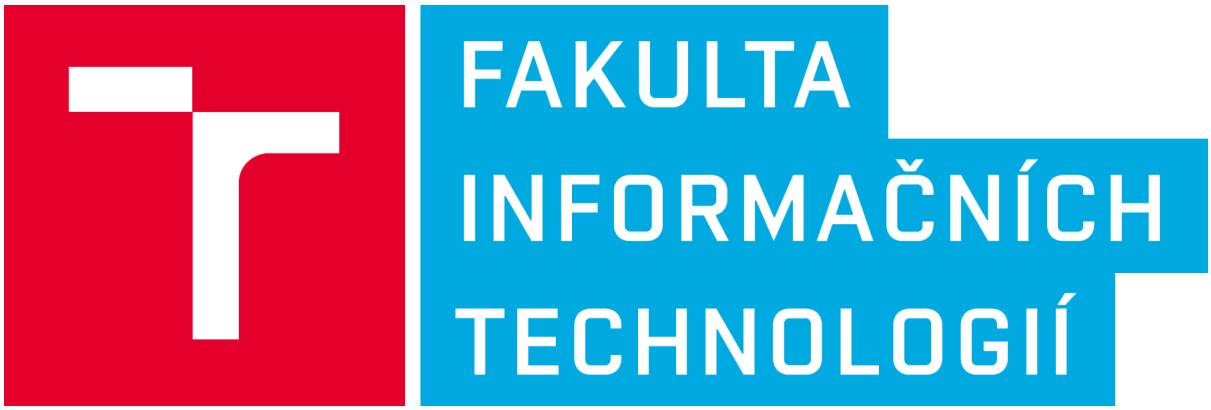
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Formální jazyky a překladače

Dokumentace skupinového projektu

Tým 087, varianta I

Martin Vlach (xvlach18) - vedoucí - 25 %
Lukáš Hekrdla (xhekrd01) - 25 %
Martin J. Salamánek (xsalam04) - 25 %
Tomáš Tlapák (xtlapa00) - 25 %

Rozšíření: BASE

1 Úvod

Tato dokumentace popisuje implementaci překladače imperativního jazyka IFJ18, který je podmnožinou jazyka Ruby 2.0. Implementace překladače je náplní společného projektu do předmětů Formální jazyky a překladače a Algoritmy. Projekt je dekomponován na bloky, kterým se blíže budeme věnovat v každé kapitole zvlášť.

2 Lexikální analýza

Hlavním úkolem lexikálního analyzátoru je rozdělit zdrojový soubor na vstupu na posloupnost tokenů na výstupu. Jednotlivé tokeny jsou po jejich zpracování následně předávány syntaktické analýze. Toto rozdělování probíhá na základě regulárních výrazů.

Dalším úkolem scanneru je odstranit ze vstupního souboru řádkové a blokové komentáře a také tzv. bílé znaky, které mohou být do souboru vkládány například za účelem zpřehlednění kódu.

Lexikální analyzátor jsme implementovali jako konečný automat (obr. 2). KA načítá znaky dokud nenarazí na oddělovač a uloží řetězec do tokenu. Pokud během načítání znaků ze vstupu skončí konečný automat v koncovém stavu, vrátí se token daného typu, pokud v koncovém stavu neskončíme jedná se o chybný token (lexikální chyba). Token uchovává informace o načteném řetězci nebo čísla a typu tokenu. V lexikálním analyzátoru jsme se snažili řešit vše, co již řešit šlo, např. rozšíření BASE. Naše implementace lexikálního analyzátoru obsahuje 3 základní funkce. Funkci pro inicializaci lexikálního analyzátoru, načtení dalšího tokenu a poslední pro korektní uvolnění paměťových zdrojů využívaných lexikálním analyzátorem.

