FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Dokumentace k projektu předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ17

Tým 54 varianta II.

Petr Marek (xmarek66) - 34 %
Petr Knetl (xknetl00) - 33 %
Jakub Štefanišin (xstefa22) - 33 %
Matěj Knapovský (xknapo04) - 0 %

Rozšíření - žádné

4. prosince 2017

Obsah

Úvod

Tato Dokumentace popisuje návrh a implementaci překladače imperativního jazyka **IFJ17** do cílového jazyka **IFJcode17**. Jazyk IFJ17 je podmnožinou jazyka FreeBASIC. Naším úkolem bylo napsat překladač a generátor kódu. Dle zadání jsme museli vytvořit tabulku symbolů pomocí hashovací tabulky. Projekt jsme rozdělili do tří částí. Každá část je popsána v samostatné kapitole.

1 Spolupráce v týmu

1.1 Rozdělení práce

Na projektu jsme původně pracovali ve čtyřčlenné skupině. V průběhu semestru jeden z nás odstoupil od řešení projektu z důvodu nedostatku bodů pro zápočet. Proto je finální práce rozdělena pouze mezi tři autory.

- Petr Marek - lexikální analýza, generování kódu
- Petr Knetl - syntaktická analýza, dokumentace
- Jakub Štefanišin - tabulka symbolů

1.2 Komunikace

Pro komunikaci jsme nejčastěji používali **Facebook messenger**, kde jsme měli od začátku projektu zavedenou skupinovou konverzaci. Dále jsme také každý týden v úterý plánovali **osobní schůzky** ve školní knihovně FITu. Také jsme používali freeware program **Teamspeak**, který umožňuje hlasovou komunikaci přes internet.

1.3 Použité vývojářské prostředky

Pro společný vývoj a verzování projektu jsme používali verzovací systém **Git** a jako hosting jsme použili soukromý repozitář na **Githubu**. Na repozitáři jsme vytvořili větev pro každou samostatnou část programu, které jsme na konci vývoje spojili do jedné hlavní větve. pro překlad jsme použili jednoduchý program **Make**. Pro lepší orientaci ve verzovacím systému Git jsme používali grafické uživatelské rozhraní **GitKraken**. Pro samostatný vývoj programu jsme každý používali svoje preferované vývojové prostředí pro jazyk C. Dokumentace byla sepsána pomocí sázecího softwaru LaTeX.

2 Implementace částí projektu

2.1 Lexikální analýza

Lexikálního analyzátor je implementován jako **konečný automat s epsilon přechody** (viz. příloha 1). Ty slouží pro přechod ke koncovému stavu při rozpoznání tokenu, nebo k chybovému stavu pokud není lexém rozpoznán. Token je reprezentován strukturou obsahující informace o lexému a řádku, na kterém se ve vstupním souboru vyskytuje. Lexikální analyzátor na požádání naplní těmito informacemi globální token. Ten se poté zpracovává dalšími částmi překladače.

2.2 Tabulka symbolů

Tabulku symbolů jsme podle zadání implementovali pomocí **Hashovací tabulky**. Tabulku jsme neimplementovali přístupem z předmětu IAL, ale způsobem, při kterém vracíme ukazatel jako návratovou hodnotu funkce. Tenhle přístup byl čistě osobní preferencí a setkali jsme se s ním u druhého projektu z předmětu IJC. Tabulka obsahuje pole ukazatelů na prvky, které tvoří jednosměrně vázané seznamy. Každá položka obsahuje název proměnné nebo funkce, její typ a ukazatel na sktrukturu dat pro daný typ položky a ukazatel na další položku v seznamu. Data o proměnné obsahují její název, datový typ a příznak, jestli byla daná proměnná inicializována. Data funkce obsahují její název, návratový typ, počet, název a typ argumentů, příznak, zda byla definována a ukazatel na lokální tabulku symbolů. Tyto data jsou využita k sémantické kontrole. Sémantické kontroly jsou implementovány vrámci syntaktické analýzy. Využívají funkce na práci s tabulkou symbolů. Mezi hlavní sémantické kontroly patří ověření kompatibility datových typů, parametrů funkcí, deklarací funkcí, definic funkcí a proměnných.

