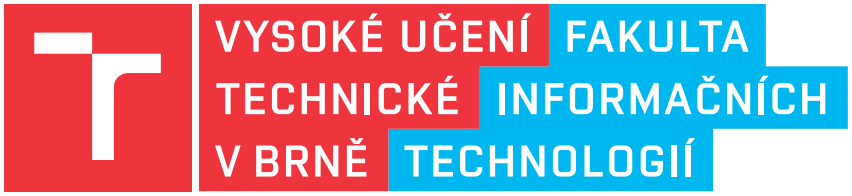

Projektová dokumentace

Implementace překladače jazyka IFJ23

Tým xhalam16, varianta TRP-izp

Marek Halamka	**(xhalam16)**	45 %
Šimon Motl	(xmotls00)	35 %
Richard Juřica	(xjuric31)	20 %
Jan Kroutil	(xkrout04)	0 %

6. prosince 2023

Obsah

1 Úvod

Překladač jazyka IFJ23 je projekt vytvořený v rámci předmětů IFJ a IAL na FIT VUT v Brně.

Cílem projektu je vytvořit překladač jazyka IFJ23, který bude překládat zdrojový kód napsaný v tomto jazyce do cílového jazyka IFJcode23 a vrací příslušný návratový kód.

Program je implementován jako konzolová aplikace, která na standardní vstup přijímá zdrojový kód jazyka IFJ23 a na standardní výstup vypisuje cílový kód jazyka IFJcode23.

Překladač je implementován v jazyce C dle normy C11[1] a je rozdělen do několika modulů. Každý modul má svůj hlavičkový soubor, který obsahuje deklarace funkcí a struktur definovaných v daném modulu.

2 Implementace

Zvolená metoda implementace je jednoprůchodový **syntaxí řízený překlad** a skládá se z částí, které jsou popsány v následujících podkapitolách.

2.1 Pomocné moduly a rozhraní

Program mimo hlavní funkce obsahuje několik důležitých modulů a rozhraní využívaných v rámci celého projektu. Jejich výčet a popis se nachází v následujících podkapitolách.

2.1.1 Error

Toto rozhraní, definované v hlavičkovém souboru `error.h`, obsahuje výčtový typ chybových stavů, které mohou nastat při běhu programu.

2.1.2 DynamicBuffer

Tento modul definovaný v souboru `dynamic_buffer.c` má za úkol uchovávat řetězce proměnné délky. Korespondující hlavičkový soubor `dynamic_buffer.h` obsahuje deklarace funkcí a struktur definovaných v tomto modulu.

Rozhraní obsahuje strukturu `dynamic_buffer`, která obsahuje **ukazatel** na alokovanou paměť, **kapacitu** alokované paměti a **velikost** obsazené paměti.

Modul obsahuje funkce pro inicializaci, uvolnění, realokaci a práci s řetězci. Buffer se v případě naplění automaticky realokuje na dvojnásobek své původní kapacity.

Pomocí výše zmíněných funkcí jsme definovali abstraktní datový typ `DynamicBuffer`, který je využíván v dalších částech projektu.

2.1.3 DynamicArray

Na stejném principu jako `DynamicBuffer` je implementován i ADT `DynamicArray`, který je definován v souboru `dynamic_array.c` s rozhraním v souboru `dynamic_array.h`.

Struktura definující tento typ obsahuje **ukazatel** na alokovanou paměť, **kapacitu** alokované paměti a **velikost** obsazené paměti. Pole uchovává proměnný počet struktury typu `ArrayItem`, které obsahují ukazatel na typ `void`. Tím jsme docílili toho, že do pole můžeme ukládat jakýkoliv typ dat.

Používá se především v generování cílového kódu, více o tom v sekci **??**.

[1] ISO/IEC 9899:2011, viz. https://www.iso.org/standard/57853.html

2.1.4 Token

Hlavičkový soubor `token.h` obsahuje definici struktury `token`, která reprezentuje token.

Struktura obsahuje typ tokenu, hodnotu tokenu a textovou reprezentaci tokenu ve zdrojovém souboru.

