

Implementace překladače imperativního jazyka IFJ07

Varianta zadání: d/2/I

Řešitelé:

Bendl Jaroslav	xbendl00	20%
Danko Martin	xdanko00	20%
Dvořák Michal	xdvora14	20%
Žuričková Daniela	xduric00	20%
Kopřiva Vít	xkopri05	20%

Kapitola 1

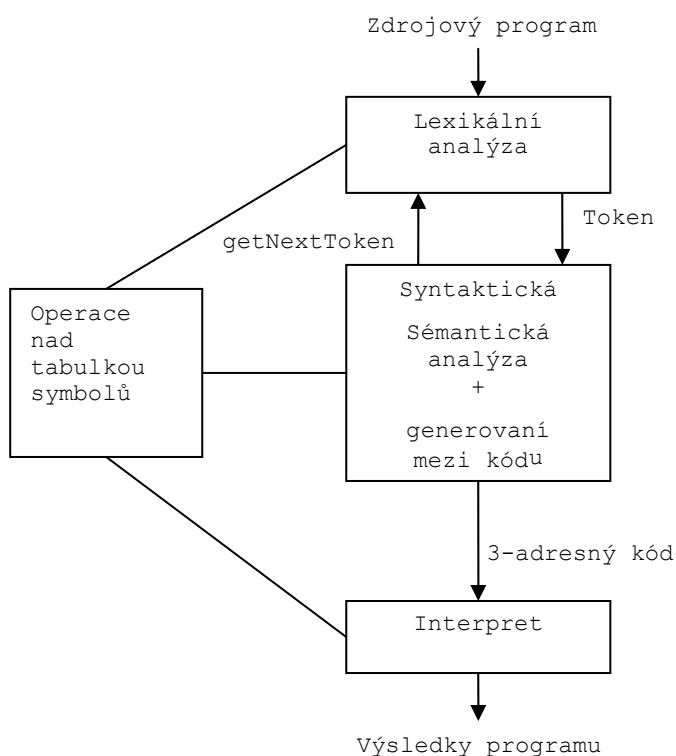
Úvod

Tato dokumentace popisuje náš způsob implementace překladače imperativního jazyka IFJ07.

V kapitole 2 se nachází pohled na projekt z hlediska předmětu IFJ. Je zde popsána zejména implementace jednotlivých částí překladače – lexikálního analyzátoru, syntaktického analyzátoru a interpretu. Dále je zde uvedena podkapitola věnující se rozdělení práce v týmu a naší vzájemné komunikaci.

V kapitole 3 se nachází pohled na projekt z hlediska předmětu IAL. Je zde rozebráno řešení řadící funkce Merge sort, implementace tabulky symbolů pomocí AVL stromu a vyhledávání klíčových slov v seřazeném poli. I tato část je doplněna demonstračními obrázky a komentovanými úryvky kódu.

V kapitole 4 je zhodnoceno celkové řešení včetně způsobu zvolené týmové práce. Nachází se zde také vyhodnocení provedených testů a jejich výsledků.



Obr. č. 1: Fáze interpretu

Kapitola 2

Projekt z hlediska předmětu IFJ

V této kapitole se nachází zejména popis jednotlivých fází překladu doplněný odkazy do zdrojového kódu, včetně úryvků. Dále je zde také podkapitola věnující se týmové práci.

2.1 Lexikální analýza

Lexikální analyzátor je vstupní částí překladače. Je umístěn v souboru *scanner.c* a provádí transformaci textu zdrojového programu na posloupnost lexikálních jednotek – tokenů. Ty jsou získávány funkcí *getNextToken*, kterou volá syntaktický analyzátor. Tato funkce vrátí typ tokenu a v případě, že načítaný lexém reprezentoval identifikátor, předá navíc ukazatel do tabulky symbolů, kam lexikální analyzátor uloží jméno proměnné. Pokud byla načítána ze souboru konstanta (celočíselná, desetinná, anebo řetězcová), lexikální analyzátor opět vrátí typ a zároveň uloží do tabulky symbolů vygenerované jméno pro konstantu (začínající znakem \$), její hodnotu a typ.

