

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Projektová dokumentace

**Implementace překladače imperativního
jazyka IFJ25**

Tým *xliskah00*, varianta *TRP-izp*

Hana Liškařová	<i>xliskah00</i>	25%
Matěj Kurta	<i>xkurtam00</i>	25%
Martin Bíško	<i>xbiskom00</i>	25%
Šimon Dufek	<i>xdufeks00</i>	25%

Implementovaná rozšíření: CYCLES, BOOLTHEN, OPERATORS,
FUNEXP, EXTSTAT

Obsah

1	Úvod	2
2	Práce v Týmu	2
2.1	Rozdělení práce mezi členy týmu	2
2.1.1	Hana Liškařová (xliškah00)	2
2.1.2	Matěj Kurta (xkurtam00)	2
2.1.3	Martin Bíško (xbiskom00)	2
2.1.4	Šimon Dufek (xdufeks00)	2
2.1.5	Společná práce	2
3	Struktura projektu	3
3.1	Popis obsahu jednotlivých souborů	3
4	Návrh a implementace	4
4.1	Sdílené	4
4.1.1	Chybové kódy a hlášení chyb (error.c, error.h)	4
4.1.2	Hlavní běh programu (main.c)	5
4.2	Lexikální analyzátor	5
4.2.1	Konečný stavový automat lexikální analýzy	5
4.3	Syntaktický analyzátor	6
4.3.1	Gramatická pravidla	6
4.3.2	LL1 tabulka	7
4.3.3	Precedenční tabulka výrazů	7
4.3.4	Precedenční pravidla	7
4.3.5	Zpracovávání výrazů	9
4.4	Semantický analyzátor	9
4.4.1	Tabulky symbolů	9
4.4.2	První průchod AST - zhora dolů - semantic_pass1	9
4.4.3	Druhý průchod AST - semantic_pass2	10
4.5	Generátor kódu	10
4.5.1	Klíčové Struktury a Konvence	11
4.5.2	Fáze Generování Kódu	11
4.5.3	Generování Výrazů a Konverze Typů	11
4.5.4	Generování Řídících Struktur a Vestavěných Funkcí	12
5	Datové struktury	12
5.1	Dynamický řetězec	12
5.2	Tabulka symbolů	12
5.3	Zásobník rozsahů (scope_stack)	13
5.4	Zásobník	14
5.5	Token	14
5.6	Syntaktický strom	15
5.7	Generátor kódu	16
6	Závěr	16

1 Úvod

Cílem projektu bylo navrhnut a implementovat překladač imperativního jazyka IFJ25. Překladač musí pokrývat všechny fáze klasického kompilátoru – lexikální analýzu, syntaktickou analýzu, sémantickou analýzu a generování mezikódu IFJcode25. Tato dokumentace nepopisuje jen jednotlivé moduly, ale i postup, kterým jsme k výslednému řešení došli.

2 Práce v Týmu

2.1 Rozdělení práce mezi členy týmu

2.1.1 Hana Liškařová (**xliskah00**)

- Návrh konečného automatu lexikální analýzy a implementace celé lexikální analýzy.
- Tvorba prezentace k obhajobě.
- Většinový podíl na tvorbě dokumentace.
- Návrh a implementace fungování sémantické analýzy. Implementace `semantic_pass1` - průchod AST od shora dolů, naplnění tabulky symbolů a implementace základních sémantických kontrol.
- Kontrola a formátování souborů před odevzdáním. Přidávání doxygen hlaviček souborů.

2.1.2 Matěj Kurta (**zkurtam00**)

- Návrh a implementace tabulky symbolů a jejích operací.
- Návrh a implementace obecného zásobníku a jeho operací.
- Implementace `semantic_pass2` v sémantické analýze - průchod AST zdola nahoru a implementace typových kontrol.

2.1.3 Martin Bíško (**xbiskom00**)

- Návrh a implementace syntaktické analýzy a konstrukce AST.
- Implementace precedenční analýzy výrazů.

2.1.4 Šimon Dufek (**xdufeks00**)

- Návrh a implementace struktury AST a jeho operací.
- Návrh a implementace generátoru kódu.

2.1.5 Společná práce

- Dokumentace.

3 Struktura projektu

Níže je zobrazená struktura odevzdaného souboru s projektem:

```
projekt/
|--- Makefile
|--- dokumentace.pdf
|--- ast.c
|--- ast.h
|--- builtins.c
|--- builtins.h
|--- codegen.c
|--- codegen.h
|--- error.c
|--- error.h
|--- expressions.c
|--- expressions.h
|--- main.c
|--- parser.c
|--- parser.h
|--- rozdeleni
|--- rozsireni
|--- scanner.c
|--- scanner.h
|--- scope_stack.c
|--- scope_stack.h
|--- semantic.c
|--- semantic.h
|--- stack.c
|--- stack.h
|--- string.c
|--- string.h
|--- symtable.c
|--- symtable.h
|--- token.c
`-- token.h
```

