

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentace k projektu do předmětů IFJ
Implementace interpretu jazyka IFJ18

Tým 61, varianta I

Vedoucí:	David Bulawa	<code>xbulaw01</code>	25
Členové:	Jakub Dolejší	<code>xdolej09</code>	25
	František Policar	<code>xpolic04</code>	25
	Tomáš Svěrák	<code>xsvera04</code>	25

Obsah

1	Úvod	2
2	Práce v týmu	2
2.1	Rozdělení práce na jednotlivých částech	2
2.2	Průběh vývoje	2
3	Implementace interpretu jazyka IFJ18	3
3.1	Lexikální analýza	3
3.2	Syntaktická analýza	3
3.3	Precedenční analýza	3
3.4	Generování	3
3.5	Použité algoritmy a datové struktury	4
3.5.1	Buffer	4
3.5.2	BVS	4
3.5.3	Dynamický string	4
3.5.4	Zásobník	4
3.5.5	Postfix	4
3.6	Testování	5
4	Závěr	5
5	Přílohy	6
5.A	Diagram konečného automatu lexikální analýzy	6
5.B	LL-gramatika	7
5.C	Precedenční tabulka	9

1 Úvod

Dokumentace popisuje implementaci překladače imperativního jazyka IFJ18, který je podmožinou jazyka Ruby2.0. Níže najdete popis průběhu vývoje, použité algoritmy a datové struktury.

2 Práce v týmu

2.1 Rozdělení práce na jednotlivých částech projektu

- David Bulawa – Rekurzivní sestup, generování, návrh LL-gramatiky
- Jakub Dolejší – Lexikální analyzátor, tabulka symbolů, generování, precedenční analýza
- František Policar – Dokumentace, prezentace, konečný automat, rekurzivní sestup
- Tomáš Svěrák – Lexikální analyzátor, generování, precedenční analýza, návrh zásobníků

2.2 Průběh vývoje

Na projektu se podílel čtyřčlenný tým, bylo tedy třeba dobře rozdělit a rozvrhnout práci. Po domluvě jsme využili verzovací systém **Subversion** na serveru github.com. Problémy a postup řešení projektu jsme konzultovali alespoň dvakrát týdně. Na začátku měl každý přidělenou část projektu, kterou měl zpracovat (vyjma složitějších částí, které jsme dělali společně), a následně jsme jednotlivé moduly začali propojovat mezi sebou.

3 Implementace interpretu jazyka IFJ18

3.1 Lexikální analýza

Lexikální analýza je činnost, kterou provádí tzv. lexikální analyzátor (SCANNER). Lexikální analyzátor rozdělí vstupní posloupnost znaků na lexémy - např. identifikátory, operátory. Tyto lexémy jsou reprezentovány ve formě tokenů, ty jsou poskytnuty ke zpracování syntaktickému analyzátoru (PARSERU). Lexikální analyzátor jsme implementovali pomocí deterministického konečného automatu (dále DKA). Jediná výjimka je v případě přechodu do blokového komentáře, kde determinismus nebylo možné zachovat (implementačně ošetřeno boolovskou proměnnou, která značí nový řádek). Následně DKA zjistí typ tokenu, do kterého uloží jeho typ, a v případě identifikátoru/čísla i jeho hodnotu(atribut). Pokud DKA došel do koncového stavu, odesílá token syntaktické analýze a vrací se zpět do počátečního stavu. V jiném případě vrací lexikální chybu.

3.2 Syntaktická analýza

Syntaktický analyzátor (PARSER) je jádro celého překladače. Na základě pravidel sestavených dle LL – gramatiky (Příloha 4.A a 4.B) jsme vytvořili funkce pro rekurzivní sestup, který je kombinovaný s precedenční analýzou pro výpočet výrazů. SA žádá LA o tokeny, které následně zpracovává. Rekursivní sestup je implementován dvouprůchodově s využitím bufferu. První průchod se skládá z částečné kontroly syntaxe a naplnění tabulky symbolů. Druhý průchod kontroluje zbylou syntaxi, sémantiku a generuje cílový kód.

3.3 Precedenční analýza

Precedenční analýza je separátní část SA, která se stará o zpracování výrazů. Nejprve jsme si vytvořili seznam redukčních pravidel a precedenční tabulku dle priority operátorů. Tabulka byla implementována pomocí dvourozměrného pole ENUMů, kde sloupce vyjadřovaly aktuální token na vstupu a řádky symbol na zásobníku. Abychom v tabulce mohli řádně indexovat, převedli jsme vstupní token automaticky na stejný datový typ jako symboly na zásobníku. Dále jsme si vytvořili funkce pro ověření pravidel a následnou redukci v případě správně nalezeného pravidla. Hlavní tělo precedenční analýzy jsme implementovali dle algoritmu z přednášky. Ten spočívá v nalezení indexu v precedenční tabulce a jeho vyhodnocení dle předepsaných pravidel. Zde jsme provedli změny ohledně práce s tokeny; namísto žádání LA o další token si pouze posuneme ukazatel na další prvek v bufferu, do kterého si token ukládáme.

Vzhledem k tomu, že jazyk IFJ18 je dynamicky typovaný, tak jsme sémantické kontroly výrazů nemohli provádět zde, ale až na úrovni generování.

3.4 Generování

Po úspěšném dokončení lexikální analýzy a prvním průchodu syntaktické analýzy se začíná generovat cílový kód. Na standardní výstup generujeme jednotlivé instrukce. Opět zde probíhá předávání řízení mezi rekurzivním sestupem a precedenční analýzou. Provádí se zde typové kontroly a konverze při výpočtu výrazů. Instrukce se generují přímo na standardní výstup.

