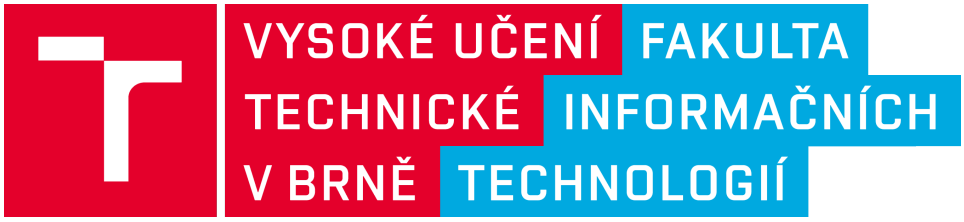


Dokumentace projektu

Implementace překladače imperativního jazyka

IFJ22

Tým xkalen07, varianta TRP

Implementovaná rozšíření: **FUNEXP**

7. prosince 2022

Vedoucí týmu:	Jan Kalenda	xkalen07	28%
Členvé týmu:	Tereza Kubincová	xkubin27	27%
	Jaroslav Streit	xstrei06	25%
	Jan Vacula	xvacul40	20%

1. Úvod

Cílem projektu je implementace překladače imperativního jazyka IFJ22 vytvořeného v jazyce C. Který je podmnožinou jazyka PHP

2. Návrh a implementace

2.1 Lexikální analýza

Tvorba lexikální analýzy se skládá ze dvou částí. První z nich bylo navržení konečného automatu. Následně byl podle něj implementován lexikální analyzátor.

Hlavní "tělo" lexikálního analyzátoru je funkce **get_token()**, volaná ze syntaktického analyzátoru vracející strukturu *token* obsahující potřebné informace jako typ lexému, číslo řádku, identifikátor nebo řetězec, klíčové slovo, nebo číselnou hodnotu (v případě datového typu také informaci, zda daná proměnná může obsahovat hodnotu *null*)

Lexikální analyzátor následně načítá znaky, dokud nepřečte lexém odpovídající jazyku IFJ22, nebo znak nebo sekvenci znaků, které se vyskytnout nesmí (v takovém případě nastává lexikální chyba s návratovou hodnotou 1). Největší část analyzátoru tvoří switch zpracovávající zejména přímočaré lexémy, jako např. porovnávání, aritmetické operace nebo závorky. Naopak např. čísla a identifikátory jsou zpracovávány mimo switch.

Funkce **get_token()** také využívá další tři pomocné funkce, dvě z nich pro zpracování řetězců pro cílový jazyk IFJcode22 (jedna zpracovává celý řetězec, druhá převádí escape sekvence na dekadická čísla, která se následně zapíší jako nová escape sekvence do řetězce) a třetí pro kontrolu klíčových slov.

2.2 Syntaktická analýza

2.2.1 Prediktivní syntaktická analýza

Syntaktická analýza se skládá ze dvou částí. První je prediktivní syntaktická analýza řízená LL tabulkou, která používá zásobník. Hlavním tělem funkce je **parse()** a používá strukturu *Parser* která je sdídlená mezi syntaktickým a sémantickým analyzátorem.

Používá pomocnou funkci **get_next_token()** která slouží pro uchování a návrat až o jeden token zpět, případně získání nového tokenu. Tento návrat používáme pro případy, kdy se provádí výraz bez přiřazení.

Dále používáme funkci **apply_rule()** která na zásobník ukládá požadované tokeny dle LL tabulky

Prediktivní syntaktický analyzátor také tvoří některé instrukce pro generétor které jsou později využívány ke generování kódu.

Sémantické kontroly jsou z velké části řešené až při běhu samotného programu, jsou však řízené instrukcemi, které přicházejí ze samotného syntaktického analyzátoru. Většina těchto kontrol je realizována pomocí hashovací tabulky implementované v souboru symbtable.c. Také je vhodné zmínit že podporujeme volání ještě nedefinované funkce, podobně jako v PHP.