3 Syntaktická analýza

Implementaci syntaktického analyzátoru jsme řešili metodou rekurzivního sestupu implementovaného dle sestavené LL gramatiky (tab. 1) a kombinací precedenční syntaktické analýzy pro vyhodnocování výrazu. Syntaktickou analýzu jsme řešili dvouprůchodově, abychom v druhém průchodu rozlišili identifikátory funkcí od indentifikátorů proměnných. Na úrovni syntaktické analýzy probíhá komunikace s lexikálním analyzátorem, pomocí kterého získáváme další tokeny pro simulaci derivačního stromu, abychom zjistili, zda je zdrojový program správně syntakticky zapsán. Dále probíhá komunikace s tabulkou symbolů.

3.1 Precedenční analýza

Precedenční syntaktickou analýzu voláme ze syntaktického analyzátoru (v našem případě je součástí parseru), když je třeba vyhodnotit výraz. Implementace je založena na precedenční tabulce (obr. 1), která je tvořena dvojrozměrným polem. Na základě této tabulky řešíme priority redukcí ve výrazu, což je nezbytné pro správně vyhodnocení výrazu. K tomuto se používá speciální zásobník pro práci s precedenční analýzou.

4 Tabulka symbolů

V naší variantě projektu je tabulka symbolů implementována jako binární vyhledávací strom, který je obsažen v globální tabulce symbolů. Jednotlivé uzly obsahují klíč představující ID proměnné nebo funkce, dále ukazatel na strukturu funkce a ukazatele na levý a pravý podstrom.

Struktura funkce ukládá název funkce, informaci o tom zda je definována, počet parametrů a ukazatel na lokální tabulku symbolů, která je implementována pomocí zásobníku.

Struktura globální tabulky tabulky symbolů uchovává informace o počtu funkcí uložených v globální tabulce symbolů, ukazatel na aktuální funkci a ukazatel na lokální tabulku symbolů, která uchovává proměnné v hlavním těle programu a je implementována jako zásobník.

5 Průběh vývoje

Při vývoji překladače jsme při komunikaci využívali skupinovou konverzaci na Messengeru a Discordu. O víkendech jsme využívali toho, že jsme se mohli scházet osobně a řešit větší problémy a části projektu efektivněji. Hojně jsme využívali metody párového programování, za účelem redundance chyb způsobených překlepy atp. Jako další výhodu vidíme v tom, že jsme mohli využít dvou pokusných odevzdání. Přesto, že s projektem jsme začali relativně včas, ke konci jsme objevili některé chyby spojené s precedenční analýzou a generování instrukcí, které jsme již nestihli vyřešit. Na projektu jsme strávili odhadem 170 hodin.

6 Testování

Během vývoje jsme si vytvářeli sadu automatizovaných testů, které jsme průběžně rozšiřovali dle odvedené práce na jednotlivých částech překladače. To jsme hojně využívali ke kontrole, zda nám přidání nových rozšíření neovlivní již správně fungující moduly. Ačkoliv se vývoj z počátku odehrával primárně v systému Windows (IDE NetBeans, MSYS), s postupem času jsme více začali využívat Linux, ať už kvůli tvoření námi známých Bash skriptů pro unit/integrační testy, či pro možnost využití Unixových utilit, např. Valgrind. K důkladnějšímu testování nás také motivovala dvě pokusná odevzdání, jež jsme s nadšením využili.

7 Rozdělení práce

Rozdělení práce v týmu jsme měli volnější. Většinou jsme projekt řešili společně, kde jsme se také průběžně domlouvali na tom, kdo bude řešit jakou část. Pro komunikace se zdrojovými kódy jsme využívali sdíleného repozitáře na Githubu.

8 Generování cílového kódu if18code

Po zjištění lexikálním syntaktické a sémantické správnosti programu, nastává generování cílového kódu ze seznamu instrukcí.

Právě tato část nám při implementaci činila největší problém, z čehož hlavním důvodem byla časová tíseň, protože jsme se soustředili spíše na ověření správnosti programu a samotné generování

nechali až na poslední chvíli. Strategicky jsme vyhodnotili priority a věnovali největší část úsilí do implementace alespoň základní funkčnosti zmiňované na demonstračním cvičení IFJ.

