
Dokumentace k projektu z předmětů IAL a IFJ

# Implementace překladače imperativního jazyka IFJ21

Tým 020, varianta II

Mach Ondřej (vedoucí) - xmacho12 – 30%

Lán Rostislav - xlanro00 – 28%

Hnatovskyj Vítek - xhnato00 – 22%

Slivka Matej - xslivk03 – 20%

# Obsah

# Práce v týmu

## Rozdělení práce mezi členy týmu

1.1. Mach Ondřej
- Vedení týmu
- Syntaktická analýza
- Semantická analýza při běhu programu
- Generace kódu
- Integrace kódu
- Testování

1.2. Lán Rostislav
- Statická semantická analýza
- Tabulka symbolů
- Tabulka s rozptýlenými položkami
- Testování

1.3. Hnatovskyj Vítek
- Návrh FSM pro lexikální analyzátor
- Lexikální analýza
- Vestavěné funkce

1.4. Slivka Matej
- Spracovanie matematických výrazov
- Sepsání LL tabulky

1.5. Společná práce
- Dokumentace

## Vysvětlení

Kvůli nutnosti orientace v již napsaném kódu v týmu nebyla práce rozdělelna rovnoměrně. To vysvětluje rozdíl v bodovém ohodnocení členů týmu.

# Řešení projektu

## 1. Úvod

Cílem tohoto projektu bylo vytvořit v jazyce C kompletní programové řešení překladače, který vstupní zdrojový kód v jazyce IFJ21 zpracuje a přeloží do jazyka IFJcode21 .

IFJ21 je staticky typovaný jazyk na bázi jazyka Teal, IFJcode21 je nízkoúrovňový jazyk podobný assembleru.

## 2. Hlavní tělo překladače

Hlavní tělo překladače je velmi krátké. Odsud je zavolána jediná důležitá funkce – parser_run. Tato funkce řídí celý chod programu, volá funkce scanneru a generuje výstupní kód.

Jejím návratem je status datového typu Status, který je enumerací všech chyb podle zadání. Tento typ je návratem každé “high-level” funkce, ve které může docházet k chybám. Ve funkci main je podle tohoto stavu vypsána chyba, případně ohlášena úspěšná kompilace. Následně se i main ukončí s návratovou hodnotou status.

## 3. Lexikální analýza

Lexikální analyzátor je implementován v souborech `scanner.c` a `scanner.h`. Funguje na principu konečného stavového automatu (FSM), který byl nejdříve zpracován na základě grafu na Obr. 1. Pomocí funkce `scanner_get_token` se načte vstup ze stdin, zpracuje se a vrátí se jeden token. Funkce zapíše do struktury `Token`, předané parametrem jako ukazatel, všechny potřebné informace. Následně vrací stavový kód type `Status`. Ten může nabývat stavu SUCCESS, ERR_LEXIICAL, nebo ERR_INTERNAL.