Typ tokenu je definován výčtovým typem `token_type`, který obsahuje všechny typy tokenů jazyka IFJ23. Může nabývat i speciálních hodnot jako `TOKEN_UNKNOWN` signalizující lexikální chybu, `TOKEN_ERROR`, který dává najevo výskyt nějaké interní chyby, `TOKEN_EOF` a `TOKEN_NONE` použitý pro komentáře.

Hodnota tokenu je typu `union`, a nabývá buď hodnoty odpovídající číslené hodnotě tokenu, nebo ukazatele na dříve definovanou strukturu `DynamicBuffer`, která reprezentuje řetězec. Typ union byl zvolen z důvodu úspory paměti, protože token může obsahovat pouze jednu z těchto hodnot.

Textová reprezentace tokenu ve zdrojovém souboru je uložena jako ukazatel na strukturu `DynamicBuffer`, který obsahuje řetězec. Je uchovávána z důvodu implementace funkce `peek_token`, která je podrobněji popsána v sekci **??**.

2.1.5 Tabulka symbolů

Tabulka symbolů slouží k uložení informací o proměnných a funkcích.

Deklarace proměnné či funkce odpovídá vytvoření záznamu v tabulce symbolů, kde klíčem je **identifikátor** proměnné.

Dle zvolené varianty zadání je tabulka symbolů implementována jako **TRP s otevřenou adresací**.

Implicitní rozptýlení využívá **lineární** určení kroku při výpočtu dalšího volného indexu.

Tabulka symbolů je implementována v souboru `symtable.c` s rozhraním v souboru `symtable.h`.

Rozhraní obsahuje funkce pro inicializaci, uvolnění, vložení a vyhledání položky v tabulce symbolů či vytvoření nové položky.

Kromě těchto funkcí obsahuje i signaturní **hashovací funkci**, která je použita při transformaci klíče, na index do tabulky symbolů.

```c
size_t hash_function(const char *str)
{
    uint32_t h = 0; // musí mít 32 bitů
    const unsigned char *p;
    for (p = (const unsigned char *)str; *p != '\0'; p++)
        h = 65599 * h + *p;
    return h;
}
```

Obrázek 1: Hashovací funkce

Při vytváření hashovací funkce jsme čerpali informace z předmětů IJC, IAL a zde: [**?**].

Tabulka se v případě naplnění automaticky **realokuje** na dvojnásobek své původní kapacity.

V programu rozlišujeme mezi `GLOBÁLNÍ` a `LOKÁLNÍ` tabulkou symbolů. Globální tabulka symbolů je vytvořena při inicializaci programu a je uvolněna při jeho ukončení. Lokální tabulka symbolů je vytvořena při vstupu do bloku a je uvolněna při jeho opuštění.

Tyto typy tabulek se liší mimojiné i v datech, které uchovávají. Funkce pro práci s tabulkou symbolů jsou implementovány tak, aby bylo možné používat stejné funkce pro oba typy tabulek. Docíleno je to pomocí ukazatele typu `void`, který je přetypován na konkrétní typ tabulky symbolů v závislosti na tom, zda se jedná o `GLOBÁLNÍ` nebo `LOKÁLNÍ` tabulku symbolů.

2.1.6 Parameter list

Parametry funkcí jsou uchovávány v seznamu, který je implementován jako **jednosměrně vázaný seznam**.

Seznam je implementován v souboru `symtable.c` s rozhraním v souboru `symtable.h`.

Rozhraní obsahuje funkce pro inicializaci, uvolnění, vložení a vyhledání položky v seznamu.

Implementací výše zmíněných funkcí je definován abstraktní datový typ `parameter_list_t`, který je využíván v dalších částech projektu.

2.1.7 Stack

Zásobník je v projektu využíván na více místech, pokaždé pro ukládání jiného typu dat.

Proto byl zásobník implementován jako **obecný zásobník**, který je definován v souboru `stack.c` s rozhraním v souboru `stack.h`.

Zásobník uchovává položky typu `Stack_Frame`, které obsahují ukazatel na data typu `void`. Tím je umožněno ukládat na zásobník jakýkoliv typ dat.

Rozhraní obsahuje známé funkce pro práci se zásobníkem. Tím je definován abstraktní datový typ `Stack`.

