VYSOKÉ UČENÍ TECHNICKÉ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

IMPLEMENTACE PŘEKLADAČE PRO IMPERATIVNÍ JAZYK IFJ24

TÝM: 253206

VEDOUCÍ: MARTIN ZŮBEK

VARIANTA: TRP-IZP

MARTIN ZŮBEK	253206	25 %
OTAKAR KOČÍ	XKOCIO00	25 %
KRYŠTOF VALENTA	XVALENK00	25 %
MAREK TENORA	XTENOR02	25 %

Obsah

Úvod

Tento dokument představuje dokumentaci k společnému projektu z předmětů IFJ a IAL. Je zde popsáno rozdělení práce, stručný popis členění implementačního řešení včetně názvu souborů a jednotlivé diagramy k potřebě implementace. Hlavní součástí je samotný popis návrhu a implementace.

Rozdělení práce a postup vývoje

Práce na projektu byla rozdělena následujícím způsobem.

- **Marek Tenora**
  - Návrh automatu pro lexikální analýzu
  - Lexikální analyzátor
  - Testování
  - Část sémantické analýzy
- **Martin Zůbek**
  - Návrh automatu pro lexikální analýzu
  - Návrh LL Gramatiky a pravidel pro výrazy
  - Precedenční syntaktická analýza
  - Sémantická analýza – výrazy
- **Otakar Kočí**
  - Návrh automatu pro lexikální analýzu
  - Návrh LL Gramatiky
  - Syntaktická analýza – rekurzivní sestup
  - Testování
  - Sémantická analýza
- **Kryštof Valenta**
  - Návrh automatu pro lexikální analýzu
  - Generování výsledného kódu
  - Testování

Vývoj jsme započali společným návrhem automatu pro lexikální analyzátor, který hned poté Marek Tenora implementoval. Následně Otakar Kočí a Martin Zůbek společně navrhli gramatická pravidla a implementovali syntaktickou analýzu. Nakonec se souběžně vyvíjela sémantická analýza a generátor kódu.

Struktura projektu

Projekt je členěn do následujících modulů.

- `first_phase.c`
- `parser.c`
- `token_buffer.c`
- `precedence.c`
- `precedence_stack.c`
- `precedence_tree.c`
- `scanner.c`
- `symtable.c`
- `semantic.c`
- `semantic_list.c`
- `gen_handler.c`
- `generate.c`

Modul `scanner.c` obsahuje implementaci lexikálního analyzátoru. Moduly `first_phase.c` a `parser.c` obsahují syntaktický analyzátor implementovaný pomocí rekurzivního sestupu. V `token_buffer.c` je implementován buffer pro ukládání tokenů pro jejich pozdější zpracování. V modulu `precedence.c` je implementována metoda precedenční analýzy pro kontrolu syntaxe výrazů. `precedence_stack.c` a `precedence_tree.c` obsahují podpůrné metody pro precedenční analýzu. Modul `symtable.c` obsahuje implementaci tabulky symbolů implementovanou pomocí zásobníku a TRP-IZP. V modulu `semantic.c` jsou implementovány různé podpůrné metody sémantického analyzátoru a modul `semantic_list.c` obsahuje implementaci pomocného lineárního seznamu pro kontrolu sémantiky výrazů. Moduly `gen_handler.c` a `generate.c` obsahují generátor kódu. V modulu `generate.c` se nacházejí pouze pomocné funkce pro výpisy, zatímco modul `gen_handler.c` obsahuje rutiny pro vytváření úseků mezikódu. V souboru `return_values.h` jsou nadefinovány návratové kódy aplikace.

Implementace a návrh jednotlivých částí

Obecný popis návrhu a implementace

Náš překladač je dvouprůchodový. Pro syntaktickou analýzu jsme zvolili kombinaci rekurzivního sestupu a precedenční analýzy pro výrazy. Sémantické akce a rutiny generátoru kódu provádíme přímo voláními odpovídajících funkcí ze syntaktické analýzy. Výjimkou jsou výrazy pro jejichž zpracování využíváme v celém překladači reprezentaci pomocí stromových struktur. Tabulka symbolů je sestavena ze zásobníku a tabulky rozptýlených položek s implicitním zřetězením prvků. V prvním průchodu naplňujeme tabulku symbolů signaturami funkcí a odkládáme tokeny získané lexikálním analyzátorem do bufferu tokenů pro pozdější zpracování v druhém průchodu. Navíc kontrolujeme existenci return příkazu v nevoidových funkcích. Buffer tokenů je smazán až na konci překladu. Vestavěné funkce jsou v překladači reprezentovány stejným způsobem jako funkce uživatelské. V našem překladači jsme neimplementovali žádná rozšíření, přidali jsme ale podporu pro volání funkcí i v příkazech return a u inicializace proměnné. Veškeré detekované chyby jsou propagovány návratovými hodnotami funkcí a pomocí globálních proměnných do funkce `main`, kde jsou vraceny návratovými kódy aplikace. I při chybách je veškerá alokovaná paměť uvolňována.

