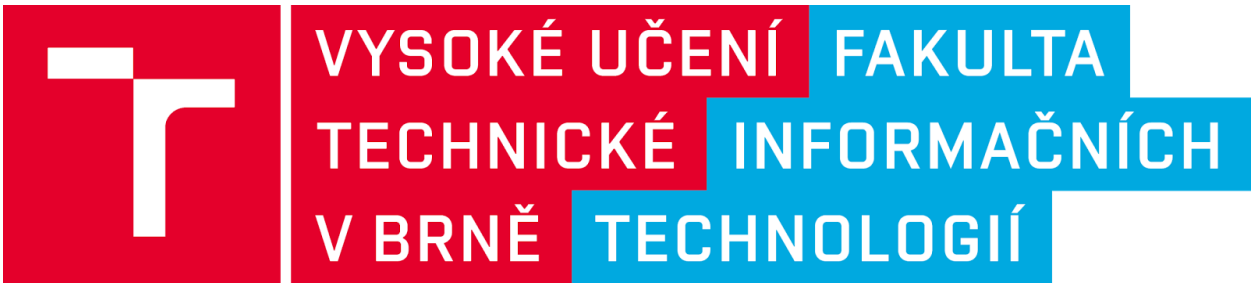
Vysoké učení technické v Brně Fakulta informačních technologií



Překladač jazyka IFJ17

Dokumentace k projektu do předmětu IFJ a IAL
Tým 122, varianta II

FUNEXP

Martin Bobčík, xbobci00 - 60%
Martin Lacika, xlacik02 - 0%
Vojtěch Meluzín, xmeluz04 - 40%
David Kala, xkalad00 - 0%

3. Prosince 2017

Obsah

Úvod

Tato dokumentace popisuje návrh a implementaci programu, překládající jazyk IFJ17 do IFJ17code. Jazyk IFJ17 je zjednodušený BASIC a IFJ17code je assemler s řadou funkcí navíc.
Dokumentace také nastíní jaké problémy jsme řešili a práci v týmu.

Implementace částí

Lexikální analýza

Lexikální analýza je založena na deterministickém konečném automatu. Je to jediná část, která přímo pracuje se vstupním textem. Jednotlivé lexémy konvertuje na tokeny a zbavuje vstup bílých znaků a komentářů.Vstupní abeceda automatu jsou znaky načítané ze standardního vstupu programu. Konečné stavy automatu určují výsledný typ tokenu.
Token je struktura pro uložení typu a dat lexému. Typ lexému je například identifikátor, určité klíčové slovo, konec řádku nebo datový typ integer. Jako data tokenu lze považovat například hodnota proměnné, nebo jméno identifikátoru. Ne všechny typy tokenů využívají datové části tokenu.
Problém u lexikální analýzy nastal v případě vyhodnocování klíčových slov. Pokud by se tato část řešila čistě konečným automatem, měl by tento automat velmi mnoho stavů. Proto, po načtení jakéhokoliv identifikátoru se jeho hodnota předá funkci *idOrKey(),* která vyhodnotí, zda se skutečně jedná o identifikátor, nebo o jedno z možných klíčových slov.

Syntaktická analýza

Syntaktická analýza je jádrem překladače. Pomocí funkce *getToken() postupně z*iskává tokeny z modulu lexikální analýzy a rekurzivním sestupem zjišťuje, zda jsou tokeny ve správném pořadí. Rekurzivní sestup se řídí námi sestavenou LL-gramatikou. Sestavení správné gramatiky nebylo vůbec jednoduché, a musela být mnohokrát přepisována. Po dosažení konsenzu, že gramatika je správně, jsme začali s implementací rekurzivního sestupu, což už bylo jednoduché.
Problém v gramatice nastal při řešení přiřazování výsledku výrazu, nebo výsledku funkce do proměnné, jelikož obě pravidla začínají stejně. Rozhodli jsme se proto zvolit rozšíření FUNEXP, kde se funkce řeší pouze jako výrazy.