3.1 Popis obsahu jednotlivých souborů

- **Makefile** – skript pro překlad projektu. Kompiluje všechny soubory do binárky `compiler` při `make`, obsahuje také pomocné cíle pro čištění (`make clean`) a testování.
- **dokumentace.pdf** – tento soubor. Projektová dokumentace ve formátu PDF.
- **main.c** – vstupní bod programu. Zajišťuje inicializaci, volání lexikální analýzy, syntaktické analýzy, sémantické analýzy a generátoru kódu, a na závěr uvolnění všech datových struktur.
- **scanner.c** a **scanner.h** – implementace lexikální analýzy jako konečného automatu. Čte znaky ze standardního vstupu a vytváří seznam tokenů, který slouží jako vstup pro syntaktickou analýzu.
- **token.c** a **token.h** – definice struktury tokenu a implementace obousměrného seznamu tokenů. Zajišťuje vytváření, průchod, mazání a základní pomocné operace nad sekvencí tokenů a tokenem samotným.

- **parser.c** a **parser.h** – syntaktická analýza. Zpracovává seznam tokenů podle gramatických pravidel a vytváří abstraktní syntaktický strom (AST).
- **expressions.c** a **expressions.h** – analýza výrazů založená na precedenční tabulce. Využívá zásobník a tokeny k parsování aritmetických a logických výrazů a převádí je do AST uzlů.
- **ast.c** a **ast.h** – definice datových struktur pro abstraktní syntaktický strom (AST) a funkce pro jeho vytváření, práci s jednotlivými uzly a těly, případně výpisu.
- **string.c** a **string.h** – implementace vlastního dynamického řetězce. Zajišťuje alokaci, realokaci, konka-naci, kopírování, mazání a výpis textu, včetně podpory pro generátor kódu.
- **error.c** a **error.h** – definice chybových kódů projektu a společná funkce `error` pro hlášení chyb na `stderr` a návrat odpovídajícího kódu.
- **symtable.c** a **symtable.h** – implementace tabulky symbolů založené na implicitně zřetězené hašovací tabulce. Uchovává informace o funkcích, proměnných a jejich vlastnostech (typ, rozsah, parametry apod.) pro potřeby sémantické analýzy.
- **stack.c** a **stack.h** – obecný zásobník (LIFO) nad ukazateli. Používá se například v analyzátoru výrazů a při práci s rámci sémantické analýzy.
- **scope_stack.c** a **scope_stack.h** – specializovaný zásobník rozsahů (scope stack) pro sémantickou analýzu. Ukládá ukazatele na lokální tabulky symbolů jednotlivých bloků a funkcí, čímž umožňuje kontrolu redeklarací a stínění identifikátorů.
- **builtins.c** a **builtins.h** – registrace vestavěných funkcí jmenného prostoru `Ifj.*` do globální tabulky symbolů (např. `Ifj.read_str`, `Ifj.write`). Ukládají se zejména názvy a arita funkcí.
- **semantic.c** a **semantic.h** – implementace první průchodové sémantické analýzy. Prochází AST, plní globální a lokální tabulky symbolů, kontroluje existenci `main()`, redeklarace, počet argumentů a základní typové kolize u literálových výrazů. - OPRAVIT AŽ BUDE HOTOVÉ
- **codegen.c** a **codegen.h** – generátor výsledného kódu. Rekurzivně prochází AST a vytváří cílový kód ve formátu `IFJcode25`, který je ukládán do dynamického řetězce a na konci vypsán na standardní výstup.
- **rozdeleni** – textový soubor s procentuálním podílem jednotlivých členů týmu.

4 Návrh a implementace

4.1 Sdílené

4.1.1 Chybové kódy a hlášení chyb (`error.c`, `error.h`)

Moduly `error.h` a `error.c` zajišťují jednotnou správu chyb v celém projektu. Soubor `error.h` definuje sadu konstant pro návratové kódy překladače, které odpovídají zadání.

V `error.c` je implementována funkce `error(int exit_code, const char *fmt, ...)`, která slouží jako centrální bod pro hlášení chyb:

- přijímá návratový kód a formátovací řetězec ve stylu `printf()`,
- pomocí `vfprintf()` vypisuje na `stderr` zprávu ve tvaru "Error:...",
- po vypsání zprávy vrací předaný `exit_code`.

To v jednom kroku zajistí vypsání srozumitelné chybové hlášky na standardní chybový výstup a zároveň vrácení správného kódu až do `main()`.

4.1.2 Hlavní běh programu (`main.c`)

Soubor `main.c` je vstupním bodem překladače a skládá dohromady jednotlivé fáze z ostatních modulů.

Na začátku se inicializuje seznam tokenů `DLLListTokens` a funkce `scanner(stdin, &token_list)` provede lexikální analýzu vstupu. Při chybě vrací `ERR_LEX` (nebo jiný kód) a `main()` tento kód propaguje ven po uvolnění seznamu tokenů.

