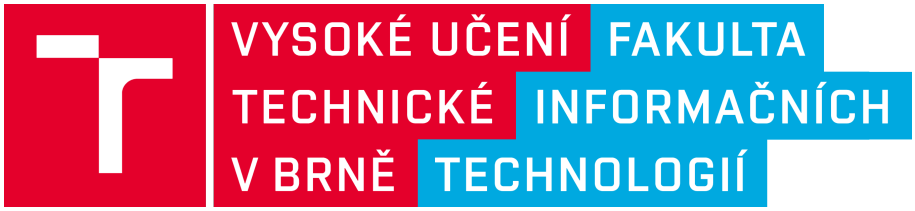
Dokumentace k projektu do předmětu IFJ

Implementace překladače imperativního jazyka IFJ24

Varianta TRP-IZP
Tým xbucek17



Formální jazyky a překladače
Akademický rok 2024/25

Matěj Bucek (xbucek17), 25%
David Dumbrovský (xdumbr00), 25%
Julie Provazníková (xprova06), 25%
Matěj Říčný (xricny01), 25%

4. prosince 2024

Obsah

1 Úvod

Cílem tohoto projektu bylo implementovat kompilátor zdrojového jazyka IFJ24 do cílového jazyka IFJcode24. Jazyk IFJ24 je podmnožinou jazyka Zig.

Pokud se během překladu nevyskytne žádná chyba, bude na standardní výstup vygenerován překlad do cílového jazyka a program se následně ukončí s návratovou hodnotou 0. Pokud při překladu k chybě dojde, je chyba oznámena na standardní chybový výstup a program se ukončí s korespondující návratovou hodnotou.

2 Implementace překladače

Projekt byl sestaven z několika dílčích částí. Tato kapitola bude zaměřena na popis implementace jednotlivých částí a způsob komunikace mezi nimi.

2.1 Lexikální analýza

Jako první část překladače jsme implementovali lexikální analyzátor. Lexikální analýza je spuštěna neprodleně po spuštění překladače invokací funkce `scanTokens(FILE*)`, která načítá znak po znaku ze zdrojového kódu, který se nachází na standardním vstupu. Přečtené znaky jsou pomocí konečného automatu převedeny na struktury typu `Token`, která se skládá z typu daného tokenu, lexému, uloženého jako řetězec, a čísla řádku, na

kterém se daný token ve zdrojovém kódu nacházel. Jednotlivé tokeny jsou skládány do obousměrně zřetězeného spojového seznamu, který je následně využit v syntaktickém analyzátoru.

Lexikální analyzátor byl implementován na základě deterministického konečného automatu popsaného v diagramu 6.1. Implementace konečného automatu je realizována ve funkci `scanTokens(FILE*)`, která postupně zpracovává jednotlivé znaky vstupu v cyklu, který se ukončí po načtení koncového znaku EOF. Každý stav automatu je reprezentován samostatnou větví příkazu `switch`, kde jednotlivé případy `case:` odpovídají konkrétním stavům automatu. Stav TRAP, uvedený v diagramu 6.1 simuluje ukončení programu chybou s návratovým kódem 1.

Lexikální analyzátor při zpracování datového typu `"[]u8"` nejdříve rozpozná znak "["a inicializuje datovou strukturu pro ukládání znaků, které následují za "[". Lexikální analyzátor pokračuje ve čtení znaků, dokud nenarazí na jeden ze znaků, které jsou považovány za ukončující (bílé znaky, EOF, =, (,) nebo {). Po načtení řetězce lexikální analyzátor porovná jeho obsah s očekávaným literálem []u8, pokud dojde ke shodě, lexikální analyzátor vytvoří nový token typu U8, jinak je program ukončen s chybou 1.

Identifikátory jsou po načtení porovnávány s prvky z množiny klíčových slov, tím se zjistí, zda se jedná o obecný identifikátor nebo klíčové slovo.

Pokud lexikální analyzátor narazí na symbol, který není součástí definovaného jazyka, ukončí program s návratovým kódem 1.

2.2 Syntaktická analýza

Syntaktická analýza je implementována pomocí rekurzivního sestupu. Funkce `parse()` je zavolána po načtení všech tokenů lexikálním analyzátorem a provede syntaktickou analýzu celého zdrojového kódu.

Pro syntaktickou analýzu výrazů byla implementována precedenční tabulka, která je využita při zpracování výrazů. Výrazy jsou zpracovány pomocí precedenční analýzy, která je implementována pomocí zásobníku a precedenční tabulky. V případě, že je zdrojový kód syntakticky správný, je vygenerován derivační strom, který je následně využit v sémantické analýze.

