Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentácia k projektu IFJ/IAL

Implementácia prekladača imperatívneho jazyka IFJ25

Tím xmicham00, varianta vv-BVS

Implementované rozšírenia: žiadne

Martin Michálik	(xmicham00)	25%	[Vedúci]
Matúš Magyar	(xmagyam00)	25%	
Šimon Škoda	(xskodas00)	25%	
Jaroslav Vrbiniak	(xvrbinj00)	25%	

Brno, 2. decembra 2025

Obsah

1 Úvod

Táto dokumentácia popisuje návrh a implementáciu prekladača jazyka IFJ25 do medzikódu IFJcode25. Cieľom projektu bolo vytvoriť funkčný prekladač, ktorý spracúva zdrojový kód v jazyku IFJ25 a generuje ekvivalentný program v jazyku IFJcode25.

Prekladač sa skladá z niekoľkých na seba nadväzujúcich častí: lexikálny analyzátor (scanner), syntaktický analyzátor (parser) využívajúci metódu rekurzívneho zostupu, precedenčný syntaktický analyzátor pre výrazy, sémantický analyzátor a generátor cieľového kódu.

Projekt bol implementovaný v jazyku C a vyvíjaný v termíne od 14.10.2025 do 03.12.2025.

2 Architektúra projektu

Zdrojové súbory v projekte sú rozdelené podľa funkcie:

- `ast.c/h` – abstraktný syntaktický strom
- `builtins.c/h` – spracovanie vstavaných funkcií
- `errors.c/h` – správa chybových kódov
- `expression.c/h` – precedenčná analýza výrazov
- `generator.c/h` – generovanie IFJcode25
- `main.c` – vstupný bod programu
- `parser.c/h` – syntaktická analýza a riadenie prekladu
- `scanner.c/h` – lexikálna analýza
- `symtable.c/h` – tabuľka symbolov (AVL strom)

3 Lexikálna analýza

Modul `scanner.c` implementuje lexikálny analyzátor ako deterministický konečný automat (DKA). Hlavná funkcia `get_next_token()` používa veľký `switch` na vetvenie podľa aktuálneho znaku, čo zodpovedá stavom DKA.

3.1 Kľúčové implementácie

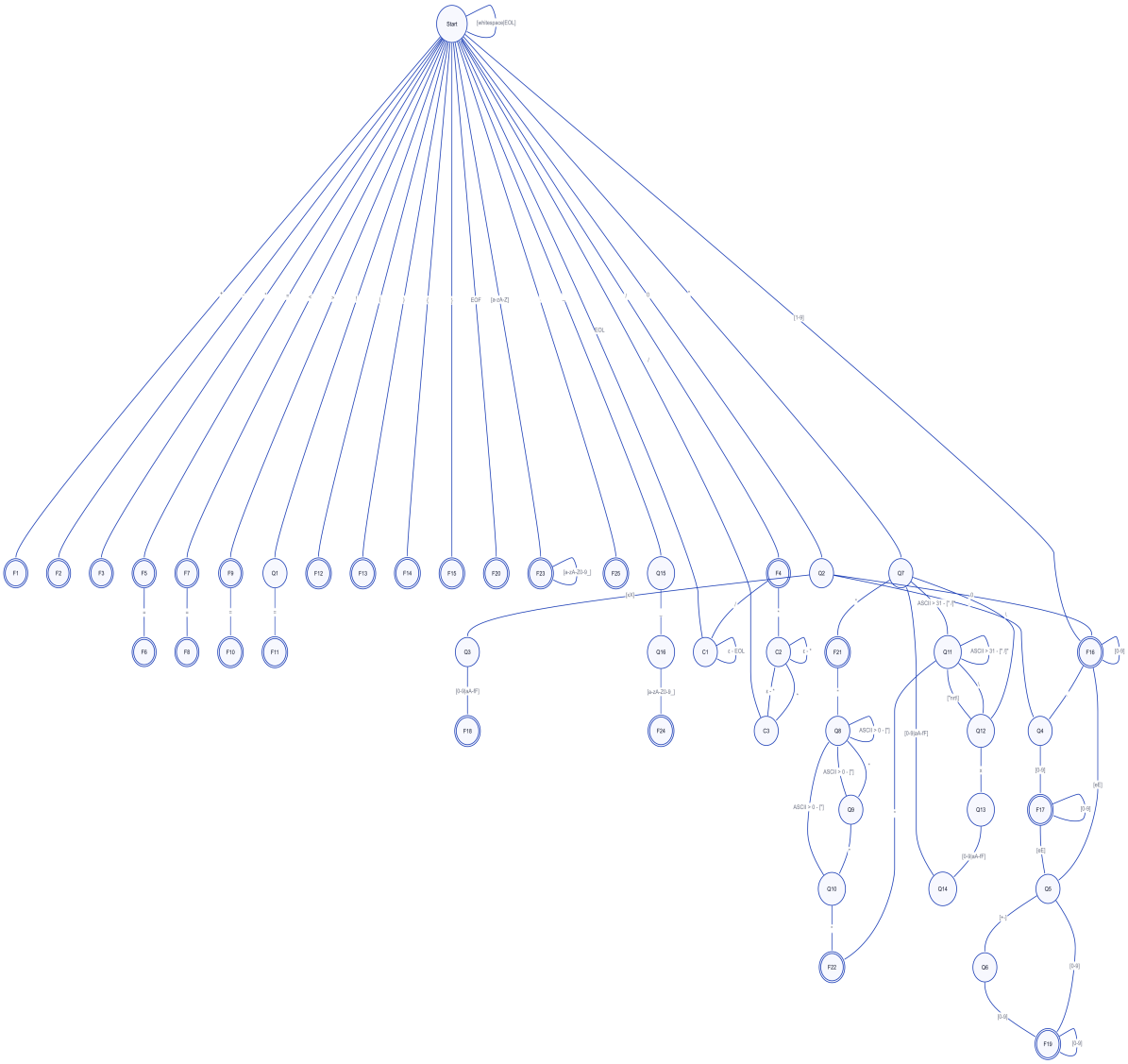
Pri spracovaní reťazcových literálov používame dynamický buffer s počiatočnou kapacitou 256 bajtov, ktorý sa automaticky zdvojnásobuje pri preplnení. Táto technika zabezpečuje efektívne spracovanie aj veľmi dlhých reťazcov bez plytvenia pamäťou.