Pokud lexikální analýza skončí úspěšně, nastaví se aktivní prvek seznamu na první token a vytvoří se kořen AST pomocí `ast_init()`. Funkce `parser(&token_list, ast_tree, GRAMMAR_PROGRAM)` zpracuje tok tokenů podle gramatiky - při syntaktické chybě skončí program s kódem `ERR_SYN`.

Na vytvořený AST navazuje sémantická analýza `semantic_pass1(ast_tree)`, která zbuduje tabulku symbolů a kontroluje základní sémantické vlastnosti programu (např. existenci `main()`, redeklarace, počty argumentů).

V případě úspěchu je alokována struktura `generator`, inicializovaná funkcí `init_code(gen, ast_tree)`. Následně `generate_code(gen, ast_tree)` projde AST a uloží výsledný `IFJcode25` do dynamického řetězce `gen->output`, který je vypsán na standardní výstup pomocí `fputs()`.

Na závěr `main()` uvolní generátor i seznam tokenů (`DLLTokens_Dispose()`) a končí s návratovým kódem `SUCCESS`.

4.2 Lexikální analyzátor

Lexikální analyzátor převádí proud znaků ze standardního vstupu na sekvenci tokenů, se kterou dále pracuje syntaktická analýza. Čte po jednom znaku ze vstupu - nikdy nedrží celý vstup v paměti a pamatuje si jen aktuální stav automatu, pozici ve vstupu (rádek, sloupec) a rozpracovaný token.

Rozhraní tvoří funkce `scanner(FILE *in, DLLListTokens *list)`, která uvnitř opakováně volá interní funkci `get_next_token()`. Každý nově rozpoznaný token se rovnou ukládá do obousměrného seznamu `DLLListTokens`, který je pak předán parseru.

Čtení znaků zajišťují pomocné funkce `get_char()` a `pushback()`, které kromě samotného znaku aktualizují i aktuální pozici a umožňují vrátit jeden znak zpět (typicky při lookaheadu u operátorů a čísel). Speciálně jsou řešeny konce řádků: kombinace `\r\n` i jednotlivé znaky `\r` / `\n` se vždy normalizují na jeden token `T_EOL`.

Z hlediska je scanner rozdělen na několik větví podle typu lexému:

- identifikátory a klíčová slova (včetně speciálních globálních identifikátorů začínajících na `_`),
- číselné literály (celá čísla v desítkové a šestnáctkové soustavě, desetinná čísla a exponenty),
- řetězce (klasické i víceřádkové),
- operátory a oddělovače,
- komentáře a bílé znaky.

Logika je rozdělená na dvě části: vlastní rozpoznání lexému konečným automatem a následné zpracování hodnoty. Například u čísel automat jen hlídá syntaxi (povolené znaky, správné umístění teček a exponentu), ale samotný převod na `long long` nebo `double` se provádí až na konci pomocí `strtoll()` / `strtod()`.

4.2.1 Konečný stavový automat lexikální analýzy

Jelikož tady v dokumentaci nejde návrh konečného automatu správně vidět, zde je odkaz na obrázek nahraný na Google drive. Jinak návrh najeznete na poslední straně tohoto dokumentu.

4.3 Syntaktický analyzátor

Hlavní funkcí syntaktického analyzátoru je rekurzivní funkce `parser`, která zpracovává tokeny a podle gramatických pravidel zhodnocuje, zda nedochází k syntaktické chybě. Při průchodu dochází i ke generování abstaktního syntaktického stromu (AST), který je následně využit pro sémantickou analýzu a generování kódu. Pro zpracování gramatických pravidel je vytvořen enum `grammar_rule`, který obsahuje data podle gramatických pravidel.

