Vysoké učení technické v Brně
Fakulta informačních technologií

Implementace překladače imperativního jazyka IFJ19

Tým 086, varianta II

Hubík Antonín (V.)	xhubik03	25 %
Nosková Daša	xnosko05	25 %
Holas David	xholas11	25 %
Mušková Kateřina	xmusko00	25 %

Implementovaná rozšíření: `BASE`

Obsah

Přehled **3**
Shrnutí . . . 3
Rozdělení práce . . . 3
Soubory projektu . . . 3

1 Lexikální analýza **4**
1.1 Graf konečného automatu . . . 4
1.2 Implementace . . . 7
1.2.1 Funkce rozhraní . . . 7
1.2.2 Lexikální analyzátor . . . 7
1.2.3 Token . . . 7
1.2.4 Odsazení ve vstupním kódu . . . 7
1.2.5 Průběžné ukládání vstupních znaků . . . 8

2 Syntaktická a sémantická analýza **8**
2.1 LL gramatika . . . 8
2.2 LL tabulka . . . 9
2.3 Precedenční tabulka . . . 10
2.4 Analýza kostry programu . . . 11
2.4.1 Činnost parseru . . . 11
2.4.2 Řešení redundance kódu . . . 11
2.4.3 Poznámky k implementaci . . . 11
2.5 Pravidla pro analýzu výrazů . . . 12
2.6 Analýza výrazů . . . 12
2.6.1 Činnost modulu . . . 12
2.6.2 Zásobník . . . 12
2.6.3 Typová kontrola . . . 13

3 Generování kódu IFJcode19 **13**
3.1 Funkce rozhraní . . . 13
3.2 Vnitřní řešení . . . 13
3.2.1 Buffer . . . 13
3.2.2 Zpracování funkcí a výrazů . . . 13
3.2.3 Typové kontroly, konverze . . . 13

4 Tabulka symbolů **14**
4.1 Funkce rozhraní . . . 14
4.2 Řešení vybraných problémů . . . 14
4.2.1 Hashovací funkce . . . 14
4.2.2 Tabulka . . . 14
4.2.3 Položka tabulky . . . 14
4.2.4 Atributy proměnné . . . 15
4.2.5 Atributy funkce . . . 15

Seznam obrázků

1 Konečný automat – Operátory, id, indent, komentář 4
2 Konečný automat – Celočíselný a desetinný literál 5
3 Konečný automat – Řetězcový literál, dokumentační řetězec 6

Seznam tabulek

1 LL-tabulka 9
2 Precedenční tabulka 10

Přehled

Shrnutí

Překladač jazyka IFJ19 funguje na principu syntaxí řízeného překladu, přičemž syntaktická analýza kostry programu je realizována metodou rekurzivního sestupu. Tabulka symbolů byla v souladu s variantou zadání implementována jako tabulka rozptýlených položek.

Rozdělení práce

Distribuce práce zejména na počátku projektu vyústila v následující rozdělení úkolů:

- **Hubík Antonín** – návrh a implementace tabulky symbolů, zpracování dokumentace projektu, příprava obhajoby, Makefile
- **Nosková Daša** – zpracování LL tabulky pro syntaktickou analýzu, návrh a implementace syntaktické analýzy kostry programu, sémantická analýza kostry programu
- **Holas David** – zpracování precedenční tabulky a pravidel pro syntaktickou analýzu výrazů, implementace syntaktické a sémantické analýzy výrazů, generování cílového kódu
- **Mušková Kateřina** – návrh konečného automatu pro lexikální analýzu, implementace lexikální analýzy, generování cílového kódu

