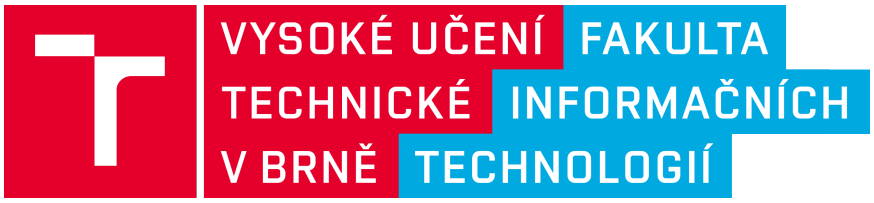


Dokumentace k projektu

Implementace překladače imperativního jazyka IFJ24

(rozšíření: FUNEXP)

Tým xpylypd00, varianta - vv-BVS

Mariia Sydorenko	xsydorm00	25%
Denys Pylypenko	xpylypd00	25%
Polina Ustiuzhantseva	xustiup00	25%
Michaela Mária Capíková	xcapikm00	25%

4. prosince 2024

Obsah

1 Úvod

Hlavní cílem projektu je vytvořit překladač jazyka IFJ24 do mezikódu IFJcode24. Překladač bude sloužit jako nástroj pro zpracování zdrojového kódu, který zajistí jeho analýzu a převod do formátu vhodného pro interpretaci.

Projekt zároveň poskytuje praktické zkušenosti s implementací klíčových částí překladače, jako je lexikální, syntaktická a sémantická analýza, a s generováním cílového mezikódu.

2 Návrh a implementace

2.1 Lexikální analýza

Celý lexikální analyzátor v našem projektu je udělán jako deterministický konečný automat podle předem vytvořeného diagramu (viz. A). Implementovaný KA je jako jeden nekonečně opakující se `switch` v hlavní funkci `getNextToken(..)`, přičemž každý případ `case` odpovídá jednomu stavu automatu.

Funkce iterativně načítá symboly ze standardního vstupu a pomocí přechodu mezi stavy automatu identifikuje lexémy, případně přiřadí, jaký je typ tokenu. Pokud další načtený znak nesouhlasí s žádným znakem, který jazyk podporuje, program je ukončen s chybou `1 (chyba v programu v rámci lexikální analýzy)`. Jinak se přechází do dalších stavů a načítají se další znaky, dokud nemáme hotový lexém.

Každý nalezený lexém je pro další zpracování uložen do struktury `Token`, která obsahuje typ, řetězcovou reprezentaci tokenu, řádek, ze kterého byl načten, a hodnoty typu long a double pro číselné literály (poslední dvě jsme v průběhu implementace zjistili, že jsou zbytečné).Takže v dalších analýzách, kdy je potřeba vědět číslo, řetězec ze struktury `Token`, kam jsme původně ukládali lexém, převede se na číslo pomocí funkcí `strtol` nebo `strtod`.

Když máme načtený správný token, předtím než ho vrátíme a ukončíme funkci, uložíme token do seznamu typu `ListOfTokens`. Toto nám umožňuje víckrát projít celým kódem během analýzy.

2.2 Syntaktická analýza

Pro syntaktickou analýzu, založenou na LL gramatice (viz. B), jsme využili metodu rekurzivního sestupu. Během syntaktické analýzy pro získání tokenů se využívá funkce `getCurrentToken()`, která načítá aktuální token ze struktury `ListOfTokens`. Vstupním bodem syntaktické analýzy je funkce `syntaxAnalysis()`, která na začátku zkontroluje správnost prologu a následně volá jednu z funkcí reprezentujících pravidla LL gramatiky. Pro každé z pravidel LL gramatiky existuje příslušná funkce, které se pak navzájem rekurzivně volají. Každá z těchto funkcí pak vrátí následující token pro další zpracování a kontrolu. Na konci syntaktické analýzy se kontroluje, jestli poslední vrácený token byl `EOF`.

Pro analýzu výrazů jsme využili precedenční syntaktickou analýzu s následujícími pravidly a precedencí operátorů v tabulce (viz. D). Pro práci s tokeny na zásobníku během precedenční analýzy byla vytvořena struktura `PrecedenceToken`, která kromě samotného tokenu obsahuje informace o tom, jestli token je terminál, zda můžeme provést redukce

a datový typ pro sémantické kontroly kompatibility typů. Každý načítaný token v rámci analýzy výrazů se pomocí funkce `tokenWrapper()` převede na `PrecedenceToken`.