9 Převzaté zdroje

Při implementaci jsme využili pro práci se stringy vzorovou knihovnu str.c dostupnou ze vzorové implementace.

10 Přílohy

PROGRAM	$\longrightarrow$	PROGRAM-BODY *eof*
PROGRAM-BODY	$\longrightarrow$	STATEMENT PROGRAM-BODY
PROGRAM-BODY	$\longrightarrow$	FUNCTION PROGRAM-BODY
PROGRAM-BODY	$\longrightarrow$	*eps*
FUNCTION	$\longrightarrow$	*def id (* PARAMS *) eol* IN-BLOCK *end*
PARAMS	$\longrightarrow$	*id* PARAMS-NEXT
PARAMS	$\longrightarrow$	*eps*
PARAMS-NEXT	$\longrightarrow$	*, id* PARAMS-NEXT
PARAMS-NEXT	$\longrightarrow$	*eps*
IN-BLOCK	$\longrightarrow$	STATEMENT IN-BLOCK
IN-BLOCK	$\longrightarrow$	*eps*
STATEMENT	$\longrightarrow$	*if* E *then eol* IN-BLOCK *else eol* IN-BLOCK *end*
STATEMENT	$\longrightarrow$	*while* E *do eol* IN-BLOCK *end*
STATEMENT	$\longrightarrow$	*id* = VALUE *eol*
STATEMENT	$\longrightarrow$	VALUE *eol*
STATEMENT	$\longrightarrow$	*eol*
VALUE	$\longrightarrow$	E
VALUE	$\longrightarrow$	*id* PARAMS-CALL
PARAMS-CALL	$\longrightarrow$	*(* FUNCTION-CALL-PARAMS *)*
PARAMS-CALL	$\longrightarrow$	FUNCTION-CALL-PARAMS
FUNCTION-CALL-PARAMS	$\longrightarrow$	*int* FUNCTION-CALL-PARAMS-NEXT
FUNCTION-CALL-PARAMS	$\longrightarrow$	*id* FUNCTION-CALL-PARAMS-NEXT
FUNCTION-CALL-PARAMS	$\longrightarrow$	*string* FUNCTION-CALL-PARAMS-NEXT
FUNCTION-CALL-PARAMS	$\longrightarrow$	*float* FUNCTION-CALL-PARAMS-NEXT
FUNCTION-CALL-PARAMS	$\longrightarrow$	*eps*
FUNCTION-CALL-PARAMS-NEXT	$\longrightarrow$	*int* , FUNCTION-CALL-PARAMS-NEXT
FUNCTION-CALL-PARAMS-NEXT	$\longrightarrow$	*id* , FUNCTION-CALL-PARAMS-NEXT
FUNTCION-CALL-PARAMS-NEXT	$\longrightarrow$	*string* , FUNCTION-CALL-PARAMS-NEXT
FUNCTION-CALL-PARAMS-NEXT	$\longrightarrow$	*float* , FUNCTION-CALL-PARAMS-NEXT
FUNCTION-CALL-PARAMS-NEXT	$\longrightarrow$	*eps*

Tabulka 1: seznam LL pravidel

Tabulka 2: LL tabulka pravidel

	eof	def	id	(	)	eol	end	,	if	then	else	while	do	=	int	string	float	$
PROGRAM	1	1	1			1			1			1						
PROGRAM-BODY	4	3	2			2			2			2						
STATEMENT			14,15 [1]			16			12			13						
FUNCTION		5																
PARAMS			6		7													
IN-BLOCK			10			10	11		10		11	10						
PARAMS-NEXT					9			8										
E [2]																		
VALUE			18															
PARAMS-CALL				19														
FUNCTION-CALL-PARAMS			22		25										21	23	24	
FUNCTION-CALL-PARAMS-NEXT			27		30										26	28	29	

[1] Problém nejednoznačnosti, které pravidlo použít je eliminován pomocí dvouprůchodové implementace.

[2] Při rozkladu na neterminál E se zavolá precedenční analýza.

Obrázek 1: Precedenční tabulka

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

Obrázek 2: deterministický konečný automat