Za účelem ulehčení řešení některých problémů, které se vyskytly při implementaci, rozhraní zásobníku poskytuje i funkci `Stack_get(stack,index)`, která vrací položku na zadaném indexu.

Jelikož struktura ukládá data jako ukazatel, je potřeba dát pozor, nad čím voláme operaci `free`. Proto byla implementována funkce `stack_empty`, která uvolní všechny ukazatele na zásobníku. Funkce předpokládá, že na zásobníku jsou pouze ukazatele na dynamicky alokovanou paměť.

Pokud dojde k naplnění zásobníku, je automaticky **realokován** na dvojnásobek své původní kapacity.

2.2 Lexikální analýza

Lexikální analýza je definována v souboru `scanner.c` s rozhraním v souboru `scanner.h` a implementována jako **deterministický konečný automat**. Graf konečného automatu je zobrazen zde.

Automat se rozhoduje na základě aktuálního stavu a načteného znaku ze vstupního souboru.

V jazyce C je implementován pomocí konstrukce `if...else if`, kde každá větev odpovídá jednomu stavu automatu. Automat také využívá a nastavuje pomocné statické globální proměnné, které jsou definovány v souboru `scanner.h`.

Hlavní funkce, kterou analýza implementuje je `get_token`, která vrací dříve popsaný `token` načtený ze vstupního souboru. Postupně načítá znaky ze vstupního souboru a předává je automatu, který na základě aktuálního stavu a načteného znaku rozhodne o dalším postupu. Pokud načtený znak neodpovídá žádnému stavu, vrátí funkce token s typem `TOKEN_UNKNOWN`, který signalizuje lexikální chybu. Načtené znaky jsou ukládány do datového typu `DynamicBuffer`, který automat postupně kontroluje, zda neodpovída některému z klíčových slov jazyka IFJ23. Pokud ano, je vrácen token s odpovídajícím typem klíčového slova.

Je-li některý z načtených tokenů typu číslo nebo řetězec, je do unie `token_value` uložena i korespondující hodnota tohoto tokenu. Pokud se jedná o desetinné číslo typu Double zadané ve speciální notaci, jako např. `1.0e-10`, je hodnota scannerem převedena na desetinné číslo. Podobně je to tak i u řetězců obsahující escape sekvence, které jsou převedeny na odpovídající znaky.

Scanner také implementuje funkci `peek_token`, která vrací následující token ze vstupního souboru, ale nečte ho. Tato funkce je využívána v syntaktické analýze pro predikci následujícího tokenu. Pro implementaci této funkce byla za potřebí implementovat pomocná funkce `unget_token`, která vrátí znaky uložené v `DynamicBuffer` zpět do vstupního souboru.

2.3 Syntaktická analýza

Jelikož je překladač implementován pomocí metody syntaxí řízeného překladu, je syntaktická analýza nejdůležitější částí celého projektu. Je implementována v souboru `parser.c` s rozhraním v souboru `parser.h`. Toto rozhraní poskytuje funkci `parse`, která je volána z hlavní funkce programu.

Syntaktická analýza se řídí LL – gramatikou předem definovanou pro jazyk IFJ23, zobrazenou zde. Využívá metodiky **rekurzivního sestupu** podle pravidel v LL – tabulce, kterou můžete nalézt zde. Dále využívá metodu **precedenční syntaktické analýzy** pro zpracování výrazů, více o této metodě v sekci **??**.

Po dobu syntaktické analýzy je volána funkce `get_token` z modulu `scanner`. Na základě typu vráceného tokenu se rozhoduje o syntaktické validitě zdrojového kódu. Funkce `parse` si volá pomocné funkce podle typu neterminálu na pravé straně aplikovaného pravidla.

2.3.1 Syntaktický strom

Definici struktury `TreeNode`, reprezentující uzel syntaktického stromu, najdeme v rozhraní. Tato struktura obsahuje typ uzlu, který je definován výčtovým typem `node_type` a pole proměnné délky ukazatelů na potomky uzlu. Dále uzel může uchovávát informace o tokenech, jako například hodnotu konstanty nebo název identifikátoru proměnné či funkce.