Datové struktury, enumerace a globální proměnné

Datové struktury

T_TOKEN
- TOKEN_TYPE -> type
- char -> *lexeme
- int -> line
- int -> length
- union -> value
  - int -> int_val
  - float -> float_val
  - char -> *str_val

T_TOKEN_BUFFER_NODE
- T_TOKEN_BUFFER_NODE -> *next
- T_TOKEN_BUFFER_NODE -> *prev

T_TOKEN_BUFFER
- T_TOKEN_BUFFER_NODE -> *head
- T_TOKEN_BUFFER_NODE -> *tail
- T_TOKEN_BUFFER_NODE -> *curr
- T_TOKEN -> *dummy_eof_token

T_STACK
- unsigned int -> count_items
- T_STACK_ITEM -> top

T_LIST_ELEMENT
- T_LIST_ELEMENT -> *next

T_HASHTABLE
- T_SYMBOL table[]
- int count

T_SCOPE
- T_HASHTABLE -> *ht
- T_SCOPE -> *parent
- int -> fc_defined_id

T_SYM_TABLE
- T_SCOPE -> *top
- int -> label_cnt
- int -> var_id_cnt
- int -> fc_derived_cnt
- char -> *current_fn_name

T_TREE_NODE
- T_TREE_NODE -> *left
- T_TREE_NODE -> *right
- T_TOKEN -> *token
- bool -> convert_to_float
- bool -> convert_to_int
- RESULT_TYPE result_type

T_STACK_ITEM
- T_STACK_ITEM -> *prev

- T_LIST_ELEMENT -> *prev
- T_TREE_NODE -> node
- LITERAL_TYPE -> literal_type

T_SYMBOL_DATA

- struct -> var
  - bool -> is_const
  - bool -> modified
  - bool -> used
  - bool -> const_expr
  - float -> float_value
  - VAR_TYPE -> type
  - int -> id
- struct -> func
  - VAR_TYPE -> return_type
  - int -> argc
  - T_PARAM -> *argv

T_PARAM

- char -> *name
- VAR_TYPE -> type

- T_TREE_NODE -> node
- STACK_ITEM_TYPE -> type
- T_TOKEN -> token

T_LIST

- T_LIST_ELEMENT -> first
- T_LIST_ELEMENT -> last
- T_LIST_ELEMENT -> active
- int size

T_SYMBOL

- Char -> *name
- SYMBOL_TYPE -> type
- T_SYMBOL_DATA -> data
- T_SYMBOL -> *next
- bool -> occupied
- bool -> deleted

FN_CALL

- char -> *name
- VAR_TYPE -> ret_type
- int argc
- T_TOKEN **argv

Enumerace

- RESULT_TYPE
- TOKEN_TYPE
- STACK_ITEM_TYPE
- RET_VAL
- LITERAL_TYPE
- OPERATOR_TYPE_OF_RULE
- SYMBOL_TYPE
- VAR_TYPE
- PRECEDENCE
- OPERATOR_INDEX
- TYPE_END

Globální proměnné

- T_SYMTABLE -> *ST
- RET_VAL -> error_flag_fp
- RET_VAL -> error_flag
- bool -> needs last_token
- int -> line_number
- int -> ord_counter
- int -> strcmp_counter
- int -> substr_counter

Lexikální analýza

Lexikální analyzátor je navržen jako deterministický konečný automat (DFA), který zpracovává jednotlivé znaky ze vstupního zdrojového souboru. Automat je implementován na základě diagramu (na konci dokumentu) a pokrývá všechny definované lexikální prvky jazyka IFJ24, jako jsou klíčová slova, identifikátory, operátory, laterály a další symboly. Automat v průběhu sleduje aktuální řádek pro lepší zpětnou vazbu a některé lexikální kontroly. Analyzátor komunikuje s okolním světem pouze přes jeden ovladač a tím je funkce `get_token(T_TOKEN *token)` která při opakovaném volání čerpá `stdin` a převádí ho na tokeny. Pro lepší manipulaci byl vytvořen ještě modul token bufferu, který umožňuje procházet tokeny v obou směrech.