4.3.1 Gramatická pravidla

1. $\langle \text{program} \rangle \rightarrow \langle \text{import} \rangle \langle \text{class_list} \rangle$
2. $\langle \text{import} \rangle \rightarrow \text{import } \text{"ifj25"} \text{for Ifj}$
3. $\langle \text{class_list} \rangle \rightarrow \langle \text{class_def} \rangle \langle \text{class_list} \rangle \mid \varepsilon$
4. $\langle \text{class_def} \rangle \rightarrow \text{class ID} \langle \text{body} \rangle$
5. $\langle \text{body} \rangle \rightarrow \{ \langle \text{command_list} \rangle \}$
6. $\langle \text{command_list} \rangle \rightarrow \langle \text{command} \rangle \langle \text{command_list} \rangle \mid \varepsilon$
7. $\langle \text{command} \rangle \rightarrow \langle \text{fun_def} \rangle \mid \langle \text{fun_call} \rangle \mid \langle \text{ifj_fun_call} \rangle \mid \langle \text{declaration} \rangle \mid \langle \text{assignment} \rangle \mid \langle \text{condition} \rangle \mid \langle \text{for} \rangle \mid \langle \text{while} \rangle \mid \langle \text{return} \rangle$
8. $\langle \text{fun_def} \rangle \rightarrow \text{static ID} \langle \text{params} \rangle \langle \text{body} \rangle$
9. $\langle \text{params} \rangle \rightarrow (\langle \text{param_list} \rangle)$
10. $\langle \text{param_list} \rangle \rightarrow \text{ID} \langle \text{param_list}' \rangle \mid \varepsilon$
11. $\langle \text{param_list}' \rangle \rightarrow , \text{ ID} \langle \text{param_list}' \rangle \mid \varepsilon$
12. $\langle \text{declaration} \rangle \rightarrow \text{var ID}$
13. $\langle \text{assignment} \rangle \rightarrow \text{ID} = \langle \text{expression} \rangle$
14. $\langle \text{expression} \rangle \rightarrow \langle \text{expression} \rangle \langle \text{exp_operator} \rangle \langle \text{expression} \rangle \mid \langle \text{value} \rangle \mid \langle \text{fun_call} \rangle \mid \langle \text{ifj_fun_call} \rangle$
15. $\langle \text{exp_operator} \rangle \rightarrow + \mid - \mid * \mid /$
16. $\langle \text{condition} \rangle \rightarrow \text{if} \langle \text{cond_expression} \rangle \langle \text{body} \rangle \text{else} \langle \text{body} \rangle$
17. $\langle \text{cond_expression} \rangle \rightarrow (\langle \text{expression} \rangle \langle \text{cond_operator} \rangle \langle \text{expression} \rangle)$
18. $\langle \text{cond_operator} \rangle \rightarrow > \mid \geq \mid < \mid \leq \mid == \mid != \mid \text{is}$
19. $\langle \text{while} \rangle \rightarrow \text{while} \langle \text{cond_expression} \rangle \langle \text{body} \rangle$
20. $\langle \text{for} \rangle \rightarrow \text{for} (\text{ID} \text{ in } \langle \text{expression} \rangle \dots \langle \text{expression} \rangle) \langle \text{body} \rangle \mid \text{for} (\text{ID} \text{ in } \langle \text{expression} \rangle \dots \langle \text{expression} \rangle)$
21. $\langle \text{fun_call} \rangle \rightarrow \text{ID} \langle \text{args} \rangle$
22. $\langle \text{ifj_fun_call} \rangle \rightarrow \text{Ifj . ID} \langle \text{args} \rangle$
23. $\langle \text{args} \rangle \rightarrow (\langle \text{arg_list} \rangle)$
24. $\langle \text{arg_list} \rangle \rightarrow \langle \text{expression} \rangle \langle \text{arg_list}' \rangle \mid \varepsilon$

25. $\langle arg_list' \rangle \rightarrow , \langle expression \rangle \langle arg_list' \rangle \mid \varepsilon$
26. $\langle return \rangle \rightarrow \text{return } \langle return_value \rangle$
27. $\langle return_value \rangle \rightarrow \text{ID} \mid \langle fun_call \rangle \mid \langle ifj_fun_call \rangle \mid \langle value \rangle$

4.3.2 LL1 tabulka

4.3.3 Precedenční tabulka výrazů

	$*, /$	$+, -$	rel	is	eq	(id/const)	\$
$*, /$	>	>	>	>	>	<	<	>	>
$+, -$	<	>	>	>	>	<	<	>	>
rel	<	<	>	>	>	<	<	>	>
is	<	<	<	>	>	<	<	>	>
eq	<	<	<	<	>	<	<	>	>
(<	<	<	<	<	<	<	=	
id/const	>	>	>	>	>			>	>
)	>	>	>	>	>			>	>
\$	<	<	<	<	<	<	<		

4.3.4 Precedenční pravidla

- $E \rightarrow E * E$
- $E \rightarrow E / E$
- $E \rightarrow E + E$
- $E \rightarrow E - E$
- $E \rightarrow E < E$
- $E \rightarrow E \leq E$
- $E \rightarrow E > E$
- $E \rightarrow E \geq E$
- $E \rightarrow E == E$
- $E \rightarrow E \neq E$
- $E \rightarrow E \text{ is } E$
- $E \rightarrow (E)$
- $E \rightarrow i$

4.3.5 Zpracovávání výrazů

Výrazy jsou zpracovány pomocí funkce `parse_expr`, která prochází list tokenů a vytváří strom výrazu. Ke zpracovávání výrazů se používá pomocný zásobník a precedenční tabulka výrazů.

Na začátku se na zásobník vloží znak \$ a následně se porovnávají tokeny s vrchním znakem na zásobníku. Tyto dva znaky se zpracují dle precedenční tabulky a podle výsledku vykoná akci. Pokud výstupem je < nebo =, vloží znak na vrchol zásobníku, pokud je znak >, tak se výraz zredukuje dle pravidel. Pokud ale v precedenční tabulce není žádný znak, program skončí s kódem `ERR_SYN`.

Zredukování výrazů probíhá tak, že se vezmou vrchní symboly zásobníku a zredukují se dle precedenčních pravidel, kde **i** je ID, konstanta nebo konkrétní hodnota a **E** je výraz.

Po průchodu celého výrazu na zásobníku zůstane jeden konečný výraz, který je i výstupem funkce.