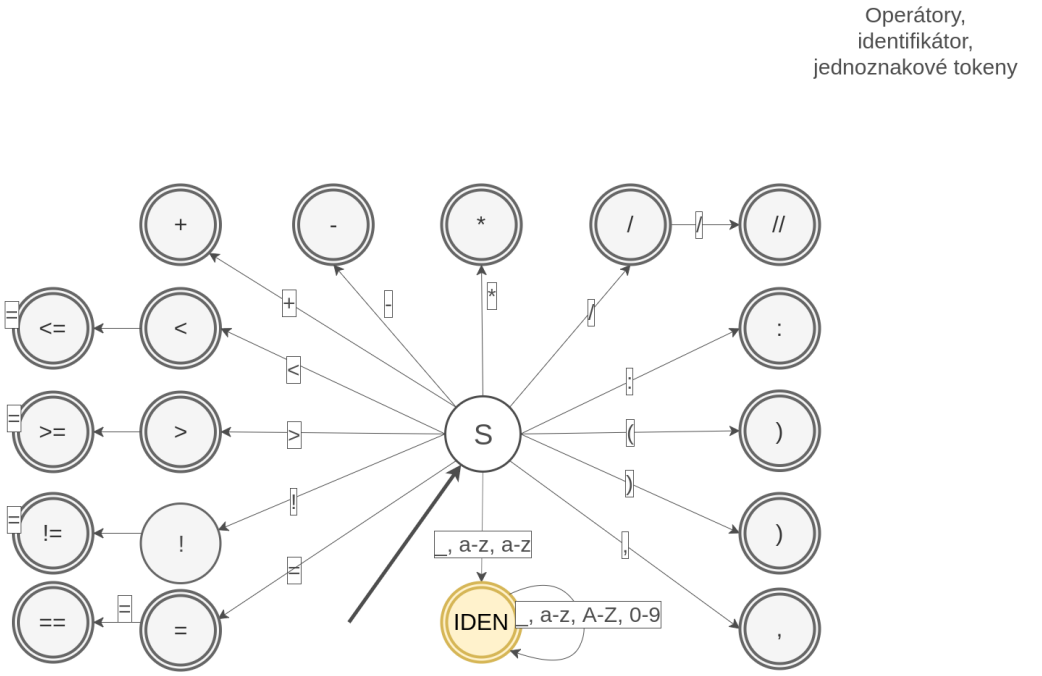
Toto rozdělení práce nezahrnuje implementaci dodatečných pomocných nástrojů a jen implicitně naznačuje autorství jednotlivých modulů programové části. Z tohoto důvodu je autor každého zdrojového souboru vždy uveden v jeho hlavičce. Jestliže některý jiný člen týmu ve větším měřítku soubor editoval, je jeho jméno napsáno v kolonce "úpravy".

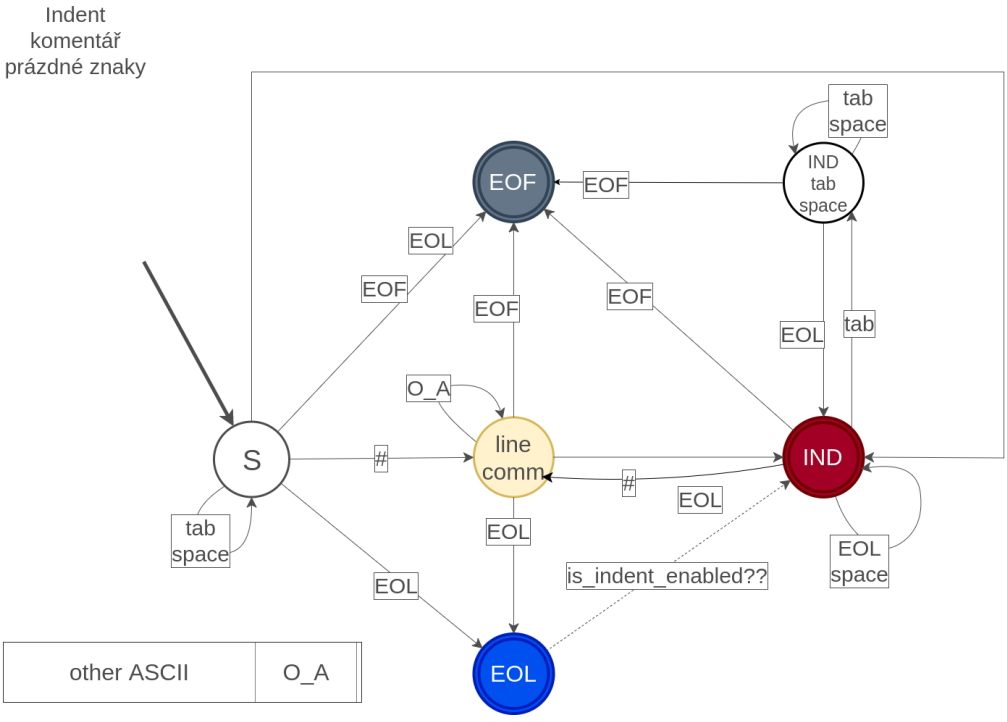
Soubory projektu

Hlavičkové soubory	**C soubory**	**Ostatní**
`dynamic_string.h`	`dynamic_string.c`	`dokumentace.pdf`
`error.h`	`error.c`	`Makefile`
`error.h`	`error.c`	`rozdeleni`
`expressions.h`	`expressions.c`	
`expr_stack.h`	`expr_stack.c`	
`generator.h`	`generator.c`	
`parser.h`	`parser.c`	
`scanner.h`	`scanner.c`	
`stack.h`	`stack.c`	
`symtable.h`	`symtable.c`	
`token.h`		