Implementace lexikálního analyzátoru je řešena deterministickým konečným automatem (viz Příloha A). Znaky ze vstupního souboru se načítají pomocí funkce *fgetc* a jsou ukládány pomocí funkcí pro práci s nekonečně dlouhými řetězci, jež jsou implementovány v souboru *str.c*.

Dojde-li k situaci, že je načten znak nepatřící danému tokenu, je navrácen zpět pomocí funkce *ungetc*. Komentáře a bílé znaky jsou přeskakovány.

2.2 Syntaktická analýza

Syntaktická analýza tvoří jádro překladače. V naší implementaci projektu jde o syntaxi řízený překlad, proto syntaktický analyzátor provádí kromě syntaktické i sémantickou analýzu, na základě které generuje tříadresný kód.

Pro syntaktickou analýzu kostry programu i jednotlivých příkazů jsme využili analýzy „shora dolů“. Toto řešení se netýká zpracování výrazů, které jsme implementovali pomocí analýzy „zdola nahoru“. Jazyk IFJ07 je definován tak, že veškeré výrazy jsou z obou stran ohraničeny znakem dolaru, proto jejich identifikace v kódu nečiní žádný problém a je tedy možné syntaktickou analýzu tímto způsobem rozdělit, což problematiku analýzy značně zjednodušuje.

2.2.1 Syntaktická analýza „shora dolů“

Syntaktická analýza „shora dolů“ je zapsaná v souboru *parser.c*. Syntaktický analyzátor postupně funkcí *getNextToken* získává od lexikálního analyzátoru tokeny a jejich sekvence pak následně vyhodnocuje pomocí gramatických pravidel.

Pro syntaktickou analýzu „shora dolů“ jsme si definovali následující LL gramatiku bez ϵ -pravidel:

```

1: <PROGRAM> -> <DECLARATION> <BODY>
2: <DECLARATION> -> int ID ; <DECLARATION>
3: <DECLARATION> -> double ID ; <DECLARATION>
4: <DECLARATION> -> string ID ; <DECLARATION>
5: <DECLARATION> -> {
6: <BODY> -> <STATEMENT> <BODY>
7: <BODY> -> }
8: <STATEMENT> -> sort ID ;
9: <STATEMENT> -> ID = <vyraz> ;
10: <STATEMENT> -> if <vyraz> <BODYORSTAT> else <BODYORSTAT>
11: <STATEMENT> -> while <vyraz> <BODYORSTAT>
12: <STATEMENT> -> read ID <IDS>
13: <STATEMENT> -> write <vyraz> <EXPRESSIONS>
14: <BODYORSTAT> -> { <BODY>
15: <BODYORSTAT> -> <STATEMENT>
16: <IDS> -> , ID <IDS>
17: <IDS> -> ;
18: <EXPRESSIONS> -> , <vyraz> <EXPRESSIONS>
19: <EXPRESSIONS> -> ;

```

Pro tyto pravidla jsme si zkonstruovali LL tabulku, ze které jsme vycházeli při návrhu algoritmu samotného syntaktického analyzátoru:

	ID	int	double	string	sort	if	while	read	write	{	}	,	;
<PROGRAM>		1	1	1						1			
<DECLARATION>		2	3	4						5			
<BODY>	6				6	6	6	6	6		7		
<STATEMENT>	9				8	10	11	12	13				
<BODYORSTAT>	15				15	15	15	15	15	14			
<IDS>												16	17
<EXPRESSIONS>												18	19

Jako metodu implementace LL analyzátoru jsme zvolili rekurzivní sestup. Jeho největší výhodou je jednoduchá implementace. V samotném kódu je každý nonterminál reprezentován samostatnou funkcí. Jednotlivé funkce se navzájem rekurzivně volají, čímž se postupně pokrývá celý strom terminálů. V rámci těchto funkcí probíhá i sémantická analýza (kontrola deklarace proměnných a jejich typu včetně implicitní konverze) a generování tříadresného kódu. Tříadresný kód je zapisován do jednosměrně vázaného seznamu pomocí funkce *TapeInsert*. Tento kód následně převezme a zpracuje interpret.