Syntaktická analýza

Syntaktická analýza shora dolů

Jak již bylo zmíněno, naše implementace je dvouprůchodová. V prvním průchodu dochází ke kontrole prologu a hlaviček funkcí. Těla funkcí jsou ignorována a pouze se počítají odpovídající složené závorky, pro zjištění konce funkce. Hlavním cílem prvního průchodu je naplnit tabulku symbolů validními uživatelskými funkcemi. Veškeré čtené tokeny jsou ukládány do bufferu tokenů (`TOKEN_BUFFER`) pro pozdější zpracování druhým průchodem.

Druhý průchod je plnohodnotným syntaxí řízeným překladem. Je implementován rekurzivním sestupem, který využívá tokenů z pomocného bufferu pro aplikaci gramatických pravidel. Samotný lexikální analyzátor již není v druhém průchodu využíván. Přechod do precedenční syntaktické analýzy je dán zanořením rekurze do konkrétních pravidel, která tento přechod umožňují. V případě cyklů a podmínek se přepíná do precedenční analýzy vždy po otevírací závorce podmínky. Pro situace, kdy je výraz na pravé straně (přiřazení, definice proměnné, návrat z funkce) je gramatika doplněna o pravidla tak, aby zůstala LL1 gramatikou a aby se do precedenční analýzy přešlo pouze ve chvíli, kdy je jisté, že na pravé straně se nachází výraz, a nikoliv volání uživatelské funkce. Kontroly syntaxe jsou v druhém průchodu prokládány sémantickými a kód generujícími akcemi, o nich více v následujících sekcích.

Moduly `parser.c` a `first_phase.c` využívají globálních proměnných `error_flag` a `error_flag_fp` pro jednodušší a přehlednější propagaci chyb. V modulu `first_phase.c` je navíc globální proměnná `needs_last_token`, která indikuje potřebu využití posledního tokenu namísto žádosti lexikálního analyzátoru o další. Tento přístup se v době implementace jevil jako nejpřímočařejší.

Precedenční syntaktická analýza zdola nahoru

Precedenční syntaktická analýza je implementována jako samostatný modul pro syntaktickou analýzu shora dolů pro syntaktickou analýzu výrazů. Samotný algoritmus je byl implementován v souladu s vyučovaným postupem v předmětu IFJ.

Implementace precedenční analýzy je založena na vytvořené gramatice a precedenční tabulce udávající precedence jednotlivých operátorů. Gramatiku a precedenční tabulku lze nalézt v Přílohách.

Pro podporu precedenční analýzy, byly implementované pomocné datové struktury zásobníku a stromu, které lze najít v kapitole Datové struktury.

Precedenční tabulka je implementována pomocí dvojrozměrného pole, kde konkrétní precedence, mezi dvěma operátory je reprezentována pomocí enumerace `PRECEDENCE.` Následně pomocí enumerace `OPERATOR_INDEX` a typu příchozího tokenu, je pomocí funkce `get_precedence` získaná precedence operátorů.

Hlavní částí je funkce `precedence_syntax_main,` která implementuje samotný algoritmus. Funkce vrací kód, který indikuje výsledek samotné analýzy. Posuv (shift) je implementován, jako vložení prvku na zásobník, s enumerací `SHIFT` z `STACK_ITEM_TYPE.` Redukce (reduce) je implementována, pomoci funkcí `count_reduce,` `can_reduce` pro ověření, zdali je gramatické pravidlo možné aplikovat a následně funkcí `reduce,` která konkrétní gramatické pravidlo provede.

Sémantická analýza

Sémantické akce jsou stejně jako volání generátoru kódu přímo vepsány mezi syntaktické kontroly v modulu `parser.c`. Tyto sémantické akce využívají funkcí tabulky symbolů (více v o tabulce symbolů) a funkcí v modulu `semantic.c`. Potřebný kontext je udržován v tabulce symbolů, případně jsou navazujícím funkcím předávány dodatečné argumenty. Sémantické akce zajišťují přidávání proměnných do tabulky symbolů a správné zanořování a vynořování rozsahů ve funkcích a blocích cyklů a podmínek pomocí volání odpovídajících funkcí tabulky symbolů. Sémantická analýza také modulu generování kódu zprostředkovává unikátní návěští příkazů řízení toku, unikátní přípony názvů proměnných a unikátní přípony pro obalující kontroly definic proměnných zanořených v cyklech a podmínkách. Zjištěné chyby jsou propagovány z pomocných funkcí nahoru, nebo je využita globální proměnná `error_flag`.