2.3 Sémantická analýza

Sémantická analýza prochází derivačním stromem, vytváří symboly v tabulce symbolů a kontroluje správnost použití identifikátorů a typů. Je také použit zásobník tabulek symbolů, kde je pro každý rozsah platnosti vytvořena nová tabulka symbolů a uložena na zásobník.

Analýza provádí typovou kontrolu, kontrolu počtu a typu argumentů funkcí a kontrolu návratových hodnot funkcí. Je implementovaná také kontrola, zdali veškeré větve funkce obsahují návratovou hodnotu.

2.4 Generátor cílového kódu

Poslední částí naší implementace je generátor cílového kódu. Generace kódu je započata invokací funkce `generate_code()`, která prve provede inicializaci pomocných struktur generátoru, zejména tabulky rámců a překladového kontextu(Možná odstranit). Po inicializaci je vygenerováno návěští a začíná průchod derivačního stromu.

2.4.1 Generování funkcí

Funkci tvoří její lokální rámec a návěští ve tvaru `název_funkce`. Lokální rámec je vždy vytvořen a vložen na zásobník rámců při invokaci dané funkce. Stejně tak je i rámec ze zásobníku odstraněn následně poté, co dojde k návratu z funkce.

Argumenty funkce jsou vždy uloženy ve speciálních proměnných, které značíme `%argX`, kde `X` je pořadí tohoto argumentu. Opravdové názvy těchto argumentů, které jsou v kódu funkce použity, jsou nahrazeny těmito pomocnými názvy. Tento postup byl zvolen především proto, že při invokaci funkce nejsou známy opravdové názvy těchto argumentů a bylo by v naší architektuře překladače obtížné tyto názvy zjistit.

2.4.2 Generování proměnných

Generace definice proměnných je vždy hlídána tabulkou rámců. Pokud je již proměnná v daném rámci definována, není definována znovu.

2.4.3 Generování výrazů

Generace výrazů využívá rekurzivní funkce `generate_temporary_expression(treeNode*, char**)`, která podstrom výrazů projde a postupně od nejhlubšího uzlu generuje instrukce pro výpočet výrazu. Název proměnné, do které byl mezivýsledek uložen, je následně předán pomocí druhého parametru této funkce.

Pokud se ve výrazu nachází invokace funkce, je vytvořen nový dočasný rámec, do kterého jsou vloženy jednotlivé argumenty. Rámec je následně vložen na zásobník rámců a funkce je invokována. Po návratu z funkce je rámec ze zásobníku odstraněn a návratová hodnota uložena do pomocné proměnné.

Pomocné proměnné generované v obou těchto případech mají jména ve formátu `%tmpX`, kde `X` je číslo, které dosud nebylo při generování použito. Hodnota dalšího použitelného čísla je uložena v kontextu generátoru kódu.

2.4.4 Generování podmínek

Generování podmínek je řešeno pomocí návěští, která jsou vytvořena pro každou podmínku. Pokud je podmínka splněna, pokračuje se beze skoku do těla podmínky. Na konci těla podmínky je skok na úplný konec části podmínky. To je důležité, jelikož za tělem podmínky může následovat sekce else. Pokud podmínka splněna není, je proveden skok buď na návěští sekce else, nebo na konec bloku podmínky.

Návěští podmínek je ve tvaru `if_X`, kde `X` je číslo, které dosud nebylo při generování použito. Hodnota dalšího použitelného čísla je uložena v kontextu generátoru kódu.

2.4.5 Generování cyklů

Při generování cyklů je důležité vytknout veškeré definice proměnných, ke kterým by mohlo dojít při provádění cyklu již před tento cyklus, aby nedošlo k jejich redefinici. K tomu je využita simulace tohoto cyklu, která globálně vypne generování jakéhokoliv příkazu, který není generace definice proměnné. Podobného principu je využito po prvním návěští cyklu, kde je globálně vypnuta generace příkazu definice proměnné. Je povoleno generování jinak veškerých instrukcí a definice proměnné v dočasném rámci, který by byl vytvořen při volání funkce uvnitř cyklu, a může tak dojít k bezchybné redefinici.

První návěští je využito pro evaluaci výrazu a ověření platnosti. Po něm následuje skok na návěští na konci bloku cyklu, pokud by došlo k tomu, že výraz není platný. Před tímto návěštím je skok na první návěští pro zaručení opakování cyklu.

Návěští cyklů je generováno ve tvaru `while_X`, kde `X` je číslo, které dosud nebylo při generování cyklu použito.