2.2.1 Syntaktická analýza „zdola nahoru“

V případě, že syntaktický analyzátor ze souboru *parser.c* narazí na token \$ (uvození výrazu), zavolá funkci *expressionParser* ze souboru *expr.c*. Tato funkce vyhodnocuje a zjišťuje relevantnost výrazů za pomoci precedenční tabulky a zásobníku. Funkce pracuje tak dlouho, dokud opět nenarazí na token \$ (v případě bezchybného vstupu).

Způsob vyhodnocování za pomoci precedenční tabulky byl zvolen pro jeho snadnější implementaci.

[illegible]

Princip precedenční syntaktické analýzy se dá popsat následujícím algoritmem:

- Na vrchol zásobníku vlož symbol $\$$ a přijmi token.
- Necht' probíhá cyklus:
 - Podle obsahu políčka precedenční tabulky na souřadnicích $[b, a]$, kde b je nejvrchnější terminální symbol na zásobníku a a je aktuální symbol na vstupu, rozhodni:

Znak **n**:

syntaktická chyba

Znak $<$ nebo $=$:

Aktuální symbol ***a*** ze vstupu vlož na zásobník. Přijmi další token a jdi na podmínku cyklu.

Znak >:

Pokud je na vrcholu terminální identifikátor, změn jej na nonterminální identifikátor a jdi na podmínku cyklu.

Pokud je na vrcholu zásobníku (E), pak odstraň závorky a jdi na podmínku cyklu.

Dále se řeší již jen binární operátory. To, že se postupuje dle priority operátorů, je zajištěno díky dobrému návrhu precedenční tabulky.

- Pokud se **a** i **b** rovnají \$, pak syntaktická analýza proběhla v pořádku. V opačném případě opakuj cyklus.

2.3 Interpret

Po provedení syntaktické analýzy následuje interpretace vygenerovaného mezikódu. V našem případě jde o tříadresný kód, který je reprezentován strukturou *TapeItem*, přičemž posloupnost těchto tříadresných instrukcí je zřetěžena do lineárního seznamu.

Kód interpretu je umístěn v souboru *interpret.c*, kde se nachází také funkce *interpretation*. Ta postupně čte jednotlivé položky s tříadresným kódem (struktura *TapeItem*) a na základě přečtené instrukce provádí příslušnou operaci s danými operandy, což jsou odkazy do tabulky symbolů. Ve zmíněné funkci *interpretation* potřebujeme pro zahájení čtení instrukcí znát odkaz na první položku s tříadresným kódem. Tento odkaz je uchován v položce *first* struktury *Tape*.

```
// struktura položky na pásce
typedef struct tapeItem
{
    Instruction instruction;
    TSItem *opr1;
    TSItem *opr2;
    TSItem *dest;
    struct tapeItem *nextItem;
} TapeItem;
```

```
// struktura pásky
typedef struct
{
    TapeItem *first;
    TapeItem *last;
} Tape;
```

Instrukční sada, kterou jsme v programu použili, se dá rozdělit takto:

- aritmetické instrukce: I_ADD, I_SUB, I_MUL, I_DIV, I_POW, I_NOT
- relační instrukce: I_LESS, I_GREATER, I_LE, I_GE, I_EQ, I_NEQ
- řídící instrukce: I_GOTO
- přenosové instrukce: I_READ, I_WRITE, I_ASSIGN
- speciální instrukce: I_SORT, I_INT_TO_DOUBLE, I_LAB