4.4 Semantický analyzátor

Sémantická analýza je implementovaná v souborech `semantic.c` a `semantic.h` a používá tabulkou symbolů (`symtable.*`), zásobník rozsahů (`scope_stack.*`) a registraci vestavěných funkcí (`builtins.*`). Z pohledu zbytku projektu má jedno hlavní vstupní rozhraní `semantic_pass1(ast_tree)`, které po úspěšném parsování spustí celou sémantickou analýzu.

Uvnitř je zpracování rozdeleno do **dvojí průchodů** přes abstraktní syntaktický strom (AST): první průchod naplňuje tabulkou symbolů, druhý průchod pak řeší konkrétní použití identifikátorů a typové kontroly výrazů.

4.4.1 Tabulky symbolů

Díky tomu, že v jazyce Wren nelze definovat vnořené funkce, bylo možné implementovat dva "typy" samotné tabulky symbolů. Prvním je globální tabulka obsahující **definice funkcí, definice vestavěných funkcí a globální promenné**. Druhým typem jsou tabulky daných rozsahů (scopes).

Přesné rozdělení tabulek vypadá takto:

- globální tabulka funkcí (uživatelských i `Ifj.*`), kde klíčem je jméno + arita,
- zásobník lokálních tabulek proměnných, reprezentovaný `scope_stack`, který odráží vnoření bloků v AST.

4.4.2 První průchod AST - zhora dolů - `semantic_pass1`

Funkce `semantic_pass1` nejprve vytvoří kontext sémantické analýzy (struktura `semantic`) a inicializuje globální tabulkou funkcí. Do té se pomocí `builtins_install()` zaregistrují vestavěné funkce `Ifj.*` v formátu (název, arita, očekávané typy parametrů).

Poté nastane sběr hlaviček uživatelských funkcí, getterů a setterů. Pomocná funkce (`collect_headers`) projde AST třídy a bloky a pro každý deklarovaný symbol vytvoří záznam v **globální tabulce**.

Klíč symbolu se zkládá z kombinace jména a arity, což umožňuje podporu přetěžování podle počtu parametrů.

Následně se průchod vrátí na těla bloků a začne zpracovávat lokální proměnné. Správu zanoření - `scope` řeší `scope_stack`, který drží zásobník ukazatelů na jednotlivé lokální tabulky symbolů (`symtable`).

Při vstupu do bloku proběhne `scopes_push()` a pro daný blok se vytvoří nová lokální tabulka. Při opuštění bloku `scopes_pop()` tabulku uvolní. Deklarace proměnných v aktuálním bloku se ukládají přes `scopes_declare_local()`, která zároveň hlídá redeklaraci ve stejném rozsahu.

V rámci `semantic_pass1` se provedou i základní kontroly nevyžadující přesnou znalost typů:

- ověření existence funkce `main()` s nulovou aritou,
- kontrola redeklarace funkcí a proměnných,

- kontrola, že `break` a `continue` se vyskytují pouze uvnitř cyklů,
- základní kontroly `return` uvnitř funkcí.

Výstupem prvního průchodu je globální tabulka funkcí a sada lokálních tabulek proměnných se správným zanořením.

4.4.3 Druhý průchod AST - `semantic_pass2`

Druhý průchod provádí kontrolu typové správnosti a řešení identifikátorů nad již kompletním stromem AST. Přitom používá informace nashromážděné v prvním průchodu (tabulky funkcí, přístupové metody, vestavěné funkce a deklarace třídních členů). Pass 2 znova inicializuje zásobník rámů a při vstupu do každého bloku vytváří nový rozsah viditelnosti.

Rozpoznávání identifikátorů Pro každý použitý identifikátor funkce `sem2_resolve_identifier` kontroluje:

- lokální proměnná nebo parametr (v `scope_stack`),
- getter/setter s daným názvem (v globální tabulce),
- globální proměnná podle vzoru `__name` (automaticky akceptována)

Typová kontrola výrazů Funkce `sem2_visit_expr` vyhodnocuje typy výrazů:

- aritmetické operace: numerické typy s unifikací (`int + num → num`),
- relační operátory: pouze numerické operandy,
- logické operátory: pouze booleovské operandy,
- operátor `is`: pravá strana musí být `Num`, `String` nebo `Null`.

Kontrola volání funkcí Vestavěné i uživatelské funkce se ověřují podle signatur `name#arity` uložených v tabulce z prvního průchodu.

- vestavěné funkce (`If j . *`): kontrola přesné arity,
- uživatelské funkce: kontrola existence signatury `name#arity` nebo správné arity .

Výstup druhého průchodu

- ověřená typová korektnost programu,
- aktualizované informace o lokálních i globálních proměnných,
- doplněné `cg_name` pro codegen,
- seznam globálních proměnných dostupný přes `semantic_get_globals()`.

4.5 Generátor kódu

Generátor kódu je implementován v souboru `codegen.c` a používá přístup **Stack-based evaluation** pro překlad **Abstraktního Syntaktického Stromu (AST)** do instrukcí pro virtuální stroj **IFJcode25**. Hlavní vstupní rozhraní je `generate_code(generator gen, ast ast)`, které se spustí po úspěšné sémantické analýze.