V sémantické analýze kontrolujeme tyto potenciální chyby:

- Volaná funkce, či použitá proměnná nebyla definována. V případě proměnné nebyla definována v blocích viditelných z aktuálního zanoření.
- Volaná funkce nebyla zavolána korektně.
    - Špatný počet, nebo typ argumentů.
    - Zahození návratové hodnoty ne-void funkce.
    - Nekompatibilní datový typ proměnné, do které se výsledek funkce přiřazuje.
- Pokus o redefinici funkce, nebo proměnné. Přiřazení do nemodifikovatelné proměnné.
- Příkaz návratu z funkce neobsahuje výraz v situaci, kdy by jej obsahovat měl. Nebo výraz přebývá u void funkce.
- Typová nekompatibilita ve výrazech (aritmetika, relace a řetězce). Nekompatibilní typ při přiřazení, nebo návratu z funkce. O kontrolách kompatibility typů ve výrazech více v následujícím odstavci.
- Nemožnost odvození typu, dle pravé strany přiřazení u definice proměnné.
- Nevyužití proměnné v jejím rozsahu platnosti. Neprovedená modifikace u modifikovatelných proměnných.
- Není definována funkce main, případně má špatnou signaturu.
- Mezi parametry funkcí jsou parametry se stejným jménem. Tato chyba je kontrolována v prvním průchodu syntaktické analýzy.

Sémantická analýza výrazů je implementována, pomoci funkce `check_expression`. Funkce přijímá pomocí argumentů strom, sestavený precedenční syntaktickou analýzou. Tento strom je následně pomocí průchodu postorder, převeden do postfixové notace, jenž je uložená v datové struktuře dvojvázaného spojového seznamu `T_LIST`, ukázaného v kapitole Datové struktury. Pomocí funkce `set_types` je přiřazen typ literálů, potažmo neliterálů, který definuje enumerace `LITERAL_TYPE`. Následně pomocí iterace přes sekvenci selekcí, a pomocí uřčených typů, je výraz sémanticky analyzován. Pokud má dojít ke konverzi typu operandů, dle specifikace v zadání, je pro budoucí generování kódu nastaven příznak, pomocí bool proměnné `convert_to_float` a `convert_to_int`, v elementu stromu. Postupně, během iterace, je v podstatě simulováno vyčíslení v postfixové notaci, pokud je výraz sémantický správný, zůstane po konci algoritmu, jen jeden element v listu, obsahující kořen stromu. Pomocí proměnné struktury stromu `result_type`, určenou

enumerací RESULT_TYPE, je nastaven finální datový typ výrazu pro další sémantické akce. Funkce následně vrací kód indikující úspěch. V případě chyby vrací kód konkrétní chyby tak, aby se mohla chyba propagovat, jak je popsáno v úvodní sekci o návrhu a implementaci.

Tabulka symbolů

Tabulka symbolů (T_SYM_TABLE) je implementována jako zásobník dílčích tabulek symbolů pro jednotlivé rozsahy (T_SCOPE). V každé dílčí tabulce se nachází tabulka rozptýlených položek s implicitním zřetězením prvků (dále jen TRP-IZP, dat. struktura T_HASHTABLE). Implementační limit na maximální počet položek (velikost pole) v jedné TRP-IZP byl nastaven na prvočíslo 1999. Využíváme implementaci pomocí dvou hashovacích funkcí (Brentova metoda). První hashovací funkce je využita pro nalezení odpovídajícího indexu a druhá pro nalezení alternativního indexu, pokud je daná buňka již využívána. Jako násobitele v těchto hashovacích funkcích jsme zvolili prvočísla 17 a 31. Prvočísla pro velikost pole a hashovací funkce jsme takto zvolili, aby bylo možné navštívit všechny buňky jedné TRP-IZP. Potřebné informace jsme čerpali z přednášek předmětu IAL.