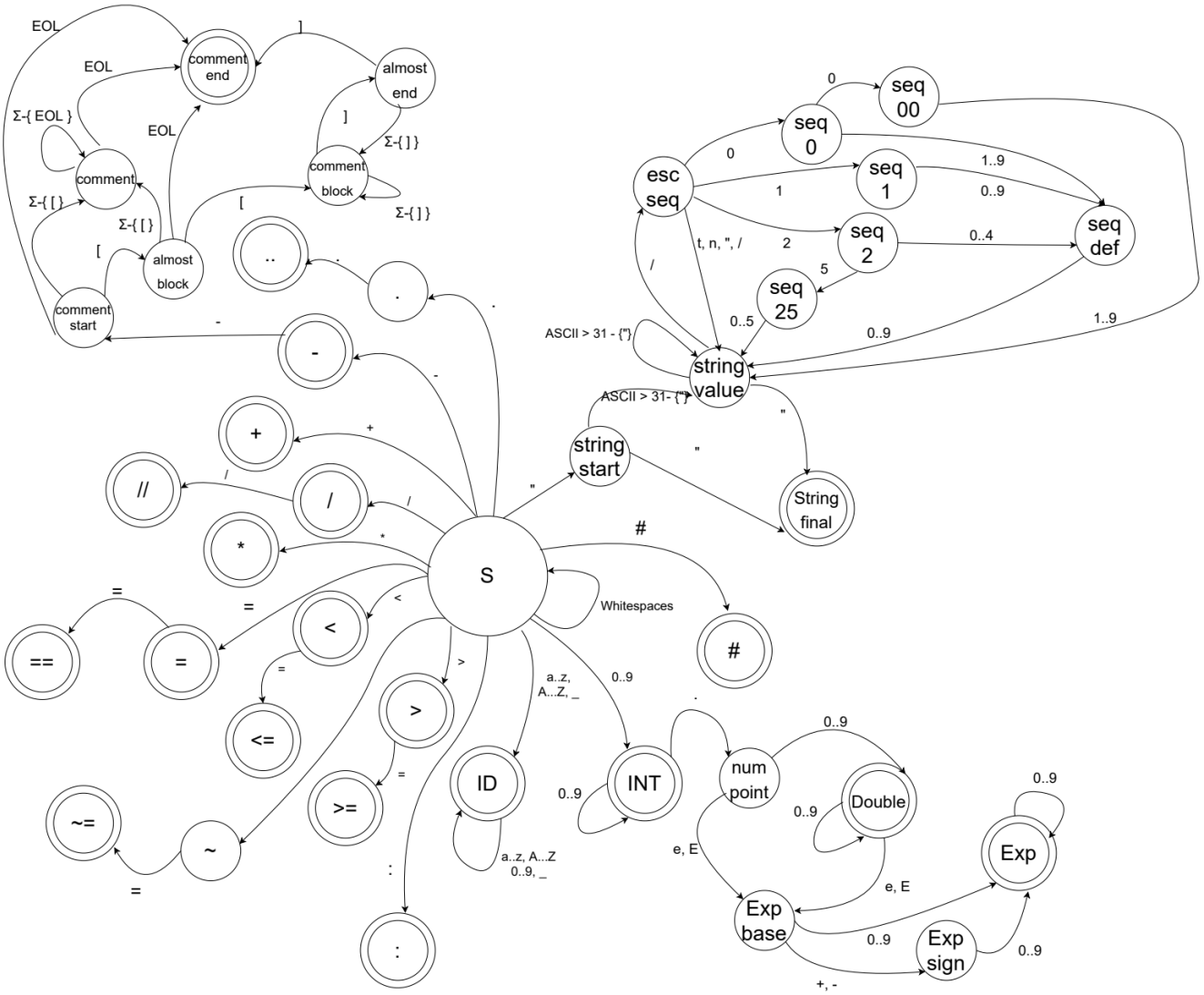
Struktura `Token` obsahuje:

- Typ tokenu.
- Jeho hodnotu uloženou jako řetězec.
- Přesnou pozici, kde se lexém nachází. Tedy číslo řádku a a počte znaků od začátku řádku, kde začíná token.

Názvy stavů FSM jsou definovány enumerací `ScannerState` a názvy typů jsou definovány v `TokenType`.

Pokud funkce `scanner_get_token` načte identifikátor, sama kontroluje, zda se nejedná o jedno z klíčových slov. Pokud ano, sama změní typ tokenu.

Obr 1. Diagram konečného automatu



# 4. Syntaktická analýza

## 4.1 Syntaktická analýza (top-down parsing)

Všechny funkce syntaktické analýzy (s vyjímkou výrazů) jsou obsaženy v souboru `parser.c`. Překlad probíhá v jednom průchodu a je zahájen voláním funkce `parser_run`. Zde se nejprve inicializují všechny moduly a datové struktury (generátor, tabulka symbolů...). Dále je zavolána funkce `nt_prog`. Nakonec jsou dealokovány datové struktury a je vypsána případná chyba.

V této implementaci syntaktické analýzy shora dolů je využita metoda rekurzivního sestupu. To v praxi znamená, že každý neterminál je přečten vlastní funkcí. Hlavním neterminálem je `<prog>`, který představuje celý validní program v jazyce IFJ21. Neterminál `<prog>` je zastoupen funkcí `nt_prog`, která simuluje jeho rozklad na `<prolog>` a `<prog_body>` vyvoláním funkcí `nt_prolog` a `nt_prog_body`. Tyto funkce volají další funkce neterminálů, dokud se nedostanou na úroveň tokenů.

