Projektová dokumentace

Implementace překladače jazyka IFJ23

Tým xsedla2e, varianta TRP-izp

Jiří Sedlák	xsedla2e	25% bodů
Nikol Škvařilová	xskvar11	25% bodů
Martin Kučera	xkucer0s	25% bodů
Tomáš Mikát	xmikat01	25% bodů

Obsah

1 Úvod **3**
1.1 Práce v týmu 3
1.2 Soubory 3
2 Implementace **4**
2.1 Lexikální analýza 4
2.1.1 Token 4
2.1.2 Scanner 4
2.1.3 Konečný stavový automat 5
2.2 Syntaktická analýza 7
2.2.1 Prediktivní analýza 7
2.2.1.1 LL-Tabulka 8
2.2.1.2 LL-Gramatika 8
2.2.2 Precedenční analýza 9
2.2.2.1 Precedenční tabulka 10
2.3 Sémantická analýza 11
2.3.1 Struktura Analyzer 11
2.3.2 Tabulka symbolů 11
2.4 Generátor kódu 12
2.4.1 Hlavička 12
2.4.2 Hlavní tělo 12
2.4.3 Patička 12
2.4.4 Formát názvů 12
3 Závěr **13**

1 Úvod

Cílem projektu bylo vytvořit překladač jazyka IFJ23 založeného na jazyce Swift, který vstupní program přeloží do mezikódu spustitelného interpretem IFJcode23.

1.1 Práce v týmu

Komunikace v týmu probíhala přes platformu Discord a při větších debatách osobně na fakultě. Každý člen týmu při vývoji aktivně používal git (přes program GitKraken) a GitHub pro vzdálené uložiště.

Rozdělení práce na projektu popisuje následující tabulka.

Jméno a přijmení	xlogin	Činnosti
Jiří Sedlák	`xsedla2e`	vedoucí týmu, registrace zadání a varianty, implementace syntaktické a precedenční analýzy (+ dokumentace)
Nikol Škvařilová	`xskvar11`	implementace lexikální analýzy (+ dokumentace), obecná část dokumentace
Martin Kučera	`xkucer0s`	implementace sémantické analýzy (+ dokumentace)
Tomáš Mikát	`xmikat01`	implementace generátoru kódu, volání generátoru kódu z ostatních souborů (+ dokumentace)

Na testování a finalizace projektu se podílel celý tým.

1.2 Soubory

Token `token.c, token.h, symtable.h, token_stack.c, token_stack.h`

Tabulka symbolů `symbtale.c, symtable.h`

Lexikální analýza `scanner.c, scanner.h`

Syntaktická analýza `parser.c, parser.h, expression_stack.c, expression_stack.h, rule_stack.h, rule_stack.h`

Sémantická analýza `param_stack.c, param_stack.h`

Generátor kódu `code_generator.c, code_generator.h`

2 Implementace

Následuje popis implementace částí překladače. Celý překlad řídí syntaktická analýza (parser). Ten následně volá funkce ze všech dalších částí.

2.1 Lexikální analýza

2.1.1 Token

Token, který se předává mezi všemi částmi překladače, je naimplementován v souborech `token.c`, `token.h` a částečně v symtable.h. Jedná se o strukturu `struct Token` (či `TokenPtr`) obsahující atributy

`char*`	`data`	obsahuje ukazatel na alokované místo s načtenými daty
`int`	`data_len`	délka načtených dat
`int`	`data_allocd`	délka zaalokovaného místa pro data
`int`	`type`	typ tokenu (enum `token_types`)
`int`	`value_type`	typ dat v tokenu (enum `Types` v `symtable.h`)

Pro práci se strukturou jsou k dispozici funkce

`TokenPtr token_init()`	zaalokuje místo na token, inicializuje jej
`bool token_add_data(TokenPtr token, char c)`	vloží `c` do dat
`void token_dispose(TokenPtr token)`	uvolní zaalokovanou paměť

Jak se token dále využívá a kde se ukládá bude rozepsáno v popisu dalších částí překladače.