Sémantická analýza

Sémantická analýza provádí řadu sémantických kontrol. Ke své práci využívá tabulku symbolů a pracuje ve stejné době jako syntaktická analýza.

Pokud pracujeme s proměnnými nebo funkcemi, kontroluje jestli již byly deklarované a jestli není porušena práce s datovými typy.
Na konci programu projde postupně veškeré symboly z tabulky symbolů a zkontroluje jestli jsou všechny definované.

Tabulka symbolů

Tabulka symbolů slouží pro uchování symbolů, které nám přicházejí od syntaktické analýzy, pro potřeby sémantické analýzy. Symboly jsou dvojího typu funkce a proměnné. Oba dva typy mají svůj vlastní datový typ, který u funkce představuje datový typ návratové hodnoty a u proměnné datový typ proměnné. Dále je ještě společný název. Potom u funkce můžeme očekávat její vlastní tabulku symbolu a parametry. U obou typů si uchováváme zda-li jsou definované.
V našem projektu jsme měli za úkol implementovat tabulku symbolů pomocí pomocí hash tabulky s rozptýlenými položkami. Základem pro tuto tabulku posloužil kód z domácího úkolu do IAL a článek s popisem hash tabulky (viz. Použitá literatura odkaz č. 1).
Nad připravenými funkcemi (př. Search, exists) jsme vytvořily nádstavbu pro jednodužší obsluhu tabulky. Vznikly nám tak funkce s prefixem “ht_”. Tyto funkce pak využíváme v sémantické analíze. Všechny funkce pro práci s hash tabulkou a nádstavbu pro jednodušší práci jsou v modulu *symtable*. Tento modul obsahuje i funkce pro práci se seznamem, které se pojí s ukládáním parametrů pro funkce v tabulce symbolů.

Generátor výstupního kódu

Na vstupu do generátoru výstupního kódu je páska tří adresního kódu. Instrukce na pásku jsou postupně přidávány v průběhu zpracování programu. Úlohou generátoru výstupního kódu je postupně přeložit jednotlivé instrukce na výstupní IFJ2017code.

Není implementováno

Rozšíření

FUNEXP

Práce v týmu

Pro práci v týmu jsme využívali komunikačních nástrojů Facebook, Skype a hlavně Discord. Pro sdílení zdrojového kódu jsme používali verzovací systém GIT na serveru www.github.com s desktopovou aplikací GitKraken.

Rozdělení práce

Martin Bobčík - Lexikální analýza, Syntaktická analýza, Sémantická analýza
Martin Lacika - Gramatika
Vojtěch Meluzín - Tabulka symbolů, Precedenční analýza
David Kala -

Procenta bodů byly rozděleny podle odvedené práce na projektu. Martin Lacika, ikdyž napsal část gramatických pravidel se svých procent vzdal ve prospěch zbytku týmu.

Závěr

Projekt jsme z důvodu neúčasti dvou (ze čtyř) kolegů nestihli včas. Nakonec se ale ukázalo, že jednotlivé části překladače nejsou tak těžké na implementaci, jak se ze začátku zdá. Na projektu je tak nejtěžší zvolit si správný tým.

Použitá literatura

Hashovací tabulka. *Algoritmy.net* [online]. 2016 [cit. 2017-12-06]. Dostupné z: https://www.algoritmy.net/article/32077/Hashovaci-tabulka

