Dokumentace projektu z předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ19

Tým 057, varianta I

Martin Koči	(xkocim05)	30%
Michal Koval	(xkoval17)	30%
Zuzana Hradilová	(xhradi16)	20%
Magdaléna Ondrušková	(xondru16)	20%

Obsah

1 Úvod

Cílem projektu bylo vytvořit překladač ze zdrojového jazyka `IFJ19`, založeného na programovacím jazyce Python, do jazyka `IFJcode19`. Program byl implementován v jazyce C.

2 Implementace

Projekt je rozdělen do několika částí, které přestože byly implementovány samostatně, kvůli jejich rozdělení mezi členy týmu, spolu úzce spolupracují a dohromady tvoří výsledný překladač.

2.1 Lexikální analýza

Před samotnou implementaci lexikální analýzy jsme navrhli konečný deterministický automat bez ε-přechodů pro jazyk `IFJ19`. Vytvořili jsme diagram, pomocí nějž jsme implementovali lexikální analýzu. Lexikální analýza je implementována v souborech `scanner.c` a `scanner.h`. Na řešení zanoření jsme použili dynamicky zásobník integerů, který je implementován v souborech `stack.c` a `stack.h`.

Hlavní funkcí lexikální analýzy je funkce `scan`, která zpracovává vstup, znak po znaku, a vyhodnocuje o jaký token se jedná. Na ukládání a předávání tokenů používáme strukturu `Token`, která obsahuje typ tokenu z výčtového typu `Tokens` a ukazatel na řetězec, pomocí kterého ukládáme atribut tokenu. Na předávání tokenu dalším analýzám slouží funkce `get_next_token`, `preload_token` a `unget_token` a k nim naimplementovaná fronta tokenů a funkce pro práci s ní. Funkce `get_next_token` je základní funkcí celého překladače a je taktéž nejvíc-krát volanou funkcí. Tokeny jsou vkládány do fronty a následně jsou zpracovány. Funkce `preload_token` zavolá `scan`, vrátí získaný token a současně ho uloží na konec fronty.

V případě chybného vstupního kódu lexikální analýza vrátí lexikální chybu, případně interní chybu překladače.

2.2 Syntaktická analýza

2.2.1 LL-gramatika

Nejčastějším problémem při tvorbě LL-tabulky bylo, že se ze začátku nejednalo o LL(1)-gramatiku. Tento problém se nakonec podařilo vyřešit pomocí vytvoření nového neterminálu, probraného na přednáškách. Mnoho chyb v LL-gramatice bylo odhaleno až při samotném programování a následně byla upravená LL-gramatika i LL-tabulka. Naše výsledná LL-gramatiku je popsána v sekci 5 na str. 5

2.2.2 LL-tabulka

LL-tabulka byla vytvořena na základe LL-gramatiky pomocí množiny predict, která se používá k vytvoření rekurzivního sestupu.

2.2.3 Metoda rekurzivního sestupu

Pro tuto metodu jsme se rozhodli z důvodů její relativně lehké implementace a také proto, že tato metoda, na rozdíl od prediktivní analýzy řízenou LL-tabulkou, nevyžaduje implementaci zásobníku.

2.2.4 Implementace a optimalizace

Při rekurzivním sestupu jsme implementovali funkce které reprezentovaly dané neterminály v LL-tabulce (str. 8). V `main.c` je načten 1. token a následně je poslán do funkce `prog`. Tato funkce je výchozí funkcí celé syntaktické analýzy. Je zde vyhodnoceno o jaký typ tokenu se jedná a jsou ověřené případné syntaktické chyby. Pokud vyhodnocení proběhlo v pořádku je volaná funkce `st-list`, která

rekurzivně vyhodnocuje program. Následně jsou zavolány i ostatní analýzy. V případě zjištění chyby je rekurzivní sestup ukončen a vrácen návratový kód 1. zjištěné chyby.

Při přiřazení výsledku funkce do proměnné je proměnná uložená do symbolické tabulky s nedefinovaným datovým typem, z důvodu neznámého návratového typu dané funkce.

V případě přiřazení konstant a výrazů, v kterých jsou všechny proměnné definovány, do proměnné, se do symbolické tabulky uloží výsledný datový typ. Výraz, který nepřiřazujeme do žádné proměnné, se vyhodnotí a pokud jsou všechny proměnné definovaného datového typu, výraz není generován z důvodu optimalizace.

