

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



Dokumentace ke společnému projektu
pro předměty IFJ a IAL

Implementace interpretu imperativního jazyka IFJ16.

Tým 086, varianta a/3/II

ZS 2016

Řešitelé:

Richtarik Lukáš	xricht25	23%
Čechák Jiří	xcecha04	20%
Mlýnek Přemysl	xmlyne04	16%
Mynarčík Petr	xmynar05	18%
Molitoris Miloš	xmolit00	23%

Rozšíření: UNARY, BASE, FUNEXP, SIMPLE, BOOLOP

7. 12. 2016

Obsah

1. Úvod	2
2. Implementace modulů a algoritmů	2
2.1 Lexikální analýza	2
2.2 Syntaktická a sémantická analýza	2
2.3 Interpret	2
2.4 Implementace tabulky symbolů	3
2.5 Implementace řazení	3
2.6 Implementace vyhledávání podřetězce v řetězci	3
3. Práce na projektu	3
4. Testování	3
5. Rozdělení práce	3
6. Implementovaná rozšíření	4
7. Závěr	4
8. Precedenční tabulka	5
9. Diagram konečného automatu	6
10. LL-gramatika	7

1. Úvod

Tato dokumentace popisuje implementaci interpretu imperativního jazyka IFJ16, který je zjednodušenou podmnožinou jazyka Java SE 8. Implementován je v jazyce C.

Interpret načte zdrojový soubor v jazyce IFJ16 a následně kontroluje, zdali v něm nejsou lexikální, syntaktické nebo sémantické chyby a během interpretace se kontroluje, zdali nenastanou běhové chyby. V případě objevení chyby nebo projevení nějaké interní chyby interpretu, interpret vypíše na standardní chybový výstup chybovou hlášku a ukončí se s návratovou hodnotou dané chyby.

Jsou zde popsány implementace modulů, použité algoritmy a popis práce na projektu. Dokumentace také obsahuje diagram konečného automatu použitého v lexikální analýze a LL-gramatiku s precedenční tabulkou, které byly použity v syntaktické analýze.

2. Implementace modulů a algoritmů

2.1 Lexikální analýza

Lexikální analýza je implementována pomocí konečného automatu. Začíná se vždy v počátečním stavu a v závislosti na načteném znaku ze zdrojového souboru se konečný automat posune do dalšího stavu, kde se již kontroluje na základě načtení následujících znaků, zdali je konkrétní lexém napsán správně.

Pokud je načten znak, který do daného lexému nepatří nebo se jedná o neočekávaný znak, jde o chybu a program je ukončen s návratovou hodnotou 1. U komentářů se kontroluje, zdali jsou korektně zapsány, u víceřádkových i ukončeny.

Po přečtení a zpracování celého lexému, teď již tokenu, je tento token předán syntaktickému analyzátoru k syntaktické a sémantické analýze.

2.2 Syntaktická a sémantická analýza

Syntaktický analyzátor volá funkci `get_token`, která se nachází v modulu `scanner` s implementací lexikální analýzy, a tato funkce předá syntaktickému analyzátoru token, který získá ze zdrojového souboru.

Nejdříve jsou získány globální proměnné a funkce. Jejich identifikátory a informace o nich, jako jsou např. parametry u funkce, jsou uloženy do globální tabulky symbolů. Lokální proměnné se ukládají pak do lokálních tabulek symbolů konkrétních funkcí. Správná syntaxe zdrojového programu se ověřuje rekurzivním postupem za použití LL-gramatiky. Zpracování výrazů je prováděno metodou zdola nahoru podle precedenční tabulky.

Během sémantické analýzy dochází dle potřeby k přetypování proměnných a ke kontrole typů proměnných, návratových hodnot z funkcí apod. V případě bezchybné analýzy vzniká tříadresný kód, který se ukládá do instrukčního listu. Ten je nejprve nelineární, kdy může v sobě obsahovat i další listy a následně je v modulu generátor převeden na lineární.