Multi-line stringy sú rozpoznávané hľadaním trojice `"` – scanner najprv rozpozná bežný string, a ak začína ďalším `"`, prechádza do režimu viacriadkového reťazca. V tomto režime sa znaky ukladajú priamo do bufferu až kým sa nenájde ukončovacia trojica.

Vnorené blokové komentáre sú implementované rekurzívne pomocou funkcie `ignore_multiline_comment()`. Pri nájdení `/*` vo vnútri komentára sa funkcia zavolá rekurzívne, čo automaticky spracuje ľubovoľne hlboké vnorenie.

Hexadecimálne čísla rozpoznávame po prefixi `0x` alebo `0X` pomocou funkcie `strtoull()` so základom 16. Exponenciálny zápis je rozpoznaný pri výskyte znaku `e`/`E` a výsledok sa vypočítava pomocou `powf()`/`pow()`.

3.2 Diagram konečného automatu



Obr. 1: Diagram konečného automatu (FSM) použitý v lexikálnej analýze.

4 Syntaktická analýza

Parser je implementovaný v module `parser.c` metódou rekurzívneho zostupu.

4.1 Správa tokenov

Parser obsahuje mechanizmus `putback_token()`, ktorý umožňuje vrátiť jeden token späť do prúdu. Toto je nevyhnutné v situáciách, keď potrebujeme „nahliadnuť dopredu“ na nasledujúci token bez jeho konzumovania.

4.2 Name mangling

Pre podporu preťažovania funkcií (rovnaké meno, rôzny počet parametrov) používame *name mangling*. Funkcia `foo` s dvoma parametrami sa interne ukladá ako `foo$arity2`. Podobne gettery a settery používajú sufixy `$get` a `$set`. Tento prístup umožňuje jednoznačnú identifikáciu každej varianty funkcie v tabuľke symbolov.

4.3 Odložená validácia volaní

Volania funkcií sa zaznamenávajú do linked listu `function_calls_list` obsahujúceho názov, aritu a číslo riadku. Samotná validácia prebieha až po spracovaní celého programu funkciou `validate_function_calls()`. To umožňuje volať funkcie, ktoré sú definované až neskôr v kóde.

4.4 Spracovanie výrazov

Modul `expression.c` implementuje precedenčnú analýzu pomocou zásobníkovej metódy. Precedenčná tabuľka 15x15 definuje vzťahy medzi operátormi pomocou symbolov `<` (posun), `>` (redukcia), `=` (rovnosť) a prázdne pole (chyba). Analýza konštruuje abstraktný syntaktický strom (AST), ktorý sa následne použije pre generovanie kódu.