Pokud se ve výrazu nebo v parametrech funkce nachází identifikátor již definované funkce, identifikátor je vyhodnocen jako chybný a je vrácena chyba s návratovým kódem 3.

Při volání funkce v těle jiné funkce se do symbolické tabulky uloží jen počet parametrů volané funkce a pokud bude později definovaná, zkontroluje se zda odpovídá počet parametrů. Pokud volaná funkce nebude v celém souboru definovaná, je vrácena taktéž chyba s návratovým kódem 3. Při definování nové proměnné v těle funkce, je zkontrolováno, zda neexistuje globální proměnná se stejným identifikátorem, a pokud ano, zda nebyla v dané funkci již dříve použita. Jestliže použita byla, jedná se rovněž o chybu s návratovým kódem 3.

2.2.5 Zpracování výrazů

Výrazy zpracováváme odděleně od ostatcích částí syntaktické analýzy, v souborech `expression_parser.c` a `expression_parser.h`. Analýzu výrazů voláme z hlavní části syntaktické analýzy pomocí funkce `callExpression`. Tato funkce dostane jako jediný parametr 1. token výrazu. V případě chyby funkce navrátí chybový kód, pokud je kontrolovaný výraz správný je zpátky do syntaktické analýzy vráceno OK.

Tokeny ze vstupu jsou načítány do seznamu, při načítání je zaznamenáván počet závorek a kontroluje se dělení nulou. Probíhá kontrola sémantické správnosti tokenu. Souběžně jsou operandy načítány do seznamu `operandList`, který se využívá při generování kódu.

Samotné zpracování výrazu probíhá pomocí precedenční analýzy. Je načten token, opět se zkontroluje jeho sémantická správnost, a následně je podle pravidel precedenční tabulky (viz. Tabulka 3 na str. 7) vložen na zásobník. Následně je výraz zpracovaný pomocí algoritmu probraného na přednášce IFJ.

Nakonec je zavolána funkce na generování výrazu s optimalizací popsanou výše.

2.3 Sémantická analýza

Sémantická analýza je používána v téměř všech částech překladače, kromě lexikální analýzy. Je zde kontrolováno zda byly funkce a proměnné definovány. Skládá se z mnoha pomocných funkcí, které zjišťují například zda je identifikátor zabudovaná funkce, kontrolování počtu parametrů a dalších. Další pomocné funkce slouží především ke kontrole sémantické správnosti výrazů.

Při definici proměnné je volaná funkce která přímo generuje kód odpovídající definici proměnné. A případně do proměnné přiřadí výstupnou hodnotu volané funkce.

V rámci sémantické analýzy probíhají další kontroly správného definování a použití proměnných a funkcí.

2.4 Tabulka symbolů

Tabulka symbolů slouží k zaznamenávání dat o proměnných, funkcích a jejich parametrech. Byla implementována jako binární vyhledávací strom, o kterém jsme se učili v předmětu IAL.

Proměnné v `IFJ19` jsou rozdělené na globální, definované v hlavním těle programu a lokální, definované ve funkci. Vzhledem k potřebě uchovávat různá data o globálních a lokálních symbolech byli funkce pro práci s tabulkami rozděleny.

K prvkům v tabulce se přistupovalo pomocí unikátního klíče (id), pomocí nějž bylo možné symboly vyhledávat a měnit obsah uložených dat o dané proměnné.

Při vložení globálního symbolu, který byl funkcí, byli taktéž uloženy data o jejích parametrech a ukazatel na seznam parametrů, který byl implementován pomocí dvousměrně vázaného lineárního seznamu, taktéž probraného v předmětu IAL.

Funkce pro práci se symboly, definované v tabulce symbolů, byly následně použity při syntaktické analýze a generovaní kódu, především pro kontrolu datových typů a vlastností funkcí (např. byla-li použitá funkce někde v programu definována, ověření počtu parametru, atd.)

2.5 Generování kódu

Generování výsledného kódu v jazyku `IFJcode19` je implementováno v souborech `generator.c` a `generátor.h`. Základní princip našeho řešení je založen na implementaci funkcí na generování jednotlivých struktur jazyka, jako například definice proměnné nebo podmínka `if`. Tyto funkce jsou volány ve správném kontextu syntaktickou a sémantickou analýzou.