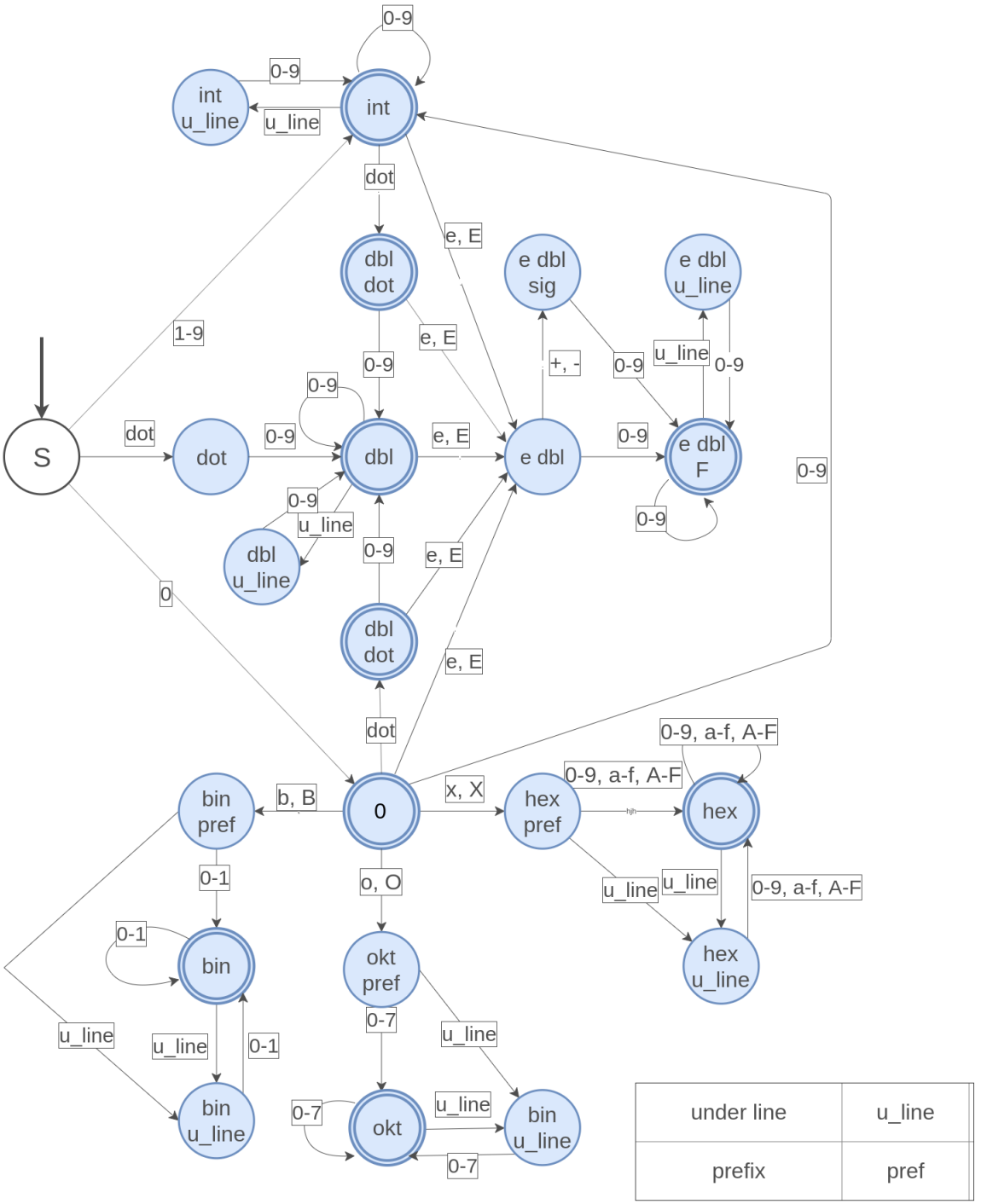
1 Lexikální analýza

1.1 Graf konečného automatu



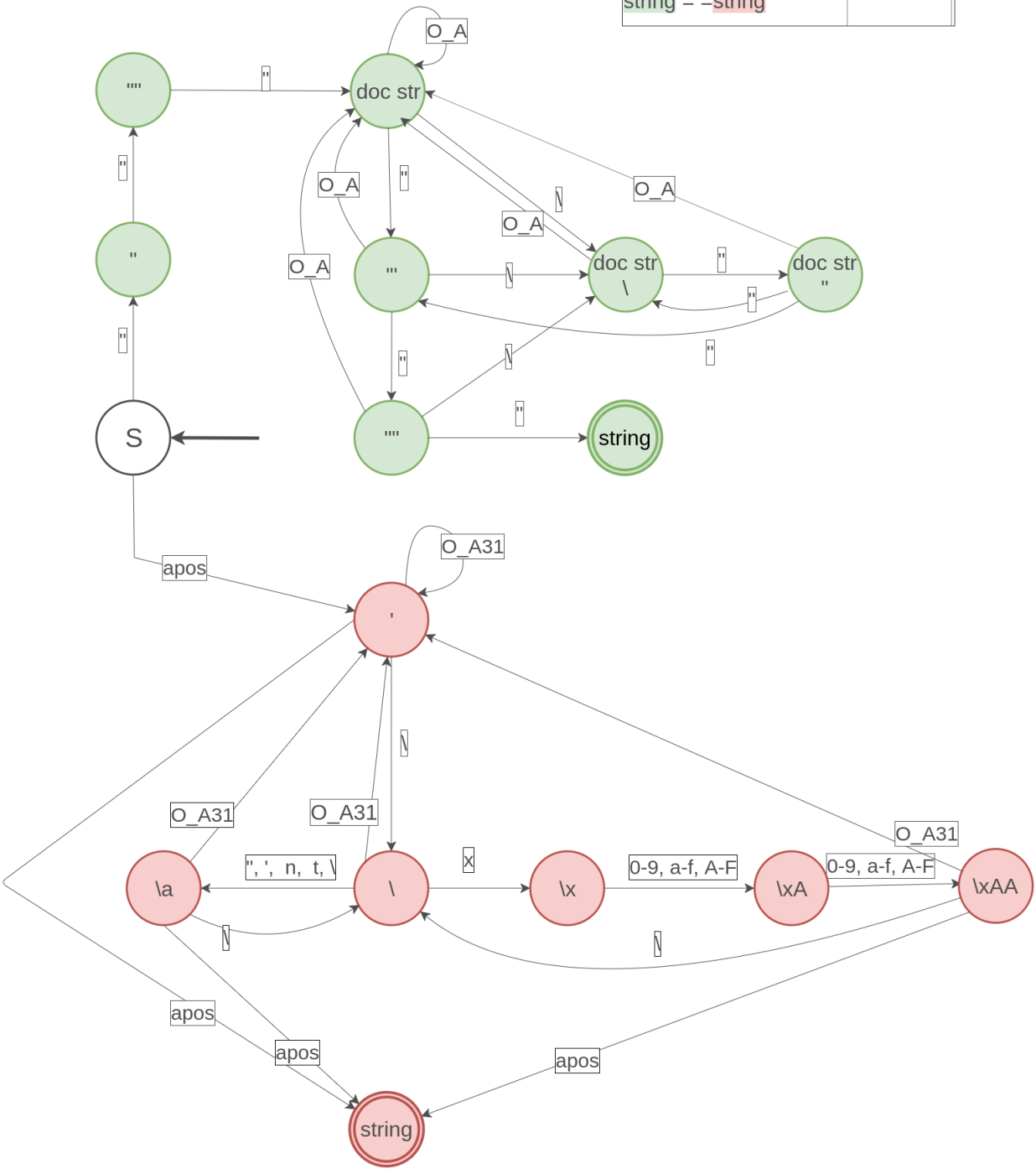


Celočíselný a
desetinný literál



Řetězcový literál,
dokumentační řetězec

ASCII>31	A31
other ASCII>31	O_A31
apostrophe	apos
string = =string	



1.2 Implementace

Konečný automat navržený pro lexikální analýzu byl implementován ve zdrojových souborech `scanner.h` a `scanner.c`

1.2.1 Funkce rozhraní

- `scanner_init()` – alokuje dynamické proměnné a struktury, nastaví inicializační hodnoty
- `get_token()` – vrátí aktuální token implementovaný v `token.h`. V případě neznámého lexému je označen jako `T_LEX_UNKNOWN`,
- `scanner_destroy()` – uvolní alokovanou paměť.

1.2.2 Lexikální analyzátor

Implementace lexikální analýzy vychází z navrženého konečného automatu. Všechny stavy (větve `case` v řidicí konstrukci `switch`) korespondují s diagramem na stránkách 4, 5 a 6. Vstupní řetězec je do LA načítán po jednotlivých znacích. V případě, že se KA pro aktuálně načtený znak ocitne v koncovém stavu, je buď vrácen token, nebo nahlášena chyba v závislosti na stavu a znaku.

1.2.3 Token

Lexikální analyzátor předává informace nezbytné k syntaktické analýze prostřednictvím ukazatele na strukturu `struct token` se složkami

- `lexeme` – obsahuje typ lexému (identifikátor, řetězcový literál, while, double...),
- `value` – je typu union a podle daného lexému se buď dále používá jako typ `string` (pro lexémy řetězcový literál a identifikátor), `long int`, `double`, nebo vůbec.

1.2.4 Odsazení ve vstupním kódu

Tokenu *indent*/*dedent* musí vždy předcházet znak konce řádku. Zde vzniká problém, zda má lexikální analyzátor aktuálnímu lexému přiřadit token *indent*/*dedent*, nebo *EOL*.

Tento problém je vyřešen ukládáním aktuálního odsazení do struktury `e_ind` se složkami:

- `ind` – úroveň odsazení,
- `enable` – příznak povolující nebo zakazující vracet token *indent*. Může být nastaven na 1 pouze ve stavu `EOL` přímo před vrácením tokenu. Hodnoty 0 nabývá ve stavu `INDENT`, jestliže už není možné, aby vracený token byl *indent* nebo *dedent*.

