Dokumentace k projektu pro předměty IFJ a IAL

Implementace interpretu imperativního jazyka IFJ14

Tým 085, varianta a/3/I

10. prosince 2014

Autoři: Vojtěch Bastl, xbastl03

Zdeněk Březina, xbrezi15

Marek Dokulil, xdokul03

Jiří Fňukal, xfnuka01

Lukáš Hubl, xhubll00

Fakulta Informačních Technologií

Vysoké Učení Technické v Brně

Obsah

Úvod

Tématem této práce je implementace překladače pro jazyk IFJ14.Překladač byl rozdělen na pět jednotlivých problémů, tak aby mohla být implementace rozdělena mezi členy týmu.

Překladač byl po vzájemné domluvě rozdělen na tyto části: Lexikální analýza, Syntaktická analýza bez zpracování výrazů, Syntaktická analýza pro výrazy, složitější struktury a interpret.

Vzhledem k tomu, že tyto podproblémy jsou spolu úzce spjaty, implementace vyžadovala neustálou spolupráci a komunikaci v týmu. Za výjimku lze považovat lexikální analýzu, která zpracovává samotný vstupní soubor, a proto musí být implementována před řešením ostatních problémů. Následně byla využita při řešení zbylých částí.

1. Implementace

V této kapitole se zaměříme na řešení problémů jednotlivých částí překladače a zároveň implementace zadaných algoritmů provyhledávání a řazení v řetězci.

1.1 Lexikální analýza

Lexikální analýza byla implementovaná jako konečný automat v souboru scanner.c a je volaná jako funkce get_token, která načítá jednotlivé znaky ze zdrojového souboru a vrací typ tokenu, které jsou spolu se stavy konečného automatu definované v souboru scanner.h. Schéma automatu je vyobrazeno v příloze.

1.2 Knuth-Morris-Prattův algoritmus (KMP)

Tento algoritmus ke své práci využívá konečný automat. Algoritmus se dá rozdělit na dvě části. V první části algoritmus prochází vzorek, který vyhledáváme a do pole Fail ukládá jednotlivé indexy, ke každé pozici ve vzorku pouze jeden, které udávají, na kterou pozici se máme vrátit, dojde-li k neshodě na aktuální pozici. V druhé části algoritmus prochází vzorek a zároveň řetězec, v kterém vyhledáváme. Pokud nedojde ke shodě, posuneme se ve vzorku na pozici, která je v poli Fail pro aktuální pozici a znova porovnáváme vyhledávaný vzorek se zadaným řetězcem. Algoritmus tedy proběhne m+n krát, přičemž m je délka řetězce v kterém vyhledáváme a n je délka vzorku, který vyhledáváme. V nejhorším případě se můžeme rovnat n a algoritmus vlastně proběhne 2n krát a má tedy lineární složitost.

Algoritmus byl implementován jako jedna funkce v souboru ial.c a byl pojmenován find. Algoritmus nám vrací pozici, kde došlo ke shodě vzorku s řetězcem a při neshodě vrací 0.

1.3 Shell sort – řazení se snižujícím se přírůstkem

Výše zmíněný algoritmus, si rozděluje řetězec na podřetězce složené se znaků vzdálených od sebe o krok. Shell sort začíná s krokem rovným polovině délky řetězce. Na jednotlivé podřetězce aplikuje Booble sort, to znamená že, každý podřetězec prochází od počátku do konce a vyměňuje prvky, tak aby prvek s větší ordinální hodnotou byl vždy vpravo. Booble sort skončí, až když při průchodu nedojde k výměně žádné dvojice prvků. Po seřazení všech podřetězců se krok zmenší na polovinu a znovu se na podřetězce aplikuje Booble sort. Algoritmus skončí, pokud se krok rovná 0. Algoritmus byl implementován jako funkce sort v souboru ial.c a vrací seřazený řetězec. Shell sort je nestabilní metoda, která pracuje in-situ,

to znamená, že nemusí respektovat pořadí položek se stejným klíčem a k seřazení nepotřebuje další pomocné pole.

1.4 Syntaktická analýza

Zkoumání, zda byl vstupní program správně syntakticky napsán, musíme rozdělit na dvě části. První část je kontrola syntaxe neobsahující výrazy. Výrazy poté řešíme pomocí druhé části. Tyto dvě části spolu musí vzájemně komunikovat a musí být vhodně použity na jednotlivé části vstupního souboru.