Výsledný kód se netiskne přímo na výstup, ale ukládá se do seznamu výsledného kódu. K tomuto slouží dříve implementovaný, jednosměrně vázaný seznam, `Code_list`, jehož elementy jsou typu `Code_line`. `Code_line` je struktura obsahující ukazatel na `Code`, což je pomocná struktura k ukládání samotné instrukce, ukazatel na `Code_line in_between`, který slouží pro vložení instrukcí před aktuální instrukci, a ukazatel na `Code_line next`, který ukazuje na další instrukci v seznamu.

Finálně přeložený kód se tiskne na výstup až na úplném konci běhu překladače, za podmínky, že všechny analýzy skončily úspěšným návratovým kód. Vestavěné funkce se ukládají do vlastního seznamu `builtin_list` a tisknou se až jako poslední.

Generování unikátních návěští jsme vyřešili pomocí counteru a zásobníku. Každé návěští začíná znakem $ a za ním následuje vlastní název návěští.

U generování funkce není generování unikátního názvu nutné, jelikož nemůže dojít k redefinici funkce.

Generování `while`, `if` a `print` přidáváme na konec návěští hodnotu příslušného counteru. V případě zanořených `while` a `if` index koncových návěští popujeme ze zásobníku, do kterého byl vložen index při generování začátečního návěští.

3 Práce v týmu

Na projektu jsme pracovali ve čtyřčlenném týmu. Po zveřejnění zadaní jsme se sešli na 1. týmové schůzce, ujasnili si, co všechno bude potřeba udělat a rozdělili si úkoly.
Na většině částech projektu jsme pracovali samostatně nebo ve dvojicích, ale často jsme se potkávali a společně konzultovali řešení. Komunikace probíhala také přes Discord a Messenger.
Zdrojové soubory jsme sdíleli pomocí GitHubu, což nám umožnilo jejich rychlé sdílení a složení výsledného projektu.

3.1 Deadliny a pokusná odevzdání

Pro každý úkol jsme si určili termín, do kterého jsme se snažili o jeho dokončení. Dodržet tyto termíny nám velmi pomohli i možnosti vyzkoušet si odevzdání projektu pomocí pokusného odevzdání. Nejdříve jsme vytvořili hlavičkové soubory, v nichž jsou definované funkce, které jsou použity v různých částech a propojují tím celý projekt. Před prvním pokusném odevzdání jsme zvládli dokončit lexikální, syntaktickou a většinu sémantické analýzy a začít s generováním kódu. Mezi prvním a druhým odevzdáním jsme se zaměřili na generování kódu, odchycení běhových chyb a testování. Později jsme náš projekt dolaďovali a opravovali chyby. Tyto možnosti vyzkoušet si projekty odevzdat „nanečisto“ pro nás byly velmi užitečné a „donutily“ nás nenechat řešení projektu na poslední chvíli.