Úrovně odsazení do zásobníku, kde nejspodnější hodnota je 0 a hodnota na vrcholu zásobníku určuje poslední odsazení. V `case INDENT` podle toho nastat tyto situace:

- Nic se neděje. Úroveň odsazení se nemění. LA nevrací žádný token, `enable` je nastavena na 0, přechází se ze stavu `INDENT` zpět na počáteční stav KA (`switch case START`).
- Indent. Úroveň odsazení se zvyšuje. Nová hodnota je přidána na vrchol zásobníku, `enable` je nastavena na 0, LA vrací token *indent*.
- Dedent. Úroveň odsazení se snižuje. Je odstraněna hodnota na vrcholu zásobníku. `enable` je nastavena na 0, LA vrací token *dedent*.

- Multiple dedent. Úroveň odsazení se snižuje. Je odstraněna hodnota na vrcholu zásobníku, `enable` je nastaven na 1, vrátí se token *dedent*, v příštím volání se opět ocitneme v koncovém stavu implementovaném v `case INDENT`
- Chyba. Úroveň odsazení je menší než hodnota na vrcholu zásobníku a současně větší než následující hodnota.

1.2.5 Průběžné ukládání vstupních znaků

Pro lexémy identifikátor a řetězcový literál je nutné načítané znaky průběžně ukládat. K tomuto a dalším účelům slouží datový typ `string`, jehož rozhraní a implementace jsou umístěny ve zdrojových souborech `dynamic_string.h` a `dynamic_string.c`.

Při přechodu do nového stavu se v závislosti na něm může na konec řetězce přidat aktuální znak. Má-li být výsledný řetězec nějakým způsobem modifikován (escape sekvence, hexadecimální zápis, ...), děje se tak ihned, jakmile jsou všechny jeho znaky k dispozici.

2 Syntaktická a sémantická analýza

2.1 LL gramatika

1.	PROG	→	eof								
2.	PROG	→	STAT	PROG							
3.	PROG	→	F_DEF	PROG							
4.	F_DEF	→	def	F_NAME	:	eol	indent	STAT	STAT_LIST	dedent	
5.	F_NAME	→	id	(<param>)							
6.	STAT	→	if	EXPR	:	eol	indent	STAT	STAT_LIST	dedent	IF_ELSE
7.	IF_ELSE	→	else	:	eol	indent	STAT	STAT_LIST	dedent		
8.	STAT	→	while	EXPR	:	eol	indent	STAT	STAT_LIST	dedent	
9.	STAT	→	ASSIGN	eol							
10.	STAT	→	FUNC	eol							
11.	STAT	→	eol	eol							
12.	STAT_LIST	→	STAT	STAT_LIST							
13.	STAT_LIST	→	eps								
14.	STAT	→	return	RETURN_TAIL	eol						
15.	RETURN_TAIL	→	EXPR								
16.	RETURN_TAIL	→	eps								
17.	FUNC	→	inputs	()							
18.	FUNC	→	inputi	()							
19.	FUNC	→	inputf	()							
20.	FUNC	→	print	(PARAM)							
21.	FUNC	→	len	(string)							
22.	FUNC	→	substr	(string, int, int)							
23.	FUNC	→	ord	(string, int)							
24.	FUNC	→	chr	(int)							
25.	ASSIGN	→	id =	FUNC_OR_EX							
26.	FUNC_OR_EX	→	id	rest							
27.	PARAM	→	TERM	PARAM_LIST							
28.	PARAM	→	eps								
29.	PARAM_LIST	→	,	PARAM							
30.	FUNC	→	eps								
31.	TERM	→	id								
32.	TERM	→	int								
33.	TERM	→	float								
34.	TERM	→	string								
35.	FUNC_OR_EX	→	FUNC								
36.	STAT	→	EXPR								

2.2 LL tabulka

Neterminál/terminál	DEF	ID	IF	ELSE	WHILE	RETURN	DEDENT	PASS	EOL	INPUTS	INPUTI	INPUTF	PRINT	LEN	SUBSTR	ORD	CHR	)	,	INT	FLOAT	STRING	$
PROG	3	2	2		2	2		2		2	2	2	2	2	2	2	2						$
STAT		9	6		8	14		11		10	10	10	10	10	10	10	10						
F_DEF	4																						
F_NAME		5																					
STAT_LIST		12	12		12	12	13	12		12	12	12	12	12	12	12	12						
PARAM		27																28		27	27	27	
EXPR																							
IF_ELSE				7																			
ASSIGN		25																					
FUNC										17	18	19	20	21	22	23	24						
RETURN_TAIL									16														
FUNC_OR_EX		26								35	35	35	35	35	35	35	35						
TERM		31																		32	33	34	
PARAM_LIST																		30	29				