1.4.1 Syntaktická analýza neobsahující výrazy

Problém správnosti syntaxe je řešen pomocí vhodné LL gramatiky a následně její implementace pomocí rekurzivního sestupu. LL gramatika je navržena vzhledem k zadání a její každý jednotlivý neterminál je charakterizován pomocí funkce. Pokud k jednomu neterminálu přísluší více LL pravidel, jejich rozlišení je zajištěno pomocí větvení. Pokud se zjistí, že musí být zpracováván výraz, program zavolá modul na zpracování výrazu a následně se znovu vrátí na rekurzivní sestup LL gramatikou. Tato technika je uplatněna i na volání funkcí ve vstupním programu, kdy nejprve syntaktická analýza zjistí, zda daná funkce je deklarována a následně při každém parametru zavolá zpracování výrazu.Syntaktická analýza zároveň pokrývá část typové kontroly konkrétně přiřazení a parametry volání funkcí včetně jejich návratového typu. Samotné znění LL gramatiky je přibaleno v příloze.

1.4.2 Syntaktická analýza obsahující výrazy

Precedenční syntaktická analýza používá pro svoji činnost precedenční tabulku. Pomocí tabulky se sleduje vztah mezi aktivním prvkem na zásobníku a aktuálním tokenem na vstupu. Mohou nastat 4 situace. Tyto situace se rozlišují pomocí následujících symbolů:

- = pushnutí tokenu ze vstupu na zásobník
- < pushnutí zarážky a následně pushnutí tokenu ze vstupu
- > redukce na zásobníku
- prázdné políčko – chyba

Precedenční syntaktická analýza probíhá do té doby, než nastane situace, kdy aktivní prvek na zásobníku je $ a vstupní token je také $.

Precedenční tabulka:

Řádky precedenční tabulky tvoří aktivní prvky na zásobníku a sloupce tvoří aktuální vstupní token. Tvoří se v pěti krocích. První krok je priorita operátorů. Když op1 má vyšší prioritu než op2, potom platí op1>op2. Druhý krok je asociativita operátoru. Tento krok se používá pro operátory se stejnou prioritou. V našem případě mají operátory levou asociativitu, tudíž platí: op1>op2>op3. V třetím kroku se zpracovávají identifikátory. V podstatě lze říci, že sledujeme, co může být za a před identifikátorem. Vezměme si například identifikátor i a operátor +. + může být před i, tudíž platí: + < i. + může být také za i a platí: i > +. Ve čtvrtém kroku se zpracovávají závorky. Postupuje se zde stejně jako u identifikátoru (co může být před a co může být za závorkou). V posledním kroku se řeší ukončovací znak. Nechť op je libovolný operátor, potom platí: $ < op a zároveň op > $.

Průběh syntaktické analýzy pro výrazy:

Nejprve je načten první token, který se pomocí podmínek vyhodnotí, zda s ním výraz může začínat. Pokud ne, vyhodnotí se tato situace jako chyba. Token je testován, zda se jedná např.: o konstantu, identifikátor, závorku atd. a podle výsledků jsou nastaveny příznaky. Jestliže je možné, aby výraz daným tokenem začínal, pushne se na zásobník spolu se zarážkou. Dále jsou pomocí cyklu while načítány další tokeny, dokud není načten ukončovací znak. Ukončovací znaky jsou: DO, THEN, END, středník a čárka. Po načtení tokenu se podle precedenční tabulky rozhodne a následně switche, jaká operace má být provedena. V případě shiftingu se zavolá funkce shifting. V případě, kdy má být provedena redukce se zavolá funkce reduction a následně funkce shifting pro pushnutí tokenu na zásobník.

Při redukci libovolné aritmetické nebo porovnávací operace se zavolá funkce pro generování 3 adresného kódu a výsledek se uloží do pomocné struktury. Zároveň je provedena typová kontrola obou operandů.

Po kompletním zredukování výrazu se ukončí funkce a jako návratová hodnota je použita struktura, ve které je její typ a název.