2.1.2 Scanner

Lexikální analýza je zpracovaná v souborech `scanner.c` a `scanner.h`.

Nejdůležitější funkcí je `void get_next_token(TokenPtr token)`. Funkce dostane zaalokovaný a inicializovaný token; čte znaky ze `stdin` a zpracovává je podle konečného stavového automatu popsaného níže. Jednotlivé znaky ukládá do `token->data`, na konci čtení (dané stavovým automatem) se rozhodne, o jaký typ tokenu a typ dat v tokenu se jedná (funkce `void get_token_type(scanner_states* state, char c, TokenPtr token)`). Tyto typy se dále využívají v syntaktické a sémantické analýze.

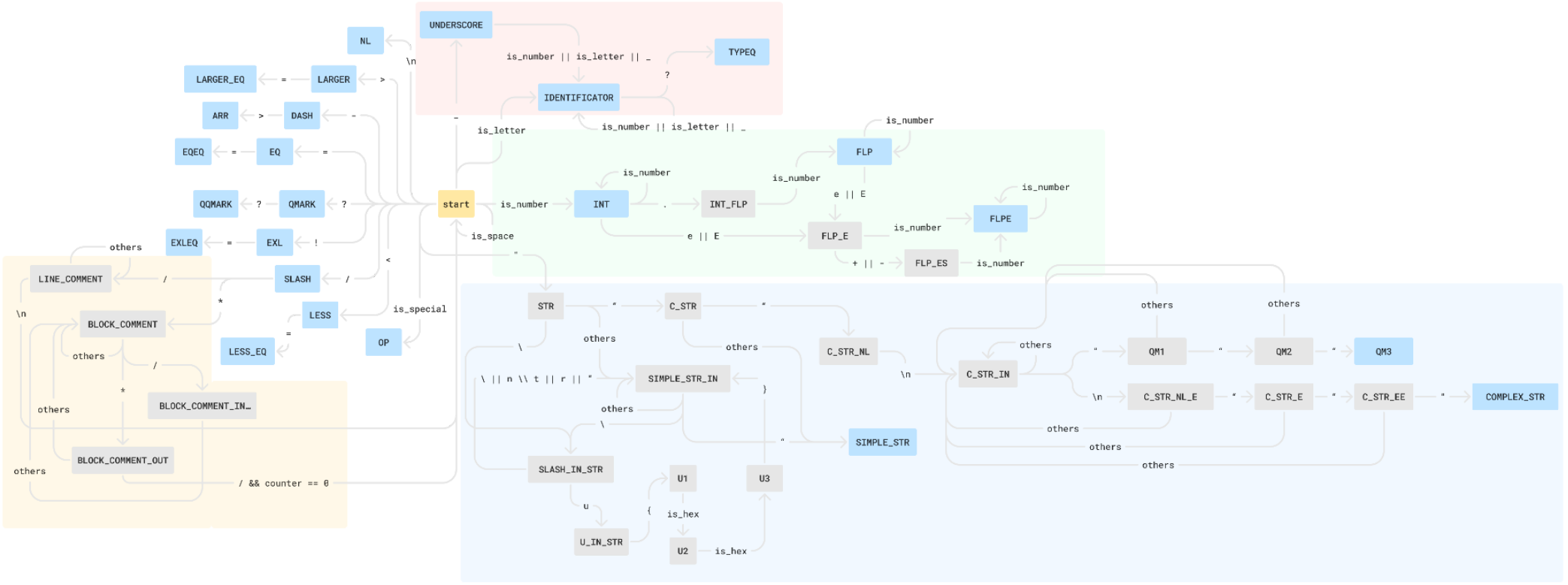
Další důležitou funkcí scanneru je funkce `void unget_token(TokenPtr token)` která umožňuje navrácení dat v tokenu zpět na stdin; tedy při dalším volání `get_next_token()` se vyrobí identický token. Uvolnění tokenu zajišťuje volající funkce. Konkrétní využití funkce je popsáno v části syntaktické a precedenční analýzy.

Scanner dále využívá spoustu vnitřních funkcí – hlavně pro zpracování jednotlivých stavů automatu – které zlepšují přehlednost kódu.