4.4.1 Precedenčná tabuľka

	(	)	i	+	-	*	/	<	<=	>	>=	==	!=	is	$
(	<	=	<	<	<	<	<	<	<	<	<	<	<	<	
)		>		>	>	>	>	>	>	>	>	>	>	>	>
i		>		>	>	>	>	>	>	>	>	>	>	>	>
+	<	>	<	>	>	<	<	>	>	>	>	>	>	>	>
-	<	>	<	>	>	<	<	>	>	>	>	>	>	>	>
*	<	>	<	>	>	>	>	>	>	>	>	>	>	>	>
/	<	>	<	>	>	>	>	>	>	>	>	>	>	>	>
<	<	>	<	<	<	<	<								>
<=	<	>	<	<	<	<	<								>
>	<	>	<	<	<	<	<								>
>=	<	>	<	<	<	<	<								>
==	<	>	<	<	<	<	<								>
!=	<	>	<	<	<	<	<								>
is	<	>	<	<	<	<	<								>
$	<		<	<	<	<	<	<	<	<	<	<	<	<	

Tabuľka 1: Precedenčná tabuľka pre výrazy (i = identifikátor/literál, $ = koniec výrazu)

Priamo počas redukcie sa vykonáva typová kontrola literálov – funkcia `check_binary_literal_types()` overuje kompatibilitu typov pre binárne operácie.

4.5 LL gramatika

Pre syntaktickú analýzu jazykových konštrukcií bola navrhnutá LL(1) gramatika. Gramatika obsahuje 24 neterminálov a 56 pravidiel:

```
1.  <prolog>          -> import STRING for Ifj EOL <eols>
2.  <program>         -> class IDENTIFIER { EOL <func_list> }
3.  <main_func>       -> static IDENTIFIER ( ) <block>
4.  <func_list>       -> }
5.  <func_list>       -> <func> <func_list>
```

6. <func> -> static IDENTIFIER (<param_list>) <block>
7. <func> -> static IDENTIFIER { <block>
8. <func> -> static IDENTIFIER = (IDENTIFIER) <block>
9. <param_list> ->)
10. <param_list> -> IDENTIFIER <param_list_tail>
11. <param_list_tail>->)
12. <param_list_tail>-> , IDENTIFIER <param_list_tail>
13. <block> -> { EOL <stmt_list> }
14. <stmt_list> -> }
15. <stmt_list> -> <stmt> EOL <stmt_list>
16. <stmt> -> <var_decl>
17. <stmt> -> <assign>
18. <stmt> -> <if_stmt>
19. <stmt> -> <while_stmt>
20. <stmt> -> <return_stmt>
21. <stmt> -> <func_call>
22. <stmt> -> <block>
23. <var_decl> -> var IDENTIFIER EOL
24. <assign> -> IDENTIFIER = <expr>
25. <assign> -> IDENTIFIER . IDENTIFIER = <expr>
26. <if_stmt> -> if (<expr>) <block> else <block>
27. <while_stmt> -> while (<expr>) <block>
28. <return_stmt> -> return <expr>
29. <func_call> -> IDENTIFIER (<arg_list>)
30. <func_call> -> Ifj . IDENTIFIER (<arg_list>)
31. <arg_list> ->)
32. <arg_list> -> <expr> <arg_list_tail>
33. <arg_list_tail> ->)
34. <arg_list_tail> -> , <expr> <arg_list_tail>
35. <expr> -> <term> <expr_cont>
36. <expr_cont> -> eps
37. <expr_cont> -> <op> <expr>
38. <term> -> IDENTIFIER
39. <term> -> INT_LITERAL
40. <term> -> FLOAT_LITERAL
41. <term> -> STRING_LITERAL
42. <term> -> null
43. <term> -> (<expr>)
44. <op> -> +
45. <op> -> -
46. <op> -> *
47. <op> -> /
48. <op> -> ==
49. <op> -> !=
50. <op> -> <
51. <op> -> <=
52. <op> -> >
53. <op> -> >=

```
54. <op>            -> is
55. <eols>          -> EOF
56. <eols>          -> EOL <eols>
```

4.6 LL tabuľka

Rozkladová tabuľka pre LL(1) analýzu. Bunky obsahujú čísla pravidiel gramatiky, prázdne bunky znamenajú syntaktickú chybu.

