni výrazu přetypovány, pokud bylo toto přetypování nezbytné.

Modul generování kódu byl implementován v souborech generation.c/.h a string_buffer.c/.h, avšak generování je voláno z rekurzivního sestupu v souboru recursive_parser.c.

4. Testování

Pro úspěšnou implementaci bylo klíčové také testování. Celkem jsme napsali přibližně 300 testů. První polovina z nich, která se zaměřovala na lexikální, syntaktickou a sémantickou analýzu, byla provedena pomocí unit testing frameworku Criterion. Bohužel se tento framework na náš projekt úplně nehodil, protože nebylo možné projekt rozdělit na samostatné jednotky (units). Přesto jsme se rozhodli vyzkoušet ho a použít kromě testů na stack a list i na celý projekt. Pro druhou půlku testů, které se zaměřovaly na generování, byl napsán testovací program v pythonu.

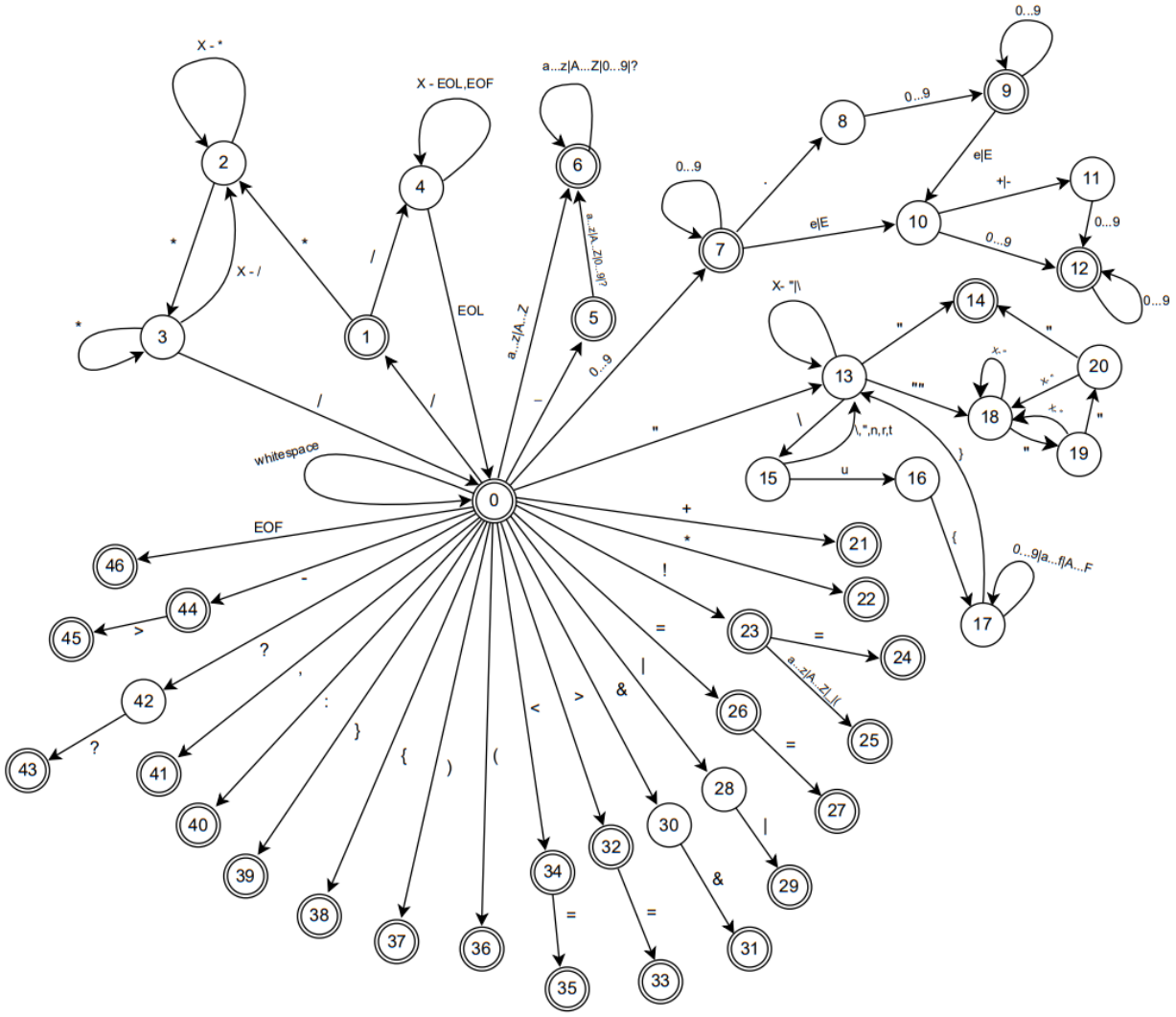
V den odevzdání projekt splňoval všechny naše testy se 100% úspěšností.