Všechny funkce rekurzivního sestupu mají návrat typu `bool`, který oznamuje, zda byl daný neterminál nalezen. Pokud kterákoli z těchto funkcí skončí s návratem `false`, její volající funkce je také okamžitě ukončena s návratem `false`. Pokud funkce `nt_prog` vrátí hodnotu `false`, znamená to, že nebyl přečten validní program. Překlad je tak ukončen syntaktickou chybou.

V těchto funkcích jsou kromě syntaktických pravidel obsaženy i sémantické kontroly a generování výstupního kódu. Parametry nejsou nijak ucelené a jsou používany zejména pro sémantickou analýzu.

## 4.2 Spracovanie výrazov (bottom-up parsing)

Výrazy sa spracúvajú oddelene spracovanej syntaxe a to precedenčnou analýzou. Syntaktická kontrola, sémantická kontrola, generácia výsledného kódu prebieha v súbore `expression.c`.

Spracovanie výrazov je zavolané akonáhle parser nájde výraz. Funkcia pracuje s tokenom ktorý ukazuje na začiatok výrazu. Po ukončení svojej práce vráti hodnotu true/false podľa toho či je matematický výraz správny (`true` - je správny, `false` - nie je správny), funkcia vygeneruje IFJcode21 a token bude ukazovať na lexému za matematickým výrazom.

Symboly sú postupne spracované pomocou precedenčnej analýzy a to tak, že sú uložené na vrchol zásobníku `SymStack` s ktorý sa dokáže dynamický zväčšiť. Symbol ktorý je uložený na zásobník sa skladá z tokenu, ktorý vracia lexikálna analýza, z `Type`-u ktorý určuje o aký dátový typ sa jedná (v prípade hodnôt), z SymbolType-u ktorý slúži na orientáciu v precedenčnej tabuľke.

Hlavná funkcia inicializuje zásobník a v cykle načíta tokeny ktoré spracované na zásobník.

- Ak precedenčná tabuľka vráti znak < tak je za vrchný terminál uložený handle a na vrchol zásobníku je uložený spracovaný token.
- Ak precedenčná tabuľka vráti znak „=“ tak je na vrchol zásobníku uložený spracovaný token.
- Ak precedenčná tabuľka vráti znak „>“ tak sa spustí funkcia `symstack_reduce` ktorá skontroluje ,že či sa medzi symbolmi „<“ a ”>” nachádza validný výraz.

`Symstack_reduce` prechádza spísané pravidla podľa ktorých sa rozhodne ako zredukuje daný výraz. Po úspešnom zredukovaní je celý výraz na zásobniku nahradený `Symbolom` typu `S_EXPR` , ktorý značí validný matematický výraz. Funkcia v tomto prípade vratí `true`, inak vracia `false`.

Pri redukcii probieha sémantická kontrola typov operandov, prípadné pretypovanie čísiel a generácia kódu.

Cyklus sa opakuje pokiaľ nie je na vrchny terminal na zásobníku $ a na vstupe nie je $. Potom sa funkcia ukončí a pokračuje hlavný parser.