2.4 Týmová práce

Z hlediska rozdělení práce v týmu bylo převzato doporučení z demonstračních cvičení, podle kterého byla práce přidělena jednotlivým osobám takto:



Lexikální analýza



**Syntaktická analýza
shora dolů**



**Syntaktická analýza
zdola nahoru**



Interpret



**Testování
Dokumentace**

Prvotní rozdělení bylo v podstatě dodrženo. Jedinou výjimkou je sloučení fáze tvorby interpretu s testováním a tvorbou dokumentace tak, že na nich pracovaly dvě osoby zároveň. To poskytlo možnost vyzkoušení zajímavé techniky extrémního programování – párového programování. Ačkoliv je zřejmé, že fáze tvorby interpretu není zdaleka tou nejtěžší, rychlost její realizace přesto překvapila. Navíc ve vztahu k testování je přítomnost více osob již tradičně zárukou rychlejšího nalezení chyb, proto použití této techniky bylo jednoznačně vyhodnoceno jako efektivní.

Co se týče komunikace v týmu, pak základ tvořilo několik schůzek celé skupiny, kdy jsme si ujasnili, co zhruba jednotlivé fáze obnáší a jaký je stav realizace projektu. Klíčová pak byla zejména dohoda společného rozhraní.

Kapitola 3

Projekt z hlediska předmětu IAL

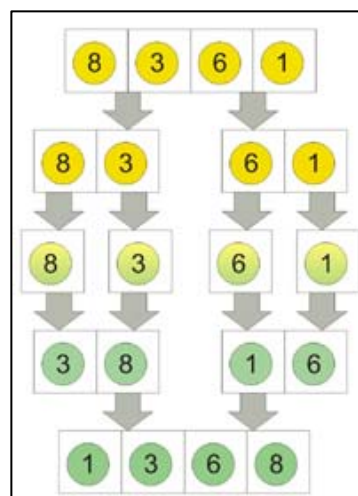
Zvolená varianta zadání požaduje řešení řadícího algoritmu pomocí metody Merge Sort. Tabulka symbolů má být implementována AVL stromem a vyhledávání klíčových slov v tabulce má být provedeno jako vyhledávání v seřazeném poli.

3.1 Metoda řazení Merge Sort

Merge sort je řadící algoritmus vyvinutý v roce 1945 matematikem von Neumannem. Jde o algoritmus založený na metodě Rozděl a panuj. V rámci našeho projektu lze jeho implementaci nalézt v souboru *mergesort.c*, přičemž funkce *mergesort* rozdělí množinu prvků na dvě přibližně stejně velké části. Tyto části se následně sekvenčním způsobem seřadí a nakonec dojde ke spojení seřazených částí do výsledné seřazené množiny – toto slučování proběhne ve funkci *merge*. Uvedený princip je implementován rekurzivně (viz obr.č.2).

Rozdělení neseřazených dat proběhne stejně jako spojování seřazených částí ($\log_2 n$)-krát. Časová složitost slévání je n . Asymptotická složitost je tedy $\Theta(\log n) \cdot \Theta(n) = \Theta(n \cdot \log n)$. Metoda je stabilní.

Oproti ostatním řadícím algoritmům podobné rychlostní třídy (Heap sort, Quick sort) vyžaduje Merge sort pomocné pole velikosti n . Lze sice provést modifikaci algoritmu tak, aby nebylo pomocné pole potřeba, ovšem jen za cenu drastického snížení výkonu u delších polí, kde pak je tato metoda prakticky nepoužitelná. Ovšem i ve standardním (a námi použitým) řešení je tento algoritmus oproti výše uvedeným pomalejší.