Při aplikování pravidel LL – gramatiky jsou vytvářeny uzly syntaktického stromu nebo v případě neúspěchu při aplikování některého z pravidel je signalizována syntaktická chyba a program je ukončen. Pro tvorbu stromu jsme využili datovou strukturu **obecného stromu**, který je zakořeněn v uzlu `root` s typem `NODE_PROGRAM` a tvoří se **shora dolů**.

Při tvorbě stromu jsou jednotlivé podstromy postupně kontrolovány sémantickou analýzou a následně generován cílový kód. Bližší popis volání sémantických akcí je zde **??**.

2.3.2 Práce s tabulkami symbolů

Parser se stará o vytvoření a naplnění tabulek symbolů. V jazyce IFJ23 existují dva typy tabulek symbolů, `GLOBÁLNÍ` a `LOKÁLNÍ`. Globální tabulka je právě jedna a je vytvořena při inicializaci programu. Do globální tabulky jsou ukládány informace o funkcích a globálních proměnných.

Lokální tabulka symbolů je vytvářena při vstupu do bloku a je uvolněna při jeho opuštění. Do lokální tabulky jsou ukládány informace o lokálních proměnných.

Syntaktická analýza také vytváří zásobník lokálních tabulek symbolů, který je využíván sémantickými akcemi. Parser zajišťuje správné přidání a odebírání lokálních tabulek symbolů ze zásobníku.

2.3.3 Zpracování výrazů pomocí precedenční syntaktické analýzy

Postup zpracování výrazů je založen na precedenční tabulce, kterou naleznete zde. V jazyce C byla implementována jako dvourozměrné pole znaků, kde každý typ znak určuje akci, která se má dál provést.

K implementaci analýzy byl využit dříve definovaný ADT `Stack`. Výrazy jsou zpracovávány **zdola nahoru** a výsledný strom je vytvářen **shora dolů**. K tomu je využit další zásobík `Stack`, kam se ukládájí jednotlivá pravidla LL – gramatiky.

2.3.4 Volání sémantických akcí a generování cílového kódu

Sémantické akce jsou volány při vytváření syntaktického stromu. Parser volá funkci `semantic` z modulu `semantic.c` s rozhraním v souboru `semantic.h`. Tato funkce přijímá ukazatel na uzel syntaktického stromu a ukazatel na zásobník lokálních tabulek symbolů. Na základě výsledku sémantické analýzy je vygenerován cílový kód. V případě, že došlo k chybě, je program ukončen s odpovídajícím návratovým kódem.

2.4 Sémantická analýza

Sémantická analýza je implementována v souboru `semantic.c` s rozhraním v souboru `semantic.h`. Toto rozhraní poskytuje funkci `semantic`, která je volána z hlavní funkce programu. Jelikož naše implementace využívá jednoprůchodový překlad, je sémantická analýza prováděna při vytváření syntaktického stromu.

Ve funkci `semantic` se rozhodne, jaká sémantická akce se má provést na základě typu uzlu syntaktického stromu a zavolá se příslušná pomocná funkce. Jednotlivé funkce poté navigují v syntaktickém stromu podle předem stanovených pravidel a provádí sémantickou kontrolu. Pokud nastane chyba, vrátí funkce odpovídající chybový kód.

Sémanticka analýza ke kontrole využívá naplněné tabulky symbolů a zásobník lokálních tabulek symbolů předaný z parseru.

2.4.1 Hledání v tabulkách symbolů

Vždy když sémantická analýza potřebuje vyhledat položku, začne od lokální tabulky symbolů na vrcholu zásobníku lokálních tabulek symbolů a postupně se prochází všechny tabulky symbolů na zásobníku. V případě, že položka není nalezena ani v jedné z tabulek na zásobníku, podívá se sémantika do globální tabulky symbolů. Pokud položka není nalezena ani v globální tabulce symbolů, je vrácen chybový kód.

V případě, že se jedná o funkci, hledá sémantická analýza rovnou v globální tabulce symbolů.