2.3 Syntaktická analýza

Syntaktická analýza je v tomto projektu rozdělena do dvou částí. Hlavní část je **syntaktická analýza rekurzivním sestupem**, která vyhodnocuje syntaktickou správnost celého vstupního kódu kromě výrazů. Výrazy jsou zpracovány pomocí **Precedenční syntaktické analýzy**. Vstupními daty syntaktické analýzy jsou tokeny vytvořené v lexikální analýze. Vygenerovaný token se vždy uloží do globální proměnné a pokud při zpracování tokenu syntaktickou analýzou nedojde k chybě, tak syntaktická analýza požádá lexikální analýzu o generování dalšího tokenu. Tento postup se opakuje do té doby než syntaktická analýza objeví chybu nebo lexikalní analýza dojde na konec vstupního souboru.

2.3.1 Rekurzivní sestup

Rekurzivní sestup je založen na **LL-gramatice** (viz. příloha 2) a pracuje od zhora dolu. Rekurzivní sestup začíná v prvním počátečním pravidle gramatiky. Z počátečního pravidla se volají další pravidla přesně podle toho jak je LL-gramatika definována. V místech kde se ve vstupním kódu vyskytuje výraz, tak rekuzrivní sestup volá precedenční analýzu pro zpracování výrazu. Po vyhodnocení výrazu se syntaktická analýze přesune zpět do rekurzivního sestupu a pokračuje v kontrole.

2.3.2 Precedenční analýza

Precedenční analýza je založena na **precedenční tabulce** (viz. příloha 3) a pracuje od spoda nahoru. Vstupním tokenům jsou podle jejich vstupního řetězce přiděleny **symboly precedenční tabulky**. Tokeny se postupně ukládají do hlavního zásobníku. Jakmile přijde na vstup token, který podle pravidel z precedenční tabulky ukončuje posloupnost tokenů vhodných k redukci, zahájí se redukce pravidla. Při redukci se redukovaná posloupnost tokenů předá z hlavního zásobníku do redukčního zásobníku. Pokud redukční zásobník vyhodnotí posloupnost tokenů jako korektní vstup pro redukční pravidlo, tak vytvoří symbol redukce a vloží jej na hlavní zásobník. Tyto kroky precedenční analýzy se opakují dokud na hlavním zásobníku nezbyde jediný samotný symbol redukce nebo dokud nenarazí na nekorektní posloupnost symbolů a vrátí chybu.

2.4 Generátor kódu

Výstupní kód se generuje v průběhu syntaktické analýzy, přičemž se využívá pouze tříadresných instrukcí. Generování výrazů je z důvodu rozdělení syntaktické analýzy do dvou částí, také rozděleno na hlavní generování a generování výrazů . Pro generování je třeba rozpoznávat typy proměnných, které se získají z tabulky symbolů, nebo přímo z tokenu. Pro získání unikátního názvu daného návěští slouží číslo uložené v globální proměnné. To se s každým vygenerovaným návěštím daného typu inkrementuje. Pro generování instrukcí výrazů slouží čtyři globální proměnné definované ve výstupním kódu. Jedna pro každý datový typ. Také je definována proměnná, sloužící jako počítač. Ta je využitá například při generování kódu celočíselného dělení. Součástí generování výrazů je i jejich sémantická kontrola.

2.5 Testy

V průběhu vývoje jsme projekt průběžně testovali. K tomu jsme využívali námi vytvořené testy. Testy jsme implementovali v podobě **Bash skriptů**.

