Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentace projektu z předmětu IFJ a IAL

Implementace překladače jazyka IFJ20

Tým 101, varianta I

Jakšík, Aleš	**(xjaksi01)**	**25%**
Vlasáková, Nela	(xvlasa14)	25%
Mráz, Filip	(xmraz01)	25%
Bělohlávek, Jan	(xbeloh08)	25%

5. 12. 2020

Obsah

1 Úvod

Zadáním projektu bylo vytvořit překladač imperativního jazyka IFJ20 a na jeho vytvoření se podíleli všichni členové týmu. Naživo jsme se viděli jen jednou, pravidelně jsme však komunikovali pomocí Messengeru a hovory jsme realizovali na společném Discordovém serveru. Práci jsme si rozdělili následovně:

Aleš Jakšík

Vedoucí týmu dostal na práci syntaktickou a sémantickou analýzu vyjma analýzu výrazů, navíc také implementoval tabulku symbolů.

Soubory:

- `parser.c/parser.h`
- `symtable.c/symtable.h`

Nela Vlasáková

Na práci dostala syntaktickou a sémantickou analýzu výrazů, k čemuž patří i implementace obousměrně vázaného seznamu.dokumentaci a testování kódu.

Soubory:

- `expression.c/expression.h`
- `exprList.c/exprList.h`

Filip Mráz

Na starost dostal generování výsledného kódu a implementaci dynamického řetězce, dále se podílel na implementaci obousměrně vázaného seznamu pro lexikální analyzátor.

Soubory:

- `generator.c/generator.h`
- `tokenList.c/tokenList.h`
- `dynamicString.c/dynamicString.h`

Jan Bělohlávek

Honzovým úkolem bylo vytvořit lexikální analyzátor, pro který také vytvořil konečný automat.

Soubory:

- `scanner.c/scanner.h`

2 Návrh

Celý překladač je složen z několika spolupracujících modulů. Každý má většinou jednu hlavní funkci, která je z jiného modulu volána, a tak tedy dojde k jakési komunikaci a předání informací. Kromě vrácení návratové hodnoty mohou takové funkce také zapisovat do různých proměnných (pomocí předání ukazatele na danou proměnnou), a oznamovat tak jiným modulům různé skutečnosti, příkladem by mohlo být například získání datového typu výrazu. Chybová návratová hodnota je předávána vždy zpět od místa, kde se vyskytla, až do `main.c`, kde je vyhodnocena, je na standartní chybový výstup vytištěna chybová hláška a program je ukončen s touto předávanou hodnotou.

K implementaci jsme využili dvou datových struktur - **binárního stromu** (pro tabulku symbolů) a **oboustranně vázaného seznamu**, který je používán jak pro nahrání celého vstupního souboru, tak například pro předávání výrazů k analýze, nebo následně předání výrazu ve formě postfixu k tištění.

Pro lexikální analýzu jsme využili **konečného automatu**, při syntaktické kontrole jsme využili LL-gramatiky a **metody rekurzivního sestupu**. Výrazy jsme zpracovali podle **precedenční syntaktické analýzy**.

3 Implementace

Jako první je ze souboru `main.c` volán syntaktický/sémantický analyzátor - `parser.c`. Ten si zavolá lexikální analyzátor (jde tedy o syntaxí řízený překlad), který projde celý vstupní soubor, provede lexikální analýzu podle konečného automatu a vytvoří obousměrně vázaný seznam S tzv. *Tokeny*.

```
typedef struct Token {
    TokenType t_type;
    tStr *atribute;
    struct Token *lptr;
    struct Token *rptr;
} *TokenPtr;
```

Kód 1: Implementace struktury tokenu

Tento seznam pak předá zpět syntaktickému/sémantickému analyzátoru, který jej projde a provede syntaktickou kontrolu podle LL gramatiky, následně pak zkontroluje sémantiku. Pokud narazí na místo, kde by se měl nacházet výraz, vytvoří z něj další obousměrně vázaný seznam a pošle jej na precedenční analýzu do `expression.c`, kde je výraz zkontrolován jak po sémantické, tak syntaktické stránce, a v případě, že je vše v pořádku, je vstupní seznam reprezentující výraz převeden na postfix a odeslán dále do generátoru výstupního kódu.

3.1 Lexikální analýza - `scanner.c`

Lexikální analýza je řízena konečným automatem. Ze vstupu načte vždy jeden znak a postupně je přidává do předem připraveného řetězce, dokud nedojde do stavu, kdy může tento řetězec přijmout nebo zamítnout. Pokud jej může přijmout, vytvoří z něj *Token* a přidá jej do obousměrně vázaného seznamu, který poté, co narazí na `EOF`, považuje soubor za lexikálně korektní a předá jej zpět do `parser.c`.