Jelikož jsme se rozhodli implementovat rozšíření FUNEXP, během precedenční analýzy může dojít k přepnutí na analýzu rekurzivním sestupem pro zpracování volání funkce (například, pokud za načteným id se vyskytne otevírací závorka). Avšak pro zpracování každého z argumentů volané funkce se analýza bude opět přepínat na precedenční analýzu.

2.3 Sémantická analýza

Za účelem sémantické analýzy děláme průchod programem dvakrát: jednou pro uložení všech uživatelem definovaných a vestavěných funkcí do globální úrovně tabulky symbolů a kontrolu přítomnosti funkce `main`. Tento průchod jsme uznali za vhodný z důvodu, že jazyk IFJ24 umožňuje volat funkci dříve, než je deklarována, a také proměnná nesmí sdílet její identifikátor. Následně je zahájen 2. průchod zdrojovým kódem, během kterého se do tabulky symbolů ukládají proměnné a zároveň se TS využívá pro kontrolu kompatibility typů, využití proměnných v jejich rozsahu platnosti a ostatní sémantické kontroly.

2.4 Generování cílového kódu v jazyce IFJcode24

Generování kódu v našem překladači probíhá přímo, což znamená kód se generuje za běhu parseru pomocí sady funkcí vytvářejících jednotlivé části kódu. Tyto funkce lze dle účelu rozdělit do skupin(např. generovaní cyklu(`while`): sem patří funkce generování začátku cyklu, generování konce zpracování podmínky, generovaní konce cyklu). Výstupní kód se ukládá do struktury dvousměrného vázaného seznamu. Po dokončení zpracovávání vstupu v případě, že žádná lexikální, syntaktická nebo sémantická chyba nebyla nalezena a nedošlo k chybě při ukládání výstupního kódu, bude uložený kód vypsán na standardní výstup.

Náš generator využívá zásobník jako klíčový nástroj pro zpracování kódu. Instrukce s datovým zásobníkem se používají u výrazů, podmínek(`if-else`), cyklů(`while`) a při předávání parametrů funkcí.

Co se týče pomocných funkcí, při skoku na pomocnou funkci se vytváří nový rámec, do kterého jsou následně vloženy parametry z datového zásobníku. Návratová hodnota je předávána také přes zásobník. Navěští u cyklů(`while`), podmíněných příkazů(`if-else`) a názvy proměnných typu id_bez_null jsou generovaný jako unikátní pomocí počitadel(např. `$$while$1)`. `ifj.strcmp` a `ifj.substring` jsou implementovány jako samostatné funkce, které se vždycky vypíšou na konci generovaného kódu a volají se pomocí instrukci `CALL`.

Deklarace funkcí pro generování kódu jsou k dispozici v souboru generator.h, jejích implementace se nachází v `generator.c`. Struktura dvousměrného vázaného seznamu, jejího elementů a funkce, deklarující práce s ním, se nacházejí v souboru `generatorBuf.h`(jejích implementace v souboru `generatorbuf.c`). `codeStack.h` deklaruje strukturu zásobníku, používaného při práce s proměnnými typu `id_bez_null`, podmíněnými výrazy a `while` cykly. V souborech `genVarList.h` a `genVarList.c` je definován a implementován jednosměrně vázaný list pro ukládání jiz definovaných proměnných ve výstupním kódu.

3 Použité datové struktury

3.1 Tabulka symbolů

Pro náš projekt jsme zvolili variantu úlohy s implementací tabulky symbolů jako výškově vyváženého binárního vyhledávacího stromu.

Naše implementace využívá stromovou strukturu AVL a pomocný zásobník. Prvkem stromu je struktura obsahující informace o identifikátoru potřebné pro sémantickou analýzu. Tato struktura má dále 2 prvky s informacemi specifickými pro proměnnou či funkci.

Při každém vstupu do bloku v zdrojovém kódě se vytvoří nová TS - strom, který je následně vložen na zásobník a po skončení platnosti bloku odstraněn. Každá TS obsahuje pouze identifikátory definované v daném bloku. Zásobník tedy obsahuje všechny aktuálně platné identifikátory.

Identifikátory se vkládají do TS na vrcholu zásobníku. Při vyhledávání se prochází položky zásobníku od vrcholu a rekurzivně se prohledává příslušná TS.