3 Závěr

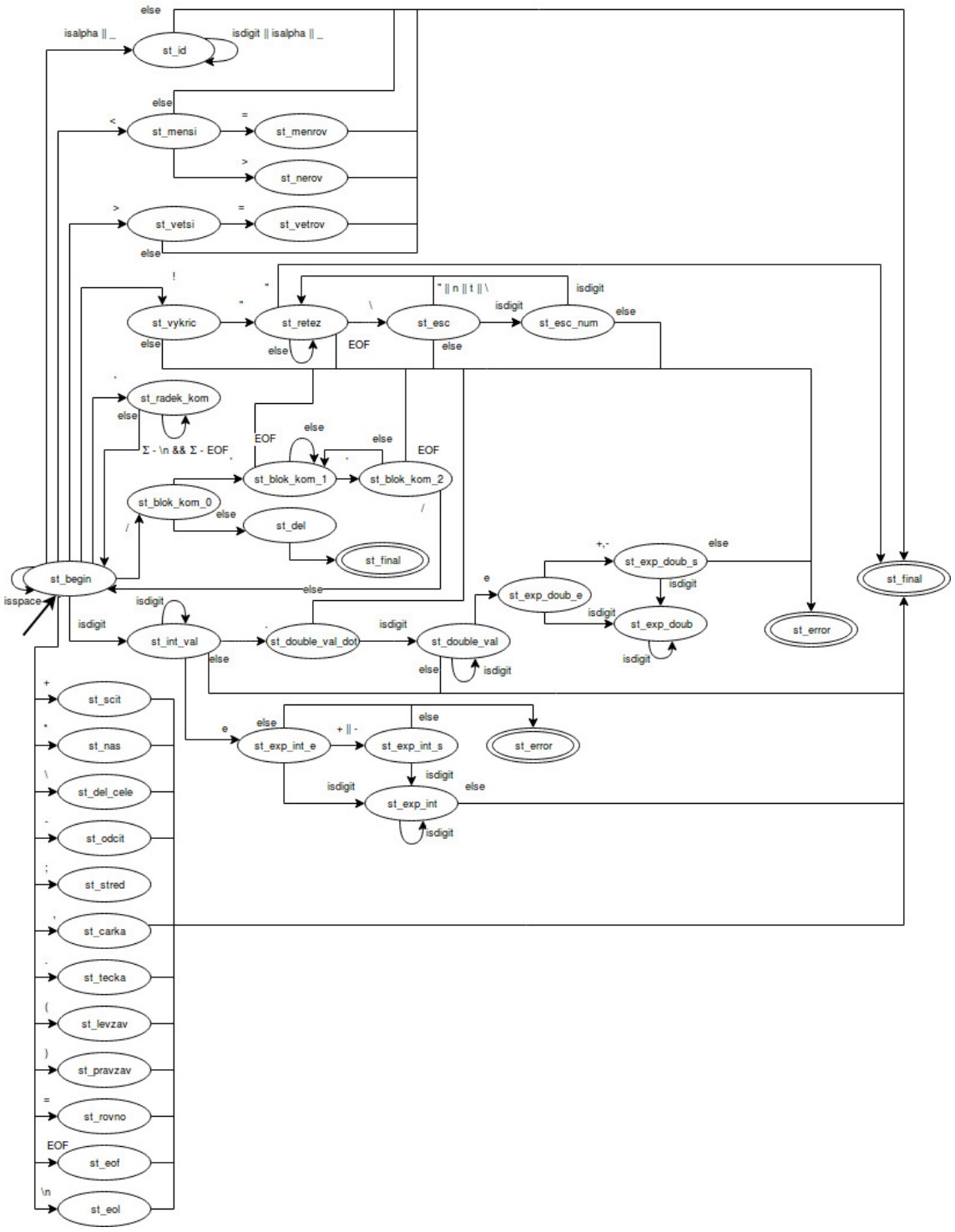
Jelikož jsme všichni členové týmu až doposud pracovali převážně samostatně, týmový projekt tahle velkých rozměrů byl pro všechny novou zkušeností. V počátcích bylo těžké hledat společná řešení a navzájem si vyhovět. V průběhu vývoje se nám povedlo najít společnou řeč a začali jsme pracovat efektivněji. Z tohoto hlediska hodnotíme projekt jako velmi přínosný a užitečný.

4 Zdroje

Jako zdroje informací jsme využívali textové materiály předmětů IFJ a IAL.