Neterminál	(	)	,	EOF	EOL	FLT	ID	INT	Ifj	Null	/	==	is	+	STR	cls	if	ret	sta	var	whi	{
arg_list	32	31				32	32	32		32					32							
arg_list_tail		33	34																			
assign							24															
block																						13
eols				55	56																	
expr	35					35	35	35		35					35							
expr_cont		36	36		36						37	37	37	37								
func																			6			
func_call							29															
func_list																			5			4
if_stmt																	26					
main_func																			3			
op											47	48	54	44								
param_list		9					10															
param_list_tail		11	12																			
program																2						
return_stmt																		28				
stmt							17										18	20		16	19	
stmt_list							15										15	15		15	15	14
term	43					40	38	39		42					41							
var_decl																				23		
while_stmt																					27	

Tabuľka 2: LL(1) rozkladová tabuľka (skrátené: ID=IDENTIFIER, FLT=FLOAT_LIT, INT=INT_LIT, STR=STRING_LIT, cls=class, ret=return, sta=static, whi=while)

Kompletná LL tabuľka obsahuje 24 neterminálov a 35 terminálov. Pre operátory (`op`) tabuľka obsahuje pravidlá 44–54 pre jednotlivé operátory (+, -, *, /, ==, !=, <, <=, >, >=, is).

5 Tabuľka symbolov

Tabuľka symbolov je kľúčová dátová štruktúra, ktorá uchováva informácie o všetkých identifikátoroch (premenných, funkciách) počas prekladu.

5.1 Implementácia AVL stromom

Podľa požiadaviek predmetu IAL sme tabuľku symbolov implementovali ako AVL strom v module `symtable.c`. Použitie samovyvažujúceho binárneho vyhľadávacieho stromu garantuje logaritmickú časovú zložitosť $O(\log n)$ pre operácie vkladania, vyhľadávania aj mazania.

Vyváženosť stromu je zabezpečená rotáciami po každom vložení:

- `far_left()` – LL rotácia (pravá rotácia)
- `far_right()` – RR rotácia (ľavá rotácia)
- `left_right()` – LR rotácia (dvojitá)
- `right_left()` – RL rotácia (dvojitá)

5.2 Ukladané informácie

Každý uzol stromu obsahuje:

- **Kľúč** – názov symbolu (pre funkcie manglované meno)
- **Typ symbolu** – premenná lokálna/globálna, funkcia, getter, setter
- **Pre premenné:** `block_id`, `nesting_level`, príznak inicializácie
- **Pre funkcie:** počet parametrov, typy parametrov, návratový typ

5.3 Správa rozsahov platnosti

Pre správnu implementáciu rozsahov platnosti (scopes) používame zásobníkový systém blokov. Každý blok (`if`, `while`, vnútorné bloky) dostáva unikátne `block_id` z počítadla. Lokálne premenné sa interne premenúvajú na tvar `meno$bN`, kde N je block ID, čo automaticky rieši shadowing premenných.

Funkcia `symtable_search_var_scoped()` prehľadáva premenné od aktuálneho bloku smerom k vonkajším blokom, čo zabezpečuje správne nájdenie najvnútornejšej viditeľnej definície.

6 Sémantická analýza

Sémantická analýza prebieha integrovane so syntaktickou analýzou a vykonáva nasledujúce kontroly:

- Kontrola definície premenných a funkcií pred ich použitím
- Kontrola redefinícií v rovnakom rozsahu platnosti
- Kontrola počtu parametrov pri volaní funkcií
- Typová kontrola literálov vo výrazoch

6.1 Odložená validácia funkcií

Volania funkcií sa zaznamenávajú do linked listu a validujú sa až po spracovaní celého programu. To umožňuje volať funkcie definované neskôr v kóde.

6.2 Sledovanie premenných funkcie

Pre potreby generovania DEFVAR inštrukcií na začiatku funkcie si udržiavame zoznam všetkých premenných použitých vo funkcii v poli `ifj_function_vars`.

7 Generovanie IFJcode25

Modul `generator.c` generuje kód syntaxou riadeným prekladom – kód sa produkuje priamo počas parsovania.

7.1 Systém bufferov

Implementovali sme trojitý buffrovací systém pomocou `open_memstream()`:

1. **user_functions_buffer** – zhromažďuje kód všetkých užívateľských funkcií (okrem main)
2. **main_function_buffer** – osobitne uchováva telo main funkcie
3. **function_body_buffer** – dočasný buffer pre aktuálne spracovávanú funkciu

Tento prístup umožňuje vygenerovať výstup v správnom poradí: najprv hlavička so skokom na main, potom vstavané funkcie, užívateľské funkcie a nakoniec main.