2.2.2 Precedenční syntaktická analýza

Precedenční syntaktický analyzátor, podobně jako prediktivní používá zásobník a svou tabulku. V tomto případě se jedná o precedenční tabulku obsahující znaky $<, >, =$ a prázdná pole pro chybné kombinace. Skládá se z funkce **precedence()** provádějící shiftující pravidla a zápis do hashovací tabulky a funkce **reduce()** provádějící rekurzivní pravidla. **reduce()** také vytváří instrukce pro všechny operátory a přes rozšíření FUNEXP také provádí vyhodnocení volání funkcí, které z tohoto důvodu chybí v LL gramatice i tabulce.
Další funkce je **prec_index()** která vybírá řádek a sloupec pro výběr hodnoty z precedenční tabulky. Implementujeme tedy algoritmus převzatý z přednášek, tedy i s přidáním znaků $< a >$ (= je ignorováno) na zásobník s rozšířením s voláním funkcí. Výběr samotné redukce probíhá, jak již bylo avizováno, ve funkci **reduce()** a je dán přesným součtem hodnot znaků na zásobníku. V případě, že žádné pravidlo neodpovídá součtu a nejedná se o funkci, dochází k chybě 2. Speciálním případem jsou také vestavěné funkce, které jsou vybírány přesně podle jejich identifikátorů a mají tak i své vlastní instrukce pro generátor. Podobně jako v prediktivním syntaktickém analyzátoru většina sémantických kontrol se provádí až za běhu generovaným kódem. Mezi vyjímky patří například pouhá deklarace proměnné nebo použití nedefinované proměnné jako parametr funkce

2.3 Generování kódu

Kód je generován v upravené podobě tříadresného kódu (déle jen 3AC) na základě instrukcí generovaných v parseru. Při zavolání funkce generate se prochází pole alokovaných instrukcí a vykonávají se jednotlivé

funkce instrukcí podle jejich identifikátoru podle enumu instrukcí. Za zmínku také stojí použití zásobníku pro ukládání čísel na návěští pro while, if a funkce kvůli definicím proměnných ve vlastním bloku, který se zaručeně provede jen jednou.
Generátor kódu IFJ22code má funkci pro téměř každý typ instrukce, ale hlavními funkcemi jsou:

- generate, generator_init, generate_add_instruction, generator_free, gen_instruction_constructor

Samotná instrukce má následující strukturu:

- operands_count, params_count, instruct, id, operands, types, params

3. Tabulka symbolů

Naše implementace hashovací tabulky má pevnou velikost a obsahuje prvky provázané do jednosměrně vázaných seznamů v případě kolize hashe. Pomocí tabulek simuluje chování paměťového modelu interpreteru a podporujeme tedy dočasně lokální a globální rámce.

Globální tabulka reprezentující globální rámec nese hodnoty proměnných v hlavním těle programu a to včetně dočasných proměnných využívaných v rámci 3AK (více v generátoru) a funckí. Lokální rámce nesou proměnné a dočasné proměnné definované v těle funkce, dočasný rámec je nastaven po skončení funkce.

4. Práce v týmu

Spolupráce proběhla ve formě setkání na rozdělení práce na scanneru, parseru, generátoru, dokumentace a testech a poté převážně přes github a discord. Na discordu se řešila převážně organizace a na githubu jsme pro každou část měli separátní větev která se později mergovala do hlavní větvě přes pull request. Použili jsme také CI/CD služby (přesněji CircleCI) a při každém pushi na remote jsme spustili automatizované testy.

4.1 Rozdělení práce v týmu

Jan Kalenda:
Precedenční SA, generátor(operátory), testovací kostra, vedení týmu, sémantická analýza

Tereza Kubincová:
Prediktivní SA, generátor(if,while,funkce) tabulka symbolů, makefile, sémantická analýza

Jaroslav Streit:
Lexikální analýza, generátor(builtins), testy

Jan Vacula:
Dokumentace, testy

Zdůvodnění pro nerovné rozdělení bodů.
28% Vedení týmu + těžší části projektu.
27% Těžší části projektu. 25% Standartní práce na projektu.
20% Pouze vypracování dokumentace.