Za zmínku stojí pomocné proměnné `bool end` a `bool true_end`, které značně ulehčují práci funkcím pro zpracovávání stavů – totiž je-li `end` nastaven na `true`, automat ještě před ukončením běhu navrátí poslední přečtenou hodnotu na `stdin` (tedy umožňuje dívat se na znak za přečtenými daty). `true_end` poslední přečtený znak nevrací. Další významnou pomocnou proměnnou je `bool do_not_add`, která, jak již název napovídá, signalizuje, že přečtený znak se nemá propsat do datové části tokenu – a tedy umožňuje přeskakovat nechtěné znaky.

2.1.3 Konečný stavový automat

Konečný stavový automat je implementován ve funkci `get_next_token` pomocí cyklu `while`, výčtu `enum scanner_states` pro zlepšení čitelnosti, proměnné udržující stav automatu `scanner_states state`, přepínače `switch`, a pomocných funkcí na zpracování jednotlivých stavů (s názvy `parse_` a `is_`).


start
UNDERSCORE
NL
LARGER_EQ
LARGER
ARR
DASH
EQEQ
EQ
QQMARK
QMARK
EXLEQ
EXL
LINE_COMMENT
SLASH
BLOCK_COMMENT
BLOCK_COMMENT_IN…
BLOCK_COMMENT_OUT
LESS
LESS_EQ
OP
TYPEQ
IDENTIFICATOR
INT
INT_FLP
FLP
FLP_E
FLP_ES
FLPE
STR
C_STR
C_STR_NL
C_STR_IN
QM1
QM2
QM3
C_STR_NL_E
C_STR_E
C_STR_EE
COMPLEX_STR
SIMPLE_STR_IN
SIMPLE_STR
SLASH_IN_STR
U_IN_STR
U1
U2
U3
is_number || is_letter || _
is_letter
is_number
is_space
is_special
is_hex
e || E
+ || -
/ && counter == 0
\ || n \\ t || r || "
others
\n

2.2 Syntaktická analýza

2.2.1 Prediktivní analýza

Syntaktická analýza je až na výrazy, které se zpracovávají precedenční analýzou, implementována prediktivní analýzou řízenou LL-tabulkou a to v souborech `parser.c` a `parser.h` s hlavní funkcí `parse()`. Pravidla jsou zapsána jako posloupnost tokenů, předpisů nebo volání funkcí, které se přidávají na zásobník `rule_stack`.

Hlavní zásobník, který se využívá při syntaktické analýze k ukládání posloupností podle pravidel je implementován strukturou `rule_stack` v souborech `rule_stack.c` a `rule_stack.h`. Položky v tomto zásobníku mají tři hlavní atributy:

`int`	`type`	Typ položky, používají se enumy: token_types, Rules, Function
`bool`	`rule`	Atribut určující, zda je položka předpis
`bool`	`function`	Atribut určující, zda je položka volání funkce

V případě, že atributy `rule` a `function` jsou oba nastaveny na `false`, jedná se o token.

Další důležitou zásobníkovou strukturou je `token_stack`, která je implementovaná ve stejnojmených souborech. Tato struktura slouží pro práci s tokeny a jejich ukládání. Hlavními funkcemi jsou `token_stack_get()`, která požádá scanner o nový token, přidá jej na stack a je zároveň návratovou hodnotou funkce. Další je `token_stack_unget()`, která token vrátí zpět s využitím funkce scanneru `unget_token()`.

Syntaktická analýza skrze `token_stack` žádá scanner o nové tokeny. V případě, že se na vrcholu zásobníku `rule_stack` nachází předpis, tedy položka s atributem `rule` nastaveným na `true`, vyhledá se v LL-tabulce pravidlo odpovídající dané kombinaci, které, v případě, že kombinace je validní, přidá na `rule_stack` příslušné položky, jinak se překlad ukončí s návratovou hodnotou 2.