2.3 Interpret

Pokud nenastane nějaká chyba, tak interpret interpretuje zdrojový program napsaný v jazyce IFJ16. Interpret prochází instrukční list, který obsahuje tříadresné kódy a ty poté postupně interpretuje. V případě získání tříadresného kódu, který představuje skokovou instrukci, pokračuje v interpretaci od odpovídající instrukce. Interpret také dle potřeby volá vestavěné funkce třídy `ifj16`.

2.4 Implementace tabulky symbolů

Tabulka symbolů je implementována jako tabulka s rozptýlenými položkami s explicitním zřetězením synonym.

K vyhledávání v tabulce symbolů a k ukládání položek do tabulky symbolů slouží klíč, kterým je identifikátor proměnné nebo funkce. V globální tabulce symbolů jsou uloženy funkce s jejich parametry a globální proměnné. Do lokálních tabulek symbolů se ukládají lokální proměnné.

2.5 Implementace řazení

K implementaci funkce na řazení řetězce byl použit dle zadání algoritmus Shell sort, který pracuje na principu vkládání.

Nejdříve se získá délka kroku, což je polovina délky řetězce. Následně se v cyklu řadí znaky řetězce, které jsou od sebe v řetězci vzdáleny o velikost kroku a s každou další iterací se délka kroku zmenší o polovinu. Jakmile má krok velikost jedna, jsou řazeny prvky vedle sebe.

2.6 Implementace vyhledávání podřetězce v řetězci

K vyhledávání podřetězce v řetězci byl použit Knuth-Morris-Prattův algoritmus, který pracuje na základě konečného automatu. Pokud je hledaný podřetězec prázdný je funkcí vrácena 0. Jinak je na základě řetězce, ve kterém vyhledáváme, vytvořeno pole, které určuje znak, kam se vrátíme v případě neúspěšného porovnávání.

Poté konečný automat postupuje po jednotlivých znacích řetězce, při shodě se posune na další znak, jinak se vrátí na znak určený dříve vytvořeným polem. Porovnává se, dokud není podřetězec nalezen, pak funkce vrátí pozici podřetězce, nebo dokud se algoritmus nedostane na konec řetězce. V takovém případě funkce vrátí -1.

3. Práce na projektu

Na týmovém setkání se rozdělily moduly, na kterých se pracovalo a po jejich dokočení a otestování se začalo pracovat na dalších. V průběhu práce se náš tým několikrát sešel k prodiskutování vývoje a změn v implementaci.

Pokud nebyla možnost se sejít, případně bylo potřeba kontaktovat jiné členy týmu, využíval se ke komunikaci společný chat a soukromá skupina na sociální síti Facebook nebo email. K zálohování souborů a sledování jednotlivých verzí jsme využívali GitHub.

4. Testování

Každý modul byl testován nejdříve zvlášť během jeho vývoje a také hlavně po jeho dokončení. V případě nalezení chyb při testování modulu, případně více modulů dohromady, byly chyby odstraněny a konkrétní modul byl znovu otestován.

Po dokončení všech modulů a sestavení projektu začalo testování interpretu jako celku za použití námi vytvořených zdrojových kódů v jazyce IFJ16.

5. Rozdělení práce

Lukáš Richtarik (xricht25) – vedoucí týmu: lexikální analyzátor, syntaktický analyzátor, sémantická analýza, generování tříadresného kódu

Miloš Molitoris (xmolit00): lexikální analyzátor, syntaktický analyzátor, sémantická analýza, instrukční list, generování tříadresného kódu

Jiří Čechák (xcecha04): algoritmy pro IAL, lexikální analyzátor, vestavěné funkce, správa paměti, tvorba závěrečných testů, dokumentace

Přemysl Mlýnek (xmlyne04): vestavěné funkce, tvorba závěrečných testů, výpomoc na dalších částech

Petr Mynarčík (xmynar05): interpret, instrukční list a generátor

6. Implementovaná rozšíření

UNARY – podpora unárního mínus a prefixové i postfixové inkrementace a dekrementace.

Řešení rozšíření UNARY:

Unární mínus

Je získáno jako token `token_sub`. Ve výrazu se vyhodnocuje, pokud je před ním jiný operátor nebo nic. Pokud je před ním nonterminal (`id`- proměnná nebo konstanta) nebo `'`' jedná se o mínus. Označí se jako `operator_una`, následuje kontrola sémantiky (může být jen před typem `int` nebo `double`) a generuje se instrukce (`T_una`) a daná proměnná je vynásobena číslem `-1`.