3.1.1 Dynamický řetězec

Práci s řetězci máme řešenou pomocí implementace vlastního dynamického řetězce.

3.2 Syntaktická a sémantická analýza

Syntaktická kontrola je řízena pravidly **LL gramatiky**, informace potřebné k sémantické kontrole (existence proměnných a funkcí, datové typy a podobně) jsou uchovávány v **tabulce symbolů** implementované pomocí binárního stromu.

3.2.1 Syntaktická analýza

Začátkem syntaktické analýzy je kontrola, jestli vstupní soubor obsahuje povinnou hlavičku. Následně se celý seznam, který obsahuje *Tokeny* reprezentující vstupní soubor, projde poprvé. Funkce `buidInFunc` projde všechny hlavičky funkcí, uloží je do tabulky symbolů spolu s informacemi o vstupních a výstupních parametrech (počet a datový typ). Také zkontroluje, že poslední funkcí je `main`. Po provedení tohoto prvního běhu se pak celý seznam projde znova a kontrolují se těla jednotlivých funkcí. Tato kontrola se řídí pravidly LL gramatiky.

3.2.2 Sémantická analýza

Při procházení těly funkcí se do tabulky symbolů nahrávají informace o proměnných. Každé tělo představuje novou tabulku symbolů a postup vytváření tabulek je následující:

1. **Vstup do těla** (funkce, konstrukce `if`, cyklus)
2. **Vytvoří se nová tabulka symbolů** a vloží se do seznamu tabulek symbolů (vždy na první místo)
   - Hlavička cyklu je vždy samostatná nová tabulka, tudíž je zaručeno, že v ní lze použít proměnné definované dříve a zároveň, že proměnné definované v hlavičce budou dostupné v jejím těle
3. **Odchod z těla** znamená odstranění tabulky ze seznamu tabulek symbolů

Ověření, že nějaká proměnná již byla definována (a případné získání datového typu a podobně), pak vypadá tak, že se první podíváme do tabulky na začátku seznamu, a pokud nic nenajdeme, jdeme hlouběji, dokud ji nenajdeme, v opačném případě pak nastává chyba.

3.2.3 Precedenční analýza výrazů

Hlavní funkcí analyzátoru výrazů je funkce `parseExp`. Ta obdrží seznam *Tokenů*, podle kterého následně vytvoří vlastní seznam, kde jsou prvky identifikovány podle symbolů precedenční tabulce.

```
typedef struct ListItem {
    bool isZero;
    PtType ptType;
    DataType dType;
    struct ListItem *next;
    struct ListItem *prev;
} *item;
```

Kód 2: Položka seznamu pro precedenční analýzu

Celá precedenční analýza je řízena precedenční tabulkou. Ta je v kódu implementována jako dvourozměrné pole znaků:

	input								
opStack		+	-	*	/	(	)	**cmp**	**$**
	+	R	R	S	S	R	S	R	R
	-	R	R	S	S	R	S	R	R
	*	R	R	R	R	R	S	R	R
	/	R	R	R	R	R	S	R	R
	(	S	S	S	S	R	S	R	R
	)	S	S	S	S	S	S	S	E
	cmp	R	R	R	R	R	E	R	R
	$	S	S	S	S	S	S	E	A

Analýza pracuje se třemi seznamy:

1. `input` reprezentující vstupní výraz
2. `idStack`, do kterého se vkládají výrazů
3. `opStack`, kam jsou vkládány operátory

Proměnná, ať už jde o nějaký identifikátor, číslo, desetinný literál nebo řetězec, je klasifikováno jako `E` (tedy jako výraz). Pokud při procházení seznamem narazíme na prvek označný jako `E`, rovnou jej vložíme do příslušného seznamu. Samotný proces precedenční analýzy pak probíhá následovně:

1. Symbol z precedenční tabulky získáme vždy zadáním "souřadnic":

   ```
   precTable[x][y]
   ```

   kde `x` reprezentuje symbol na konci seznamu `OpStack` a `y` reprezentuje symbol, na který se právě díváme ve vstupním seznamu.
2. Pokud dostaneme `R`, provedeme redukci
3. Pokud dostaneme `S`, provedeme přesunutí symbolu ze vstupu `input` na `opStack`
4. Pokud dostaneme `A`, kontrolujeme, konečné podmínky přijetí výrazu a pokud projdou v pořádku, výraz převádíme na postfix

Redukce je prováděna na základě následujících pravidel:

```
                (E)  →  E
 E + E  →  E          E - E   →  E
 E / E  →  E          E * E   →  E
 E < E  →  E          E > E   →  E
E >= E  →  E          E <= E  →  E
E == E  →  E          E != E  →  E
```

3.3 Generování kódů