2.5 Generování cílového kódu

Generování cílového kódu je implementováno v souboru `code_gen.c` s rozhraním v souboru `code_gen.h`. Toto rozhraní poskytuje spoustí funkcí pro generování cílového kódu, které je obvykle volány z hlavní funkce programu. Jelikož naše implementace využívá jednoprůchodový překlad, je generování cílového kódu prováděno při vytváření syntaktického stromu. Využívá struktur ADT `DynamicBuffer`, ADT `DynamicArray` a především ADT `Stack` pro práci s lokálními proměnnými.

2.6 Generování výrazů

Je implementováno pomocí rekurzivního volání funkce `generateExpression`, která příjmá ukazatel na uzel syntaktického stromu, na základě jehož struktury se rozhoduje, jaká akce se má provést.

Klíčovým faktorem při generování je vlastnost předáného uzlu, a to konkrétně počet jeho potomků. Relační operátory, které se nenachází v instrukční sadě cílového jazyka `IFJcode23`, jsou převedeny na ekvivalentní operátory, které se v instrukční sadě nachází.

2.7 Generování vestavěných funkcí

Každá vestavěná funkce má svou vlastní funkci pro generování kódu, která na základě informací o funkci vygeneruje odpovídající instrukci z instrukční sady cílového jazyka `IFJcode23`.
Výjimkou je funkce `substring`, která nemá v instrukční sadě ekvivalentní instrukci. Proto je tato funkce implementována na základě známých algoritmů pro hledání v řetězcích.

3 Práce v týmu

3.1 Komunikace a verzovací systém

V týmu jsme komunikovali buď osobně, nebo vzdáleně pomocí aplikace Discord. Na této platformě jsme si vytvořili vlastní server, kde jsme si vytvořili kanály pro komunikaci a sdílení souborů.

Pro verzování projektu jsme zvolili verzovací systém Git se správou repozitářů pomocí služby GitHub. Náš repozitář je dostupný na adrese `https://github.com/xhalam16/IFJ-projekt`.

3.2 Prvotní rozdělení práce

Prvotní rozdělení bylo domluveno všemi členy týmu následovně:

- **Marek Halamka** – Lexikální analýza, tabulky symbolů – 25 %
- Šimon Motl – Syntaktická analýza – 25 %
- Richard Juřica – Sémantická analýza – 25 %
- Jan Kroutil – Generování cílového kódu – 25 %
- Společně – ADT Stack, ADT DynamicBuffer, ADT DynamicArray, dokumentace

3.3 Finální rozdělení práce

Ovšem kvůli nerovnoměrné práci na projektu některých členů týmu muselo být rozdělení práce změněno na:

Člen	**Implementované části**	**Body**
Marek Halamka	Lexikální analýza, ADT Stack a ADT DynamicBuffer, tabulky symbolů, sémantická analýza, dokumentace	45 %
Šimon Motl	Syntaktická analýza, generování kódu, ADT DynamicArray, dokumentace	35 %
Richard Juřica	Generování kódu	20 %
Jan Kroutil		0 %

Tabulka 1: Finální rozdělení práce

3.4 Zdůvodnění odchylek od rovnoměrného rozdělení

Někteří členové týmu se neúčastnili na projektu vůbec, nebo se účastnili minimálně. Proto bylo nutné práci rozdělit mezi zbylé členy týmu, aby se projekt stihl včas dokončit. To vedlo k nerovnoměrnému rozdělení práce (viz. tabulka výše) a tím pádem i k nerovnoměrnému rozdělení bodů.

4 Závěr

Díky projektu jsme se naučili pracovat v týmu a využívat verzovací systém Git. Dále jsme si osvojili znalosti o tom jak překladač funguje a jaké fáze překladu musí projít zdrojový kód, než je spuštěn.

I přes některé komplikace, které jsme museli řešit, jsme s výsledkem spokojeni. Jsme si vědomi, že některé části projektu jsme mohli implementovat lépe, ale vzhledem k časové tísni jsme se rozhodli, že je lepší projekt dokončit včas, než se zdržovat a riskovat, že projekt nestihneme včas dokončit.

Naše řešení se místy může trošku odchýlit od oficiálních materiálů, převážně z důvodu, že jsme s implementací začali s předstihem než byla témata detailně vysvětlena, ale **závazné metody** jsme samozřejmě **dodrželi**.