## 4.3 Precedenčná tabulka

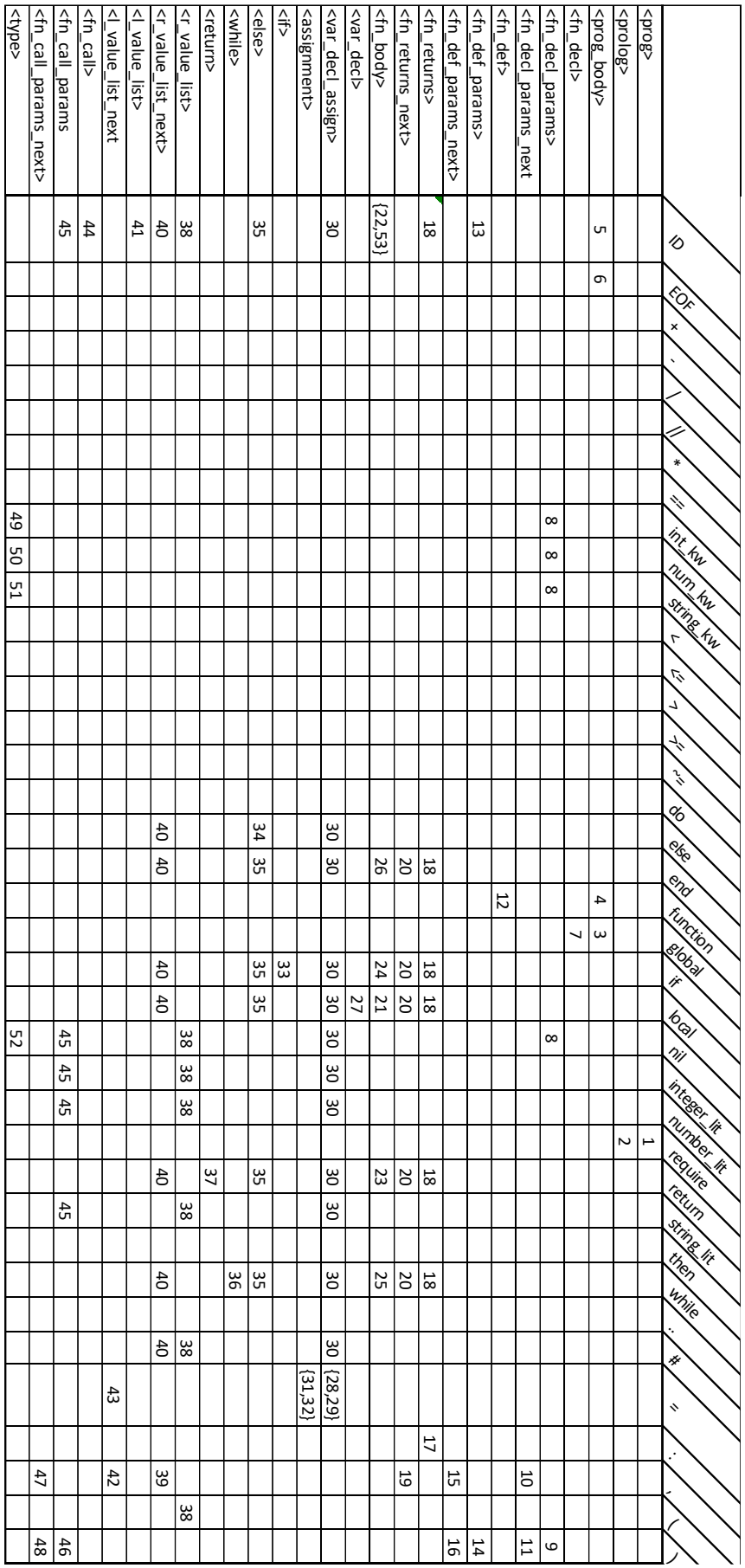
| | + , - | /,*,// | ( | ) | i | >,<,>=,<= | ==, != | # | $ | .. |
|---|---|---|---|---|---|---|---|---|---|---|
| +,- | > | < | < | > | < | > | > | < | > | > |
| /,*,// | > | > | < | > | < | > | > | < | > | > |
| ( | < | < | < | = | < | < | < | < | | < |
| ) | > | > | | > | | > | > | | > | > |
| I | > | > | | > | | > | > | | > | > |
| >,<,>=,<= | < | < | < | > | < | > | > | < | > | < |
| ==, != | < | < | < | > | < | < | > | < | > | < |
| # | > | > | < | > | > | > | < | > | > | > |
| $ | < | < | < | | < | < | < | < | | < |
| .. | < | < | < | > | < | > | > | < | > | < |

## 4.4 Gramatická pravidla

1. <prog> -> <prolog> <prog_body>
2. <prolog> -> require string
3. <prog_body> -> <fn_decl> <prog_body>
4. <prog_body> -> <fn_def> <prog_body>
5. <prog_body> -> <fn_call> <prog_body>
6. <prog_body> -> eps
7. <fn_decl> -> global ID : function (<fn_decl_pams>) <fn_returns>
8. <fn_decl_params> -> <type> <fn_decl_params_next>
9. <fn_decl_params> -> eps
10. <fn_decl_params_next> -> , <type> <fn_decl_params_next>
11. <fn_decl_params_next> ->eps
12. <fn_def> -> function ID (<fn_def_params>) <fn_returns> <fn_body> END
13. <fn_def_params> -> ID : <type> <fn_def_params_next>
14. <fn_def_params> -> eps
15. <fn_def_params_next> -> , ID : <type> <fn_decl_params_next>
16. <fn_def_params_next> -> eps
17. <fn_returns> -> : <type> <fn_returns_next>
18. <fn_returns> ->eps
19. <fn_returns_next> -> , <type> <fn_returns_next>
20. <fn_returns_next> -> eps
21. <fn_body> -> local <var_decl> <fn_body>
22. <fn_body> -> <assignment> <fn_body>
23. <fn_body> -> return <return> <fn_body>
24. <fn_body> -> if <if> <fn_body>
25. <fn_body> -> while <while> <fn_body>
26. <fn_body> -> eps
27. <var_decl> -> local ID : <var_decl_assign>
28. <var_decl_assign> -> = <fn_call>
29. <var_decl_assign> -> = <expr>
30. <var_decl_assign> -> eps
31. <assignment> -> <l_value_list> = <fn_call>