Prefixová a postfixová inkrementace a dekrementace

Jsou získány jako tokeny `token_inc` a `token_dec`. Mohou být provedeny pouze nad lokálními a globálními proměnnými typu `int` a při použití na konstantách je hlášena chyba 6. Pokud se nachází na řádce samy, jsou vyhodnoceny ihned.

Prefixová inkrementace a dekrementace – ve výrazu se vyhodnocují, pokud před nimi není nonterminál nebo `'`'. Označí se jako `operator_Einc/operator_Edec`, následuje kontrola sémantiky a poté se vygeneruje instrukce `T_INC/T_DEC` a u dané proměnné je přičtena/odečtena 1.

Postfixová inkrementace a dekrementace – ve výrazu se vyhodnocují, pokud před nimi je nonterminál nebo `'`'. Označí se jako `operator_Einc/operator_Edec`, následuje kontrola sémantiky a na pomocný zásobník je uložena daná proměnná. Instrukce se generuje po vyhodnocení výrazu.

BASE – podpora zápisu čísel i ve dvojkové, osmičkové a šestnáctkové soustavě.

FUNEXP – podpora volání funkcí uvnitř výrazu a výrazů v parametrech funkcí.

SIMPLE – podpora podmíněných příkazů `if` i bez části `else`. V podmíněných příkazech a cyklech lze místo složeného příkazu v složených závorkách použít i jedno řádkový příkaz.

BOOLOP – podpora typu `boolean`, jeho definice a výpis, booleovských výrazů, hodnot a operátorů.