5. Závěr

V závěrečných slovech této dokumentace bychom chtěli zdůraznit synergii a kolektivní úsilí našeho týmu během implementace překladače pro imperativní jazyk IFJ23. Celý proces nám nejen posílil dovednosti v oblasti softwarového inženýrství, ale i schopnost spolupráce a plánování v týmu. Přes překážky jsme díky vzájemné podpoře dosáhli kvalitních výsledků. Projekt nám poskytl cenné zkušenosti v oblasti kompilátorů a zdokonalil naše dovednosti v projektovém řízení. Jsme hrdí na dosažené výsledky a věříme, že získané zkušenosti nám poslouží v budoucí profesní dráze.

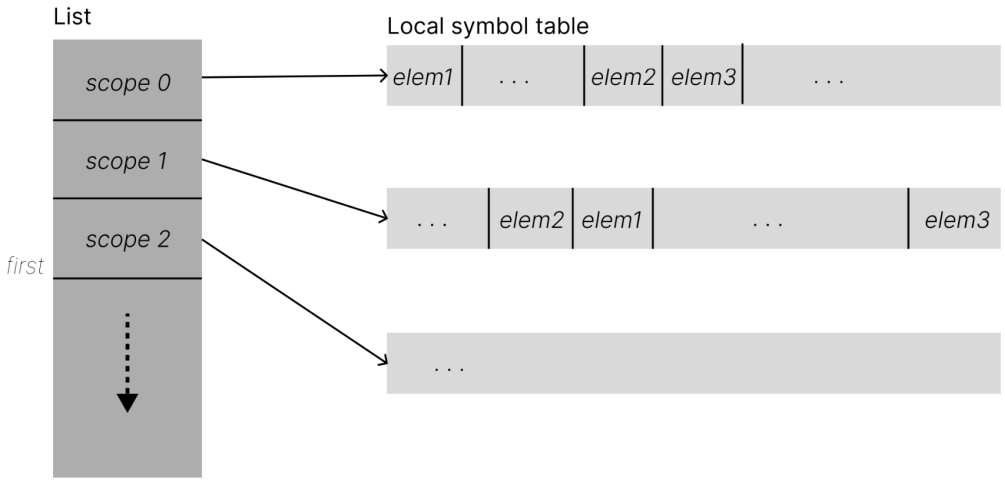
6. Příloha

6.1. Diagram konečného automatu pro lexikální analýzu



0. start
1. division
2. block comment start
3. block comment end
4. one line comment
5. underscore
6. identifier or keyword
7. integer number
8. decimal point
9. decimal value
10. start of exponent in number
11. exponent sign
12. exponent number
13. start of normal string
14. end of normal and multiline string
15. escape in string
16. escape character
17. escape character value
18. start of multiline string
19. normal quote or first closing
20. normal quote or second closing
21. plus
22. multiplication
23. exclemation mark (operator)
24. not equal
25. not
26. assign
27. equal
29. or
31. and
32. greater
33. greater or equal
34. less
35. less or equal
36. left bracket
37. right bracket
38. left curly bracker
39. right curly bracket
40. colon
41. coma
43. operator ??
44. minus
45. arrow
46. end of file

6.2. Návrh tabulky symbolů



6.3. LL-gramatika

1. <PROGRAM> → <COMMAND> <PROGRAM>
2. <PROGRAM> → <FUNCTION_DEFINITION> <PROGRAM>
3. <PROGRAM> → END_OF_FILE
4. <FUNCTION_DEFINITION> → <FUNCTION_HEAD> <SCOPE_BODY>
5. <FUNCTION_HEAD> → KEYWORD_FUNC IDENTIF L_BRACKET <PARAM_LIST> R_BRACKET <_FUNC_IDENTIF_LBRACKET_ARGLIST_RBRACKET>
6. <_FUNC_IDENTIF_LBRACKET_ARGLIST_RBRACKET> → ARROW <DATA_TYPE>
7. <_FUNC_IDENTIF_LBRACKET_ARGLIST_RBRACKET> → eps
8. <PARAM_LIST> → <PARAM> <COMMA_PARAM>
9. <PARAM_LIST> → eps
10. <COMMA_PARAM> → COMA <PARAM> <COMMA_PARAM>
11. <COMMA_PARAM> → eps
12. <PARAM> → IDENTIF <_PARAM_NAME>.
13. <PARAM> → UNDERSCORE <_PARAM_NAME>.
14. <_PARAM_NAME> → IDENTIF COLON <DATA_TYPE>
15. <_PARAM_NAME> → UNDERSCORE COLON <DATA_TYPE>
16. <SCOPE_BODY> → L_CRLY_BRACKET <COMMAND_SEQUENCE> R_CRLY_BRACKET
17. <COMMAND_SEQUENCE> → <COMMAND> <COMMAND_SEQUENCE>
18. <COMMAND_SEQUENCE> → eps
19. <COMMAND> → <RETURN_COMMAND>
20. <COMMAND> → <VARIABLE_DEF>
21. <COMMAND> → <CONDITIONAL_COMMAND>
22. <COMMAND> → <WHILE_COMMAND>
23. <COMMAND> → IDENTIF <_IDENTIF>
24. <_IDENTIF> → ASSIGN <EXPRESSION>
25. <_IDENTIF> → ASSIGN <FUNCTION_CALL>
26. <_IDENTIF> → L_BRACKET <ARG_LIST> R_BRACKET
27. <DATA_TYPE> → KEYWORD_INT.
28. <DATA_TYPE> → KEYWORD_INT_NIL
29. <DATA_TYPE> → KEYWORD_DOUBLE
30. <DATA_TYPE> → KEYWORD_DOUBLE_NIL
31. <DATA_TYPE> → KEYWORD_STRING
32. <DATA_TYPE> → KEYWORD_STRING_NIL
33. <DATA_TYPE> → KEYWORD_BOOL
34. <DATA_TYPE> → KEYWORD_BOOL_NIL