1.5 Tabulka symbolů

Tabulky symbolů jsou implementovány jako vyhledávací binární stromy. Jako klíč se v nich používá název identifikátoru, který musí být jedinečný. K porovnání klíčů se používá námi implementovaná funkce key, která porovnává dva řetězce a rozhodne, jaký z nich je větší. Pro

přehlednost a zjednodušení práce s tabulkou symbolů se vytváří jedna globální tabulka symbolů a následně lokální tabulka symbolů pro každou funkci.

U obou druhů tabulek symbolů obsahuje každý uzel ukazatel na levý a pravý podstrom uzlu a strukturu tData. Ve struktuře tData se nachází název (klíč), typ identifikátoru a pomocná proměnná, která určuje jeho definovanost.

Globální tabulka symbolů má navíc ukazatel na lokální tabulku symbolů a proměnou arg typu string. Tyto hodnoty jsou nastaveny pouze v případě, že daný prvek v globální tabulce symbolů je typu funkce. Ukazatel tak slouží pro propojení uzlu, který je typu funkce, a příslušné lokální tabulky symbolů. Proměnná arg uchovává informace o typu argumentů funkce. Lokální tabulka symbolů obsahuje navíc proměnou sloužící k zjištění pořadí argumentů. To je nutné pro jejich typovou kontrolu.

Pro používání tabulek symbolů jsou naimplementovány funkce GlobTableInit, GlobItemInsert, GlobTableInsert, LokTableInsert, tableSearch, tableSearchLok, tableSearchGlob, TableFree, TableFreeLok a ItemFreeAktu. Některé z těchto funkcí se mezi sebou vzájemně volají. Například funkce tableSearch, která slouží pro prohledání lokální a poté případně i globální tabulky symbolů, využívá funkce tableSearchLok a tableSearchGlob.

1.6 Rámce

Rámce jsou stejně jako tabulky symbolů implementovány jako vyhledávací binární stromy. Slouží k uchovávání hodnot u daných prvků a jsou vytvářeny až v momentě, kdy máme už všechny tabulky symbolů kompletní.

Struktura globálního a lokálního rámce se nijak neliší. To umožňuje zjednodušené předávání argumentů. Rozdílný je pouze způsob jejich uchování. Globální rámec se zachová pomocí globální proměnné. Lokální rámce se ukládají na zásobník, čehož se využívá při volání funkcí v programu.

Každý uzel rámce obsahuje název (klíč), typ identifikátoru, ukazatel na levý a pravý podstrom uzlu a strukturu tHodnota. V této struktuře je zachována číselná hodnota, nebo řetězec. Dále pak definovanost a pořadí argumentu.

Rámce umožňují na rozdíl od tabulek symbolů vkládání dalších uzlů. Tyto nové uzly se využívají jako pomocné proměnné a musí vlastnit unikátní název. Binární stromy tabulek symbolů a rámců jsou do doby přidání nového uzlu naprosto shodné.

Pro rámce jsou implementovány funkce RamecInit, GlobRamecInit, VytvorRamec, VytvorRamecGlob, SearchRamecPoradi, SearchRamec, SearchRamecPom, PridatHodnotu, PridatPom a FreeRamec. Pro práci se zásobníkem pak funkce PushR a PopTopR.

1.7 Interpret

Poslední částí překladače je interpret, který má na starosti vykonávání programu, který je zapsaný ve zdrojovém souboru.

Při syntaktické a sémantické analýze se generuje tříadresový kód, který se ukládá do seznamu. Tento seznam je implementován jako jednosměrný lineární seznam, který v sobě uchovává hodnoty o typu instrukce, názvu proměnných, a hodnot konstant. V tomto se generují i návěští pro podmínky a cykly. Také zde jsou implementovány funkce pro vyhledávání pozice v seznamu, které jsou důležité při podmínkách, cyklech a volání funkcí.

Po úspěšné syntaktické a sémantické analýze se začne vykonávat program, to má na starosti funkce Interpret. Tato funkce prochází lineárně seznamem instrukcí a vykonává tříadresový kód. Načte první položku seznamu a na základě instrukce provede určitou operaci s operandy. Po vykonání se načte další instrukce, takhle postupuje až do konce seznamu. Výjimkou jsou instrukce pro skoky. Ty provádí skok na určitou položku v seznamu na základě návěští. Interpret také pracuje s rámci, ty se vytváří, když je zavolána nějaká funkce. V tomto rámci se uchovávají hodnoty aktuálních proměnných, se kterými interpret pracuje. Další funkcí je typová kontrola a ošetření inicializace proměnných.