32. <assignment> -> <l_value_list> = <r_value_list>
33. <if> -> if <expr> then <fn_body> <else> END
34. <else> -> else <fn_body>
35. <else> -> eps
36. <while> -> while <expr> do <fn_body> end
37. <return> -> return <r_value_list>
38. <r_value_list> -> <expr> <r_value_list_next>
39. <r_value_list_next> -> , <expr> <r_value_list_next>
40. <r_value_list_next> -> eps
41. <l_value_list> -> ID <l_value_list_next>
42. <l_value_list_next> -> , ID <l_value_list_next>
43. <l_value_list_next> -> eps
44. <fn_call> -> ID ( <fn_call_params> )
45. <fn_call_params> -> <expr> <fn_call_params_next>
46. <fn_call_params> -> eps
47. <fn_call_params_next> -> , <expr> <fn_call _params_next>
48. <fn_call_params_next> -> eps
49. <type> -> integer_kw
50. <type> -> number_kw
51. <type> -> string_kw
52. <type> -> nil
53. <fn_body> -> <fn_call> <fn_body>

## 4.5 LL tabulka



| | ID | EOF | + | - | / | // | * | == | int_kw | num_kw | string_kw | < | <= | > | >= | ~= | do | else | end | function | global | if | local | nil | integer_lit | number_lit | require | return | string_lit | then | while | .. | # | = | : | , | ( | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <prog> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <prolog> | | | | | | | | | | | | | | | | | | | | | | | | | | 1 | | | | | | | | | | | | |
| <prog_body> | 5 | 6 | | | | | | | | | | | | | | | | | 4 | 3 | | | | | | 2 | | | | | | | | | | | |
| <fn_decl> | | | | | | | | | | | | | | | | | | | | 7 | | | | | | | | | | | | | | | | | |
| <fn_decl_params> | | | | | | | | 8 | 8 | 8 | | | | | | | | | | | | | 8 | | | | | | | | | | | | | | |
| <fn_decl_params_next | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 10 | | 11 |
| <fn_def> | | | | | | | | | | | | | | | | | | | 12 | | | | | | | | | | | | | | | | | | | |
| <fn_def_params> | 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 14 | |
| <fn_def_params_next> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 15 | | 16 |
| <fn_returns> | 18 | | | | | | | | | | | | | | | | | 18 | | | | 18 | 18 | | | | | | | | | | | | 17 | | | |
| <fn_returns_next> | | | | | | | | | | | | | | | | | | 20 | | | | 20 | 20 | | | | | | | | | | | | | 19 | | |
| <fn_body> | {22,53} | | | | | | | | | | | | | | | | | 26 | | | | 24 | 21 | | | | | | | | | | | | | | | |
| <var_decl> | | | | | | | | | | | | | | | | | | | | | | | 27 | | | | | | | | | | | | | | | |
| <var_decl_assign> | 30 | | | | | | | | | | | | | | | 30 | 30 | | | | | 30 | 30 | 30 | 30 | | 30 | 30 | | 30 | | | | | | | |
| <assignment> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <if> | | | | | | | | | | | | | | | | | | | | | | 33 | | | | | | | | | | | | | | | | |
| <else> | 35 | | | | | | | | | | | | | | | 34 | 35 | | | | | 35 | 35 | 38 | 38 | | 35 | 35 | | 35 | | | | | | | |
| <while> | | | | | | | | | | | | | | | | | | | | | | | | 38 | 38 | | | 37 | | | 36 | | | | | | |
| <return> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <r_value_list> | 38 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <r_value_list_next> | 40 | | | | | | | | | | | | | | | 40 | 40 | | | | | 40 | 40 | | | | | | | | | | | | | | | |
| <l_value_list> | 41 | | | | | | | | | | | | | | | | | | | | | | | | | | | 40 | | | | | | | | | | |
| <l_value_list_next | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <fn_call> | 44 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <fn_call_params | 45 | | | | | | | | | | | | | | | | | | | | | | | 45 | 45 | | | | 45 | | | | | | | | | |
| <fn_call_params_next> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 47 | | |
| <type> | | | | | | | | 49 | 50 | 51 | | | | | | | | | | | | | | | | | | | | | | | | | | | 48 | |