35. <RETURN_COMMAND> → KEYWORD_RETURN <_RETURN>
36. <_RETURN> → <EXPRESSION>
37. <_RETURN> → eps
38. <VARIABLE_DEF> → KEYWORD_VAR IDENTIF <_VARLET_IDENTIF>
39. <VARIABLE_DEF> → KEYWORD_LET IDENTIF <_VARLET_IDENTIF>
40. <_VARLET_IDENTIF> → COLON <DATA_TYPE> <_VARLET_IDENTIF_COLON_TYPE>
41. <_VARLET_IDENTIF> → ASSIGN <EXPRESSION>
42. <_VARLET_IDENTIF> → ASSIGN <FUNCTION_CALL>
43. <_VARLET_IDENTIF_COLON_TYPE> → ASSIGN <EXPRESSION>
44. <_VARLET_IDENTIF_COLON_TYPE> → ASSIGN <FUNCTION_CALL>
45. <_VARLET_IDENTIF_COLON_TYPE> → eps
46. <CONDITIONAL_COMMAND> → KEYWORD_IF <_IF>
47. <_IF> → KEYWORD_LET IDENTIF <SCOPE_BODY> <_IF_LET_IDENTIF_BODY>
48. <_IF> → <EXPRESSION> <SCOPE_BODY> <_IF_LET_IDENTIF_BODY>
49. <_IF_LET_IDENTIF_BODY> → KEYWORD_ELSE <_IF_LET_IDENTIF_BODY_ELSE>
50. <_IF_LET_IDENTIF_BODY> → eps
51. <_IF_LET_IDENTIF_BODY_ELSE> → <SCOPE_BODY>
52. <_IF_LET_IDENTIF_BODY_ELSE> → <CONDITIONAL_COMMAND>
53. <WHILE_COMMAND> → KEYWORD_WHILE <EXPRESSION> <SCOPE_BODY>
54. <FUNCTION_CALL> → IDENTIF L_BRACKET <ARG_LIST> R_BRACKET
55. <ARG_LIST> → eps
56. <ARG_LIST> → <ARG> <COMMA_ARG>
57. <COMMA_ARG> → COMA <ARG> <COMMA_ARG>
58. <COMMA_ARG> → eps
59. <ARG> → IDENTIF <_ARG_NAME>
60. <ARG> → <ARG_VAL>
61. <_ARG_NAME> → COLON <_ARG_NAME_COLON>
62. <_ARG_NAME> → eps
63. <_ARG_NAME_COLON> → <ARG_VAL>
64. <_ARG_NAME_COLON> → IDENTIF
65. <ARG_VAL> → INT
66. <ARG_VAL> → FLOAT
67. <ARG_VAL> → STRING
68. <ARG_VAL> → KEYWORD_NIL