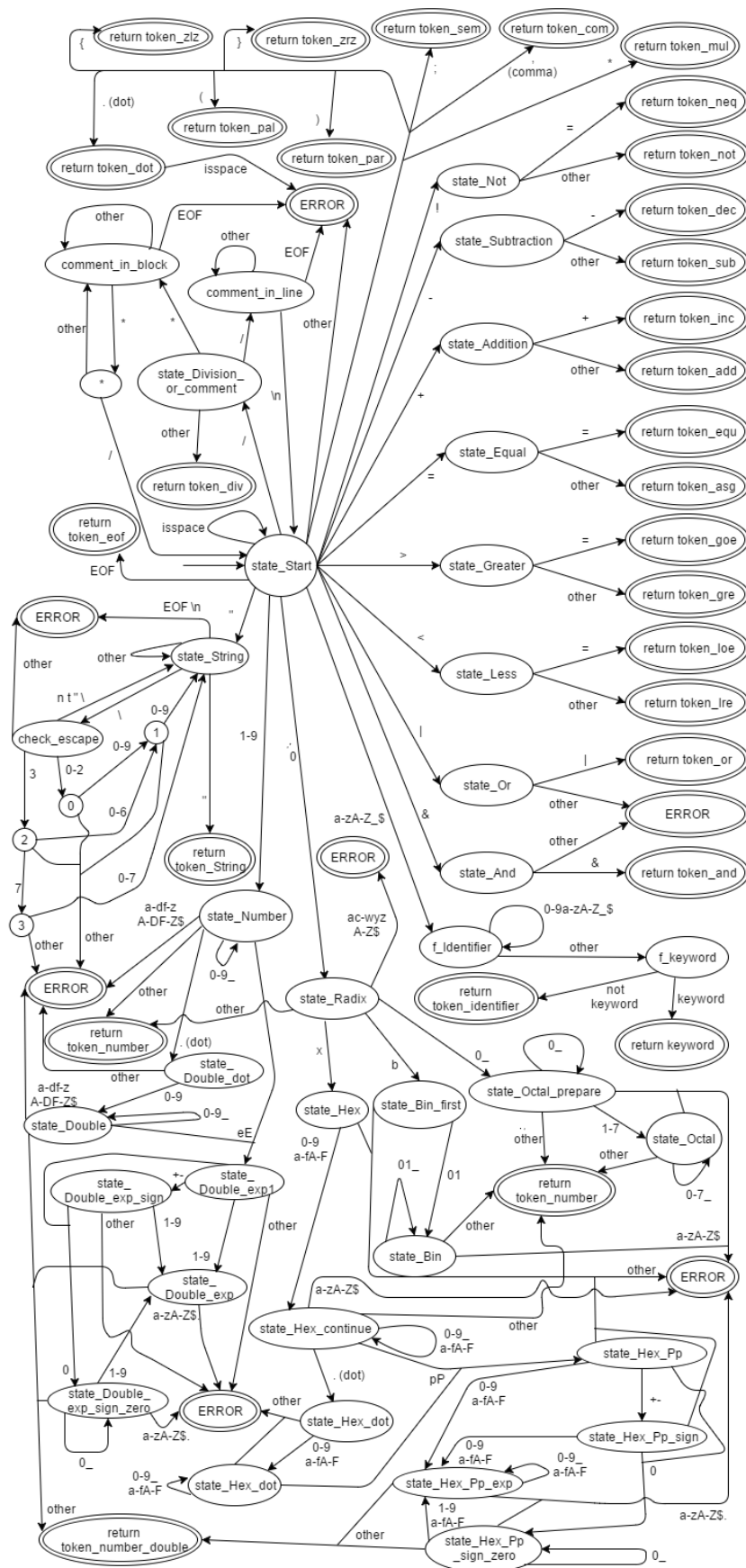
7. Závěr

Práci na tomto projektu jsme si ověřili své znalosti v programování v jazyce C a získali i mnoho nových. Dále jsme mohli aplikovat své nově získané teoretické znalosti z předmětu IFJ a teoretické i částečně praktické znalosti (díky projektům v průběhu semestru) z předmětu IAL. Také jsme všichni určitě obohaceni o velkou zkušenost práce v malém týmu a komunikace s ostatními členy týmu.

8. Precedenční tabulka

	unar-	+	-	*	/	E++	E--	==	!=	>	<	>=	<=		&&	!	()	,	ID	Fn	\$	++E	--E
unar-	F	>	>	>	>	<	<	>	>	>	>	>	>	>	>	>	<	>	F	<	<	>	<	<
+	<	>	>	<	<	<	<	>	>	>	>	>	>	>	<	<	<	>	>	<	<	>	<	<
-	<	>	>	<	<	<	<	>	>	>	>	>	>	>	<	<	<	>	>	<	<	>	<	<
*	<	>	>	>	>	<	<	>	>	>	>	>	>	>	>	<	<	>	>	<	<	>	<	<
/	<	>	>	>	>	<	<	>	>	>	>	>	>	>	>	<	<	>	>	<	<	>	<	<
E++	F	>	>	>	>	F	F	>	>	>	>	>	>	>	>	F	F	>	>	F	F	>	F	F
E--	F	>	>	>	>	F	F	>	>	>	>	>	>	>	>	F	F	>	>	F	F	>	F	F
==	<	<	<	<	<	<	<	>	>	>	>	>	>	<	<	<	<	>	>	<	<	>	<	<
!=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	<	<	<	>	>	<	<	>	<	<
>	<	<	<	<	<	<	<	>	>	>	>	>	>	<	<	<	<	>	>	<	<	>	<	<
<	<	<	<	<	<	<	<	>	>	>	>	>	>	<	<	<	<	>	>	<	<	>	<	<
>=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	<	<	<	>	>	<	<	>	<	<
<=	<	<	<	<	<	<	<	>	>	>	>	>	>	<	<	<	<	>	>	<	<	>	<	<
	<	>	>	<	<	<	<	>	>	>	>	>	>	>	<	<	<	>	>	<	<	>	<	<
&&	<	>	>	>	>	<	<	>	>	>	>	>	>	>	>	<	<	>	>	<	<	>	<	<
!	<	>	>	>	>	F	F	>	>	>	>	>	>	>	>	F	<	>	>	<	<	>	F	F
(<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	=	<	<	>	<	<
)	F	>	>	>	>	F	F	>	>	>	>	>	>	>	>	>	F	>	>	F	F	>	F	F
,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	=	<	<	F	<	<
ID	F	>	>	>	>	>	>	>	>	>	>	>	>	>	>	>	F	>	>	F	F	>	F	F
Fn	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	=	F	>	F	F	F	F	F
\$	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	F	F	<	<	F	<	<
++E	F	>	>	>	>	F	F	>	>	>	>	>	>	>	>	F	F	>	>	<	<	>	F	F
--E	F	>	>	>	>	F	F	>	>	>	>	>	>	>	>	F	F	>	>	<	<	>	F	F