Každý symbol je v tabulce symbolů uložen v datové struktuře T_SYMBOL, kde má své jméno a další údaje. Rozlišení, zdali se jedná o funkci nebo proměnnou zajišťuje enumerace SYMBOL_TYPE. Konkrétní informace o symbolu jsou uloženy v T_SYMBOL_DATA, která je sjednocena pro funkce i proměnné. Parametry funkcí jsou uchovány v T_PARAM. Datové typy rozlišuje VAR_TYPE.

Rozhraní tabulky symbolů se skládá ze sady funkcí pro její manipulaci. Nejvýznamnějšími jsou funkce pro přidání a odstranění rozsahu (zanoření), funkce pro přidání symbolu (funkce, nebo proměnná) a funkce pro vyhledání symbolu. Součástí jsou i pomocné funkce pro její mazání, pro získání id proměnné, pro kontrolu, jestli byly všechny proměnné v rozsahu využity atp. Tyto funkce jsou převážně volány při sémantických akcích a při generování kódu.

Odkaz na tabulku symbolů je pro jednodušší sdílení mezi všemi moduly uložen v globální proměnné ST. Naše implementace nevyužívá funkce pro mazání záznamu z tabulky, jelikož je po opuštění daného rozsahu daná tabulka smazána celá.

Generátor kódu

Přehled

Generativní část kompilátoru IFJ24 je zodpovědná za převod interní reprezentace programu na sekvenci instrukcí, které mohou být vykonány interpretem IFJ24. Tato část zahrnuje soubory generate.c a gen_handler.c, kdy generate.c soubor slouží jako "instrukční sada", zabalená do pomocných funkcí, které jsou volány z dalších částí programu. Soubor gen_handler.c se dále zaměřuje na zpracování vstupů z předchozí etapy překladače a generování odpovídajících instrukcí pro interpret.

Kromě vyhodnocování výrazů se generátoru nepředávají žádná další data ve formě AST. Místo toho jsou potřebné informace předávány přímo jako argumenty generativních funkcí. Hlavní funkce této komponenty zahrnují generování hlaviček programů, zpracování volání funkcí, správu výrazů i implementaci vestavěných funkcí a řídicích struktur.

Při návrhu a implementaci generátoru kódu jsme zvažovali více možných přístupů, přičemž finální řešení bylo zvoleno s ohledem na modularitu a čitelnost. Implementace započala vytvořením instrukčního souboru, který jednotlivé instrukce zapouzdřuje do funkcí s argumenty, které se do instrukce následně plní prostřednictvím placeholderů. To nám umožnilo nejen následnou lepší čitelnost kódu, ale také implementaci specifických instrukcí pro např. dané datové typy, které tento přístup vyžadují.

Následně byl implementován soubor `gen_handler.c`, který volá funkce ze souboru generate.c a obsahuje funkce vytvářející sekvence instrukcí pro interpret tak, aby byla zachována funkčnost vstupního programu. Soubor se logicky dělí na kategorie dle typu instrukcí, které funkce vykonávají jako:

Funkce potřebné pro interpret

Tyto funkce generují sekce, které jsou pro interpret a správné vykonání zbytku instrukcí klíčové. Generují tak hlavičku souboru, včetně deklarace globálních pomocných proměnných, direktivu `.IFJcode24`, funkci main a ukončující volaní.

Vyhodnocení výrazů

Výrazy vyhodnocuje jedna funkce, která přijímá `AST` s operátory a operandy. Výraz se vyhodnocuje v zásobníku tak, že dle typu dat v listu funkce vloží na vrchol zásobníku hodnotu, nebo nad zásobníkem zavolá danou operaci potřebnou pro vyhodnocení daného výrazu. Výsledek zůstává na vrcholku zásobníku a za pomoci dalších funkcí se přesouvá do správné destinace.

Předávání výsledků, mezivýsledků, parametrů a argumentů

Téměř ve všech situacích řešíme předávání výsledků, argumentů atp. vložením na zásobník, ze kterého jsou následně vyjmuty dle potřeby.

Cykly a podmínky

Pro zpracování cyklů a podmínek je nejprve vyhodnocen výraz, který určuje podmínky vstupu do podmínky nebo smyčky a její případné opakování. Tento výraz je vyhodnocen a následně porovnán, čímž je podmíněn vstup do podmínky, smyčky nebo její opakování. V případech, kde dochází k redefinici proměnných, je zajištěno, že po prvním průchodu jsou instrukce těchto definicí přeskočeny.