6.4. LL-tabulka

	END_OF_FILE	KEYWORD_FUNC	IDENTIF	L_BRACKET	R_BRACKET	ARROW	EPSILON	COMA	UNDERSCORE	COLON	L_CRLY_BRACKET	R_CRLY_BRACKET	ASSIGN	EXPRESSION	KEYWORD_INT	KEYWORD_INT_NIL	KEYWORD_DOUBLE	KEYWORD_DOUBLE_NIL	KEYWORD_STRING	KEYWORD_STRING_NIL	KEYWORD_BOOL	KEYWORD_BOOL_NIL	KEYWORD_RETURN	KEYWORD_VAR	KEYWORD_LET	KEYWORD_IF	KEYWORD_ELSE	KEYWORD_WHILE	INT	FLOAT	STRING	KEYWORD_NIL	$
<PROGRAM>	3	2	1																				1	1	1	1		1					
<FUNCTION_DEFINITION>		4																															
<FUNCTION_HEAD>		5																															
<_FUNC_IDENTIF_LBRACKET_ARGLIST_RBRACKET>						6	7																										
<PARAM_LIST>			8				9		8																								
<COMMA_PARAM>							11	10																									
<PARAM>			12						13																								
<_PARAM_NAME>			14						15																								
<SCOPE_BODY>											16																						
<COMMAND_SEQUENCE>			17				18																17	17	17	17		17					
<COMMAND>			23																				19	20	20	21		22					
<_IDENTIF>				26									24 25																				
<DATA_TYPE>															27	28	29	30	31	32	33	34											
<RETURN_COMMAND>																							35										
<_RETURN>							37							36																			
<VARIABLE_DEF>																								38	39								
<_VARLET_IDENTIF>										40			41 42																				
<_VARLET_IDENTIF_COLON_TYPE>							45						43 44																				
<CONDITIONAL_COMMAND>																										46							
<_IF>														48											47								
<_IF_LET_IDENTIF_BODY>							50																				49						
<_IF_LET_IDENTIF_BODY_ELSE>											51														52								
<WHILE_COMMAND>																												53					
<FUNCTION_CALL>			54																														
<ARG_LIST>			56				55																						56	56	56	56	
<COMMA_ARG>							58	57																									
<ARG>			59																										60	60	60	60	
<_ARG_NAME>							62			61																							
<_ARG_NAME_COLON>			64																										63	63	63	63	
<ARG_VAL>																													65	66	67	68	

6.5. Precedenční tabulka

	+	-	*	/	==	!=	<	<=	>	>=	AND	OR	!	NOT	??	id	INT	FLOAT	STRING	BOOL	NIL	(	)	$
+	>	>	<	<	>	>	>	>	>	>	>	>	<	<	>	<	<	<	<	<	<	<	>	>
-	>	>	<	<	>	>	>	>	>	>	>	>	<	<	>	<	<	<	<	<	<	<	>	>
*	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	<	<	<	<	<	<	>	>
/	>	>	>	>	>	>	>	>	>	>	>	>	<	<	>	<	<	<	<	<	<	<	>	>
==	<	<	<	<	-	-	-	-	-	-	>	>	<	<	>	<	<	<	<	<	<	<	>	>
!=	<	<	<	<	-	-	-	-	-	-	>	>	<	<	>	<	<	<	<	<	<	<	>	>
<	<	<	<	<	-	-	-	-	-	-	>	>	<	<	>	<	<	<	<	<	<	<	>	>
<=	<	<	<	<	-	-	-	-	-	-	>	>	<	<	>	<	<	<	<	<	<	<	>	>
>	<	<	<	<	-	-	-	-	-	-	>	>	<	<	>	<	<	<	<	<	<	<	>	>
>=	<	<	<	<	-	-	-	-	-	-	>	>	<	<	>	<	<	<	<	<	<	<	>	>
AND	<	<	<	<	<	<	<	<	<	<	>	>	<	<	<	<	<	<	<	<	<	<	>	>
OR	<	<	<	<	<	<	<	<	<	<	<	>	<	<	<	<	<	<	<	<	<	<	>	>
!	>	>	>	>	>	>	>	>	>	>	>	>	-	<	>	-	-	-	-	-	-	-	>	>
NOT	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	<	<	>	>
??	<	<	<	<	<	<	<	<	<	<	>	>	<	<	<	<	<	<	<	<	<	-	>	>
id	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	-	-	>	>
INT	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	-	-	>	>
FLOAT	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	-	-	>	>
STRING	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	-	-	>	>
BOOL	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	-	-	>	>
NIL	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	-	-	-	-	-	-	-	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	-
)	>	>	>	>	>	>	>	>	>	>	>	>	-	>	-	-	-	-	-	-	-	-	>	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	-	-

6.6. Gramatika pro výrazy

1. <E> → <E> ADD <E>
2. <E> → <E> SUB <E>
3. <E> → <E> MUL <E>
4. <E> → <E> DIV <E>
5. <E> → L_BRACKET <E> R_BRACKET
6. <E> → id
7. <E> → FLOAT
8. <E> → INT
9. <E> → STRING
10. <E> → KEYWORD_NIL
11. <E> → <E> EQUAL <E>
12. <E> → <E> NOT_EQUAL <E>
13. <E> → <E> LESSER <E>
14. <E> → <E> LESSER_EQUAL <E>
15. <E> → <E> GREATER <E>
16. <E> → <E> TGREATER_EQUAL <E>
17. <E> → <E> AND <E>
18. <E> → <E> OR <E>
19. <E> → NOT <E>
20. <E> → BOOL
21. <E> → <E> NIL_COALESCING <E>
22. <E> → <E> EXCL_MARK