5 Přílohy

Příloha 1: Konečný automat lexikální analýzy



Příloha 2: LL-gramatika a LL-tabulka pro Rekurzivní sestup

1. <start state> ⟶ <function> <scope>
2. <scope> ⟶ 'Scope' <st-list> 'End' 'Scope'
3. <function> ⟶ <function-head> <st-list> <function-tail> <function>
4. <function> ⟶ <function-dec> <function>
5. <function> ⟶ ε
6. <function-dec> ⟶ 'Declare' 'Function' <function-id> '(' <par> ')' 'As' <type>
7. <function-head> ⟶ 'Function' <function-id> '(' <par> ')' 'As' <type> 'EOL'
8. <function-tail> ⟶ 'End' 'Function'
9. <par> ⟶ <id> 'As' <type> <next-par>
10. <par> ⟶ ε
11. <next-par> ⟶ ',' <par>
12. <next-par> ⟶ ε
13. <type> ⟶ 'Integer'
14. <type> ⟶ 'Double'
15. <type> ⟶ 'String'
16. <st-list> ⟶ <stat> <st-list>
17. <st-list> ⟶ ε
18. <stat> ⟶ 'Dim' <id> 'As' <type> <eval>
19. <eval> ⟶ '=' <expr>
20. <eval> ⟶ ε
21. <stat> ⟶ <id> '=' <assign>
22. <assign> ⟶ <expr>
23. <assign> ⟶ <function-id> '(' <call-par> ')'
24. <call-par> ⟶ <id> <call-next-par>
25. <call-par> ⟶ ε
26. <call-next-par> ⟶ ',' <call-par>
27. <call-next-par> ⟶ ε
28. <stat> ⟶ 'Input' <id>
29. <stat> ⟶ 'Print' <expr> ; <pr-expr>
30. <pr-expr> ⟶ <expr> ; <pr-expr>
31. <pr-expr> ⟶ ε
32. <stat> ⟶ 'If' <expr> 'Then' 'EOL' <st-list> 'Else' 'EOL' <st-list> 'End' 'If'
33. <stat> ⟶ 'Do' 'While' <expr> 'EOL' <st-list> 'Loop'
34. <stat> ⟶ 'Return' <expr>

	Scope	*End*	*Dec.*	*Func.*	(	)	*ID*	*As*	*EOL*	,	*Int.*	*Doub.*	*String*	*Dim*	=	*Input*	*Print*	;	*If*	*Then*	*Else*	*Do*	*While*	*Loop*	*Ret.*	$	*expr.*
< *startstate* >	1		1	1																							
< *scope* >	2																										
< *function* >			4	3																							
< *function* – *dec* >			6																								
< *function* – *head* >				7																							
< *function* – *tail* >		8																									
< *par* >						10	9																				
< *next* – *par* >						12				11																	
< *type* >											13	14	15														
< *st* – *list* >		17					16							16		16	16		16			16			16		
< *stat* >							21							18		28	29		32			33			34		
< *eval* >		20					20							20	19	20	20		20			20			20		
< *assign* >																											22
< *call* – *par* >						25	24																				
< *call* – *next* – *par* >						27				26																	
< *pr* – *expr* >		27					27							27		27	27		27			27			27		30

Příloha 3: Precedenční tabulka

	*	/	\	+	-	=	<>	<	<=	>	>=	(	)	i	int	db	sr	$	R
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>	<
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>	<
\	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>	<
+	<	<	<	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>	<
-	<	<	<	>	>	>	>	>	>	>	>	<	>	<	<	<	<	>	<
=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	<	<	>	<
<>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	<	<	>	<
<	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	<	<	>	<
<=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	<	<	>	<
>	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	<	<	<	<
>=	<	<	<	<	<	>	>	>	>	>	>	<	>	<	<	<	<	<	<
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<		<
)	>	>	>	>	>	>	>	>	>	>	>		>						
i	>	>	>	>	>	>	>	>	>	>	>		>		<	<	<	>	
int	>	>	>	>	>	>	>	>	>	>	>		>					>	
db	>	>	>	>	>	>	>	>	>	>	>		>					>	
sr	>	>	>	>	>	>	>	>	>	>	>		>					>	
$	<	<	<	<	<	<	<	<	<	<	<	<		<	<	<	<		<
R	>	>	>	>	>	>	>	>	>	>	>		>		<	<	<	>	