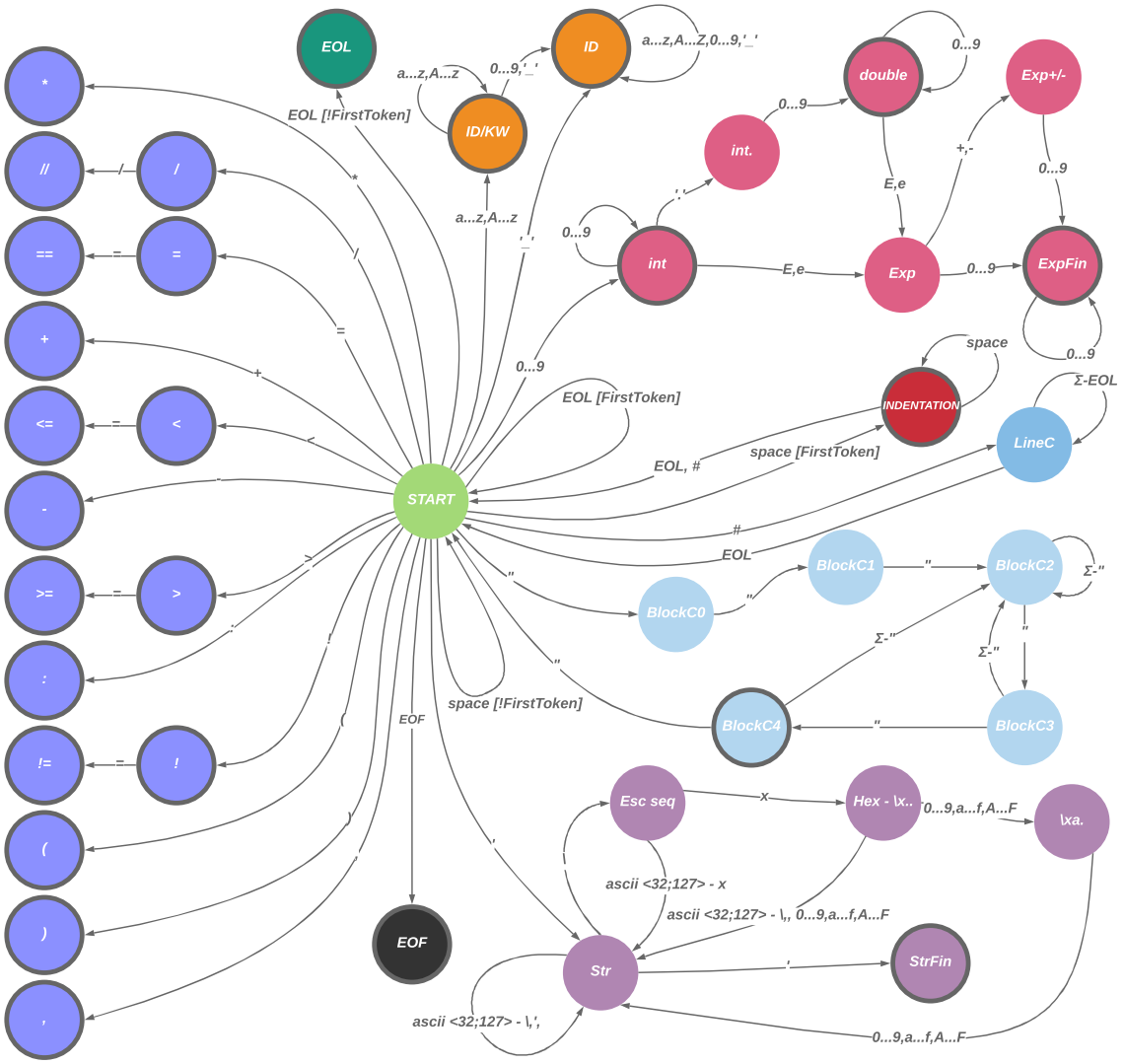
3.2 Rozdělení práce

Jednotlivé části projektu byly rozděleny podle následující tabulky. Rozhodli jsme se rozdělit procenta podle vykonané práce jednotlivých členů týmu, rozsahu napsaného kódu a času stráveného řešením projektu.

Martin Koči	Vedoucí týmu, syntaktická analýza, sémantická analýza	30%
Magdaléna Ondrušková	Syntaktická analýza, zpracování výrazů, testování	20%
Zuzana Hradilová	Tabulka symbolu, testování, dokumentace	20%
Michal Koval	Lexikální analýza, generování kódu	30%

Tabulka 1: Rozdělení práce v týmu

4 Diagram konečného automatu pro lexikální analýzu



Obrázek 1: Návrh konečného automatu

Legenda:
EOL - znak konce řádku
EOF - znak konce souboru
Σ - abeceda (kromě EOF)
a...b - znak v rozmezí od a po b
a,b,... - libovolný znak z výčtu
space - mezerník
ascii <a ; b> znaky ASCII tabulky v rozmezí od a po b
[x] - flag x musí být nastavený na true
[!x] - flag x musí být nastavený na false
Σ-a, ... - abeceda kromě znaku a ...