Volání funkcí je vyřešeno přidáním položky, jejíž atribut `function` je nastavený na `true`, na zásobník rule_stack. Takže v případě, kdy je na vrcholu zásobníku `rule_stack` položka s atributem `function` nastaveným na `true`, provede se příslušné volání funkce.

Pro poslední případ, kdy se na zásobníku `rule_stack` nachází položka, která má atributy `rule` i `function` nastaveny na `false`, se typ dané položky porovná s typem příchozího tokenu. V případě, že jsou stejné, se položka odstraní ze stacku a parser zažádá o další token. V opačném případě se překlad ukončí s návratovou hodnotou 2.

Nové řádky jsme museli implementovat odlišně od ostatních tokenů. Pokud parser narazí na token s typem `NEWLINE`, nastaví si vnitřní proměnnou `new_line` na `true` a načítá další tokeny dokud nedostane token s jiným typem než `NEWLINE`. Vnitřní proměnná `new_line` se dále používá při začátku nového výrazu. Pokud by proměnná `new_line` nebyla nastavena na `true` a zároveň by následující token měl typ jednoho z klíčových slov nebo `ID`, analýza se ukončí s návratovou hodnotou 2.

2.2.1.1 LL-Tabulka

	ID	VALUE	TYPE	WHILE	IF	ELSE	FUNC	RETURN	LET	VAR	EQUALS	D_DOT	PLUS	MINUS	MULT	DIV	EXL_MARK	QQ_MARK	EXL_EQ_MARK	EQUALS_EQUALS	LARGER_THAN	LARGER_EQUALS	SMALLER_THAN	SMALLER_EQUALS	L_BRAC	R_BRAC	L_CBRAC	R_CBRAC	COMMA	ARROW	NIL	UNDERSCR	NEWLINE	END
R_G_BODY	37			16	15		10	41	1	2																		53						45
R_BODY	38			18	17			42	3	4																		54						46
R_VAR_DEF											7	6																						
R_VAR_ASG	8	8	8	8	8	8	8	8	8	8	9																							
R_F_DEF_F	12																									11						12		
R_F_DEF_N																										13			14					
R_CONF_DEF	20	20							19																20									
R_EXPR	21	22															52								52									
R_EXPR_ID	26	26	26	26	26	26	26	26	26	26			24	24	24	24	24	24	24	24	24	24	24	23	26		26						26
R_EXPR_OP	27	27	27	27	27	27	27	27	27	27			25	25	25	25	25	25	25	25	25	25	25	25		27		27						27
R_F_PAR_F	29	30																								28								
R_F_PAR_ID												31													47	32			32					
R_F_PAR_N																										36			33					
R_F_PAR	34	35																																
R_STAT											40														39									
R_RET_DEF	44	44																																
R_F_DEF_ID	48																															49		
R_F_RET_DEF																											51			50				
R_F_PAR_NA	55	56																																

