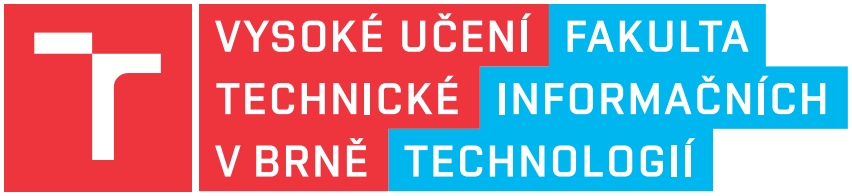

Projektová dokumentace

Implementace překladače jazyka IFJ23

Tým xhalam16, varianta TRP-izp

29. listopadu 2023

Marek Halamka	**(xhalam16)**	TBA,%
Šimon Motl	(xmotls00)	TBA %
Richard Juřica	(xjuric31)	TBA %
Jan Kroutil	(xkrout04)	TBA %

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.

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

2.1.3 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.

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 2.2.

2.1.4 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: citace.

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.

2.1.5 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.

2.1.6 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 na obrázku ODKAZ.

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`.

2.3 Syntaktická analýza

2.4 Sémantická analýza

2.5 Generování cílového kódu