5 LL Gramatika

1: <prog> -> <st-list>
2: <st-list> -> <stat><st-list>
3: <st-list> -> EOF
4: <stat> -> def id (<params>: EOL INDENT <func-nested-st-list>
5: <stat> -> id <after_id>
6: <stat> -> expr <eof-or-eol>
7: <stat> -> pass <eof-or-eol>
8: <stat> -> while expr : EOL INDENT <nested-st-list>
9: <stat> -> if expr : EOL INDENT <nested-st-list>else : EOL INDENT <nested-st-list>
10: <params> ->)
11: <params> -> id <next-param>
12: <next-param> -> , id <next-param>
13: <next-param> ->)
14: <arg-params> ->)
15: <arg-params> -> <value><arg-next-params>
16: <arg-next-params> -> , <value><arg-next-params>
17: <arg-next-params> ->)
18: <value> -> none
19: <value> -> float
20: <value> -> string
21: <value> -> int
22: <value> -> id
23: <func-nested-st-list> -> <func-nested-stat><next-func-nested-st-list>
24: <func-nested-stat> -> pass <eof-or-eol>
25: <func-nested-stat> -> if expr : EOL INDENT <func-nested-st-list>else : EOL INDENT <func-nested-st-list>
26: <func-nested-stat> -> expr <eof-or-eol>
27: <func-nested-stat> -> while expr : EOL INDENT <func-nested-st-list>
28: <func-nested-stat> -> id <after_id>
29: <func-nested-stat> -> return <after-return>
30: <next-func-nested-st-list> -> DEDENT
31: <next-func-nested-st-list> -> <func-nested-st-list>
32: <nested-st-list> -> <nested-stat><next-nested-st-list>
33: <nested-stat> -> pass <eof-or-eol>
34: <nested-stat> -> if expr : EOL INDENT <nested-st-list>else : EOL INDENT <nested-st-list>
35: <nested-stat> -> expr <eof-or-eol>
36: <nested-stat> -> while expr : EOL INDENT <nested-st-list>
37: <nested-stat> -> id <after_id>
38: <next-nested-st-list> -> DEDENT
39: <next-nested-st-list> -> <nested-st-list>
40: <after_id> -> = <assign>
41: <after_id> -> <def-id>
42: <assign>->expr <eof-or-eol>
43: <assign> -> id <def-id>
44: <def-id> -> (<arg-params><eof-or-eol>
45: <def-id> -> <eof-or-eol>
46: <after-return> -> expr <eof-or-eol>
47: <after-return> -> <eof-or-eol>
48: <eof-or-eol> -> EOL
49: <eof-or-eol> -> EOF

6 LL Tabulka

	eof	eol	def	id	(	:	indent	=	return	pass	while	if	else	,	)	none	float	string	int	dedent	expr
<prog>	1		1	1						1	1	1									1
<st-list>	3		2	2						2	2	2									2
<stat>			4	5						7	8	9									6
<params>				11											10						
<nested-st-list>				32						32	32	32									32
<assign>				43																	42
<next-param>														12	13						
<arg-params>				15											14	15	15	15	15		
<value>				22												18	19	20	21		
<arg-next-params>														16	17						
<nested-stat>				37						33	36	34									35
<func-nested-st-list>				23					23	23	23	23									23
<func-nested-stat>				28					29	24	27	25									26
<def-id>		45			44																
<after-id>		41			41				40												
<next-func-nested-st-list>				31					31	31	31	31								30	31
<nex-nested-st-list>				39						39	39	39								38	39
<after-return>		47																			46
<eof-or-eol>	49	48																			

Tabulka 2: LL tabulka

7 Precedenční tabulka

	+	-	*	/	//	<	<=	>	>=	==	!=	(	)	id	int	float	string	none	$
+	>	>	<	<	<	>	>	>	>	>	>	<	>	<	<	<	<		>
-	>	>	<	<	<	>	>	>	>	>	>	<	>	<	<	<			>
*	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<			>
/	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<			>
//	>	>	>	>	>	>	>	>	>	>	>	<	>	<	<	<			>
<	<	<	<	<	<							<	>	<	<	<	<		>
<=	<	<	<	<	<							<	>	<	<	<	<		>
>	<	<	<	<	<							<	>	<	<	<	<		>
>=	<	<	<	<	<							<	>	<	<	<	<		>
==	<	<	<	<	<							<	>	<	<	<	<	<	>
!=	<	<	<	<	<							<	>	<	<	<	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	<	
)	>	>	>	>	>	>	>	>	>	>	>		>						>
id	>	>	>	>	>	>	>	>	>	>	>		>						>
int	>	>	>	>	>	>	>	>	>	>	>		>						>
float	>	>	>	>	>	>	>	>	>	>	>		>						>
string	>					>	>	>	>	>	>		>						>
none										>	>		>						>
$	<	<	<	<	<	<	<	<	<	<	<	<		<	<	<	<	<	

Tabulka 3: Precedenční tabulka k analýze výrazů