LLtabulka

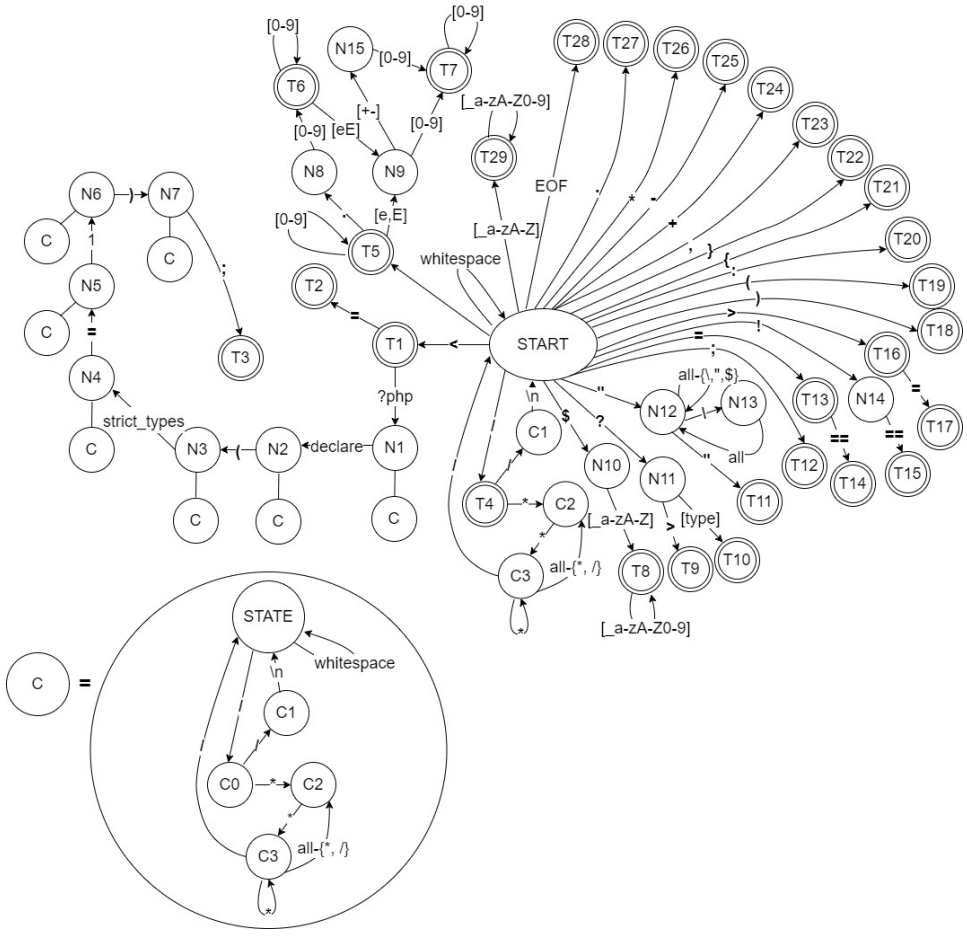
	VALID	ID	IF	WHILE	RETURN	FUNCTION	GLOBAL	VAR	TYPE	;	=	,	EOF	END	EPS	STRING	INT	FLOAT	(	OPERATOR
<prog>	1																			
<st-list>		2	2	2	2	2	2	2					3	4		2	2	2	2	
<st>		7	11	12	10	5	6	9								7	7	7	7	
<small-st>										15	14									15
<param-def>									16						8					
<param-name>								17												
<param-list>												19			18					
<body>		20	20	20	20		20	20							13	20	20	20	20	

1. `<prog\> -> VALID <st-list>`
2. `<st-list> -> <st> <st-list>`
3. `<st-list> -> EOF`
4. `<st-list> -> END`
5. `<st> -> FUNCTION ID(<param-def>) : TYPE {<body>}`
6. `<st> -> GLOBAL VAR <small-st>`
7. `<st> -> <expr>`
8. `<st> -> <param-def> → eps`
9. `<st> -> VAR <small-st>`
10. `<st> -> RETURN <expr>`
11. `<st> -> IF <expr> {<body>} ELSE {<body>}`
12. `<st> -> WHILE <expr> {<body>}`
13. `<body> -> eps`
14. `<small-st> -> = <expr>`
15. `<small-st> -> <expr>`
16. `<param-def> -> TYPE <param-name> <param-list>`
17. `<param-name> -> VAR`
18. `<param-list> -> eps`
19. `<param-list> -> , <param-def>`
20. `<body> -> <st> <body>`

Precedenční tabulka

	* /	"+ - ."	i	(	)	R	"=== !=="	$	*f*	,
* /	>	>	<	<	>	>	>	>	<	>
"+ - ."	<	>	<	<	>	>	>	>	<	>
i	>	>			>	>	>	>		>
(	<	<	<	<	=	<	<		<	=
)	>	>			>	>	>	>		>
R	<	<	<	<	>		>	>	<	>
"=== !=="	<	<	<	<	>	<		>	<	>
$	<	<	<	<		<	<		<	
f				=						
,	<	<	<	<	=	<			<	=

FSM


START
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10
T11
T12
T13
T14
T15
T16
T17
T18
T19
T20
T21
T22
T23
T24
T25
T26
T27
T28
T29
N1
N2
N3
N4
N5
N6
N7
N8
N9
N10
N11
N12
N13
N14
N15
C1
C2
C3
?php
declare
strict_types
[0-9]
[eE]
[+-]
[e,E]
[_a-zA-Z]
[_a-zA-Z0-9]
whitespace
EOF
\n
all-{\,",$}
all-{*, /}
[type]
C
=
STATE
C0