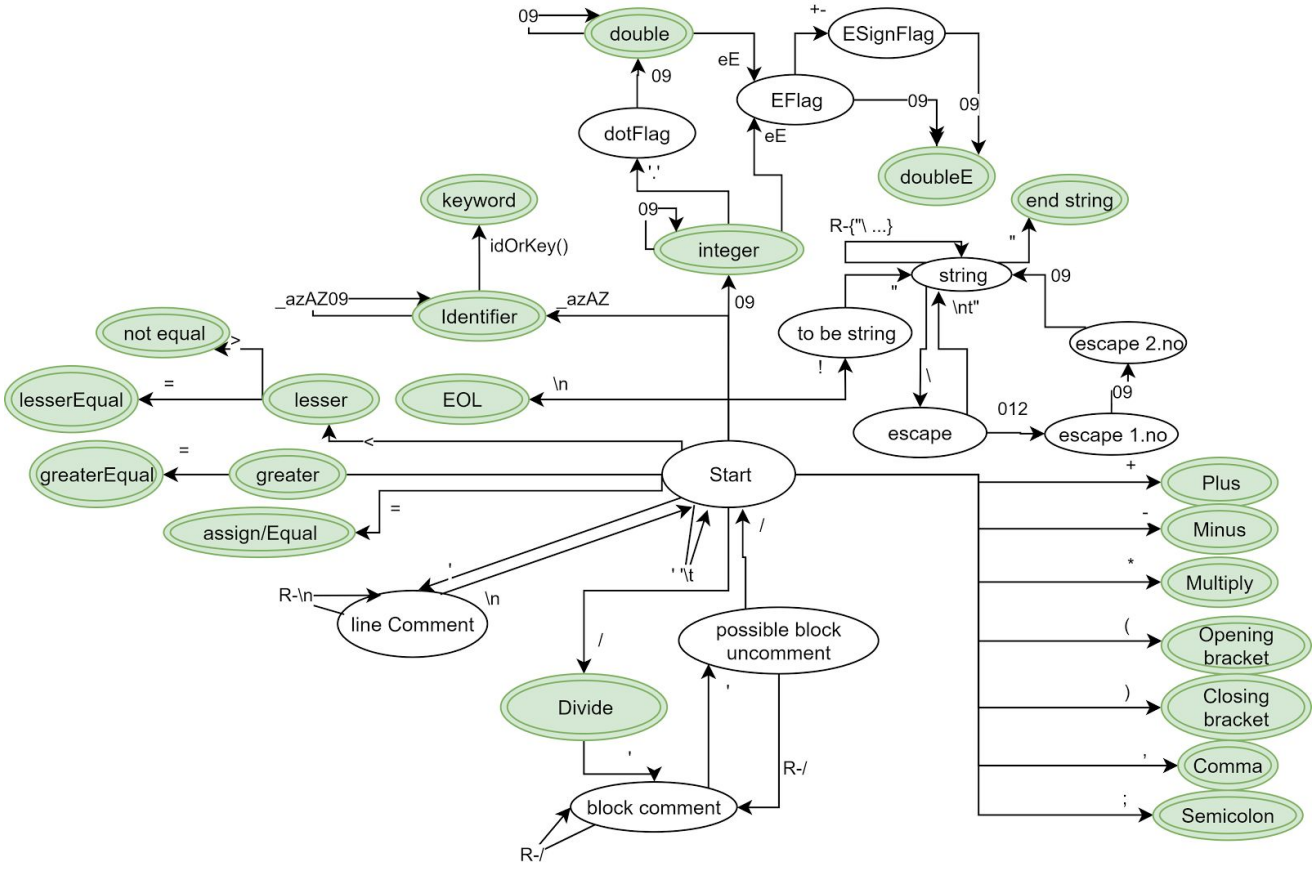
Č. 2 - Přednášky IFJ 2017

Č. 3 - Přednášky IFJ 2007

Č. 4 - Opora IFJ

Č. 5 - Democvičení IFJ

Konečný automat



LL-gramatika

```
1.  <prog> -> KEY_SCOPE END_OF_LINE <scope-st-list>
2.  <prog> -> KEY_DECLARE KEY_FUNCTION IDENTIFIER OPENING_BRACKET
<param-list> KEY_AS <data-type> END_OF_LINE <prog>
3.  <prog> -> KEY_FUNCTION IDENTIFIER OPENING_BRACKET <param-list>
KEY_AS <data-type> END_OF_LINE <fun-st-list>
4.  <prog> -> END_OF_LINE <prog>
5.  <end-prog> -> END_OF_LINE <end-prog>
6.  <end-prog> -> END_OF_FILE
7.  <param-list> -> CLOSING_BRACKET
8.  <param-list> -> IDENTIFIER KEY_AS <data-type> <param>
9.  <param> -> CLOSING_BRACKET
10. <param> -> COMMA IDENTIFIER KEY_AS <data-type> <param>
11. <fun-st-list> -> <fun-stat> <fun-st-list>
```