2.2.1.2 LL-Gramatika

1. <R_G_BODY> -> LET ID <R_VAR_DEF> <R_G_BODY>
2. <R_G_BODY> -> VAR ID <R_VAR_DEF> <R_G_BODY>
3. <R_BODY> ->LET ID <R_VAR_DEF> <R_BODY>
4. <R_BODY> -> VAR ID <R_VAR_DEF> <R_BODY>
5. <R_VAR_DEF> -> ε
6. <R_VAR_DEF> -> D_DOT TYPE <R_VAR_ASG>
7. <R_VAR_DEF> -> EQUALS <R_EXPR>
8. <R_VAR_ASG> -> ε
9. <R_VAR_ASG> -> EQUALS <R_EXPR>
10. <R_G_BODY> -> FUNC ID L_BRAC <R_F_DEF_F> R_BRAC <R_F_RET_DEF> L_CBRAC <R_BODY> R_CBRAC <R_G_BODY>
11. <R_F_DEF_F> -> ε
12. <R_F_DEF_F> -> <R_F_DEF_ID> D_DOT TYPE <R_F_DEF_N>
13. <R_F_DEF_N> -> ε
14. <R_F_DEF_N> -> COMMA <R_F_DEF_ID> D_DOT TYPE <R_F_DEF_N>
15. <R_G_BODY> -> IF <R_CONF_DEF> L_CBRAC <R_BODY> R_CBRAC ELSE L_CBRAC <R_BODY> R_CBRAC <R_G_BODY>
16. <R_G_BODY> -> WHILE <R_CONF_DEF> L_CBRAC <BODY> R_CBRAC <R_G_BODY>
17. <R_G_BODY> -> IF <R_CONF_DEF> L_CBRAC <R_BODY> R_CBRAC ELSE L_CBRAC <R_BODY> R_CBRAC <R_BODY>
18. <R_BODY> -> WHILE <R_CONF_DEF> L_CBRAC <BODY> R_CBRAC <R_BODY>
19. <R_CONF_DEF> -> LET ID

```
20.  <R_CONF_DEF> -> Začátek precedenční syntaktické analýzy
21.  <R_EXPR> -> ID <R_EXPR_ID>
22.  <R_EXPR> -> Začátek precedenční syntaktické analýzy
23.  <R_EXPR_ID> -> L_BRAC <R_F_PAR_F> R_BRAC
24.  <R_EXPR_ID> -> Začátek precedenční syntaktické analýzy
25.  <R_EXPR_OP> -> Začátek precedenční syntaktické analýzy
26.  <R_EXPR_ID> -> ε
27.  <R_EXPR_OP> -> ε
28.  <R_F_PAR_F> -> ε
29.  <R_F_PAR_F> -> ID <R_F_PAR_ID>
30.  <R_F_PAR_F> -> VALUE <R_EXPR_OP> <R_F_PAR_N>
31.  <R_F_PAR_ID> -> D_DOT <R_F_PAR_NA>
32.  <R_F_PAR_ID> -> <R_F_PAR_N>
33.  <R_F_PAR_N> -> COMMA <R_F_PAR>
34.  <R_F_PAR> -> ID <R_F_PAR_ID>
35.  <R_F_PAR> -> VALUE <R_EXPR_OP> <R_F_PAR_N>
36.  <R_F_PAR_N> -> ε
37.  <R_G_BODY> -> ID <R_STAT> <R_G_BODY>
38.  <R_BODY> -> ID <R_STAT> <R_BODY>
39.  <R_STAT> -> L_BRAC <R_F_PAR_F> R_BRAC
40.  <R_STAT> -> EQUALS <R_EXPR>
41.  <R_G_BODY> -> RETURN <R_RET_DEF>
42.  <R_BODY> -> RETURN <R_RET_DEF>
43.  <R_RET_DEF> -> ε
44.  <R_RET_DEF> -> <R_EXPR>
45.  <R_G_BODY> -> ε
46.  <R_BODY> -> ε
47.  <R_F_PAR_ID> -> L_BRAC <R_F_PAR_F> R_BRAC
48.  <R_F_DEF_ID> -> ID ID
49.  <R_F_DEF_ID> -> UNDERSCR ID
50.  <R_F_RET_DEF> -> ARROW TYPE
51.  <R_F_RET_DEF> -> ε
52.  <R_EXPR> ->  Začátek precedenční syntaktické analýzy
53.  <R_G_BODY> -> ε
54.  <R_BODY> -> ε
55.  <R_F_PAR_NA> -> ID <R_F_PAR_N>
56.  <R_F_PAR_NA> -> VALUE <R_F_PAR_N>
```

2.2.2 Precedenční analýza

Pro syntaktickou analýzu výrazu je použita precedenční analýza, která je implementovaná v souboru `expression.c` a její rozhraní v `expression.h`. Hlavní funkcí je `parse_epxression()` která řeší samotnou analýzu. Další funkcí je `get_translated_token_type()`, která pro zadaný token vrátí správný typ, který používá precedenční analýza. Typy jsou zaznamenány v enumu `ExpressionTypes`. Tento překlad je důležitý, protože dovoluje indexovat precedenční tabulku přímo pomocí typu

položky v zásobníku expression_stack. Další funkcí je `expression_get_next_token()`, která slouží pro získání tokenu od scanneru, jeho uložení na zásobník tokenů a správné ukončení precedenční analýzy.