2 Rozdělení práce v týmu.

2.1 Vojtěch Bastl – Vedoucí týmu

- Syntaktická analýza bez zpracování výrazů
- Implementace tabulky symbolů
- Spojování jednotlivých částí v celek
- Testování překladače

2.2 Zdeněk Březina

- Implementace rámců
- Spolupráce na implementaci tabulky symbolů

2.3 Marek Dokulil

- Lexikální analýza
- Implementace vestavěných funkcí
- Doplnění funkcí v str.c
- Tvorba dokumentace

2.4 Jiří Fňukal

- Interpret

2.5 Lukáš Hubl

- Syntaktická analýza pro výrazy

3 Závěr

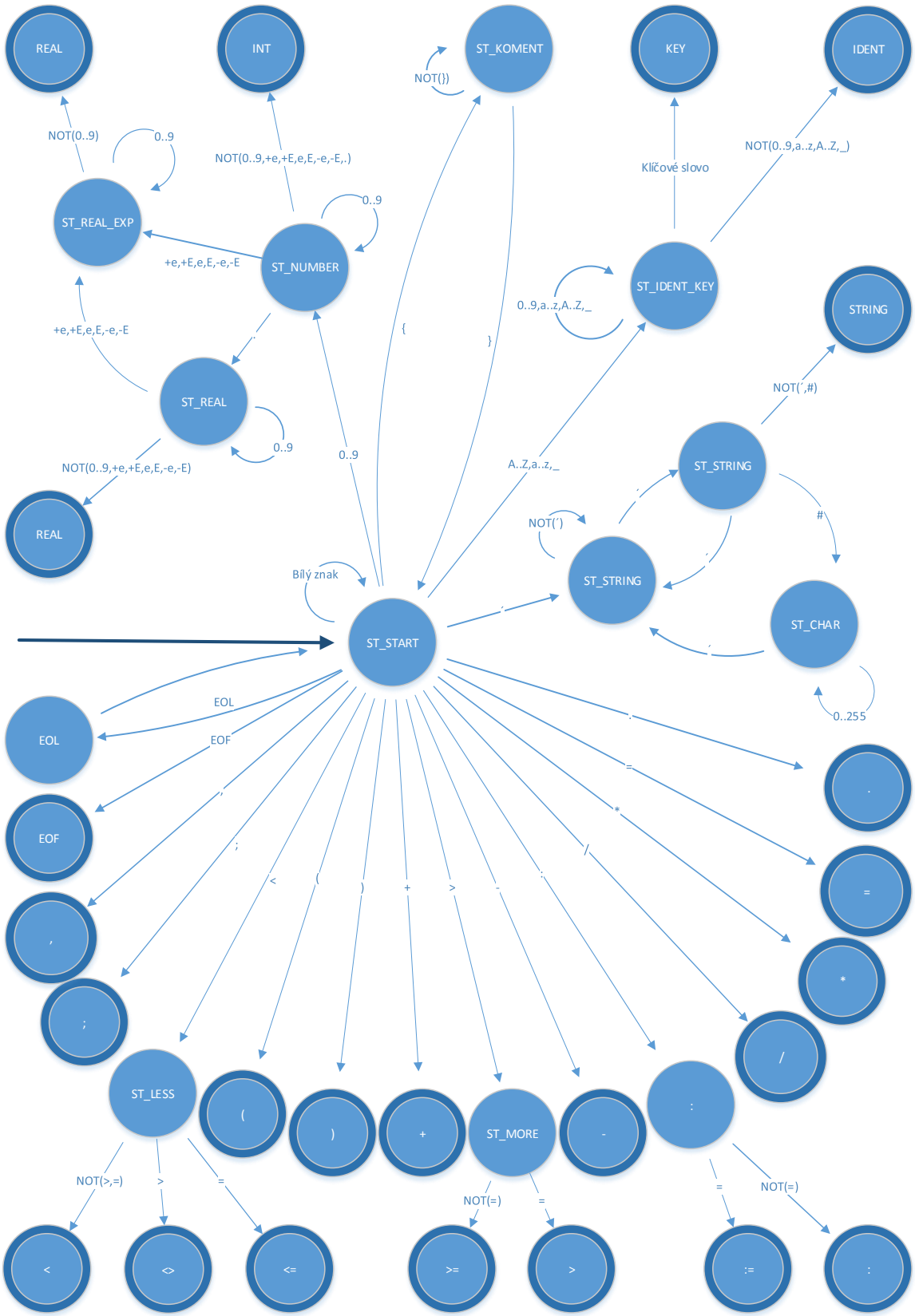
Řešení jednotlivých podproblému, stejně jako následná implementace překladače pro nás byla časově náročná a v některých případech i vcelku obtížná. Přestože jsme v průběhu řešení projektu museli čelit nejednomu problému, povedl se nám projekt dokončit ve stanovený čas. Vzhledem k časové nekompatibilitě si jsme vědomi několika nedostatků.