4.5.1 Klíčové Struktury a Konvence

Generátor kódu používá strukturu `generator` pro udržení stavu a řízení toku:

- **output**: Dynamický řetězec, cílový výstup `IFJcode25`.
- **counter**: Čítač pro generování unikátních návěští a dočasných proměnných (např. `whileStart123`).
- **loop_stack**: Zásobník pro sledování návěští `start_label` a `end_label` aktuálně vnořených cyklů (`while`), nezbytný pro implementaci `break` a `continue`.
- **Dočasné GF proměnné**: Sada globálních dočasných proměnných (`GF@tmp1`, `GF@fn_ret`, `GF@tmp_type_1/r`) sloužících pro uchování mezivýsledků a typových kontrol.
- **Adresování**: Funkce `var_frame_parse` automaticky doplňuje prefixy rámců (`GF@` pro globální, `LF@` pro lokální) pro správné adresování.

4.5.2 Fáze Generování Kódu

Fáze 1: Inicializace (`init_code`) Tato fáze připravuje kód a deklaruje globální rozsah:

- Vypíše hlavičku `.IFJcode25`.
- Generuje instrukce `DEFVAR` pro všechny dočasné globální proměnné.
- Volá `sem_def_globals`, která vygeneruje `DEFVAR` a `MOVE name nil@nil` pro všechny globální proměnné programu.

Fáze 2: Hlavní Průchod a Ukončení (`generate_code`)

- **Generování main**: Nejprve je vygenerována funkce `main` (`generate_main`), která je implicitním vstupním bodem programu.
- **Generování zbytku**: Poté rekurzivně vygeneruje kód pro všechny ostatní funkce a bloky (`generate_block`).

4.5.3 Generování Výrazů a Konverze Typů

Generování výrazů (`generate_expression_stack`) klade velký důraz na zajištění dynamické typové kompatibility.

1. **Sčítání / Konkatenace (AST_ADD)**: Využívá `generate_add_conversion`. Pro odhalení, zda se jedná o Sčítání, nebo konkatenaci.
2. **Násobení / Opakování (AST_MUL)**: Využívá `generate_mul_conversion`. Pokud se na levé straně výrazu nachází string a na pravé int, generuje se cyklus pro **opakování řetězce** (`generate_repetition`).
3. **Dělení (AST_DIV)**: Vždy konvertuje oba numerické operandy na `float` před instrukcí `DIV`.
4. **Běhová kontrola typů**: Funkce `generate_type_check` používá instrukci `TYPE` k ověření kompatibility typů a skok `JUMP IFNEQ` na chybové návěští `ERR26`.
5. **Auto-korekce**: `process_auto_corecion` zajišťuje, že při numerických operacích jsou oba operandy (pokud je to nutné) převedeny na `float`.

4.5.4 Generování Řídících Struktur a Vestavěných Funkcí

Vlastní funkce

- Generují návěští, CREATEFRAME, PUSHFRAME.
- Parametry se načítají ze zásobníku volání pomocí POPS do lokálních proměnných.
- Návratová hodnota je uložena do GF@fn_ret. Ukončení probíhá pomocí POPFRAME a RETURN.

Řízení toku

- **if/else**: Používá JUMPIFEQ GF@tmp_if bool@false else_label.
- **while cyklus**: Na loop_stack se ukládají návěští whileStart a whileEnd. Příkazy **break** a **continue** generují JUMP na příslušná návěští ze zásobníku.

Vestavěné funkce (`generate_ifjfunction`)

- **ifj.write**: Obsahuje logiku pro kontrolu, zda je float celým číslem, s následnou konverzí FLOAT2INT pro čistý výpis celých čísel.
- **ifj.substring**: Implementuje podřetězec pomocí cyklu využívajícího STRLEN, GETCHAR a CONCAT.

5 Datové struktury

5.1 Dynamický řetězec

V projektu se často pracuje s textovými hodnotami, jako jsou vstupní lexémy nebo generovaný cílový kód. Proto jsme zavedli vlastní datový typ pro dynamické řetězce, který umožňuje manipulaci s textovými daty neznámé či měnící se délky. Tento datový typ je definován v rozhraní `string.h` a implementován v souboru `string.c`.

Řetězec se vytváří funkcí `string_create()`, která alokuje počáteční kapacitu (nebo `DEFAULT_SIZE`). Při nedostatku místa se interní pomocná funkce zvětší kapacitu na dvojnásobek. Základní operace jsou:

- `string_append_char()` – přidání jednoho znaku na konec,
- `string_append_literal()` – připojení C řetězce,
- `string_concat()` – konkatenace dvou dynamických řetězců,
- `string_clear()` – nastavení na prázdný řetězec,
- `string_destroy()` – uvolnění vnitřních dat i struktury,
- `string_to_file()` – zápis generovaného mezikódu do souboru.