2.5 Překlad

Součástí odevzdaného archivu je soubor Makefile. Překlad se provádí příkazem `make`, který využívá překladač gcc. Příkaz `make` vytvoří spustitelný program s názvem `program`.

3 Speciální datové struktury

V rámci implementace překladače jsme vytvořili několik speciálních datových struktur, které byly využity v různých částech překladače.

3.1 Tabulka s rozptýlenými položkami

Podle zadání byla implementována tabulka s rozptýlenými položkami, která je implicitně zřetězená. Tato struktura je použita v sémantickém analyzátoru jako tabulka symbolů a nachází se v souboru `symtable.h`. Tabulka má definovanou maximální velikost 200 položek.

3.2 Tabulka rámců

Tabulka rámců je datová struktura, která simuluje fungování rámců v interpretru cílového jazyka IFJcode24. Každý rámec obsahuje informace o proměnných, které jsou v něm definovány.

4 Práce v týmu

V této kapitole popíšeme způsob, jakým jsme pracovali na projektu a jak jsme si rozdělili práci.

4.1 Komunikace

Pro komunikaci v týmu jsme využívali platformu Discord, která nám umožňovala rychlou a snadnou výměnu informací. Na Discordu jsme se pravidelně scházeli a konzultovali aktuální stav projektu.

4.2 Verzovací systém

Pro správu zdrojových kódů jsme využili verzovací systém Git. Kód byl hostován na platformě GitHub, kde byl vytvořen soukromý repozitář pro tento projekt.

4.3 Rozdělení práce

V rámci týmu jsme si práci rozdělili na dvojice, přičemž každá dvojice se věnovala konkrétní části projektu.

Jméno	Práce
Matěj Bucek	vedoucí týmu, tabulka s rozptýlenými položkami, generátor kódu, testování, dokumentace, závěrečná prezentace
David Dumbrovský	lexikální analýza, zpracování výrazů
Julie Provazníková	generátor kódu, testování, dokumentace, závěrečná prezentace
Matěj Říčný	syntaktická analýza, semantická analýza