7.2 Odložené DEFVAR

IFJcode25 vyžaduje, aby DEFVAR inštrukcie pre lokálne premenné boli na začiatku funkcie. Keďže premenné môžu byť deklarované kdekoľvek v tele funkcie, zbierame ich do zoznamu `ifj_function_vars` a DEFVAR generujeme až po spracovaní celého tela funkcie pomocou `generate_all_function_defvars()`.

7.3 Escaping reťazcov

Funkcia `convert_to_escaped_string()` konvertuje reťazce do formátu vyžadovaného IFJcode25. Znaky s ASCII hodnotou ≤ 32, mriežka (#) a spätné lomítko sa konvertujú na \xyz formát (trojciferný ASCII kód).

7.4 Polymorfné operátory

Operátory + a * sú polymorfné – ich správanie závisí od typov operandov. Generujeme runtime kontroly pomocou inštrukcie `TYPE`, ktoré zisťujú typy operandov na zásobníku a vetvia výpočet na správnu variantu (sčítanie vs. konkatenácia, násobenie vs. opakovanie reťazca).

7.5 Labely a rámce

Počítadlo `ifj_label_counter` generuje unikátne návestia pre riadiace štruktúry. Funkcie používajú CREATEFRAME/PUSHFRAME pre vytvorenie lokálneho rámca. Parametre sa vyberajú zo zásobníku v opačnom poradí (posledný vložený = prvý vybraný).

8 Chybové hlásenia

Prekladač vracia nasledujúce návratové kódy chýb:

Kód	Popis chyby
1	Chyba v rámci lexikálnej analýzy (ERR_LEXICAL)
2	Chyba v rámci syntaktickej analýzy (ERR_SYNTAX)
3	Nedefinovaná funkcia alebo premenná (ERR_SEM_UNDEF)
4	Redefinícia funkcie alebo premennej (ERR_SEM_REDEF)
5	Nesprávny počet parametrov/typ vstavanej f. (ERR_SEM_PARAMS)
6	Chyba typovej kompatibility vo výrazoch (ERR_SEM_TYPE_COMPAT)
10	Ostatné sémantické chyby (ERR_SEM_OTHER)
25	Behová chyba – zlý typ parametru (ERR_RUNTIME_PARAM_TYPE)
26	Behová chyba – typová kompatibilita (ERR_RUNTIME_TYPE_COMPAT)
99	Interná chyba prekladača (ERR_INTERNAL)

Tabuľka 3: Tabuľka návratových kódov

9 Testovanie

Súčasťou projektu sú testovacie súbory `scanner_test.c` a `symtable_test.c`. Tieto súbory boli použité na testovanie pre pozitívne testy syntakticky správnych programov a negatívne testy lexikálnych, syntaktických a sémantických chýb. Samozrejme, toto nebolo jediné testovanie – testovanie prebiehalo aj mimo týchto súborov počas celej doby práce na projekte.

10 Práca v tíme

Projekt bol vyvíjaný v tíme štyroch členov. Na začiatku sme si stanovili rozdelenie úloh podľa vertikálneho prístupu, kde každý člen zodpovedá za určitú časť prekladača.

10.1 Rozdelenie úloh

- **Martin Michálik** – vedenie projektu, generátor kódu, parser, koordinácia
- **Matúš Magyar** – parser, generator, testovanie, dokumentácia
- **Šimon Škoda** – lexikálny analyzátor, tabuľka symbolov, precedenčná analýza
- **Jaroslav Vrbiniak** – tabuľka symbolov, testovanie

10.2 Komunikácia a verzia

Pre správu zdrojového kódu sme používali verzovací systém Git s repozitárom na GitHub. Pravidelne sme sa stretávali na konzultáciách, kde sme riešili návrh rozhraní medzi modulmi a aktuálne problémy.

11 Záver

Projekt implementuje funkcionalitu prekladača IFJ25 podľa špecifikácie. Kód je modulárny a testovaný. Tabuľka symbolov implementuje AVL strom pre efektívne vyhľadávanie. Kľúčom tohto projektu bolo rozšíriť si naše programovacie schopnosti v jazyku C, naučiť sa pracovať v tíme, pochopiť, ako funguje prekladač a ako ho vytvoriť.