5.2 Tabulka symbolů

Tabulka symbolů je implementována jako hashovací tabulka pro ukládání a následnou práci s informacemi. Implementována je v souborech `symtable.h` a `symtable.c`. Využívá otevřené adresování pro řešení kolizí. Používá známou hashovací funkci `djb2` kvůli její nízké koliznosti a tabulka používá fixnou velikost `SYMTABLE_SIZE`. Při vkládání nových symbolů se ingoruje `st_insert()` pokud symbol již existuje.

- `data_type` – typ dat, tedy druh hodnoty

- `symbol_type` – kategorie (proměnná, funkce...)
- `st_data` – struktura obsahující veškerá metadata symbolu, včetně typů, počtu parametrů, informací o rozsahu (scope) a odkazu na uzel AST
- `st_symbol` – reprezentace jedné položky v tabulce obsahující klíč, data a příznaky `occupied` a `deleted` pro správu kolizí
- `syntable` – hlavní struktura tabulky symbolů obsahující pole symbolů a informaci o počtu uložených prvků

Rozhraní tabulky:

- `st_hash()` – hashovací funkce implementující algoritmus djb2
- `st_init()` – inicializuje novou prázdnou tabulkou symbolů a alokuje paměť
- `st_find()` – vyhledá symbol v tabulce podle klíče, vrátí ukazatel na symbol nebo NULL
- `st_insert()` – vloží nový symbol do tabulky, pokud ještě neexistuje; alokuje paměť pro klíč i data a nastaví základní atributy
- `st_get()` – získá datovou strukturu symbolu podle klíče, vrací `st_data`
- `st_free()` – uvolní veškerou alokovanou paměť tabulky, včetně klíčů, datových struktur a samotné tabulky
- `st_dump()` – vypíše lidsky čitelný obsah tabulky pro ladící účely; zobrazí indexy, klíče, typy a parametry
- `st_FOREACH()` – iteruje přes všechny obsazené položky v tabulce a pro každou volá uživatelem definovanou callback funkci
- `my_strdup()` – pomocná funkce pro duplikování řetězců s alokací nové paměti

V projektu existuje jedna globální tabulka symbolů pro funkce, přístupové metody a globální proměnné a dále samostatné lokální tabulky pro jednotlivé bloky a funkce. Lokální tabulky jsou spravovány pomocí `scope_stack` (viz oddíl o zásobníku rozsahů).

5.3 Zásobník rozsahů (`scope_stack`)

Samotná tabulka symbolů (`syntable.c/h`) řeší uložení a vyhledávání symbolů v jednom rozsahu - globální tabulku funkcí nebo lokální proměnné uvnitř jedné funkce. Sémantická analýza ale pracuje s zanořenými bloky. Dle našeho návrhu je potřeba mít pro každý blok vlastní `syntable` a zároveň nad nimi udržovat informaci o aktuálním rozsahu (scope).

K tomu slouží pomocná struktura `scope_stack`, která nad obecnou tabulkou symbolů staví správu rozsahů (scopes). Je navázáný na náš obecný zásobník `stack`, ve kterém jsou uloženy ukazatele na `syntable*`. Každý prvek zásobníku tedy odpovídá jednomu bloku a uchovává symboly z tohoto bloku. Na vrcholu zásobníku je vždy aktuální (nejvnitřejší) rozsah.

Typický průběh v sémantické analýze je následující:

- při vstupu do nového bloku se zavolá `scopes_push()`, která vytvoří novou prázdnou `syntable` pro daný rozsah,
- nové lokální proměnné se deklarují přes `scopes_declare_local()`, která v aktuálním rámci zakáže redeklaraci stejného jména,

- při vyhledávání identifikátoru se používá `scopes_lookup()`, která prochází rámce od vrcholu dolů a respektuje stínění,
- při opuštění bloku se volá `scopes_pop()`, která daný rámec sejmé a jeho `symtable` uvolní pomocí `st_free()`.

Globální tabulka symbolů (funkce programu a vestavěné `Ifj.*` a globály) je udržována samostatně jako jeden `symtable` mimo `scope_stack`.

5.4 Zásobník

Stejně jako u dynamického řetězce, několik oddělených částí projektu potřebuje pracovat s zásobníkem na principu LIFO. Rozhodli jsme se proto zavést jeden sdílený zásobník pro veškeré účely překladače. Obecný zásobník je v implementován nad ukazateli. Je definován v souborech `stack.h` a `stack.c`.

Vnitřní reprezentace je jednosměrně vázaný seznam. Každý prvek `stack_item` obsahuje:

- `void *data` – ukazatel na uložená data libovolného typu,
- `stack_item *next` – odkaz na další prvek.

Struktura `stack` obsahuje pouze ukazatel `top` na vrchol zásobníku.

Rozhraní zásobníku:

- `stack_init()` – inicializace prázdného zásobníku,
- `stack_push()` – vložení ukazatele na vrchol,
- `stack_push_value()` – alokace a uložení kopie hodnoty,
- `stack_pop()` – odebrání vrcholu a vrácení jeho `data`,
- `stack_top()` – nahlédnutí na vrchol bez odebrání,
- `stack_is_empty()` – test, zda je zásobník prázdný,
- `stack_free()` – uvolnění všech uzlů.