Obr. č. 2: Implementace Merge sortu

3.2 Tabulka symbolů

Tabulka symbolů je implementována jako AVL strom. AVL strom je často používanou modifikací dokonale vyváženého stromu, avšak na rozdíl od něj není rekonfigurace prvků za účelem vyvážení tak náročná, jelikož je použita slabší definice vyváženosti – výšková vyváženost. Podle této definice je strom vyvážený právě tehdy, když se výšky jeho podstromů rovnají nebo liší nejvýše o 1. Autoři AVL stromu, Adelson-Velskij a Landis, dokázali, že jeho výška nikdy nepřekročí 1.45-násobek výšky dokonale vyváženého stromu. Z toho vyplývá, že asymptotická časová složitost přidávání, vyhledávání nebo zrušení uzlu je $\Theta(\log n)$.

V rámci našeho projektu používáme jedinou tabulku symbolů pro proměnné, pomocné proměnné i konstanty. Implementace tabulky symbolů se nachází v souboru *avl.c* stejně jako funkce pro vkládání symbolu do tabulky – *insertToTS* a vyhledání v symbolu v tabulce – *searchTS*. Symbol je pak reprezentován datovou strukturou *TSItem*, jak je znázorněno zde:

```
// Struktura položky AVL stromu
typedef struct TSItemStruct
{
    tString *name;
    TSItemType type;
    TSItemValue value;
    int balance;
    struct TSItemStruct *left;
    struct TSItemStruct *right;
} TSItem;
```

```
// Hodnota id nebo konstanty
typedef union
{
    int intValue;
    double doubleValue;
    tString *stringValue;
} TSItemValue;
```

```
// Typ id nebo konstanty
typedef enum
{
    tNone,
    tInt,
    tDouble,
    tStr
} TSItemType;
```

3.3 Vyhledávání klíčových slov

Vyhledávání klíčových slov probíhá v seřazeném poli. Implementace je umístěna v souboru *keyword_table.c*. Funkce *sentKeyword* na základě předaného řetězce postupně prochází tabulku klíčových slov. Pokud je řetězec stejný jako aktuálně přečtené klíčové slovo, vrátí funkce typ přečteného klíčového slova (např. *tTypeDouble* či *tIf*, ...). V případě, že je aktuálně přečtené klíčové slovo lexikograficky větší než hledaný řetězec, vyhledávání se ukončí a funkce vrátí typ *tIdentifier*. Protože je totiž pole seřazené, víme, že dále již řetězec určitě nenajdeme.