3.2 tBuffer

Buffer na ukládání lexémů byl implementován pomocí struktury `tBuffer`, která umožňuje dynamickou práci s řetězci. Struktura obsahuje ukazatel na `data`, aktuální velikost uloženého řetězce `size` a maximální kapacitu `length`. Pro práci s bufferem slouží funkce `bufferInit` k inicializaci, `bufferAddChar` k přidávání jednotlivých znaků do bufferu a `bufferFree` k uvolnění paměti. Tato implementace umožňuje efektivní ukládání a manipulaci s lexémy při jejich zpracování v rámci lexikální analýzy. Implementace se nachází v souborch `buffer.(c|h)`.

3.3 ListToken

V rámci projektu byl implementován seznam pro ukládání tokenů. Struktura `ListToken` představuje prvek jednosměrně vázaného seznamu, který obsahuje uložený token `token` a ukazatel na další prvek seznamu `nextToken`. Hlavní struktura seznamu, `ListOfTokens`, pak zahrnuje ukazatel na první prvek seznamu `firstToken`, ukazatel na aktuálně aktivní prvek `activeToken` a ukazatel na prvek pro čtení tokenů od začátku `tokenToGet`. Navíc obsahuje proměnnou `currentLength`, která uchovává aktuální délku seznamu.

Inicializuje se pomocí funkce `init_list_of_tokens(&list)`. Po prvním průchodu se všechny tokeny uloží do tohoto seznamu a lze je opakovaně procházet pomocí funkce `get_token_from_list(&list)`, avšak předtím je potřeba zavolat funkci `i_want_to_get_tokens(&list)`, která provede důležité nastavení pro načtení tokenů od začátku. Pokud je potřeba znovu číst tokeny od začátku, je nutné před použitím `get_token_from_list(&list)` znovu zavolat `i_want_to_get_tokens(&list)`. Na konci programu je třeba seznam uvolnit pomocí `free_list_of_tokens(&list)`. Implementace seznamu se nachází v souborch `scanner.(c|h)`.

4 Práce v týmu

4.1 Způsob práce v týmu a vývojový cyklus

S týmem jsme naplánovali pravidelné schůzky, kde jsme diskutovali aktuální stav projektu, přemýšleli o dalších krocích a řešili případné problémy.

Komunikace probíhala přes Discord. Pro plánování a řízení projektu jsme použili nástroj Jira, kde jsme vytvářeli úkoly, jejich podúkoly a snažili se je sledovat. Verzování kódu jsme řešili pomocí Gitu, přičemž repozitář byl hostován na platformě GitHub. Vývoj probíhal převážně v prostředí Visual Studio Code. Na dokumentaci jsme využili platformu Overleaf. Pro návrh deterministického konečného automatu (DKA) jsme používali nástroj draw.io.

Pracovali jsme agilním způsobem, kdy jsme si každý týden stanovili konkrétní cíle a na konci týdne vyhodnocovali dosažené výsledky. Každý člen týmu byl zodpovědný za určitou oblast projektu.

4.2 Rozdělení práce mezi členy týmu

Člen týmu	Přidělená práce
Mariia Sydorenko	Syntaktická analýza, testování, dokumentace
Denys Pylypenko	Lexikální analýza, testování, dokumentace
Polina Ustiuzhantseva	Generátor kodu, testování, dokumentace
Michaela Mária Capíková	Sémantická analýza, tabulka symbolů, dokumentace

Tabulka 1: Rozdělení práce mezi členy týmu

5 Testování

Při testování jsme postupovali krok za krokem, abychom se ujistili, že všechny části projektu fungují správně a dobře spolupracují. Každý člen týmu měl na starosti testování své části kódu a připravil si k tomu odpovídající sadu testů. Díky tomu jsme mohli rychle odhalit případné chyby už na začátku vývoje.

Na týmových schůzkách jsme se zaměřovali na to, jak jednotlivé části spolupracují. Testovali jsme, jestli mezi moduly probíhá správná komunikace a zda program jako celek funguje, jak má. Tento přístup nám pomohl najít a opravit spoustu chyb, které by mohly ovlivnit stabilitu celého projektu.

6 Výčet souborů

Soubory	Význam
buffer.(c|h)	Definice a implementace bufferu na ukládání lexémů v Tokenu
codeStack.(c|h)	Definice struktury zásobníku pro ukládání aktuálních názvů while a if-else návěští a jejich případných id_bez_null názvů proměnných a jeho funkci, implementace daných funkci
generator.(c|h)	Generator
generatorBuf.(c|h)	Definice a implementace funkci pro buffer generatoru
list.(c|h)	Seznam se využívá k ukládání tokenů, slouží k předávání parametrů funkcí do generátoru kódu.
parser.(c|h)	Parser, syntaktická analýza
precedence_token.h	Definuje strukturu PrecedenceToken a funkce pro práci s tokeny během precedenční analýzy.
precedence.h	Struktura typu PrecedenceToken
semantic.h	Funkce pro sémantickou analýzu
scanner.(c|h)	Scanner, lexikální analýza
stack.(c|h)	Implementace zásobníku, který ukládá data typu PrecedenceToken
symtable.(c|h)	Tabulka symbolů
error.(c|h)	Výpis chyb
genVarList.(c|h)	Definice struktury jednosměrného vázaného seznamu a jeho funkci, slouží pro ukládání definovaných proměnných ve výstupním kódu