12. <fun-st-list> -> KEY_END KEY_FUNCTION END_OF_LINE <prog>
13. <fun-st-list> -> KEY_DIM IDENTIFIER KEY_AS <data-type> <assign>
<fun-st-list>
14. <fun-if-stat-list> -> <fun-stat> <fun-if-stat-list>
15. <fun-if-stat-list> -> KEY_ELSE END_OF_LINE
16. <fun-if-stat-list> -> <fun-stat> <fun-else-stat-list>
17. <fun-if-stat-list> -> KEY_END KEY_IF
18. <fun-while-stat-list> -> <fun-stat> <fun-while-stat-list>
19. <fun-while-stat-list> -> KEY_LOOP
20. <fun-stat> -> END_OF_LINE
21. <fun-stat> -> IDENTIFIER OPERATOR_ASSIGN <expression> END_OF_LINE
22. <fun-stat> -> KEY_INPUT IDENTIFIER END_OF_LINE
23. <fun-stat> -> KEY_PRINT <print-list>
24. <fun-stat> -> KEY_IF <expression> KEY_THEN END_OF_LINE
<fun-if-stat-list> <fun-else-stat-list> END_OF_LINE
25. <fun-stat> -> KEY_DO KEY_WHILE <expression> END_OF_LINE
<fun-while-stat-list> END_OF_LINE
26. <fun-stat> -> KEY_RETURN <expression> END_OF_LINE
27. <scope-st-list> -> <scope-stat> <scope-st-list>
28. <scope-st-list> -> KEY_END KEY_SCOPE <end-prog>
29. <scope-st-list> -> KEY_DIM IDENTIFIER KEY_AS <data-type> <assign>
<scope-st-list>
30. <scope-if-stat-list> -> <scope-stat> <scope-if-stat-list>
31. <scope-if-stat-list> -> KEY_ELSE END_OF_LINE
32. <scope-if-stat-list> -> <scope-stat> <scope-else-stat-list>
33. <scope-if-stat-list> -> KEY_END KEY_IF
34. <scope-while-stat-list> -> <scope-stat> <scope-while-stat-list>
35. <scope-while-stat-list> -> KEY_LOOP
36. <assign> -> OPERATOR_ASSIGN <expression> END_OF_LINE
37. <assign> -> END_OF_LINE
38. <scope-stat> -> END_OF_LINE
39. <scope-stat> -> KEY_INPUT IDENTIFIER END_OF_LINE
40. <scope-stat> -> KEY_PRINT <print-list>
41. <scope-stat> -> KEY_IF <expression> KEY_THEN END_OF_LINE
<scope-if-stat-list> <scope-else-stat-list> END_OF_LINE
42. <scope-stat> -> KEY_DO KEY_WHILE <expression> END_OF_LINE
<scope-while-stat-list> END_OF_LINE
43. <scope-stat> -> IDENTIFIER OPERATOR_ASSIGN <expression> END_OF_LINE
44. <param-id-list> -> CLOSING_BRACKET
45. <param-id-list> -> <expression> <param-id>
46. <param> -> CLOSING_BRACKET
47. <param> -> COMMA <expression> <param-id>
48. <print-list> -> <expression> SEMICOLON <print>

49. <print> -> END_OF_LINE
50. <print> -> <expression> SEMICOLON <print>
51. <data-type> -> KEY_STRING
52. <data-type> -> KEY_DOUBLE
53. <data-type> -> KEY_INTEGER

54. <expression>

Precedenční tabulka

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