Kapitola 4

Závěr

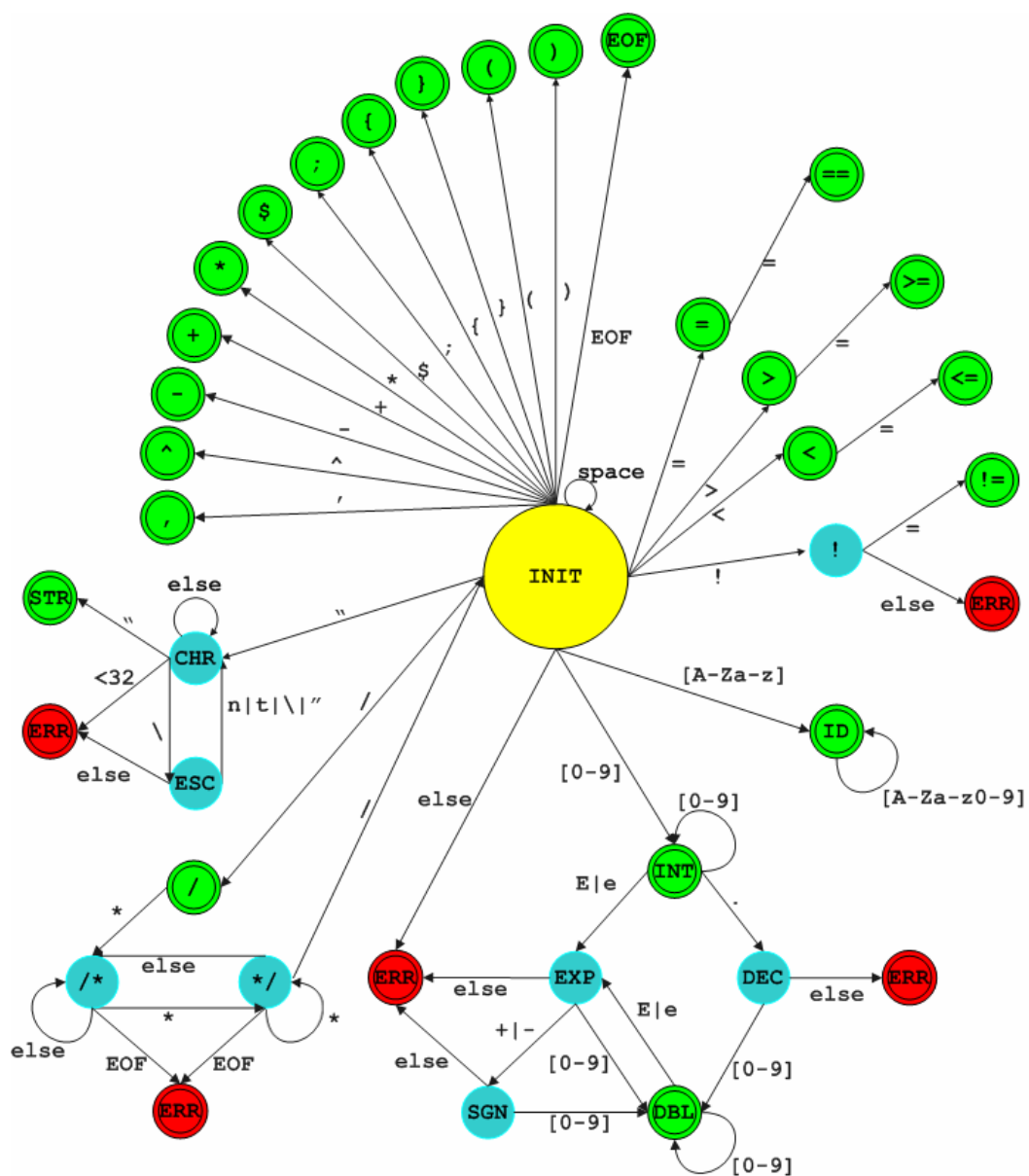
Při řešení tohoto projektu jsme si prakticky vyzkoušeli tvorbu překladače imperativního jazyka. Z hlediska předmětu IFJ jsme se naučili implementovat lexikální analyzátor, syntaktický analyzátor (zahrnující i sémantickou analýzu a generování tříadresného kódu) a interpret. Důležitá byla zejména volba rozhraní, které hrálo klíčovou roli při spojování jednotlivých částí do výsledného celku. Z hlediska předmětu IAL jsme získali znalosti práce s AVL stromem a jeho vyvažováním a také si vyzkoušeli implementaci řadícího algoritmu Merge sort.


Uvědomujeme si, že tyto nově nabyté zkušenosti mají významné uplatnění v praxi, proto jsme tento projekt nevzali na lehkou váhu a po jeho dokončení máme pocit, že dané problematice lépe rozumíme. Kromě toho jsme si vyzkoušeli týmovou práci. Díky znalostem z předmětu IUS a doporučení ohledně rozdělení práce v týmu z prvních přednášek IFJ jsme nenarazili na žádné vážnější komunikační ani jiné problémy. Nicméně jsme si alespoň ověřili, že při dodržení určitých zásad je možné dosáhnout efektivní týmové práce.

Zvolené řešení projektu klade důraz na dodržení všech požadavků zadání. Program byl otestován na řadě zkušebních zdrojových kódů, které jsme si sami napsali, přičemž výstup po odstranění drobných chyb odpovídal správným výsledkům. Úspěšně jsme jej otestovali v prostředí operačních systémů Windows a Linux.

Příloha A

Graf lexikálního analyzátoru



POZN: Chybový stav ERR
reprezentuje jediný stav.
 Vícekrát je uveden pouze pro
lepší přehlednost.