2.3 Precedenční tabulka

Na zásobníku/vstup	i	+	-	*	/	//	>=	>	<=	<	==	!=	(	)	$
i		>	>	>	>	>	>	>	>	>	>	>		>	>
+	<	>	>	<	<	<	>	>	>	>	>	>	<	>	>
-	<	>	>	<	<	<	>	>	>	>	>	>	<	>	>
*	<	>	>	>	>	>	>	>	>	>	>	>	<	>	>
/	<	>	>	>	>	>	>	>	>	>	>	>	<	>	>
//	<	>	>	>	>	>	>	>	>	>	>	>	<	>	>
>=	<	<	<	<	<	<							<	>	>
>	<	<	<	<	<	<							<	>	>
<=	<	<	<	<	<	<							<	>	>
<	<	<	<	<	<	<							<	>	>
==	<	<	<	<	<	<							<	>	>
!=	<	<	<	<	<	<							<	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	=	
)		>	>	>	>	>	>	>	>	>	>	>		>	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<		

Barevné vysvětlivky	
	Priorita
	Asociativita
	Identifikátory
	Závorky
	Konec výrazu

2.4 Analýza kostry programu

Syntaktická analýza kostry programu se řídí metodou rekurzivního sestupu a pravidly LL gramatiky. Syntaktická i sémantická analýza kostry programu (všeho kromě výrazů) je implementována ve zdrojových souborech `parser.h` a `parser.c`.

2.4.1 Činnost parseru

Syntaktická analýza žádá tokeny od LA pomocí funkce `get_token()` a kontroluje jejich platnost. Jestliže od LA obdrží hodnoty NULL nebo `T_LEX_UNKNOWN`, nastaví příslušnou chybu překladu. Každý neterminál LL-gramatiky je zpracováván svou vlastní funkcí. K vyhodnocení výrazů je volána funkce `eval_expression()` z modulu zpracovávajícího výrazy.

Současně se syntaktickou analýzou probíhá i analýza sémantická. Zpracovávané proměnné a funkce se ukládají do lokální či globální tabulky symbolů. Abychom zjistili, zda právě analýza probíhá uvnitř funkce, nastavujeme boolovskou proměnnou `inFunction`, která se při vstupu do funkce zpracovávající neterminál F_DEFINITION nastaví na 1. Tato informace se nachází při několika sémantických kontrolách, například, jestli se příkaz `return` nevyskytuje mimo tělo funkce.

2.4.2 Řešení redundance kódu

Vzhledem k povaze parseru zde bylo společně s analýzou výrazů nejvyšší riziko neúnosné redundance kódu. Redundance byla snížena použitím následujících pomocných funkcí:

- `all_fun_defined()` – prochází celou globální tabulku symbolů a kontroluje, zda byly všechny volané funkce již definované,
- `func_table_control()` – kontroluje, zda položka tabulky se stejným jménem jako funkce přijatá z LA už existuje v globální tabulce. Jestliže ano, zkoumá, zda se jedná o volání (a odkud se funkce volá), nebo pokus o redefinici,
- `func_check_param()` – v případě, že funkce se v tabulce nachází bez parametrů, přiřadí je, jinak porovná, jestli se jejich počet shoduje se vstupem,
- `help_conditions()` – snižuje redundanci při syntaktické kontrole příkazů `if` (`else`), `while` a `def`,
- `check_build_in_func_param()` – kontroluje typ parametru předaného vestavěné funkci.

2.4.3 Poznámky k implementaci

Token *identifikátor*
LL gramatika uvedená v podkapitole 2.1 se mírně liší od implementace ve zpracování tokenu *identifikátor* (`T_ID`). V implementaci byla použita pomocná funkce `help_id()`, pro rozlišení, zda byl identifikátor použit ve výrazu, přiřazení či ve volání funkce. Při rozhodování, o který případ se jedná, je použitý token následující po `T_ID`. Obdobný postup je aplikován v případě přiřazování funkce anebo výrazu ve funkci `func_or_expr()`.

Vnořené funkce
Na základě příspěvku na fóru k projektu analýza neřeší, zda byly všechny vnořené funkce definovány před voláním funkce, kde jsou použity.