# 5. Sémantická analýza

Statická sémantická analýza je prováděna v modulu `parser.c` s využitím pomocných funkcí a struktur z modulů `types.c`, pro práci s datovými typy a seznamy z jazyka ifj21 a `symtable.c`, pro práci s tabulkou s rozptýlenými položkami - dále jen hašovací tabulkou.

## 5.1 Využité datové struktury

Pro účely sémantické analýzy bylo potřeba několik pomocných struktur. Ve funkcích rekurzivního sestupu, kde se vyskytují identifikátory jsou využity funkce pro práci s tabulkou symbolů. Při deklaraci / definici funkce, nebo deklaraci proměnné je identifikátor přidán do tabulky symbolů. V naší implementaci jsou funkce i proměnné v jedné tabulce.

Tabulka symbolů je struktura `SymTab`, která obsahuje seznam hašovacích tabulek a ukazatel na hašovací tabulku, která se nachází na jeho začátku. Každá tabulka představuje rozsah platnosti proměnných, popř. funkcí.

Hašovací tabulka je implementována jako struktura `Htab`, obsahující pole, do kterého jsou položky rozptýleny, jeho velikost a ukazatel na další hašovací tabulku.

Položky hashovací tabulky `HtabItem` jsou struktury, ve kterých je obsažena struktura `HTabPair` s informacemi o samotné funkci / proměnné a ukazatel na další položku, pokud by dvě nebo více položek bylo hašovací funkcí přiřazeno na stejný `index`.

Struktura `HtabPair` se skládá z dvojice `key`, což je název proměnné / funkce a struktury `HTabValue`, ve které jsou již samotná data. Pro funkce jsou v této struktuře tyto složky :

- `Defined` - boolovská hodnota, určuje, zda byla funkce dříve definována
- `paramList` - seznam uchovávající typy vstupních parametrů
- `returnList` - seznam uchovávající typy výstupních parametrů
- `specialFn` - boolovská hodnota pro identifikaci builtin funkce write

U proměnných jsou zde používány složky:

- `ID` - integer, unikátní identifikátor proměnných při generaci kódu
- `varType` - pro určení datového typu proměnné

V modulu `types.c` jsou obsaženy dvě datové struktury. Jednosměrně vázaný seznam `TypeList`, jehož typu jsou výše zmíněné proměnné `paramList` a `returnList`. Seznam `TypeList` je tvořen položkami `TypeListItem`. Ten obsahuje datový typ položky a odkaz na další.

# 6. Generace kódu

Výsledný kód je generovanán přímo ve funkcích rekurzivního sestupu a funkcích pro redukci výrazů. Funkce, které se týkají generování kódu jsou shromážděny v modulu `generator.c`.
Před generováním kódu je třeba nastavit výstupní soubor a nastavit vnitřní stav generátoru. O toto se stará funkce `gen_init`. Přesně opačný efekt má funkce `gen_destroy`, která na konci generování dealokuje zdroje.

Nejdůležitější funkce jsou `gen_print` a `gen_prepend`, které jsou v podstatě wrapper pro `fprintf`.

Po inicializaci scanneru se `gen_print` a `gen_prepend` chovají stejně, a to tak, že svůj vstup přímo vypisují na výstup. Pokud ale zavoláme funkci `gen_buffer_start`, pak `gen_print` začne ukládat svůj výstup do vyrovnávací paměti. Toto je naše ošetření pro deklarace proměnných uvnitř cyklu while. Na všechy běžné instrukce je použita funkce `gen_print`, zatímco instrukce `DEFVAR` je přímo poslána na výstup pomocí `gen_prepend`. Při zavolání funkce `gen_buffer_stop` je obsah vyrovnávací paměti vypsán na výstup a `gen_print` se opět chová obvyklým způsobem.