Tabulka 2: Soubory použité na řešení

7 Rozšíření na prémiové body

7.1 FUNEXP

Volání funkce může být součástí výrazu, zároveň mohou být výrazy v parametrech volání funkce.

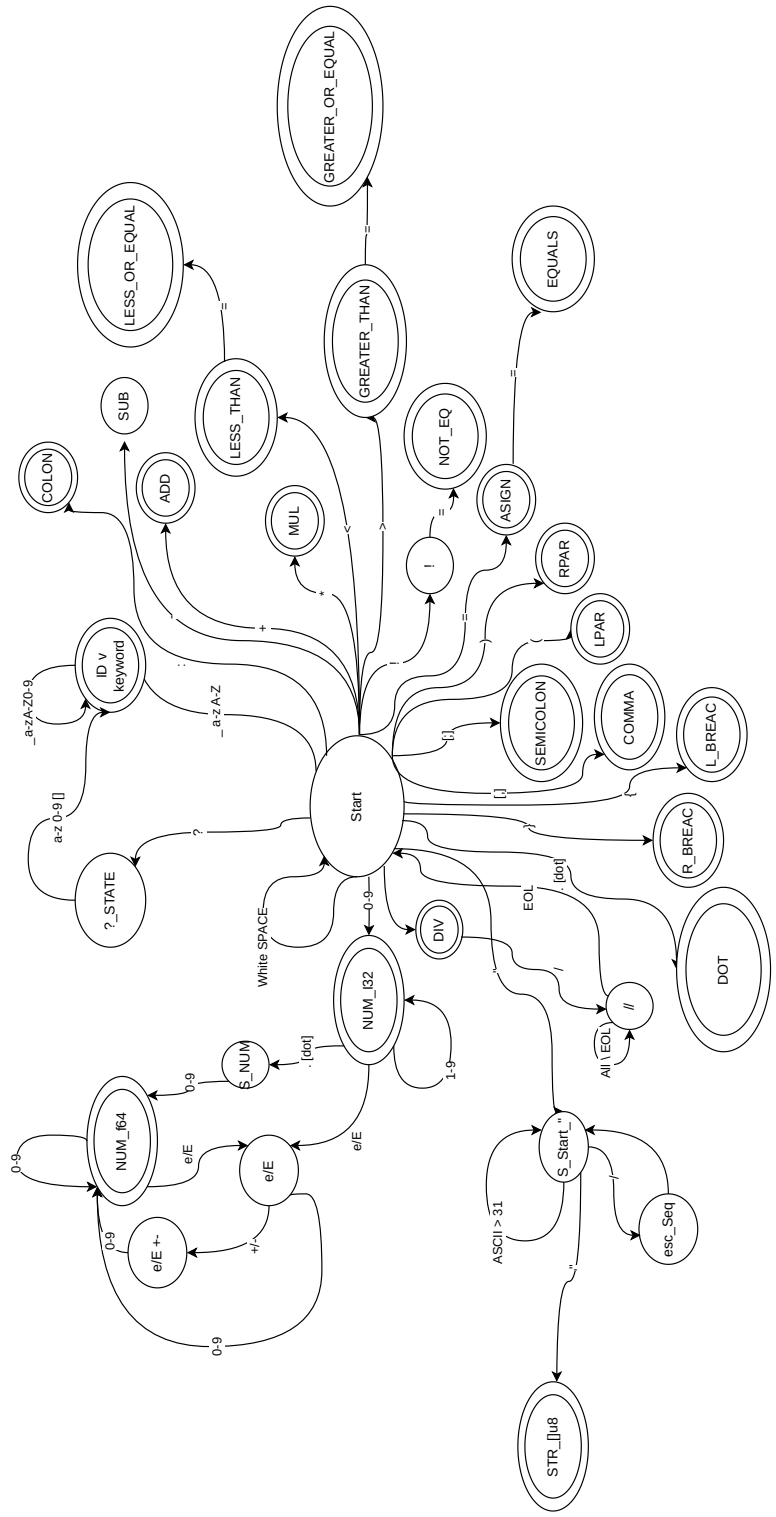
8 Závěr

Celý projekt byl náročný, ale nakonec nám přinesl spoustu nových zkušeností a znalostí. Postupně jsme se díky přednáškám z IFJ a IAL do problému dostali a zvládli ho v nějaké míře dokončit.