Díky tomu, že zásobník pracuje nad obecnými ukazateli `void *`, lze stejnou implementaci znova použít v různých částech překladače. Typová kontrola je řešena až na úrovni volajícího modulu pomocí přetypování ukazatelů na konkrétní struktury.

5.5 Token

Každý token reprezentuje jednu lexikální jednotku (klíčové slovo, identifikátor, literál, operátor, oddělovač, EOL, EOF) a nese svůj typ a případnou hodnotu. Definice je v `token.h`, implementace v `token.c`.

Struktura `struct token` obsahuje:

- `token_type type` – typ tokenu,
- `string value` – textová podoba lexému (např. identifikátor, řetězec),
- `double value_float` a `long long value_int` – numerická hodnota,
- `int depth` – pomocná hodnota pro zanoření blokových komentářů.

Typ `token_type` je rozlišen na:

- obecné typy: T_NONE, T_EOF, T_EOL,
- identifikátory
- klíčová slova
- literály
- operátory a oddělovače (aritmetické, relační, logické, závorky, čárky, rozsahové operátory).

Základní používané funkce nad tokenem:

- `token_create()` – alokace a inicializace nového tokenu,
- `token_clear()` – reset hodnot na výchozí stav,
- `token_destroy()` – uvolnění dynamického řetězce i struktury tokenu,

Seznam tokenů

Tokeny z lexikální analýzy jsou ukládány do **obousměrně vázaného seznamu** `DLLListTokens`, který předává lexikální analýza syntaktické analýze.

Struktura `DLLListTokens` drží:

- `first` – první prvek seznamu,
- `active` – aktuálně zpracovávaný token,
- `last` – poslední prvek,
- `length` – počet prvků.

Používané operace:

- `DLLTokens_Init()`, `DLLTokens_Dispose()` – vytvoření a uvolnění seznamu,
- `DLLTokens_InsertFirst()`, `DLLTokens_InsertLast()` – vložení tokenu na začátek/konec,
- `DLLTokens_First()`, `DLLTokens_Next()` – pohyb aktivního prvku,
- `get_token_type_ignore_eol()` – vrátí typ dalšího neprázdného tokenu, přeskakuje `T_EOL`.

5.6 Syntaktický strom

Struktury v `ast.h`:

- Struktura syntaktického stromu je definována v souboru `ast.h`. Strom je vytvořen v rámci syntaktické analýzy a je využíván v celém zbytku kompilátoru. Struktura je složena z hlavní struktury `texttast`, která obsahuje prolog a ukazatel na třídu `Program`.
- Struktura `ast_class` obsahuje ukazatel na případnou další třídu, Jméno třídy a ukazatel na `ast_block`, který obsahuje tělo třídy.
- Struktura `ast_block` slouží, jako tělo funkce/třídy/podmínky/cyklu i samostatně odděleného bloku. Každý blok má ukazatel na první, aktuálně zvolený a následující `ast_node`(příkaz/konstrukci) a ukazatel na nadřazený blok.
- Struktura `ast_node` obsahuje jeden příkaz/konstrukci vstupního kódu a všechny důležité informace.

- Struktura `ast_function` obsahuje název funkce, její parametry a blok s tělem funkce.
- Struktura `ast_expression` se dosti podobá struktuře `ast_node`, jelikož se jedná o kolekci identifikátorů, volání funkcí, identit, unárních a binárních výrazů. Slouží pro vytváření složitějších výrazů.
- Struktura `ast_parameter` obsahuje jméno/hodnotu parametru a odkaz na další parametr. Slouží především pro volání funkcí.
- Struktura `ast_ifj_function` slouží, pro označení speciálních vestavěných funkcí IFJ. Obsahuje jméno funkce a odkaz na první parametr.
- Struktura `ast_fun_call` je ekvivalentní struktuře `ast_ifj_function`, ale pro uživatelsky vytvořené funkce.

5.7 Generátor kódů

Generátor kódů je struktura obsahující vnitřní counter pro vytváření návěští, již popisovaný dynamický řetězec pro výstupní kód a zísobník pro ukládání zanořených návěští cyklů pro příkazy `break` a `continue`.

6 Závěr

Cílem projektu bylo vytvořit překladač pro jazyk IFJ25, který je podmnožinou jazyka Wren, do cílového jazyka IFJ25Code. Projekt zahrnuje lexikální, syntaktickou a sémantickou analýzu, generování kódu a jeho interpretaci. Hlavní část překladače byla úspěšně realizována a dokáže zpracovat základní konstrukce jazyka podle specifikace.

Výsledný překladač má nedostatky zejména v oblastech rozšířených funkcí, kde nejsou plně implementována všechna požadovaná pravidla. Problémem jsou i zanořené cykly. Klíčová funkcionalita jádra překladače však funguje spolehlivě.