2.5 Pravidla pro analýzu výrazů

Neterminál		Handle			Symbolický zápis
E_E	→	T_ID			
E_E	→	T_NONE			
E_E	→	T_INT			$E \rightarrow i$
E_E	→	T_STRING			
E_E	→	T_DOUBLE			
E_E	→	E_E	T_PLUS	E_E	$E \rightarrow E + E$
E_E	→	E_E	T_MINUS	E_E	$E \rightarrow E - E$
E_E	→	E_E	T_MUL	E_E	$E \rightarrow E * E$
E_E	→	E_E	T_DIVISION	E_E	$E \rightarrow E / E$
E_E	→	E_E	T_F_DIVISION	E_E	$E \rightarrow E // E$
E_E	→	E_E	T_GE	E_E	$E \rightarrow E >= E$
E_E	→	E_E	T_GT	E_E	$E \rightarrow E > E$
E_E	→	E_E	T_LE	E_E	$E \rightarrow E <= E$
E_E	→	E_E	T_LT	E_E	$E \rightarrow E < E$
E_E	→	E_E	T_EQUAL	E_E	$E \rightarrow E == E$
E_E	→	E_E	T_N_EQUAL	E_E	$E \rightarrow E != E$
E_E	→	T_L_BRACKET	E_E	T_R_BRACKET	$E \rightarrow (E)$

2.6 Analýza výrazů

Syntaktická analýza výrazů pracuje podle souboru pravidel pro postup zdola nahoru uvedeného v podkapitole 2.5. Syntaktická i sémantická analýza výrazů je implementována ve zdrojových souborech `expressions.h` a `expressions.c`.

2.6.1 Činnost modulu

Analýza výrazů používá jediný neterminál (E), který představuje libovolný podvýraz a je implementován strukturou datového typu `expr` (`*expression`). Každý podvýraz je stejně jako proměnná nějakého datového typu, který je určen během typové kontroly. Analýza dále pracuje se symboly z precedenční tabulky, které vkládá na zásobník. Pro usnadnění práce a odstranění nutnosti převodů byly lexémy, symboly z tabulky symbolů a speciální lexém označující neterminál E sloučeny do jednoho datového typu `symbol`.

Samotná pravidla jsou implementována jako pole struktur obsahujících levou stranu pravidla, pravou stranu pravidla a funkci, která zajišťuje jeho provedení. Provedení pravidla spočívá v převodu pravé strany na levou.

Rozhraní modulu tvoří jediná funkce `eval_expression()`, která generuje IFJcode19 pro výraz a vrací 0, případně vrací -1 a nastavuje chybový příznak.

2.6.2 Zásobník

Zpracování výrazů využívá zásobník implementovaný v souborech `expr_stack.h` a `expr_stack.c`. Místo zásobníku rozšířeného jde vzhledem k jednodušší realizaci o běžný zásobník, který musí vzhledem k datům, která je potřeba ukládat, pojmout položky typu `token` i `expr`.

2.6.3 Typová kontrola

Operandy výrazu mohou být datového typu ze zadání (`float`, `int`, `string`, `none`, `bool`) nebo speciálního typu `undef`, který značí, že datový typ bude znám až za běhu. Na základě datového typu je provedena statická, nebo dynamická typová kontrola. Že jeden z operandů výrazu je typu `undef`, nemusí nutně znamenat, že výsledkem bude také typ `undef`.

3 Generování kódu IFJcode19

Generování cílového kódu je implementováno ve zdrojových souborech `generator.h` a `generator.c`. Překladač generuje přímo IFJcode19, bez použití tříadresného kódu.

3.1 Funkce rozhraní

- `generator_init()` – alokuje dynamické proměnné a struktury, nastaví inicializační hodnoty,
- `generator_destroy()` – uvolní alokovanou paměť,
- funkce rozhraní pro generování IFJcode19, deklarované ve tvaru *gen_název_generovaného_celku()* (např. `gen_fnc_call()`, `gen_while_beg()`, ...). Tyto funkce však negenerují IFJcode19 samy o sobě, nýbrž volají privátní pomocné funkce, např. `gen_priv_jump_if()` a další.

3.2 Vnitřní řešení

3.2.1 Buffer

Pro účely generátoru je v souboru `generator.c` implementován buffer (fronta položek typu `string`), který je používán k průběžnému ukládání definic funkcí (`func_buffer`) a hlavního těla programu (`main_buffer`. Obsahy obou bufferů jsou vypsány na standardní výstup pomocí funkce `gen_end()` po ukončení analýzy vstupu (bez ohledu na návratový kód překladače).

Rozšířením oproti klasickému jednoduchému bufferu je schopnost vkládat na předem označené místo fronty definice proměnných. Tato vlastnost je využita v případě, že definice proměnné proběhla uvnitř `if-else` nebo cyklu `while`.

3.2.2 Zpracování funkcí a výrazů

Při generování volání funkce je využit ještě jeden buffer pro její parametry, který je postupně plněn (pomocí `gen_id_param()` a `gen_const_param()`) a jeho obsah je použit v okamžiku volání funkce generovaného pomocí `gen_fnc_call()`.

V samotném IFJcode19 jsou parametry funkce předávány ze zásobníku do příslušných proměnných parametrů. Návratová hodnota putuje také přes zásobník (stejně jako u výrazů).

Při volání vestavěné funkce se její obsah do IFJcode19 generuje přímo, čímž je dosaženo rychlejšího programu.

Zpracování výrazů probíhá na datovém zásobníku a výsledek tedy zůstává na jeho vrcholu.