Práce v týmu pro nás byla velkou zkušeností. Naučili jsme se pracovat a komunikovat v týmu, což je pro nás přínosem do budoucna. Sdílení a spolupráce na jednotlivých souborech byla usnadněna pomocí verzovacího systému GitHub.

Při řešení projektu jsme použili některé soubory z jednoduchého interpretu, který nám byl poskytnut. Jde především o soubory str.c a str.h, přičemž str.c byl doplněn o některé potřebné funkce. Dále jsme použili funkci generateVariable, která je taktéž převzata z jednoduchého interpretu.

Příloha A

Konečný automat



Příloha B

LL gramatika

1.	<START>	->	<GLOBDEK><FUNC><SLOZ> .
2.	<FUNC>	->	eps
3.	<FUNC>	->	function id (<ARG>) : <TYPE> ; <FORWARD>
4.	<FORWARD>	->	<DEK><SLOZ> ;<FUNC>
5.	<FORWARD>	->	forward ; <FUNC>
6.	<ARG>	->	eps
7 .	<ARG>	->	id : <TYPE><ARGDAL>
8 .	<ARGDAL>	->	eps
9.	<ARGDAL>	->	; id : <TYPE><ARGDAL>
10.	<CYKLUS>	->	while <VYRAZ> do <SLOZ>
11.	<KDYZ>	->	if <VYRAZ> then <SLOZ><ELSE>
12.	<ELSE>	->	else <SLOZ>
13.	<POKYN>	->	<KDYZ>
14.	<POKYN>	->	<CYKLUS>
15.	<POKYN>	->	<PRIKAZ>
16.	<POKYN>	->	READLN(id)
17.	<POKYN>	->	WRITE(<VYPIS>)
18.	<POKYN>	->	<SLOZ>
19.	<SLOZ>	->	begin <PRVNI> end
20.	<PRVNI>	->	eps
21.	<PRVNI>	->	<POKYN><DALSI>
22.	<DALSI>	->	eps
23.	<DALSI>	->	; <POKYN><PRVNI>
24.	<PRIKAZ>	->	id := <VYRAZ>
25.	<GLOBDEK>	->	var id : <TYPE> ; <GLOBDEKDAL>
26.	<GLOBDEK>	->	eps
27.	<GLOBDEKDAL>	->	eps
28.	<GLOBDEKDAL>	->	id : <TYPE> ; <GLOBDEKDAL>
29.	<TYPE>	->	integer
30.	<TYPE>	->	boolean
31.	<TYPE>	->	string
32.	<TYPE>	->	real
33.	<VYPIS>	->	term <DVYPIS>
34.	<DVYPIS>	->	, <VYPIS>
35.	<DVYPIS>	->	eps
36.	<DEK>	->	var id : <TYPE> ; <DEKDAL>
37.	<DEK>	->	eps
38.	<DEKDAL>	->	eps
39.	<DEKDAL>	->	id : <TYPE> ; <DEKDAL>

Příloha C

Precedenční tabulka

	*	/	+	-	<	>	<=	>=	=	<>	(	)	id	$
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
=	<	<	<	<	>	>	>	>	>	>	<	>	<	>
<>	<	<	<	<	>	>	>	>	>	>	<	>	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
id	>	>	>	>	>	>	>	>	>	>		>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	

- Aktuální token na vstupu.
- Aktivní token na zásobníku.

pravidla redukcí
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)
E -> id

Příloha D

Metriky kódu

Počet řádků zdrojového textu: 5500

Počet Zdrojových souborů: 18

Počet funkcí: 87