Precedenční analýza používá pomocnou strukturu `expression_stack`, jejíž implementace je v souboru `expression_stack.c` a její rozhraní v `expression_stack.h`. Slouží pro ukládání položek, které obsahují typ, definovaný jako int (používají se hodnoty z enumu `ExpressionTypes`), expression, který je definovaný jako `bool` a určuje, zda se jedná o Expression. Tento atribut se také používá pro vytvoření handle u levé závorky a vytvoření dna zásobníku. A poslední atribut je previous, který je definovaný jako ukazatel na předchozí položku v zásobníku. To nám dovoluje jednoduše kontrolovat, zda je možné uplatnit redukční pravidlo.

Spouští se podle pravidel v LL-Gramatice a ukončuje se podle speciálních požadavků v závislosti na uplatněném pravidle. V případě, že se precedenční analýza spustila pro výraz během přiřazování do proměnné, ukončí se, pokud narazí na nějaké klíčové slovo nebo na token s typem `ID`, přičemž za ním musí následovat token s typem `EQUALS` a musí mu předcházet token s typem `NEWLINE`. Pro případ, že analýza byla spuštěna pro argument funkce, ukončí se pokud narazí na token s typem `R_BRAC` za kterým následuje token s typem `NEWLINE` a token s typem `ID` nebo typem některého z klíčkových slov.

Precedenční analýza je řízena precedenční tabulkou, která stanovuje, jaká operace se má na zásobníku `expression_stack` vykonat. V případě `E_SFT` se na vrchol zásobníku `expression_stack` vloží nová položka dle aktuálního příchozího tokenu a požádá se o následující token. Pro případ `E_RED` se provede příslušné redukční pravidlo; pokud to není možné, precedenční analýza se ukončí s návratovou hodnotou 2. Pokud je přeložený typ následujícího tokenu `E_END` a na vrcholu zásobníku `expression_stack` se nachází položka s typem `E_END` a atributem `expression` nastaveným na `false`, tak se precedenční analýza ukončí s návratovou hodnotou 0.

2.2.2.1 Precedenční tabulka

	E_EXL	E_MUL	E_DIV	E_PLS	E_MIN	E_E	E_NE	E_GR	E_SM	E_EGR	E_ESM	E_QQ	E_LBRAC	E_RBRAC	E_ID	E_VALUE	E_END
E_EXL	E_ERR	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_ERR	E_SFT	E_ERR	E_SFT	E_SFT	E_RED
E_MUL	E_SFT	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_DIV	E_SFT	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_PLS	E_SFT	E_SFT	E_SFT	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_MIN	E_SFT	E_SFT	E_SFT	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_E	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_NE	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_GR	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_SM	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_EGR	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_ESM	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_ERR	E_RED	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_QQ	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_EQL	E_SFT	E_RED	E_SFT	E_SFT	E_RED
E_LBRAC	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_RED	E_SFT	E_EQL	E_SFT	E_SFT	E_RED
E_RBRAC	E_ERR	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_ERR	E_RED	E_ERR	E_ERR	E_RED
E_ID	E_ERR	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_ERR	E_RED	E_ERR	E_ERR	E_RED
E_VALUE	E_ERR	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_RED	E_ERR	E_RED	E_ERR	E_ERR	E_RED
E_END	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SFT	E_SCS

2.3 Sémantická analýza

Sémantická analýza je zpracována v souborech `analyzer.c` a `analyzer.h`.

Jde o sadu funkcí, které postupně volá parser a předává jim zásobníky s tokeny na levé a pravé straně. Funkce provedou sémantické kontroly a případné přidání položky do tabulky symbolů.