3.2.3 Typové kontroly, konverze

Nekonstantní parametry funkcí, jejichž datový typ není možné zkontrolovat v parseru, jsou kontrolovány dynamicky v kódu (`gen_param_ID_check()`. Generátor produkuje i další dynamické typové kontroly a konverze (`gen_bool_conversion()`, `gen_types()`). Pro dynamickou typovou kontrolu ve výrazech se používá makro `check_type()`, které umožňuje generovat pouze typové kontroly/konverze, které jsou potřeba. Dělení nulou je kontrolováno vždy před operací dělení a to dynamicky.

4 Tabulka symbolů

Tabulka symbolů byla v souladu se zadáním projektové varianty II implementována jako abstraktní datový typ tabulka s rozptýlenými položkami. Vzhledem k subjektivně větší míře intuitivity pro potřeby projektu bylo zvoleno provedení s explicitně zřetězenými synonymy, kde každý řádek je jednosměrně zřetězený lineární seznam.

4.1 Funkce rozhraní

- **Práce s tabulkou** – `symtab_init()`, `symtab_find()`, `symtab_find_insert()`, `symtab_size()`, `symtab_find_remove()`, `symtab_reset()`, `symtab_destroy()`
- **Práce s položkou** – `get_item_type()`, `symtab_delete_item()`, `symtab_first_item()`, `symtab_next_item()`
- **Vkládání atributů** – `add_var_at()`, `set_is_init()`, `set_var_t()`, `add_fun_at()`, `set_is_def()`, `set_ret_t()`, `set_args()`, `set_fn_calls()`
- **Čtení atributů** – `get_is_init()`, `get_var_t()`, `get_is_def()`, `get_ret_t()`, `get_args_cnt()`, `get_args()`, `get_fun_calls_cnt()`, `get_fun_calls()`

4.2 Řešení vybraných problémů

Popisy některých strukturních vlastností a funkcionalit hashovací tabulky demonstrují řešení hlavních problémů při implementaci. Řešení většiny dalších problémů je podobné.

4.2.1 Hashovací funkce

Pro rozptýlení položek byl použit BKDR hash implementovaný podle vzoru ve studijních materiálech předmětu IAL.

4.2.2 Tabulka

Rámec tabulky byl implementován ve struktuře `symtab`, složené z proměnných

- `size` – drží aktuální počet položek v poli,
- `index_width` – počet řádků tabulky,
- `ptr_arr[ ]` – pole o proměnné velikosti ukazatelů na počátky zřetězených seznamů představujících jednotlivé řádky tabulky.

Tato implementace uživateli dovoluje pomocí funkce `symtab_init()` vytvořit tabulku velikosti podle jeho vlastního uvážení.

4.2.3 Položka tabulky

Položka tabulky, implementovaná ve struktuře `symt_item` má následující složky:

- `id` – název identifikátoru, unikátní klíč položky,
- `iden_t` – typ identifikátoru (proměnná, funkce, nedefinováno),
- `is_init_def` – určuje zda byla proměnná inicializována či funkce definována,
- `data_ret_t` – datový typ proměnné či návratový typ funkce,

- `pars` – ukazatel na strukturu obsahující parametry funkce, pro proměnné je vždy NULL,
- `calls` – ukazatel na strukturu obsahující seznam jiných funkcí volaných v této, pro proměnné je vždy NULL
- `next` – ukazatel na zřetězené synonymum.

Položka tabulky vzniká úspěšným provedením funkce `symtab_find_insert()` v případě, že již neexistuje jiná se stejným `id`. Právě vzniklá položka bez vložených atributů je vždy identifikátor nedefinovaného typu.

4.2.4 Atributy proměnné

Jak je patrné z předchozího odstavce, atributy proměnné jsou přímo součástí každé položky tabulky a je možné nastavit jejich hodnotu pomocí funkcí `add_var_at()`, `set_is_init()` a `set_var_t()`. Úspěšné použití jedné z těchto funkcí na identifikátor nedefinovaného typu nastaví typ na proměnnou, na položky typu funkce je nelze použít.

4.2.5 Atributy funkce

Atributy funkce určující její návratový typ a zda byla definována jsou vzhledem ke shodným datovým typům zastoupeny stejnými proměnnými. Toto řešení oproti původnímu, kde byly atributy funkce a proměnné v položce tabulky zastoupeny ukazateli na další úroveň struktur, vyžaduje nižší počet přístupů do paměti, ale přitom zachovává abstrakci díky rozhraní. Funkce `set_is_def()` a `set_is_init()` pracují analogicky s jejich protějšky v předchozím odstavci a lze je použít jen na položky typy funkce nebo nedefinovaného. Totéž platí pro funkce `add_fun_at()`, `set_args()` a `set_fn_calls()` které navíc ukládají i zadaný počet parametrů/volaných funkcí.