Modul `generator.c` obsahuje pomocné funkce na vypisování často používaných věcí. Jednou z nich je `gen_new_label`, která je použita na generování unikátních čísel pro návěští. Tato návěští jsou používána u if podmínek, while cyklů nebo běhových sémantických kontrol. Další pomocnou funkcí je `gen_print_value`, která dokáže z tokenu literálu vygenerovat hodnotu v cílovém jazyce.

Funkce ve vygenerovaném kódu mají stejná jména jako ve zdrojovém kódu. Argumenty dostávají na tzv. Dočasném rámci (`TF`) pod jmény `%param0, %param1` atd. Návraty funkcí jsou předávány přes zasobník, a to v takovém pořadí, že poslední návrat bude na vrcholu zásobníku.

Všechny lokální proměnné funkcí jsou přejmenovány na `$ID.`Tento unikátní identifikátor je jim přiřazen v tabulce symbolů, aby nedocházelo ke kolizi stejnojmenných proměnných.

Vyhodnocování výrazů také probíhá zejména na zásobníku. Zde se také dělá velká část běhových kontrol. Pro účel běhových kontrol jsou zavedené proměnné `a, b, c, d` na globálním rámci (`GF`). S nimi je nakládáno podobně, jako kdyby šlo o registry.

Integrované funkce (s výjimkou `write`) jsou napsány ručně a připojeny na začátek každého vygenerováného kódu ze souboru `include.c`. Vlastnosti těchto funkcí jsou také uloženy do tabulky symbolů, aby překladač mohl správně reagovat na jejich signaturu. Zde se nachází problém s funkcí `write`, která nemá statické typy parametrů, a jejich počet je neomezený. Tato funkce zcela obchází volání, kód pro vypsání je vypsán "na místě" pro každý její argument.

# 7. Organizace projektu

## 7.1 Vývojové prostředí

Pro psaní projektu bylo využito integrované prostředí CLion. Toto IDE podporuje GNU Make, který byl požadavkem na projekt. Také má velmi dobrou integraci s verzovacím systémem git. Všechen vývoj byl prováděn na operačních systémech GNU/Linux.

CLion se později ukázal být skvělou volbou, a to hlavně kvůli debugování a refaktorování. Často jsme naráželi na problémy typu nefunkční sémantické kontroly apod. CLion umožnil relativně rychle spustit debugování a zobrazit všechny lokální proměnné. Toto výrazně urychlilo vývoj oproti více tradičnímu gdb z příkazové řádky.

Refaktorování se výrazně uplatnilo při přidávání parametrů do funkcí rekurzivního sestupu. Ty byly potřeba na předávání atributů sémantické analýzy. Funkce na refaktorování ušetřily spoustu času tím, že automaticky opravily argumenty v každém výskytu funkce.

## 7.2 Verzovací systém

Pro kontrolu verzí byl použit verzovací systém git s hostingem na platformě Github. Všichni členové týmu pracovali na větvi main. Díky malému rozsahu projektu nebyl problém s konflikty. Pro větší projekt bychom ale určitě použili oddělené větve pro vývoj každé součásti zvlášť. Github nabízí několik speciálních funkcí, jako zobrazení aktivity každého člena atd. Těchto jsme aktivně nevyužívali, ale opět mohou být užitečné při vedení většího projektu.

## 7.3 Komunikace

Pro komunikaci jsme zvolili vlastní server na Discordu kvůli jeho praktičnosti .
S rychlostí komunikace problémy nebyly, ale pro vystvětlení složitějších konceptů by se hodilo více osobního kontaktu. Kvůli tomu byla integrace oddělených částí poměrně složitá a vyžadovala větší zásahy do kódu. Kdyby měla tato situace nastat znova, bylo by vhodné alespoň začátek každého modulu napsat společně. Díky tomu by mělo více členů přehled o celkové architektuře programu a integrace by nebyla tak problémová.

## 7.4 Rozdělení práce

Rozdělení úkolů v týmů také nebylo jednoduché, a to hlavně kvůli provázání syntaktického analyzátoru se sémantickým analyzátorem a generováním kódu. Většina těchto aktivit je shromážděna v jediném modulu parser.c. Pro implementaci sémantických kontrol se ukázala být velmi efektivní technika "párového programování". Pro jejich doplnění je potřeba časté vysvětlování struktury parseru.