Tabulka 1: Rozdělení práce

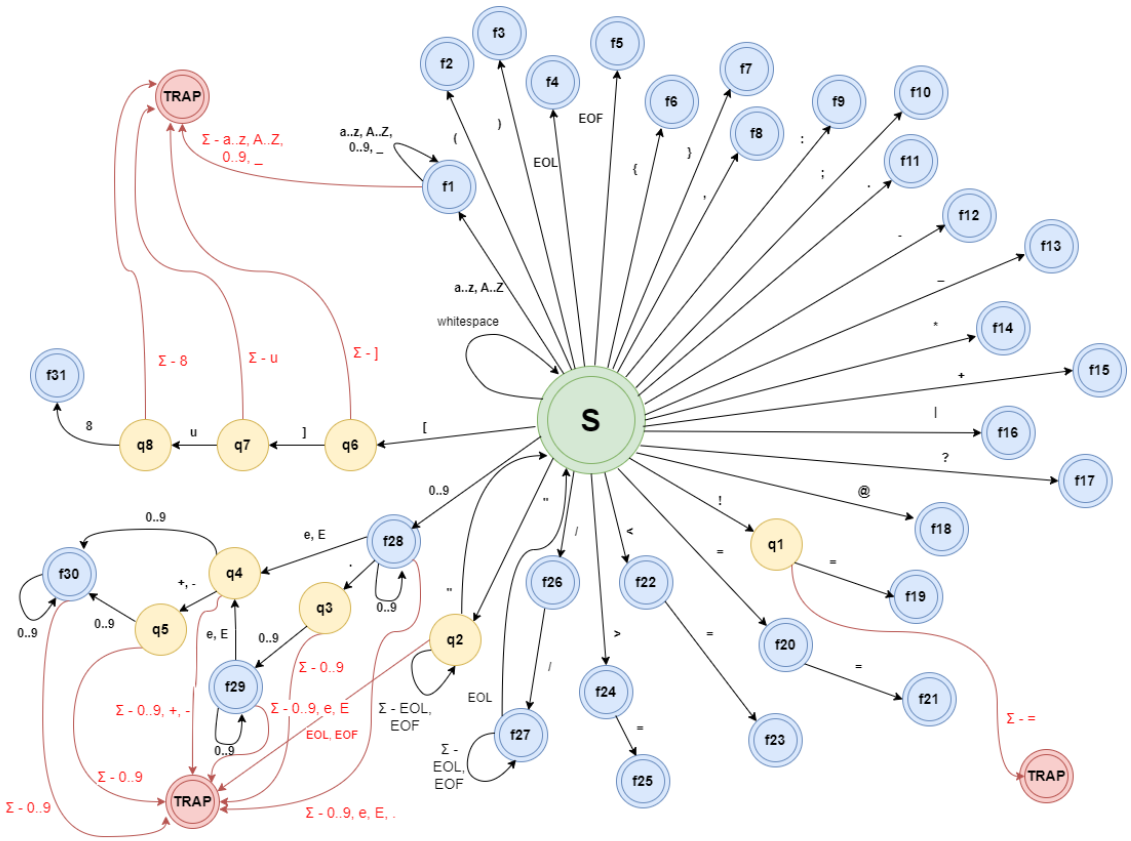
5 Závěr

Díky včasnému zahájení práce jsme měli dostatek času na implementaci klíčových funkcí, a to do podoby, se kterou jsme spokojeni.

Velkou roli hrála i možnost pokusného odevzdávání, které jsme využily v obou případech a díky které jsme mohli inkrementálně zlepšovat a opravovat náš překladač. Kromě toho jsme projekt testovali jak na veřejně dostupných testovacích sadách, tak na vlastních testech, abychom zajistili co nejvyšší kvalitu výsledného řešení.

6 Přílohy

6.1 Konečný automat



S - Počáteční stav
f1 - Identifikátor
f2 - Levá závorka
f3 - Pravá závorka
f4 - EOL
f5 - EOF
f6 - Levá složená závorka
f7 - Pravá složená závorka
f8 - Čárka
f9 - Dvojtečka
f10 - Středník
f11 - Tečka
f12 - Minus
f13 - Podtržítko
f14 - Krát
f15 - Plus
f16 - Roura
f17 - Otazník
f18 - Zavináč
q1 - Vykřičník
f19 - Nerovnost
f20 - Přiřazení
f21 - Rovná se
f22 - Méně
f23 - Méně rovno
f24 - Více
f25 - Více rovno
f26 - Děleno
f27 - Komentář
q2 - Řetězec
f28 - Číslo
q3 - Desetinné tečka
f29 - Desetinné číslo
q4 - Exponenciální část
q5 - Znaménko exponentu
f30 - Exponenciální tvar čísla
q6 - Levá hranatá závorka
q7 - Pravá hranatá závorka
q8 - []u
f31 - Datový typ řezu

Obrázek 1: Deterministický konečný automat

6.2 LL-gramatika

```
1. <program_start> -> CONST IFJ = @import("ifj24.zig"); <program>

2. <program> -> PUB FN ID ( <parameters> ) <return_type> <function_body> <program>
3. <program> -> EOL <program>
4. <program> -> EOF

5. <parameters> ID <var_type> <parameters_list>
6. <parameters> -> ε

7. <parameters_list> -> , ID <var_type> <parameters_list>
8. <parameters_list> -> ε

9. <function_body> -> { <statement> }
10. <function_body> -> ε

11. <statement> -> CONST ID <var_type> <var_value> ; <statement>
12. <statement> -> VAR ID <var_type> <var_value> ; <statement>
13. <statement> -> _ <var_value> ; <statement>
14. <statement> -> IF ( <null_or_expression> ) <is_null> { <statement> } ELSE { <statement> } <statement>
15. <statement> -> WHILE ( <null_or_expression> ) <is_null> { <statement> } <statement>
16. <statement> -> IFJWRITE ( <expression> ) ; <statement>
17. <statement> -> RETURN <expression> ;
18. <statement> -> ε

19. <is_null> -> | ID |
20. <is_null> -> ε

21. <var_type> -> : <var_type_null>
22. <var_type> -> ε
23. <var_type_null> -> ? <type>
24. <var_type_null> -> <type>

25. <type> -> I32
26. <type> -> F64
27. <type> -> []U8

28. <var_value> -> = <expression>

29. <return_type> -> <var_type_null>
30. <return_type> -> VOID
```

Obrázek 2: LL-gramatika

6.3 LL-tabulka

	CONST	PUB	EOL	EOF	ID	,	(	)	VAR	_	IF	WHILE	IFJWRITE	RETURN	|	:	?	I32	F64	[]U8	=	VOID	$
<program_start>	1																						
<program>		2	3	4																			
<parameters>					5																		6
<parameters_list>						7																	8
<function_body>							9																10
<statement>	11							18	12	13	14	15	16	17									
<is_null>							20								19								20
<var_type>																21					22		22
<var_type_null>																	23	24	24	24			
<type>																		25	26	27			
<return_type>																		29	29	29		30	

Obrázek 3: LL-tabulka

6.4 Precedenční tabulka

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

Obrázek 4: Precedenční tabulka