Týmová spolupráce pro nás byla klíčová. Díky dobré organizaci a pravidelným schůzkám jsme projekt dotáhli do konce. Problémy, které se objevily, jsme vyřešili díky společné práci, konzultacím ve fóru, průběžnému testování a také testovacím případům z Discordu.

Projekt nám dal nejen hlubší pochopení fungování překladačů, ale také spoustu praktických zkušeností s týmovou prací a řešením komplexních problémů.

A Diagram konečného automatu



Obrázek 1: KDA

B LL – Gramatika

1. <prog> -> const ifj = @ import ("ifj24.zig") ; <code>
2. <code> -> var <variable_definition> <code>
3. <code> -> const <variable_definition> <code>
4. <code> -> id <assignment_or_function_call> <code>
5. <code> -> if (<expression>) <id_without_null> { <code> } <else>
6. <code> -> while (<expression>) <id_without_null> { <code> }
7. <code> -> pub fn id (<list_of_parameters>) <function_return_type> { <code> }
8. <code> -> ifj <standard_function_call> ; <code>
9. <code> -> ϵ
10. <variable_definition> -> id <variable_type> = <expression> ;
11. <variable_type> -> : <data_type>
12. <variable_type> -> ϵ
13. <assignment_or_function_call> -> = <expression> ;
14. <assignment_or_function_call> -> (<arguments>) ;
15. <arguments> -> <expression> <arguments_tail>
16. <arguments> -> ϵ
17. <arguments_tail> -> , <params>
18. <arguments_tail> -> ϵ
19. <id_without_null> -> | id |
20. <else> -> else { <code> }
21. <else> -> ϵ
22. <list_of_parameters> -> id : <data_type> <list_of_arguments_tail>
23. <list_of_parameters> -> ϵ
24. <list_of_parameters_tail> -> , <list_of_arguments>
25. <list_of_parameters_tail> -> ϵ
26. <standard_function_call> -> . id (<params>)
27. <data_type> -> i32
28. <data_type> -> ?i32
29. <data_type> -> f64
30. <data_type> -> ?f64
31. <data_type> -> []u8
32. <data_type> -> ?[]u8
33. <function_return_type> -> <data_type>
34. <function_return_type> -> void

Pozn.: <expression> značí přepnutí na precedenční analýzu výrazu

C LL – Tabulka

	const	var	id	if	while	pub	ifj	:	=	(	,	|	else	.	i32	?i32	f64	?f64	[]u8	?[]u8	void	ε
<prog>	1																					
<code>	3	2	4	5	6	7	8															9
<variable_definition>			10																			
<variable_type>								11														12
<assignment_or_function_call>									13	14												
<arguments>																						16
<arguments_tail>											17											18
<id_without_null>												19										20
<else>													21									22
<list_of_parameters>			23																			24
<list_of_parameters_tail>											25											26
<standard_function_call>														27								
<data_type>															28	29	30	31	32	33		
<function_return_type>															34	34	34	34	34	34	35	

Tabulka 3: LL – tabulka použitá při syntaktické analýze

D Precedenční tabulka

	+	-	*	/	==	!=	<	>	<=	>=	(	)	term	$
+	>	>	<	<	>	>	>	>	>	>	<	>	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
==	<	<	<	<							<	>	<	>
!=	<	<	<	<							<	>	<	>
<	<	<	<	<							<	>	<	>
>	<	<	<	<							<	>	<	>
<=	<	<	<	<							<	>	<	>
>=	<	<	<	<							<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
term	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	

Tabulka 4: Použitá při precedenčnı syntaktické analýze výrazů.

Pozn.: term může být id proměnné, číselný literál, řetězcový literál, volání funkce nebo null.

Pravidla:

- $E \rightarrow term$
- $E \rightarrow (E)$
- $E \rightarrow E + E$
- $E \rightarrow E - E$
- $E \rightarrow E * E$
- $E \rightarrow E/E$
- $E \rightarrow E == E$
- $E \rightarrow E! = E$
- $E \rightarrow E < E$
- $E \rightarrow E > E$
- $E \rightarrow E <= E$
- $E \rightarrow E >= E$