Literatura

[1] Lassonde School of Engineering: Hash Functions. [online], varianta sdbm, [vid. 2023-12-01]. URL http://www.cse.yorku.ca/~oz/hash.html

Precedenční tabulka

	2	1	(	)	id	0	4	END
2	>	<	<	>	<	<	>	>
1	>	>	<	>	<	<	>	>
(	<	<	<	=	<	<	<	
)	>	>		>		>	>	>
id	>	>		>		>	>	>
0	>	>		>		>	>	>
4	<	<	<	>	<	<	<	>
END	<	<	<		<	<	<	

Tabulka 2: Precedenční tabulka

Vysvětlivky k tabulce:

- **0** – Operátor s prioritou 0, tedy `!`
- **1** – Operátor s prioritou 1, tedy `*` a `/`
- **2** – Operátor s prioritou 2, tedy `+` a `-`
- **3** – Operátor s prioritou 3, tedy `<`, `<=`, `>`, `>=`, `==` a `!=`
- **4** – Operátor s prioritou 4, tedy `??`
- **id** – Identifikátor
- **END** – Konec výrazu

LL – gramatika

1. <program> → <command> EOL <program>
2. <program> → EOF
3. <command> → ε
4. <command> → <assign>
5. <command> → <declaration>
6. <command> → <func_declaration>
7. <command> → <if_statement>
8. <command> → <while>
9. <command> → <func_call>
10. <assign> → <left_value> = <right_value>
11. <left_value> → identifier
12. <left_value> → <declaration>
13. <left_value> → <declaration_keyword> identifier
14. <right_value> → <expression>
15. <right_value> → <func_call>
16. <declaration> → <declaration_keyword> identifier : <datatype>
17. <declaration_keyword> → let
18. <declaration_keyword> → var
19. <datatype> → INT
20. <datatype> → DOUBLE
21. <datatype> → STRING
22. <datatype> → INT?
23. <datatype> → DOUBLE?
24. <datatype> → STRING?
25. <value> → identifier
26. <value> → int
27. <value> → double
28. <value> → string
29. <value> → nil
30. <func_call> → identifier (<param_list1>)
31. <param_list1> → <param> <param_list1_next>
32. <param_list1> → ε
33. <param_list1_next> → , <param> <param_list1_next>
34. <param_list1_next> → ε
35. <param> → <value>
36. <param> → identifier : <value>
37. <func_declaration> → func identifier (<label> identifier : <datatype> <param_list0> <return_type> { EOL <body>
38. <return_type> → ε
39. <return_type> → -> <datatype>
40. <param_list0> →)
41. <param_list0> → , <label> identifier : <datatype> <param_list0>
42. <label> → _
43. <label> → identifier
44. <body> → <command> EOL <body>
45. <body> → return <return_value> EOL <body>
46. <body> → }
47. <return_value> → ε
48. <return_value> → <func_call>
49. <return_value> → <expression>
50. <if_statement> → if <condition> { <body> else { <body>
51. <condition> → let identifier
52. <condition> → <expression>
53. <while> → while <expression> { <body>

LL – tabulka

	<program>	<command>	<assign>	<left_value>	<right_value>	<declaration>	<declaration_keyword>	<datatype>	<value>	<func_call>	<param_list1>	<param_list1_next>	<param>	<func_declaration>	<return_type>	<param_list0>	<label>	<body>	<return_value>	<if_statement>	<condition>	<while>
ε	1	3									32	34			38			44	47			
identifier	1	4, 9	10	11	15				25	30	31		35, 36				43	44	48			
let	1	4, 5	10	12, 13		16	17											44			51	
var	1	4, 5	10	12, 13		16	18											44				
func	1	6												37				44				
if	1	7																44		50		
while	1	8																44				53
,												33				41						
EOF	2																					
INT								19														
DOUBLE								20														
STRING								21														
INT?								22														
DOUBLE?								23														
STRING?								24														
int									26		31		35									
double									27		31		35									
string									28		31		35									
nil									29		31		35									
)																40						
—																	42					
}																		46				
→															39							
return																		45				

Tabulka 3: LL – tabulka

Konečný automat pro lexikální analýzu

include/fffffff.drawio.png

Obrázek 2: Konečný automat pro lexikální analýzu