9. Diagram konečného automatu



10. LL-gramatika

		PREDICT
1	CLASS \rightarrow class identifier { VNUTRO_TRIEDY } CLASS end_program	class
2	CLASS $\rightarrow \epsilon$	end_program
3	VNUTRO_TRIEDY \rightarrow STATIC type identifier TRIEDNY_CLEN	static type
4	VNUTRO_TRIEDY $\rightarrow \epsilon$	}
5	STATIC \rightarrow static	static
7	PREMENNA_DEFINITION \rightarrow type identifier	type
8	PREMENNA_INIT \rightarrow operator_assignment EXRESSION	operator_assignment
9	PREMENNA_INIT $\rightarrow \epsilon$;
10	TRIEDNY_CLEN \rightarrow (FUNCTION_PARAMS){VNUTRO_FUNKCIE} VNUTRO_TRIEDY	(
11	TRIEDNY_CLEN \rightarrow PREMENNA_INIT; VNUTRO_TRIEDY	operator_assignment
12	FUNCTION_PARAMETERS \rightarrow PREMENNA_INIT; VNUTRO_TRIEDY	type
13	FUNCTION_PARAMETERS $\rightarrow \epsilon$)
14	PRVY_PARAMETER \rightarrow PREMENNA_DEFINITION	type
15	DALSI_PARAMETER \rightarrow , PREMENNA_DEFINITION DALSI_PARAMETER	,
16	DALSI_PARAMETER $\rightarrow \epsilon$)
17	VNUTRO_FUNKCIE $\rightarrow \epsilon$	}
18	VNUTRO_FUNKCIE \rightarrow STATEMENT VNUTRO_FUNKCIE	if while identifier return type increment decrement
19	STATEMENT \rightarrow if (EXPRESSION) STATEMENT_BODY ELSE	if
20	STATEMENT_BODY \rightarrow {VNUTRO_FUNKCIE}	{
21	STATEMENT_BODY \rightarrow STATEMENT	if while identifier return type increment decrement
22	ELSE $\rightarrow \epsilon$	if while identifier type return } increment decrement
23	ELSE \rightarrow else STATEMENT_BODY	else
24	STATEMENT \rightarrow while (EXPRESSION) STATEMENT_BODY	while
25	STATEMENT \rightarrow identifier DOT_IDENTIFICATOR ROZLISENIE_IDENTIFIKATORA	identifier
26	DOT_IDENTIFICATOR $\rightarrow \epsilon$	operator_assignment (
27	DOT_IDENTIFICATOR \rightarrow .identifier	.
28	ROZLISENIE_IDENTIFIKATORA \rightarrow operator_assignment EXPRESSION;	operator_assignment
29	ROZLISENIE_IDENTIFIKATORA \rightarrow (FUNCTION_ARGUMENTS);	(
30	STATEMENT \rightarrow PREMENNA_DEFINITION PREMENNA_INIT;	type
31	FUNCTIONS_ARGUMENTS $\rightarrow \epsilon$)
32	FUNCTIONS_ARGUMENTS \rightarrow EXPRESSION SECOND_ARGUMENT	! (identifier - value
33	SECOND_ARGUMENT \rightarrow , EXPRESSION SECOND_ARGUMENT	,
34	SECOND_ARGUMENT $\rightarrow \epsilon$)
35	STATEMENT \rightarrow return EXPRESSION;	return
36	ROZLISENIE_IDENTIFIKATORA \rightarrow increment;	increment
37	ROZLISENIE_IDENTIFIKATORA \rightarrow decrement;	decrement
38	STATEMENT \rightarrow increment identifier;	increment
39	STATEMENT \rightarrow decrement identifier;	decrement