Funkce

Při volání funkcí se případné argumenty vyjmou ze zásobníku, a jsou tak funkci zpřístupněny. Funkce jsou jedinou částí generovaného kódu, která využívá rámců.

Kromě výše uvedeného generátor zahrnuje i další funkce potřebné pro vytvoření kódu, který je funkčně ekvivalentní vstupního programu a plně interpretovatelný interpretem IFJ24.

Diagramy a tabulky

LL1 Gramatika

Velkými písmeny jsou značeny neterminály. Malými písmeny a symboly terminály. Pokud je v tabulce některý z neterminálů vícekrát na levé straně pravidla, tak je tím značena alternativa. Neterminál EXPRESSION, značí části zpracovávané precedenční syntaktickou analýzou.

Pravidlo		
1. START	->	PROLOG FN_DEF_NEXT END
2. END	->	eof_token
3. PROLOG	->	const ifj = import (string) ;
4. FN_DEF_NEXT	->	epsilon
5. FN_DEF_NEXT	->	FN_DEF FN_DEF_NEXT
6. FN_DEF	->	pub fn identifier (PARAMS) FN_DEF_REMAINING
7. FN_DEF_REMAINING	->	TYPE { CODE_BLOCK_NEXT }
8. FN_DEF_REMAINING	->	void { CODE_BLOCK_NEXT }
9. PARAM	->	identifier : TYPE
10. PARAMS	->	epsilon
11. PARAMS	->	PARAM PARAM_NEXT
12. PARAM_NEXT	->	epsilon
13. PARAM_NEXT	->	, PARAM_AFTER_COMMA
14. PARAM_AFTER_COMMA	->	epsilon
15. PARAM_AFTER_COMMA	->	PARAM PARAM_NEXT
16. TYPE	->	type_int
17. TYPE	->	type_float
18. TYPE	->	type_string
19. TYPE	->	type_int_null
20. TYPE	->	type_float_null
21. TYPE	->	type_string_null
22. CODE_BLOCK_NEXT	->	epsilon
23. CODE_BLOCK_NEXT	->	CODE_BLOCK CODE_BLOCK_NEXT
24. CODE_BLOCK	->	VAR_DEF
25. CODE_BLOCK	->	IF_STATEMENT
26. CODE_BLOCK	->	WHILE_STATEMENT
27. CODE_BLOCK	->	RETURN
28. CODE_BLOCK	->	ASSIGN_EXPR_OR_FN_CALL
29. CODE_BLOCK	->	BUILT_IN_VOID_FN_CALL
30. CODE_BLOCK	->	ASSIGN_DISCARD_EXPR_OR_FN_CALL
31. VAR_DEF	->	const identifier VAR_DEF_AFTER_ID
32. VAR_DEF	->	var identifier VAR_DEF_AFTER_ID
33. VAR_DEF_AFTER_ID	->	: TYPE = ASSIGN
34. VAR_DEF_AFTER_ID	->	= ASSIGN

35. IF_STATEMENT	->	if (EXPRESSION) IF_STATEMENT_REMAINING
36. IF_STATEMENT_REMAINING	->	{ CODE_BLOCK_NEXT } else { CODE_BLOCK_NEXT }
37. IF_STATEMENT_REMANING	->	| identifier | { CODE_BLOCK_NEXT } else { CODE_BLOCK_NEXT }
38. WHILE_STATEMENT	->	while (EXPRESSION) WHILE_STATEMENT_REMAINING
39. WHILE_STATEMENT_REMAINING	->	{ CODE_BLOCK_NEXT }
40. WHILE_STATEMENT_REMAINING	->	| identifier | { CODE_BLOCK_NEXT }
41. RETURN	->	return RETURN_REMAINING
42. RETURN_REMAINING	->	;
43. RETURN_REMAINING	->	ASSIGN
44. BUILT_IN_VOID_FN_CALL	->	ifj . identifier (ARGUMENTS) ;
45. ASSIGN_EXPR_OR_FN_CALL	->	identifier ID_START
46. ASSIGN_DISCARD_EXPR_OR_FN_CALL	->	discard_identifier = ASSIGN
47. ID_START	->	= ASSIGN
48. ID_START	->	FUNCTION_ARGUMENTS
49. ASSIGN	->	EXPRESSION ;
50. ASSIGN	->	identifier ID_ASSIGN
51. ASSIGN	->	ifj . identifier (ARGUMENTS) ;
52. ID_ASSIGN	->	EXPRESSION ;
53. ID_ASSIGN	->	;
54. ID_ASSIGN	->	FUNCTION_ARGUMENTS
55. FUNCTION_ARGUMENTS	->	(ARGUMENTS) ;
56. ARGUMENTS	->	epsilon
57. ARGUMENTS	->	ARGUMENT ARGUMENT_NEXT
58. ARGUMENT_NEXT	->	epsilon
59. ARGUMENT_NEXT	->	, ARGUMENT_AFTER_COMMA
60. ARGUMENT_AFTER_COMMA	->	epsilon
61. ARGUMENT_AFTER_COMMA	->	ARGUMENT ARGUMENT_NEXT
62. ARGUMENT	->	identifier
63. ARGUMENT	->	int
64. ARGUMENT	->	float
65. ARGUMENT	->	string
66. ARGUMENT	->	null