2.3.1 Struktura Analyzer

Analyzátor obsahuje datovou strukturu `struct Analyzer`. Struktura obsahuje atributy

`int`	`depth`	aktuální zanoření
`int`	`*block`	pole aktuálních bloků pro dané zanoření
`int`	`block_allocd`	délka zaalokovaného místa pro bloky
`SymTablePtr`	`symtable`	odkaz na tabulku symbolů
`TokenStackPtr`	`functionStack`	zásobník s nedefinovanými funkcemi pro kontrolu na konci zdrojového kódu

2.3.2 Tabulka symbolů

Tabulka symbolů je implementována jako hash tabulka v souborech `symtable.h` a `symtable.c`. Jde o datovou strukturu struct Symtable s atributy

`int`	`depth`	aktuální zanoření
`int`	`block`	aktuální blok
`int`	`type`	datový typ (enum `Types`)
`char*`	`id`	identifikátor proměnné
`char*`	`value`	hodnota v proměnné
`bool`	`isFunction`	je položka funkce?
`bool`	`isVar`	je položka modifikovatelná proměnná?
`bool`	`isDefined`	je položka definovaná?
`bool`	`skipDefCheck`	přeskočení funkce check_definition()
`bool`	`isNil`	je položka nil?
`bool`	`isLiteral`	je položka literál?
`ParamStackPtr`	`paramStack`	odkaz na zásobník s parametry funkce
`SymTableItem*`	`next`	odkaz na další položku v tabulce

Globální proměnné, lokální proměnné, funkce a jejich parametry se ukládají do jedné tabulky. Globální a lokální proměnné rozlišujeme pomocí atributu `depth` a `block`. Vždy, když se zanoří do podmínky, cyklu nebo funkce, inkrementuje se číslo hloubky a bloku. Při vynoření se dekrementuje číslo hloubky. Bloky se počítají pro každou hloubku zvlášť. Tím je vždy docíleno unikátní kombinace hloubky a bloku.

Funkce rozlišujeme pomocí atributu `isFunction` a její parametry se ukládají to atributu `paramStack`.

2.4 Generátor kódu

Generace kódu IFJcode23 na standardní výstup je implementovaná v souborech `code_generator.c` a `code_generator.h`. Samotná generace probíhá během syntaktické analýzy pomocí funkcií, které sa volají ve správném pořadí ostatními soubory nebo samotným generátorem kódu.

2.4.1 Hlavička

Na začátku se pomocí funkce `code_header()` vypíše povinná hlavička IFJcode23 kódu a definují se pomocné proměnné, pomocné a vestavěné funkce.
Pomocné funkce slouží primárně na zjednodušení a zkrácení kódu. Volají se automaticky při určitých operacích, které buď potřebují delší kód na vyřešení, nebo potřebují kontrolu typů proměnných a jejich možné přetypování.

Vyjímkou vestavěných funkcí je funkce `write()`, která je kvůli variabilnímu počtu vstupních argumentů lépe implementovaná jako flexibilně volaná funkce uvnitř programu.

2.4.2 Hlavní tělo

Na generování těla programu slouží všechny funkce mimo funkce v hlavičce a patičce. Velká část kódu je zaměřená na práci se zásobníkem, na který sa ukládají dočasné a výsledné hodnoty. Na těchto hodnotách se potom uskutečňují žádané operace, nebo se vybírají ze zásobníku podle potřeby.

Při volání `if` a `while` sa kvůli stejnému názvu návěští musí tyto návěští rozlišovat pomocí inkrementujícího číslování.

2.4.3 Patička

Na konci sa pomocí funkce `code_footer()` vygeneruje vyčištění zásobníku a ukončení programu.

2.4.4 Formát názvů

Pro dosáhnutí lepší přehlednosti v kódu a zjednodušení rozpoznání typů se před jednotlivé názvy/pojmenování vypisuje prefix podle jejich typu:

prefix	co značí
$	proměnná
!	pomocná proměnná
&	návěští funkce
&&	pomocné návěští nebo návěští pomocné funkce
?	dočasná proměnná pro posílání parametrů do funkce

3 Závěr

Výsledkem projektu je překladač jazyka IFJ23, který úspěšně přeloží vstupní program do mezikódu, nebo ohlásí lexikální/syntaktickou/sémantickou chybu.

Celý tým se podílel na vývoji a členové aktivně komunikovali. Podařilo se nám využít obou pokusných odevzdání, které nám poskytly hodnotnou zpětnou vazbu.