3.5 Použité algoritmy a datové struktury

3.5.1 Buffer

Ačkoliv jsme měli využívat LL(1) gramatiku, tak v několika případech bylo nutno se podívat o více než jeden token dopředu. Z tohoto důvodu jsme implementovali buffer, do kterého jsme tokeny nahrávali. Pokud se tedy SA potřebovala podívat o více tokenů dopředu a následně je vrátit, tak jednoduše posunula ukazatel na aktuální prvek v bufferu dozadu.

3.5.2 BVS

Vzhledem k tomu, že naše skupina má variantu 1, tak jsme tabulku symbolů implementovali pomocí binárního stromu. Každá položka (uzel) BVS v sobě obsahuje vyjma ukazatele na levý a pravý podstrom a ukazatele na pod tabulku, informace, zda se jedná o proměnnou či funkci, či byla definovaná a počet parametrů v případě funkce.

3.5.3 Dynamický string

Vzhledem k tomu, že v jazyce C neexistuje typ string, tak jsme byli nuceni si vytvořit vlastní datový typ pro řetězec, abychom mohli provádět kupříkladu konkatenci. V tomto případě jsme se inspirovali soubory `str.c` a `str.h` z ukázkového interpretu dostupného na stránkách předmětu.

3.5.4 Zásobník

Zásobník (STACK) je dynamická datová struktura používaná pro dočasné ukládání dat. Má uplatnění pro precedenční syntaktickou analýzu. Nejvíce se využívá při převodu infixového zápisu na postfixový.

3.5.5 Postfix

Pro vyčíslení výrazů a vygenerování cílového kódu u precedenční syntaktické analýzy jsme použili postfixový zápis. Ten značně zjednodušil práci s mezivýsledky, kdy jsme si nemuseli vytvářet pomocné proměnné.

4 Závěr

Už ze začátku bylo jasné, že se bude jednat o zatím největší projekt, co jsme zde měli, proto jsme ho nechtěli podcenit. V průběhu práce na projektu jsme se potýkali s řadou problémů, které povětšinou vyřešilo diskuzní fórum, stránky z přednášek, či vzájemné diskuze.

5.A Diagram konečného automatu lexikální analýzy

5.A Diagram konečného automatu lexikální analýzy



5.B LL-gramatika

1. PROG -> SEC END_ROW PROG
2. PROG -> eol PROG
3. PROG -> ''
4. PROG -> eof
5. PROG -> bcomment PROG
6. END_ROW -> eol
7. END_ROW -> eof
8. SEC -> identifier eq FCE_EXPR
9. SEC -> while EXPR do eol PROG end
10. SEC -> if EXPR then eol PROG else eol PROG end
11. SEC -> def identifier lbracket PARAMS rbracket eol PROG end
12. PARAMS -> ''
13. PARAMS -> identifier PARAM_LIST
14. PARAM_LIST -> ''
15. PARAM_LIST -> comma identifier PARAM_LIST
16. CALL_PARAMS -> ''
17. CALL_PARAMS -> ITEM CALL_PARAM_LIST
18. CALL_PARAM_LIST -> ''
19. CALL_PARAM_LIST -> comma ITEM CALL_PARAM_LIST
20. FCE_EXPR -> EXPR
21. FCE_EXPR -> FCE
22. BR_OR_NOT -> lbracket CALL_PARAMS rbracket
23. BR_OR_NOT -> CALL_PARAMS
24. FCE -> identifier BR_OR_NOT
25. ITEM -> integer
26. ITEM -> float
27. ITEM -> string
28. ITEM -> nil
29. ITEM -> identifier
30. EXPR -> ''

NONTERMINAL	EOL	EOF	BCOMMENT	IDENTIFIER	EQ	WHILE	DO	END	IF	THEN	ELSE	DEF	LBRACKET	RBRACKET	COMMA	INTEGER	FLOAT	STRING	NIL	\$
PROG	2	4	5	1		1		3	1		3	1								3
END_ROW	6	7																		
SEC				8		9			10			11								
PARAMS				13										12						
PARAM_LIST														14	15					
CALL_PARAMS	16	16		17										16		17	17	17	17	
CALL_PARAM_LIST	18	18												18	19					
FCE_EXPR	20	20		21																
BR_OR_NOT	23	23		23									22			23	23	23	23	
FCE				24																
ITEM				29												25	26	27	28	
EXPR	30	30					30			30										

5.C Precedenční tabulka

	*	/	+	-	<	<=	>	>=	==	!=	()	i	\$
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>
+	<	<	>	>	>	>	>	>	>	>	<	>	<	>
-	<	<	>	>	>	>	>	>	>	>	<	>	<	>
<	<	<	<	<							<	>	<	>
<=	<	<	<	<							<	>	<	>
>	<	<	<	<							<	>	<	>
>=	<	<	<	<							<	>	<	>
==	<	<	<	<							<	>	<	>
!=	<	<	<	<							<	>	<	>
(<	<	<	<	<	<	<	<	<	<	<	=	<	
)	>	>	>	>	>	>	>	>	>	>		>		>
i	>	>	>	>	>	>	>	>	>	>	=	>		>
\$	<	<	<	<	<	<	<	<	<	<	<		<	

1 $E = E * E$

2 $E = E / E$

3 $E = E + E$

4 $E = E - E$

5 $E = E < E$

6 $E = E <= E$

7 $E = E > E$

8 $E = E >= E$

9 $E = E == E$

10 $E = E != E$

11 $E = (E)$

12 $E = i$