LL tabulka

	eof_token	const	ifj	=	import	(	string	)	;	pub	fn	identifier	{	}	void	:	,
START		1.															
END	2.																
PROLOG		3.															
FN_DEF_NEXT	4.									5.							
FN_DEF										6.							
FN_DEF_REMAINING															8.		
PARAM												9.					
PARAMS								10.				11.					
PARAM_NEXT								12.									13.
PARAM_AFTER_COMMA								14.				15.					
TYPE																	
CODE_BLOCK_NEXT		23.	23.									23.		22.			
CODE_BLOCK		24.	29.									28.					
VAR_DEF		31.															
VAR_DEF_AFTER_ID				34.												33.	
IF_STATEMENT																	
IF_STATEMENT_REMAING													36.				
WHILE_STATEMENT																	
WHILE_STATEMENT_REMAINING													39.				
RETURN																	
RETURN_REMAINING			43.						42.			43.					
BULT_IN_VOID_FN_CALL			44.														
ASSIGN_EXPR_OR_FN_CALL												45.					
ASSIGN_DISCARD_EXPR_OR_FN_CALL																	
ID_START				47.		48.											
ASSIGN			51.									50.					
ID_ASSIGN						54.			53.								
FUNCTION_ARGUMENTS						**55.**											
ARGUMENTS							**57.**	56.				**57.**					
ARGUMENT_NEXT								58.									**59.**
ARGUMENT_AFTER_COMMA							**61.**	60.				**61.**					
ARGUMENT							**65.**					**62.**					

,	type_int	type_float	type_string	type_int_null	type_float_null	type_string_null	var	if	EXPRESSION	else	|	while	return	.	discard_identifier	int	float	null
	7.	7.	7.	7.	7.	7.												
13.																		
	16.	17.	18.	19.	20.	21.												
							23.	23.				23.	23		23.			
							24.	25.				26.	27		30.			
							32.											
								35.										
											37.							
												38.						
											40.							
													41.					
									47.									
															46.			
									49.									
									52.									
																57.	57.	57.
59.																		
																61.	61.	61.
																63.	64.	66.

Gramatika pro výrazy

```
S -> R                  E -> string_literal
S -> E                  E -> null
E -> E + E              R -> E < E
E -> E – E              R -> E > E
E -> E * E              R -> E <= E
E -> E / E              R -> E >= E
E -> (E)                R -> E == E
E -> id                 R -> E != E
E -> int_literal        R -> (R)
E -> float_literal
```

Precedenční tabulka

Id v tabulce označuje jak identifikátor, tak jakýkoliv literál, který gramatika v předešlé kapitole povoluje.

	id	+	-	*	/	<	>	<=	>=	==	!=	(	)	$
id		>	>	>	>	>	>	>	>	>	>		>	>
+	<	>	>	<	<	>	>	>	>	>	>	<	>	>
-	<	>	>	<	<	>	>	>	>	>	>	<	>	>
*	<	>	>	>	>	>	>	>	>	>	>	<	>	>
/	<	>	>	>	>	>	>	>	>	>	>	<	>	>
<	<	<	<	<	<							<	>	>
>	<	<	<	<	<							<	>	>
<=	<	<	<	<	<							<	>	>
>=	<	<	<	<	<							<	>	>
==	<	<	<	<	<							<	>	>
!=	<	<	<	<	<							<	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	
)		>	>	>	>	>	>	>	>	>	>		>	>
$	<	<	<	<	<	<	<	<	<	<	<	<		

Konečný automat pro lexikální analýzu